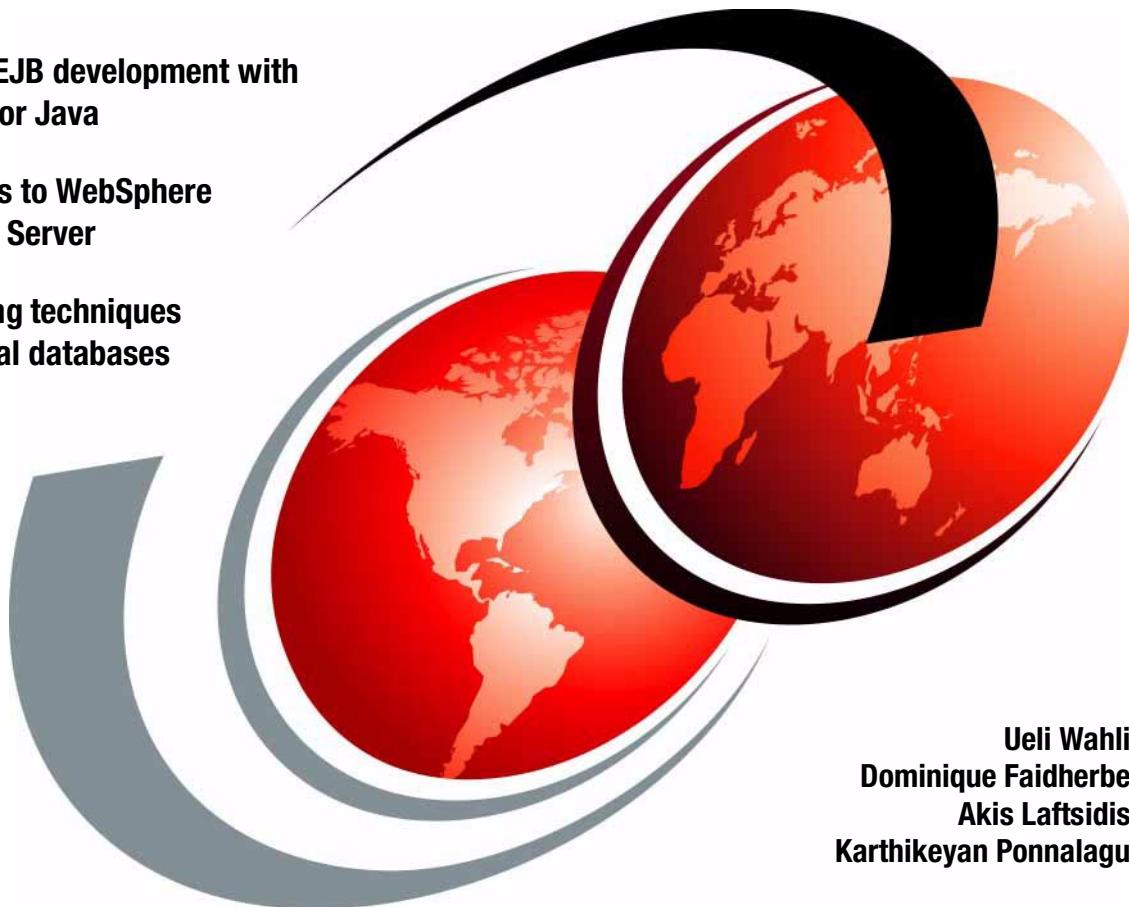


EJB Development with VisualAge for Java for WebSphere Application Server

Jumpstart EJB development with
VisualAge for Java

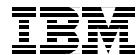
Deploy EJBs to WebSphere
Application Server

EJB mapping techniques
for relational databases



Ueli Wahli
Dominique Faidherbe
Akis Laftsidis
Karthikeyan Ponnalagu

Redbooks



International Technical Support Organization

**EJB Development with VisualAge for Java for
WebSphere Application Server**

July 2001

Take Note! Before using this information and the product it supports, be sure to read the general information in “Special notices” on page 501.

First Edition (July 2001)

This edition applies to VisualAge for Java Version 3.5.3 and WebSphere Application Server Version 3.5.3 for use with the Windows NT or Windows 2000 operating system.

Comments may be addressed to:

IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

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 2001. 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.

Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition may also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-6144-00
for EJB Development with VisualAge for Java for WebSphere Application Server
as created or updated on June 29, 2001.

First Edition, July 2001

This revision reflects the addition, deletion, or modification of new and changed information described below.

This book is basically a rewrite of the redbook *Enterprise JavaBeans Development Using VisualAge for Java*, SG24-5429, updated for VisualAge for Java Version 3.5.3 and WebSphere Application Server Version 3.5.3.

New information

- ▶ WebSphere Test Environment Control Center
- ▶ Enterprise Access Builder session beans
- ▶ Access beans
- ▶ Generic EJB test client
- ▶ Deployment to WebSphere using XMLConfig
- ▶ Mapping to multiple tables and using composers

Changed information

- ▶ Enhanced custom finder methods
- ▶ Official product support for inheritance and associations
- ▶ Design considerations and best practices

This book should be used together with the redbook *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*, SG24-5754.

Contents

Summary of changes	iii
First Edition, July 2001	iii
Preface	xix
The team that wrote this redbook	xx
Special notice	xxi
IBM trademarks	xxi
Comments welcome	xxi
Part 1. Introduction	1
 Chapter 1. Enterprise JavaBeans and J2EE	3
Server-side component architecture	4
Why EJBs?	5
Object distribution	5
Object persistence	5
Independence from database schema	6
Transaction management	6
Platform independence	6
Scalable environment	7
Secure access	7
Middle-tier architecture	8
Development roles	8
Multiple client types	8
Integration with CORBA	8
Java 2 Platform, Enterprise Edition	9
Enterprise JavaBeans (EJB)	9
Java Remote Method Invocation and RMI-IIOP	9
Java Naming and Directory Interface (JNDI)	9
Java Database Connectivity (JDBC)	9
Java Transaction API (JTA) and Java Transaction Service (JTS)	10
Java Messaging Service (JMS)	10
Java servlets and JavaServer Pages	10
Java IDL	10
Java Activation Framework (JAF)	11
Java Mail	11
J2EE architecture overview	11
EJB specifications	12
EJB 1.0	12

EJB 1.1	13
EJB 2.0	14
General restrictions of EJB	15
IBM products and J2EE.....	16
Adopting EJB technology	18
Chapter 2. EJBs: Basic concepts and architecture	19
EJB basics	20
Distributed component-oriented applications.....	20
CORBA overview	20
RMI overview	21
The solution.....	22
RMI over IIOP	22
Enterprise JavaBeans.....	22
JavaBeans versus Enterprise JavaBeans.....	23
Enterprise Java Server	24
EJB container	24
Entity bean	26
Container-managed persistence (CMP)	26
Bean-managed persistence (BMP).....	26
Comparing CMP with BMP beans.....	27
Session bean	27
Stateful session bean	28
Stateless session bean.....	28
Comparing stateless with stateful beans.....	28
Deployment of enterprise beans	29
Comparing entity and session beans.....	30
End-to-end enterprise application development.....	32
EJB container responsibilities	33
Container-managed transactions	34
Data caching	34
Persistence	34
Remote accessibility	35
Naming	35
Bean life-cycle management.....	35
State management	35
Security	36
An enterprise bean and its interfaces	36
Bean class.....	36
Remote interface.....	37
Home interface	37
Key class.....	37
Finding the home of an enterprise bean	38

Chapter 3. IBM products for EJB development and deployment	39
WebSphere Application Server, Advanced Edition	40
Overview	40
Architecture	40
J2EE and WebSphere	41
WebSphere tooling	42
JetAce tool	42
EJBDeploy tool	43
VisualAge for Java Enterprise Edition	44
Overview	44
Integrated Development Environment	44
Database connectivity	45
Enterprise connectivity	45
J2EE development environment	45
EJB page	46
Container-managed persistence	47
WebSphere Test Environment	48
Deployment	49
Testing and debugging	49
RAD support	51
Team environment	51
Chapter 4. ITSO bank example	53
Bank model	54
Bank database	55
Database definition DDL	59
Database content	61
Creating the database and tables	62
Part 2. Enterprise JavaBeans development	63
Chapter 5. EJB development environment	65
EJB page	66
Enterprise Beans pane	67
Types or Properties pane	68
Members pane	69
Source pane	69
Actions	70
Toolbar icons	70
Action from the Enterprise Beans pane	70
Actions for types	72
Actions for properties	72
Actions for members	73
Actions in the Source pane	73

Testing enterprise beans.....	73
Management of EJB groups and enterprise beans	74
Management of EJB groups	74
Management of enterprise beans	74
Team development	75
Team development using a shared repository	75
Team development for enterprise beans	76
Three approaches to team development of enterprise beans.....	77
Recommendations for team development	78
Chapter 6. WebSphere Test Environment	79
WebSphere Test Environment overview	80
Servlet engine	82
Persistent name server	82
JSP execution monitor	82
Data sources	82
Differences between WTE and WebSphere Application Server	83
Using the WebSphere Test Environment.....	83
Servlet engine	84
Persistent name server	86
JSP execution monitor	88
Data sources	89
Testing enterprise Java beans	91
EJB server configuration	91
Setting the EJB server properties	92
Starting EJB servers	93
EJB test client	94
Advanced features of the EJB test client	98
Testing servlets with EJBs	103
Summary.....	103
Chapter 7. Container-managed persistence entity beans	105
Persistence basics	106
EJB persistence terms and definitions	106
Life cycle of EJB instances	106
Developing a CMP bean with VisualAge for Java	108
Overview	108
EJB group	109
Creating a CMP bean	110
Understanding the generated types	116
Understanding the generated methods.....	117
Adding new methods to enterprise bean.....	118
Adding methods to the remote interface.....	119

Marking methods read-only.....	119
Customizing the home interface	120
Database schema and map.....	121
Generating the schema and mapping from the EJB group.....	122
Deployment.....	126
Adjusting deployment attributes	126
Generating deployment code	127
Test environment	128
Creating an EJB server configuration	128
Setting EJB server properties	129
Starting the servers.....	129
Using the EJB test client.....	130
Defining the bank model	132
Bank account	132
Transaction record	133
Business methods.....	134
Mixing CMP and BMP entity beans.....	136
Test the BankAccount and TransRecord beans	136
Summary	137
Pros and cons	137
Chapter 8. Bean-managed persistence entity beans.....	139
Bean-managed persistence.....	140
Why BMP?	140
Creating a BMP entity bean with VisualAge for Java	140
Creating the TransactionRecord entity bean.....	141
A quick tour of the generated classes.....	143
Adding and defining BMP fields	143
Changing the key class.....	144
A first look at the bean methods	144
Implementing the bean methods.....	146
Modifying the home and the remote interfaces.....	153
Testing the BMP entity bean.....	154
Summary	156
Pros and cons	156
Chapter 9. Session beans.....	157
Session bean basics	158
Session bean lifetime	158
Conversational state	158
Stateful session bean	159
Conversational session beans	159
Activation and passivation of session beans.....	160

Life cycle of a stateful bean.....	160
Stateless session bean	162
Non-conversational session beans	162
Life cycle of a stateless session bean.....	163
Implementing session beans.....	164
Session bean class.....	165
Remote interface.....	171
Home interface	172
Developing session beans with VisualAge for Java.....	174
Creating a session bean	174
Deploying the bean	177
Testing the transfer bean inside VisualAge for Java.....	178
Using session beans for database queries	180
Session bean with JDBC coding.....	181
Session bean with a Select bean	182
Session bean with a stored procedure	183
Enterprise Access Builder session bean	185
EAB command	185
EAB tools and runtime	186
EAB session bean tool	186
Summary	190
Chapter 10. Custom finder methods.....	193
What are custom finders methods?	194
How to write custom finder methods?	194
BeanFinderHelper interface	194
BeanFinderObject class	197
When to use which custom finder?	200
Mixing the approaches	200
Alternative to custom finders.....	201
Testing finder methods	201
Finder methods and transactions	202
Greedy enumeration	202
Lazy enumeration	203
Finders and complex associations.....	203
Summary	204
Chapter 11. Access beans	205
What are access beans?	206
JavaBean wrapper	208
Copy helper.....	209
Rowset	210
Access beans and associations	211

Creating an access bean with VisualAge for Java	212
Using access beans in client programs.	214
Summary	215
Chapter 12. Transaction management	217
What is a transaction?	218
Overview	218
The ACID properties	220
Transaction support in J2EE	220
Java Transaction Service	221
Java Transaction API	221
Transaction attributes	221
Method-level attributes	225
Isolation	227
Concurrency control	227
Data locking	228
Problems of concurrent transactions.	228
Isolation levels.	230
Isolation levels in JDBC	232
Mapping JDBC isolation levels to DB2	232
Transaction demarcation	232
Container-managed transaction	232
Bean-managed transaction.	233
Client-managed transaction	234
Recapitulation	235
Distributed transactions.	236
XA protocol	236
Two-phase commit	236
Transaction context.	237
Unsuccessful prepare	237
Real world scenario.	237
Distributed transaction with DB2.	238
New transaction support in WebSphere 3.5.3	238
Troubleshooting	238
Deadlocks	239
Find for update	239
Summary and guidelines	240
Guidelines for transactions	240
Guidelines for applications	241
Chapter 13. Client programming	243
Model-view-controller	244
How to access EJBs	245

The basics	245
Simple test application	248
Using a finder method	249
Simple servlet	250
Externalizing strings	253
Using access beans	254
Customer access bean	254
Simple test application with access bean	255
Using a rowset access bean for a finder method	256
Using a copy helper in a servlet	257
Using access beans in an applet or application	258
Java applets and security	261
Using a session facade to entity beans	261
Facade session beans	261
Facade session design	262
Servlet example using a session facade	264
Using an access bean for the session facade	270
Testing the servlet with the facade bean	271
Client comparison	271
Direct access	271
Access beans	272
Facade beans	273
Recapitulation of approaches	274
Summary	275
Chapter 14. Deployment to WebSphere Application Server Advanced .	277
Code deployment from VisualAge for Java	278
Preparation before going any further	278
Enterprise bean JAR files	278
Creating a deployed JAR file	279
Non-deployed JAR file	280
Editing the deployment descriptors using JetAce	281
Installation of the deployed code in WebSphere	283
Set data source for the EJB container	283
Installing a JAR file	283
Setting EJB properties	285
Installing a non-deployed JAR file	286
Installing session beans	286
Testing with the VisualAge EJB test client	287
Deploying and running the client applications	288
Create client JAR files	288
Java JDK and the class path	288
Test the applications	289

Test the servlets	289
Deployment using the XML configuration tool	290
Setting up security in WebSphere	291
Tracing and debugging	291
OLT tracing	292
Distributed Debugger	294
Debugging an application	295
Summary	296
Part 3. Advanced topics	297
Chapter 15. Advanced mapping for container-manager entity beans ..	299
CMP architecture	300
EJB model	300
Database schema	301
Mapping	305
Design and mapping approaches	307
Top-down	307
Bottom-up	309
Meet-in-the-middle	310
Mapping examples	311
Converters and composers	311
Using composers	312
Using converters	314
Entity model with advanced mapping	315
Setting up the structure	315
Mapping the beans	319
Using secondary tables	321
How to use secondary tables	321
Creating a two-table schema and mapping for the Customer bean	321
Summary	327
Chapter 16. Inheritance	329
Bank example	330
Inheritance overview	330
Mapping schemes for inheritance model	331
Characteristics of EJB inheritance	332
Characteristics of Java inheritance	333
EJB inheritance	333
Requirements for EJB inheritance	333
Limitations	334
Home interface specifics	334
Developing the inheritance hierarchy	334
Implementing the inheritance with VisualAge for Java	336

Setting up the structure	336
Creating the subclass beans	336
What is inherited?	337
Schema and map for a model with inheritance	338
Generating a single table database schema and map	338
Creating a root/leaf table model	341
Tailoring the methods	345
Testing the Checking and Saving beans	346
Java inheritance	348
Developing using Java inheritance	348
Summary	351
Chapter 17. Associations	353
Association support in VisualAge for Java	354
Reviewing the bank model	354
1:m association	355
Association editor	355
Generated methods and classes	357
Schema and mapping	358
Generating the deployed code	361
m:m association	361
Define the intermediate entity and two 1:m associations	362
Define the schema for an m:m association	363
Define the map for an m:m association	364
Generated methods for m:m association	365
Finder method for m:m association	365
Banking session bean	368
Create access beans for client programming	369
Entity properties	370
Testing associations with the EJB test client	371
Mandatory association	371
Deployment to WebSphere	372
Using XMLConfig for deployment	373
Testing the EJBs in WebSphere	375
Reverse engineering from an existing database	376
Importing existing tables into a schema	376
Creating an EJB model from the schema	377
Observations	378
Summary	379
Chapter 18. Client programming for inheritance and associations	381
Entity bean types with inheritance	382
Listing the type of bank accounts	382

Using access beans with inheritance	384
Programming with associations	384
Following associations with access beans	386
Using a custom finder method to traverse an association	388
Using associations in a servlet and JSP application	389
Servlet to retrieve customer and bank accounts	390
JSP to display customer and bank accounts	391
Servlet with simple bank transactions	392
JSP to display the result balances	394
Servlet with session bean	395
Running the advanced examples	397
WebSphere Test Environment	397
WebSphere Application Server	397
Summary	398
Chapter 19. Exception handling	399
Introduction	400
Checked and unchecked exceptions	400
Exception guidelines	401
Benefits	401
Exceptions and EJB	402
Application exceptions	402
Client view	404
System exception	404
Transaction exceptions	406
Enhancing standard EJB exceptions	408
Limitations	408
EJB 2.0	409
Summary	409
Chapter 20. Security for enterprise beans	411
Websphere security	412
Security concepts	412
Principal	413
WebSphere clients	413
Access control list (ACL)	413
Realm	413
Authentication	413
Authorization	414
Non-repudiation	414
Controlling information access	414
Security role	414
Methods	415

Method-level security	415
Security setting in VisualAge for Java.....	416
Security components	417
Security server	417
Implementing EJB security	418
Summary	418
Chapter 21. EJB modeling with Rational Rose.....	419
Preparation	420
VisualAge for Java	420
Bank scenario	420
Model in Rational Rose	420
Importing the model into an EJB group.....	421
Tailoring the model in VisualAge for Java.....	422
Summary	424
Part 4. EJB design guidelines and best practices	425
Chapter 22. EJB design considerations	427
When EJB.....	428
Isolate business logic components	428
Independence from database implementation	428
Multiple client access	428
Concurrent read and update	429
Access multiple data sources	429
Method-level security	429
Which EJB	429
Session bean	429
Entity	430
JavaBean	431
Design patterns.....	432
Model-view-controller	432
Session entity facade	434
Batch session bean.....	436
Command framework	436
Best design practices	437
Data access strategies	437
Logic splitting	437
The big picture: architecture	440
About the model	441
Logic	441
State management	441
Security	442
Transactions	442

Chapter 23. Best practices for EJBs	443
Transaction tips	444
Persistence tips	444
General tips	444
Entity associations	445
Implementation tips	447
Performance tips	449
Part 5. Appendixes	451
Appendix A. Setting up the environment	453
Installation instructions for products	454
Windows NT or Windows 2000	454
DB2 Version 7.1 Enterprise Edition	454
WebSphere Application Server Advanced Version 3.5	455
VisualAge for Java Enterprise Version 3.5	455
WebSphere Studio Advanced Version 3.5	456
Setting up the WebSphere execution environment	456
Configure the WebSphere administration server	456
Creating an application server with a Web application	460
Create Web application directories	464
Adding a servlet to the Web application	464
Start and stop of the application server	465
Setting up VisualAge for Java	466
JDBC driver	466
EJB development environment	466
Redbook project	466
Product support	467
Appendix B. Early information: deployment to WebSphere Version 4	469
VisualAge for Java Version 4	470
Account example	470
Generating the deployment JAR files	474
EJB 1.1 JAR file	475
WebSphere Application Server Version 4	475
Setup	475
Application Assembly Tool	475
Create an enterprise application	477
Create and add an EJB module	477
Create and add a Web module	478
Install enterprise application	482
Create client application	486
Running the account sample applications	487
Problems	489

Deployment of a regular EJB JAR file.....	489
A word of caution	490
Appendix C. Additional material	491
Locating the Web material	491
Using the Web material	492
System requirements for downloading the Web material	492
How to use the Web material	492
VisualAge for Java project	493
Running the examples	494
Related publications	497
IBM Redbooks	497
Other resources	498
Referenced Web sites	499
How to get IBM Redbooks	499
IBM Redbooks collections.....	499
Special notices	501
Abbreviations and acronyms	503
Index	505

Preface

This IBM Redbook provides detailed information on how to effectively use VisualAge for Java Enterprise for the development of applications based on the Enterprise JavaBeans architecture, and deployment of such applications to a WebSphere Application Server. This redbook is a companion book to the redbooks on IBM Patterns for e-business.

In Part 1 we introduce Enterprise JavaBeans as a part of Java 2 Enterprise Edition and cover the basic concepts and the architecture. We also introduce the IBM products that support this architecture.

In Part 2 we describe the IBM development and test environment for enterprise beans, and we create rather simple container-managed and bean-managed entity beans, session beans, finder methods, access beans, and client programs. We explain the basic transaction management facilities and describe how to deploy enterprise beans to a WebSphere Application Server.

In Part 3 we explore advanced facilities for mapping entity beans to database tables, including the extended IBM support for inheritance and associations. We describe how these advanced facilities are used in client programs and touch on exception handling, security, and modeling with Rational Rose.

In Part 4 we provide design guidelines and best practices when developing applications based on enterprise beans.

Throughout the book we provide examples based on a simple banking application with an underlying relational database.

The team that wrote this redbook

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

Ueli Wahli is a Consultant I/T Specialist at the IBM International Technical Support Organization in San Jose, California. Before joining the ITSO 17 years ago, Ueli worked in technical support at IBM Switzerland. He writes extensively and teaches IBM classes worldwide on application development, object technology, VisualAge products, data dictionaries, and library management. Ueli holds a degree in Mathematics from the [Swiss Federal Institute of Technology](#).

Dominique Faidherbe is a Consultant for the EMEA AIM Partner Technical Enablement Team in La Hulpe, Belgium. He has nine years of experience in object technology. He started in IBM as an Education Specialist for the Object Technology University and then moved to technical support for the San Francisco project, first as a Q&A Team Leader and then as a Consultant. His latest assignment is as a Consultant for WebSphere and VisualAge for Java, helping Business Partners and Systems Integrators. He holds an Engineering degree in Computer Science and Management from the [Faculte Polytechnique de Mons](#) in Belgium.

Akis Laftsidis is a Technical Sales Specialist in the IBM Software Group in Stockholm, Sweden. He has five years of experience in application development and design, using object- and component-oriented techniques. Akis started in IBM Global Services where he worked as a consultant for three years. Most recently, he was working with software pre-sales in Application Integration Middleware, dedicated to the WebSphere and VisualAge product family. Akis holds a master's degree in Electronic Engineering and Computing from the Aristotle University of Thessaloniki in Greece.

Karthikeyan Ponnalagu is a Software Engineer at the IBM Software lab in Bangalore, India. He has two years of experience in application development. He holds a degree in Electronics and Communication Engineering. His areas of interest include object-oriented languages, distributed system architecture, and Linux kernel programming.

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

Joaquin Picon and his two teams of residents produced the previous redbooks on IBM's EJB technology (SG24-5429 and SG24-5754).

Special notice

This publication is intended to help application developers using VisualAge for Java to create distributed object applications for WebSphere Application Server based on the Enterprise JavaBeans architecture and specification. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge for Java Enterprise Edition and WebSphere Application Server Advanced Edition. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge for Java and WebSphere Application Server for more information about what publications are considered to be product documentation.

