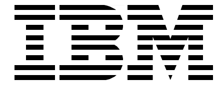
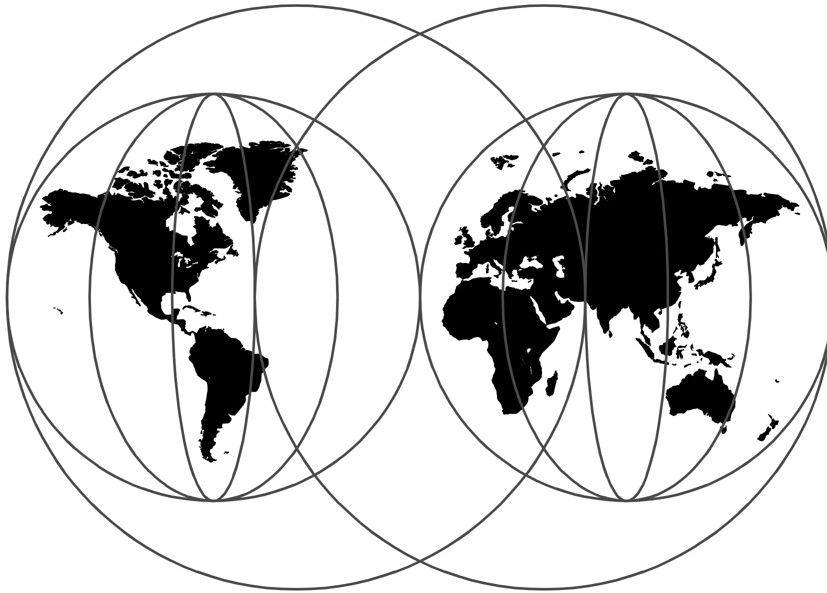


# IBM CBConnector Cookbook Collection: CBConnector Bank Implementation

*Alex Gregor, Henri Jubin, The Team*

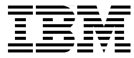


**International Technical Support Organization**

http://www.redbooks.ibm.com

SG24-5119-00

SG24-5119-00



International Technical Support Organization

# IBM CBConnector Cookbook Collection CBConnector Bank Implementation

August 1998

**Take Note!**

Before using this information and the product it supports, be sure to read the general information in Appendix A, "Special Notices" on page 111.

**First Edition (August 1998)**

This edition applies to Component Broker Connector Version 1, Release Number 2 for use with Windows NT/4 Service Pack 3.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JN9B Building 045 Internal Zip 2834
11400 Burnet Road
Austin, Texas 78758-3493

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

**© Copyright International Business Machines Corporation 1998. All rights reserved**
Note to U.S Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

# Contents

**Figures** . . . vii

**Tables** . . . ix

**Preface** . . . xi
Other Redbooks in the CBConnector Series . . . xii
The Team That Wrote This Redbook . . . xiii
Comments Welcome . . . xv

**Part 1. Introduction Motivation and Design** . . . 1

**Chapter 1. Introduction** . . . 3
1.1 Motivation . . . 3
1.2 Scope . . . 4
1.3 Contents . . . 5
1.4 Informal Description of the CBConnector Bank Application . . . 7
1.5 How to Read This Book . . . 8
1.6 Audience . . . 9

**Chapter 2. Design** . . . 11
2.1 Design . . . 11
2.2 Business Object Model . . . 12
2.2.1 Initial CBConnector Bank Business Object Model . . . 13
2.2.2 A CBConnector Policy Model . . . 16
2.2.3 Amended CBConnnector Bank Business Object Model . . . 18
2.3 Object Relationships . . . 22
2.4 Data Model . . . 23
2.5 Summary . . . 25

**Chapter 3. Rational Rose Implementation** . . . 27
3.1 Setting Up the Rational Rose Bridge . . . 27
3.2 The Rational Rose Model . . . 28
3.3 Rational Rose Model in Object Builder . . . 30
3.4 A Bridge Too Far . . . 31

**Part 2. Implementation** . . . 33

**Chapter 4. Client Implementation** . . . 35
4.1 Introduction . . . 35
4.2 The Client Back-End . . . 35
4.2.1 The Role of A Proxy . . . 37

© Copyright IBM Corp. 1998 

4.2.2 Using a Proxy . . . . . . . . . . 37
4.2.3 The CBConnector Bank Proxies . . . . . . . . . . 39
4.2.4 A Note on Proxy Optimization . . . . . . . . . . 44
4.3 The Client Front-End (Graphical User Interface) . . . . . . . . . . 44
4.3.1 The View Structure . . . . . . . . . . 47
4.3.2 Operation Panels . . . . . . . . . . 49
4.3.3 BO Action Classes . . . . . . . . . . 50
4.3.4 Common Data Container . . . . . . . . . . 50
4.3.5 DataSupplier Class . . . . . . . . . . 51
4.3.6 Hierarchy . . . . . . . . . . 52
4.3.7 Hierarchy Specification Format . . . . . . . . . . 54
4.4 Automatic Proxy Generation . . . . . . . . . . 56
4.4.1 Introduction . . . . . . . . . . 56
4.4.2 The Proxy Information Object . . . . . . . . . . 57
4.4.3 The IDL Parser . . . . . . . . . . 57
4.4.4 The Proxy Factory . . . . . . . . . . 61
4.4.5 Putting It All Together . . . . . . . . . . 64
4.4.6 Four Steps to Creating Your Own Proxy Generator . . . . . . . . . . 65

**Chapter 5. Middle Tier** . . . . . . . . . . 67
5.1 Introduction . . . . . . . . . . 67
5.2 Architecture of the Middle Tier . . . . . . . . . . 67
5.3 Persistent Implementation . . . . . . . . . . 68
5.3.1 Managed Object Container . . . . . . . . . . 68
5.3.2 Containers . . . . . . . . . . 68
5.4 CBConnector Bank Architecture . . . . . . . . . . 69
5.4.1 Functions Used in the CBConnector Bank Sample . . . . . . . . . . 69
5.4.2 CBConnector Banking Objects . . . . . . . . . . 70
5.4.3 CBConnector Bank Scenario . . . . . . . . . . 72
5.4.4 CBConnector Bank Configuration . . . . . . . . . . 74
5.4.5 Constructs . . . . . . . . . . 78
5.4.6 Address . . . . . . . . . . 79
5.4.7 Customer . . . . . . . . . . 82
5.4.8 Teller . . . . . . . . . . 86
5.4.9 Note . . . . . . . . . . 86
5.4.10 CBBankCheckAccount . . . . . . . . . . 87
5.4.11 SavingsAccount . . . . . . . . . . 89
5.4.12 TransactionRecord . . . . . . . . . . 91
5.4.13 Policy . . . . . . . . . . 92
5.5 Problems Encountered . . . . . . . . . . 95
5.6 Conclusion . . . . . . . . . . 96

**Chapter 6. Persistent Implementation** . . . . . 97
6.1 Importing the Bean to Object Builder . . . . . 97
6.2 Implementing the DO . . . . . 97
6.3 Using Mapping Helpers . . . . . 98
6.4 Fixing Generated Code . . . . . 100

**Chapter 7. Legacy Tier** . . . . . 101
7.1 Data Implementation . . . . . 101
7.2 Host CICS Transaction . . . . . 101
7.2.1 Host Flow Example . . . . . 102
7.3 Creating Procedural Adaptor Object . . . . . 104
7.3.1 Screen Scraping - Implementation . . . . . 104
7.3.2 Screen Scraping - Testing . . . . . 105

**Chapter 8. Summary** . . . . . 107

**Part 3. Appendices** . . . . . 109

**Appendix A. Special Notices** . . . . . 111

**Appendix B. Related Publications** . . . . . 113
B.1 International Technical Support Organization Publications . . . . . 113
B.2 Redbooks on CD-ROMs . . . . . 113
B.3 Other Publications . . . . . 113

**How to Get ITSO Redbooks** . . . . . 115
How IBM Employees Can Get ITSO Redbooks . . . . . 115
How Customers Can Get ITSO Redbooks . . . . . 116
IBM Redbook Order Form . . . . . 117

**Glossary** . . . . . 119

**List of Abbreviations** . . . . . 127

**Index** . . . . . 129

**ITSO Redbook Evaluation** . . . . . 131

# Figures

1. Three Tier Architecture . . . . . . . . . . 6
2. Middle-Tier Architecture. . . . . . . . . . 8
3. Organization of the Book . . . . . . . . . . 9
4. The Original Concept Relationships and Specializations . . . . . . . . . . 13
5. Party Class Description . . . . . . . . . . 14
6. Entity Class Description. . . . . . . . . . 15
7. Transaction Class Description . . . . . . . . . . 15
8. Note and Address Class Descriptions . . . . . . . . . . 16
9. Policy Manager and Policies . . . . . . . . . . 16
10. Policy Namespace . . . . . . . . . . 17
11. Policy Class Description . . . . . . . . . . 18
12. Amended Object Model . . . . . . . . . . 20
13. Savings Account, Check Account, and Transaction Class Information . . . 21
14. Customer, Note, Teller, and Address Class Description . . . . . . . . . . 22
15. The Data Model . . . . . . . . . . 24
16. Initial Rational Rose Object Model. . . . . . . . . . 29
17. Final Rational Rose Model. . . . . . . . . . 32
18. The Structure of the CBConnector Bank Proxies . . . . . . . . . . 36
19. The CBConnector Bank Sample Application User Interface . . . . . . . . . . 46
20. Typical Steps in Performing an Operation . . . . . . . . . . 47
21. Hierarchy List Panel. . . . . . . . . . 52
22. Segment of the Hierarchy Model . . . . . . . . . . 53
23. Global Architecture . . . . . . . . . . 67
24. Address Hierarchy . . . . . . . . . . 69
25. CBConnector Bank Objects. . . . . . . . . . 71
26. Architecture . . . . . . . . . . 72
27. Teller GUI. . . . . . . . . . 73
28. Data Object Inheritance . . . . . . . . . . 75
29. Persistent Objects . . . . . . . . . . 77
30. CBBankConstructs IDL and Graphics . . . . . . . . . . 79
31. CBBankAddress and CBBankAddressHome. . . . . . . . . . 80
32. CBBankAddress IDL . . . . . . . . . . 81
33. CBBankAddressHome IDL . . . . . . . . . . 81
34. CBBankAddressHome createAddress. . . . . . . . . . 82
35. CBBankCustomer IDL . . . . . . . . . . 84
36. CBBankCustomerHome IDL . . . . . . . . . . 85
37. CustomerMapper IDL. . . . . . . . . . 85
38. CBBankTeller IDL . . . . . . . . . . 86
39. CBBankNote IDL . . . . . . . . . . 87
40. CBBankNoteHome IDL . . . . . . . . . . 87

© Copyright IBM Corp. 1998 

41. CBBankCheckAccount IDL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
42. CBBankCheckAccountHome IDL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
43. CBBankSavingsAccount . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
44. CBBankSavingsAccountHome IDL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
45. CBBankTransactionRecord IDL. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
46. CBBankTransactionRecordHome IDL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
47. Policy Naming Hierarchy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
48. CBBankPolicy Exceptions IDL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
49. CBBConnector Bank Policy IDL. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
50. CBBankPOlicy Manager IDL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
51. Data Object Inheritance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
52. GenderType Conversion in PAO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
53. Mapping Objects with PAO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
54. Flow of the CICS Host Application. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103

# Tables

1. Namespace Retrievals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2. Object Relationships Used in the Model . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3. Role of Note Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
4. Role of Address Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5. Role of Customer Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
6. Role of Customer Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
7. Role of Note Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
8. Role of Note Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
9. Role of Note Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
10. Role of Note Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
11. Role of Policy Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

© Copyright IBM Corp. 1998

# Preface

The CBConnector Bank is a sample bank application developed for Component Broker Connector Release 1.2. This redbook explains how the CBConnector Bank application was developed, from object model to installed Business Objects, and is part of the International Technical Support Organization's (ITSO) *IBM CBConnector Cookbook Collection*.

Component Broker Connector (CBConnector) is a new member of the IBM Transaction Series product family that supports distributed object computing in a multitier environment. CBConnector provides a middle-tier application 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, CBConnector 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.

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

Component Broker consists of two parts that support these three dimensions. 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.

CBConnector'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.

We developed the CBConnector Bank as an application that demonstrates the capabilities of Component Broker Release 1.2. This application will grow to encompass the new functionality of subsequent releases of Component Broker.

© Copyright IBM Corp. 1998

The application has three tiers: the client tier, the middle tier–the Component Broker server–and the legacy tier, where a CICS application resides. We call this the "legacy system" in quotes because it is a newly developed system, one whose sole purpose is to act as a persistent store to the bank application. However, this tier would normally be an existing application to be integrated into the new, object-oriented architecture that Component Broker represents.

This redbook provides a detailed description of how we implemented the three tiers and discusses the CBConnector-specific issues and concerns we encountered during this development.

System Engineers who already have a basic understanding of Component Broker concepts can use this book as a stepping stone to go from our samples to develop more complex, real-life applications.

## Other Redbooks in the CBConnector Series

So far, there are four books in the CBConnector redbook series. Below is a short description of what each book covers.

- *IBM Component Broker Overview,* SG24-2022. This redbook provides a general understanding of what CBConnector is and provides an overview of the underlying architecture and concepts.
- *IBM CBConnector Cookbook Collection* is a series of three redbooks whose the objective is to give the reader hands-on experience by installing and running sample applications and providing step-by-step (or cookbook) instructions on how to develop CBConnector applications. The different parts that make up the *Cookbook Collection* are:
  - *First Steps,* SG24-2033. This redbook is the introductory book. It leads you through elementary examples that show the basic functionality of CBConnector.
  - *CBConnector Bank Implementation,* SG24-5119. This is the redbook you hold in your hand right now. It takes developers one step further and explains the development process of a sample bank application, from object model to installed Business Objects. It discusses CBConnector-specific issues and concerns encountered during development.
  - *CBConnector Bank User Guide,* SG24-5121. This redbook provides instructions on how to install and run the bank application discussed in the *CBConnector Bank Implementation* redbook.

In addition to the CBConnector redbooks, we recommend to read the books in the *CBConnector Library*, which is a collection of complete installation guides, programming guides and references.

## The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working for the International Technical Support Organization, Austin Center. The project was designed and managed by Alex Gregor and Henri Jubin. The writers were:

**Dr. Andy Bond** is a Principal Research Scientist at the Co-operative Research Centre for Distributed Systems Technology in Brisbane, Australia. He has over 10 years experience in the field of distributed systems, including load sharing, distributed system architecture design, DCE, CORBA, and experimental middleware technologies. He has written many publications in these areas. He holds a Ph.D. degree from Victoria University of Wellington, New Zealand, where his topic was "Adaptive Load Sharing in a Distributed Workstation Environment".

**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 Connector in the IBM Transaction Series.

**Mark Fitzpatrick** is a Principal Consultant at the Distributed Systems Technology Centre in Brisbane, Australia. He has worked in the IT business for 20 years. His areas of expertise include object-oriented technologies, distributed architectures, such as CORBA, and application development in languages such as Smalltalk and Java.

**Alex Gregor** is an IBM Senior Software Engineer working at the International Technical Support Organization (ITSO), Austin Center in the OO/AD group. His responsibilities include technical support for IBM application frameworks.

**Henri Jubin** currently works for the 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 with IBM France. He has dealt with topics such as object-oriented technology, OS/2, Windows NT, and OpenDoc.

**Zoran Lerch** is a self-employed data-processing consultant and an IBM BestTeam partner, focusing on Java and JavaBeans technologies and on the broad spectrum of their application.

**Dr. Andry Rakotonirainy** is a Senior Research Scientist at the Co-operative Centre for Distributed System Technology Center (DSTC) in Brisbane, Australia. He holds a Ph.D from INRIA (Institut National de Recherche Informatique et Automatique) France for his doctoral work titled "Advanced Transaction Models". He has significant experience in, and has written publications on, distributed systems.

**Hanne Rygg Johnsen** is a Systems Engineer in the Nordic Object Technology Practice (OTP), IBM Norway. She has four years experience in developing object-oriented systems with Smalltalk and Java, 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 providing customer consulting services in object-oriented analysis and design.

**Pasi Salminen** is a project manager at Profit Ltd. in Finland. He has six years experience in Object Technology and three years in distributed computing. Profit Ltd. is building large-scale insurance systems using Object Technology. Its main product, Once&Done, is a system for managing the complete 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 to industry standards and running on all major platforms.

**Terje Storstein** has worked as a Systems Engineer in IBM Norway for almost 30 years. He started in the mainframe area, worked with the distributed systems of the 1970s, and went on to work in the client/server era of the 1980s and 1990s. Currently, he is working in the e-business department of the Norwegian Global Services.

**Hennie van Wijk** is a Technology Consultant at Amalgamated Banks of South Africa (ABSA Bank). He is part of a team responsible for defining Application Architecture for ABSA, specifically dealing with Object Technology Strategies and OO Analysis and Design. He has extensive mainframe experience including application development with COBOL, CICS, IMS, and DB2 database analysis and design, which have kept him occupied for the past 12 years. He is currently involved in research and development into retrofitting "legacy systems" for reuse in a distributed computing environment.

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

**IBM Development Laboratory in Austin, Texas:**

Mei-Mei Fu
Greg Truty

**IBM Development Laboratory in Rochester, Minnesota:**

Eric N. Herness
Kevin Sutter

**IBM Development Laboratory in Santa Teresa, California:**

Harry Nayak
David Wisneski

**IBM Development Laboratory in Toronto, Canada:**

Chris Brealy
Tim Francis
Suman Kalia
Christina P. Lau
Yen Lu

**IBM International Technical Support Organization:**

Eugene Deborin
Joe DeCarlo
Steve Gardner
Bob Haimowitz
Hanspeter Nagel

**IBM EMEA Application Development Software Centre, La Gaude:**

Jean Pierre Augias
Jean Michel Fauconnier
Joaquin Picon
Bruno Georges
Phiippe Gregoire

## 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 “ITSO Redbook Evaluation” on page 131 to the fax number shown on the form.
- Use the electronic evaluation form found on the Redbooks Web sites:

  For Internet users `http://www.redbooks.ibm.com`
  For IBM Intranet users `http://w3.itso.ibm.com`

- Send us a note at the following address:

  `redbook@us.ibm.com`

# Part 1. Introduction Motivation and Design

Part 1 introduces you to the dialectic of our book and discuss issues related to the idea of creating a sample application in the banking domain. We called the sample application the CBConnector Bank.

One of the primary emphases of IBM's Component Broker Connector (CBConnector) is to support the evolution of the business enterprise. Its support for distribution and object modeling makes it the ideal tool to address evolving business market needs. The integration of legacy resource managers allows the leveraging of the huge resources invested in existing applications, including transactions and data.

This book shows how we designed and built a banking application sample with IBM Component Broker Connector. The sample demonstrates the ability of CBConnector to integrate legacy middleware (Universal Database (UDB), CICS) within a multitier CORBA architecture.

© Copyright IBM Corp. 1998

# Chapter 1. Introduction

The book follows a development cycle, starting with the object model and continues with the implementation of the three different tiers. We provide a user guide in the *CBConnector Bank User Guide,* SG24-5121, to install the application.

In the following sections, we describe the motivation, scope, and provide a short description of the CBConnector Bank.

## 1.1 Motivation

In today's financial services industry, customers demand fast and convenient access to their banking solutions. They want more variety in products, and they want them to be delivered at low cost across a wide array of delivery channels. For example, banks must evolve and adapt to emerging customer needs, at low cost and without ignoring existing legacy applications.

The reuse of legacy systems is an issue that software architects cannot avoid. Software builders cannot afford to periodically rebuild their software from scratch. Many software designs are now produced by combining and elaborating on existing architectural design fragments.

Work on architectures for software systems and studies for better ways to support software development have been around for quite a while. There is agreement now that Object Technology is the best basis for software architectures. The properties of object modeling, such as encapsulation and inheritance, facilitate the prototyping of new software and the integration of legacy applications into an object environment. The use of Object Technology considerably speeds up code maintenance and development time, consequently reducing the development cost. Therefore, Object Technology has become the de-facto standard for designing and building software.

One of the primary purpose of IBM's Component Broker Connector (CBConnector) is to support the evolution of the business enterprise. Its support for distribution and object modeling makes it the ideal tool to address evolving business market needs. The integration of legacy resource managers allows the leveraging of the huge resources invested in existing applications including transactions and data.

This book shows how we designed and built a banking application sample with IBM Component Broker Connector. The sample demonstrates the ability

© Copyright IBM Corp. 1998

of CBConnector to integrate legacy middleware (Universal database (UDB), CICS) within a multi-tier CORBA architecture.

## 1.2 Scope

This book presents a banking application sample built with IBM Component Broker. It explains the different development phases from the design to implementation. The banking application described in this book gives you the opportunity to study an application that fully exploits CBConnector solutions for enterprise distributed objects. It is a guide to integrating client/server, object-oriented, relational database, and transactions in the business enterprise.

The banking sample demonstrates distributed object computing in a multitier environment. It consists of an application that invokes transactions from the client front-end (GUI) to a set of UDB and CICS back-ends through the middle tier. It shows the integration of Object Technology, such as CORBA and Rational Rose modeling with existing enterprise transactions in CICS, relational databases such as DB2, and procedural languages such as COBOL.

The book starts with an informal description of the banking model followed by the Rational Rose description of the same model. The study of the architecture leads us to concentrate on the composition of different components of the architecture. We discovered limitations in the toolkit we used, in addition to the constraints brought by the integration of Object Technology with procedural databases. Therefore, we had to refine the model into a simpler model.

The business logic of the refined model was implemented using the Component Broker Toolkit and Object Builder. Object Builder is a tool that application developers use to define and implement the Business Objects running on the middle tier. The business logic we defined consists of the implementation of methods, the relationship between objects, and the mapping from CORBA-IDL to the DB2 data model.

The above mapping allows the client front-end to transparently access persistent data stored with DB2. A typical CICS customer application was defined at the back-end. In order to demonstrate that our middle tier can interoperate, we created two databases, one on Windows NT and another one on MVS. We used the CICS and IMS Connecting (CICON) tool to access the second DB2 database through CICS transactions.

Transactions that access the NT DB2 database are coordinated by the CBConnector Transaction Service, which is conformant with the CORBA - Object Transaction Service (OTS). The CBConnector transaction monitor:

1. Coordinates a legacy transaction monitor such as CICS, which in turn accesses DB2
2. Coordinates the resource manager with the X/Open - XA interface

At the other spectrum of the architecture lies the client front-end. The client application developer does not need to know how the server is being implemented (database, language, services). The client simply deals with a CORBA-IDL interface and queries the middle tier. A GUI was developed to ease the task of the client to query the middle tier. A Business Object Proxy was also developed to ease and speed up the development of the client front-end that accesses the middle tier business object.

This book elaborates on all the steps we summarized in this section toward building the banking application. The banking sample is provided with this book.

## 1.3 Contents

The structure of the book reflects the way different tasks were split and assigned to the team members. The core of the book is organized in three parts. They are design, implementation and samples. The design section answers the question "what model" and "why this model". The implementation answers the question "how to implement the model". The sample presents in detail the written code and installation.

1. The chapter on the design describes the architecture. It is an important phase of the software development:

   > *If a project has not achieved a system architecture, including its rationale, the project should not proceed to full-scale system development. Specifying the architecture as a deliverable enables its use throughout the development and maintenance process.*
   >
   > - Barry Boehm, 1995

   The first part of the chapter describes, in a simple and generic way, the requirement and the architecture from which the CBConnector Bank application was built. It is an abstract system specification consisting primarily of objects described in terms of their behaviors, interfaces, attributes and object interconnections, and relationships. It is a description beyond the algorithms and data structures of the computation; it shows how we designed and specified the overall system structure.

This chapter also describes the refinements we've made from our initial model aim to the model that we have actually implemented. A selection among design alternatives was necessary due to the limitations of the current version of the software we were using and intrinsic constraints brought about by the use of legacy systems such as CICS and UDB.

The second part of the design more formally presents the model using Rational Rose and Object Builder. It does not assume much expertise or skills in Rose methodology, and it starts with a reminder of the general principles of Rational Rose.

2. The implementation is defined with Object Builder which comes with the CBC Toolkit. This chapter is structured according to the well-known three-tier model. The three-tier model is an extension of the client-server model (two tiers). The main motivation for the three-tier model is that the client is lightweight and the server is split into a second tier and a third tier. The second tier provides the business logic; this tier is also called the middle layer. The third tier contains the data or any Resource Managers (see Figure 1 on page 6). Note that an architecture is well-designed when each tier can be integrated and interoperate easily with other components.

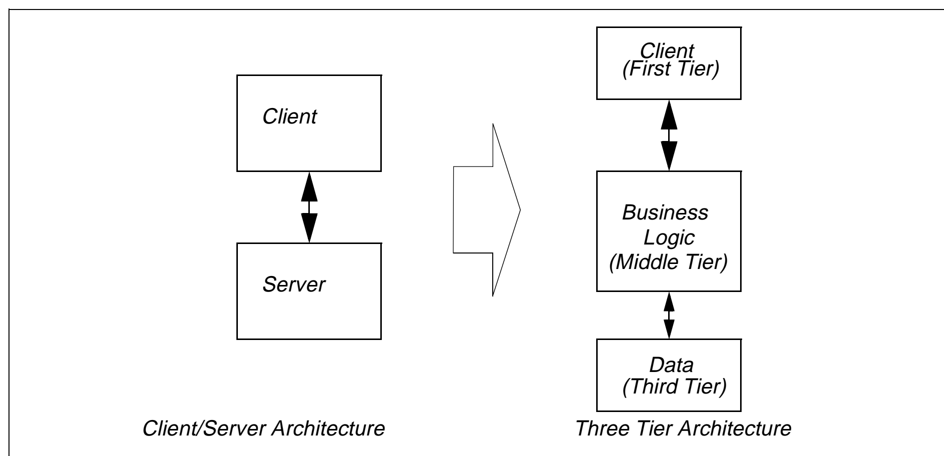



*Figure 1. Three Tier Architecture*

The details of the content of our three-tier architecture are as follows:

- The client part shows how a client contacts the middle tier. It deals with GUI Java code and the proxy that allows the client to connect to the server through the CBConnector ORB broker.
- The middle tier part of CBConnector contains the description of the business logic. It is essentially the middleware that provides clients

with the ability to interoperate with other middleware and resource managers (UDB, CICS).

- The legacy tier describes how to reuse a legacy CICS transaction application as a resource component in Component Broker. It refers to the Procedural Application Adaptor (PAA), CICON, and CICS.

3. The last part of this book presents the samples we used. It describes the limitations encountered and the solutions we chose.

## 1.4 Informal Description of the CBConnector Bank Application

This section informally presents the CBConnector Bank application. In the rest of the document, names written in **bold** are objects we created in the model.

A **Customer** of a Bank creates either a **Check_Account** or **Savings_Account** by providing various personal customer information such as **name** and **Address**. Once the account is created, the **Customer** can query his/her account, transfer an amount of money from account to account and read **Notes** that the **Teller** created. A **Teller** queries accounts and **Customers**. They create **Notes** for **Customers** and freeze, open, or close the **Check_Account** or **Savings_Account**.

A **Teller** can dynamically set/get **Policy** name-value pairs such as default interest rate. The Policy Manager provides the ability to change dynamically the value of static variables used in CBConnector program. Those pairs can be used by all objects in the banking application. Each method can be invoked within a transaction, a log called a Transaction Record is appended to each account upon update so that the **Teller** can query the history of operations applied to accounts.

Figure 2 on page 8 shows how the objects used in the middle tier are related and how they are stored.