IBM trademarks

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

e (logo)® 	Redbooks
IBM ®	Redbooks Logo 
AIX	AS/400
CICS	DB2
Domino	MQSeries
Notes	OS/2
OS/390	S/390
SP1	SP
SecureWay	VisualAge
WebSphere	Wizard

Comments welcome

Your comments are important to us!

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

- ▶ Use the online **Contact us** review redbook form found at:
ibm.com/redbooks
- ▶ Send your comments in an Internet note to:
redbook@us.ibm.com
- ▶ Mail your comments to the address on page ii.



Part 1

Introduction

In this part we introduce Enterprise JavaBeans as part of the Java 2 Enterprise Edition (J2EE). We describe the concepts and architecture of enterprise beans and introduce entity and session beans, as well as the responsibilities of an EJB container.

Then we look at the IBM products for EJB development and deployment, VisualAge for Java and WebSphere Application Server.

Finally we introduce the ITSO bank example that will be used throughout the book for illustration of the concepts and tasks associated with development and deployment of enterprise beans.



1

Enterprise JavaBeans and J2EE

In this chapter we introduce Enterprise JavaBeans in the context of Java 2 Enterprise Edition (J2EE), an end-to-end technical platform which enables developing, deploying and managing multi-tier, server-centric applications.

Server-side component architecture

Since its introduction a few years ago, Enterprise JavaBeans (EJB) technology has gained momentum among platform providers and enterprise development teams. This is because the EJB server-side component model simplifies development of middleware components that are transactional, scalable, and portable. Enterprise JavaBeans servers reduce the complexity of developing middleware by providing automatic support for middleware services, such as transactions, security, and database connectivity.

Background

During the early 90s, traditional enterprise information system providers began responding to customer needs by shifting from the two-tier, client-server application model to more flexible three-tier and multi-tier application models.

The new models separated business logic from system services and the user interface, placing it in a middle tier between the two. The evolution of new middleware services—transaction monitors, message-oriented middleware, object request brokers, and Web Application Servers—gave additional impetus to this new architecture. In addition the growing use of the Internet and intranets for enterprise applications contributed to a greater emphasis on lightweight, easy to deploy clients that run in Web browsers.

Multi-tier design simplifies developing, deploying, and maintaining enterprise applications. It enables developers to focus on the specifics of programming their business logic, relying on various backend services to provide the infrastructure, and client-side applications (both standalone and within Web browsers) to provide the user interaction.

Once developed, business logic can be deployed on servers appropriate to existing needs of an organization. However, no standard component architecture existed for the middle tier, and forced developers to focus on plumbing details specific to the particular mix of platform, operating system, and middleware services. This limited developers to deploy a single application on a wide variety of platforms, or to readily scale applications to meet changing business conditions.

Why EJBs?

Numerous Web sites are up and running using Java without any EJB technology. Developers have been using servlet/JSP models and managing transactions themselves using commit, rollback functionality that is built-in to JDBC without the help of application servers.

But when doing so, the application developers are confronted with many challenges. Some of the most important ones are managing concurrency, persistence and transactions. As a result, the developers have to either develop proprietary code or buy supporting frameworks.

These problems are solved by using enterprise beans. The use of enterprise beans allow developers to focus on the business logic and release them from coding infrastructure and middleware logic, and developers become more productive and efficient.

As with most other technologies, enterprise beans do not provide the unique solution to all problems. Using enterprise beans has advantages and disadvantages. However, the advantages overcome the disadvantages especially for more complex applications that require a sophisticated robust and distributed persistent model.

Once again, not every application environment may benefit from using enterprise beans. To help you decide whether this technology is appropriate, this section provides some reasons to consider using it.

Object distribution

When using Enterprise JavaBeans, distributed objects are used for building an enterprise-scale system. In short, this means that the parts of your program can be deployed to as many different physical machines and in as many separate OS processes as appropriate to achieve the performance, scalability and availability goals of your system.

Object persistence

Making an object persistent means that its state, the values of its variables, can be preserved across multiple invocations of a program that references that object. In most cases the state of a persistent object is stored in a relational database.

Unfortunately, objects and relational databases differ a lot from each other. Relational databases have limited modeling capabilities, such as object inheritance and encapsulation, compared to Java. Additionally, SQL data types

do not exactly match Java data types, leading to conversion problems. All these problems are solved when using CMP entity beans.

Independence from database schema

EJB technology enables a clear separation of business logic from database access. The business logic is independent of the database schema and can be deployed into organizations with different or changing database schemas.

Transaction management

Concurrent access to shared data can be one of the biggest headaches to a developer. The consideration of all the related issues as database locking, concurrency or even loss of data integrity can lead to the creation of highly complex schemes, managing access to shared data at the database level.

Enterprise beans automatically handle these complex threading and simultaneous shared data issues. As mentioned previously, the EJB container provides all the necessary transaction services to enterprise beans for control access to back-end data in a transactional manner.

Apart from that, many applications require the ability to access multiple data sources. For instance, a program may use data in both a middle-tier DB2 database and a mainframe CICS or IMS system accessible through MQSeries. The key is that some applications require that this access is fully transactional – that data integrity is maintained across the datasources.

For example, an application may demand that entering a user order consists of storing the detailed order information in a DB2 database and simultaneously placing a shipment order with a CICS system through MQSeries. If either the database update or the MQ enqueueing fails, then the entire transaction should roll back.

In the past, the only choices with which to build systems like these were transaction monitors, such as Encina and CICS, which used non-standard interfaces and required development in languages like COBOL, C or C++.

Enterprise beans support multiple concurrent transactions with commit and rollback capabilities across multiple data sources in a full two-phase commit-capable environment.

Platform independence

The majority of companies give a lot of importance to being independent from platform, vendor, and application server implementation. The EJB architecture,

which is an industry standard component architecture, allows a company to base its architecture on portable components.

Enterprise beans can be developed using any EJB development tool that complies to the standards. These beans can then be deployed on any application server that complies to the same standards on almost any available platform.

This gives a freedom of choices to companies, when selecting their development and runtime environment. There is no dependency on a certain application server or development tool, allowing a company to always select the software that best meets its needs.

Scalable environment

Some of the most critical factors when designing an enterprise-scale system are scalability, availability and failover support. EJB applications can scale from a small single-processor to a large multi-processor environment or to a mainframe environment, all without any modification.

This scalability can be achieved without sacrificing ease of development or standardization. Enterprise JavaBeans can provide this kind of highly scalable, highly available system by utilizing the following features:

- ▶ *Object caching and pooling:* EJB containers automatically pool enterprise bean instances, reducing the amount of time spent in object creation and garbage collection. This results in more processing cycles being available to do real work.
- ▶ *Workload management:* EJB servers can be easily scaled in a clustered environment by grouping multiple EJB servers into server clusters. When clients access enterprise beans deployed in these server clusters, the application behaves as the enterprise beans were deployed in a single EJB server. Workload is distributed across the EJB servers in the server clusters.

Secure access

Certain types of applications have security restrictions that have previously made them difficult to be implemented in Java, for instance, applications that must restrict access to a subset of business methods or business data.

Prior to Enterprise JavaBeans, there was no way to restrict access to an object or method by a particular user. Also restricting access at the database level, and then “catching” errors thrown at the JDBC level, or by restricting access at the application level by custom security code was very difficult and complicated.

However, enterprise beans now allow method-level security. Users and user groups can be created and can be granted or denied access rights to any bean or method.

Middle-tier architecture

Very often companies consider their application software, particularly the business rules and data structures that make up the business logic, as corporate assets. Therefore, they are concerned on protecting these assets from the public Internet.

Enterprise beans enable a company to use a middle-tier architecture so that presentation logic can be separated from business logic. This separation makes possible the use of a second firewall between these two different layers for higher isolation of all application components that contain business logic.

Development roles

EJB allows the right person to do the right job. Business developers can focus on implementing components with business functions, rules and workflow. The application designers work on assembling the various components provided by the business developers. And finally, the deployer is responsible for the installation of the application on the server.

Multiple client types

Often, a single application will have multiple client types that need access to the same set of information. For instance, an application may have a Web-based front end for external customers, and a more complete Java application front end for internal users. Traditionally, this problem has been solved by writing two versions of the same application that share the same data.

However, this is not efficient either in programming time or utilizing the database, if multiple database locks could be held at one time. The EJB solution to this problem is to embed common data and business logic in a single set of EJBs that can be accessed by different client types.

Integration with CORBA

Enterprise beans are built on technology that is a combination of Java Remote Method Invocation (RMI) and Component Object Request Broker Architecture (CORBA). Clients access an enterprise bean using RMI over IIOP. This allows pure CORBA clients to access enterprise beans as EJB clients.

Java 2 Platform, Enterprise Edition

The Java 2 Platform, Enterprise Edition (J2EE) is a robust suite of middle-ware application services for server side application development. J2EE is an extension of the Java 2 Platform, Standard Edition (J2SE).

J2EE makes all Java enterprise APIs and functionality available and accessible in a well integrated fashion. This helps in simplifying complex problems in the development, deployment, and management of multi-tier, server-centric enterprise solutions. Let's look at the technologies included with J2EE.

Enterprise JavaBeans (EJB)

EJB defines how server-side components are written and provides a standard architectural contract between the components and the application servers and containers that manage them. The EJB specification provides a solution for a clear separation of the business logic and the intricacies of dealing with persistency, transactions, and other middleware-related services.

Java Remote Method Invocation and RMI-IIOP

RMI is a mechanism for invoking methods remotely on other machines. EJB relies on Java RMI as a communications API between components and their clients. Sun Microsystems with IBM and others has recently developed a more portable version of RMI, which uses the Object Management Group's (OMG) Internet Inter-ORB protocol. IIOP is necessary for J2EE deployments to be interoperable with CORBA systems.

Java Naming and Directory Interface (JNDI)

JNDI is a standard for naming and directory services. In EJB-based applications when the client code requests access to an EJB component, JNDI is used to locate and retrieve the component to service the client. JNDI enables writing portable directory and naming service code that works with multiple directory services, such as LDAP and CosNaming.

Java Database Connectivity (JDBC)

JDBC provides uniform access to a wide range of relational databases. JDBC enables Java programmers to represent database connections, SQL statements and retrieving results in a portable way. JDBC 2.0 has built in support for database connection pooling.

Java Transaction API (JTA) and Java Transaction Service (JTS)

JTA is a high level transaction API that allows applications and J2EE servers to manage transactions. JTS specifies the implementation of a transaction manager, which supports JTA and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 specification at the level below the API.

Java Messaging Service (JMS)

By combining Java technology with enterprise messaging, the JMS API provides a new, powerful tool for solving enterprise computing problems. The JMS API improves programmer productivity by defining a common set of messaging concepts and programming strategies that will be supported by all JMS technology-compliant messaging systems.

Java servlets and JavaServer Pages

A servlet can be thought of as an applet that runs in the server side. Servlets provide a component-based, platform independent method for building Web-based applications without the limitations of CGI programs.

JSP technology is an extension of servlet technology created to support authoring of HTML and XML pages. It enables combining fixed template data with dynamic content data. It is a good alternative for servlets.

Servlets and JSPs are suited for simple request/response Web models. We recommend that applications developed with EJB follow the model-view-controller (MVC) design pattern. This pattern defines an architectural three-way split for an application, separating data elements (the *model*) from the data presentation (the *view*) and from the manipulation of the data into the presentation (the *controller*). Refer to Chapter 22, “EJB design considerations” on page 427.

The J2EE relies heavily on JSPs as the view part of the MVC pattern to bridge the gap between EJB and HTML code, placing JSPs at the key boundary between data and presentation. Similarly, servlets are used as the controller and EJBs as the model part of the MVC pattern, respectively.

Java IDL

Java Interface Definition Language (IDL) is an implementation of the CORBA specification and enables inter-operability and connectivity with heterogeneous objects. It is basically an object request broker provided with JDK 1.2. The Java IDL enables distributed Web applications to transparently invoke operations on remote network services using the industry standards IDL and IIOP from OMG.

Java Activation Framework (JAF)

JAF enables developers to determine the type of an arbitrary piece of data, to encapsulate access to the data, to discover the functional operations available on it, and to instantiate the appropriate bean to perform the operation(s). It also enables you to dynamically register types of arbitrary data and actions associated with particular kinds of data. Additionally, it enables a program to dynamically provide or retrieve JavaBeans that implement actions associated with some kind of data.

Java Mail

The Java Mail API provides a platform- and protocol-independent framework to build Java based mail and messaging applications. It basically allow the applications to have e-mail capabilities. Java Mail depends on Java Activation Framework to encapsulate message data and to handle interactions.

J2EE architecture overview

Figure 1-1 shows an overall view comprising the different J2EE technologies.

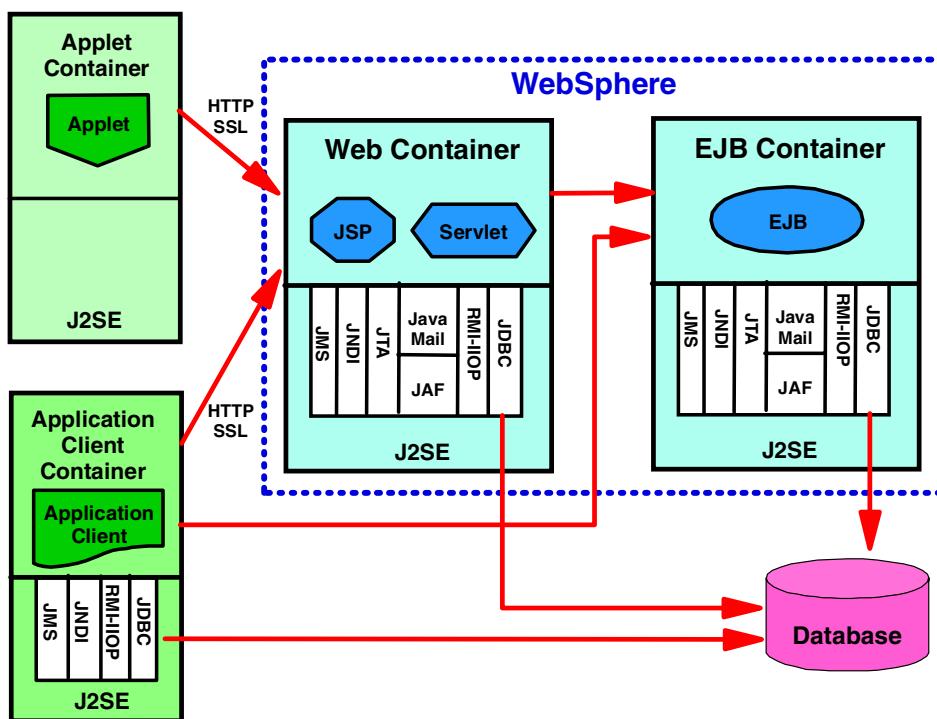


Figure 1-1 J2EE architecture diagram

Web containers and EJB containers

Containers are basically runtime environments that provide components with specific services. For example, as shown in Figure 1-1, Web containers provide runtime support to clients by processing requests through invoking JSPs and servlets and returning results from the components to the client. Similarly EJB containers provide automated support for transaction and state management of EJB components, as well as look up and security services.

EJB specifications

In this section, we describe the features and limitations of the different EJB versions that have been specified by Sun and the Java community.

EJB 1.0

Here are the features and limitations of EJB 1.0.

Features

EJB 1.0 introduced the following features:

- ▶ Enterprise bean instances are created and managed at runtime by a container.
- ▶ An enterprise bean can be customized at deployment time by editing its environment properties.
- ▶ Client access is mediated by the container and the EJB server on which the enterprise Bean is deployed.
- ▶ Flexible component model.
- ▶ Support for component distributions and security.
- ▶ Stateless and stateful session beans are defined and must be supported.
- ▶ Container-managed persistence and bean-managed persistence entity beans are defined and are not mandatory in this version.
- ▶ For transaction support either the original javax.jts package, or the new javax.transaction package can be used.

Limitations

Though some of the features are attractive, there are some limitations as stated below:

- ▶ Entity bean support is not mandatory.

- ▶ Java RMI-IIOP support is not mandatory, which eludes interoperability with other heterogeneous components.
- ▶ The deployment descriptor is not available in a text format.
- ▶ Requires a separate deployment file for each enterprise bean, causing large applications composed of many beans getting slowed down.
- ▶ Interoperability between containers is not defined.
- ▶ No standard container API.

EJB 1.1

EJB 1.1 attempts to provide a high degree of application compatibility for enterprise beans that were written for the EJB 1.0 specification.

Differences over 1.0

EJB 1.1 addresses many of the limitations and loop holes found in EJB 1.0. The most significant changes are listed here:

- ▶ Entity bean support, both container- and bean-managed persistence, is required.
- ▶ Java RMI-IIOP argument and reference types must be supported. That is the client API must support the Java RMI-IIOP programming model for portability, but the underlying communication protocol can be anything.
- ▶ The `javax.ejb.deployment` package has been dropped in favor of a XML based deployment descriptor.
- ▶ Declarative security authorization (access control) has changed to be more role driven.
- ▶ Isolation levels are now managed explicitly through JDBC (BMP), the database or other vendor specific mechanisms.
- ▶ The bean-container contract has been enhanced to include a default JNDI context for accessing properties and resources, for example, JDBC and JMS.
- ▶ The basic EJB roles have been expanded and redefined to better separate responsibilities involved in the development, deployment, and hosting of enterprise beans.
- ▶ Allows using `java.lang.String` as a primary key type.
- ▶ Allows a session bean instance to be removed upon a time-out while the instance is in the passivated state.
- ▶ Allows enterprise beans to read system properties.

The EJB 1.0 enterprise bean code has to be changed or recompiled to run in an EJB 1.1 container, in the following situations:

- ▶ An enterprise bean that uses the javax.jts.UserTransaction interface needs to be modified to use the new javax.transaction.UserTransaction.
- ▶ An enterprise bean written to the EJB 1.0 specification has to be modified to use the getCallerPrincipal() and isCallerInRole(String roleName) methods instead of the deprecated getCallerIdentity() and isCallerInRole(Identity) methods to work in all EJB 1.1 containers.
- ▶ An enterprise bean with container-managed persistence written to the EJB 1.0 specification has to be recompiled to work with all EJB 1.1 compliant containers, because the required return value of ejbCreate(...) is different in EJB 1.1 than in EJB 1.0.
- ▶ An entity bean in EJB 1.0, whose finders do not define the FinderException in the methods' throws clauses, must be changed. EJB 1.1 requires that all finders define the FinderException.
- ▶ In EJB 1.1, an entity bean must not use the UserTransaction interface.
- ▶ The enterprise bean in EJB 1.0 uses the UserTransaction interface and implements the SessionSynchronization interface at the same time, which is not allowed in EJB 1.1.

Limitations

In spite of many added features over EJB 1.0, EJB 1.1 still faces the following limitations:

- ▶ Data modeling capability is very simple.
- ▶ Mapping of the CMP bean to the database schema is of limited use.

EJB 2.0

EJB 2.0 is the latest version of the specification. At the time of this writing, EJB 2.0 is a draft specification. The most important changes in the specification are those made to container-managed persistence (CMP) and the introduction of a completely new bean type, the message-driven bean.

Features supported in EJB 2.0

EJB 2.0 provides these enhancements:

- ▶ Integration of EJB with JMS.
- ▶ Message-driven beans.
- ▶ Implement additional business methods in the home interface which are not specific for bean instance.

- ▶ EJB query language (EJB QL), which enables the ability to support searches based on the object schema instead of data schema.

Changes in EJB 2.0

EJB 2.0 contains these changes over EJB 1.1:

- ▶ The new CMP component model is radically different from the old CMP model, because it introduces an entirely new participant, the *persistence manager*.
- ▶ There is a completely new way of defining container-managed fields, as well as relationships with other beans and dependent objects.
- ▶ The introduction of the MessageDrivenBean class (the message bean). It provides a component model for the enterprise beans acting as JMS clients, allowing them to be deployed in the rich and robust environment of the EJB container system.

General restrictions of EJB

EJB development places some restrictions on the developers for better component management and easy service. They are discussed as follows:

- ▶ An enterprise bean must not use read/write static fields. Therefore, we recommend that all static fields in the enterprise bean class are declared as *final*.
- ▶ An enterprise bean must not use thread synchronization primitives to synchronize execution of multiple instances.
- ▶ An enterprise bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard.
- ▶ An enterprise bean must not use the `java.io` package to attempt to access files and directories in the file system.
- ▶ An enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicasting.
- ▶ The enterprise bean must not attempt to query a class to obtain information about the declared members that are not otherwise accessible to the enterprise bean because of the security rules of the Java language.
- ▶ The enterprise bean must not attempt to create a class loader, obtain the current class loader, set the context class loader, set the security manager, create a new security manager, stop the JVM, or change the input, output, and error streams.

- ▶ The enterprise bean must not attempt to set the socket factory used by ServerSocket, Socket, or the stream handler factory used by URL.
- ▶ The enterprise bean must not attempt to manage threads.
- ▶ The enterprise bean must not attempt to directly read or write a file descriptor.
- ▶ The enterprise bean must not attempt to obtain the security policy information for a particular code source.
- ▶ The enterprise bean must not attempt to load a native library.
- ▶ The enterprise bean must not attempt to gain access to packages and classes that the usual rules of the Java programming language make unavailable to the enterprise bean.
- ▶ The enterprise bean must not attempt to define a class in a package.
- ▶ The enterprise bean must not attempt to access or modify the security configuration objects (Policy, Security, Provider, Signer, and Identity).
- ▶ The enterprise bean must not attempt to use the subclass and object substitution features of the Java serialization protocol.
- ▶ The enterprise bean must not attempt to pass *this* as an argument or method result. The enterprise bean must instead pass the result of SessionContext.getEJBObject() or EntityContext.getEJBObject().

IBM products and J2EE

Table 1-1 shows the support in VisualAge for Java and WebSphere Version 3.5.3 for the various J2EE technologies.

Table 1-1 Support of J2EE technologies by IBM products

Technology	Version	Comments
JDK	1.2	Fully supported
JDBC	2.0	WebSphere adds a unique prepared statements cache implementation.
JMS	1.0.1	IBM MQSeries supports native JMS and can be used by WebSphere.
JNDI	1.2	WebSphere 3.5 supports most of the JNDI 1.2. It uses CosNaming as a service provider.

Technology	Version	Comments						
EJB	1.0	<p>WebSphere supports EJB 1.0 plus:</p> <ul style="list-style-type: none"> ▶ Container-managed persistence (CMP) ▶ Uses <code>javax.transaction.UserTransaction</code> ▶ Restricts bean-managed transactions to session beans (as required by EJB 1.1 specification) <p>WebSphere does not currently support these EJB 1.1 features:</p> <ul style="list-style-type: none"> ▶ XML deployment descriptors ▶ Use of JNDI within EJB environment: <ul style="list-style-type: none"> – Lookup of home interfaces via EJB references and links defined in the EJB's environment – The new <code>HomeHandle</code> class and the associated API changes to <code>EJBHome</code> ▶ Use of the <code>javax.security.Principal</code> interface <p>WebSphere and VisualAge for Java extend the EJB specification with the following features:</p> <table> <tr> <td>Access beans:</td> <td>Simplify client application using EJBs</td> </tr> <tr> <td>Association:</td> <td>Support relationship between CMP beans</td> </tr> <tr> <td>Inheritance:</td> <td>Support polymorphism and reuse</td> </tr> </table>	Access beans:	Simplify client application using EJBs	Association:	Support relationship between CMP beans	Inheritance:	Support polymorphism and reuse
Access beans:	Simplify client application using EJBs							
Association:	Support relationship between CMP beans							
Inheritance:	Support polymorphism and reuse							
Servlet	2.2	Fully supported						
JSP	1.1	It also has a high quality tool support through WebSphere Studio.						
JTA and JTS	1.0	Supported with (two phase commit) distributed transactions						
RMI-IIOP	1.0	Fully supported						
JAF	1.1	Fully supported						
Java Mail	1.1	Not Supported						
XML DOM/SAX		Fully supported						
HTTP	1.1	Fully supported						

Adopting EJB technology

The EJB technology is relatively new compared to CORBA and RMI. However, it has gained a lot of attention from many organizations, institutes and companies. Today, there are many parties that are committed to this new open standard and they are doing big investments in this technology by improving and advancing the EJB specifications.