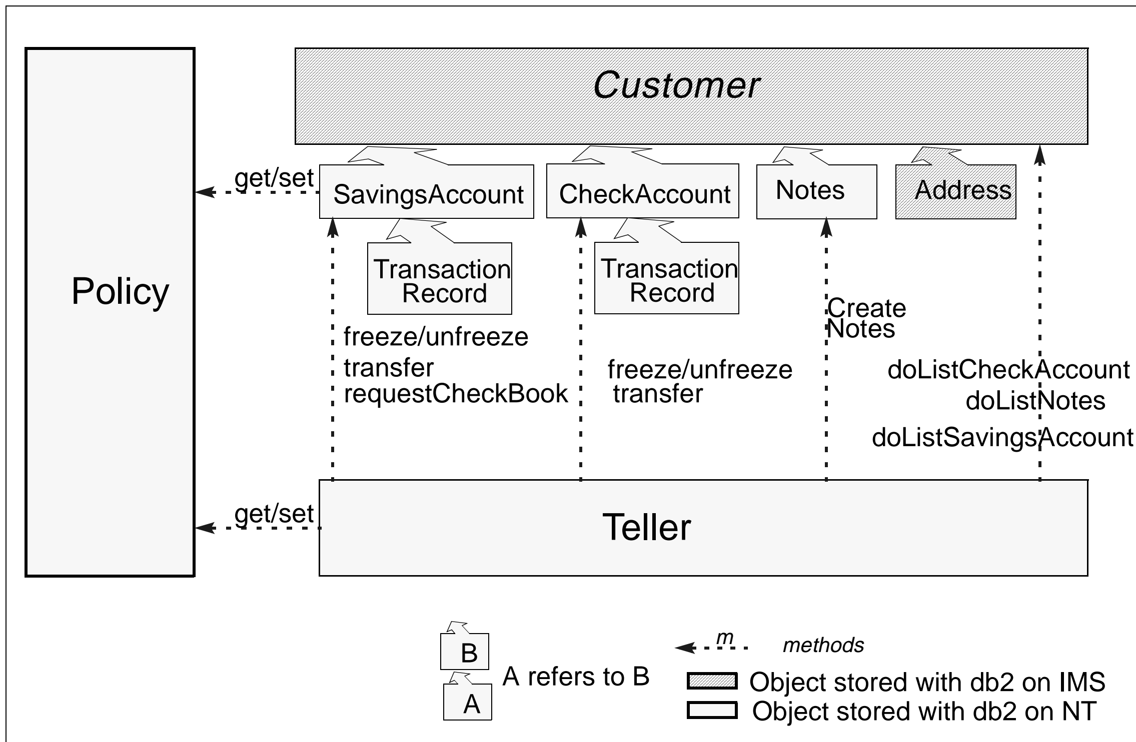



*Figure 2. Middle-Tier Architecture*

## 1.5 How to Read This Book

The book is structured into two parts: Implementation and Design as shown in Figure 3 on page 9. You are invited to read the introduction in order to be aware of the big picture. The first three parts of the design (Business Model, Relations, Data Model) must be read together. The design chapter also describes the refinements we have made between the initial theoretical model and the implemented model; this section has to be read with the first three chapters. The Rational Rose part can be read separately. The three chapters of the implementation (Client, Middle Tier, Legacy Tier) can be read independently.

In order to study teh CBConnector Bank models and code, you have to have on hand a copy of the *CBConnector Cookbook Collection: CBConnector Bank User Guide*, SG24-5121.

The recommended way to start working with the CD-ROM is as follows:

- Create a CBBank directory on your hard drive.
- Map the hard drive to **O**.
- Copy the *\CBBANK\Middle Tier* subdirectory from the CD-ROM *User Guide*
- Execute the `ATTRIB -R` command on your new directory in order to remove the read-only flag from the copied files.
- Start the CBConnector Object Builder
- Load the Object Model from the level of *OBmodel* directories.
- Specify the *InstallShield* directory in the CBConnector Object Builder.

If you would like to stude the CBConnector Java Client code, you can import the interchange file to your VisualAge for Java workbench. Please follow these steps:

- Create a *CBBank* directory on your hard drive.
- Map the hard drive to **O**.
- Copy from the *\CBBank\Client\FrontEnd\interchange* subdirectory the interchange file.
- Execute the `ATTRIB -R` command on your new directory in order to remove the read-only flag from the copied files.
- Import the interchange file to your VisualAge for Java workbench.

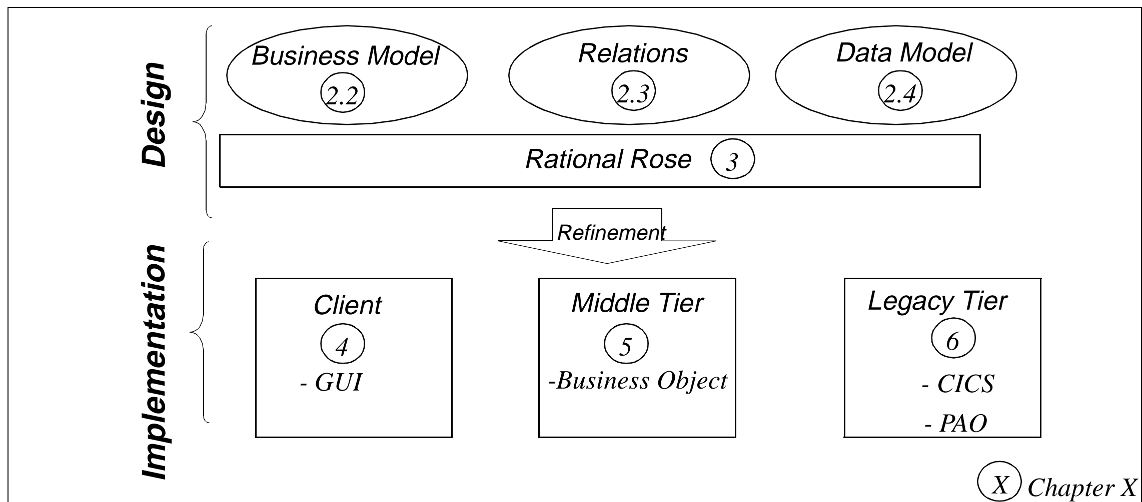



*Figure 3. Organization of the Book*

## 1.6 Audience

The intended audience of the book consists of bankers, computer scientists, and professionals with an interest in IBM Component Broker. Familiarity with object modeling is helpful background for the rest of the book. Knowledge about Component Broker is a definite advantage.

The book complements the IBM *CBConnector Cookbook Collection Overview* and IBM *CBConnector Cookbook Collection: First Steps* redbooks, and the product documentation. We deliberately chose to reuse paragraphs from those redbooks in order to bring you a self-contained document.

# Chapter 2. Design

The banking example we have used is loosely based on some sample requirements from the banking domain. It is not meant to cover all banking requirements, but rather to demonstrate the benefits of a distributed, object-based solution to the traditionally centralized banking environment.

We begin by describing some design philosophies which have guided us in our initial design and subsequent mapping to object and data models.

The Business Object model describes both our initial object model design as well as the more restricted model we eventually used. These restrictions were governed by the versions of the Object Builder and CBConnector technologies with which we worked.

The data model brings together the object model and the requirements for database integration at the back-end level. It takes the class descriptions provided in the Business Object class descriptions and adds key information needed in the database table definitions.

Chapter 3, “Rational Rose Implementation” on page 27 describes the specification of the Business Object model using Rational Rose. It is possible to import the Rational Rose specification into Object Builder, generating the model to use within Object Builder. Unfortunately, many of the constructs generated by Rational Rose were not usable in the current Object Builder release. Thus, the Rational Rose model was more a theoretical exercise. Further description of the Rational Rose work is described in Chapter 3, “Rational Rose Implementation” on page 27.

## 2.1 Design

Banking has long been the home of large financial packages which have been time-consuming and expensive to extend or replace. An object environment exhibits many characteristics which alleviate these problems. In particular, a distributed object environment, such as CORBA, provides a rich environment embracing software evolution and interaction.

- Objects use *inheritance* to extend and specialize other object definitions. Common attributes and methods can be shared between components with no replication of specification and implementation.
- The separation of specification and implementation is a powerful concept supporting *portability* and *interoperability*. Portability is the ability to migrate software to another platform within a predictable time and effort.

© Copyright IBM Corp. 1998

Interoperability is the ability to glue disparate components together using a common object interaction infrastructure.

- Separate versions of component implementation can *coexist* in a distributed object world. New versions can seamlessly replace older versions with client objects binding to their server objects at runtime.
- Objects can represent the function of *legacy systems* through object wrapping. The object provides a standard interface specification and transforms any method or data queries into calls to the legacy environment. The separation of specification and implementation is key to this ability.

## 2.2 Business Object Model

The Business Objects implement the business logic in the middle tier of the three-tier architecture. They provide data and function to the user interface, and in turn, access the data layer, providing information from the back-end database systems.

We first describe our initial Business Object model and follow with the amended specification, which was restricted by the tools and environment with which we worked. It is likely that this set of restrictions will change with future product releases.

### 2.2.1 Initial CBConnector Bank Business Object Model

The initial Business Object model for the banking environment is depicted in Figure 4. It shows the Business Objects along with their inherited virtual parents.

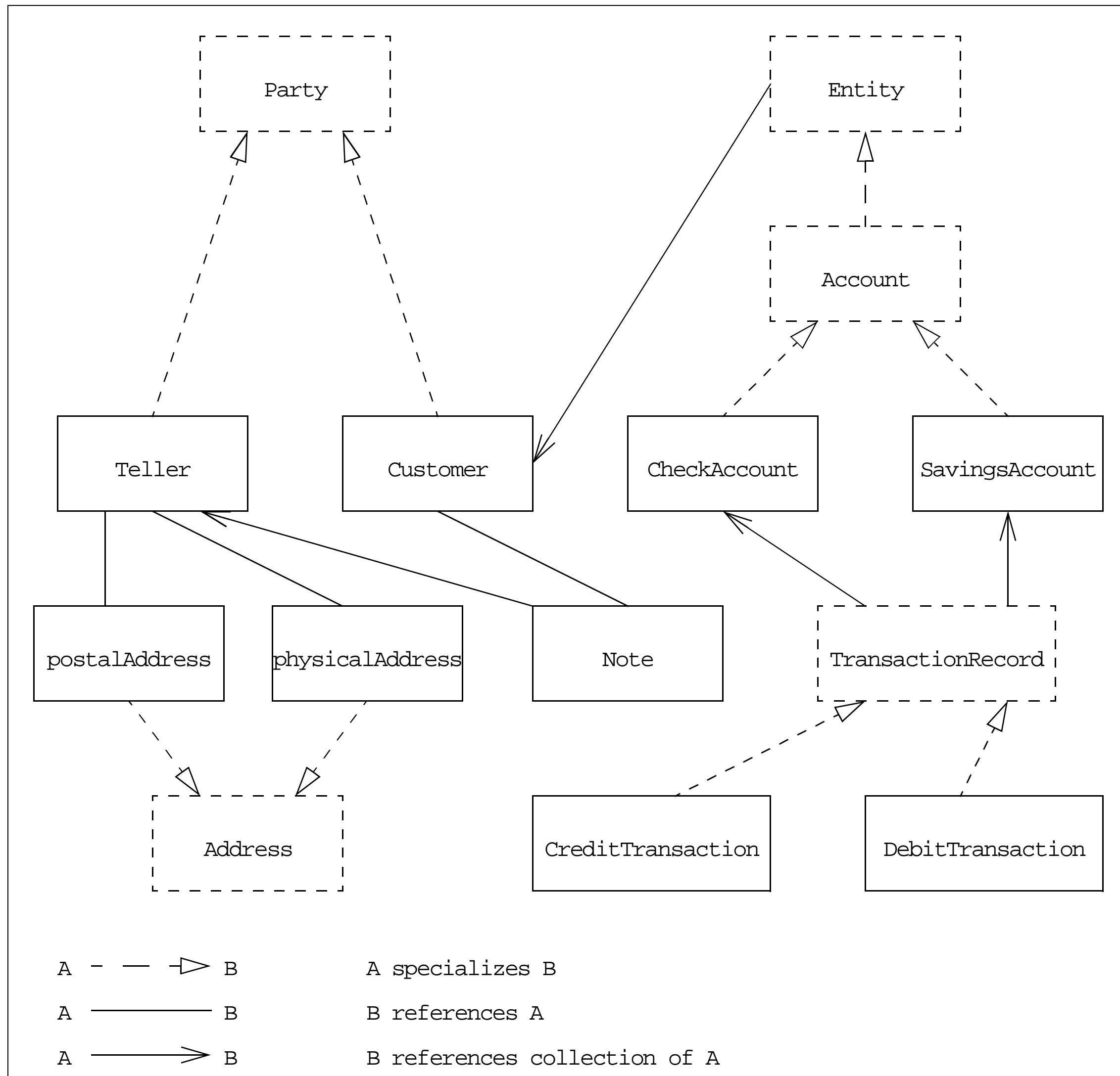



*Figure 4. The Original Concept Relationships and Specializations*

This is our initial Business Object model. It is designed to demonstrate object-oriented concepts in a simple example banking environment.

We now describe the data and operations associated with the Business Objects. It should be noted that this is our initial, theoretical design. After analysis and design it was necessary to modify the design to reflect the restrictions of the tools and environment in which the example was implemented. These restrictions will be noted following this initial description.

- A **Party** is a container for the role of people within the bank. Currently, it is specialized into the **Teller** and **Customer** roles. Further specialization would provide other roles such as manager, accountant, and so forth.

  Figure 5 describes the class data and methods for each of the Party, Customer, and Teller objects. The party contains the name of the person.

- The **Teller** is responsible for generating notes associated with a customer. These are stored within the Customer Object and reference the originating teller. A teller stores **position**, username (**id**), and **password** information in addition to the name from the `party` abstract class.

- The **Customer** includes the usual customer information. The postal and physical addresses are references to **Address** objects. The **id** is some external information, such as a driver's license or passport identifying the customer external from any internal bank identification. This is useful for identifying a customer when they have no bank reference available (for example, they have forgotten or lost their account number). Each customer has a sequence of accounts and notes associated with the customer.

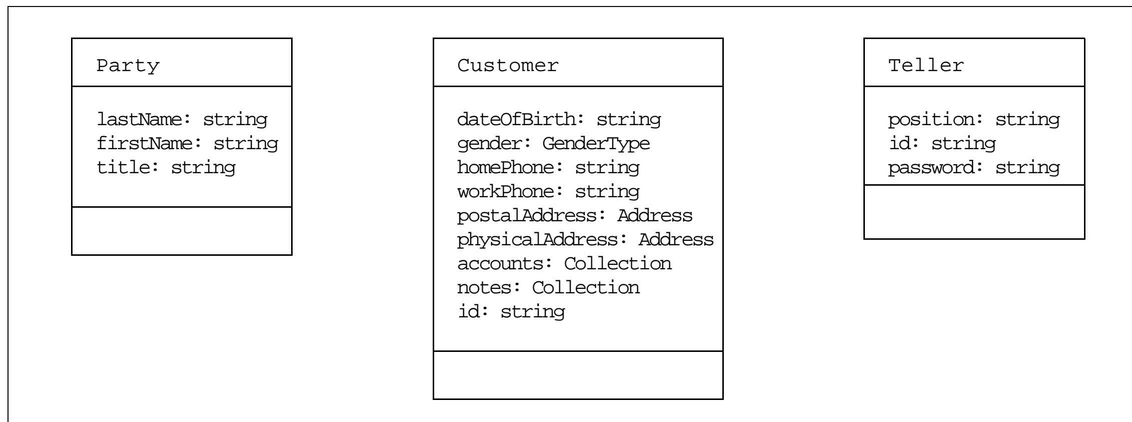



*Figure 5. Party Class Description*

- An **entity** is a generic container for a banking product. At this stage, we have only introduced **Account,** but would foresee insurance policies and wills falling into the same container. It stores a **name** and **creationDate** (as shown in Figure 6).

- The **Account** holds data and function generic to the set of account types. This includes a **balance**, the **status** of the account (open, frozen, closed), appropriate interest rates, and fees. Each account maintains a collection of transactions dealing with the account. The operations to suspend and activate as well as credit and debit are available.

  We assume accounts are never deleted but instead moved to a closed status. This will be performed in the **delete** method.

- A **SavingsAccount** has a minimum balance that must be maintained and the **CheckAccount** a maximum overdraft. In addition, checkbooks can be requested for the `CheckAccount`.

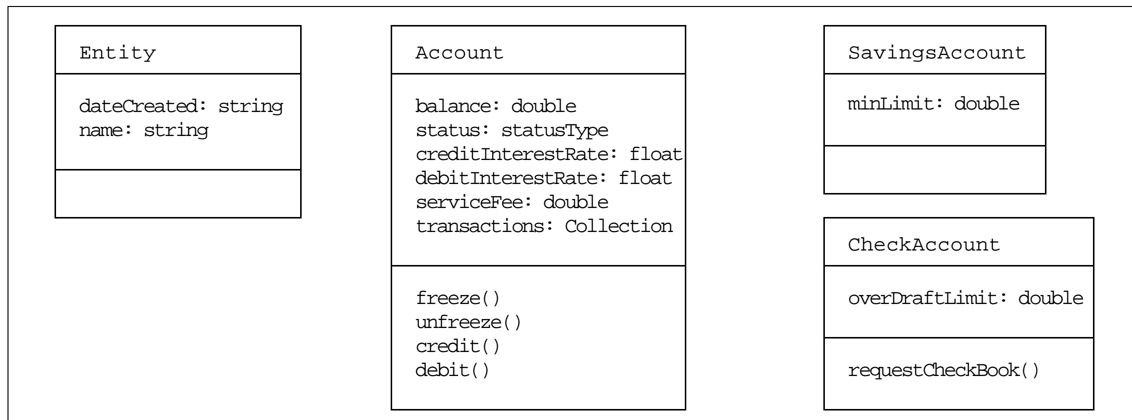



*Figure 6. Entity Class Description.*

- A **TransactionRecord** tracks the transactions performed on an account. They are either a **CreditTransaction** or a **DebitTransaction**, as shown in Figure 7.

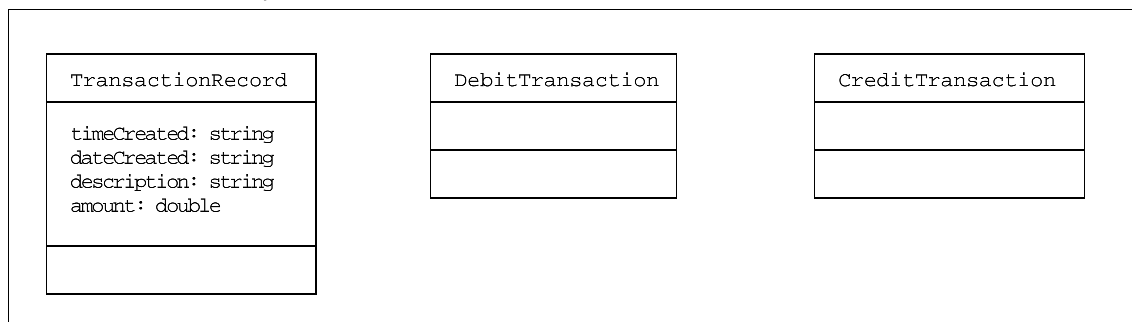



*Figure 7. Transaction Class Description*

- The two leaf nodes of **Address** and **Note** are described in Figure 8. Addresses are subtyped into **PhysicalAddress** and **PostalAddress**. The **Note** has a single reference to the teller who created the note.

*Figure 8. Note and Address Class Descriptions*

The CBConnector banking objects form an interconnected object model fulfilling the promises of a distributed object environment by providing legacy integration, portability, interoperability, and evolution support through interface specification and implementation.

### 2.2.2 A CBConnector Policy Model

A second subsystem was designed that acts as an independent policy provider to the banking subsystem. The Policy Manager (see Figure 9) manages a set of policies that map a name into a value.

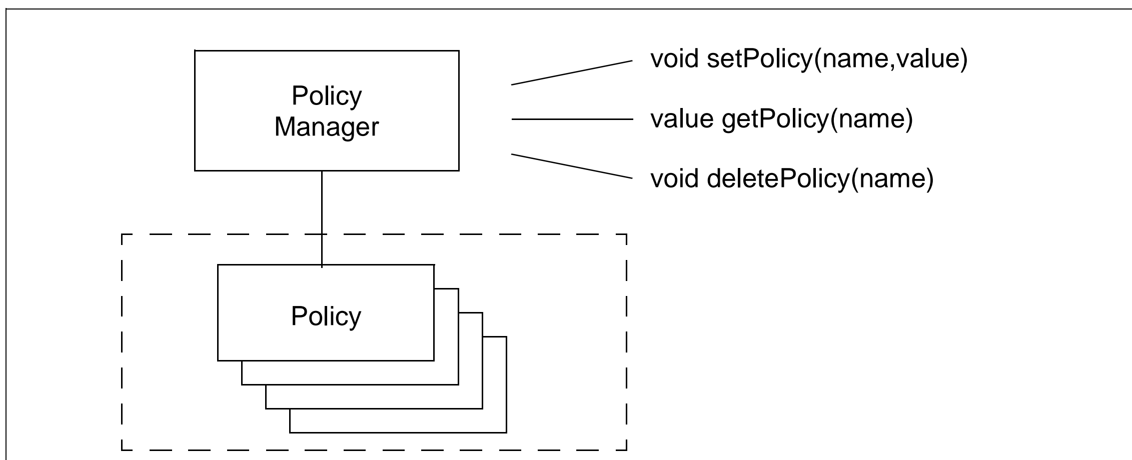



*Figure 9. Policy Manager and Policies*

The policy namespace is managed as a tree of name contexts and components (see Figure 10). Each name is a sequence of contexts and a

component name, for example /CBBank/Account/Check/interestRate. The context tree has a set of components at each node.

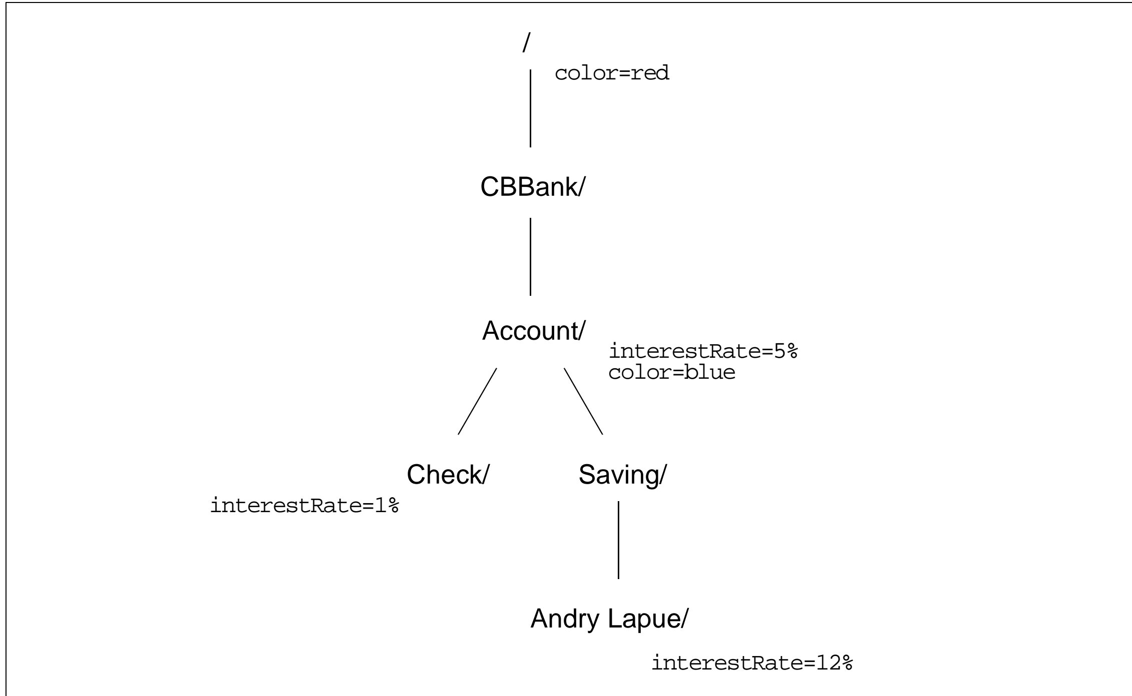



*Figure 10. Policy Namespace*

Policy retrievals initially split the policy name into the context and component parts. The tree is searched for the most complete context set that contains the component name. This is best demonstrated by a set of examples based on the policy hierarchy, as shown in Figure 10.

*Table 1. Namespace Retrievals*

| Name | Result |
|---|---|
| /CBBank/Account/Saving/Andry Lapue/color | blue |
| /CBBank/color | red |
| /CBBank/interestRate | *Exception* |
| /CBBank/Account/Check/interestRate | 1% |
| /CBBank/Account/Saving/InterestRate | 5% |
| /CBBank/Account/Check/Clint Eastwood | 1% |

The hierarchical namespace is defined by the policy creator and subsequently used by the policy users.

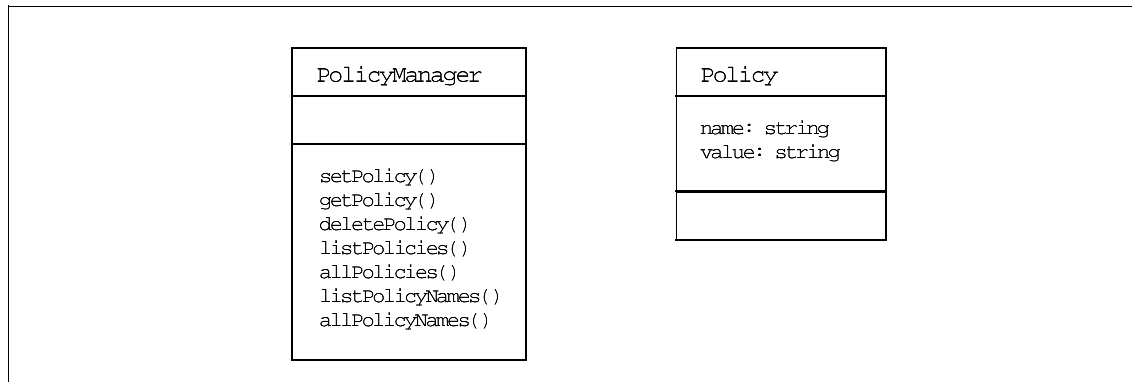



*Figure 11. Policy Class Description*

The Policy Manager is responsible for policies and is the only object which deals with Policy Objects. The Policy Manager can be instructed to perform several operations.

- Policies can be *set* by binding a policy name and policy value pair.
- Given a policy name, a policy value can be *retrieved*.
- Named policies can be *deleted*.
- *List policies* will return a list of policies given an SQL SELECT statement. Alternatively, all policies can be obtained.
- Similarly, the list operation can deal only with policy names.

In the ideal world, policy names would be context and component structures, and the values would be the CORBA Any type. Unfortunately, the current limitations of Object Builder have limited us to strings. These limitations will likely change in future. In addition, the sequences of policies and names returned from the list operations would be true CORBA sequences, but current restrictions have found us using strings with components delimited by a given deliminator character. See 5.4.13, “Policy” on page 92, for more details.

## 2.2.3 Amended CBConnnector Bank Business Object Model

Unfortunately, the idealized model had to be simplified due to the versions of the tools and environment in which we developed the solution. These restrictions will change with future releases. The restrictions include:

- Object Builder 1.2 does not support component *inheritance*. The inheritance hierarchies in the model were flattened, thus reducing the total number of Business Object interfaces. Entity, account, debit transaction, credit transaction, physical address, postal address, and party were pushed into their subtypes. Common methods and attributes were copied into the remaining interfaces.

  This appears rather strange in a Business Object model, but was only implemented this way due to technology restrictions. Future versions of Object Builder should fix these restrictions.

- The data objects supporting our Business Objects are backed by CICS and UDB back-ends. Specifically, the Customer and Address Objects are managed through a CICS transaction (a single CICS transaction, in fact) and are thus limited in their reference to other objects. They must use object keys that can be produced from a CICS transaction to reference a collection of other objects.

  Objects persistent through UDB must provide a Primary Key for persistency to the database. Many of our objects contain no combination of attributes guaranteeing a unique key; so additional keys have been added as attributes.

- Object Builder provides an object relationship facility that describes one-to-many relationships. Object Builder then provides add, remove, and list operation implementations for these object relationship sets. Unfortunately, in combination with a persistant implementation, components cannot currently be listed after they are added to the collection. Consequently, we have implemented these relationships ourselves, generating and accessing collections of object references through iterators.

Figure 12 describes the Business Object model adopted for the bank. Seven basic object building blocks are provided. Note that the relationships between objects are, at present, not explicitly present in the object model.

The mapping between the relational database object worlds is a significant issue. Our approach has seen the relational database environment impose its

implementation on the once pure object model. This is not an acceptable solution, but it was our only option at the time. A better answer would shield the legacy database environment through a relational/object mapping layer.

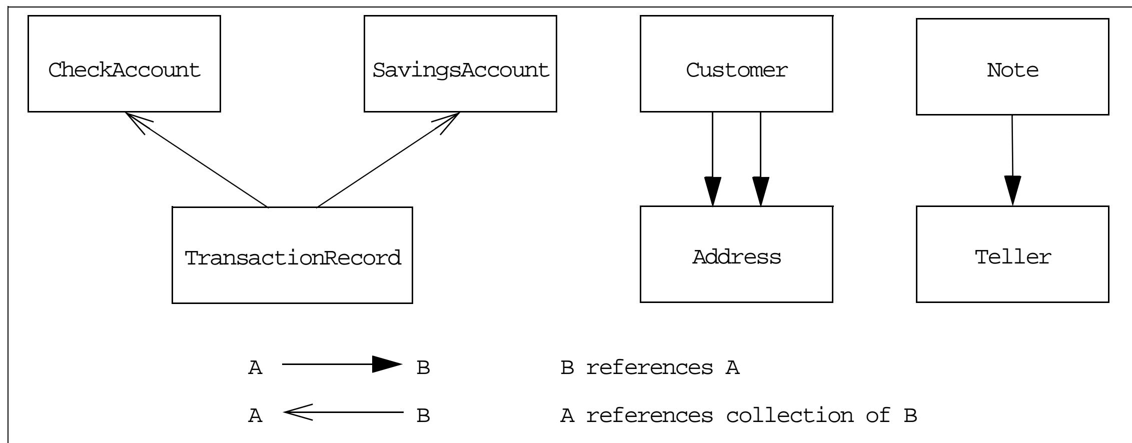



*Figure 12. Amended Object Model*

It would probably be more accurate to term this a data model rather than an object model, but let us persist nonetheless.

The Business Objects are similar to those in the original model, but they now explicitly contain the methods and data of their previous parent interfaces as well as include keys needed by the database layer.

Figure 13 describes the **SavingsAccount**, **CheckAccount**, and **TransactionRecord** class attributes and methods.

- The **SavingsAccount** now contains a unique key to distinguish the account within the persistent UDB database and a reference to the customer through the customer key. To find a customer's account, we must search for relevant account objects. However, since the customer is backed by CICS, it can not use any list of object references for this collection and must instead use the customer key in the account collection to find the relevant accounts. A savings account also has a collection of transaction references. This collections is maintained privately and accessed through the **listCollections()** method.
- The **CheckAccount** is of similar composition.
- The **TransactionRecord** uses the unique combination of creation date/time with customer and account keys to create the Transaction

Record key. In addition, the type of transaction is distinguished by a transaction kind type.

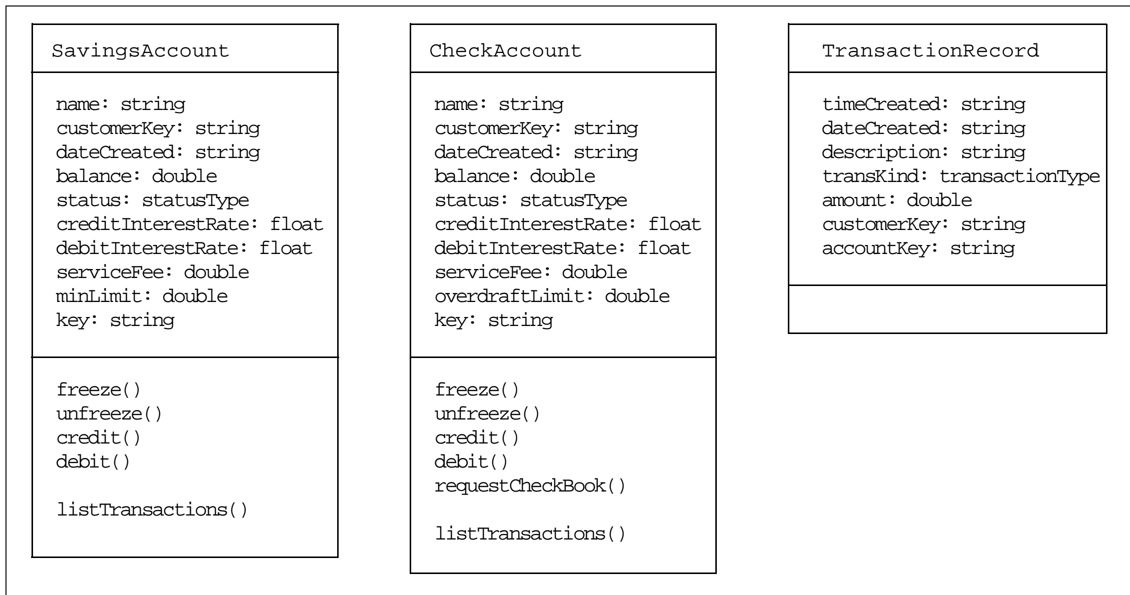



*Figure 13. Savings Account, Check Account, and Transaction Class Information*

The remaining Business Objects are described in Figure 14.

- The **Customer**, backed by CICS, contains a unique key as well as methods to obtain accounts and associated notes. These will be found through a lookup on the customer key.

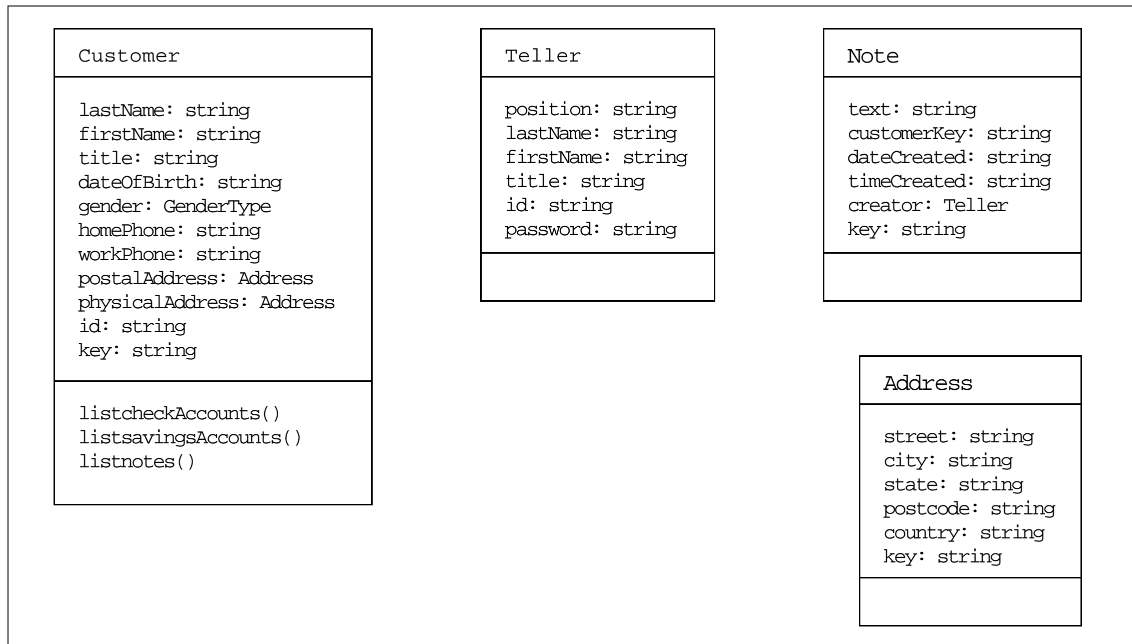



*Figure 14. Customer, Note, Teller, and Address Class Description*

- The **Teller** has a unique key provided through the id, which is used by the **Teller** (along with the password) to log in to the user interface.
- A **Note** contains the customer key for similar reasons to the Account Objects (such as a reference from the CICS-backed Customer Objects). Once again, they include a unique note key.
- **Address** is similar.

## 2.3 Object Relationships

Object relationships should ideally be provided by either single object references or reference collections. As described earlier the limitations of CICS and UDB backed objects means we have implemented these object relationships in several interesting ways.

- A simple one to one object relationship is defined using a simple *object reference* attribute.

- Object Builder provides an *object relationship* ability when defining interfaces. These relationships manifest themselves as reference collections in the implementation, with methods to add, remove, and list references. This would be the obvious way of providing a one-to-many object relationship.
- It is also possible to reference objects *indirectly* through their unique key. For example, since we are restricted by an object backed through CICS, the only possible reference mechanism is through an object key.

The relationships used in our model are described in Table 2.

*Table 2. Object Relationships Used in the Model*

| From | To | Cardinality | Type |
|---|---|---|---|
| Customer | Savings Account | one-to-many | Indirectly |
| Customer | Check Account | one-to-many | Indirect |
| Customer | Note | one-to-many | Indirect |
| Customer | Physical Address | one-to-one | Object Reference |
| Customer | Postal Address | one-to-one | Object Reference |
| Note | Teller | one-to-one | Object Reference |
| Savings Account | Transactions | one-to -many | Object Relationship |
| Savings Account | Transactions | one-to-many | Object Relationship |

## 2.4 Data Model

The data model associated with the Business Object model is described in 2.2.3, “Amended CBConnnector Bank Business Object Model” on page 18. We have already described the influence that the CICS and database back-ends have had on our object model. Figure 15 describes the data model used by the back-end to store objects from within the CICS back-end. In our example only the **Customer** and **Address** components have been

implemented (through a single CICS screen). Primary keys are highlighted in italics.

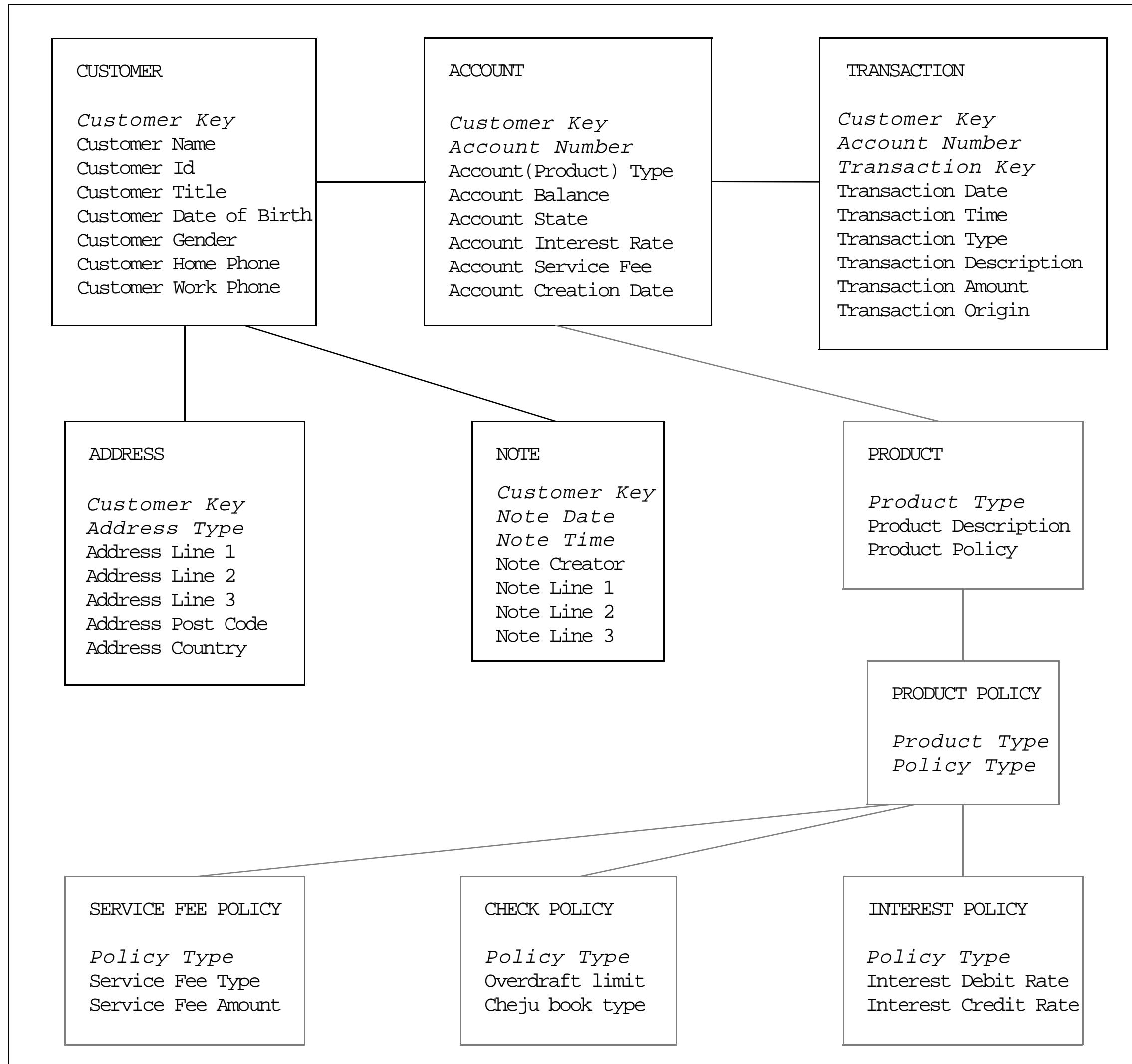



*Figure 15. The Data Model*

The current implementation includes the **CUSTOMER** and **ADDRESS** Data Objects. Future transactions can be added for the other types.

## 2.5 Summary

The banking industry is facing new challenges as both retail and commercial sectors require rapid service evolution to meet industry competition and consumer needs. The Information Technology supporting these services must provide rapid software evolution, legacy system support, component interoperability, and coexistance of service implementations. These are characteristics of a distributed object architecture and in particular, OMG's CORBA technology. By using languages such as Java, strategic applications can be developed quickly and efficiently within a distributed CORBA environment.

Our Business Object model separates essential banking components into Managed Objects with the division of interface specification and implementation. Interface inheritance provides specialization of shared concepts and isolation of behavior implementation. The environment and tools restricted the use of a rich object model, and instead, we were limited to a flat model structure using only attributes with basic types. This chapter described the motivation of our initial model and the subsequent restricted model.

# Chapter 3. Rational Rose Implementation

Rational Rose is an object-oriented analysis and design modeling tool. It can be used to design an application and then to export the design to Object Builder, where the implementation can be completed. This chapter covers the setup and use of the Rational Rose bridge to Object Builder. It describes an initial object model constructed following the analysis and design work. We look at the technique for migration to Object Builder then finally provide a post-analysis of the Rational Rose approach following implementation in Object Builder. Our conclusion is that the analysis, design, and implementation cycle is indeed a cycle, and iterative development of the application is essential and inevitable. Thus, we provide a reimplementation of the Rational Rose model as finally reflected in the Object Builder model.

## 3.1 Setting Up the Rational Rose Bridge

Rational Rose must be modified to enable exporting of models into Object Builder.

1. Copy the *Rose_cpp.pty* and *Rose_cpp.mnu* from the CD-ROM's directory, *drive:\CBroker\Rose*, to the directory where Rational Rose is installed, for example, *C:\Program Files\Rational\Rational Rose Modeler 4.0*.
2. Update the *rose.ini* file in the *drive:\winnt* directory.
   - Find the entry `[Rational Rose 4.0]`, and update its contents, as follows (substitute your Rational Rose installation directory):

```
ROSE_MENU_PATH="C:\Program Files\Rational\Rational Rose Modeler
4.0\.rose_cpp.mnu"
ROSE_PTY=C:\Program Files\Rational\Rational Rose Modeler
4.0\.Rose_cpp.pty
```

   - Find the entry `[Virtual Path Map]`, and add the following statement immediately after the following entry (substitute your CBConnector installation directory):

```
BOSS_PATH=C:\CBroker\rose
```

© Copyright IBM Corp. 1998

## 3.2 The Rational Rose Model

The design objectives, leading to the first object model for the CBConnector Bank application, were the following:

1. The application's primary aim is to present the functionality of CBConnector.
2. The operations the application is providing have to resemble real-life application requirements as much as possible.
3. The complexity of the application has to be limited to allow the use of it for presentation purposes without extensive, in-depth domain knowledge and to keep it focused on the primary goal, which is presenting the possibilities of CBConnector.
4. The boundaries of the application functionality have to be chosen while keeping in mind the resources allocated to the project

Responding to these somewhat conflicting requirements (as, to some extent, they usually occur in real-life projects, too), we came up with the following model, which resembles some parts of the banking domain. This, however, is deliberately narrowed down to a very few object classes and operations on them.

So, the following is the object model we created in the design phase. This is included on the sample CD-ROM in *CBBank\CBroker\Common\RationalRose\InitialModel.mdl.*

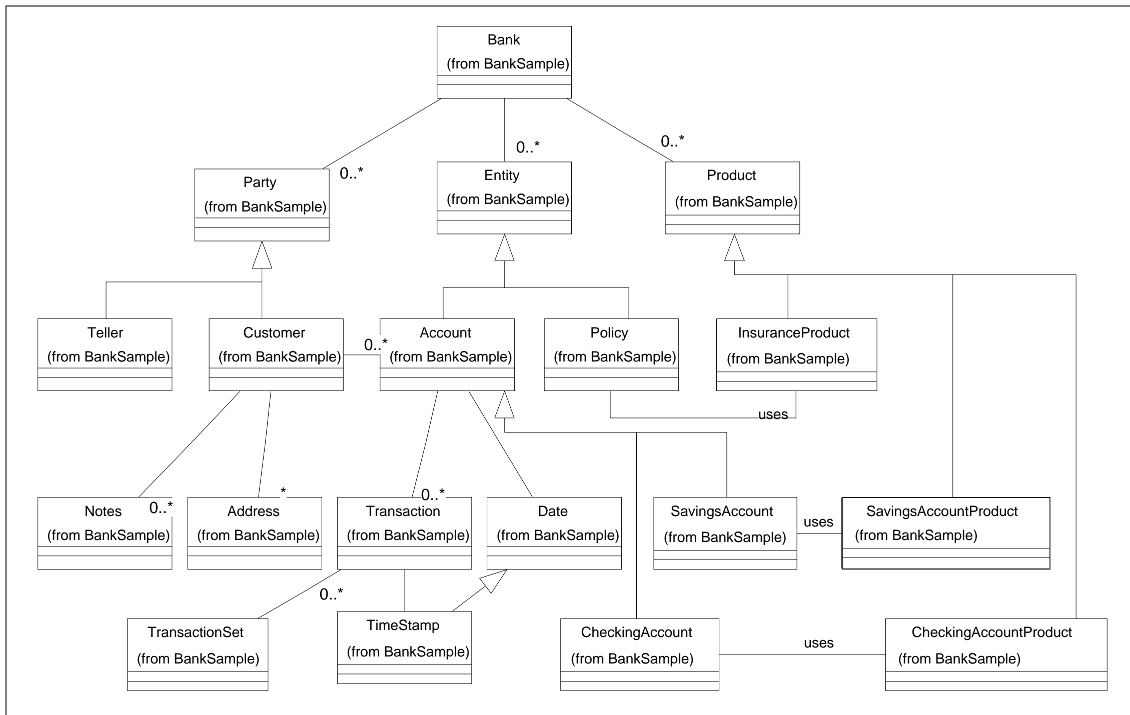



*Figure 16. Initial Rational Rose Object Model*

There is a **Bank** class, objects of which aggregate parties (of class **Party**), entities (of class **Entity**) and products (of class **Product**).

Parties are tellers (of class **Teller**), working at the bank, and customers. Customers have addresses (of class **Address**) and can aggregate notes (of class **Note**). Although it is included in the model, we did not elaborate on the teller properties.

The bank's products are the types of financial services the bank provides. They are not to be confused with actual entities, such as "Jane's savings account". They are actually categories, such as "checking account product", "insurance policy product", "cash loan product" and others, defining conditions and strategies for all entities of this product. There is one product per category of entities. The product is, in a sense, behaving like the class (static) part of the entity category definition. Therefore, we have a one-to-one relation between entity classes and their corresponding product classes.

Among **Entity** classes, instances of which represent the actual financial services objects relating to individual customers, we narrowed out model to the **CheckAccount** and **SavingsAccount** classes, both being subclasses of **Account**.

We included a **Policy** class in the model to hint to other entity classes that would appear in a real-life banking application, but we refrained from detailing or implementing this class.

An **Account** aggregates the **Transactions** made against it.

Customers can have a number of different accounts. So a **Customer** object aggregates **Account** objects.

We did the modeling using Rational Rose. This forced the introduction of some classes (**Date** and **TimeStamp**) that, being trivial, should normally be implemented as structures.

## 3.3 Rational Rose Model in Object Builder

The Rational Rose model can be used as input to the Object Builder via the Rational Rose Bridge. In order to do so, the model has to be extended to reflect the fact that all its classes (which will be mapped to business objects) inherit from the class **IManagable**. Initially, the catalogs *boim.cat*, *services.cat*, and *managed.cat* should be loaded into Rational Rose from the *CBroker\rose* subdirectory.

As recommended in the IBM *CBConnector Cookbook Collection: First Steps,* redbook, SG24-2033, the Managed Object Framework is imported into a new Rational Rose model before beginning the creation of the object model. This provides the essential Object Builder (and thus CBConnector) components.

The Rational Rose object model is migrated to Object Builder using the new operation added to the Rational Rose File menu. The bridge will start Object Builder, which then begins the translation process.

Three steps steps are necessary in the translation process:

1. Top-level classes and packages from Rational Rose are exported as either separate Object Broker models or as a single model. Each class and package has a selection checkbox for choosing its inclusion in the created models. Multiple models should be selected when more than one person will work on the ensuing models. This option is also useful when any non-trivial number of classes are used. This allows individual compilation of model components without the need for total model compilation (which

in Object Builder 1.2 would compile every Java file on any source modification).

2. Classes are mapped into Business Object interfaces in the model but the user may also select Implementation, Key, and Copy Helper generation. These are best attempts at generating components that Object Builder will use. For example, the key will include any attributes marked as `isPrimaryKey` in the Rational Rose model and the Copy Helper all public attributes.
3. Mappings from Rational Rose one-to-many associations into Object Builder representations. These can be either object relationships or simple sequences of object references. Object relationships will automatically generate add, remove, and list operations, while the sequences must be manually programmed by the user.

The exported model should finally be examined to verify the generated business object interfaces and the Implementation, Key, and Copy Helpers that were generated.

## 3.4 A Bridge Too Far

The Rational Rose model was generated following our early analysis and design process. We made an early decision not to maintain the model in step with changes in the Object Builder model. Thus, we used a one-off bridging process to produce an initial Object Builder model and then maintained this model within the Object Builder environment. This had both disadvantages and advantages. Our initial learning curve with the Object Builder and CBConnector environment was steep, and our early development phase often required reimplementation of the application. We learned about restrictions with Object Builder Release 1.2, which guided our eventual implementation.

Unfortunately, our final Object Builder model could have taken advantage of many facilities in the bridge, but we were not aware of the potential use of these at the time we built the Rational Rose model. The lesson here is that the analysis, design, and implementation of any application is an iterative task. Given more time, we would now redefine the model in Rational Rose and once again export to Object Builder. By keeping as much code as possible out of Object Builder, it can easily be hooked into a new Object Builder model using Business Object external code referencing.

In the spirit of the iterative design model, a final Rational Rose model is shown in Figure 17. The final model file is included on the sample CD-ROM in *CBBank\CBroker\Common\RationalRose\FinalModel.mdl*.

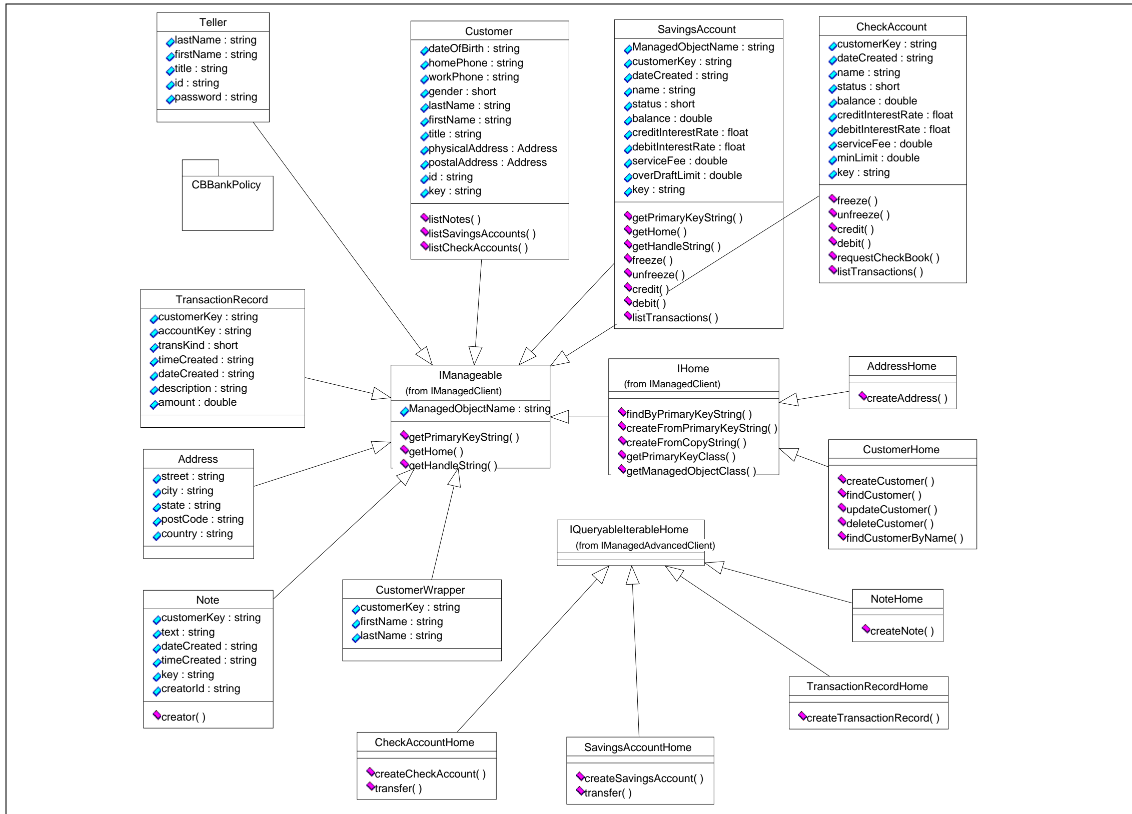



*Figure 17. Final Rational Rose Model*

As described in 2.2.3, “Amended CBConnnector Bank Business Object Model” on page 18, the model is a flat structure that uses only simple data types. The **IManageable**, **IHome**, and **IQueryableIterableHome** are provided by CBConnector. Further description of this model and the subsequent implementation in Object Builder is described in 5.3, “Persistent Implementation” on page 68.

# Part 2. Implementation

In Part 2, we discuss the client sample development approach, the middle tier and our legacy system implementation. We recommend that you work in concert with the CDROM while studying the model and implementations on all three tiers. Now we proceed to the walkthrough of the CBConnector Bank implementation.

© Copyright IBM Corp. 1998

# Chapter 4. Client Implementation

In this chapter, we illustrate the client implementation as a sample approach to the client development paradigm.

## 4.1 Introduction

We begin by describing the design and implementation of the CBConnector Bank application client. For reasons of modularity, we decided to split the client into two logical partitions–the "client front-end" and the "client back-end". The client front-end is concerned with the presentation layer and the GUI interactions with the user. The back-end is concerned with handling the interactions with CORBA and the distributed objects that make up the application. The intent is to hide from the GUI programmer all the complexities of the distributed world. Another advantage of this approach is that it will facilitate the development of different types of client front-end. So, for example, an applet-based client could communicate with the back-end through the use of Java servlets.

It must be stressed that the approach we have adopted here is merely an example of how you might structure your client application. There are many possible approaches that are equally valid and how you structure your own applications will depend on your own particular circumstances and requirements.

## 4.2 The Client Back-End

The client back-end is structured as described in chapter 10 of the *IBM CBConnector Cookbook Collection: First Steps*, SG24-2033, redbook. For a complete description, please refer to that redbook. We describe the structure briefly here.

The components of the CBConnector Bank client application are shown in Figure 18 on page 36. There are two general classes, **CBCBase** and **CBProxy**, and one **Proxy** class for every Business Object that exists in the application. We describe each component below.

***Graphical User Interface***

The graphical user interface sees the world completely in terms of the Business Object proxies. It has no interaction with any Component Broker facilities.