Adopting this technology is a step forward to doing the same things with a more structured and standard way, and there should not be any hesitation in the relatively longer skill developing cycle compared to other traditional approaches. This is an investment that will be returned in many ways.



EJBs: Basic concepts and architecture

This chapter starts with a short description of some well-known distributed object technologies, such as CORBA and RMI, and presents some of the major weaknesses that developers encountered when using them. It then shows how the technology of Enterprise JavaBeans was created, in order to combine the best of CORBA and RMI, and provide an improved and more sophisticated programming model.

In the following section, there is an introduction to the EJB model and the two different types of enterprise beans, entity and session. These are described briefly and compared at the end, by giving some general guidelines to the developer.

After the basic concepts about enterprise beans, the architecture of the EJB runtime environment is presented, in respect to the EJB server and EJB container responsibilities.

EJB basics

In this section we explain how the EJB technology was invented from CORBA and RMI, as well as the basic concepts of using EJBs.

Distributed component-oriented applications

Distributed computing is the solution to highly available applications that have to be reached from remote locations in an efficient way. Customers, business partners, suppliers and other parties have access to the same business logic and data at any time and any place. This business logic and data can be located on different systems. As the distributed computing evolved, new technologies were introduced to allow objects to run on one physical machine and then to be used by client applications on different machines.

Today a number of distribution technologies are available. Some of the most known are the Object Management Group (OMG) Component Object Request Broker Architecture (CORBA), Sun Java Remote Method Invocation (RMI) protocol, as well as Microsoft Distributed Component Object Model (DCOM). In the Java world, only the first two technologies stand out as especially important. In the next paragraph, we describe briefly CORBA and RMI and show how they relate to the more recent technology of EJB.

CORBA overview

CORBA was developed by a consortium of companies (the Object Management Group) during the early 1990s to provide a common, language- and vendor-neutral standard for object distribution. CORBA as an architecture has been well accepted and successfully used in many projects.

The CORBA architecture is built around a special layer, the Object Request Broker (ORB), that facilitates communication between clients and objects. The ORB is responsible for handling the object requests from a client and passing over the parameters from method invocations.

Low-level communication between different object spaces (ORBs) is done by using the Internet-Inter ORB Protocol (IIOP). By using this standard protocol, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network can interoperate with a CORBA-based program from the same or another vendor, on any other computer, operating system, programming language, and network.

Apart from the ORB, there are two other key building blocks in the CORBA model, the Interface Definition Language (IDL), which normalizes the differences caused by language or operating system dependencies; and the CORBA services, which provide standard ways for CORBA objects to interact like naming and transaction.

The greatest advantages of using CORBA is that it is a standard interface that enables interoperability between different vendor's products and that CORBA is language neutral. CORBA clients and servers can be written in a variety of computer languages, including Java, C++, C, Smalltalk and Ada. This is possible by implementing remote interfaces for the CORBA distributed objects in IDL.

But when using CORBA to build distributed systems in Java, the development effort is higher, because many parts of the system have to be implemented in two languages: IDL and Java.

Additionally, the development tools and runtime environment for CORBA applications can also be expensive and may not fully implement the CORBA services. All these, in combination with the fact that developers started to look for simpler solutions, raised the interest in Java RMI.

RMI overview

Remote Method Invocation (RMI) enables the programmer to create distributed Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts.

A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the naming service or by receiving the reference as an argument or a return value. RMI uses object serialization to marshal and demarshal parameters, something that allows RMI programs to pass by value objects so that the server can operate on a local copy. Another advanced feature of RMI is the distributed garbage collection, a feature that is not available in CORBA today.

However, there are some known deficiencies of RMI, such as multiple-language support as well as all of the services that CORBA supports. For instance, RMI supports a naming service, but no transactions or persistence services as CORBA does. Java RMI does not even contain any provision for application level security. Also, the configuration of RMI ports in firewalls can be very difficult. All these limitations led to the combination of the two technologies.

The solution

The two previous sections showed that neither CORBA nor Java RMI were sufficient enough to address all the issues that a Java developer was facing, when building enterprise applications. Although these two technologies have some holes, they can complement each other in a unique way.

RMI over IIOP

RMI over IIOP (RMI-IIOP) combines the best features of RMI with those of CORBA. Like RMI, RMI-IIOP allows developers to use only Java. Developers do not have to develop in both Java and IDL. RMI-IIOP allows developers to build classes that pass any serializable Java object as remote method argument or return value. By using IIOP as communication protocol, RMI-IIOP applications are interoperable with other CORBA applications. The synthesis of these two technologies results to a unique combination of power and ease of use, the *Enterprise JavaBeans*.

Enterprise JavaBeans

Two years ago, a new technology was born, the Enterprise JavaBeans. Enterprise JavaBeans provided a way to improve the existing CORBA model by enhancing it with some of the best Java RMI features. The EJB model enables the Java developer to produce pure distributed applications in a simple manner. The key concepts of using enterprise beans are:

- ▶ EJBs are found or created by using an object factory that is inherited from CORBA. An EJB developer creates a Java interface, the home interface, which defines the ways in which remote objects are created or found. The EJB factory, which implements these interfaces, is called *EJBHome*. Clients locate EJB homes through the Java Naming and Directory Service (JNDI).
- ▶ EJBs are accessed through a simple Java interface, the *remote interface*. The remote interface is inherited from Java RMI and provides to the programmer all the externally accessible methods of the remote object. In the EJB world, it is the *EJBObject* that implements the remote interface, and allows the client to use the business logic that the remote object implements.
- ▶ EJBs use RMI-IIOP. This means that there is a well-defined, standard mapping of EJB interfaces to CORBA IDL. This assures the interoperability between different systems that implement EJB servers or CORBA systems. This also provides firewall support to EJB applications.
- ▶ There are standard ways for building persistent EJBs, both through vendor-provided frameworks (the container-managed persistence model, or CMP), and through a user-developed persistence mechanism (the bean-managed persistence model, or BMP).

- ▶ There is a standard transaction model for EJBs offered through the Java Transaction API (JTA). The underlying transaction service allows an application to update data in multiple data sources within a single (distributed) transaction.
- ▶ EJBs are built upon a security model that allows the person deploying the EJB to determine what access should be granted to whom at an EJB or method level.
- ▶ An EJB is a component that implements business logic in a distributed enterprise application. An EJB container is where EJBs reside. The EJB container is responsible for making the EJBs available to the client. EJBs are deployed into EJB containers and run on Enterprise Java Servers (EJS).
- ▶ There are two types of EJBs that developers can build: *session beans* and *entity beans*.

Figure 2-1 shows the different types of enterprise beans, as well as the bean hierarchy.

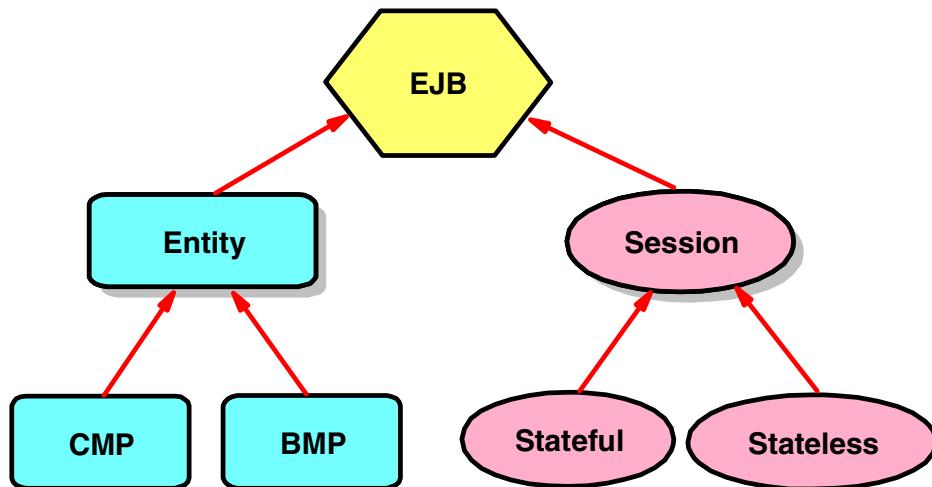


Figure 2-1 Enterprise bean hierarchy

JavaBeans versus Enterprise JavaBeans

JavaBeans is the component model for Java. Each JavaBean has properties, methods and events. Enterprise JavaBeans also describes among other things a component model. However, these two component models are not identical.

JavaBeans are intended to be reusable software components that can be visually manipulated in a builder tool. Therefore, the JavaBeans specification

describes in detail the APIs for connecting beans to each other through the event model, where one bean acts as an event listener and the other as an event source. Bean customization is done at assembly time by using properties. JavaBeans do not require a special container. They can exist inside a Java Virtual Machine.

Enterprise JavaBeans are strongly associated with a services framework. No event model is used by Enterprise JavaBeans. Customization is done at runtime by using a deployment descriptor. Enterprise JavaBeans do require an EJB container, in which they are deployed. This container should provide all the services defined by the framework and it must reside within an enterprise Java server.

So, the two component models are used in different ways. The JavaBeans model supports application assembly in a builder tool, while the Enterprise JavaBeans model supports a distributed object model. JavaBeans do not have their own container, so trying to locate them on a remote physical machine is theoretically impossible. Non-graphical server-side JavaBeans (for example, RMI server beans) could be also used instead of Enterprise JavaBeans, but that would require the implementation of an enterprise Java server framework.

Enterprise Java Server

The Enterprise Java Server (EJS) is that part of the application server that hosts EJB containers. It can host one or multiple containers. Containers are transparent to the client—there is no client API to manipulate the container, and there is no way for a client to tell in which container an enterprise bean is deployed.

EJS provides the runtime environment for one or more EJB containers. Enterprise servers manage low-level system resources, allocating resources to the containers as they are needed. However, bear in mind that the EJB container and the Enterprise Java Server are not clearly separated constructs. The EJB specification only defines a bean-container contract and does not define a container-server contract.

EJB container

An EJB container is a system that functions as a runtime environment for enterprise beans by providing all the primary services that are needed for bean management. All the services are well defined within the framework. The framework is implemented through the bean callback methods.

These methods are purely for system management purposes and they are called only by the container when it is interacting with the deployed beans. All the interaction takes place in a transparent way to the bean developer and also to the client. Some of the primary tasks that the container performs are connecting clients to beans, managing a bean's life cycle, providing persistence, performing transaction coordination, as well as security.

- ▶ The EJB container houses the enterprise beans and make them available to clients. A client can invoke an enterprise bean by looking up the name of the bean in a centralized naming space called Java Naming and Directory Interface (JNDI). After an enterprise bean is requested, the container will instantiate the bean (if not earlier) and deliver a remote interface to the client. Then the client can call any of the bean methods by using the remote interface. If the bean instance is not used for a certain period of time, the container will passivate the bean and activate it again whenever appropriate.
- ▶ The container is also responsible for managing the persistence of a bean by synchronizing the state of the bean instance in memory with the respective record in the data source. Concurrent access from multiple clients is managed through the transaction services. The container manages the transaction processing on behalf of the EJB beans.
- ▶ The EJB container provides a security domain to enterprise beans. The container is responsible for enforcing the security policies defined at the deployment time whenever there is a method call.

For a detailed description of the container services see “EJB container responsibilities” on page 33.

Figure 2-2 shows the runtime environment for enterprise beans.

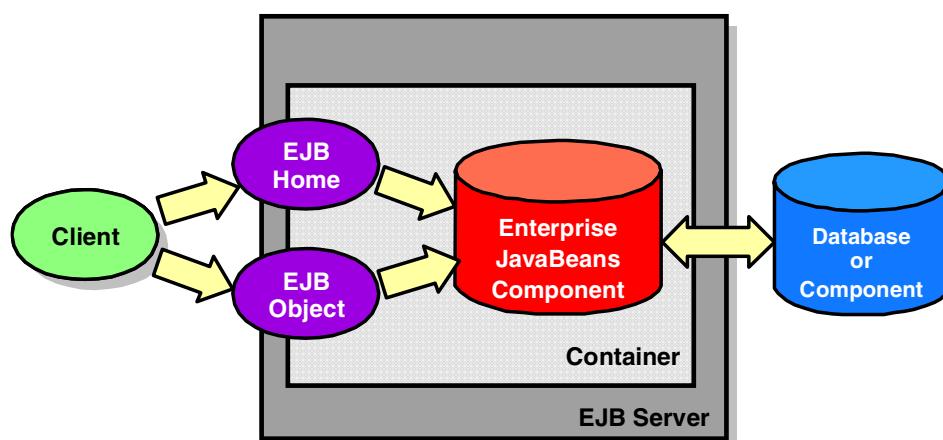


Figure 2-2 EJB server and EJB container

Let's now see the two different types of enterprise beans that can be deployed in a container, as defined by the EJB 1.0 specification.

Entity bean

Entity beans are used to represent permanent data and provide associated methods to manipulate that data. In the most common case, the permanent data is stored in a data source, such as a relational or object database. In more complex situations, the permanent data can result from the invocation of an application, stored procedure, or even the execution of a CICS transaction. In most cases, an entity bean must be accessed in some transactional manner. Instances of an entity bean are unique, and they can be accessed by multiple users.

For example, the information about a bank account can be encapsulated in an entity bean instance. An account enterprise bean might contain an account ID, a customer ID, and a balance as its main fields.

Depending on the way the persistence is managed, there are two flavors of entity beans.

Container-managed persistence (CMP)

Entity beans that delegate their persistence to their EJB container are called container-managed persistence (CMP) entity beans.

With CMP entity beans, you do not have to know which source is used to provide the persistent state of the bean. You just have to specify which fields are persistent. All the required JDBC code for accessing the database is generated for you. Therefore, there is absolute portability and the EJB developer can focus on the business logic.

For more details on container-managed entity beans see Chapter 7, “Container-managed persistence entity beans” on page 105.

Bean-managed persistence (BMP)

Entity beans that manage their own persistence are called bean-managed persistence (BMP) entity beans.

With a BMP entity bean, the EJB developer manages the persistent state of the bean by coding database calls or any type of access to permanent storage. It is the developer's responsibility to save and restore the state of the bean when called by the container through the ejbLoad and ejbStore methods—these are

the callback methods mentioned in the EJB container section. Most times, the developer uses JDBC for coding the persistence logic; however, other techniques can be also used, such as SQLJ or CICS transactions.

For more information on bean-managed entity bean see Chapter 8. “Bean-managed persistence entity beans” on page 139.

Comparing CMP with BMP beans

A BMP entity bean is inappropriate for large applications. This becomes obvious when you think about a large number of entity beans, each accessing a given database. For scalability, use a container-managed persistence entity bean.

However, BMPs may provide better portability than CMPs, because less container-generated code is used. Also CMPs may be necessary when mapping one entity bean to a join of tables, or associating an entity bean with another bean. The price for portability is the extra development effort that BMPs require.

A common practice for using BMPs is when you are using a database that is not supported for container-managed persistence.

Session bean

A session bean encapsulates typical business processes and may contain a conversational state associated with a particular client. These states are not stored in a permanent data source and will not survive a server failure, unlike the data in an entity bean. Session beans implement business logic, business rules, and workflow.

Nevertheless, a session bean can update data in an underlying database, usually by accessing an entity bean. For this reason, a session bean can be transaction aware. When created, instances of a session bean are identical, although, some session beans can store semipermanent data that makes them unique at certain points in their life cycle. A session bean is always associated with a single client and the life time of a session bean is typically that of its client.

For example, the task associated with transferring money between two bank accounts can be implemented in a session bean. Such a transfer session bean has to find two instances of an account entity bean (by using the account IDs), and then subtract a specified amount from one account and add the same amount to the other account.

Session beans are non-persistent enterprise beans. There are two types of session beans, *stateful* session beans and *stateless* session beans.

Stateful session bean

A stateful session bean acts on behalf of a single client and maintains client-specific session information (called conversational state) across multiple method calls and transactions. It exists for the duration of a single client/server session.

In order for a container to manage efficiently a large number of bean instances, it can take an instance out of memory and store it in permanent storage. The container uses object serialization to convert the bean state into a binary stream or BLOB (binary large object). It can then write the serialized data to permanent storage. This is called *passivation*. When this bean is invoked again, the container creates a new instance and initializes it with the data saved during passivation. This is called *activation*. Therefore, if the session bean contains a conversational state that must be preserved between method invocations, the session bean indicates stateful management mode.

Stateless session bean

A stateless session bean, by comparison, does not maintain any conversational state. Stateless session beans are pooled by their container to handle multiple requests from multiple clients. At any given moment a stateless session bean executes on behalf of only a single client.

The container considers the session bean stateless and never uses passivation, but instead can destroy it in case of memory resource shortage. Because all instances of stateless session beans are identical, the container can use any available instance to satisfy a client request.

Comparing stateless with stateful beans

When designing an enterprise application, many times the developer needs to take design decisions on what type of session bean should be used. This decision may not be always simple. The goal of this section is to simplify the decision making process.

Table 2-1 gives a comparison of stateless and stateful session beans.

Table 2-1 Stateful versus stateless session beans

Stateful	Stateless
<ul style="list-style-type: none">▶ Can retain state between method invocations▶ Can service business processes that span multiple methods or transactions▶ Can be aware of any client history	<ul style="list-style-type: none">▶ Cannot retain state between methods▶ Typically used for single request, single method invocation▶ Anonymous method provider, not aware of any client history or state

The type of session bean to use could be determined by trying to answer these two questions:

- ▶ Does the bean need to know the state of the client to perform business logic?
- ▶ Does the bean need to maintain state between method calls?

If the answer to any of these questions is yes, then you should consider using stateful beans. If the answer is no for both questions, then using stateless beans should be fine for your application.

However, consider these two points when you intend to use stateful beans:

- ▶ The state may be lost if a failure occurs (for example, network or system failures, system reboots).
- ▶ Stateful EJBs cannot easily be replicated, leading to poor performance in a clustered environment.

For more information on session beans see Chapter 9. “Session beans” on page 157.

Deployment of enterprise beans

After the development is done, all the enterprise bean classes are contained in a JAR file, the EJB-JAR. Before an enterprise bean can be installed in an application server, the enterprise bean must be deployed. During deployment, several application server-specific classes are generated. The deployment descriptor contains attribute and environment settings that define how the application server invokes enterprise bean functionality. Every enterprise bean must have a deployment descriptor that contains attributes used by the application server; these attributes can often be set for the entire enterprise bean or for individual methods of the bean.

The deployment descriptor is part of the contract between the bean developer and the bean consumer. This contract verifies that all the necessary information

for assembling and deploying the application, is passed between the bean provider, the application assembler, and the application deployer.

The role of the deployment descriptor is to capture the declarative information (for example, information that is not included directly in the enterprise beans code) that is intended for the consumer of the beans.

There are two basic kinds of information in the deployment descriptor:

- ▶ *Enterprise bean structural information.* Structural information describes the structure of an enterprise bean and declares an enterprise bean's external dependencies. Providing structural information in the deployment descriptor is mandatory for the bean developer. The structural information cannot, in general, be changed because doing so could break the enterprise bean. For example, isolation levels and transaction attributes are structural information.
- ▶ *Application assembly information.* Application assembly information describes how the enterprise beans are composed into a larger application deployment unit. Providing assembly information in the deployment descriptor is optional for the bean developer. Assembly level information can be changed without breaking the enterprise bean function, although, doing so may alter the behavior of an assembled application.

Comparing entity and session beans

So far we explained the two different types of enterprise beans, entity and session. Both entity and session beans provide a distributed component architecture for enterprise applications and share a lot of functionality. However, they are designed to serve different purposes. The decision—what logic to implement in a session bean and what in an entity bean—is not always simple.

The key differences between the two are that session beans represent business processes and typically contain business logic, while the entity beans encapsulate permanent business entities, such as business data and typically contain data-related logic.

Let's take an example of a typical Internet banking application. An Internet customer interacts with an electronic bank teller through a browser. A customer can check the balances of different accounts (checking or savings), transfer money from one account to the other, pay bills, or invest in funds.

Therefore, the electronic bank teller should consist of components that implement and provide all these services to the customer. The bank teller components can be well represented by session beans, as they have a business process conversation with the client.

On the other hand, the different accounts owned by a customer, as well as the customer personal data, can be well represented by entity beans, because this is the customer's permanent data that has to be accessed by the client.

Let's now take the component for transferring money from one account to the other and name it account manager. So, the client interacts with the account manager, does all the transfer between accounts, and then disconnects from the account manager session bean.

After the conversation is over, the account manager bean is free to service another client. However, the new account balance must be stored in a permanent source, so that the account data are persistent over several conversations. This is managed successfully by the account entity beans.

Therefore, session beans are performing application logic that uses persistent entity beans as the data that they are manipulating. In other words, the session beans act as a high-level interface to business processes, by masking the lower-level subsystem of entity beans used behind the scenes. This design approach is known as a facade.

The reason for doing so is based on the fact that entity beans model permanent business entities; therefore, they typically achieve higher level of reuse than the session beans. On the contrary, session beans typically model a business process, which can be modified or tuned over time based on new business requirements.

For instance, consider our Internet banking application with the account manager bean. This session bean knows how to withdraw and deposit money by calling methods on an account entity bean. One day you may decide to replace your session bean account manager with a different account manager. But you would still want all your customer's bank accounts to remain the same.

Table 2-2 gives some general guidelines that can be used when selecting among session beans and entity beans.

Table 2-2 Session beans versus entity beans

Session beans	Entity beans
<ul style="list-style-type: none">▶ Session beans cannot be persisted; they represent business processes and contain business logic as well as rules and workflow.▶ Session beans never include permanent data, but provide access to data.▶ A session bean can contain data-related logic as well, such as a session bean performing a database read via JDBC or SQLJ.▶ Use session beans, when you want to run a business process in a transactional, distributed and secure environment.▶ Session beans are only used by a single user at a time.	<ul style="list-style-type: none">▶ Entity beans can be persisted; they contain permanent data and data-related business logic.▶ Use entity beans, when you have permanent business data that you want to access as a distributed component.▶ Entity beans are shared by multiple clients.

End-to-end enterprise application development

Let's go through a scenario where you, as an IT project manager, have a project where you are going to use EJBs. What steps are you going to follow to develop your application?

First, you need to analyze the market for EJB container/servers, also called application servers. The vendors or container/server providers have lots of information published on the Web. However, be aware that some companies only sell containers, not application servers, which means you will have to buy two products instead of one. For more information see:

<http://www.flashline.com>

The next step is to determine which EJBs you could reuse instead of writing them yourself. Look up the [IBM DeveloperWorks Components Zone - component downloads](#) on the Internet to find out about which companies provide EJB (bean providers), and visit their sites.

After that research, you are now finally ready to start your development and assemble the enterprise beans to form your application; you are now a bean assembler.