© Copyright IBM Corp. 1998

### *CBCBase*

**CBCBase** is an abstract class which contains only static methods. It provides a selection of utility methods for carrying out common ORB-related functions such as:

- **resolveORB** - initialize the ORB
- **resolveNameService** - get a reference to the Name Service
- **resolveFactoryFinder** - get a Factory Finder for a given scope
- **resolveHome** - get a home for a given Factory Finder and object interface

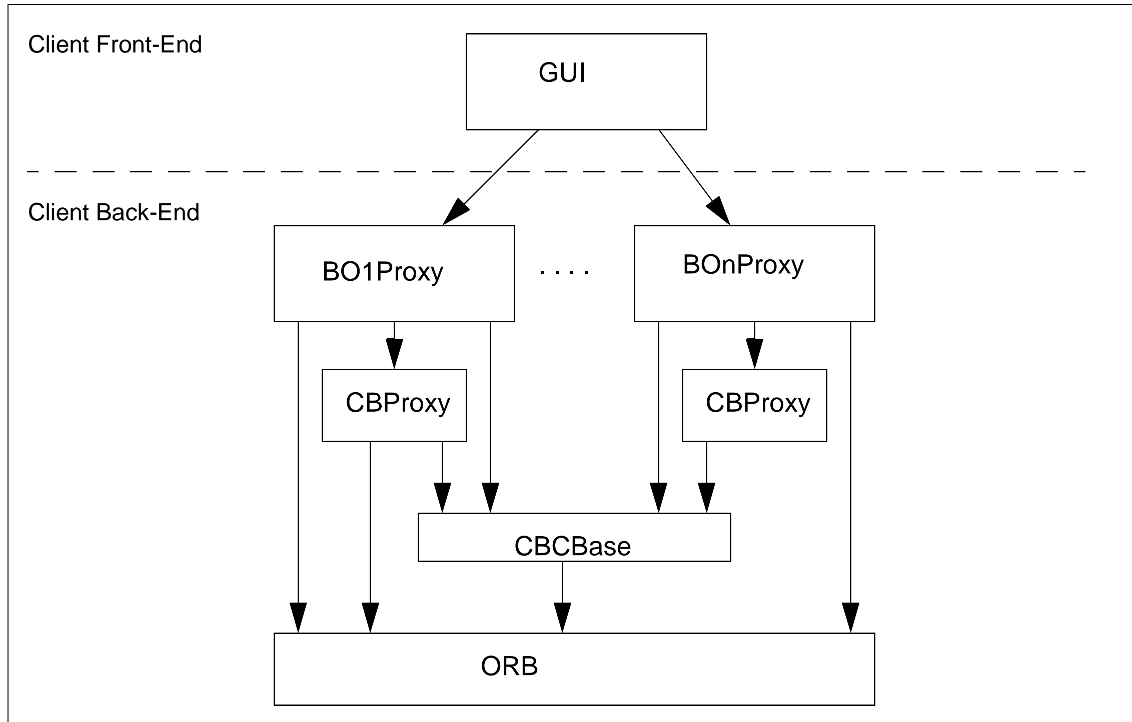



*Figure 18. The Structure of the CBConnector Bank Proxies*

### *CBProxy*

CBProxy is an abstract class from which all the Business Object proxies inherit. It defines instance variables for holding generic objects such as a Factory Finder and a home which are common to all proxies. It provides methods to carry out some functions which must be provided by all proxies, for example:

- **init** - do the proxy initialization
- **createMO** - create a Managed Object
- **findMO** - find a Managed Object

***Business Object Proxies***
A Business Object proxy exists for each Business Object. It is implemented as a JavaBean and is described more fully in the next section.

### 4.2.1 The Role of A Proxy

The primary role of the proxy is to provide the GUI client with a complete interface to the distributed world. It does this by wrapping underlying CBConnector objects and then providing either a straight delegation model or, for certain functions, a purpose build mapping to equivalent CBConnector facilities.

The proxy encompasses the following CBConnector elements:

- *A Copy Helper* - The Copy Helper is used to hold a local copy of the state of the remote Business Object. The proxy provides JavaBean-compatible getter and setter methods for all the attributes of the Business Object which are delegated to the corresponding methods on the Copy Helper.
- *A Key Helper* - The key helper is for constructing the Primary Key of the underlying object.
- *A Specialized Home* - Most Business Objects have a specialized home which provides extra facilities in addition to those found in a normal home. For example, a specialized home in the CBConnector Bank application is often used to create the Business Object.
- *The (remote) Business Object* - This is the CORBA object reference of the remote Business Object. All the operations on the Business Object will be mirrored on the proxy and will be delegated to the Business Object when invoked by the (GUI) client.

### 4.2.2 Using a Proxy

This section describes the process for using Proxy Objects.

#### 4.2.2.1 Creation/Initialization

Proxy creation takes places in two stages–object creation using the normal Java mechanisms and proxy initialization. Every proxy has an **init** method that must be called before the proxy becomes usable. The **init**

method takes a `scope` as a parameter and performs its initialization at three levels:

- Base Initialization

  This is performed only once per session (by **CBCBase**) and carries out such tasks as ORB initialization and acquiring a reference to the naming service.

- Generic Proxy Initialization

  This level is carried out by **CBProxy** and is responsible for finding a Factory Finder (using the scope provided as a parameter) and then using that Factory Finder to find a home for the Business Object represented by the proxy. If multiple homes are located in the same scope, then finding a Factory Finder is optimized by the use of a local hashtable, which maps scopes to Factory Finders.

- Specific Proxy Initialization

  Any initialization that is relevant only to the proxy being created is performed here. For example, a specialized home may be "created" by narrowing the reference to the home acquired in the generic initialization stage.

In addition to the default constructor and the **init** method, every proxy provides an additional constructor with a `scope` parameter so that these two steps can be combined.

Once initialized, the proxy is ready to accept requests from clients.

#### 4.2.2.2 The CRUD Methods

In addition to the operations defined on the Business Object's interface, most proxies provide a set of standard **create**, **read**, **update** and **delete** methods. (These methods may not be on all proxies if, for example, it is not appropriate for a client to create a Business Object of that type).

- Create

  To create a Business Object, the client will first set all the relevant attributes. The proxy provides Java Bean-compatible setter methods so that the client can automatically tie the bean to the user interface. As the user enters values into fields, they are automatically propagated to the corresponding setter method which places them in the Copy Helper attribute. When all the attributes are set, the client invokes the **create()** method which will cause the Business Object to be created using either the facilities of the home or, if one is provided, the specialized home. This will return a reference to the remote Business Object which is held locally in the proxy for future use.

- Read

  A read operation is provided to allow the state of a Business Object to be acquired and copied to the local Copy Helper. The proxy must have an object reference to the Business Object. Because the proxy is implemented as a bean with bound properties, a user interface bean that is connected to the proxy will have its fields updated automatically.

- Update

  The **update** method allows the contents of the local Copy Helper to be written to the remote Business Object.

- Delete

  The **delete** method allow the removal of the remote Business Object. It uses the **remove()** method, which causes the permanent deletion of the object.

#### 4.2.2.3 Exception Handling

In keeping with the objective of completely encapsulating the CORBA world, the proxies will trap all exceptions generated while accessing the ORB infrastructure or by remote invocations. Two groups of exceptions are defined by the CBConnector BankProxy package. **CBBankException** is the superclass of all exceptions of interest to the client. Whenever a proxy catches a remote exception, it generates the equivalent exception in the **CBBankException** hierarchy. Whenever an unexpected ORB exception is caught, it is logged by the proxy, and the generic **ORBException** is passed back to the client. This allows the GUI to alert the user that a failure has occurred and that details can be found in the system log.

#### 4.2.2.4 Transactions

In the CBConnector Bank application, most transactions are controlled at the server. The exceptions to this are whenever the Query Service is being used. This happens explicitly in the client in **CustomerMapperProxy.findAllCustomerKeys** and **queryAccount** in both **CheckAccountProxy** and **SavingsAccountProxy**. It also happens in the **listTransactions** method of the Account proxies because the server implementation uses the Query Service and relies on the client to start and commit the transaction.

### 4.2.3 The CBConnector Bank Proxies

In this section, we describe in more detail the behavior of the individual Proxy Objects that comprise the CBConnector Bank client back-end.

#### 4.2.3.1 Customer

The customer is the primary Business Object in the CBConnector Bank application. We describe here some of its more interesting and non-standard behavior.

- Address attributes. In addition to the attributes with primitive types such as `firstname`, `lastname`, and so forth., the customer has two address attributes (physical address and postal address) which are defined as interface types. These must be handled in a special way in the customer proxy. In effect, it allows the client to see these as address Proxy Objects. To do this, the customer proxy defines two instance variables of type *AddressProxy* and provides special versions of the corresponding getter and setter methods. Instead of delegating direct to the **CustomerCopy**, the setter will set the local variable with the supplied proxy, and the getter will simply return the **AddressProxy** object. How are these **AddressProxy** objects created? When creating a new customer, the onus is on the client to first create the address proxies for both addresses, set their attributes as normal, and then to invoke the setters on the **CustomerProxy**. The `create` method on the **CustomerProxy** will then ensure that both address Business Objects are created and the corresponding attributes in the CustomerCopy updated, before the customer Business Object is created. On a customer read operation, the **CustomerProxy** is responsible for creating the **AddressProxy** objects. There are a pair of private utility methods–**readPhysicalAddress** and **readPostalAddress**–that facilitate this operation.
- Key attribute. The Customer Interface has a key attribute that is kept hidden from the GUI client. This refers to a system generated key that is unique for every customer. This attribute is needed because of the persistence requirements of the model. This key must be made available to other proxies (such as the Account proxies) which need to be associated with a particular customer. The **CustomerProxy** provides a utility method, **findCustomerKey**, which returns the customer key based on the customer's `firstname` and `lastname`. Both of these attributes must be set in the customer before this method can be called. The **findCustomerKey** method has two approaches to providing this mapping. The simplest is to use the **findCustomerByName** method on the customer specialized home and then to access the key of the returned customer Business Object. This approach assumes that there is only one customer with a given name. If there is more than one, it simply returns the first. This is probably an acceptable limitation in a demonstration application. The

**findCustomerByName** method uses a **CustomerMapper** on the server side to find the key.

- Direct Query. Another approach is to use the CBConnector Query service directly on the **CustomerMapper** object from the client side. This is a more efficient mechanism and also provides the ability to get all customers keys for customers with the same name. We use the **findCustomerKey** method on the **CustomerMapperProxy** object. This returns a Vector of all customer keys which match the given first and last names. The **CustomerMapperProxy** object is not made available to the GUI client (its constructors are made private). It can, however, access its functionality by using the static **findAllCustomersKeys** method and supplying the `firstname` and `lastname` as parameters.
- dateOfBirth attribute. This attribute is handled in a special way because of the different ways dates are represented at the server and at the GUI client level. The server represents dates as a string with a set format. The GUI client likes to handle dates as an instance of the **GregorianCalendar** class. The getters and setters of the dateOfBirth attribute provide this transparency by using the methods **fromGregorian** and **toGregorian** to provide the transformations.
- Lists. The Customer Business Object provides several methods for iterating over **CheckAccounts**, **SavingsAccounts** and **Notes** that belong to the customer. The customer proxy wraps these list operations with its own versions, such as **listNotes**, which returns a Java Enumeration to the client instead of the CORBA Iterator. The client must step through the Enumeration, and each iteration will result in a remote access to the server. As a convenience, the customer proxy also provides methods, such as **listAllNotes**, which carry out the iteration and return a Vector of all the results. While this can be more convenient for the client, it must be used with care since it could potentially result in a large number of accesses across the ORB.

#### 4.2.3.2 Address

The **Address** Business Object is the only object in the CBConnector Bank application that does not have a key. It is always accessed through the **Customer**. Accordingly, the CRUD methods of the **AddressProxy** are protected and cannot be accessed directly from the client. Otherwise, the **AddressProxy** does not have any interesting features.

#### 4.2.3.3 CheckAccount and SavingsAccount

The CheckAccount and SavingsAccount proxies are virtually identical to each other. This duplication was necessary because of the absence of inheritance from the model as explained elsewhere in this redbook.

- Key. The Account Business Objects have a two-part key; the first part is the account key, and the second part is the key of the customer to which the account belongs.
- Creating accounts. **AccountProxy** objects are created in the usual way. However, before the `create` method is called the client must associate a **CustomerProxy** with the account. He does this by using the `useCustomer` method. The **CustomerProxy** parameter passed to this method must have its key field set because this how the account is associated with the customer at the server back-end.
- Accessing accounts. Accounts are usually accessed in one of two ways. The first way is to use the **list** (or **listAll**) method on the **CustomerProxy** to return an Enumeration or a Vector of **AccountProxies**. The other way is to access the account directly using the **queryAccount** method. Before this method is called, the account key must have been set in the proxy using `setKey`. **queryAccount** uses the Query Service to find the correct account. Note that this will normally be invoked transparently to the GUI client; the client simply issues the standard read request and the query will be invoked, if required.
- Lists. The Account proxies provide methods for iterating over the Transaction Records that belong to the account. Two methods are provided: **listTransactions** and **listAllTransactions**. The first returns a Java Enumeration over that the client can iterate to access each element one at a time. The second provides a convenient way to get all Transaction Records returned in one operation. As for the **CustomerProxy** lists, care must be taken when using this second approach.
- Operations. The **Account** Business Object provides several operations for managing the behavior of accounts. The proxies provide straight delegation with a couple of exceptions. The **setBalance** method has been made private so that it is not invokable by the client. Generally, the balance should only be changed by crediting or debiting funds. The **transfer** method has been made accessible on the proxies although it is actually implemented on the account (specialized) homes.

#### 4.2.3.4 Teller

The **Teller** Business Object is the only object in the CBConnector Bank application that does not have a specialized home. The **TellerProxy** still has a standard `create` method but this is delegated to the **createUsingHome** method, which creates a teller using the standard

home facilities. The **Teller**'s key field is its **id**, and this must be set before issuing a read operation.

#### 4.2.3.5 Note

The **Note** Business Object is fairly straightforward.

- *Key*. The **Note**'s key field is comprised of its own key plus the key of the customer to which it belongs.
- Creating Notes. NotesProxies are created in the usual way but before calling the `create` method the `useCustomer` method must be called to associate the correct customer key with the Note. As when creating Accounts, the CustomerProxy parameter passed to the `useCustomer` method must have its key field set because this how the note is associated with the customer at the server back-end.
- creator attribute. The note has a creator attribute which is an interface type referring to the Teller who created the note. As with addresses on the **Customer**, the **NoteProxy** handles this in a special way by defining an instance variable of type *TellerProxy* and providing special versions of the **getCreator** and **setCreator** methods. The **setCreator** takes a **TellerProxy** as a parameter and updates both the local instance variable and the teller in the **NoteCopy**. The **getCreator** simply returns the local instance variable. Before creating a **Note**, the client must first create a **TellerProxy** and ensure that it contains a reference to a teller Business Object. On retrieval, the **read** method on the note is responsible for creating the **TellerProxy**. A method called **readCreator** is provided to do this.

#### 4.2.3.6 TransactionRecord

The **TransactionRecord** object is the only object in the CBConnector Bank application that can not be created directly by the client. As such, it has none of the CRUD methods associated with the other proxies. The only way a **TransactionRecordProxy** is created is by the **listTransactions** or **listAllTraSnsactions** methods on the account proxies.

The **TransactionRecord** defines a key field, but this is never used for retrieval. Also, although the **TransactionRecord** Interface defines its attributes as read/write, the proxy makes the setters private to prevent the GUI client from changing them.

#### 4.2.3.7 CustomerMapper

This object is normally only used at the server and is never made visible to the GUI client. It provides a mapping from customer name to customer key

which allows customers to be retrieved in a more natural way. Direct query over customers is not possible because the **Customer** is implemented in a CICS service. The use of a **CustomerMapper** to find customers is described in Section 4.2.3.1

### 4.2.4 A Note on Proxy Optimization

For this application, we decided to introduce some optimizations to the way proxies are initialized. At initialization time, each proxy will acquire a Factory Finder based on a given scope and acquires a home based on a specified object interface string. The methods to do this, **resolveFactoryFinder** and **resolveHome**, are in the **CBCBase** class. The original implementation of these methods always uses the ORB to resolve the objects. As the CBConnector Bank application is continually creating proxies, it was decided to optimize these methods.

A hash table called scopeTable was introduced to cache scopes and their corresponding Factory Finders. Whenever **resolveFactoryFinder** is asked to find a Factory Finder, it first looks in the scopeTable to see if the given scope has already been resolved. If so, it simply returns the corresponding Factory Finder. Otherwise, it goes to the Name Service, and when the Factory Finder is found, it caches it in the scopeTable.

Another hash table, called ffTable, was created to map from Factory Finders to the homes found by that Factory Finder. The mapping to the home is actually from the object interface string; so the ffTable provides a link from the **factoryFinder** to another hash table which stores this relationship. When **resolveHome** is called with a Factory Finder and an object interface, it first looks up the ffTable to find the factoryFinder entry. The hash table referenced by this **factoryFinder** is then searched to find the home corresponding to the object reference. If an entry exists, then the home is simply returned. Otherwise, the Factory Finder is used to resolve the home in the normal way, and a new entry is put in the table.

## 4.3 The Client Front-End (Graphical User Interface)

The design of the front-end for the CBConnector Bank application has to satisfy certain functional and quality criteria. It had to be implemented in Java so that it could be used both as a stand-alone application by a bank teller and as a home banking user interface running as an applet within a browser. Of course, the available functionality should vary due to different access rights of the teller operating the program in the bank and of the customer doing so from a browser over the World Wide Web.

To simplify maintenance and logistics, only one version of the front-end program should exist, being capable of both running as a Java application or as an applet.

The front-end functionality had to provide for the basic teller functions of maintaining customer information (creating, querying, updating or deleting customer information, querying the customer's accounts and changing their status), maintaining account information, withdrawing or depositing money.

A fully functional bank teller application would undoubtedly go far beyond the scope of this project. So emphasis was given to developing a subset of the full functionality while trying to satisfy the various design criteria as much as possible. Therefore, we dare ask the benevolent reader to attribute functional omissions and/or simplifications, which will be easily spotted by the experienced bank IT specialist, to the restricted scope of this project.

The interaction with the user should be straightforward and intuitive, and it should enable the user to freely organize his/her desktop by optimally resizing the graphical user interface components.

Furthermore, the development being part of a larger development, the front-end design had to account for having to cope with an iterative development and implementation process. To do so, the design emphasized the strict division of the model and the view parts. It concentrated the interactions with the Component Broker middle tier in the model (the client back-end). It placed all user interaction and concurrency aspects in the client front-end.

The front-end part provides the user with a frame containing four panes:

- The context pane presents a hierarchical view of the model structure (bank, customers, products) and of operations that can be performed on the objects.
- The action pane contains detailed information pertinent to an operation in progress.
- The utility pane allows certain utility operations.
- The operation status pane allows easy tracking of multiple operations in progress.

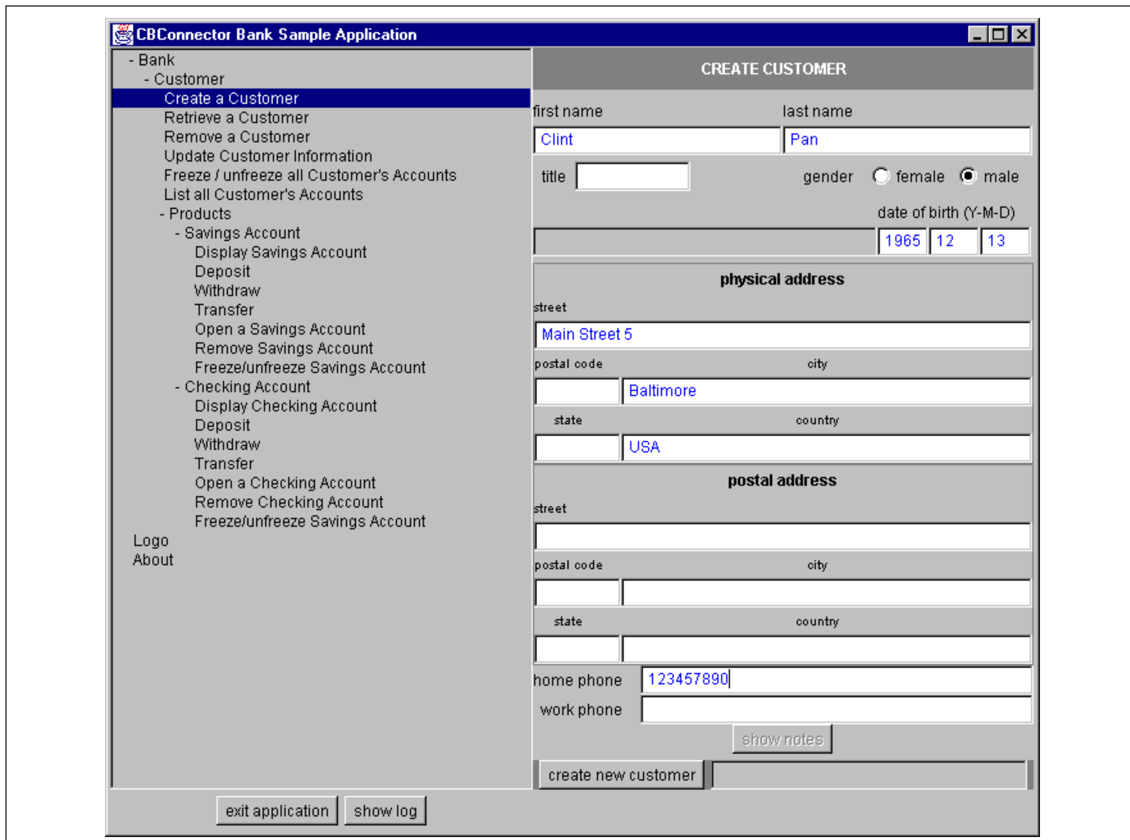



*Figure 19. The CBConnector Bank Sample Application User Interface*

Additionally, a logging frame can be displayed containing log information about operations performed or currently in progress.

Selecting an operation in the context pane loads the action pane with the appropriate interaction view. Typically, the user provides some information necessary to perform the operation and then requests to proceed with the operation. To perform the operation, the view will request some actions from the model part of the front-end. The model part will occasionally have to cross the Object Request Broker to fulfill the request. Depending on the system architecture and the current workload, this process may take some time. To avoid disabling the user and thus effectively impeding his/her productivity, the operation in progress should proceed without blocking the graphical user interface. That way, the user can initiate several operations in parallel.

The operation status pane is at any time providing information on the status of operations in progress using color-coded, light-emitting-diode (LED) icons. The icons can be conveniently used to select the interaction view for the corresponding operation to view the details of the operation.

The following figure depicts the typical steps involved in performing an operation.

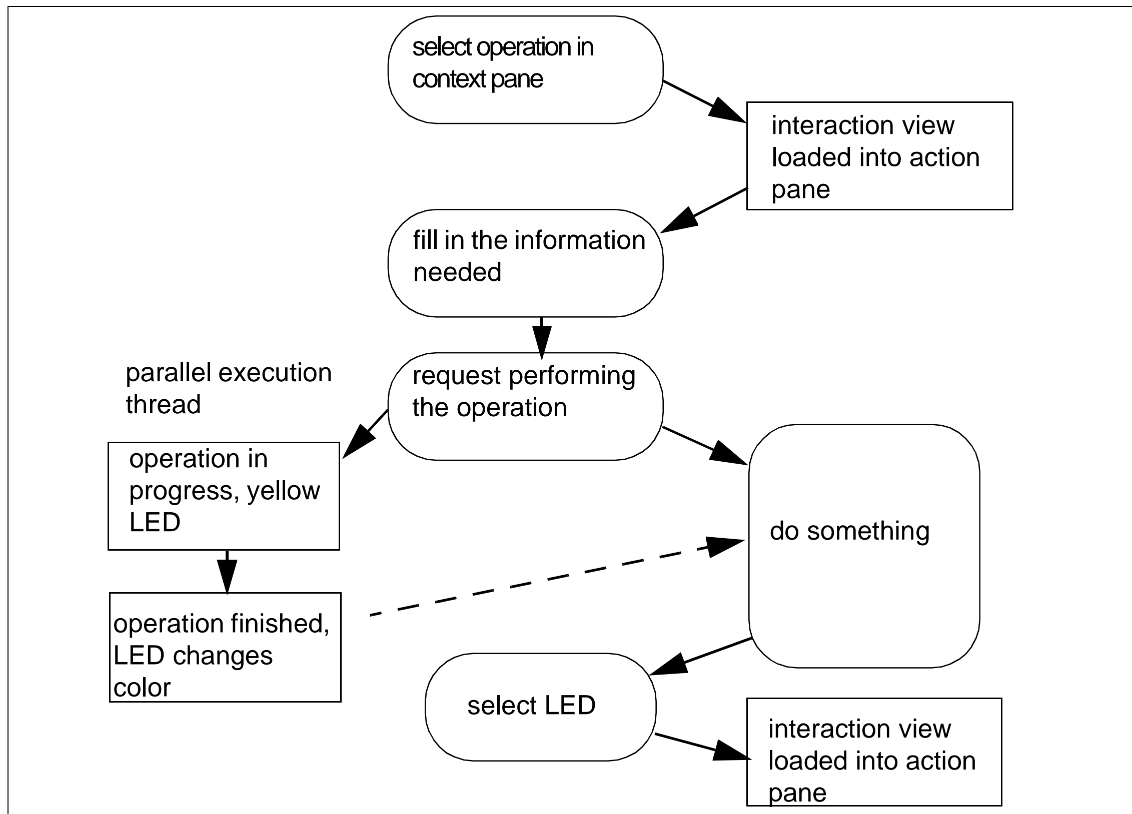



*Figure 20. Typical Steps in Performing an Operation*

### 4.3.1 The View Structure

An operation is performed through a series of steps. There are basically two different types of the steps:

1. *Data entry* and/or *viewing* steps, where certain data is displayed to the user and/or the user can enter or modify data. To perform these steps, the program must present a graphical user interface and wait for user interaction.

2. *Data retrieval* and *processing* steps, during which the program is communicating with distributed objects to update its data or performing other tasks that do not request human interaction.

Our client solution mirrors this situation by providing two class families:

1. The *operation panel* family, inheriting from the **OperationPanel** class,
2. The *Business Object action* (BO action) family, inheriting from the **BOAction** class.

The operation panel family of classes are derived from the **java.awt.Panel** class and can be displayed in an application frame or an applet. The display of the operation panel is enacted in the main thread. Only one operation panel can be visible and active at a time in the GUI.

An operation panel provides the user interaction interface and consists of the required data fields and one or more buttons the user can use to signal transition to the next step. Typically, the next step after the interaction with the operation panel will involve data retrieval or some other type of processing. A step of this type will be defined in an object of some BO action subclass, and the operation panel will delegate the processing to it. A BO action object has no visual appearance.

Data retrieval or processing steps can well be time-consuming because they may involve a lot of ORB communication and/or other time-demanding processes. To enable the user to go on using other functions of the GUI, the operation panel will spawn a new thread and commit the performance of the data retrieval/processing step to it. Therefore, the BO action object has to be a **Runnable** object, its **run** method performs the processing needed.

The operation panel provides a field (a Label or a TextField) to receive the status of the retireval step. The BO action object provides three methods to signal the busy state, normal completion and the error completion status by placing color-coded messages in the status field of the operation panel. These methods also handle the creation and state transitions of the status-reporting LEDs.

Before starting the processing step, the BO action object's **run** method creates the CBConnector Proxy Objects needed and sets their state to reflect the data supplied by the operation panel (this may be only a subset of the proxy's state, depending on the operation requested). It then performs the actual processing and reflects the changes in the operation panel.

### 4.3.2 Operation Panels

The operation panel object is a subclass of **java.awt.Panel** and is typically displayed in the action pane of the graphical user interface. Its visual appearance presents operation-relevant data fields for inspection or update. It also provides one or more buttons the use of which activates the next step of the operation. A status field of type **java.awt.TextField** is provided in the panel to inform the user of the status of data retrieving/processing steps performed by the BO action objects.

One of the quality criteria we have pursued was the flexibility of the user interaction, such as his/her freedom of organizing the desktop according to his/her needs and preferences. Therefore, the user interface had to be able to graciously cope with resizing actions the user does.

Of course, there are practical limits to the size of the user interface that can be meaningfully handled. To enforce these limits, a user trying to resize the application frame below them will be overridden, and the frame will "bounce back" to the minimal size acceptable. Upon start of the application, the user interface uses the whole available screen. These resizing strategies can't, of course, be applied when the application is run as an applet within a browser. In that case, the responsibility for supplying an acceptable size lies with the HTML-page designer.

The behavior we just described enforces just the size limits within which the user may freely resize the user interface. It is important that the interface can adapt to these different sizes and preserve an agreeable look. Of very practical interest is that components of the user interface don't simply "disappear" from it by being clipped when the frame gets too small (provoking unnecessary delay and frustration as the user searches for the component). To satisfy the stated criteria, we have implemented all operation panel classes using **java.awt.GridBagLayout** as the layout manager.

It is unproductive and frustrating when an initiated data retrieval or processing step fails because of incomplete or inconsistent data entered. To reduce the possibility of such mistakes, buttons that initiate the steps are disabled when the user-supplied data needed is missing or inconsistent. The criteria vary, of course, between different operational panels, and they are implemented in the **processChange** method. Any change in the relevant data fields causes **processChange** to be executed, readjusting the enablement of the step-initiating buttons.

For productivity and consistency reasons, we implemented some parts of the operation panels as separate panel subclasses and aggregated them.

Operation panels were produced using VisualAge for Java. This limited, to some extent, the use of inheritance since it is not possible to modify and extend the visual appearance in the subclass.

### 4.3.3 BO Action Classes

The only functionality provided by a BO action class is performing a data retrieval or processing step in the operation. It is a typical "command" class, having no visual appearance. It implements the **java.lang.Runnable** Interface to be able to be run in a separate thread. The only method (apart from the constructors) a specific subclass of **BOAction** has to implement is the `run` method, and it should contain all the code needed to perform the step.

The **run** method first informs the user that this step is under way by invoking the method **signalBusy** (inherited, together with **signalErrorCompletion** and **SignalOKCompletion**, from the **BOAction** class). It then creates Proxy Objects where needed and invokes methods on them.

Upon termination of the **run** method execution the step has been performed and the spawned thread dies. However, the step could have been successfully performed or it may have failed. It is the responsibility of the `run` method implementation to inform the user of the success or failure by using the methods **signalErrorCompletion** or **SignalOKCompletion**, respectively.

### 4.3.4 Common Data Container

The need arises for some information to be accessible through all of the application. Information on scope names, for instance, will be needed in many places. The same applies to some object references.

To provide a consistent facility for accessing common information, we implemented the **CommonDataContainer** class. In principle, it is a dictionary used for storing information available to keyed access. It provides the **set** method to store the piece of common information and methods **get** and **getObject** to retrieve `String` data or general object references, respectively. To enable the setting of not-hardcoded information (data provided through an external source, such as a file), the **setup** method is provided. It parses the stream input and initializes common data items. However, only `String` items can be acquired that way. The format of the stream data follows the same section notation and comment syntax as described in the **Hierarchy** class. It parses the section `common data container` for data item definitions of the following form:

```
<key>=<value>
```

The following is an example of the section:

```
//*******************************************************
# common data container
//*******************************************************

// parameters related to the ORB

hostName=coffee04.lagaude.ibm.com // the bootstrap host name
portNumber=900 // the port number used

// parameters related to the business objects

customerScope=/host/resources/factory-finders/TomatoScope
addressScope=/host/resources/factory-finders/TomatoScope
accountScope=/host/resources/factory-finders/TomatoScope
checkAccountScope=/host/resources/factory-finders/TomatoScope
savingsAccountScope=/host/resources/factory-finders/TomatoScope
noteScope=/host/resources/factory-finders/TomatoScope
tellerScope=/host/resources/factory-finders/TomatoScope
```

### 4.3.5 DataSupplier Class

Reading the *CommonDataContainer* file when executing the program as an applet within a browser presents a sandbox problem. To avoid the use of the file, the applet mode of the program uses a **DataSupplier** object to get the information needed. **DataSupplier** provides the information in equivalent format. Although it has to be present when compiling this software, the **DataSupplier** class will not be used in application mode.

The **DataSupplier** class has to mirror the information in the *CommonDataContainer* file. When this information is altered, you should produce a new **DataSupplier** class. You can do so by calling the utility DataSupplierCreator, which is included with the application.

To run DataSupplierCreator, type on the command line:

```
java DataSupplierCreator CommonDataContainerFile
```

or

```
java DataSupplierCreator CommonDataContainerFile DataSupplierPath
```

The `CommonDataContainerFile` parameter should specify the fully qualified file name of the file containing the common data information. The *`DataSupplierPath`*, if present, is the directory where the **DataSupplier** class will be created. If you omit the second parameter, the **DataSupplier** class is created in the current directory.

Note: For the DataSupplierCreator program to run correctly, the javac compiler must be installed and accessible (through the PATH environment variable).

### 4.3.6 Hierarchy

The **Hierarchy** class presents the user with a hierarchical context menu. It reflects the hierarchy of the application objects and the operations that can be invoked on them.

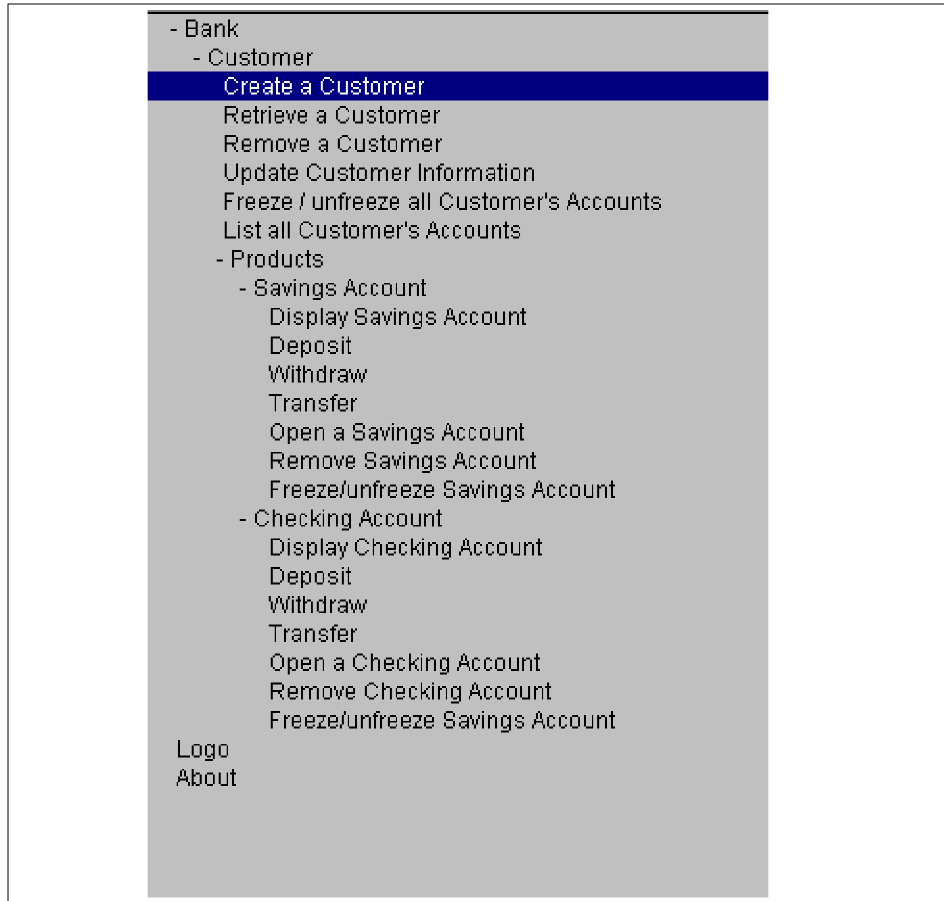



*Figure 21. Hierarchy List Panel*

The objects are represented in the hierarchy by nodes that can be exploded to show lower hierarchy levels, or the nodes can be imploded to hide them. The leaves represent operations that can be performed on the objects. Selecting a leaf generates the **OperationSelectedEvent**. In the application,

this event triggers the corresponding operation by loading the appropriate operation panel into the action pane.

The **Hierarchy**-type object, being the view part, collaborates with its corresponding model that is represented by a **HierarchyNode** object. This node is the root of the actual hierarchy tree.

A node aggregates a number of objects, all of them being **HierarchyNode** or **HierarchyLeaf** objects. It can, on request, provide a collection of its aggregations. It has also knowledge of its exploded or imploded state. A **HierarchyLeaf** has an operation panel associated to it, and this panel is activated when the leaf is selected.

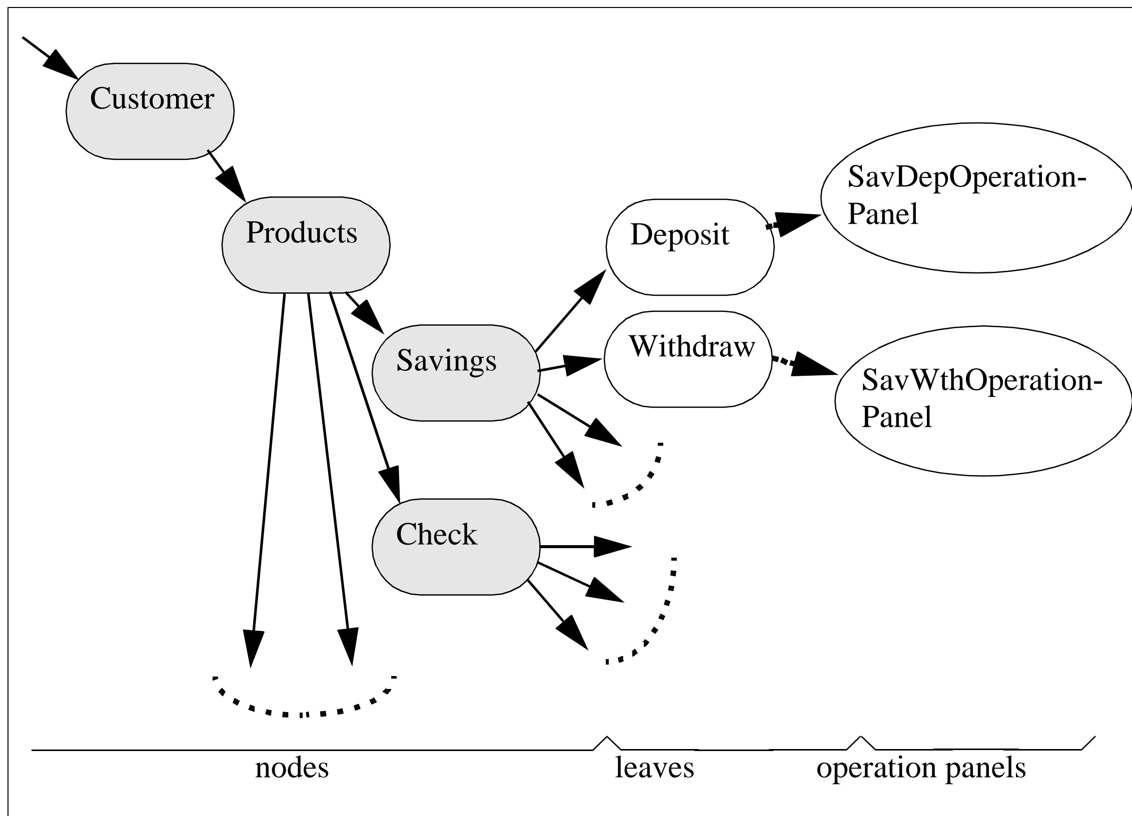



*Figure 22. Segment of the Hierarchy Model*

There are a number of ways to provide the hierarchy information to the **Hierarchy**. One is to assemble the tree by creating node and leaf objects within the application, build the structure and pass it (the reference of the root

node) to the **Hierarchy** object. Another is to provide a **BufferedReader** containing the hierarchy information to the **Hierarchy** object. That way, the hierarchy structure can be modified without the need for code modification. The next section provides a brief informal description of the specification format.

As a convenience, the **Hierarchy** can extract the same information from a file when given a string file name.

To enable the use of **Hierarchy** in untrusted applet configurations, the external structure data can be provided by an object of the class **DataSupplier**. The drawback of using a **DataSupplier** object is that it contains the hardcoded hierarchy structure, so that, to change the structure, the code of the class has to be changed. To simplify this process, a Java utility program, **DataSupplierCreator**, is included. The first, mandatory command line parameter is the name of the file containing the structure definition. The second parameter, if supplied, is the directory path where the **DataSupplier** class will be generated. If no path is supplied, the class is generated in the current directory. To run the DataSupplierCreator, the JDK must be installed and javac-accessible through the `PATH` environment variable. The result of the DataSupplierCreator execution will be a *DataSupplier.class* file for use with the **`Hierarchy`**.

### 4.3.7 Hierarchy Specification Format

The specification of the hierarchy structure in a file has to adhere to the following rules.

Any part of a source line following a double slash (`//`) is regarded as a comment and ignored. Blank lines are also ignored.

The file can contain more sections, each one preceeded by a section header in the form of a number sign (`#`) followed by the section name. The section containing the hierarchy structure information has to be preceeded by

```
# hierarchy tree
```

and terminated by the start of another section or by the end of file. All other sections are ignored by the **`Hierarchy`**.

A node is specified by a node statement, followed by its subnodes and/or leaves and the end statement. The node statement has the form:

`N`*xtext*`|exploded`

or

`N`*xtext*`|imploded`

where *x* is the authorization level and *text* the node's name that will be displayed in the hierarchy list. `exploded` or `imploded` define the initial state of the node when it is first displayed.

The end statement has the simple form:

`E`

A leaf is defined by a leaf statement of the form:

`L`*xtext*`|`*operationPanel*

where *x* is the authorization level for this operation, *text* is the leaf's name to be displayed in the hierarchy list, and *operationPanel* is the name of the operation panel to be activated by this leaf. The package name has to be provided in dotted notation. *operationPanel* is the suffix automatically added to the name.

The following is an example of the hierarchical tree definition as it appears in the specification file. The given text may be preceded by other textual information, pertinent to other program segments and will be ignored by the hierarchy tree. Additional text at the end, if starting with a section header of the form `#`*<sectionName>*, is ignored, too.

```
//**************************************************************
// Sample hierarchy tree definition
//**************************************************************
# hierarchy tree
// Bank context
  N0Bank                             | exploded
// Customer context
    N0Customer                       | exploded
      L0Create a Customer            | CreateCustomer
      L0Query a Customer             | RetrieveCustomer
      L0Remove a Customer            | DeleteCustomer
      L0Update Customer Information  | ChangeCustomer
// Customer's products context
      N0Products                     | exploded
// Customer Savings Accounts
        N0Savings Account            | exploded
          L0Query Savings Account    | QuerySavingsAccount
          L0Deposit                  | DepositSavingsAccount
          L0Withdraw                 | WithdrawSavingsAccount
          L0Transfer                 | TransferSavingsAccount
          L0Open a Savings Account   | OpenSavingsAccount
```

```
        E
// Customer Checking Accounts
        N0Checking Account            | exploded
          L0Query Savings Account     | QueryCheckAccount
          L0Deposit                   | DepositCheckAccount
          L0Withdraw                  | WithdrawCheckAccount
          L0Transfer                  | TransferCheckAccount
          L0Open a Checking Account   | OpenCheckAccount
        E
      E
    E
  E
// end of operation tree definition (may be followed by # ...)
```

## 4.4 Automatic Proxy Generation

In this section, we will discuss how you can make accessing Business Objects a lot easier and faster by creating Business Object Proxies automatically.

### 4.4.1 Introduction

To simplify client programming with Component Broker, we introduced the Business Object Proxy. The mission of this proxy is to hide Component Broker-specific details from the client developer. The client developer sees the attributes and the methods of the proxy, but doesn't need to bother about initializing the ORB, CORBA types, narrowing Managed Objects, and other Component Broker-specific things.

The proxy will mainly act as a wrapper to the Business Object. It holds a reference to the Business Object, and gets and sets its parameters, and passes methods to it.

However, creating a proxy for each Business Object is a tedious and repetitive chore that practically begs to be automatically generated. From the IDL specifications generated by Object Builder, it is possible to build 90 to 95 percent of the proxy accessing the Business Object. This has two big advantages. First of all, you don't need to write the same code *n* times for *n* Business Objects, and second, if you change the code for your proxy implementation, you can just regenerate your proxies to apply these changes to all the proxies at once.

In order to make a proxy generator, you need to read the Business Object information from the IDL specification. Each interface in the IDL file will correspond to a Business Object, which in turn will be the basis of a proxy.

From this specification, you may create a proxy that wraps each of the Business Objects based on method templates.

The main concepts behind a proxy generation module are:

- A Proxy Information Object that holds the information read in the IDL file
- An IDL parser that reads the information you need from the IDL file and puts it in the Proxy Information object
- A Proxy Factory that generates the proxy source code based on the Proxy Information object

## 4.4.2 The Proxy Information Object

When we parse an interface in the idl file, we need an object where we can put all the information we need in order to generate a proxy later on. We call this object a **ProxyInfo** object. This object contains the name of the interface, a collection of fields, or attributes of the interface, a collection of the methods that this interface implements, and a special collection of methods that access object relationships.

### 4.4.2.1 The ProxyField

To hold attribute information, we created a **ProxyField** object that has a name, a type, and an extra field. The extra field is used to hold additional information, like if this attribute is read-only or if it is a Business Object.

### 4.4.2.2 The ProxyMethodField

For methods, we extended the **ProxyField** to hold information about the parameters that this method takes as input, and which exception it may throw. The type field is used to store the method's return type. This field is called the `ProxyMethodField`.

## 4.4.3 The IDL Parser

The IDL parser creates a new **ProxyInfo** object for each defined interface, and fills it with the information specified for this interface.

In our Object Builder model, we defined a separate module for each object. This results in a separate IDL file for each object. We have a method that lists all the interfaces we want to read and goes through each file in turn. All exceptions are defined in a separate constructs file.

In each IDL file, an interface is defined with fields and methods. Below, we give a description of which strategy we used to extract the information of the file, without giving code details. The code may be read from the CD-ROM.

All manipulation of the file will be executed on a string representation of the file. The method that converts the file to a string is:

**String fileString = ProxyMgr.IdlParser.readFullyIdlFile(String fileNameAndPath);**

#### 4.4.3.1 Reading the Interfaces

Normally, this method would be called by a **ProxyFactory**, to know which interfaces are specified by this file. It returns a collection of interface names that can be iterated through to create a **ProxyInfo** object for each interface.

The interfaces start with the keyword `interface` and are listed separated by semicolons and a new line.

The method to get all interfaces is:

**Vector ProxyMgr.IdlParser.getInterfaces(String fileString);**

**Note:** Since we have the solution of one object per file, we have our interfaces listed in the **proxyMgr.ProxyFactory.getModuleList** method instead.

#### 4.4.3.2 Reading the Exceptions

After the interfaces have been defined, a definition of exceptions follows. Similar to reading the interface definitions, reading and generating the exceptions would be done by the **ProxyFactory** before the generation of the proxies begin.

Exceptions are defined by the keyword `exception`, followed by the exception name. The name is captured, and we find the next exception by looking for the next exception keyword:

**Vector ProxyMgr.IdlParser.getExceptions(String fileString);**

**Note:** Since we have the solution of one object per file, we have our exceptions listed in a separate constructs file, and the method to generate them is the **ProxyMgr.ProxyFactory.generateExceptions** method.

#### 4.4.3.3 Reading the Business Object Specification

We're now all set to do the actual work, namely to generate a proxy for each of the interfaces we've found. The **IdlParser** takes the interface name as input and returns a **ProxyInfo** object containing lists of the attributes and methods defined for the interface.

When attributes and methods are read, we need to make sure that we read them within the current interface scope, so that we don't mix these attributes up with attributes defined for another interface. An interface scope begins with the `interface` keyword followed by the interface name, and the scope ends with the keyword `end` followed by the `interface` keyword and interface name. We pass these positions on to the **readAttributes** and **readMethods** methods to make sure they operate within the correct attribute scope.

The method that reads the interface is as follows:

**ProxyInfo readIdlFile(String fileString, String interfaceName)**

#### *Reading the attributes*

Attributes are defined by the attribute keyword `attribute`, followed by the name of the attribute and the type, and are separated by semicolons. For each attribute read, a new **ProxyField** object is created and populated with the data, and added to the current **ProxyInfo** object's attributes list.

The attribute definitions are read from right to left. We know that the definition begins with the `attribute` keyword and ends with the a semicolon. The word right before the semicolon is the attribute name, then follows the type, delimited by blanks. If the `read-only` keyword precedes the `attribute` keyword, the field's extra field is set to read-only.

When the attribute definition has been read, the type undergoes some evaluation and conversion methods.

First, the type is looked up in a table of IDL types and their corresponding proxy types to return the correct type value for an untranslatable type. The type for a string is represented as string in the IDL file, and it needs to be converted to `String`. Similarly, `IManagedCollections::IIterator` is changed to `com.ibm.ICollectionsBase.IIterator`.

Second, certain naming rules are presumed when analyzing attribute types. All `::` notation in the IDL is converted to `.`; For example, `CBBank::TransactionRecord` is converted to `CBBank.TransactionRecord`. If the type begins with the Business Object package name, here `CBBank`, it is assumed that this should be a proxy type wrapping the Business Object, and not the Business Object itself. Only if the type ends with `Type` is it assumed that this is a defined enumerated type and not a proxy.

The method returns the last position of the attribute definitions, so that the next process, reading the methods, knows where to begin.

The method call to read attributes for an interface is as follows:

**int endAttributes = readAttributes(ProxyInfo proxyInfoObject, String fileString, int startInterface, int endInterface)**

#### *Reading the methods*

Methods don't have any keywords associated with them; so we trust that they begin where the attribute definition ended and that they are separated by semicolons. The methods are also read from right to left. Right before the semicolon, the exceptions are defined, followed by the keyword `raises`. Some methods don't raise exceptions. Then follows a list of parameters and their corresponding types, if any. Finally, the method name and the return type is defined.

For each method read, a new **ProxyMethodField** object is created and populated with the data in the interface, and added to the current **ProxyInfo** object's methods list.

Parsing the methods is a relatively complex task; so we divided it into three subtasks, namely parsing the exceptions, parsing the parameters, and parsing the name and return type.

#### *Parsing the exceptions*

The exceptions follow the keyword `raises` and are separated by commas. For each exception, a new **ProxyField** object is created and added to the exceptions list of the current method.

The exceptions have three naming aspects: the exception name given in the IDL file, the exception that the Business Object method raises, and finally the name that the proxy will raise for this exception. As an example, the method **findAccount(Object acccountKey)** raises an exception that an element with the given key does not exist. The exception is defined in the IDL file as `ICollectionsBase::IElementNotFound`. The exception thrown by the Business Object is named `com.ibm.ICollectionsBase.IElementNotFound`. Finally, the exception the client sees, the one thrown by the generated proxy, is `CBBankNotFoundException`. The exception string is looked up in a hand-coded exception table where a ProxyField is returned that holds all three exception names.

#### *Parsing the parameters*

Parameters are defined within two parentheses and are separated by commas. The IDL notation of in and out is ignored; so only the parameter name and type are used to populate the `ProxyField` parameter that is added to the method's parameter list. In the same way as the attributes, the types are validated and converted.

***Parsing the method info***

The method name is set for this method, and the return type is set. Finally, the method field is added to the methods list of the current **ProxyInfo** object.

If the method's return type is an iterator type, the method needs special treatment, and the extra field is set to `iterator`. The method is added to the iterator methods list. The type of object this method iterates upon is found by looking up what the return type of the method accessing the iterator returns. The type is put in the extra field of the method. If there are no methods accessing the iterator, a dummy type is set, which needs to be manually corrected.

The method that reads all the method definitions for an interface follows:

**readMethods(ProxyInfo proxyInfoObject, String fileString, int startMethods, int endInterface)**

### 4.4.4 The Proxy Factory

So, now that we have an object that holds all the necessary information about our Business Object, how do we generate the source code for a Proxy Object to access it?

The Proxy Object is described in Section 4.2. To generate such an object, we need to generate the class definition, with its import statements, variables and so on. Then we generate the "general" methods, such as constructors and the **create**, **update** and **delete** methods. After that, it's time to look at our **ProxyInfo** object, and generate getter and setter methods for all the attributes defined here. Finally, we generate methods to represent all the methods defined in the **ProxyInfo** object.

The main method to generate source code from a **ProxyInfo** object is the following:

**void ProxyMgr.ProxyFactory.generateProxy (ProxyInfo proxyInfoObject)**

This method takes the **ProxyInfo** object as input and generates the source file for a Proxy Object. This generated proxy wraps a Business Object of the type this **ProxyInfo** object describes.

The method is divided into submethods, where each method returns a string representing the code for a method in the generated Proxy Object. The particularities of each method are described below.

#### 4.4.4.1 Creating the Class Definition

The class definition needs to include the packages that this proxy depends upon. How the import statements should look depends on how the Object Builder model is defined. Nevertheless, given the model, it should be easy to automatically define which packages this proxy uses.

If the proxy has attributes that have another Business Object proxy as type, make sure to import the package for this Business Object.

The **Proxy** class has an abstract proxy class as a superclass. This is the class that contains the general, interface-independent methods for the proxies.

If an interface has an attribute which is another Business Object, for instance, it has a Proxy Object as its type. Accessing this object requires additional effort. The proxy can't access a Business Object directly; so the class definition must include an attribute that holds the reference to the Business Object proxy.

#### 4.4.4.2 Creating Getter and Setter Methods

For each attribute defined in the **ProxyInfo** attributes list, we create a getter and setter method to access the attribute. The getter method calls the Business Object's getter method for this attribute. The setter method, you guessed it, calls the Business Object's setter method for this attribute.

However, there is a catch; the setter methods are bound in order to update the user interface when an attribute has changed. To make an attribute bound, the **firePropertyChange** method is called with the new and old attribute values. The **firePropertyChange** method only takes objects as parameters. If the attributes are of a primitive type, they need to be converted to objects. For example, **int** is converted to **Integer**. To achieve this, the type is looked up in a type table to see whether the type is a primitive, and if so, what the corresponding object is for this primitive.

Also, if the attribute to be accessed is not a regular type but another Business Object proxy, it is the attributes defined in the class definition that are to be accessed.

```
String createGetSetMethods(ProxyInfo proxyInfoObject)
```

#### 4.4.4.3 Creating Methods

For each method defined in the **ProxyInfo** methods list, we create a method to access the corresponding Business Object method. Methods are defined with a name, a return type, parameters, and exceptions.

To build a proxy method accessing a Business Object method, we need to build the method declaration from the method return type, name, parameters and exceptions, as defined in the **ProxyMethodField** object. Then we build the catch statement, where each Business Object exception raised for this method is caught, and the corresponding proxy method is thrown.

The method string created consists of the method declaration, the corresponding method call to the Business Object, and a catch exception statement after the method call.

```
String ProxyMgr.ProxyFactory.createMethods(ProxyInfo proxyInfoObject)
```

#### 4.4.4.4 Creating Iteration Methods

Object relationships denoting a one-to-many relationship between Business Objects require special attention in our proxy, as described earlier in Section 4.2.3.1.

These are different from objects acting as attributes, since this is a one-to-one relationship, while object relationships implement a one-to-many relationship.

We need to create the extra methods hiding the Business Object iterator from the client developer and create an **Iterator** class that can be treated as a regular **Enumeration** class from the client side.

```
String ProxyMgr.ProxyFactory.createIterationMethods(ProxyInfo
proxyInfoObject)
```

#### 4.4.4.5 Creating Iteration Classes

The iteration methods refer to an iteration class, which implements an enumeration on the proxy class. This is the class that the main proxy has an object relationship with. For each method in the iteration methods list of the **ProxyInfo** object, we create an **Iteration** class for the type defined in the extra field of the method field. The **Iteration** class implements the same interface as the **Enumeration** class.

```
String ProxyMgr.ProxyFactory.createIteratorClasses(ProxyInfo
proxyInfoObject)
```

#### 4.4.4.6 Creating General Methods

For the remaining methods, such as **create**, **delete**, **update**, and the like, there is a method defined in the **ProxyFactory** for each of these methods that returns a string which is the source code for this method. These methods are mostly a copy of the method as defined in the template Proxy Object, with all

references to names and types replaced by the information stored in the **ProxyInfo** object.

Some methods, such as **init** and **update**, go through all the attributes of the Proxy Object and set or get a value. In this case, the attributes list in the **ProxyInfo** object is iterated through to generate a code statement.

If the object has attributes that have another Business Object proxy as their type, these attributes get special attention in some methods. They need to be created and initialized in the **create** method. In addition, a **read** method for each attribute must be defined, and these **read** methods are called by the main **read** method.

### 4.4.5 Putting It All Together

We have a set of IDL files in **ProxyMgr.ProxyFactoy.getSourceDirectory()**, which contain the interfaces specified by **ProxyMgr.IdlParser.getModuleList**. The following method will generate all the source code for the interfaces in the *ProxyMgr.ProxyFactory.getTargetDirectory*:

```
void ProxyMgr.ProxyFactory.generateAllProxies()
```

The method will generate source code for the exceptions in the constructs file, generate a **ProxyInfo** object for each interface, and create source code for each **ProxyInfo** object.

When the source code is generated, it may be imported into the developing too to see if there are any unresolved problems. To avoid too many problems, make sure that the rest of the model is imported, such as the Business Objects, enumerated types and so on. Also, make sure the abstract superclass for the proxy is loaded.

The most common problem that will arise is that there are erroneous references to packages, and some typing problems may appear. It is important to go through the code, and set the correct initial values in the **init** method of the proxy, see that the correct exceptions are thrown, and that the type names are correct. You may need to update the exception table and type conversion table.

It is important to keep in mind that a proxy generation tool is useful to generate 90 to 95 percent of your code that otherwise would have to be hand-coded, but human intervention is necessary for the last validation.

### 4.4.6 Four Steps to Creating Your Own Proxy Generator

The proxy generator presented here may be reused, but most probably you'll need to modify it to fit your project's development style. This section describes the things you should consider when building a proxy generator.