Once the application is completed, you can deliver it to the deployment team in the form of an EJB-JAR file. This team is in charge of deploying the code to the specific container/server you have chosen to buy. This operation is done using tools delivered with the container and consists mainly of generating Java code to produce a deployed JAR file.

At this point, the deployer still needs to install the application in the container/server environment. At the end of this operation, the application is live and can be either tested or placed in production. The system administrator is now in charge.

EJB container responsibilities

A container is defined as a level of responsibility where a set of services are performed on behalf of the enterprise bean. A container vendor will provide a set of tools and classes to carry out the various services. The basic services provided by the EJB container are transactions, naming, and persistence (Figure 2-3).

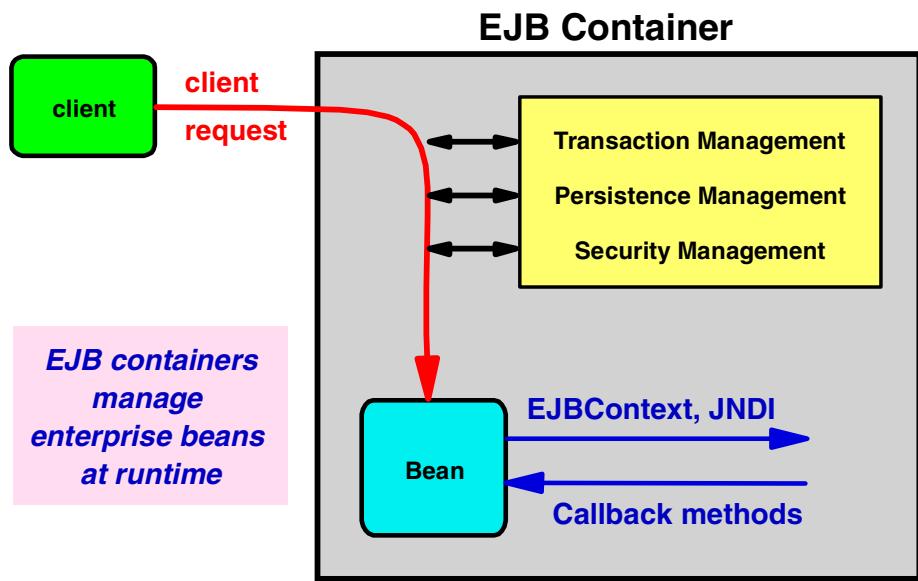


Figure 2-3 EJB container responsibilities

Container-managed transactions

Transactions are a safe conduit to have multiple components take part in distributed object operational activities. The EJB container handles the transaction operations as well as the coordinating activities between the transactions. Currently, only flat and distributed transactions are supported (but not nested transactions). Because the container manages the transactions, you can write Web applications without using an explicit transaction API.

Data caching

Some containers allow smart caching of entity beans, which allow some operations to occur in memory rather than at database level. This avoids accessing the non changing data again and again by making a database connection. There are three caching options available for the container in committing a transaction: options A, B, or C.

Option A

The container caches a readily available instance between transactions, which has explicit access to the state of the object in the persistent storage. This option is supported by WebSphere but should only be used in a single node system.

Option B

The container caches the instance between transactions, which does not have access to the persistent object state. This option is not supported by WebSphere.

Option C

The container does not cache the instance between transactions. The instance is returned to the pool after a transaction is completed. This option is supported by WebSphere and should be used in multiple node systems.

Persistence

Entity beans are persistent objects that represent data in an underlying storage system such as a database. EJB containers can provide transparent persistence to CMP entity beans. The container automatically performs all data retrieval and storage operations on behalf of the bean and the developer does not have to code any data access.

CMP enables a clear separation of business logic from data access logic. CMP also enables the container to efficiently pool and recycle the database connections and to cache data in memory and batch multiple database operations in a single connection.

The most common mechanisms used in persistence are object-to-relational persistence and object database persistence. Object-to-relational persistence involves mapping fields to relational database table columns. In object database persistence mechanism, object-oriented databases are used. This enables storing the objects as an object column type, which is saving the object's definition and state as it is.

Remote accessibility

Remote accessibility enables remote invoking of a native component by converting it into a networking component. EJB containers use the Java RMI interfaces to specify remote accessibility.

Naming

A typical EJB container uses JNDI to return a Java RMI remote stub of the EJB component when an EJB client performs a JNDI lookup for a home object. The clients can also use RMI's lookup facility. The `lookup` method of the `Naming` class knows how to contact a remote RMI registry, which is listening on a well defined port. But one serious problem with this method is that it cannot find CORBA object implementations, which ties up the client only with RMI remote objects. Therefore, the client must use JNDI.

Bean life-cycle management

The container is responsible for controlling the life cycle of the deployed components. As EJB clients start giving requests to the container, the container dynamically instantiates, destroys and reuses the beans as appropriate. For example, if a client requests for the service of a bean that does not exist in memory, it instantiates the bean and assigns it to the client. If there is memory constrain, then it may destroy some beans which are no longer needed by any client. Sometimes, if the bean request by a client is in the memory and is not in use currently, then the container may reassign the bean instance to this client. This technique is called *instance pooling*.

State management

Stateless beans can be directly reassigned from an inactive client to an active client as need arises. This is possible because no state is recorded for the first client. This reuse results in very good resource usage with a limited number of bean instances, and more client requests can be managed.

On the other hand, the container must use transparent state management for stateful session beans, in order for sharing limited instances with multiple clients. In this case, the container can take a stateful bean that has not been invoked by

its client for a stipulated time and serialize its state to the permanent storage. This technique is called *passivation*. Then it can reassign the bean instance to another client that requires the bean's service. When the first client makes a request for the bean, the preserved state can be retrieved and used again, perhaps in a different in-memory bean object or a newly instantiated bean. This technique is called *activation*.

Security

The security service provides a complete framework for distributed object security. The container's role in terms of security is to handle the authorization and authentication of bean users for the task they want to accomplish. This is achieved through access control lists (ACLs). An ACL is a list of users, the groups they belong to, and their rights. This also ensures confidentiality and non-repudiation. Containers enable the beans to automatically run as a security identity, which is much easier than programming security into the bean's business methods.

An enterprise bean and its interfaces

Every enterprise bean must have a bean class, a remote interface, and a home interface. An entity bean also has a key class.

Bean class

An enterprise bean class contains the implementation details of the component. All the business methods are defined in this class. The class implementation for session beans is very different from the implementation for entity beans. A session bean class contains business logic that can be used for computing prices or transferring funds from one account to another. For entity beans, an enterprise bean class typically implements all the methods required for manipulating the bean persistent fields. It also contains the callback methods used by the container to manage the life cycle of a bean instance. Clients, whether they are other enterprise beans or servlets, never access objects of this class directly. Instead, they use the home and remote interfaces to access the bean.

Remote interface

When a client wants to use an instance of an enterprise bean class, the client never invokes the method directly on the actual bean instance. The main reason is that bean classes cannot be called across the network directly, because they are not network-enabled. The client uses instead the remote interface to invoke the business methods defined in the bean class. This interface is known as the EJB object.

Typical methods in the remote interface are:

- ▶ The getter and setter methods to manipulate individual fields of the bean
- ▶ Business methods that manipulate multiple fields

Home interface

Clients use EJB objects and never beans directly. Therefore, there must be a way for the clients to acquire references to EJB objects. The home interface is used for this purpose. This interface contains methods that allow the client to create, find, and remove instances of the bean. The home interface for session beans does not have any finder methods, because session beans are not persistent objects. This interface is known as the EJB home.

Typical methods in the home interface are:

- ▶ create (one or more with different sets of parameters)
- ▶ findByPrimaryKey (retrieve one instance)
- ▶ remove (one instance)
- ▶ custom finder methods (find by other criteria, one or multiple instances)

Key class

For entity beans, there is also a key class that encapsulates the key field (or key fields) of the bean. This class is used in the findByPrimaryKey method of the home interface.

Unlike an entity bean, a session bean does not have a primary key class, because instances of session beans cannot be shared and therefore cannot be uniquely identified from each other.

Finding the home of an enterprise bean

Before any work can be done with an enterprise bean, it must be found and instantiated, or it must be created. These activities are performed on the home interface; therefore, the first activity of a client is to find the home.

The EJB container provides access to homes through JNDI. Typical coding in the IBM Version 3.5 toolset to find a home is:

```
InitialContext initialContext = new InitialContext();
Object objHome = initialContext.lookup("itso/ejb35/cmp/Customer");
CustomerHome customerHome = (CustomerHome)
    javax.rmi.PortableRemoteObject.narrow(objHome,CustomerHome.class);
```

We will explain this in more detail in “How to access EJBs” on page 245.

This remote access to homes can be expensive. Many application servers provide caching functionality for homes to increase performance. Also client applications can store homes in instance or class variables to speed up the access.

JNDI name of a home

Notice the name that was used to find the home:

itso/ejb35/cmp/Customer

In Version 3.5 of the IBM tool set, the home name is composed of the fully qualified name, based on the package structure of the enterprise bean. In earlier versions, short names, such as Customer, were the default. The actual name that is used at execution time is part of the deployment descriptor of the enterprise bean.



IBM products for EJB development and deployment

This chapter contains brief descriptions of IBM products supporting the EJB specifications.

IBM provides two products for EJB development and execution:

- ▶ The first product, **WebSphere Application Server**, provides the runtime infrastructure.
- ▶ The second product, **VisualAge for Java**, provides the development environment.

WebSphere Application Server, Advanced Edition

In this section, we describe WebSphere Application Server Advanced Edition (WAS).

Overview

WAS is the runtime product implementing most of the J2EE specifications (see “Java 2 Platform, Enterprise Edition” on page 9 for details). It currently supports most technologies as defined in the J2EE V1.2 with some variations.

The forthcoming WAS 4.0 will fully support J2EE V1.2.

Architecture

WAS uses a repository-based administration which allows the creation of domains. A domain has a single administration database which is shared by all the nodes or physical systems in the domain (Figure 3-1).

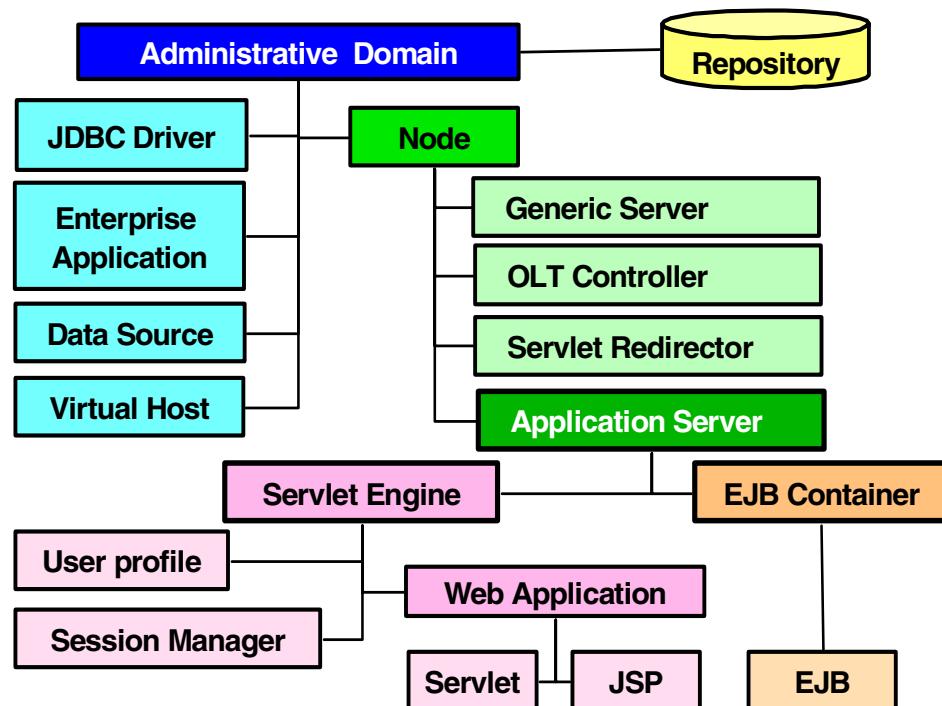


Figure 3-1 WebSphere Application Server: administrative domain

As Figure 3-1 shows, the nodes can contain numerous elements, such as application servers, OLT controllers, generic servers and servlet redirectors.

Most users will mainly deal with application servers, which contain the different elements (EJBs, servlets, and JSPs) needed for their applications. Usually, an application server has one servlet engine and one EJB container.

J2EE and WebSphere

As explained in “IBM products and J2EE” on page 16, WAS provides most services that a J2EE server should provide:

JNDI

The JNDI name space is kept locally by the administrative server. The JNDI naming service is a persistent service provided through the CORBA CosNaming API implemented as EJBs; this service listens on port 900. There is another naming service component named Location Service Daemon (LSD), which uses its own Object Request Broker (ORB) and is required for persistent object references. LSD listens on port 9000.

JDBC data sources

WAS supports the JDBC 2.0 specification and uses data sources to access databases. The WAS implementation of data sources provides connection pooling for better performance.

Data sources are specified for the domain and are then made available to the nodes by installing the associated JDBC driver on the different nodes.

EJB container

WAS uses one container for all EJB types and supports four database vendors for CMP entity beans: DB2, Oracle, Sybase, and Microsoft SQL Server.

A data source can be specified for all CMP entity beans or you can set one data source per CMP entity bean type. Other properties of the container can be set through the administration client such as database access (option A or C) and cache size.

Servlet engine

As specified in the Java Servlet 2.2 specification, the servlet engine used in WebSphere is grouping servlets in Web applications. This partitioning enables you to maintain a separation between servlets from different applications.

A servlet engine also has a session manager attached as well as a user profile manager.

Web applications

A Web application contains mainly a number of servlets and a JSP compiler. A tailored class path can be set for each Web application. See “Creating a Web application” on page 462.

Session manager

A session manager is setup automatically for each servlet engine. It manages the way HTTP sessions are tracked (using cookies or URL encoding), created (in memory or persistent), and how long they will survive (diverse timeouts can be set).

User profile management

This offers a way to track users. It can be viewed as a basic personalization suite. See *WebSphere Personalization Solutions Guide*, SG24-6214, for more information.

WebSphere tooling

Once EJBs have been written and packaged in a EJB-JAR file, they have to be deployed before being used inside WAS. You can either use the JetAce tool provided by WAS or VisualAge for Java to do this.

This operation will tie the EJBs to the container by generating

- ▶ Home classes (*EJSRemoteBeanHome*, *EJSBeanHomeBean*)
- ▶ EJB object classes (*EJSRemoteBean*)
- ▶ Mapping code (*EJSFinderBeanBean*, *EJSJDBC_PersisterBean*)
- ▶ Proxies that are stubs and ties used for remote access of the EJBs
(_*EJSRemoteBean_Tie*, _*EJSRemoteBeanHome_Tie*, _*Bean_Stub_BeanHome_Stub*)

Where, *Bean* should be replaced by the name of the EJB.

JetAce tool

JetAce is a tool that allows you to edit the deployment descriptor of an EJB in the deployment JAR file. It is invoked by using the command line utility *jetace.bat* as shown in Figure 3-2. A deployment descriptor for a bean can be defined or modified. Once the deployment descriptor is defined, the bean can then be deployed.

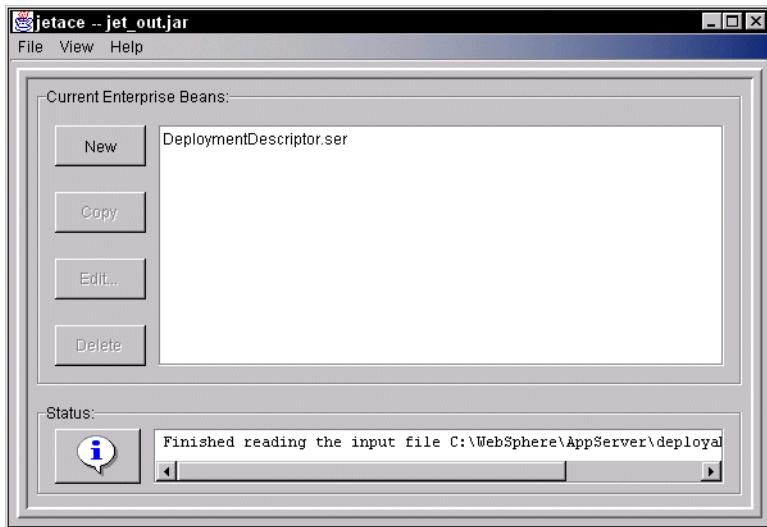


Figure 3-2 JetAce deployment tool

Since Version 3.5.2, it is also possible to define a deployment descriptor for an EJB by importing an XML descriptor. This function is not EJB 1.1 compliant.

If you plan to use workload management (WLM), a command line utility named `wlmjar.cmd` can enable WLM for your EJBs. It takes a deployed JAR and generates the smart proxies that will handle the load management. If you deploy the beans from VisualAge for Java, this is not necessary because WLM-enabled proxies are automatically generated.

EJBDeploy tool

In the next version of the IBM tool set, an EJBDeploy command line tool will be available to deploy EJB 1.1 compliant beans.

VisualAge for Java Enterprise Edition

The preferred tool for Java and EJB development is VisualAge for Java. The tool offers an integrated development environment in which a number of tools can be plugged in.

Overview

VisualAge for Java provides a complete suite for enterprise development ranging from simple database tooling to sophisticated EJB development. The core of the tool is the Integrated Development Environment (IDE), which relies on a repository. All the rest is built as additional features which can be added or removed on demand (Figure 3-3).

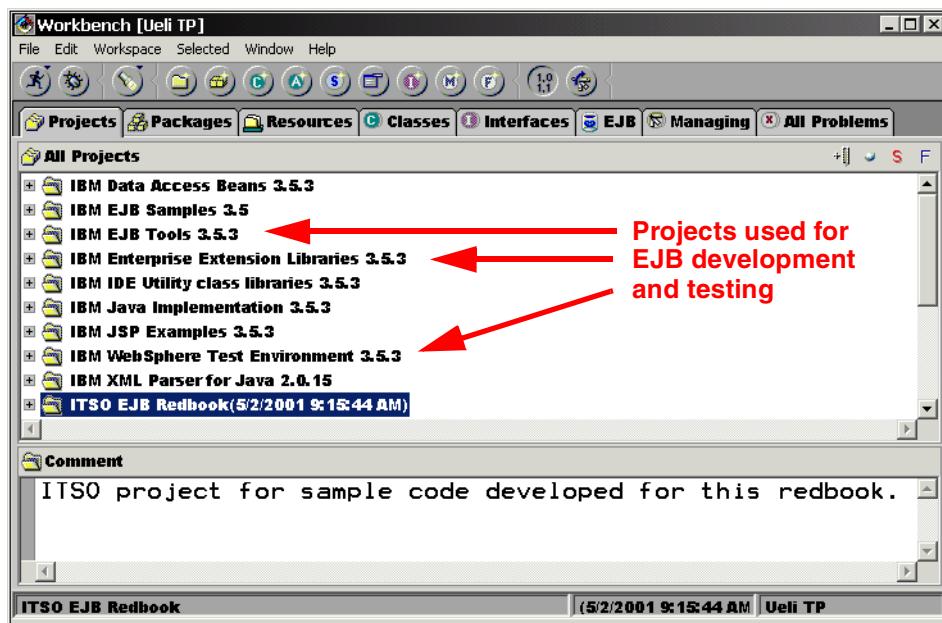


Figure 3-3 VisualAge for Java - Workbench

Integrated Development Environment

The Integrated Development Environment (IDE) provides a different perspective for object development as it is offering an object view of the code. Classes are broken up into class definition and methods. Also, a class can be viewed from different angles from the plain source view to the JavaBeans view or the Hierarchy view as shown in Figure 3-4.

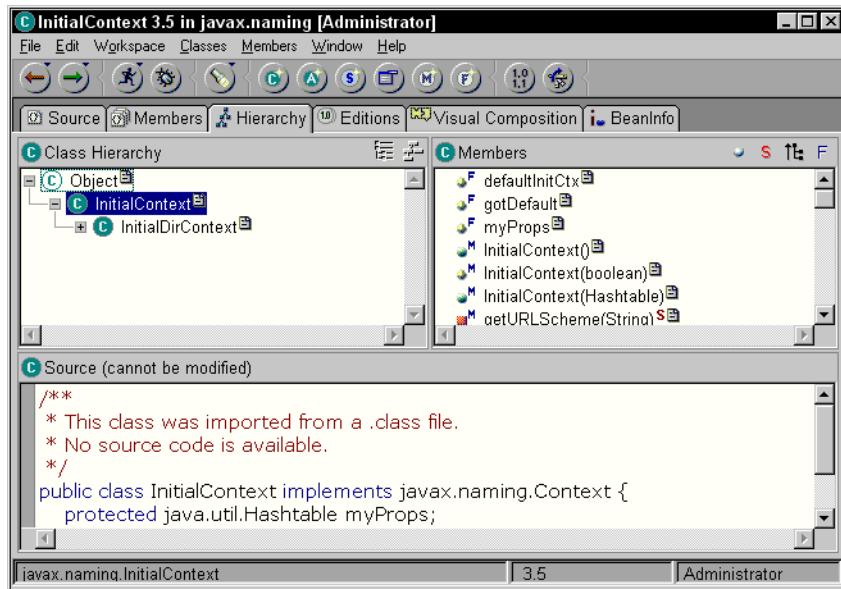


Figure 3-4 VisualAge for Java - hierarchy view

Database connectivity

VisualAge for Java has different tools to help you connect to databases. The simplest are the data access beans that encapsulate SQL statements in JavaBeans. SQLJ, Persistence Builder, Stored Procedure Builder, or just plain JDBC coding are other ways to access the database with greater flexibility and performance.

Enterprise connectivity

VisualAge for Java offers a large number of ways to connect to your enterprise data and transactions systems, such as Domino, CICS, IMS, C++ legacy code, SAP, AS/400 or CORBA. XML and XMI (XML metadata interchange) are supported to help you transfer data between heterogeneous systems.

J2EE development environment

VisualAge for Java contains a modified version of the WAS runtime. The main differences are that VisualAge for Java does not allow you to administer your domain through the GUI administration client and that work load management (WLM) is not supported.

The main development place for EJB is the *EJB page* in Workbench. It allows you to do most of the tasks related to EJBs.

Servlet development is done through some SmartGuides and normal Java development. JSPs, however, have to be developed externally with a tool such as WebSphere Studio.

Testing of servlets, JSPs, and EJBs can be done using the WebSphere Test Environment (WTE). This tool is the WAS modified runtime running within VisualAge for Java.

Because EJBs are the main subject of the book, let's have a deeper look at the EJB Environment.

EJB page

The EJB development environment is available as a VisualAge for Java feature that has to be loaded in the Workbench.

The EJB development environment shown in Figure 3-5 enables you to perform most development tasks using SmartGuides. You will be able to:

- ▶ Create EJB groups (a logical grouping of EJBs used to create associations and mapping)
- ▶ Create new EJBs (BMP, CMP and session beans)
- ▶ Create EJBs with inheritance
- ▶ Create associations
- ▶ Create access beans
- ▶ Map entity beans to relational databases
- ▶ Create EJBs from an existing database schema
- ▶ Create/edit the deployment descriptor
- ▶ Deploy your EJBs

For more information about the EJB development environment, refer to Chapter 5. "EJB development environment" on page 65.