#### 4.4.6.1 Create a Template Proxy

First of all, you need to create the template proxy, or a Proxy Object for a given Business Object, that implements the methods you need for all your proxies. This proxy should inherit from an abstract proxy class, where all the common, interface-independent methods are defined. For example, if you have a Business Object Customer, you create a proxy **CBBank.CustomerProxy** that inherits from **CBBankProxy**. When this proxy behaves like you want it to, you can use this proxy as a generic template for all the methods you define for an automatically generated proxy.

#### 4.4.6.2 Create a ProxyInfo Object

The **ProxyInfo** object acts as a bridge form the IDL file to the generated source code. You may find that the **ProxyInfo** object presented in this book is sufficient to hold all the information you need, or you may want to modify it if your model requires it.

#### 4.4.6.3 Create the IDL Parser

The IDL parser presented here is tallor-made for the code generated by the model we have used and is not very robust. Other models may introduce additional considerations in order to obtain all the information correctly.

We defined that every object is defined in separate modules. If you choose to have one module for all objects, and thereby in one IDL file, the file needs to be parsed differently. Only the task of reading the files will change because parsing each interface will be the same.

There are other and more robust ways to obtain information about Business Objects than parsing the IDL file. If the objects have been installed on the Component Broker server, they may be read by accessing the Interface Repository. Following naming standards to separate attribute getters and setters from the other methods, you may, by reflection methods, access the Business Objects' attributes and methods.

#### 4.4.6.4 Create the Proxy Factory

Taking the template proxy's source code, you must create methods to generate this source code for a given **ProxyInfo** object. The simplest way to do this is to copy the code of your proxy template object into a method that generates source code for this method. In this method, replace all specific

references to the template proxy, and its Business Object reference, with generic ones retrieved from the **ProxyInfo** object.

When this work is finished, you may verify if the methods are generated correctly by loading the generated code for the template proxy's IDL file and comparing the template proxy with the newly generated proxy. The next step is to generate proxies for different IDL files, and most probably, you'll have to go back to take out some forgotten references to your template proxy!

If there are places where it is impossible to generate correct code, such as setting a default value for an enumeration type, it is a good solution to generate code you know will provoke an unresolved problem. In this way, you will be forced to correct the error before you publish the proxy.

# Chapter 5. Middle Tier

Having explained the client application development approach, we now describe the middle tier of the CBConnector Bank.

## 5.1 Introduction

This chapter shows how we implemented CBConnector Bank objects with the Object Builder Component Broker Toolkit. During the residency that produced this redbook, we started with a transient implementation and then moved to the persistent implementation. However, in this chapter, we present only the persistent implementation.

## 5.2 Architecture of the Middle Tier

The core of CBConnector lies in its middle-tier application server. The architecture of the CBConnector middle tier is presented in *IBM CBConnector Cookbook Collection: First Steps,* SG24-2033. Please consult that redbook for complete details. This section is a quick summary of the middle tier.

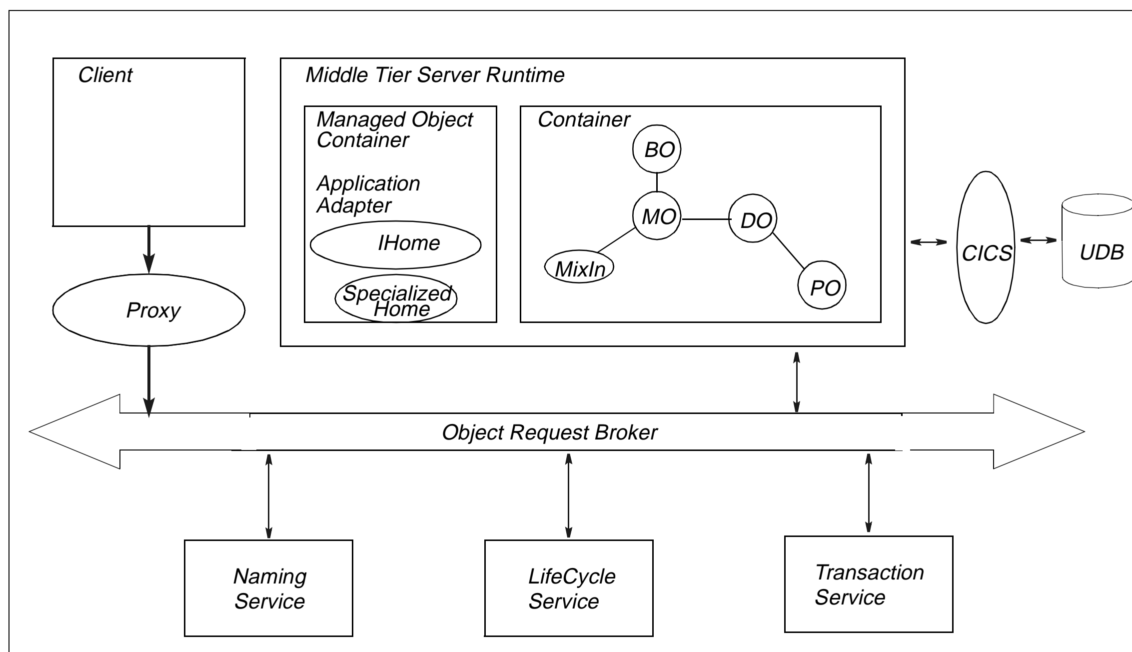



*Figure 23. Global Architecture*

© Copyright IBM Corp. 1998

## 5.3 Persistent Implementation

This section defines the terminology we used in the rest of the document.

### 5.3.1 Managed Object Container

The managed object container consists of application adaptors and the Managed Object framework. It manages the container that holds the business logic.

#### 5.3.1.1 Generic Home Objects

These are the factory and collection homes. They provide generic lifecycle functionality for objects. They are also used to implement the query facility as well as iteration over the set of objects they contain.

#### 5.3.1.2 Specialized Home Objects

The specialized home allows you to customize the LifeCycle functions. They also allow the client to have additional methods.

### 5.3.2 Containers

The container controls the processing and memory management. It contains the Business Object (BO), Data Objects (DO) and the Persistent Object (PO).

#### 5.3.2.1 Business Objects (BO)

The Business Object (BO) contains the actual business logic of the server. It contains the signature and the implementation of methods.

#### 5.3.2.2 Data Objects (DO)

The Data Object (DO) contains the logic to store, update, retrieve, and delete objects from UDB (Universal Database DB2). It isolates the BO from specific back-end database and transaction monitors. It collaborates with the application adaptor to implement the BO methods.

#### 5.3.2.3 MixIn

The main role of the MixIn Object is to synchronize method calls on DOs so that invocations are thread safe. It also provides various mechanisms, such as policy management of DOs or refreshment of DO data, when a transaction is rolled back.

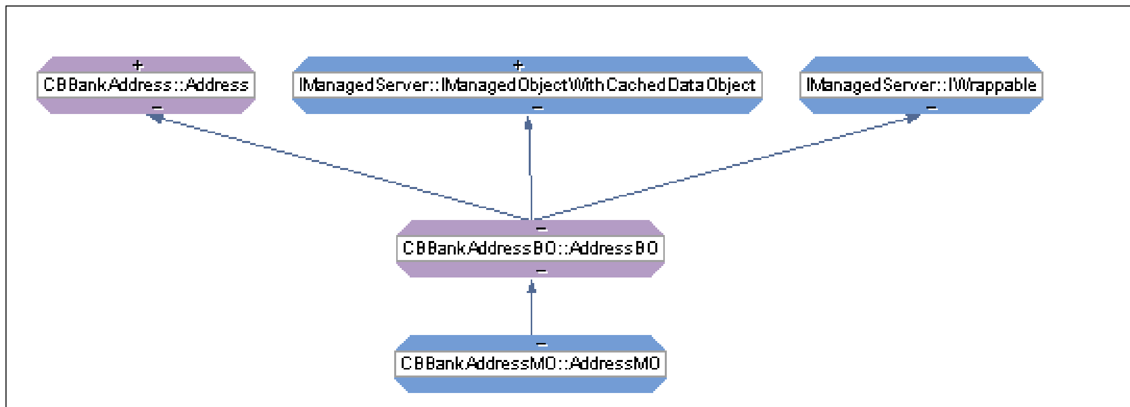



*Figure 24. Address Hierarchy*

#### 5.3.2.4 Persistent Objects (PO)

The Persistent Object (PO) encapsulates the embedded SQL statements needed to access UDB. It allows insertion, update, deletion, and retrieval of persistent data. It provides communication with UDB.

#### 5.3.2.5 Managed Objects (MO)

The Managed Objects (MO) wrap the DO and BO. They contain the set of interfaces (CORBA IDL), implementations, and conventions that must be followed in order to create and use business objects in Component Broker.

## 5.4 CBConnector Bank Architecture

This section describes the architecture of CBBank. It includes the CBConnector functions we used and the description of objects.

### 5.4.1 Functions Used in the CBConnector Bank Sample

The following features of CBConnector were exploited in the CBConnector Bank samples.

- Specialized Home.
- Iterator over generic and Specialized Home.
- PAA: Procedural Application Adaptor. A PAA provides a context for Managed Objects. We use the DB2 application adaptor that is a BOIM (Business Object Application Adaptor) framework with appropriate configurations and additional mechanisms that support DB2 access with caching.
- Transaction (over CICS and the Object Transaction Service (OTS)).

- Session: PAA uses the notion of session to help manage resources within the context of a unit-of-activity scope.
- Transient Objects (objects that lack a persistent store).
- Persistent Objects (backed with UDB).
- CICS and UDB back-end. Transactions are run through two databases. UDB(NT) on one and UDB with CICS on MVS on the other hand.
- Query Service: The language for the Component Broker Query Service is Object Oriented-Structured Query Language (OO-SQL). OO-SQL is used in the Policy Manager.

### 5.4.2 CBConnector Banking Objects

The 16 objects we created are listed in Figure 25 on page 71. The graphic shows the graphical interface of Object Builder. We created two types of objects:

- Default (Normal) Object: This object is a Managed Object. It contains all the methods associated with an object. The delete, create, and retrieval of the object are provided by the associated default home.
- Specialized Home Object: The specialized home is also a Managed Object. It provides a richer interface to homes than what is provided by the default. Its functionality is limited. In this book we mainly use it to find, create and delete Managed Objects but it could be used to create your own interface to manipulate the Home Object.

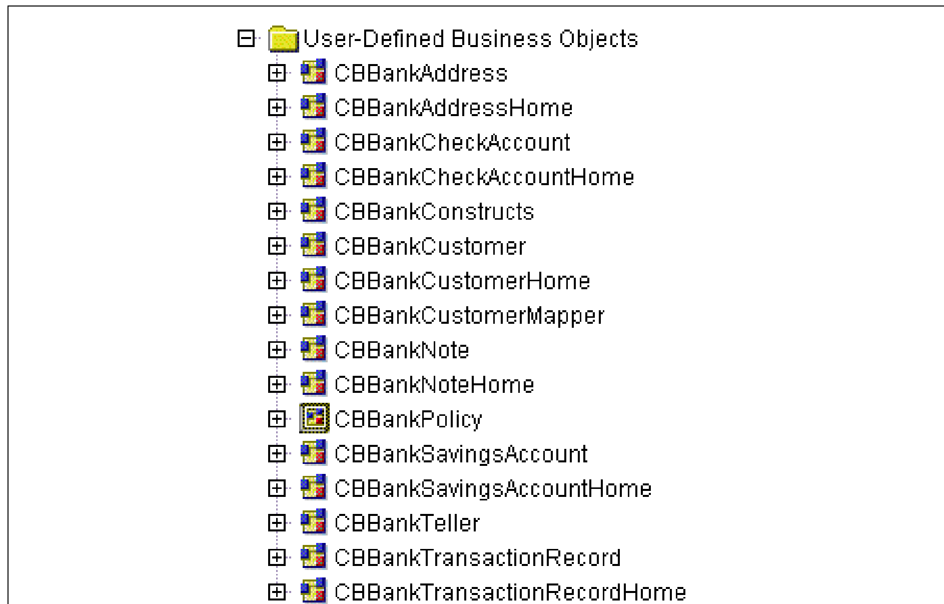



*Figure 25. CBConnector Bank Objects*

The model and the objects involved in the model were presented in Chapter 2, “Design” on page 11. The CBConnector Bank **Customer** object is the core object of the architecture in Figure 26 on page 72.

Relations between objects are iterators. The **Customer** object has iterators over **CheckAccount**, **SavingsAccount**, and **Notes**. **CheckAccount** and **SavingsAccount**, in turn, have iterators over **TransactionRecord**. A **TransactionRecord** is appended to **Check/Savings Account** each time a method on the object is invoked.

The **Teller** creates **Notes** for a **Customer** and queries on the list of **Check/Savings Account**, **Notes**, and **TransactionRecord** belonging to a **Customer**.

All objects can query (**get/set**) the Policy Manager. We mainly use it to store default and dynamic values, such as the name of a server or **default_interest_rate** for `Customer`s.

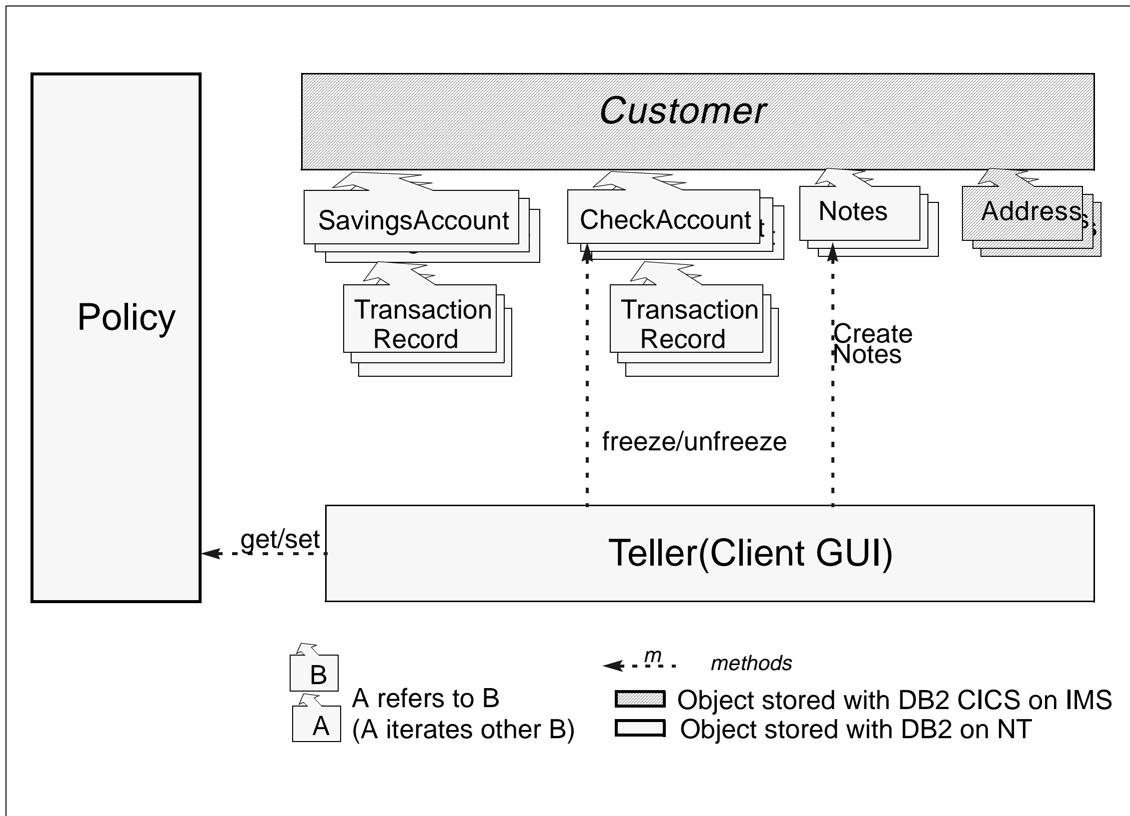



*Figure 26. Architecture*

The rest of this chapter describes, one by one, all CBConnector Bank objects. The **CBBankAddress** and **CBBankAddressHome** objects are among the first objects that we describe. Thus we will detail the **Address** objects by showing a few lines of code. Other objects are created in the same fashion, but we will not go into details.

### 5.4.3 CBConnector Bank Scenario

This section describes the different steps we have to undertake to get the whole CBConnector Bank working. We describe it as a scenario.

The middle Tier is driven by the GUI client front end. The client GUI is a teller that allows you to create your own Banking scenario. The GUI is described in details in 4.3, “The Client Front-End (Graphical User Interface)” on page 44. All objects of CBConnector Bank and their methods can be instantiated and called by the teller as the menus in Figure 27 on page 73 show.

The instantiation of the Policy Manager is out of the scope of the GUI; we assumed that it is running together with the CBConnector Bank server.

The typical scenario that a teller undertakes is as follows

1. Create a **Customer** (see Figure 27 on page 73).
2. Create a **Checking** or **Savings Account** for a **Customer**.
3. Record a **Note** for a **Customer**.
4. Transfer (Debit/Credit) an amount of money in a transaction from one account to another. A **TransactionRecord** is associated with each debited and credited account.
5. A **Customer** can list the **Checking/Savings Accounts** or **Note** he/she owns.

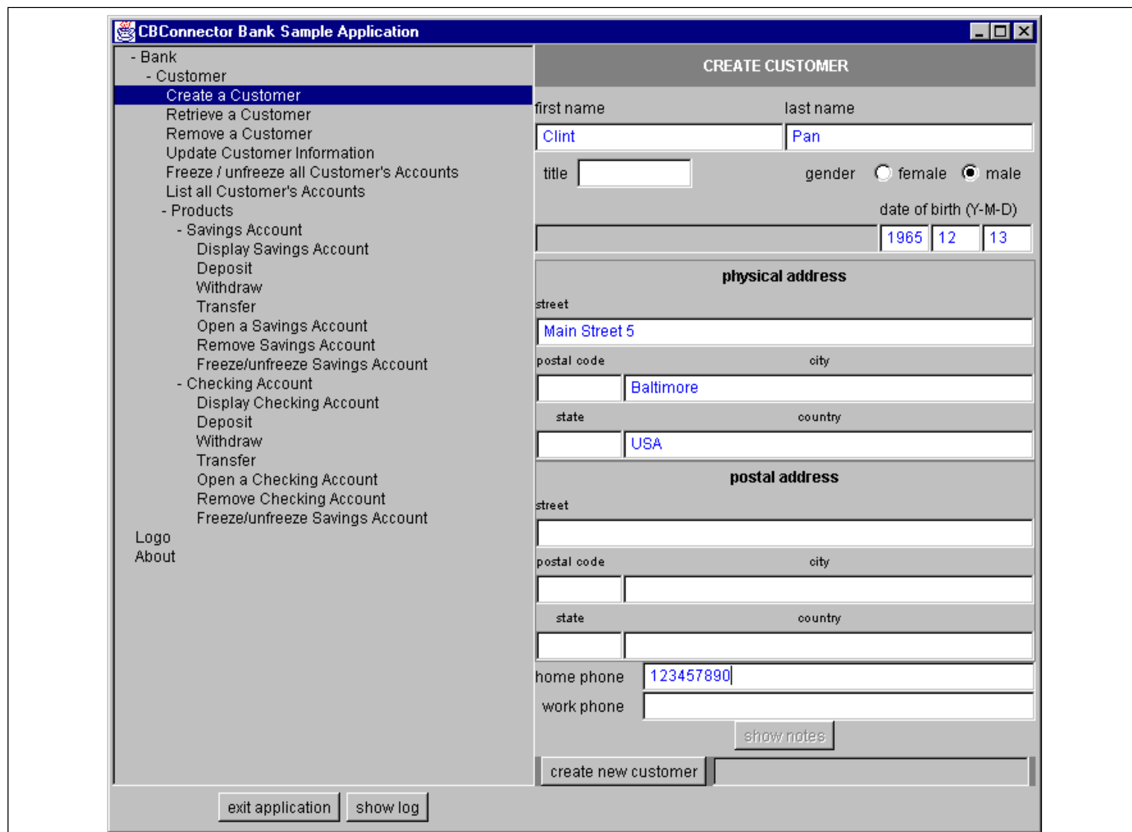



Figure 27. Teller GUI

### 5.4.4 CBConnector Bank Configuration

This section describes the different configurations we have made during the implementation phase with CBC Object Builder.

#### 5.4.4.1 Business Objects

Buisness Object, Data Object and Persistent Objects were built with the following approaches:

- Most of the methods are written in Java, except the method for specialized Home Objects that had to be written in C++ (as Java support for specialized was not available). The code for methods are stored in external files to ease software evolution and group programming. It also allows separate testing of methods.
- All objects are associated with home objects except the **Teller**. Home objects are unique and allow us to control the creation of the object as well as methods to find them.
- All object are queryble except the **Customer**.
- All object interfaces are defined in a separate file and separate module (see Figure 31 on page 80) so that we have clean name-scoping, and each object can be tested and compiled separately.
- The Pattern for Handling Data State is Delegating for all the objects.

#### 5.4.4.2 Persistent and Data Objects

This section gives some details about the CICS and Procedural Application Adaptor. The specification of the PO and DO are straightforward. The DOImpl contains the mapping between the PO and the DO. The mapping is based on the (UDB) database schema defined in the back-end.

#### 5.4.4.3 Importing Beans (CICS/Procedural Adaptor)

When you start working with Object Builder, you must include the *.jar* file created with CICON to your CLASSPATH. The easiest and cleanest way to do it is to copy *ob.bat*, and modify this newly created batch file to include a *.jar* file to the CLASSPATH. Thus, the *.jar* file is always in the CLASSPATH whenever you start Object Builder. Do not forget to enable Object Builder beta capabilities.

After opening the project you must import the bean (**User defined PAO schemas -> Import bean**). The Object Builder asks you for the name of the class or let's you select a class from the *.jar* file. This class or *.jar* file must be in the CLASSPATH. See the *CICS & IMS Application Adaptor Quick Beginnings* guide for more details.

#### 5.4.4.4 Implementing the Data Object

During the configuration of the DO, make sure that you select the correct options when implementing the DO. The options are

- BOIM with any key (Environment)
- Procedural Adaptors (Form of persistent behavior)
- Delegating (Data access pattern)
- Default (Handle for storing pointers)

These options make DO to inherit from **IPAAExtLocalToServer::IDataObject**, which delegates all method calls to the PAO.

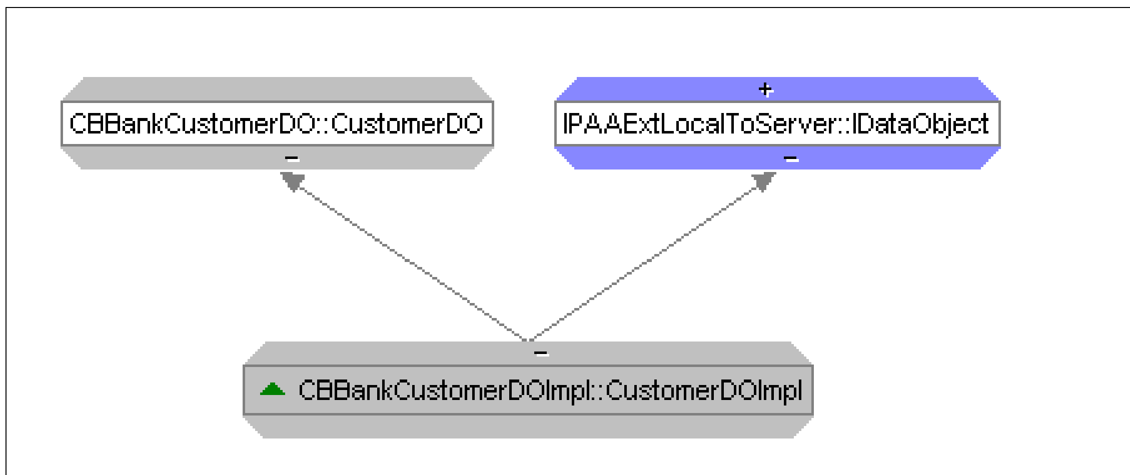



*Figure 28. Data Object Inheritance*

#### 5.4.4.5 Using Mapping Helpers

If the target PAO attributes happen to differ from your BO interface attributes (for instance, you are aggregating objects), then you must use mapping helpers to resolve this problem. The mapping helper is a C++ class containing only methods, because this class is never instantiated by the DO implementation.

By using mapping helpers, you can do whatever mapping is required between DO implementation and PAO. Objects are always passed in methods in general form, such as a handle string. In Object Builder, you can define the name of class which does the mapping and names of methods for mapping the data in both directions.

While you are doing the bindings between DO and PAO, Object Builder automatically asks you to enter the name of mapping helper class if you have incompatible bindings (such as enumeration to string).

The following example shows a mapping helper, which maps enumeration `GenderType` to `String` in the PAO and vice versa.

```
void GenderMapper::PAOToEnum( const CORBA::String& strGender,
CBBankConstructs::GenderType& gender )
{
    if( *strGender == 'F' )
      gender = CBBankConstructs::FEMALE;
    else
      gender = CBBankConstructs::MALE;
}
void GenderMapper::enumToPAO( const CBBankConstructs::GenderType&
gender,
      CORBA::String& strGender )
{
if( gender == CBBankConstructs::FEMALE )
     strGender = CORBA::string_dup( "F" );
else
     strGender = CORBA::string_dup( "M" );
}
```

The mapping gets more complex when you need to map imbedded objects to CICS. Our sample implements mapping helper to move **Address** objects back and forth to CICS while **Customer** is being stored/retrieved. If your object contains more than one attribute, you need to map the given object to many PAO attributes and define the name of the mapping helper class and mapping methods.

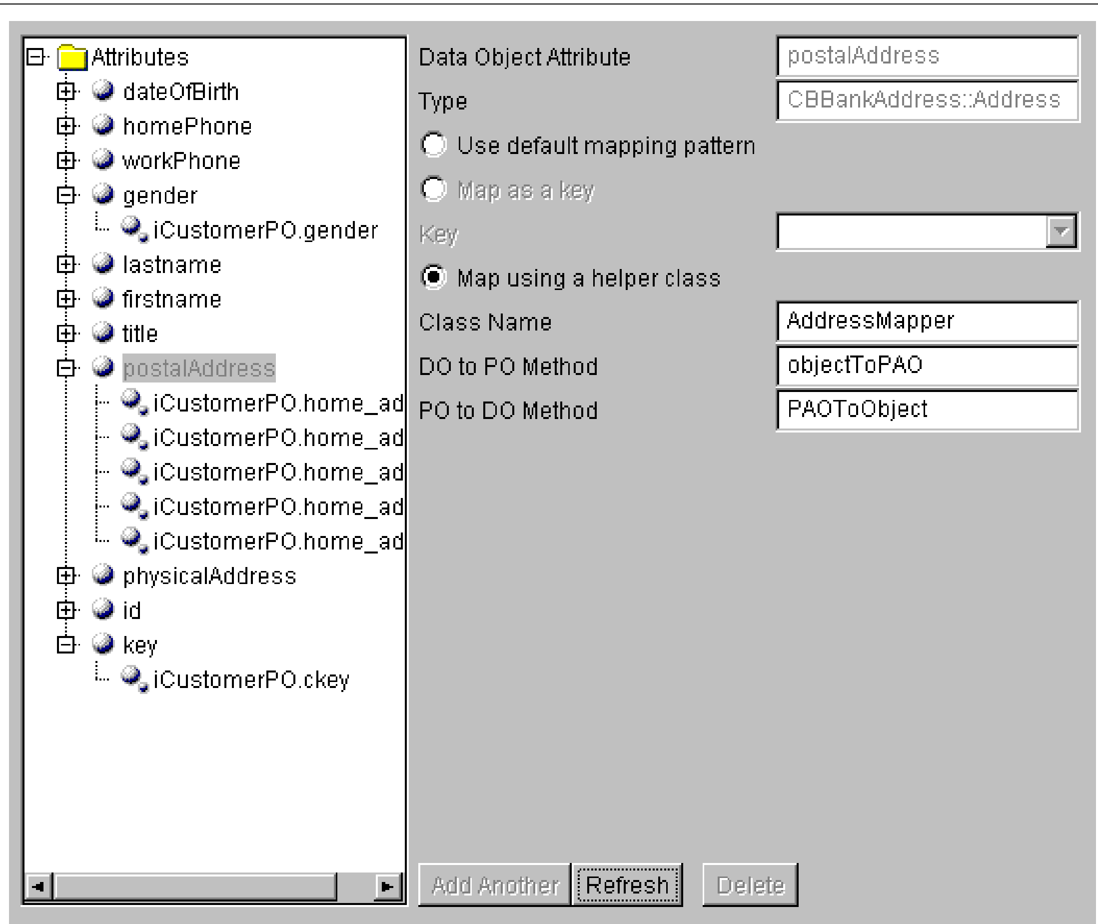



*Figure 29. Persistent Objects*

#### 5.4.4.6 Packaging

The packaging concerns the way the server is handles. The following configuration was adopted:

- Each object is managed in a separate container. Each container handles policies (termination, memory management, transaction) and the caching mechanism for the Managed Object. If the caller is outside a transaction, a transaction is automatically created; otherwise a transaction is not created.
- If an exception occurs within an active transaction, the transaction is rolled back. If the call is outside a transaction, the call is abandoned.