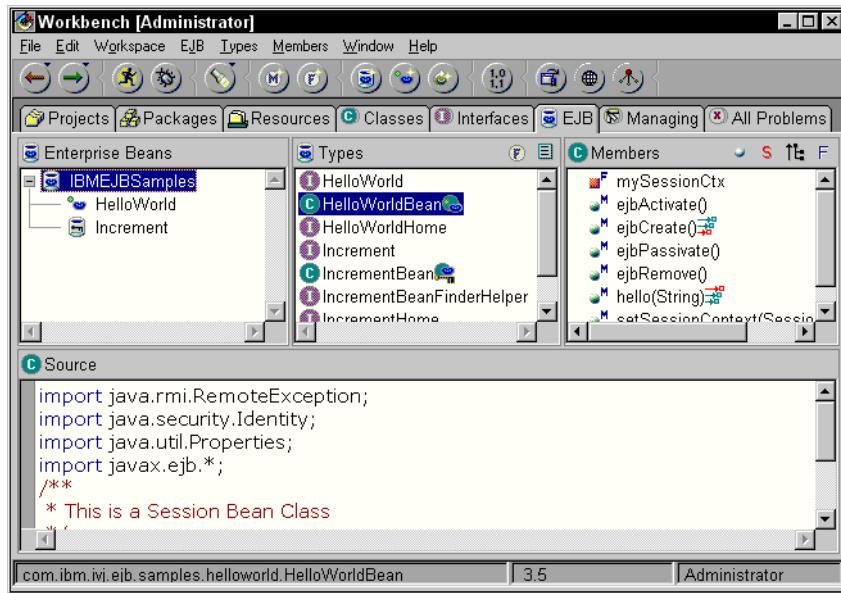


Figure 3-5 VisualAge for Java - EJB page

Container-managed persistence

VisualAge for Java offers a sophisticated mapping for CMP entity beans. The approach brings two layers between the DB and the EJBs. Figure 3-6 shows an example of how to do the mapping from EJBs to the database.

Let's define some of the terms:

EJB model	An EJB group
DB schema	Contains the table definitions and some basic type conversions
Mapping	Defines how EJB properties are mapped to table columns and how associations are mapped to foreign keys
DDL	Code to create table in the DB, generated from the schema
Deployed code	Generated classes that "glue" your EJBs to the container and will perform the database operations as instructed by the mapping

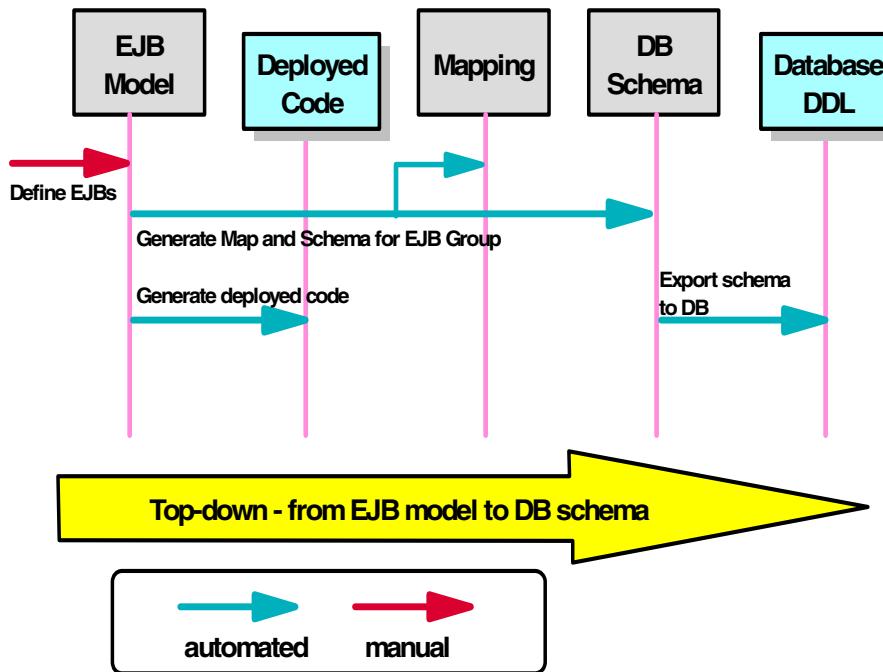


Figure 3-6 EJB mapping - architecture

For more information about mapping of entity beans to database tables, refer to Chapter 15. “Advanced mapping for container-manager entity beans” on page 299.

WebSphere Test Environment

The WebSphere Test Environment (WTE) is the runtime version of WAS to test and debug code developed with VisualAge for Java. Figure 3-7 shows one of the elements hosted in the WTE, the persistent name server, which acts as a JNDI server.

You can also find in the WTE a servlet engine, an HTTP server, a JSP compiler and execution monitor, and a data source manager.

For more information about the WebSphere Test Environment, refer to Chapter 6. “WebSphere Test Environment” on page 79.

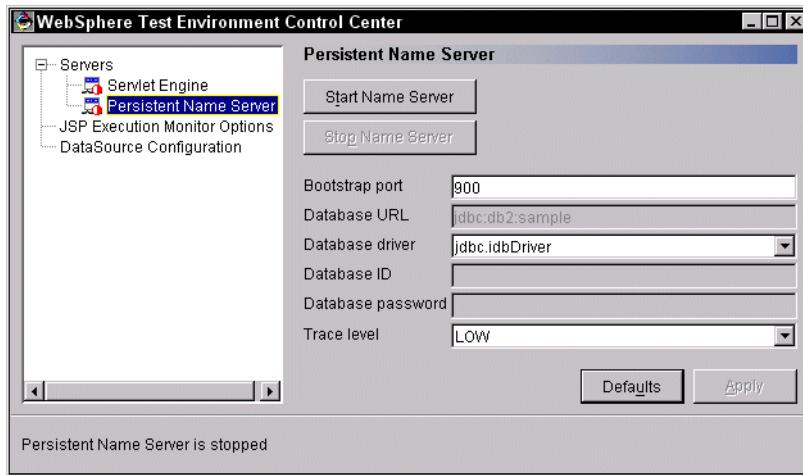


Figure 3-7 VisualAge for Java - WebSphere Test Environment

Deployment

Once the EJBs and their mappings are defined, the code can be deployed using the *EJB -> Generate deployed code* menu. Afterwards, the deployed beans can be exported to:

- ▶ EJB JAR—a non-deployed version of your code. JetAce can then be used to deploy the EJBs before loading them into WAS.
- ▶ Deployed JAR—a deployed version of your beans, WebSphere ready and WLM-enabled.
- ▶ Client JAR—all the code you need for the implementation of the client. This includes the interfaces of the beans, the proxies, and the dependent classes.

All export SmartGuides include a search utility to help you select the dependent classes/interfaces referenced in the EJB code.

Testing and debugging

Once the deployed code is generated, you can test the code. Just make sure to start the WTE and the persistent name server. After bringing up the server configuration with your beans and starting it, you can launch the EJB test client provided by the EJB development environment and check your beans (Figure 3-8). See “Testing enterprise Java beans” on page 91 for detailed information.

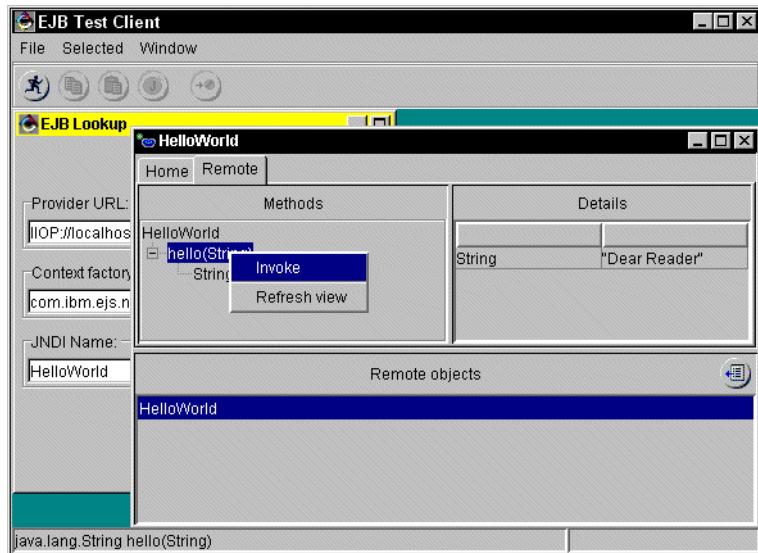


Figure 3-8 VisualAge for Java - EJB client

If a problem occurs or if you get inconsistent results, you can start debugging your code inside the VisualAge for Java debugger as shown in Figure 3-9.

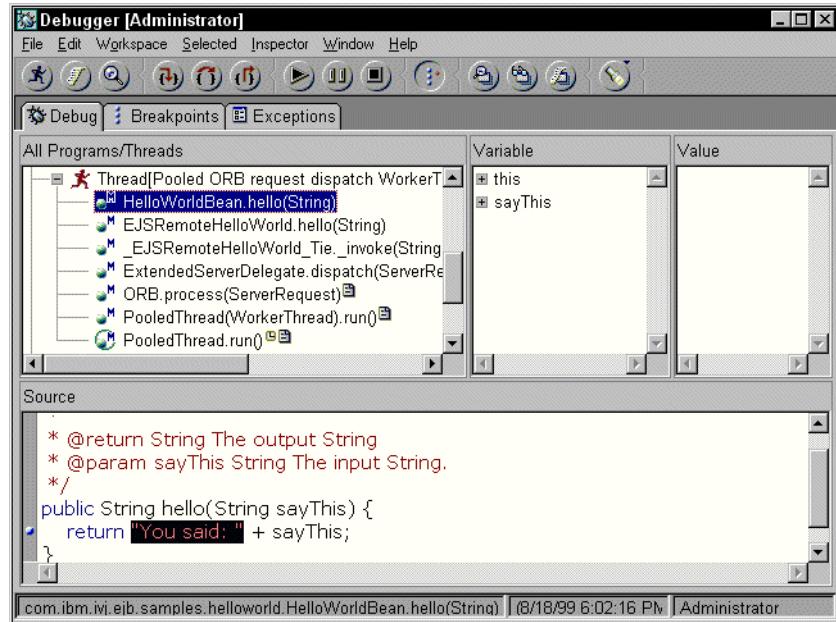


Figure 3-9 VisualAge for Java - Debugger

RAD support

If you think of developing a fat client for your EJB application, VisualAge for Java has a RAD support in the form of the Visual Composition Editor (VCE), which allows you to visually build GUIs using Swing or AWT. It supports the JavaBeans specification and extends it with the capability to connect different beans together.

Team environment

As most development tool, VisualAge for Java allows multiple developers to work together. This can be done using either the VisualAge for Java team server or an external source code manager.

The team server allows, unlike other environments, parallel development on the same classes. For more information about team development, refer to “Team development” on page 75.



4

ITSO bank example

In this chapter we describe the model and database for a banking application. In subsequent chapters we implement parts of the model as enterprise beans, and parts of the business logic as session beans.

Bank model

The ITSO bank model consists of a few entities and relationships (Figure 4-1).

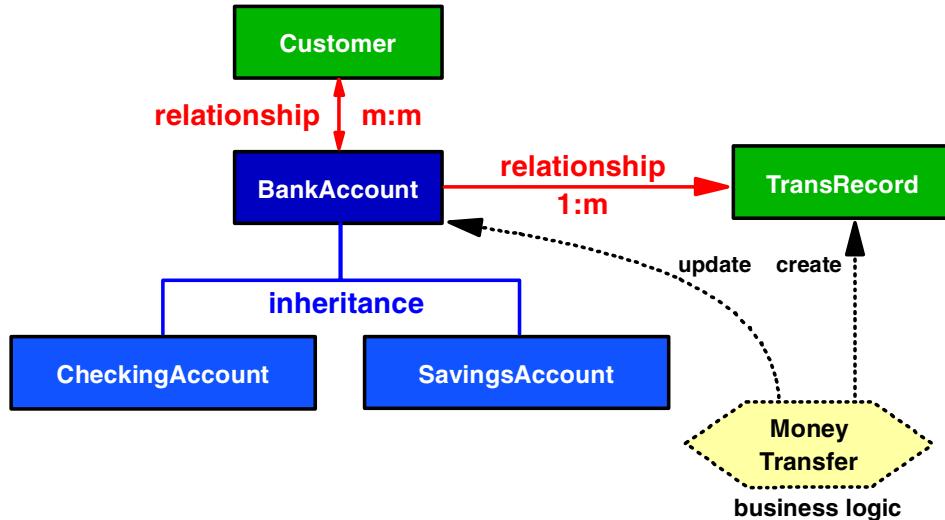


Figure 4-1 Bank model

The entities and relationships in the model are:

Customer	A customer of the bank.
BankAccount	A generic bank account. A customer may have multiple bank accounts and an account may be owned by multiple customers. A bank account is either a checking or a savings account.
Checking	A subclass of the generic bank account.
Savings	A subclass of the generic bank account.
TransRecord	A transaction record that is generated for each banking transaction, such as a deposit, withdrawal, or transfer of money between two accounts. A bank account may have many transaction records.

We will implement the entities as enterprise beans. For illustration purposes we will use container-managed and bean-managed entity beans.

We will also use the advanced features of the IBM EJB product set to implement the inheritance of the bank accounts and the relationships between entities.

We will implement a money transfer between two accounts as a session bean.

Bank database

The bank model is based on an underlying relational database. The **EJBBANK** database consists of the tables shown in Figure 4-2.

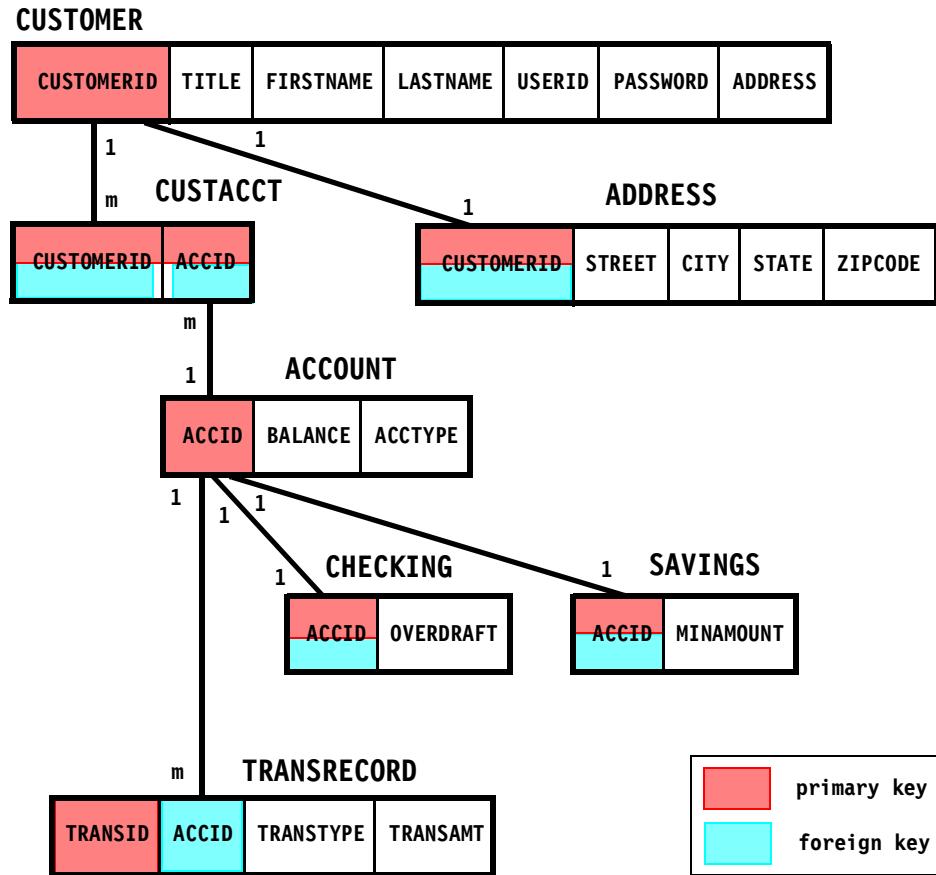


Figure 4-2 Bank database

The m:m relationship between CUSTOMER and ACCOUNT is implemented with an intermediate table, CUSTACCT, that contains two foreign keys pointing the the customer and the related account.

CHECKING and SAVINGS are dependent tables. Each account is either a checking or a savings account and, therefore, is implemented with a row in the ACCOUNT table and a row in either the CHECKING or SAVINGS table.

ADDRESS is a dependent table of CUSTOMER.

The CUSTOMER table includes all the information recorded for a bank customer (Table 4-1).

Table 4-1 Customer table

Column name	Type	Length	Key	Nulls	Description
CUSTOMERID	INTEGER		PK	No	Customer ID
TITLE	CHAR	3	No	No	Title
FIRSTNAME	VARCHAR	30	No	No	First name
LASTNAME	VARCHAR	30	No	No	Last name
USERID	CHAR	8	No	Yes	User ID
PASSWORD	CHAR	8	No	Yes	Password
ADDRESS	BLOB	2000	NO	YES	Address object

The address information can be stored either as a BLOB in the CUSTOMER table, or as individual fields in the ADDRESS table. We will explore the mapping of customer entity beans into one table and into two tables.

The ADDRESS table includes the detailed address fields when a customer is mapped into two tables (Table 4-2).

Table 4-2 Address table

Column name	Type	Length	Key	Nulls	Description
CUSTOMERID	INTEGER		PK, FK	No	Customer ID
STREET	CHAR	20	No	Yes	Street name, number
CITY	CHAR	12	No	Yes	City
STATE	CHAR	12	No	Yes	State or country
ZIPCODE	CHAR	10	No	Yes	Postal code

The address table includes the customer ID as a foreign key, which also serves as the primary key. Therefore, only one entry in the address table can point to a matching entry in the customer table.

Information about bank accounts is spread into three tables. This design enables us to implement an inheritance model with two types of bank accounts, checking and savings.

The ACCOUNT table includes all the information common to both checking and savings accounts (Table 4-3).

Table 4-3 Account table

Column name	Type	Length	Key	Nulls	Description
ACCID	CHAR	8	PK	No	Account ID
BALANCE	DEC	(8, 2)	No	No	Balance
ACCTYPE	VARCHAR	8	No	No	Account type (CHECKING, SAVINGS)

The account type column serves as an indicator to the type of account. This column will be used in the inheritance model.

The CHECKING table contains the fields that are specific to checking accounts, in our case an overdraft limit (Table 4-4). The balance of the account may be negative up to this limit.

Table 4-4 Checking table

Column name	Type	Length	Key	Nulls	Description
ACCID	CHAR	8	PK, FK	No	Account ID
OVERDRAFT	DEC	(8, 2)	No	Yes	Overdraft amount

The SAVINGS table contains the fields that are specific to savings accounts, in our case a minimum balance amount (Table 4-5). The balance of the account must be at least the minimum.

Table 4-5 Savings table

Column name	Type	Length	Key	Nulls	Description
ACCID	CHAR	8	PK, FK	No	Account ID
MINAMOUNT	DEC	(8, 2)	No	Yes	Minimum amount

The TRANSRECORD table includes the information recorded for successful banking transactions (Table 4-6).

A deposit transaction generates a record with transaction type D (debit), a withdrawal generates transaction type C (credit), and a money transfer generates a record for each account involved, one with transaction type D (to account) and one with transaction type C (from account).

Table 4-6 Transrecord table

Column name	Type	Length	Key	Nulls	Description
TRANSID	TIMESTAMP	26	Yes	No	Transaction ID
ACCID	CHAR	8	FK	No	Account ID
TRANSTYPE	CHAR	1	No	No	Transaction type (D = Debit C = Credit)
TRANSAMT	DEC	(8, 2)	No	No	Transaction amount

The CUSTACCT table includes the relationship information between customers and accounts, that is, what customer owns which accounts and which accounts can be accessed by which customer (Table 4-7).

Table 4-7 Customer - account relationship table

Column name	Type	Length	Key	Nulls	Description
CUSTOMERID	INTEGER		PK, FK	No	Customer ID
ACCID	CHAR	8	PK, FK	No	Account ID

Each column is a foreign key pointing to either the customer or the account table. Both columns together form the primary key.

Database definition DDL

The database objects can be created using the DDL shown in Figure 4-3 and and Figure 4-4.

```
echo --- create the EJBBANK database ---
CREATE DATABASE EJBBANK

echo --- connect to EJBBANK database ---
CONNECT TO EJBBANK

echo --- drop tables ---
DROP TABLE ITSO.TRANSRECORD
DROP TABLE ITSO.CUSTACCT
DROP TABLE ITSO.ADDRESS
DROP TABLE ITSO.CHECKING
DROP TABLE ITSO.SAVINGS
DROP TABLE ITSO.ACCOUNT
DROP TABLE ITSO.CUSTOMER

echo --- create tables ---
CREATE TABLE ITSO.CUSTOMER (
    customerid    INTEGER      NOT NULL,          \
    title         CHAR(3)      NOT NULL,          \
    firstname     VARCHAR(30)  NOT NULL,          \
    lastname      VARCHAR(30)  NOT NULL,          \
    userid        CHAR(8),                \
    password      CHAR(8),                \
    address       BLOB(2000),              \
                                PRIMARY KEY (CUSTOMERID) )
CREATE TABLE ITSO.ADDRESS (
    customerID   INTEGER      NOT NULL,          \
    street        CHAR(20),                \
    city          CHAR(12),                \
    state         CHAR(12),                \
    zipcode       CHAR(10),                \
                                PRIMARY KEY (CUSTOMERID) )
CREATE TABLE ITSO.CUSTACCT (
    customerid    INTEGER      NOT NULL,          \
    accid         CHAR(8)      NOT NULL,          \
                                PRIMARY KEY (CUSTOMERID,ACCID) )
CREATE TABLE ITSO.ACCOUNT (
    accid         CHAR(8)      NOT NULL,          \
    balance       DEC(8,2)     NOT NULL,          \
    acctype       VARCHAR(8)   NOT NULL DEFAULT 'CHECKING', \
                                PRIMARY KEY (ACCID) )
CREATE TABLE ITSO.CHECKING (
    accid         CHAR(8)      NOT NULL,          \
    overdraft    DEC(8,2)     NOT NULL DEFAULT 200.00, \
                                PRIMARY KEY (ACCID) )
```

Figure 4-3 Bank DDL: ejbbank.ddl (part 1)

```

CREATE TABLE ITSO.SAVINGS (
    accid      CHAR(8)      NOT NULL,          \
    minamount   DEC(8,2)      NOT NULL DEFAULT 100.00, \
                                         PRIMARY KEY (ACCID) )
CREATE TABLE ITSO.TRANSRECORD (
    transid    TIMESTAMP    NOT NULL,          \
    accid      CHAR(8)      NOT NULL,          \
    transtype   CHAR(1)      NOT NULL,          \
    transamt   DEC(8,2)      NOT NULL,          \
                                         PRIMARY KEY (TRANSID) )

echo --- referential integrity ---
ALTER TABLE ITSO.TRANSRECORD          \
    ADD CONSTRAINT "AccountTransrecord" FOREIGN KEY (ACCID) \
        REFERENCES ITSO.ACCOUNT ON DELETE RESTRICT
ALTER TABLE ITSO.CUSTACCT            \
    ADD CONSTRAINT "CatoCustomer" FOREIGN KEY (CUSTOMERID) \
        REFERENCES ITSO.CUSTOMER ON DELETE RESTRICT
ALTER TABLE ITSO.CUSTACCT            \
    ADD CONSTRAINT "CatoAccount"  FOREIGN KEY (ACCID) \
        REFERENCES ITSO.ACCOUNT ON DELETE RESTRICT
ALTER TABLE ITSO.ADDRESS             \
    ADD CONSTRAINT "CustAddress"   FOREIGN KEY (CUSTOMERID) \
        REFERENCES ITSO.CUSTOMER ON DELETE RESTRICT
ALTER TABLE ITSO.CHECKING           \
    ADD CONSTRAINT "CheckingAccount" FOREIGN KEY (ACCID) \
        REFERENCES ITSO.ACCOUNT ON DELETE RESTRICT
ALTER TABLE ITSO.SAVINGS             \
    ADD CONSTRAINT "SavingsAccount" FOREIGN KEY (ACCID) \
        REFERENCES ITSO.ACCOUNT ON DELETE RESTRICT

echo --- execute GRANT statements ---
GRANT CONNECT ON DATABASE      TO PUBLIC
GRANT ALL     ON ITSO.CUSTOMER  TO PUBLIC
GRANT ALL     ON ITSO.ACCOUNT   TO PUBLIC
GRANT ALL     ON ITSO.CHECKING  TO PUBLIC
GRANT ALL     ON ITSO.SAVINGS   TO PUBLIC
GRANT ALL     ON ITSO.TRANSRECORD TO PUBLIC
GRANT ALL     ON ITSO.CUSTACCT  TO PUBLIC
GRANT ALL     ON ITSO.ADDRESS   TO PUBLIC

echo --- connect reset ---
CONNECT RESET