### 5.4.5 Constructs

The following table shows the functions of **CBBankConstructs**.

*Table 3. Role of Note Objects*

| Name | Function Performed | Remarks |
|---|---|---|
| CBBankConstructs | Contains the definitions, exceptions, and enumerations used in CBConnector Bank | CBBankConstruct is a module in the Object Builder terminology. It is not an object. |

#### 5.4.5.1 CBBankConstructs

All exceptions and enumerations of the CBConnector Bank Objects are declared in **CBBankConstructs**. This module does not have any interfaces and thus methods. The list of **CBBankConstructs** enumerations and exceptions is shown in Figure 31 on page 80. This approach implies that each object that uses these constructs must scope it with the name of the CBConnector Bank module.

```
module CBBankConstructs {

  exception Frozen {
  }; // end exception  Frozen
  exception Unfrozen {
  }; // end exception  Unfrozen
  exception Closed {
  }; // end exception  Closed
  exception NegativeAmount {
  }; // end exception  NegativeAmount
  exception BeyondDebitLimit {
  }; // end exception  BeyondDebitLimit
  exception ProcessingFailed {
    string details;
  }; // end exception  ProcessingFailed
  enum StatusType {
    FROZENSTATUS,
    OPENSTATUS,
    CLOSEDSTATUS
  }; // end enum StatusType
  enum TransactionType {
    CREDIT,
    DEBIT
 }; // end enum TransactionType
  enum GenderType {
    MALE,
    FEMALE
  }; // end enum GenderType
```

**IDL**

- Constructs
  - Frozen
    - Members
  - Unfrozen
    - Members
  - Closed
    - Members
  - NegativeAmount
    - Members
  - BeyondDebitLimit
    - Members
  - ProcessingFailed
    - Members
      - details
  - StatusType
    - Members
      - FROZENSTATUS
      - OPENSTATUS
      - CLOSEDSTATUS
  - TransactionType
    - Members
      - CREDIT
      - DEBIT
  - GenderType
    - Members
      - MALE
      - FEMALE
  - CustomerNotFound
    - Members

**Object Builder Graphical Interface**

*Figure 30. CBBankConstructs IDL and Graphics*

### 5.4.6 Address

**CBBankAddress** and **CBBankAddressHome** are modules related to Address (see Figure 24 on page 69). **CBBankAddress** has all the persistent attributes, no methods and BO, MO, DO. **CBBankAddressHome** has only a

BO and MO and one method that creates and returns a unique Primary Key. The functionality of each **Address** object is described in Table 4 on page 80.

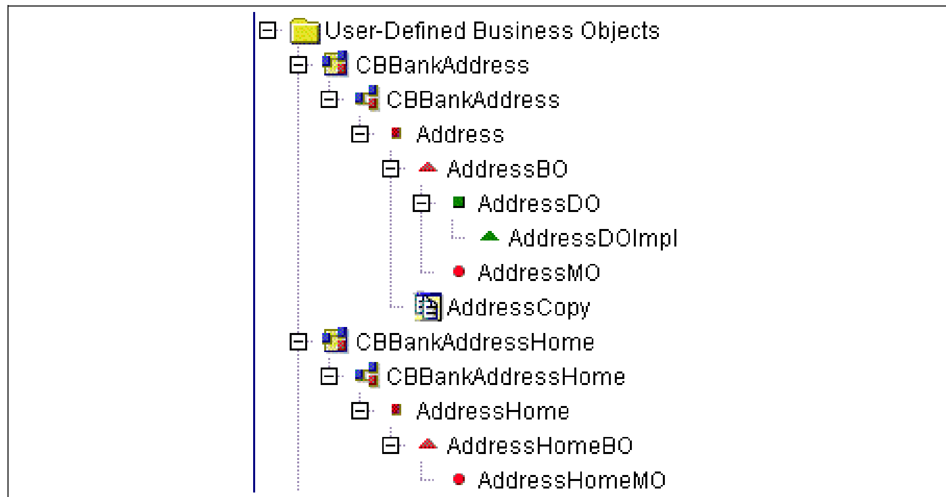



*Figure 31. CBBankAddress and CBBankAddressHome*

The following table shows the functionality of **CBBankAddress** and **CBBankAddressHome**.

*Table 4. Role of Address Objects*

| Object Name | Function Performed | Remarks |
|---|---|---|
| CBBankAddress | Stores persistent Address attributes (street, city....) | see 5.4.6.1, "CBBankAddress" on page 80, for the IDL description |
| CBBankAddressHome | creates a CBBankAddress object and its Primary Key. (see Figure 34 on page 82 for the code) | see 5.4.6.2, "CBBankAddressHome" on page 81, for the IDL description |

### 5.4.6.1 CBBankAddress

The IDL for **CBBankAddress** is:

```
module CBBankAddress {
interface Address : IManagedClient::IManageable
  {
    attribute string street;
    attribute string city;
    attribute string state;
    attribute string postcode;
    attribute string country;
  }; // end interface Address
}; // end of module  CBBankAddress
```

*Figure 32. CBBankAddress IDL*

#### 5.4.6.2 CBBankAddressHome

The IDL of the **CBBankAddressHome** is:

```
module CBBankAddressHome {
interface AddressHome : IManagedClient::IHome
  {
    CBBankAddress::Address createAddress(in ::ByteString copyString )
raises (IManagedClient::IDuplicateKey,IManagedClient::IInvalidKey,
IManagedClient::IInvalidCopy,CBBankConstructs::ProcessingFailed);
  }; // end interface AddressHome
}; // end of module  CBBankAddressHome
```

*Figure 33. CBBankAddressHome IDL*

The **createAddress** method creates a **CBBankAddress** object from the `copyString` parameter. The code is shown in Figure 34 on page 82.

```
CBBankAddress::Address createAddress(in ::ByteString copyString)
{...
     // Create an addressCopy
    addressCopy = CBBankAddressCopy::AddressCopy::_create();
    addressCopy->fromString(copyString);

     // generate a Primary Key
    keyGenerator = IBOIMExtLocal::IUUIDPrimaryKey::_create( );
    keyGenerator->generate( );

     // create an address from a Primarykey
    object = createFromPrimaryKeyString( *keyGenerator->toString( ) );
    address = CBBankAddress::Address::_narrow(object);

     // Fill the new address object from addressCopy
    address->street( addressCopy->street( ) );
    address->city( addressCopy->city( ) );
    address->postcode( addressCopy->postcode( ) );
    address->state( addressCopy->state( ) );
    address->country( addressCopy->country( ) );
return address;
...
}
```

*Figure 34. CBBankAddressHome createAddress*

### 5.4.7 Customer

We created three types of objects related to **Customer**. They are **CBBankCustomer**, **CBBankCustomerHome**, and **CBBankCustomerMapper**.

A **CBBankCustomer** object is created through the **CBBankCustomerHome**. The **CBBankCustomer** creates a Primary Key associated with the new **CBBankCustomer** object. All attributes of the **CBBankCustomer** are made persistent through CICS. Thus, a key is needed to query on customers. CICS allows you to query on **Customer** if and only if you know the key. Therefore, you have to memorize the key of the object when it is created. This is too constraining for the client. We created a **CBBankCustomerMapper** to map a

Primary Key with a `lastname` or `firstname` back-ended with UDB so that the client can find a **Customer** by name.

*Table 5. Role of Customer Objects*

| Object Name | Function Performed | Remarks |
|---|---|---|
| CBBankCustomer | -Store attributes(firstname,....);<br>-Relation to listSavingsAccounts and listcheckAccounts<br>-DO, PO (mapping to UDB schema) | UDB-CICS backend |
| CBBankCustomerHome | -create CBBankCustomer and generates unique key from a string that the client pass in.<br><br>- <delete, update, find> Customer | -createCustomer call CBBankCustomerWrapper to create a tuple <key, firstname, lastname> so that the client can retrieve a customer by name. |
| CBBankCustomerMapper | create <key, firstname, lastname> tuple<br>This object is mapper between key and <firstname, lastname>. | UDB backend |

#### 5.4.7.1 CBBankCustomer

**CBBankCustomer** makes all attributes persistent that are related to **Customer**. It has iterators for **SavingsAccount**, **CheckingAccount**, and **Note**.

```
module CBBankCustomer {
  interface Customer : IManagedClient::IManageable
  {
    attribute string dateOfBirth;
    attribute string homePhone;
    attribute string workPhone;
    attribute ::CBBankConstructs::GenderType gender;
    attribute string lastname;
    attribute string firstname;
    attribute string title;
    attribute CBBankAddress::Address postalAddress;
    attribute CBBankAddress::Address physicalAddress;
    attribute string id;
    attribute string key;
    IManagedCollections::IIterator listsavingsAccounts( );
    IManagedCollections::IIterator listcheckAccounts( );
    ::IManagedCollections::IIterator listnotes( );
  }; // end interface Customer
}; // end of module  CBBankCustomer
```

*Figure 35. CBBankCustomer IDL*

#### 5.4.7.2 CBBankCustomerHome

**CBBankCustomerHome** is a specialized home that contains the methods to create **CBBankCustomer**, **findCustomer**, **updateCustomer**, and **deleteCustomer**.The IDL is described in Figure 36 on page 85. **findcustomer** requires a `firstname` or `lastname`, and the object reference is returned. The **CBBankMapper** object is contacted to retrieve a key by name.

```
module CBBankCustomerHome {
interface CustomerHome : IManagedClient::IHome
  {
    CBBankCustomer::Customer createCustomer(in ::ByteString copyString
) raises (IManagedClient::IDuplicateKey,IManagedClient::IInvalidKey,
CBBankConstructs::ProcessingFailed);
    CBBankCustomer::Customer findCustomer(in ::ByteString
primaryKeyString ) raises
(IManagedClient::IInvalidKey,IManagedClient::INoObjectWKey,
CBBankConstructs::ProcessingFailed);
    void updateCustomer(in CBBankCustomer::Customer customer,
in::ByteString copyString ) raises
(CBBankConstructs::ProcessingFailed);
    void deleteCustomer(in CBBankCustomer::Customer customer ) raises
(CBBankConstructs::ProcessingFailed);
  }; // end interface CustomerHome
}; // end of module  CBBankCustomerHome
```

*Figure 36. CBBankCustomerHome IDL*

#### 5.4.7.3 CBBankCustomerMapper

This object maps a **CBBankCustomer** Primary Key to a customer's `lastname` and `firstname`. This object was created to find a customer by name.

```
module CBBankCustomerMapper {
interface CustomerMapper : IManagedClient::IManageable
  {
    attribute string customerKey;
    attribute string firstName;
    attribute string lastName;
  }; // end interface CustomerMapper
}; // end of module  CBBankCustomerMapper
```

*Figure 37. CustomerMapper IDL*

### 5.4.8 Teller

A teller is implicitly created when the GUI interface is launched. The user has to input a login name and a password to be a **Teller** .

*Table 6. Role of Customer Objects*

| Object Name | Function Performed | Remarks |
|---|---|---|
| CBBankTeller | -Store attributes (firstname,....);<br>-DO, PO (mapping to UDB schema) | The Teller role is represented as a GUI interface. |

#### 5.4.8.1 CBBankTeller

The following IDL shows the attributes of **Teller**.

```
module CBBankTeller {
interface Teller : IManagedClient::IManageable
  {
    attribute string position;
    attribute string lastname;
    attribute string firstname;
    attribute string title;
    attribute string id;
    attribute string password;
  }; // end interface Teller
}; // end of module  CBBankTeller
```

*Figure 38. CBBankTeller IDL*

### 5.4.9 Note

Notes are sent by the **Teller** to a **Customer**. We have two objects: **CBBankNote** and **CBBankNoteHome** associated with **Note**.

*Table 7. Role of Note Objects*

| Object Name | Function Performed | Remarks |
|---|---|---|
| CBBankNote | Store persistently all attributes of Note. | Store in UDB NT |
| CBBankNoteHome | Create a CBBankNote with a unique key. | |

#### 5.4.9.1 CBBankNote

The following IDL shows the attributes of **CBBankNote**.

```
module CBBankNote {
  interface Note : IManagedClient::IManageable
  {
    attribute string customerKey;
    attribute string text;
    attribute CBBankTeller::Teller creator;
    attribute string dateCreated;
    attribute string timeCreated;
    attribute string key;
  }; // end interface Note
}; // end of module  CBBankNote
```

*Figure 39. CBBankNote IDL*

#### 5.4.9.2 CBBankNoteHome

The following figure shows the **createNote** method of **CBBankNoteHome**.

```
module CBBankNoteHome {
interface NoteHome : IManagedAdvancedClient::IQueryableIterableHome
{
    CBBankNote::Note createNote(in ::ByteString copyString ) raises
(IManagedClient::IDuplicateKey,IManagedClient::IInvalidKey,
IManagedClient::IInvalidCopy,CBBankConstructs::ProcessingFailed);
  }; // end interface NoteHome
}; // end of module  CBBankNoteHome
```

*Figure 40. CBBankNoteHome IDL*

### 5.4.10 CBBankCheckAccount

**CheckAccount** and **SavingsAccount** are implemented in a very similar way. They both have a specialized home.

*Table 8. Role of Note Objects*

| Object Name | Function Performed | Remarks |
|---|---|---|
| CBBankCheckAccount | -Make persistent attributes such as balance, interest_rate ...<br>-credit/debit<br>-freeze, close, open account<br>- get the interest_rate from the Policy Manager | Backed with UDB on NT. |
| CBBankCheckHome | - Create a CheckingAccount<br>- Transfer amount | |

#### 5.4.10.1 CBBankCheckAccount

The following figure shows the methods and attributes of **CBBankCheckAccount**.

```
module CBBankCheckAccount {
  interface CheckAccount : IManagedClient::IManageable
  {
    attribute string customerKey;
    attribute string dateCreated;
    attribute string name;
    attribute ::CBBankConstructs::StatusType status;
    attribute double balance;
    attribute float creditInterestRate;
    attribute float debitInterestRate;
    attribute double serviceFee;
    attribute double overDraftLimit;
    attribute string key;
    void freeze( ) raises
(CBBankConstructs::Frozen,CBBankConstructs::Closed);
    void unfreeze( ) raises
(CBBankConstructs::Unfrozen,CBBankConstructs::Closed);
    void credit(in double amount ) raises
(CBBankConstructs::NegativeAmount,CBBankConstructs::Frozen,
CBBankConstructs::Closed,CBBankConstructs::ProcessingFailed);
    void debit(in double amount ) raises
(CBBankConstructs::NegativeAmount,CBBankConstructs::BeyondDebitLimit,
CBBankConstructs::Frozen,CBBankConstructs::Closed,
CBBankConstructs::ProcessingFailed);
    void requestCheckBook(in unsigned short Number ) raises
(CBBankConstructs::Frozen,CBBankConstructs::Closed,
CBBankConstructs::ProcessingFailed);
    ::IManagedCollections::IIterator listTransactions( );
  }; // end interface CheckAccount
}; // end of module  CBBankCheckAccount
```

*Figure 41. CBBankCheckAccount IDL*

#### 5.4.10.2 CBBankCheckAccountHome

The following IDL shows the methods of **CBBankCheckAccountHome**.

```
module CBBankCheckAccountHome {
interface CheckAccountHome :
IManagedAdvancedClient::IQueryableIterableHome
  {
    CBBankCheckAccount::CheckAccount createCheckAccount
(in ::ByteString copyString ) raises
(IManagedClient::IDuplicateKey,IManagedClient::IInvalidKey,
IManagedClient::IInvalidCopy,CBBankConstructs::ProcessingFailed);
    void transfer(in CBBankCheckAccount::CheckAccount fromAccount,
in CBBankCheckAccount::CheckAccount toAccount,in double amount )
raises
(CBBankConstructs::Frozen,CBBankConstructs::Closed,
CBBankConstructs::NegativeAmount,CBBankConstructs::BeyondDebitLimit,
CBBankConstructs::ProcessingFailed);
  }; // end interface CheckAccountHome
}; // end of module  CBBankCheckAccountHome
```

*Figure 42. CBBankCheckAccountHome IDL*

### 5.4.11 SavingsAccount

The following table shows the different functions of **CBBankSavingsAccount** and **CBBankSavingsAccountHome**.

*Table 9. Role of Note Objects*

| Object Name | Function Performed | Remarks |
|---|---|---|
| CBBankSavingsAccount | -Make persistent attributes such as balance, interest_rate...<br>-credit/debit<br>-freeze, close, open account<br>- get the interest_rate from the Policy Manager | Backed with UDB on NT. |
| CBBankSavingsHome | - Create a SavingsAccount<br>- Transfer amount | |

#### 5.4.11.1 CBBankSavingsAccount

The following IDL shows the attributes and methods of **CBBankSavingsAccount**.

```
module CBBankSavingsAccount {
interface SavingsAccount : IManagedClient::IManageable
  {
    attribute string customerKey;
    attribute string dateCreated;
    attribute string name;
    attribute CBBankConstructs::StatusType status;
    attribute double balance;
    attribute float creditInterestRate;
    attribute float debitInterestRate;
    attribute double serviceFee;
    attribute double minLimit;
    attribute string key;
    void freeze( ) raises
(CBBankConstructs::Frozen,CBBankConstructs::Closed);
    void unfreeze( ) raises
(CBBankConstructs::Unfrozen,CBBankConstructs::Closed);
    void credit(in double amount ) raises
(CBBankConstructs::NegativeAmount,CBBankConstructs::Frozen,
CBBankConstructs::Closed,CBBankConstructs::ProcessingFailed);
    void debit(in double amount ) raises
(CBBankConstructs::NegativeAmount,CBBankConstructs::BeyondDebitLimit,
CBBankConstructs::Frozen,CBBankConstructs::Closed,
CBBankConstructs::ProcessingFailed);
    ::IManagedCollections::IIterator listTransactions( );
  }; // end interface SavingsAccount
}; // end of module  CBBankSavingsAccount
```

*Figure 43. CBBankSavingsAccount*

#### 5.4.11.2 CBBankSavingsAccountHome

The following IDL shows the methods of **CBBankSavingsAccountHome** in IDL.

```
module CBBankSavingsAccountHome {
interface SavingsAccountHome :
IManagedAdvancedClient::IQueryableIterableHome
  {
    CBBankSavingsAccount::SavingsAccount createSavingsAccount
(in ::ByteString copyString ) raises
(IManagedClient::IDuplicateKey,IManagedClient::IInvalidKey,
IManagedClient::IInvalidCopy,CBBankConstructs::ProcessingFailed);
    void transfer(in CBBankSavingsAccount::SavingsAccount
fromAccount,in CBBankSavingsAccount::SavingsAccount toAccount,
in double amount ) raises
(CBBankConstructs::Frozen,CBBankConstructs::Closed,
CBBankConstructs::NegativeAmount,CBBankConstructs::BeyondDebitLimit,
CBBankConstructs::ProcessingFailed);
  }; // end interface SavingsAccountHome
}; // end of module  CBBankSavingsAccountHome
```

*Figure 44. CBBankSavingsAccountHome IDL*

### 5.4.12 TransactionRecord

The following table shows the function of **TransactionRecord** objects.

*Table 10. Role of Note Objects*

| Object Name | Function Performed | Remarks |
|---|---|---|
| CBBankTransactionRecord | Make persistent TransactionRecord attributes such as time created, description... | UDB back-end |
| CBBankTransactionRecordHome | Creates a CBBankTransactionRecord object | |

#### 5.4.12.1 CBBankTransactionRecord

The following IDL shows the attribute of **CBBankTransactionRecord**.

```
module CBBankTransactionRecord {
interface TransactionRecord : IManagedClient::IManageable
  {
    attribute string customerKey;
    attribute string accountKey;
    attribute ::CBBankConstructs::TransactionType transKind;
    attribute string timeCreated;
    attribute string dateCreated;
    attribute string description;
    attribute double amount;
  }; // end interface TransactionRecord
}; // end of module  CBBankTransactionRecord
```

*Figure 45. CBBankTransactionRecord IDL*

#### 5.4.12.2 CBBankTransactionRecordHome

The following IDL shows the methods of **CBBankTransactionRecordHome** in the IDL.

```
module CBBankTransactionRecordHome {
interface TransactionRecordHome :
IManagedAdvancedClient::IQueryableIterableHome
  {
    CBBankTransactionRecord::TransactionRecord
createTransactionRecord(in ::ByteString copyString ) raises
(IManagedClient::IDuplicateKey,IManagedClient::IInvalidKey,
IManagedClient::IInvalidCopy,CBBankConstructs::ProcessingFailed);
  }; // end interface TransactionRecordHome
}; // end of module  CBBankTransactionRecordHome
```

*Figure 46. CBBankTransactionRecordHome IDL*

### 5.4.13 Policy

The Policy was developed independently in another model. The current CBConnector Bank model has a link to the Policy Model.

The Policy Objects allow you to set, get and delete <name, value> pairs stored persistently in UDB. Names are structured as strings in a logical tree. Policy provides you the ability to dynamically change the value of static variables used in the CBConnector Bank application.

Such functionality can be used to implement various types of services. Our policy can be used as a Name Service or even as a database.

For example, it allows you to set a default policy such as

```
//account/default_interest_rate = 5.7"
```

and eventually change it without recompiling the whole model.

It allows the retrieval of values even if the full name doesn't match the name stored in the database.

Example:

Let say we have a database with:

```
setPolicy("/account/default_interest_rate", 5.7)
```

```
getPolicy("/account/check/Clint.Eastwood/default_interest_rate")
```

This will return `5.7` because the database has a `default_interest` for all types of account.

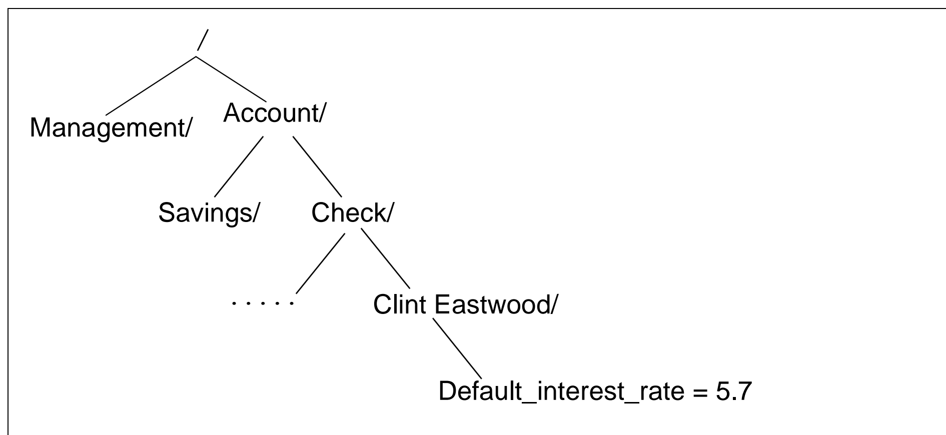



*Figure 47. Policy Naming Hierarchy*

#### 5.4.13.1 Policy Description

The following table shows the function of Policy Objects.

*Table 11. Role of Policy Objects*

| Object Name | Function Performed | Remarks |
|---|---|---|
| CBBankPolicy | -Store persistently <name, value> with getPolicy and setPolicy | UDB NT backend |

| Object Name | Function Performed | Remarks |
|---|---|---|
| CBBankPolicyManager | - get/setPolicy<br>- List all policies<br>- Execute OO-SQL clause | Only the manager sees the CBBankPolicy |

#### 5.4.13.2 CBBankPolicy Exceptions

The **listPolicy** method described in the Policy Manager are supposed to return a list of Policys. However, the current version of Object builder does not support sequence as a return value. Therefore, we had to return a string and insert a delimiter between each item. The following exceptions were defined for **CBBankPolicy**.

```
module CBBankPolicy {
exception InvalidName {
  }; // end exception  InvalidName
  exception InvalidValue {
  }; // end exception  InvalidValue
  exception NotFound {
  }; // end exception  NotFound
  exception ProcessingFailed {
  }; // end exception  ProcessingFailed
  exception InvalidDelimiter {
  }; // end exception  InvalidDelimiter
  exception InvalidQuery {
  }; // end exception  InvalidQuery
}; // end of module  CBBankPolicy
```

*Figure 48. CBBankPolicy Exceptions IDL*

#### 5.4.13.3 CBBankPolicy

The following IDL shows the attributes of **CBBankPolicy**.

```
interface Policy : IManagedClient::IManageable
 {
   attribute string<100> name;
   attribute string<50> value;
  }; // end interface Policy
```

*Figure 49. CBBConnector Bank Policy IDL*

#### 5.4.13.4 Policy Manager

The Policy Manager Interface provides methods that iterate over policies.

```
interface PolicyManager : IManagedClient::IManageable
  {
    readonly attribute string singletonKey;
    void setPolicy(in string<100> name,in string<50> value ) raises
(CBBankPolicy::InvalidName,CBBankPolicy::InvalidValue,
CBBankPolicy::ProcessingFailed);
    string<50> getPolicy(in string<100> name ) raises
(CBBankPolicy::InvalidName,CBBankPolicy::NotFound,
CBBankPolicy::ProcessingFailed);
    void deletePolicy(in string<100> name ) raises
(CBBankPolicy::InvalidName,CBBankPolicy::NotFound,
CBBankPolicy::ProcessingFailed);
    string listPolicies(in string sqlWhereClause,in string delimiter )
raises
(CBBankPolicy::InvalidQuery,CBBankPolicy::InvalidDelimiter,
CBBankPolicy::ProcessingFailed);
    string allPolicies(in string delimiter ) raises
(CBBankPolicy::InvalidDelimiter,CBBankPolicy::ProcessingFailed);
    string listPolicyNames(in string sqlWhereClause,in string delimiter
) raises
(CBBankPolicy::InvalidQuery,CBBankPolicy::InvalidDelimiter,
CBBankPolicy::ProcessingFailed);
    string allPolicyNames(in string delimiter ) raises
(CBBankPolicy::InvalidDelimiter,CBBankPolicy::ProcessingFailed);
  }; // end interface PolicyManager
```

*Figure 50. CBBankPOlicy Manager IDL*

## 5.5 CBConnector 1.2 Limitations

This section outlines the limitations we encountered while we were building the CBC Banking sample. We encountered these limitations at three stages. They are the Object Builder, CICS and runtime. These limitations changed significantly the model we aimed to build. The final model looks pretty much like an Entity/Relationship (E/R) model. We recommend you read the *Late-Breaking News* document supplied with the product.

*Object Builder:*

- The return type cannot be a sequence or any complex structure. We use strings and put delimiters on a string to simulate a structure.
- Object relationship is not supported; we use iterators instead.

- Inheritance is not supported; as far as we know, there is no clean way to simulate inheritance.
- The generated IDL is wrong when the enumerated type is added. In order to remedy the problem, we manually changed the IDL and imported it into the model. A similar approach can be taken with XML.
- CORBA Any type is not supported in the Copy Helper.

***CICS:***

- Sessions and transactions: CICS has to be called in a session meaning that the current transaction has to be suspended.
- The following corrections must be done for the DO implementation file generated by Object Builder.
  - All occurrences of the isolated word `string` must be changed to `CORBA::String_var`.
  - All occurrences of `TempHandle` imbedded in an identifier should be changed to `HandleTemp`. By imbedded, we mean identifiers that contain more text than simply the word `tempHandle`.
- Access to CICS back-end from Java client

  In order to successfully access the CICS back-end from the Java client, we have to implement the following:

  We created a Transient Object called **CustomerWrapper** with the same interface as the **Customer** Object on the client.

  The **CustomerWrapper** can maintain the sessions and all attributes of the **Customer** Object. If you would like to study the code, please load the CICS Customer Model to the Object Builder.

## 5.6 Conclusion

The chapter showed how we implemented the model we defined in Chapter 2, “Design” on page 11. All objects, together with their IDLs, were detailed. The relationships and interactions between objects were also presented. The CBConnector Bank sample demonstrated the use of important features of CBConnector. We outlined the limitation of the current version of the toolkit and suggested solutions to remedy these limitations. This sample can be used as a guide to build other complex applications with CBConnector.

# Chapter 6. Persistent Implementation

This chapter explains the import process of the JavaBean to Object Builder after creation in the CICON tool. Please, refer to the *CICS and IMS Application Adaptor Quick Beginnings* guide and to the *IBM CBConnector Cookbook Collection: First Steps redbook* for additional information.

## 6.1 Importing the Bean to Object Builder

When you start working with Object Builder, you must include the *.jar* file created with CICON to your CLASSPATH. The easiest and cleanest way to do this is to copy *ob.bat*, and modify this newly created batch file to include a *.jar* file to the CLASSPATH. This way, the *.jar* file is always in the CLASSPATH whenever you start Object Builder. Don't forget to enable Object Builder beta capabilities.

After opening the project, you must import the bean (**User defined PAO schemas -> Import bean**). The Object Builder asks you for the name of the class or lets you select a class from the *.jar* file. This class or *.jar* file must be in the CLASSPATH. The only thing you have to do after loading the *.jar* file is to select key fields from the fields in the *.jar* file. See the *CICS & IMS Application Adaptor Quick Beginnings* guide for more details.

## 6.2 Implementing the DO

Be sure that you select the correct options when implementing the DO. The options are

- BOIM with any key (Environment)
- Procedural Adaptors (Form of persistent behavior)
- Delegating (Data access pattern)
- Default (Handle for storing pointers)

These options make DO to inherit from **IPAAExtLocalToServer::IDataObject**, which delegates all method calls to the PAO.

© Copyright IBM Corp. 1998

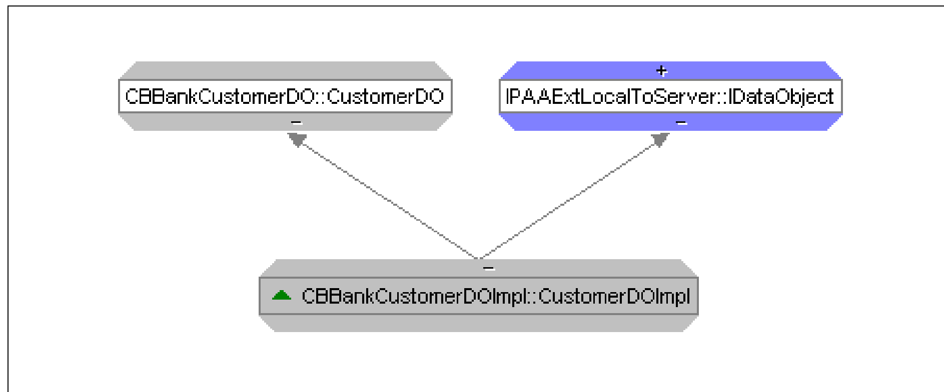



*Figure 51. Data Object Inheritance*

## 6.3 Using Mapping Helpers

If the target PAO attributes happen to differ from your BO interface attributes (for instance, you aggregating objects), you must use mapping helpers to resolve this problem. The mapping helper is a C++ class containing only methods, because this class is never instantiated by the DO implementation. Using mapping helpers, you can do whatever mapping is required between the DO implementation and the PAO. Objects are always passed in methods in general form, such as a handle string. In Object Builder, you can define the name of class that is doing the mapping and names of methods for mapping the data in both directions. While doing the bindings between the DO and PAO, Object Builder automatically asks you to enter the name of mapping helper class if you have incompatible bindings (such as enumeration to string).

Here's an example of the mapping helper that maps enumeration `GenderType` to `String` in the PAO and vice versa.

```
void GenderMapper::PAOToEnum(
    const CORBA::String& strGender,
    CBBankConstructs::GenderType& gender )
{
    if (*strGender == 'F' )
        gender = CBBankConstructs::FEMALE;
    else
        gender = CBBankConstructs::MALE;
}

void GenderMapper::enumToPAO(
        const CBBankConstructs::GenderType& gender,
        CORBA::String& strGender )
{
    if ( gender == CBBankConstructs::FEMALE )
        strGender = CORBA::string_dup ( "F" );
    else
        strGender = CORBA::string_dup ( "M" );
}
```

*Figure 52. GenderType Conversion in PAO*

The mapping gets more complex when one needs to map imbedded objects to CICS. Our sample implements a mapping helper to move **Address** objects back and forth to CICS while **Customer** is being stored/retrieved. If your object contains more than one attribute, you need to map a given object to many PAO attributes and define the name of the mapping helper class and mapping methods.

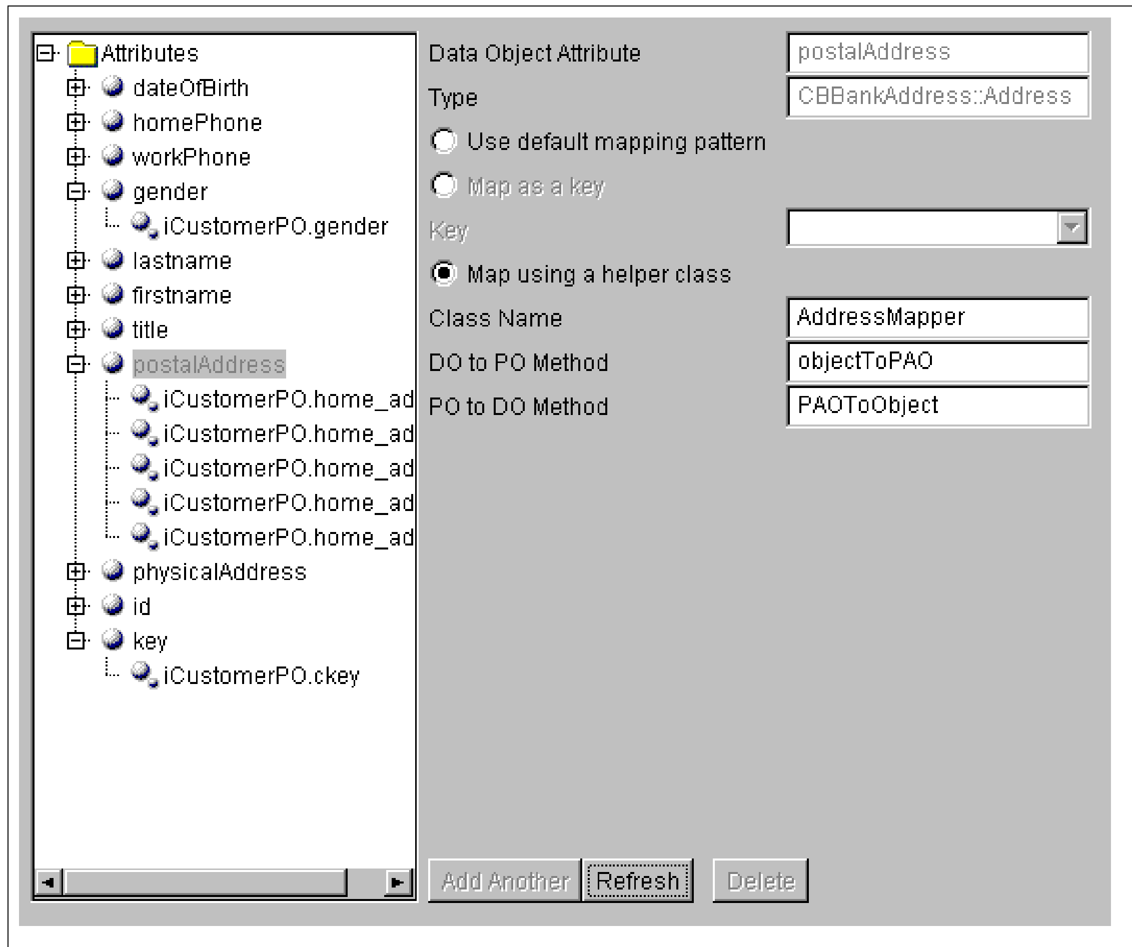



*Figure 53. Mapping Objects with PAO*

## 6.4 Fixing Generated Code

The following corrections must be done for the DO implementation file generated by Object Builder.

- All occurrences of the isolated word `string` must be changed to `CORBA::String_var`.
- All occurrences of `TempHandle` imbedded in an identifier should be changed to `HandleTemp`. By imbedded, we mean identifiers that contain more text than simply the word `tempHandle`.

# Chapter 7. Legacy Tier

This chapter briefs you about our legacy tier. Please study this on our CD-ROM in the *Legacy* subdirectory.

## 7.1 Data Implementation

As you have seen in 2.4, “Data Model” on page 23, the data model reflects some of the familiar entities in a banking domain. We decided on a relational database implementation for our data model and selected DB2 for the physical implementation. A separate DB2 database was created for the data implementation. On this database, we created separate tables for the entities reflected on the data model shown in Figure 15 on page 24.

Although all the DB2 tables were created, we decided to only use the CUSTOMER and ADDRESS tables on the 'legacy tier'. We did not want to create a full banking application, but rather demonstrate that we could use DB2 and CICS in a distributed computing environment.

The data management is done through a series of CICS programs. The data logic has been separated from the business logic and screen handling. We did this by implementing the CRUD methods for the data in separate IO-modules for each DB2 table. The physical implementation of the data for any of the other tables can therefore be done with the minimum effort by copying any of the existing IO modules and changing only the column names to reflect those of the specific table.

We are confident that if the need arises, the data implementation would be able to be used and expanded to reflect any true banking environment.

## 7.2 Host CICS Transaction

Most mainframe sites have applications running under Customer Information Control System (CICS), and these applications have been developed and extended over the years. The idea behind the Component Broker Connector Procedural Application Adaptor (PAA) is to enable customers to use their existing CICS applications in new applications written for Component Broker Connector.

In the residency that produced this redbook, we were in the opposite situation; we had to create a CICS application that had to be representative of what would be found in customer sites.

© Copyright IBM Corp. 1998

The tool provided by IBM to support the PAA is the CICS and IMS Connecting (CICON) tool. The CICON tool is an extension to VisualAge for Java (VAJava). CICON supports two methods of communication with CICS: Basic Mapping Support (BMS) "screen-scraping" and by use of the CICS External Call Interface (ECI). The result of the CICON process for screen-scraping, or ECI, is a JavaBean that is imported into Component Broker Connector.

The screen-scraping technique may be used on most existing CICS systems, just as people have used 3270 emulator Application Programming Interface (API) in order to make front-ends to existing CICS systems.

The ECI approach, however, is a little more demanding when it comes to the design of the original CICS application. Because the ECI is a Remote Procedure Call (RPC) mechanism, it expects a certain structure to the modularity of the CICS application. In our case, we could design the application so that both approaches could be used, as described below. **Due to time constraints, we only implemented the JavaBean doing the screen-scraping.**

### 7.2.1 Host Flow Example

Our CICS COBOL application was designed around an extendable menu and is supported by a program named CBCPGC01. For every type of function supported on the menu, an additional program is needed. In our sample, only the Customer function is supported, and it is represented by the CBCPGCUS program.

The Customer function processes the following information: first name, last name, title, ID number, date of birth, gender, home phone, and work phone. In addition, two types of address information are processed, each with five fields of information.

When the Customer function is selected from the menu (by entering CUST in the function field), a CICS LINK to the CBCPGCUS program is performed. This program is responsible for collecting the necessary information through its own map (CUSTDTL) and presenting the information to the appropriate IO module. It communicates with the IO module using the Communication Area of the CICS LINK feature. The result of the IO operation is presented by CBCPGCUS on its associated map.

Each IO module supports the **create**, **retrieve**, **update**, **delete** (CRUD) methods of a specific SQL table. The communication area used for passing information between CBCPGCUS and the IO modules (a customer table IO module and an address table IO module) consists of a general part and an

application-specific part. The general part contains an action field specifying the CRUD method to execute, a return code field and a message field. The application part contains all the information necessary to perform the relevant CRUD methods.

The main reason for this design is to be able to support both screen scraping and ECI with the same application. The ECI implementation bypasses the menu program (CBCPGC01) and the customer function program (CBCPGCUS) and communicates directly with the appropriate IO module. The picture below shows the flow.

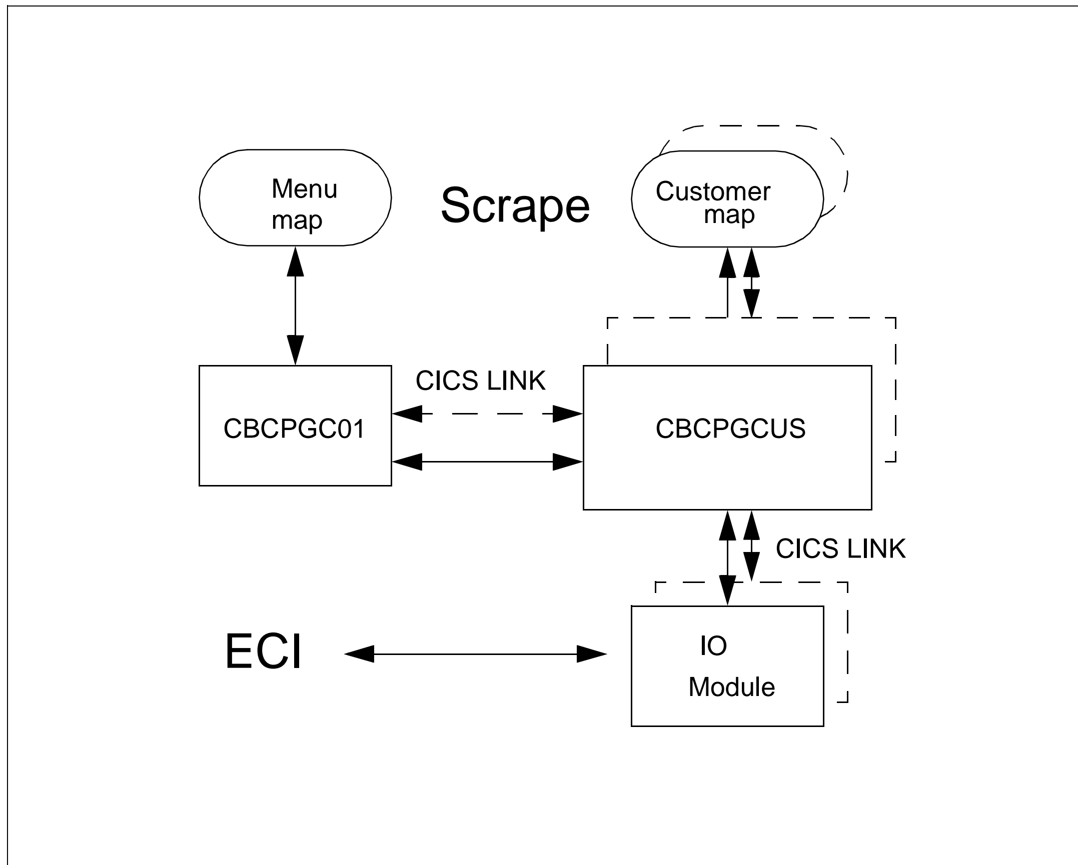



*Figure 54. Flow of the CICS Host Application*

## 7.3 Creating Procedural Adaptor Object

The Procedural Adaptor Object (PAO) interacts with the Data Objects of the Component Broker Connector server so that the attributes of a Data Object are maintained transparently by the PAO. The PAO, on the other hand, is supported by CICS Transaction Objects (TO), performing either screen scraping or ECI to CICS.

The PAO is created by using CICON. Consult the *CICS and IMS Application Adaptor Quick Beginnings* guide and the *IBM CBConnector Cookbook Collection: First Steps* redbook to learn the exact steps you have to perform in order to create the PAO class.

The PAO class is created as a new class inheriting from the **CBProceduralAdaptorObject** class. You have to specify the attributes (called properties in VAJava) through the BeanInfo tab. Note that you must specify either **#key#** or **#general#** in the short description field of every attribute; otherwise the CICON parsers will not see the attributes.

### 7.3.1 Screen Scraping - Implementation

The screen-scraping mechanism uses the API of the Host on Demand 3270 emulator to put data into CICS BMS maps as well as for retrieving data from them.

The CICON tool helps you to perform the following tasks:

- Initial Navigation and Logon
- Creation of Transaction Objects (TO)

Consult the *CICS and IMS Application Adaptor Quick Beginnings* and the *IBM CBConnector Cookbook Collection: First Steps* redbook in order to learn the exact steps you have to perform in order to do the tasks mentioned above.

The Initial Navigation and Logon is dependent on your specific operating environment; so we will not try to explain how to do it. Note, however, that when you create your logon class, you have to add an attribute named `ANY` (of type `String`) to the class in order to get it to work.

You need to create your Transaction Objects, one per CRUD method. This is done by using the visual composition mode of VAJava. You must create a closed loop of connections from the Start Object (of type `Start`) through your CICS objects, and finally back to the Start Object.

It seems that you have to make an artificial transition from the last CICS object in the loop to the Start Object. A dummy map with a field of 50 bytes from position 0 is used as the transition criteria, together with the CLEAR action.

The different CICS objects represent the appropriate states of your application in order to specify input and output for the screen scraping process.

The TO classes created for the different CRUD methods need some manual update in order to function. The reason is that the error handling associated with screen scraping normally involves interpreting error messages given by the application, and targeted for the end user.

When the visual composition is finished, you have to generate the CRUD method of the TO by selecting the Class Settings on the Property setting of the visual background. The general advice on this method generation, is to create the method as an internal method, and then create the CRUD method manually and invoke the generated method. In this way, you may handle errors in your handwritten CRUD method.

### 7.3.2 Screen Scraping - Testing

VAJava allows you to test the client application that you have created. In order to do this, you need a main routine in which to instantiate an object of the PAO class, and invoke its CRUD methods. Note that it is important to invoke the **setWorkspaceId** method and the **find** method on the PAO object before invoking the relevant CRUD method.

Before you can do the unit test, you have to add information in the PAO constructor about your CICS system. You must invoke the method **setLogonLogoffClassname** with the name of the **LogonLogoff** class you created as a parameter. The TCP/IP address of the CICS system and the port number are specified as parameters to the methods **setHostName** and **setPortNumber**. Finally, you may invoke the **setDebugScreenEnabled** method with `true` as a parameter in order to monitor the actual screen-scraping process.

When you are satisfied with your Unit Testing through CICON, it is a good idea to export your bean, and then try to Unit Test it outside of VAJava from a command line. In this way, you sort out the necessary `CLASSPATH` statements needed for the bean to execute. S

# Chapter 8. Summary

By showing in detail all the development processes of a banking application, this redbook is a practical guide for building a real application with CBConnector.

In it, we address a large number of concepts covering object-oriented design methodologies, including:

- Rational Rose and CORBA
- OO programming, such as C++ and Java
- Procedural programming languages, such as COBOL
- OO databases, such as OO-SQL
- Procedural databases, such as DB2
- CICS IMS adaptors, graphical interfaces and transaction processing (CICS and OTS)

The chosen banking application gave us the opportunity to study a realistic application that uses all the above concepts. The application also exploits almost all the CBConnector functions needed to demonstrate the ability of CBConnector to:

- Integrate all the above concepts
- Interoperate between different middlewares

© Copyright IBM Corp. 1998

# Part 3. Appendices

© Copyright IBM Corp. 1998

# Appendix A. Special Notices

This publication is intended to help designers and developers to build application using the CBConnector development environment. The information in this publication is not intended as the specification of any programming interfaces that are provided by the product reference manuals.

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, 500 Columbus Avenue, Thornwood, NY 10594 USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, 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 responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere.

© Copyright IBM Corp. 1998

Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

You can reproduce a page in this document as a transparency, if that page has the copyright notice on it. The copyright notice must appear on each page being reproduced.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

| | |
|---|---|
| CICS | DB2 |
| IBM ® | MVS |
| VisualAge | |

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Incorporated.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

Pentium, MMX, ProShare, LANDesk, and ActionMedia are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be trademarks or service marks of others.

# Appendix B. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

## B.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 115.

- *IBM Component Broker Connector Overview*, SG24-2022
- *IBM CBConnector Cookbook Collection, First Steps* SG24-2033 (in press)
- *IBM CBConnector Cookbook Collection, CBConnector Bank User Guide* SG24-5121 (in press)

## B.2 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

| CD-ROM Title | Subscription Number | Collection Kit Number |
|---|---|---|
| System/390 Redbooks Collection | SBOF-7201 | SK2T-2177 |
| Networking and Systems Management Redbooks Collection | SBOF-7370 | SK2T-6022 |
| Transaction Processing and Data Management Redbook | SBOF-7240 | SK2T-8038 |
| Lotus Redbooks Collection | SBOF-6899 | SK2T-8039 |
| Tivoli Redbooks Collection | SBOF-6898 | SK2T-8044 |
| AS/400 Redbooks Collection | SBOF-7270 | SK2T-2849 |
| RS/6000 Redbooks Collection (HTML, BkMgr) | SBOF-7230 | SK2T-8040 |
| RS/6000 Redbooks Collection (PostScript) | SBOF-7205 | SK2T-8041 |
| RS/6000 Redbooks Collection (PDF Format) | SBOF-8700 | SK2T-8043 |
| Application Development Redbooks Collection | SBOF-7290 | SK2T-8037 |

## B.3 Other Publications

These publications are also relevant as further information sources:

- *IBM Component Broker Connector - Quick Beginnings,* G04L-2375 Available with the CBConnector package

© Copyright IBM Corp. 1998

- *IBM Component Broker - Programming Guide,* G04L-2376 - Available with the CBConnector package
- *IBM Component Broker - CICS and IMS Application Adaptor Quick Beginnings,* G04L-2703 - Available with the CBConnector package
- *IBM Component Broker - System Administration Guide,* SC09-2704 - Available with the CBConnector package
- *IBM Component Broker - Application Development Tools,* SC09-2705 - Available with the CBConnector package
- *IBM Component Broker - Application Programming Guide,* SC09-2708 - Available with the CBConnector package

These publivcations are relevant as further information sources:

- *Orbix 2, Programming Guide*
- *Orbix 2, Reference Guide*
- *The Essential Distributed Objects Survival Guide,* ISBN 0-471-12993-3
- *Client/Server Programming with JAVA and CORBA,* ISBN 0-471-16351-1
- *Understanding CORBA,* ISBN 0-13-459884-9

# How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at `http://www.redbooks.ibm.com/`.

## How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Redbooks Web Site on the World Wide Web**

  `http://w3.itso.ibm.com/`

- **PUBORDER** – to order hardcopies in the United States
- **Tools Disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLCAT REDPRINT
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  ```

  To get BokkManager BOOKs of redbooks, type the following command:

  ```
  TOOLCAT REDBOOKS
  ```

  To get lists of redbooks, type the following command:

  ```
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  ```

  To register for information on workshops, residencies, and redbooks, type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1998
  ```

- **REDBOOKS Category on INEWS**
- **Online** – send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL

**Redpieces**

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). 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.

© Copyright IBM Corp. 1998

## How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** – send orders to:

| | **IBMMAIL** | **Internet** |
|---|---|---|
| In United States | usib6fpl at ibmmail | usib6fpl@ibmmail.com |
| In Canada | caibmbkz at ibmmail | lmannix@vnet.ibm.com |
| Outside North America | dkibmbsh at ibmmail | bookshop@dk.ibm.com |

- **Telephone Orders**

| | |
|---|---|
| United States (toll free) | 1-800-879-2755 |
| Canada (toll free) | 1-800-IBM-4YOU |
| Outside North America | (long distance charges apply) |
| (+45) 4810-1320 - Danish | (+45) 4810-1020 - German |
| (+45) 4810-1420 - Dutch | (+45) 4810-1620 - Italian |
| (+45) 4810-1540 - English | (+45) 4810-1270 - Norwegian |
| (+45) 4810-1670 - Finnish | (+45) 4810-1120 - Spanish |
| (+45) 4810-1220 - French | (+45) 4810-1170 - Swedish |

- **Mail Orders** – send orders to:

| | | |
|---|---|---|
| IBM Publications | IBM Publications | IBM Direct Services |
| Publications Customer Support | 144-4th Avenue, S.W. | Sortemosevej 21 |
| P.O. Box 29570 | Calgary, Alberta T2P 3N5 | DK-3450 Allerød |
| Raleigh, NC 27626-0570 | Canada | Denmark |
| USA | | |

- **Fax** – send orders to:

| | |
|---|---|
| United States (toll free) | 1-800-445-9269 |
| Canada | 1-800-267-4455 |
| Outside North America | (+45) 48 14 2207 (long distance charge) |

- **1-800-IBM-4FAX (United States)** or **(+1) 408 256 5422 (Outside USA)** – ask for:

  Index # 4421 Abstracts of new redbooks
  Index # 4422 IBM redbooks
  Index # 4420 Redbooks for last six months

- **On the World Wide Web**

| | |
|---|---|
| Redbooks Web Site | http://www.redbooks.ibm.com |
| IBM Direct Publications Catalog | http://www.elink.ibmlink.ibm.com/pbl/pbl |

**Redpieces**

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). 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.

## IBM Redbook 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 Application Adaptor. 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. Workload Management 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

© Copyright IBM Corp. 1998

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 Application Adaptor 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 run time 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 run time.

**Dynamic Skeleton.** An interface-independent type of skeleton. It is used by CBConnector. servers to handle requests whose signatures are not known until run time.

**Embedded Aggregation.** A form of aggregation where the sub-objects remain visible from outside of the aggregate.

**Extended Naming Context (ENC).** 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 reside 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.Proxy Object**. 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 Application Adaptor. 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 Application Adaptor 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, LifeCycle, 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.

**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 that 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 Transaction 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 Application Adaptor 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. Workload Management 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.

## 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 |

© Copyright IBM Corp. 1998

| | |
|---|---|
| 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 |
| 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 |
| WLM | Workload Management Enhanced Client |
| .XML | Extended Markup Language |

# Index


## Symbols
/Procedural Adaptor 74

## A
abbreviations 127
acronyms 127
Address 41
Audience of the book 9

## B
Bean importation 97
Beans 74
BO Action Classes 50
BO Specification Reading 58
BOAction 48
Business Object 37
Business Objects 74
Business Objects (BO) 68

## C
CheckAccount 41
CICS 74
CICS Transaction 101
Class Account 15, 30
Class Address 16, 22, 29, 40
Class Bank 29
Class CBCBase 36
Class CheckAccount 20, 30
Class CreditTransaction 15
Class Customer 14, 22
Class Customers 30
Class DataSupplier 51
Class DebitTransaction 15
Class Entity 14, 29
Class Hierarchy 52
Class Note 16, 22, 29
Class Party 14, 29
Class PhysicalAddress 16
Class Policy Manager 18
Class PostalAddress 16
Class Product 29
Class SavingsAccount 15, 20, 30
Class Teller 14, 22, 29
Class TransactionRecord 15, 20
Class Transactions 30
Code Generation 100
Coexist 12
Container 50
Containers 68
Copy Helper 37
createMO 37
CustomerMapper 43

## D
Data implementation 101
Data Model 24
Data Objects (DO) 68
Distributed object environment 11
DO Implementation 97

## E
entity 14
Environment variable BOSS_PATH 27
Environment variable ROSE_MENU_PATH 27
Exception 58
Exception Handling 39

## F
findMO 37

## G
Generic Home Objects 68
Graphical User Interface 35

## H
Hierarchy 52
Hierarchy file format 54
Host flow 102
How to read the book 5

## I
IDL Parser 57
Inheritance 11, 19
Interface 58
Interoperability 11

## K
Key Helper 37


© Copyright IBM Corp. 1998

## L
Legacy systems 12
Limitations 18

## M
Managed object container 68
Managed Objects (MO) 69
Mapping Helpers 98
MixIn 68

## N
Note 43

## O
Object relationship 19
Object Relationships 22
Operation Panels 49
OperationPanel 48

## P
PAA
  Procedural Application Adaptor 69
PAO 75, 104
Persistent 74
Persistent Objects (PO) 69
Policy Description 93
Policy Model 16
Policy Namespace 17
Portability 11
Procedural Adaptor Object 104
Proxies Bank 39
Proxy 37, 57
Proxy Base 38
Proxy Factory 61
Proxy Generator 65
Proxy Generic 38
Proxy Optimization 44

## R
Relational database mapping to objects 19
Relationship 23
resolveFactoryFinder 36
resolveHome 36
resolveNameService 36
resolveORB 36

## S
SavingsAccount 41
Scope of the book 4
Screen Scaping 105
Screen Scraping 104
Specialized Home 37
Specialized Home Object 68, 70

## T
Teller 42
TransactionRecord 43

## V
View Structure 47

# ITSO Redbook Evaluation

IBM CBConnector Cookbook Collection CBConnector Bank Implementation
SG24-5119-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.ibm.com
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
**_ Customer _ Business Partner _ Independent Software Vendor _ IBM employee**
**_ None of the above**

**Please rate your overall satisfaction** with this book using the scale:
**(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)**

Overall Satisfaction __________

**Please answer the following questions:**

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

What other redbooks would you like to see published?

**Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)**

© Copyright IBM Corp. 1998

IBM CBConnector Cookbook Collection CBConnector Bank Implementation

SG24-5119-00

SG24-5119-00
Printed in the U.S.A.