```

Figure 4-4 Bank DDL: ejbbank.ddl (part 2)

Database content

The database can be filled with sample data as shown in Figure 4-5.

```
echo --- load the EJBBANK database ---
CONNECT TO EJBBANK
INSERT INTO ITSO.CUSTOMER
    (customerid, title, firstname, lastname, userid, password) VALUES \
    (101, 'Mr', 'Dom', 'Faidherbe', 'cust101', 'DF'), \
    (102, 'Mr', 'Akis', 'Laftsidis', 'cust102', 'AL'), \
    (103, 'Mr', 'Kart', 'Ponnalagu', 'cust103', 'KP'), \
    (105, 'Ms', 'Unknown', 'Lady', null, null), \
    (106, 'Mr', 'Ueli', 'Wahli', 'cust106', 'UW')
INSERT INTO ITSO.ADDRESS
    (customerid, street, city, state, zipcode) VALUES \
    (106, 'Steinway Ave', 'Campbell', 'California', '95008')
INSERT INTO ITSO.ADDRESS (customerid) VALUES (101),(102),(103),(105)
INSERT INTO ITSO.ACCOUNT
    (accid, acctype, balance) VALUES \
    ('101-1001', 'CHECKING', 80.00), \
    ('101-1002', 'SAVINGS', 375.26), \
    ('102-2001', 'SAVINGS', 9375.26), \
    ('102-2002', 'CHECKING', 75.50), \
    ('103-3001', 'SAVINGS', 100.00), \
    ('105-5001', 'CHECKING', 0.00), \
    ('106-6001', 'CHECKING', 1000.00), \
    ('106-6002', 'SAVINGS', 2000.00), \
    ('106-6003', 'SAVINGS', 3000.00)
INSERT INTO ITSO.CHECKING
    (accid, overdraft) VALUES \
    ('101-1001',200.00), ('102-2002',200.00), \
    ('105-5001',200.00), ('106-6001',300.00)
INSERT INTO ITSO.SAVINGS
    (accid, minamount) VALUES \
    ('101-1002',100.00), ('102-2001',100.00), ('103-3001',150.00), \
    ('106-6002',100.00), ('106-6003',250.00)
INSERT INTO ITSO.CUSTACCT
    (customerid, accid) VALUES \
    (101,'101-1001'), (101,'101-1002'), \
    (102,'102-2001'), (102,'102-2002'), \
    (103,'103-3001'), (105,'105-5001'), \
    (106,'106-6001'), (106,'106-6002'), (106,'106-6003'), (106,'105-5001')
INSERT INTO ITSO.TRANSRECORD
    (transid, accid, transtype, transamt) VALUES \
    (CURRENT TIMESTAMP, '101-1001', 'C', 80.00 )
INSERT INTO ITSO.TRANSRECORD
    (transid, accid, transtype, transamt) VALUES \
    (CURRENT TIMESTAMP, '101-1002', 'D', 200.00 )
INSERT INTO ITSO.TRANSRECORD
    (transid, accid, transtype, transamt) VALUES \
    (CURRENT TIMESTAMP, '106-6001', 'D', 66.66 )
INSERT INTO ITSO.TRANSRECORD
    (transid, accid, transtype, transamt) VALUES \
    (CURRENT TIMESTAMP, '106-6002', 'C', 66.66 )
CONNECT RESET
```

Figure 4-5 Bank SQL sample data: ejbbank.sql

Creating the database and tables

The DDL and SQL statements are provided in the sample code on the Redbooks Web site as files ejbbank.ddl and ejbbank.sql.

In a DB2 command window, switch to the directory of the sample files and run these commands:

```
db2 -fejbbank.ddl  
db2 -fejbbank.sql
```

You can always rerun these commands to restore the database.



Part 2

Enterprise JavaBeans development

In this part we describe how to develop enterprise beans with VisualAge for Java. We start with the EJB development environment and the WebSphere Test Environment and then develop simple entity and session beans using VisualAge for Java.

We continue with custom finder methods, access beans to simplify client programming, and basic transaction concepts. We then show simple examples of client programming and deploy the EJBs and simple applications to a WebSphere Application Server.



EJB development environment

VisualAge for Java provides the EJB development environment for all activities around creating, debugging, and deploying EJBs.

The EJB page of the VisualAge for Java Workbench is the central hub for all development activities.

Team development of enterprise beans brings a set of challenges to a development team and must be carefully planned and executed.

The EJB development environment is a feature that must be loaded into the Workbench. Use *File -> QuickStart* (or F2) and *Add Feature* to select the *IBM EJB Development Environment* feature.

Attention: To follow the directions given in this chapter, VisualAge for Java must be installed as described in Appendix A. “Setting up the environment” on page 453 and configured as described in “Setting up VisualAge for Java” on page 466.

EJB page

All EJB development is performed on the EJB page of the Workbench (Figure 5-1).

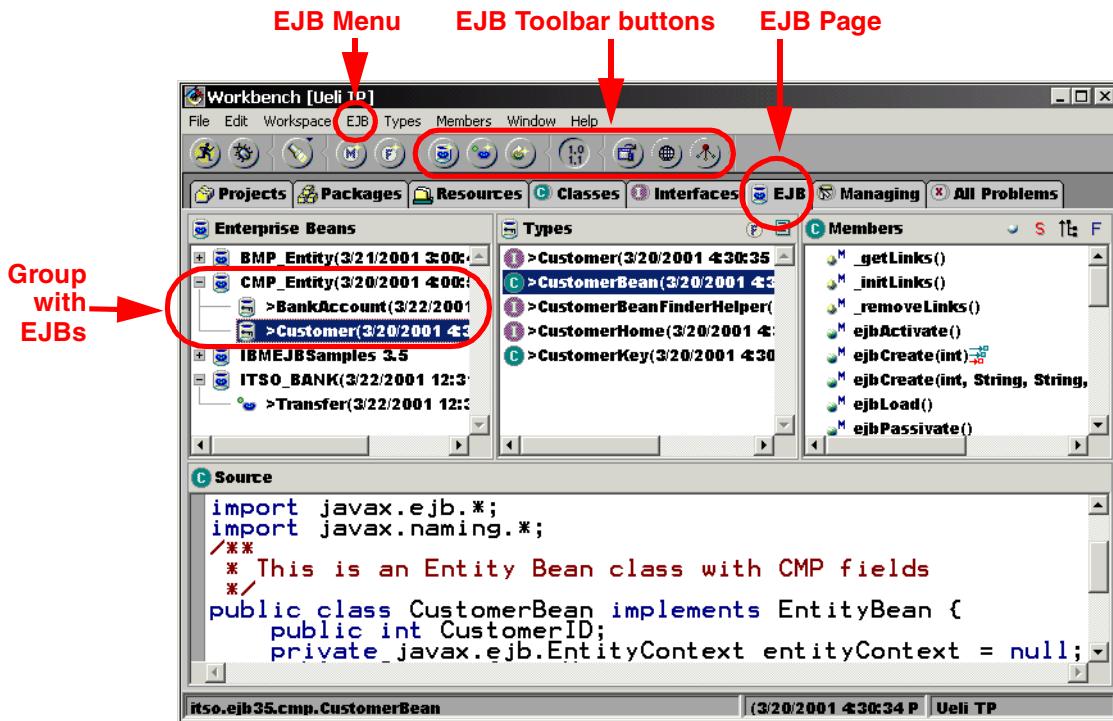


Figure 5-1 Workbench EJB page

The EJB of the Workbench is structured into four panes:

- ▶ Enterprise Beans
- ▶ Types (or Properties)
- ▶ Members
- ▶ Source

Actions on selected objects are performed using:

- ▶ Toolbar buttons
- ▶ EJB menu
- ▶ Context menu (right-click)

Enterprise Beans pane

This pane contains all the EJB groups loaded into the Workbench, independent of the project they belong to. Each group can be expanded to show the enterprise beans (entity beans and sessions beans) that belong to the group.

EJB group

Enterprise beans can be organized into many EJB groups, with the restriction that enterprise beans that are related through inheritance or associations must belong to the same group.

EJBReserved package

Each EJB group belongs to a Workbench project. The control or meta information of an EJB group is stored in a special package of the project. This package has the name of the group with the suffix EJBReserved, for example, CMP_EntityEJBReserved (Figure 5-2).

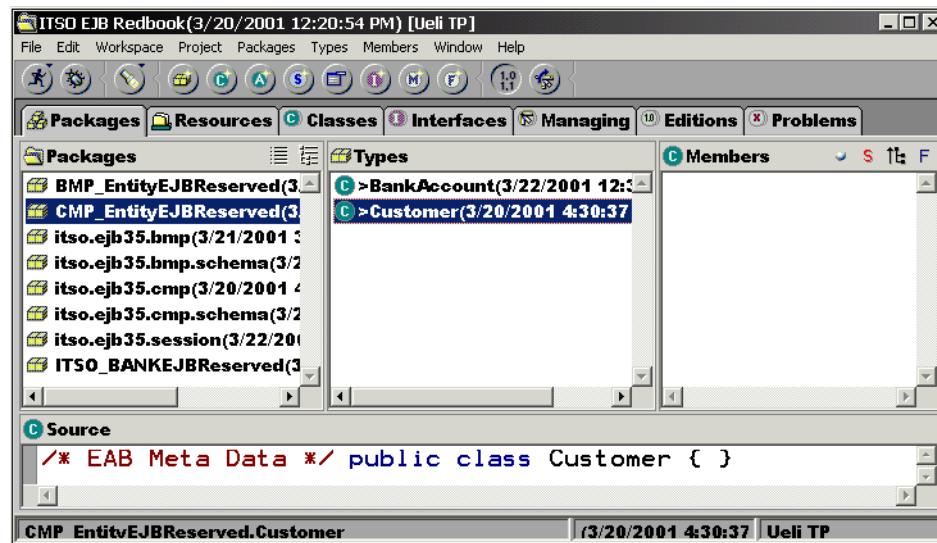


Figure 5-2 EJBReserved package

The EJBReserved package contains skeleton classes for the contained enterprise beans, but no Java code is associated with these classes. The metadata is stored in the repository.

The Java code of the enterprise beans is stored in regular packages, such as itso.ejb35 cmp.

Types or Properties pane

The Types or Properties pane shows the classes or fields associated with the group or EJB selected in the Enterprise Beans pane. There are three views in the Types pane (Figure 5-3):

- ▶ Bean and interfaces
- ▶ Bean, interfaces and deployed code
- ▶ Properties (fields)

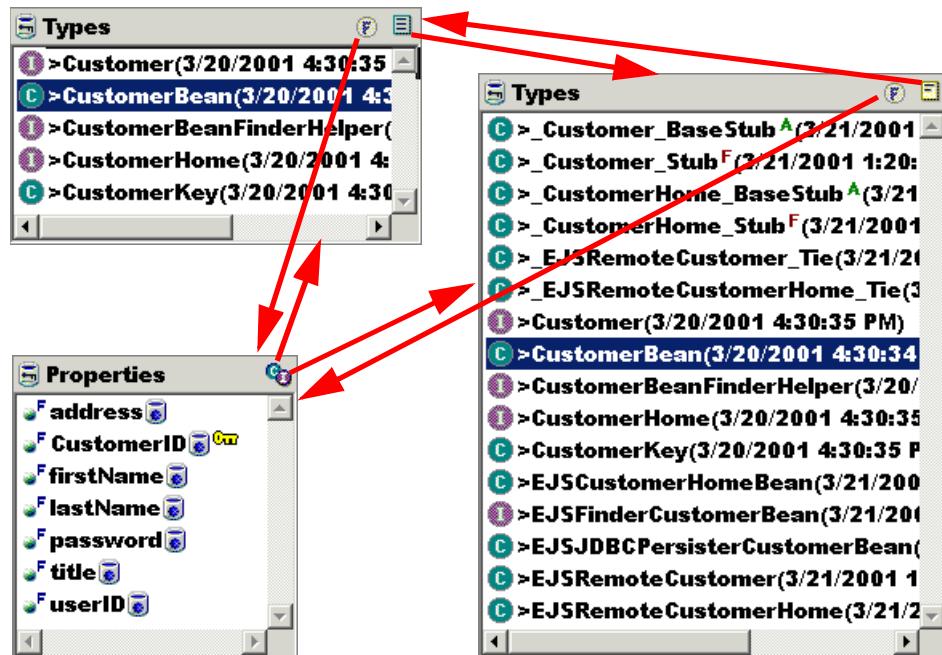


Figure 5-3 Types or properties pane

Switching between the three views is accomplished through the icons on the top-right of the pane:

- ▶ The icon displays the properties.
- ▶ The icon displays the types (classes and interfaces).
- ▶ The icon toggles between showing only the bean definition or also all the generated deployment classes.

The icons displayed next to properties indicate behavior:

- ▶ The icon next to a property indicates a container-managed (CMP) field.
- ▶ The icon denotes the key field of the bean.

Members pane

The Members pane lists methods and fields of the type selected in the Types pane (Figure 5-4).

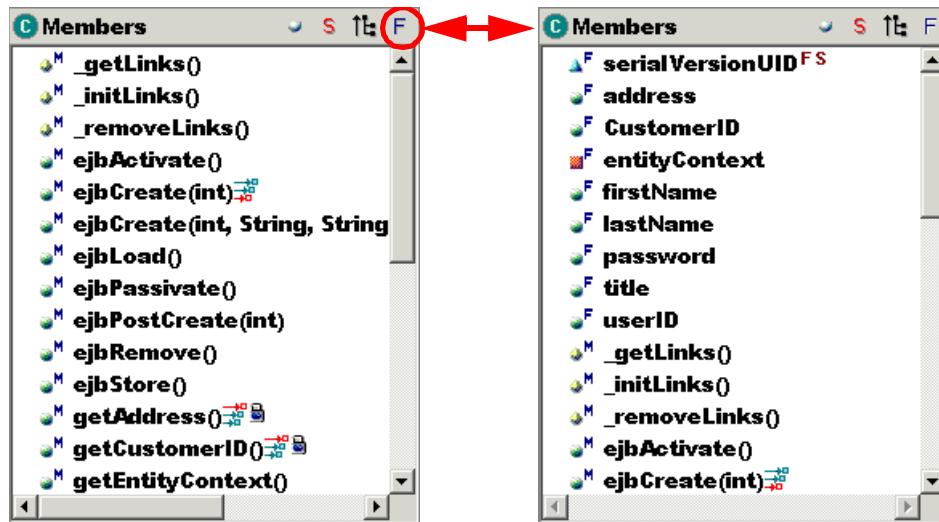


Figure 5-4 Members pane

A number of views are available through the four icons on the top:

- ▶ **Public** methods and fields only (default is public, protected and private)
- ▶ **Static** methods and fields only (default is static and instance)
- ▶ **Inherited** methods (default is methods of selected type only)
- ▶ **Fields** and methods (default is methods only)

The four icons are independent, for example, you can select to view only public static fields and methods.

The icon with the arrows indicates a method promoted to the remote or home interface. The icon indicates a read-only method.

Source pane

The Source pane shows the Java source code of the type selected in the Types pane, or of the method or field selected in the Members pane.

You can edit the code in the Source pane and save the changes. You can set breakpoints from the context menu, or *Edit -> Breakpoint*, or double-click in the left border stripe of the pane.

Actions

All the actions to perform a step in the development process are driven from the EJB page, by selecting an entry in the Enterprise Beans, Types, or Members pane, and an action from a toolbar icon, a toolbar menu, or the context menu.

Toolbar icons

In addition to the standard Workbench toolbar icons, the EJB page provides tailored icons for EJB groups and enterprise beans:

- ▶ The  icon opens a SmartGuide to add an EJB group.
- ▶ The  icon opens a SmartGuide to add a bean to an EJB group.
- ▶ The  icon opens a SmartGuide to add an access bean to an enterprise bean (see Chapter 11. “Access beans” on page 205 for more information).
- ▶ The  icon opens the EJB server configuration (see “EJB server configuration” on page 91 for more information).
- ▶ The  icon opens the map browser to define the mapping of CMP entity beans to database tables (Figure 7-9 on page 124).
- ▶ The  icon opens the schema browser to define the tables in a relational database for the CMP entity beans (Figure 7-8 on page 123).
- ▶ The standard M and F icons on the toolbar open a SmartGuide to define a method or field for a class.

Action from the Enterprise Beans pane

The easiest way to perform an action for an EJB group or an enterprise bean is to use the context menu, which is identical to the EJB menu (Figure 5-5).

The major actions in the sequence that would be used when developing a new enterprise bean are:

- ▶ *Add -> EJB Group*: define a new EJB group to contain enterprise beans.
- ▶ *Add -> Enterprise Bean*: define an enterprise bean in an EJB group (also *Add -> Enterprise Bean with Inheritance* if the new bean inherits from an existing bean).
- ▶ *Add -> Association*: define a relationship between two enterprise beans.
- ▶ *Add -> CMP Field*: define fields for a CMP entity bean.
- ▶ *Add -> Schema and Map from EJB Group*: generate the database tables and the mapping of the CMP enterprise beans to the tables.

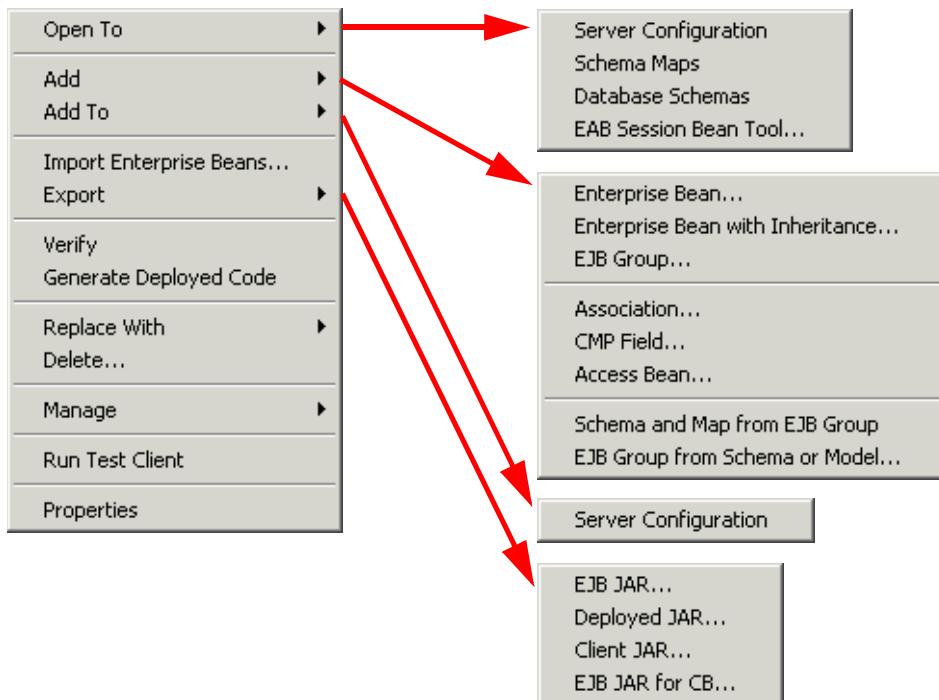


Figure 5-5 EJB menu

- ▶ *Open To -> Database Schemas*: open the schema browser to check and modify the table definitions.
- ▶ *Open To -> Schema Maps*: open the map browser to check and modify the mapping of the enterprise beans.
- ▶ *Add -> Access Bean*: optionally define access beans.
- ▶ *Properties*: set the deployment properties for an enterprise bean.
- ▶ *Verify*: check correctness and completeness of all definitions.
- ▶ *Generate Deployed Code*: generate the deployment classes for a bean or a group.
- ▶ *Add To -> Server Configuration*: add the EJB group to an EJB server to test the enterprise beans (also *Open To -> Server Configuration* once the group has been added to a server).
- ▶ *Run Test Client*: after starting the persistent name server and the EJB server to test the enterprise beans.
- ▶ *Export -> Deployed JAR*: generate a JAR file for deployment to WebSphere without remapping of entity beans to database tables.

- ▶ *Export -> EJB JAR*: generate a JAR file for deployment to WebSphere with remapping.
- ▶ *Export -> EJB JAR for CB*: generate a JAR file for deployment to WebSphere Enterprise (CB Connector).
- ▶ *Export -> Client JAR*: generate a JAR file for client programming.

There are a few actions for special activities:

- ▶ *Import Enterprise Beans*: import existing enterprise beans from a JAR file into a group.
- ▶ *Open To -> EAB Session Bean Tool*: create a session bean for an existing EAB command, for example, and interaction with a CICS or IMS system.
- ▶ *Add -> EJB Group from Schema or Model*: create an EJB group from an existing database schema (in the schema browser) or from a Persistence Builder model. This is reverse engineering when the database tables already exist.

The rest of the actions have to do with versioning and team support:

- ▶ *Delete*: delete an enterprise bean or group.
- ▶ *Replace With*: replace the definition with another edition from the repository.
- ▶ *Manage*: versioning actions, such as, Version, Release, Create Open Edition, and Change Owner.

Actions for types

There are no extra actions related to enterprise beans for types. All standard actions are available from the context menu or from the toolbar Types menu.

Actions for properties

If the Types pane is toggled to show the properties, then you have two major actions available from the context menu or from the toolbar Properties menu: to promote a normal field to be a CMP (container-managed) field, and to make a field a key field (Figure 5-6).



Figure 5-6 Properties actions

Actions for members

In the Members pane there are no special actions for fields, but there are some special actions for methods (Figure 5-7).

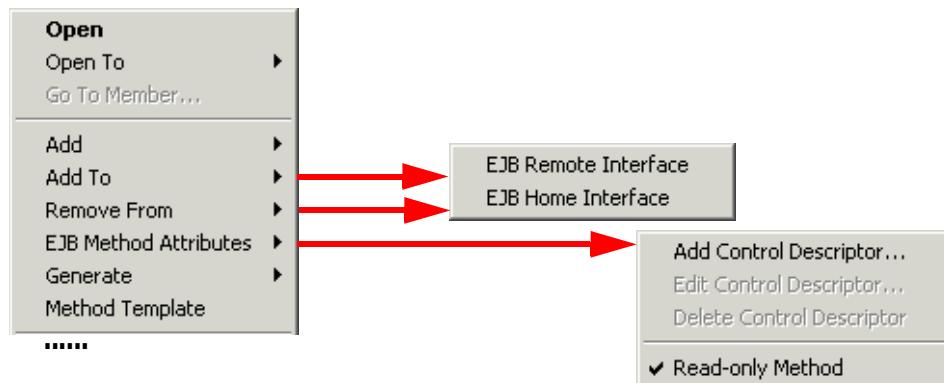


Figure 5-7 Member method actions

The special actions are:

- ▶ Add a method (for example, a business method) to the EJB remote interface.
- ▶ Add a finder or create method to the EJB home interface.
- ▶ Remove a method from the remote or home interface.
- ▶ Set deployment properties (control descriptor with transaction, isolation, and run properties) that overwrite the properties defined for the enterprise bean.
- ▶ Specify that a method is read-only (this is a performance option in the EJB container).

Actions in the Source pane

There are no special EJB related actions in the Source pane.

Testing enterprise beans

Enterprise beans are tested in VisualAge for Java using the WebSphere Test Environment. The persistent name server and an EJB server must be started. This is described in “Testing enterprise Java beans” on page 91.

Management of EJB groups and enterprise beans

Here are some hints and tips in regard to management of EJB groups and enterprise beans.

Management of EJB groups

EJB groups are a VisualAge for Java technique of grouping enterprise beans into management groups. The management actions for an EJB group are:

- ▶ **Version:** Version the group in the EJB pane; this will version the special package (groupEJBReserved). You should version the Java code in the deployed packages at the same time.
- ▶ **Delete:** Delete the group in the EJB pane; this will delete the special package (groupEJBReserved). The deleted group (and its enterprise beans) still exists in the repository.
- ▶ **Load:** Load an existing EJB group into the Workbench. Select *Add -> EJB Group*, then select *Add EJB group(s) from the repository* and select existing groups to be loaded.
- ▶ **Rename:** There is no rename action. However, you can delete the group, create a new group, and then load the enterprise beans from the repository (see Load and Move in the next section, “Management of enterprise beans”).

Management of enterprise beans

Enterprise beans within an EJB group can also be managed individually:

- ▶ **Version:** Version the enterprise bean in the EJB pane; this will version the class in the special package. You should version the Java code in the deployed packages at the same time.
- ▶ **Delete:** Delete the enterprise bean in the EJB pane; this will delete the class in the special package. The deleted bean still exists in the repository.
- ▶ **Load:** Load an existing enterprise bean into a group in the Workbench. Select *Add -> Enterprise Bean*, then select *Add enterprise beans from the repository* and select existing beans to be loaded.
- ▶ **Rename:** There is no rename action. You have to delete the enterprise bean and recreate it.
- ▶ **Move** to another group: You can move an enterprise bean to another group by deleting it first, then loading it into the new group from the repository.

Team development

VisualAge for Java Enterprise provides team development in two ways:

- ▶ Team development feature using a LAN server with EMSRV with a shared repository
- ▶ External version control feature using an external vendor tool through the source code control interface (SCCI) API

Restriction: For enterprise beans development, external version control is not supported, because the meta information about the enterprise beans is stored in the repository only and cannot be stored and versioned in an external system.

Team development using a shared repository

VisualAge for Java provides a comprehensive team development environment:

- ▶ The EMSRV control program runs on a LAN server, managing the shared repository. The LAN server can run on Windows NT, Windows 2000, AIX, OS/2, Netware, HP-UX, Solaris, and Redhat Linux.
- ▶ VisualAge for Java is not required on the team server.
- ▶ Each developer has VisualAge for Java Enterprise installed and connects to the shared repository instead of a local repository.

Here are some basic facilities associated with team development:

- ▶ The major operations in team development are:
 - version (type, package, project): freeze the code
 - release (type, package) a version to the next higher level
 - create open edition of a version for further changes
- ▶ Each development component (project, package, type) has an owner and certain functions can only be performed by the owner.
- ▶ For each package, the owner defines a group of developers that are allowed to change the types within the package. This is called the package group.
- ▶ Each developer within the package group can create an open edition and version the types, however, only the owner of the type can release a version to the package. This means that the type owner has to merge the changes of all developers before releasing the type. For simplicity, we suggest that each class is maintained completely by one developer.
- ▶ Only the package owner can version and release a package.
- ▶ Only the project owner can version a project.

For detailed information about team development, see *VisualAge for Java Enterprise Version 2 Team Development*, SG24-5245. Although this redbook was written for Version 2 of VisualAge for Java, the basic concepts are still valid. However, at that time there was no EJB development feature, and developing enterprise beans in a team environment requires more rigid procedures.

Team development for enterprise beans

All development of enterprise beans is done on the EJB page of the Workbench. We described in “EJBReserved package” on page 67 that meta information about enterprise beans is stored in a special package for each EJB group. The source code of the beans and the deployment classes is stored in regular packages. In addition, meta information about the schema and mapping of CMP entity beans is stored in special classes in the reserved package or in a regular package.

This meta information leads to special treatment of enterprise beans development in a team environment.

Here are some of the factors that affect how a team can work on enterprise beans:

- ▶ When you create an EJB group, you must be the owner of the associated project, because creating an EJB group will automatically generate its corresponding reserved package in the project.
- ▶ When you create an EJB, you must be a package group member of both the associated source code package and the reserved metadata package. This is because creating an EJB will generate a Java source code class in the source code package and a metadata class in the reserved package.
- ▶ You must be the owner of an EJB group before you can version it, because the corresponding reserved package will also automatically be versioned when you version the group. To version or release an EJB, you must be the owner of the EJB. In VisualAge for Java 3.5.3, any developer in the package group can version an EJB, but only the owner can release it.
- ▶ Because the metadata is not Java-based, it cannot be exported as a Java file, even if its representation in the IDE is a Java class. If you export your project as a Java file, the generated code will be preserved but the metadata will be lost. The only way you can preserve the metadata is to export the versioned package in repository format.

Three approaches to team development of enterprise beans

In the VisualAge for Java team environment, you can work with EJBs on one of three different levels:

- ▶ **Project isolation**—a project is restricted to a single EJB group and a single developer.
- ▶ **EJB group isolation**—a project can contain multiple EJB groups, but each EJB group is restricted to a single developer and all of the EJB group developers share a single user ID that is unique to the project.
- ▶ **EJB isolation**—a project can contain multiple EJB groups that hold multiple EJBs, but each EJB is restricted to a single developer and all of the EJB developers share a single user ID that is unique to the project. (At no time should more than one developer be working on the same EJB at the same time.)

The first option is the easiest to implement. It may dictate that you create multiple projects, up to the level that each project contains only one EJB group.

The second and third option forces you to have developers share a user ID within VisualAge for Java. Normally, each developer has his or her own user ID, but because of the fact the developers who version an EJB group must own the reserved package, it is necessary to use the same user ID. Each developer may have his or her own user ID for other development, but they must use the shared user ID when working on enterprise beans. You can switch user IDs while working with VisualAge for Java through *Workspace -> Change Owner*.

Because project-level team development of EJBs involves the least complexity and effort, we generally recommend that you use project-level development and share and reuse code at the project level. However, there may be situations where you have to work with EJBs at the EJB group level or the EJB level.

Note that when multiple developers work on one group or enterprise bean, VisualAge for Java does not enforce a safe procedure. It is up to the team to enforce the guidelines of safe development, for example, that at no time more than one developer works on the same EJB at the same time.

Attention: In VisualAge for Java 3.5.3, there is some relax from the strict rules cited here. You can have multiple developers with their own IDs working on an EJB group. However, only one developer at a time should work on one EJB, and only one developer at a time should work with the schema and mapping.

Recommendations for team development

Here are recommendations when developing enterprise beans as a team:

- ▶ Define small projects, each containing one or a few EJB groups, and one developer who owns the project.
- ▶ Enterprise beans that are related through associations or inheritance must be within the same group.
- ▶ Separate entity beans and session beans into different groups.
- ▶ Define a shared user ID for each project where more than one developer works on enterprise beans. (This is not necessary any more in VisualAge for Java 3.5.3.)
- ▶ If more than one developer works on a group or enterprise bean:
 - Always version the work before another developer works on the same group or enterprise bean.
 - Load the latest version from the repository before changing the group or enterprise bean.
 - Never have multiple open editions of an EJB group or enterprise bean by multiple developers.
- ▶ Only one developer at a time should work with the schema and mapping. Save the schema and mapping information of CMP entity beans into the reserved package of the group.
- ▶ Perform all versioning actions from the EJB page of the Workbench and not from the project view. Versioning of an EJB group or enterprise bean automatically versions the reserved package or the meta class within the reserved package.
- ▶ Export the reserved package together with the deployed code in repository format for archiving.



WebSphere Test Environment

VisualAge for Java includes the WebSphere Test Environment (WTE), which contains the WebSphere Application Server, Advanced Edition runtime environment. WTE provides the runtime environment for JSPs, servlets and EJBs, enabling developers to deploy and test their applications inside VisualAge for Java.

The combination of a development tool and runtime environment gives unique capabilities to the developer to build and test applications within the same environment. There is no need to use an external testing server, because VisualAge for Java already integrates one.

The developer also does not have to write a client to test enterprise beans. VisualAge for Java provides a universal EJB test client, that can also be used for testing EJBs deployed on another server, such as WebSphere Application Server.

All these advanced features dramatically improve the productivity of the developer.

WebSphere Test Environment overview

The WebSphere Test Environment (WTE) is a feature that has to be loaded into the Workbench. However, this is done automatically for you when you add the EJB development environment to the workspace, because WTE is one of the prerequisite features.

Figure 6-1 shows the major functions of the WTE:

- ▶ Servlet engine with Web applications consisting of servlets and JSPs
- ▶ JSP execution monitor for JSP debugging
- ▶ EJB server to host enterprise beans
- ▶ Persistent name server for registering data sources and enterprise beans
- ▶ EJB test client to test the functions of enterprise beans running in the EJB server

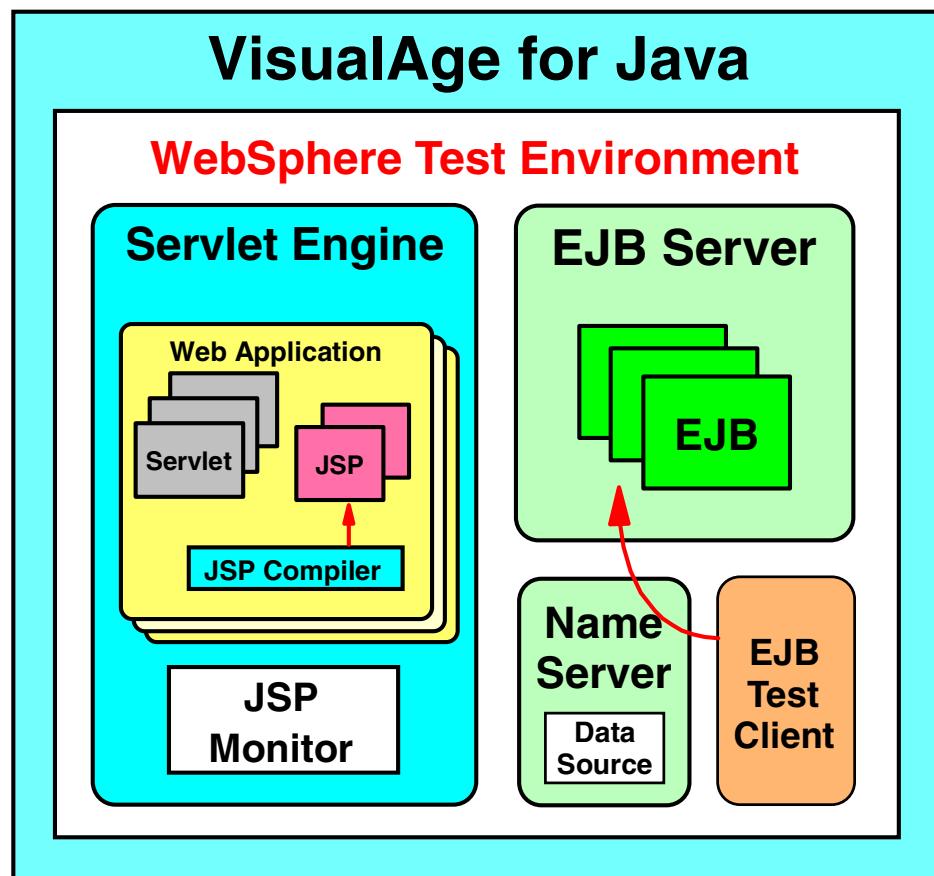


Figure 6-1 WebSphere Test Environment overview

WTE allows you to efficiently test servlets, JSP files and EJBs for WebSphere. The incremental compilation of VisualAge for Java allows you to continue debugging your programs every time you update the code inside the methods without the need of restarting WTE.

Each of the servers can be started and stopped independently, with the exception that the EJB server requires the persistent name server for registering the enterprise beans that are started within the EJB server.

VisualAge for Java Version 3.5 introduced the WebSphere Test Environment Control Center, from which the servlet engine, the persistent name server, and the JSP execution monitor are controlled.

The EJB server is controlled through its own configuration facility from the EJB page of the Workbench (see “Testing enterprise Java beans” on page 91).

The WebSphere Test Environment Control Center is launched by selecting *Workspace -> Tools -> WebSphere Test Environment* (Figure 6-2).

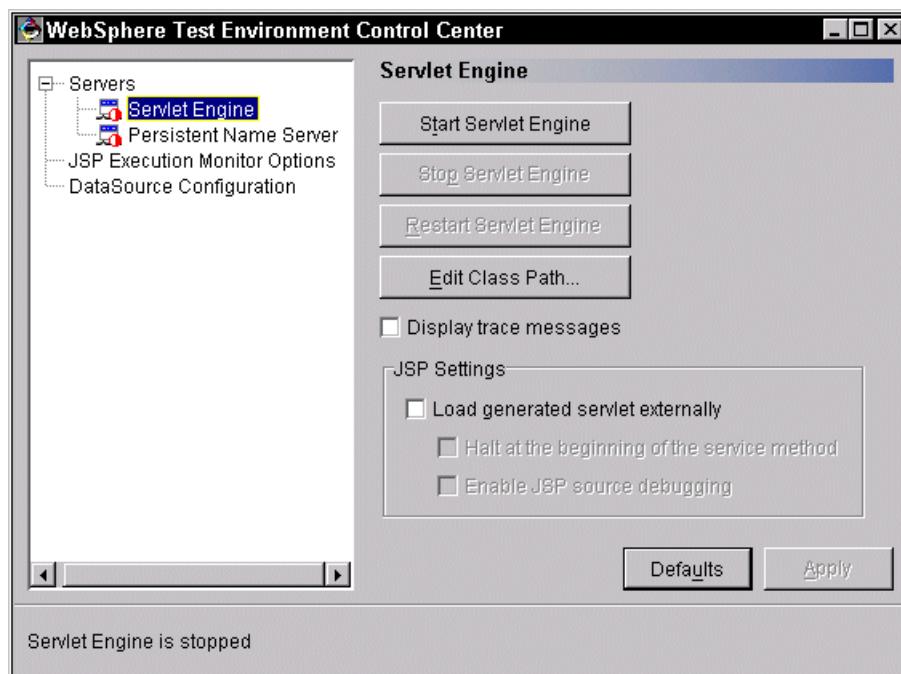


Figure 6-2 WebSphere Test Environment Control Center

Let's take now a closer look at each of the components.

Servlet engine

The servlet engine is used for deploying and testing your servlets and JSPs. It also contains a Web server that listens on port 8080 by default and can serve static HTML files and images, therefore, enabling the testing of a complete Web application.

A Web application usually consists of HTML files, JSPs, and servlets. WTE comes with a default application (default_app), within which servlets are running. If you want to organize your code differently, you can create multiple Web applications. For more information on configuring Web applications, you can read the documentation in VisualAge for Java, or consult the redbook *How about Version 3.5? VisualAge for Java and WebSphere Studio Provide Great New Function*, SG24-6131.

In this chapter we will concentrate more on the components of WTE that are related to EJB testing.

Persistent name server

The persistent name server provides the naming service (JNDI) to applications running inside VisualAge for Java. It holds references to data sources and homes of enterprise beans.

JSP execution monitor

When writing JavaServer Pages, the developer can use this tool for monitoring the execution of JSPs. The monitor displays the source file, the generated JSP Java code, and the HTML output of the JSP. This is very useful when testing JSPs.

Data sources

One of the features of JDBC 2.0 is the ability to use a DataSource object to get a connection to a database. A data source adds an abstract layer over a database to enable connection pooling. Using data sources allows a Java program to use a connection to a database for as long as it needs it. After that, the connection will be returned to the pool and be reused instead of being destroyed. This makes the program faster, as opening a database connection requires a lot of time.

Differences between WTE and WebSphere Application Server

The WebSphere Test Environment is a subset of the WebSphere Application Server (WAS), Advanced Edition. This means that although WTE encompasses the runtime for JSPs, servlets, and EJBs, it does not contain the entire WAS.

The WebSphere Test Environment offers the following:

- ▶ Lightweight runtime environment that loads quickly
- ▶ Standalone all-in-one unit testing
- ▶ No dependency on WAS installation or availability
- ▶ No dependency on an external database unless entity bean support is required
- ▶ Ability to debug live server-side code using the IDE debugger
- ▶ Support for multiple Web applications

As a subset of the WebSphere Advanced Server, the WTE does not offer certain features that are offered by WAS:

- ▶ WebSphere administration server and services
- ▶ XML configuration tool, which produces XML grammar
- ▶ WAS personalization APIs
- ▶ Work load management
- ▶ Security services including
 - Secure Socket Layer (SSL) and secure HTTP (HTTPS)
 - HTTP-style user ID/password authentication challenge
 - Security context/API on enterprise beans
 - Security APIs on servlet sessions or other security classes typically involved in sign on, authentication, or authorization
- ▶ JetAce tool
- ▶ EJB instance pooling

Using the WebSphere Test Environment

In this section we describe how you can start, stop and configure the WebSphere test environment services using the Control Center.

Servlet engine

Use the servlet engine to run one or multiple Web applications inside VisualAge for Java. The WTE servlet engine allows you to configure the servlet class path and other parts of a Web application. Refer to Figure 6-2 on page 81 for configuration of the servlet engine.

Servlet engine class path

When serving servlets or Java bean classes that reside in the workspace, be sure to edit the class path in order to add the projects that contain the servlets and Java beans. To edit the class path, click the *Edit Class Path* button.

Figure 6-3 shows how to tailor the class path by selecting projects loaded in the Workbench.

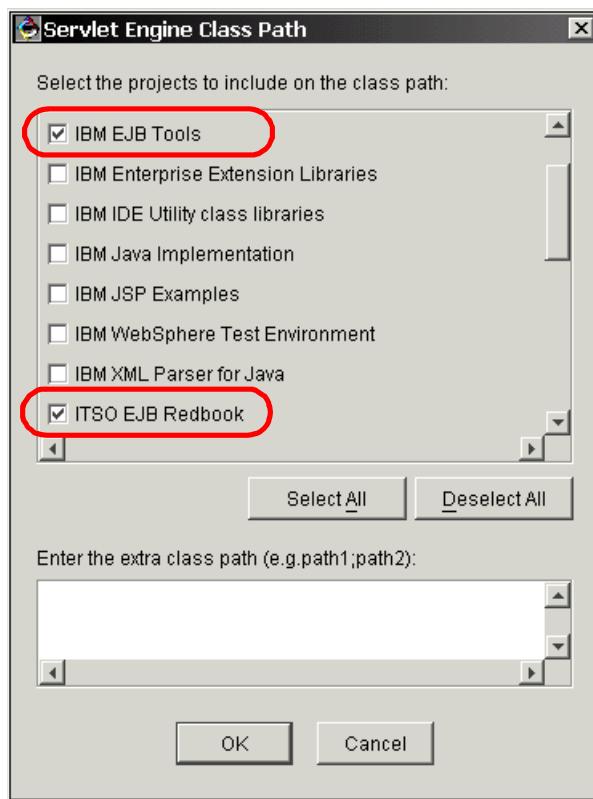


Figure 6-3 Servlet engine class path

Select the project that contains your servlets, for example, *ITSO EJB Redbook*, as well as projects that contain required code. To test servlets or JSPs that access EJBs, you should select the *IBM EJB Tools* project.

If you get messages about classes not found when testing servlets and JSPs, it is usually this class path setting that is incomplete. After the class path has been modified, you must restart the servlet engine for changes to take effect.

Servlet engine settings

The other settings for the servlet engine (Figure 6-2 on page 81) are as follows:

- ▶ *Display trace messages*—displays more messages in the Console window when the servlet engine is started.
- ▶ *Load generated servlet externally*—compiled JSPs are not loaded into the Workbench, instead they are run and debugged from the external directory in the WebSphere Test Environment project resource directory:
`d:\IBMVJava\ide\project_resources\IBM WebSphere Test Environment\temp\JSP1_0\default_app`
- ▶ *Halt at the beginning of the service method*—the debugger is invoked when the service method in the compiled JSP is called. At this point you can step through the compiled JSP code.
- ▶ *Enable JSP source debugging*—instead of showing the generated Java code of a servlet, the debugger shows the original JSP source code. At this point you can step through the JSP source, however, debugging facilities are limited.

We recommend that you use the *Load generated servlet externally* option, because importing the generated JSP Java code tends to fill the repository with unnecessary code. For debugging we recommend that you use the *Halt at the beginning of the service method* option; JSP source debugging does not seem very effective and we would rather suggest for you to use the JSP monitor for that purpose.

Starting the servlet engine

You start the servlet engine by selecting *Servers -> Servlet Engine*, and clicking the *Start Servlet Engine* button. You can check the Console window for the status and the trace messages of the servlet engine (Figure 6-4). They are displayed under the following process:

```
com.ibm.ivj.control.WebControlCenter.main()->
com.ibm.ivj.control.node.ServletEngineRunner.main()
```

Once you have the WebSphere Test Environment running in VisualAge for Java, you can serve JSP and HTML files from the designated document root. The default document root is set to the directory:

```
d:\IBMVJava\IDE\project_resources\IBM WebSphere Test Environment\
hosts\default_host\default_app\web\
```

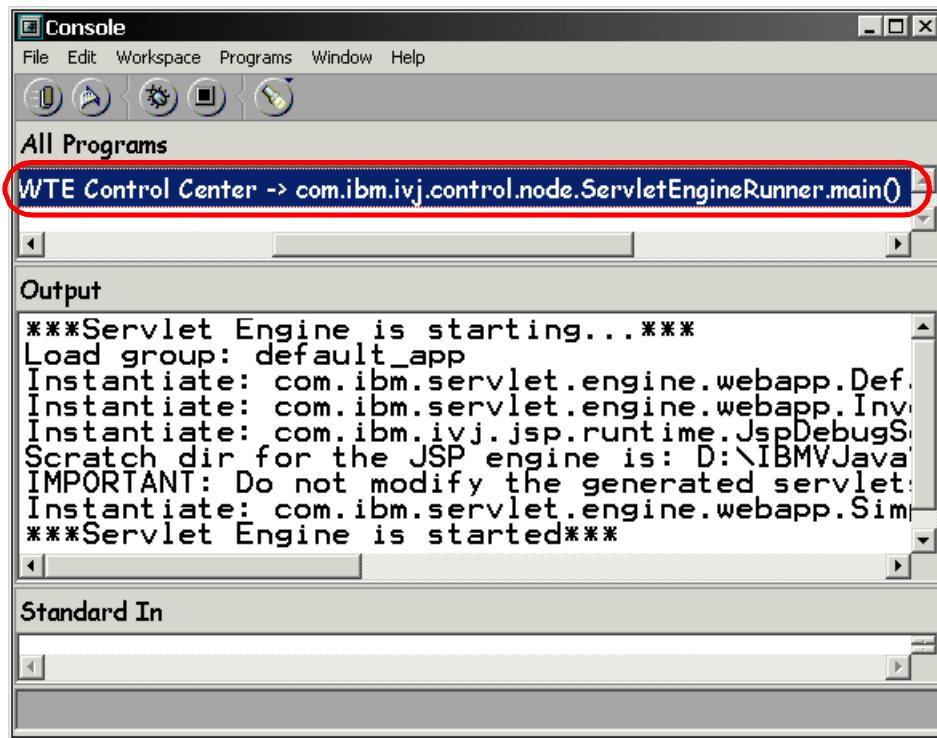


Figure 6-4 Console window with servlet engine started

Stopping and restarting the servlet engine

The control center provides two buttons to stop or restart the servlet engine.

Restarting the servlet engine is required if:

- ▶ Settings have changed, for example, the class path or debugging from external directory.
- ▶ Servlets must be reloaded and initialized, for example, to pick up changes in the servlet configuration file (.servlet file).

Persistent name server

When you select the persistent name server (PNS) from the Control Center, the Persistent Name Server pane appears on the right side (Figure 6-5).

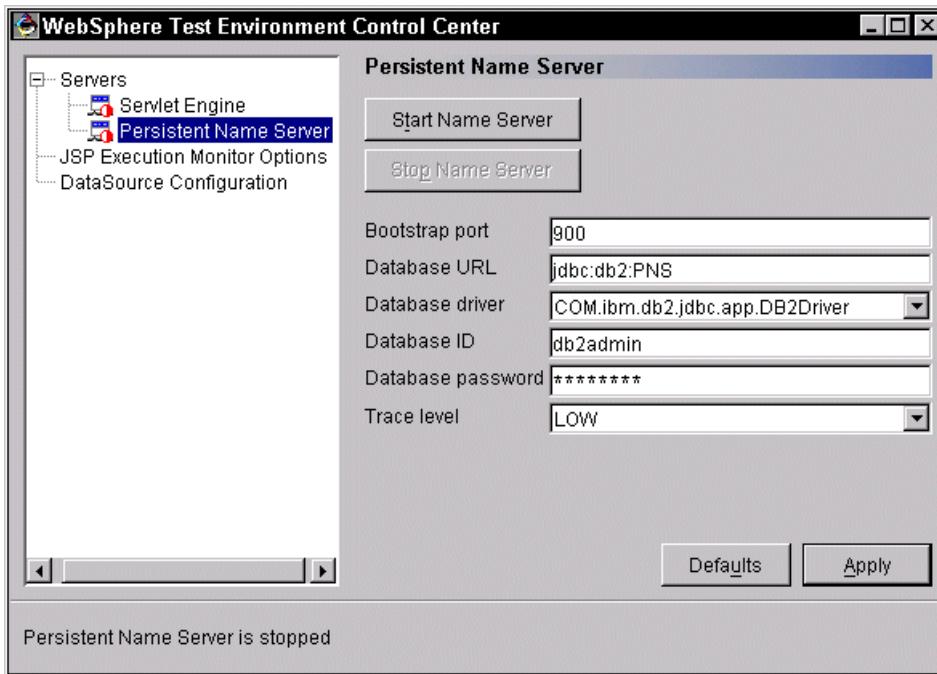


Figure 6-5 Persistent name server dialog

Configuring the persistent name server

Before starting the persistent name server, let's take a look at some settings that you can configure:

- ▶ *Bootstrap port* is by default 900. This is the port from which you can retrieve the initial context, when finding home interfaces.

Note: WebSphere Application Server also uses port 900. If WAS is running on the same machine, you cannot start the persistent name server with port 900. Use a different port in this case, but this requires changing the port in your application code for testing. Alternatively, stop WAS when using the WebSphere Test Environment.

- ▶ *Database driver* defines the type of the persistence store. By default it is the *InstantDB*, *jdbc:idbDriver*, which is a simple database engine. We recommend that you use instead a relational database for better data protection and performance. For IBM environments, you can use DB2.
- ▶ *Database URL*, *Database ID*, and *Database password* must be filled in when you select a relational database as the persistent store. Here, for example, we are using a DB2 database called *PNS*.

You have to create the database manually before you can use this setting, for example, by using the DB2 command:

```
db2 create database PNS
```

The first time you start the persistent name server the tables in the database (or in the InstantDB) are created.

- ▶ *Trace level* can be *low*, *medium* or *high*. Set to high when you want to get a detailed description of all the activities of the persistent name server. Use this setting when dealing with naming problems in your application.

Do not forget to click the *Apply* button at the end for saving the changes.

Starting and stopping the persistent name server

You start the name server by clicking the *Start Name Server* button. After the persistent name server is started, you will see a message in the status line of the Control Center. You can stop the server by clicking the *Stop Name Server* button.

You can check in the Console window the status and trace messages of the persistent name server; they will be displayed under the following process:

```
com.ibm.ivj.control.WebControlCenter.main() ->  
com.ibm.ivj.control.node.NameServerRunner.main()
```

The persistent name server is ready when you see the final Console message:

```
[timestamp] 2135 EJServer      E Server open for business.
```

JSP execution monitor

Use the JSP execution monitor to monitor the execution of JSP source, the JSP generated Java source, and the HTML output (Figure 6-6).

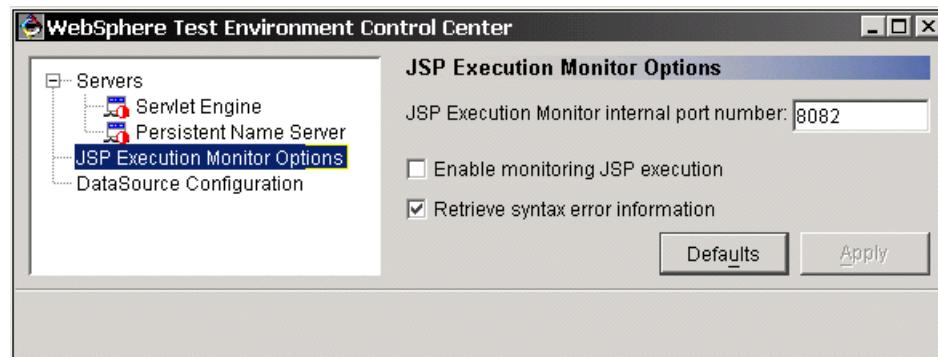


Figure 6-6 JSP execution monitor

The JSP execution monitor is activated by selecting the *Enable monitoring JSP execution* check box. In addition, you can select the *Retrieve syntax error information* option to get more meaningful messages if an error in the JSP source exists and the Java code cannot be generated or imported.

Click *Apply* to activate the setting. Then next time a JSP is invoked, the JSP execution monitor starts up. There is no need to restart the servlet engine.

Data sources

DataSource objects are used for connection pooling to access relational databases. Follow these steps for creating and configuring data sources:

1. Launch the WebSphere Test Environment Control Center.
2. Select *Persistent Name Server*. In the Persistent Name Server pane, click *Start Name Server*.
3. Select *DataSource Configuration* (Figure 6-7). This pane lists all the data sources that have been already defined; initially the list is empty.

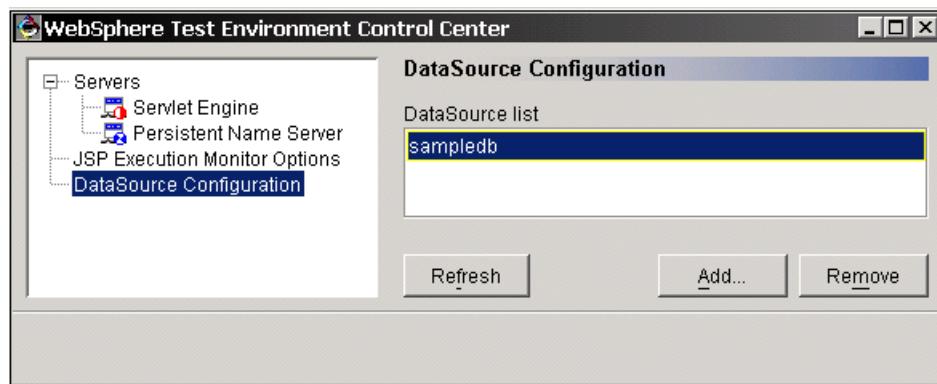


Figure 6-7 *DataSource configuration*

4. In the DataSource pane, click *Add*. The Add DataSource dialog opens. Figure 6-8 shows the data source we defined for this redbook.
5. Enter appropriate values for the data source:
 - In the *DataSource name* field, enter the name that will be used to locate the data source
 - In the *Database driver* field, select the JDBC driver class.
 - In the *Database URL* field, specify the URL of the database.
 - In the *Database type* field, select *JDBC* or *JTA*. *JTA* is for two-phase commit in conjunction with other databases.
 - In the *Description* field, enter an optional description.

We recommend that you accept the defaults for the remaining fields.

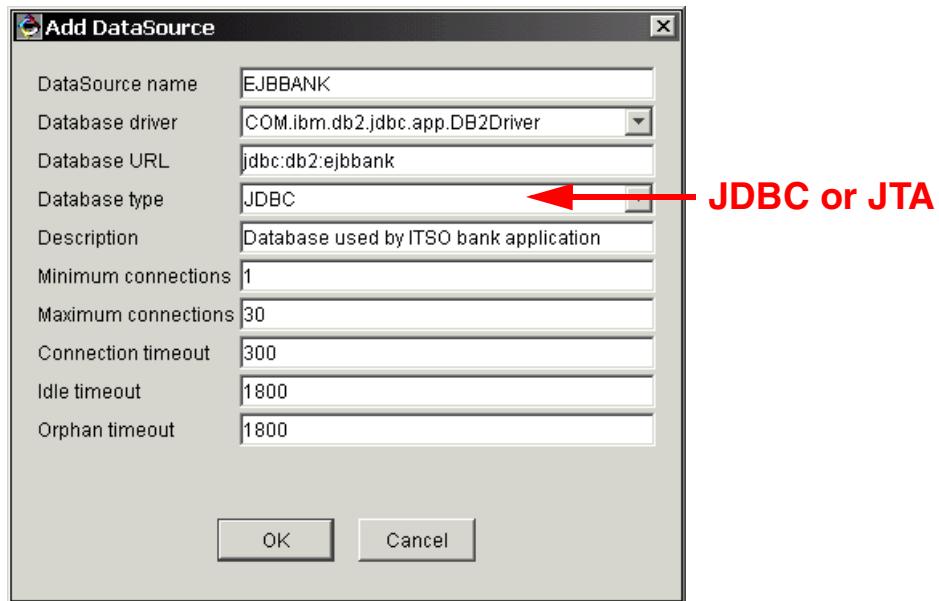


Figure 6-8 Add DataSource dialog

6. Click **OK** to create the DataSource object, and to bind it into the persistent name server context.
7. We suggest that you create a second DataSource object named EJBBANKJTA using the JTA driver. You can use this data source when testing combinations of CMP and BMP entity beans.
The JTA driver is required if enterprise beans participate in distributed transactions (see “Distributed transactions” on page 236).

Note: You have also to add the JDBC driver classes to the *Workspace class path* of VisualAge for Java. This class path is set in the *Windows -> Options* menu in the *Resources* pane.

Once you have created a data source, you are not able to edit it. The only way to change a data source is to replace it by creating a new one with the same name.

You can also create data source objects programmatically using the new DataSource factory API. For more information on that, read the documentation in VisualAge for Java.

Testing enterprise Java beans

This section describes all the steps that you must follow for setting up the test environment for enterprise Java beans.

EJB server configuration

Once you have generated the deployed classes for your enterprise beans, you can create an *EJB server configuration* that consists of one or more EJB groups containing enterprise Java beans that you want to test.

The number of EJB servers that you should configure for your EJB groups depends on the number of data sources your beans require. If you have a set of CMP entity beans that use one data source and another set that use a different one, you should use, for each set of beans, a different EJB group and EJB server.

To create an EJB server configuration:

1. In the Enterprise Beans pane of the EJB page, select one or more EJB group(s).
2. Select from the context menu *Add To -> Server Configuration*. Your EJB group is added to a server configuration and the EJB Server Configuration browser appears (Figure 6-9).

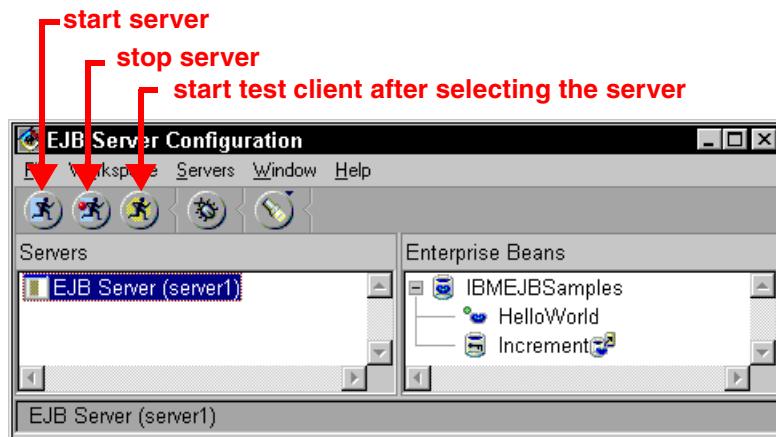


Figure 6-9 EJB Server Configuration browser

In the EJB Server Configuration browser, an EJB server entry is added to the Servers pane. In the Enterprise Beans pane, your EJB group appears. Before using the EJB server, you may have to set the properties.

Setting the EJB server properties

These properties allow you to configure the EJB server. Any changes made will take effect after you start the server. To set the properties, select the EJB server in the EJB Server Configuration browser and from the context menu select *Properties*. The Properties for EJB Server dialog appears (Figure 6-10).

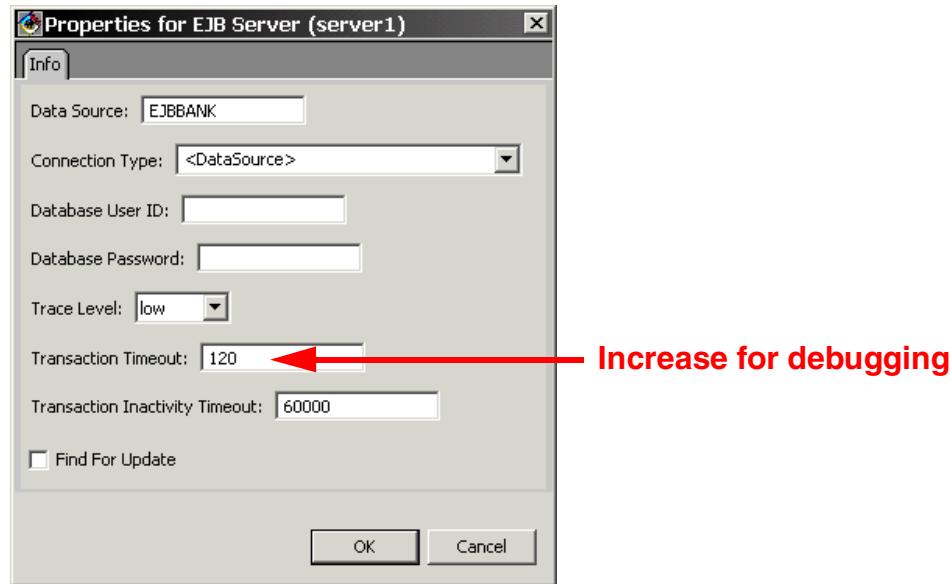


Figure 6-10 EJB server properties

- ▶ The *Connection Type* field determines whether you want to use a database JDBC driver or a data source for defining your entity beans persistent store. If you are going to use a database, select the corresponding database driver, otherwise select <DataSource>.
- ▶ The *Data Source* field contains the name of the data source, if you selected <DataSource> as connection type; otherwise it contains the URL for the database (for example, jdbc:db2:ejbbank).
- ▶ If your database is set up so that a user ID and password is required to access the database, then specify a user ID and password in the *Database User ID* and *Database Password* fields.
- ▶ The *Trace Level* list box specifies whether you want a low, medium, or high level trace displayed in the Console window.
- ▶ The *Transaction Timeout* field defines the number of seconds a transaction is allowed to proceed before it is automatically terminated. The default value is

120 seconds. We recommend that you set a higher value (1200) to avoid transaction timeouts when debugging EJB applications.

- ▶ The *Transaction Inactivity Timeout* field defines the number of milliseconds a transaction can remain inactive before it is automatically terminated. The default value is 60,000 milliseconds.
- ▶ If you want the container to get an exclusive lock on the enterprise bean when the enterprise beans are retrieved (for example, using `findByPrimaryKey`), select the *Find for Update* check box.

Attention: Before starting the EJB server, you must ensure that you have defined the database tables for your entity beans and that they exist in the database system. When following a top-down approach where the tables are not created yet, avoid using the *Create Database Table* from the EJB server context menu in the EJB Server Configuration window. Instead, open the schema browser from the EJB page. In the Tables pane, select the tables for your group and from the context menu select *Export Tables -> Export Selected Tables with Keys to Database*.

Starting EJB servers

Before starting the EJB server, you have to ensure the following things:

- ▶ You have defined the schema and map for your CMP entity beans, if any.
- ▶ The database tables that are used by the schema exist.
- ▶ Your bean classes do not have any errors.
- ▶ You have generated the deployed code for all the beans.
- ▶ You have started the persistent name server.

Now you are ready to start the EJB server. Click on the running man icon in the toolbar, or alternatively, in the Servers pane of the EJB Server Configuration browser, select the EJB server, and from the context menu select *Start Server*.

You can use the Console window for monitoring the EJB server. In the Console, you can determine which enterprise beans are actually running. When the server is ready, you should see the following message for the EJB Server process:

```
[timestamp] 4805 EJServer      E Server open for business.
```

Note: You can change the source code of any bean running in the EJB server configuration while debugging it. As long as you do not change the API of your beans. Changing method signatures or adding methods to home and remote interfaces will automatically stop the servers.

EJB test client

The EJB development environment provides a test client that you can use to test the home and remote interface methods of your enterprise beans. Using the test client, you can create beans, invoke methods and pass user-defined arguments to ensure that they work correctly.

When you run the test client, you can set breakpoints in your enterprise bean implementation and use the debugger as you would do for any application in VisualAge for Java.

Here we will use the test client to test `HelloWorld`, an enterprise bean from the *IBM EJB Samples* project. This project is automatically added to the workspace when you add the EJB Development Environment. If it is not there, go to the Projects page and select *Selected -> Add -> Project*. Then select the *Add projects from the repository* and find the IBM EJB Samples project. When you are done, click *Finish*.

Using the EJB test client

To start the test client:

1. Ensure that you have created an EJB server configuration that contains the enterprise beans that you want to test, and that you have started the EJB server.

The deployed code for the `HelloWorld` bean has already been generated. You only need to add the IBMEJBSamples group to a server configuration and start the EJB server.

2. Select the EJB server that is running; then from the context menu select *Run Test Client*. You can also use the running man icon with yellow background (in the toolbar) to start the test client.

The EJB Test Client window opens (Figure 6-11).

3. In the *JNDI Name* list of the EJB Lookup frame, select the JNDI name of the enterprise bean that you want to test.

The JNDI names of the enterprise beans that you selected, before starting the test client, are available through the pull-down menu.

The JNDI name of an enterprise bean is defined in the *Properties* menu of the bean. By default, VisualAge for Java Version 3.5 assigns a name of:

```
package/beannname ==> itso/ejb35/cmp/Customer
```

VisualAge for Java Version 3.0 assigned the short name of the bean (for example, `Customer`).



Figure 6-11 EJB test client JNDI lookup

Note: You do not need to open a separate test client window for each enterprise bean that you want to test. If you do not find your EJB in the list, simply type in its JNDI name.

4. Click *Lookup*. This constructs an initial context and queries the name server for the selected JNDI name. When an instance of the home is retrieved from the name server, the Home page for the enterprise bean appears (Figure 6-12).

The bean that we have found here is a session bean. Before we can use a session bean, we have to create an instance using the *create* method.

For an entity bean, we create a new instance using the *create* method, or we retrieve an existing instance using the *findByPrimaryKey* method.

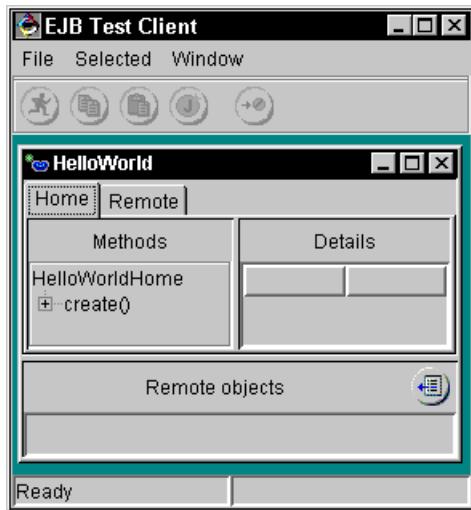


Figure 6-12 EJB test client home interface

5. Select the create method and from the context menu select *Invoke* (or click the running man icon). This will create a new instance of the bean and returns its remote object. Now you are ready to use any of the bean methods that are available through the remote interface (Figure 6-13).

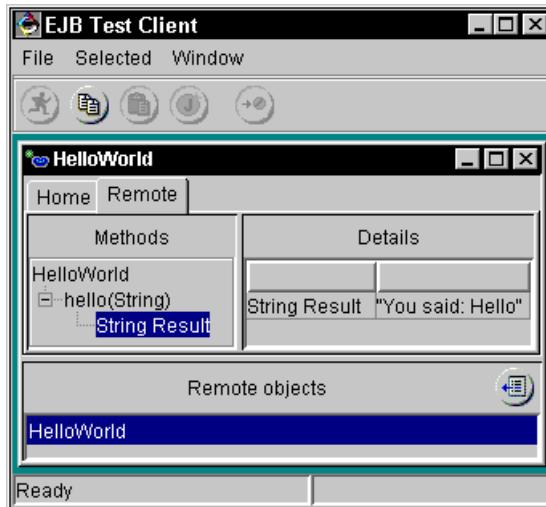


Figure 6-13 EJB test client remote interface

6. The HelloWorld bean has only one method, hello. Select this method and in the Details pane, enter a value of String type as an argument, for example, Hello. Then select the hello method and from the context menu select *Invoke* (or click the running man icon). This executes the method. In the details pane you can see the result of the method (Figure 6-13).

If the test client does not respond, it usually means that the debugger is stopped at a breakpoint. In that case, you should just continue execution in the debugger. The test client does not respond until the execution of the method has ended. Sometimes, breakpoints can be hidden in collapsed processes, so be careful.

You should not terminate the running thread in the debugger, when using the test client. If you do so, then you have to find the test client process in the All Programs/Thread pane in the Debugger window and select *Terminate* from the context menu.

If you terminate the test client using the Windows Task Manager, you will terminate VisualAge for Java.

Debugging an EJB

We have already mentioned that you can use the debugger together with the EJB test client. You should place breakpoints in your bean classes and invoke the corresponding method in the EJB client.

When the breakpoint is reached, the Debugger window will open. Step through your code as usually. Think twice before pressing the *Stop* button. This can cause some problems to the test client (see previous section). If you want to see or delete any of your breakpoints, go to the Breakpoints pane. You can also open the debugger from the menu bar by selecting *Window- > Debug -> Debugger*. For more information on how to use the debugger, consult the documentation of VisualAge for Java.

Check the Console window for the output of the EJB servers and the name server. Very often you may see interesting trace messages displayed there. If you have a problem that persists, you can set the trace level higher for either the EJB server or the name server. The default for both is *low*.

Important: Be careful about transaction timeouts, when debugging your beans. When you invoke a method from the test client, you should go through all the breakpoints in your code before the transaction times out. You can set a higher value to the transaction timeout in the EJB server properties, “Setting the EJB server properties” on page 92.

Advanced features of the EJB test client

This section presents some of the advanced features of the test client, such as testing EJBs deployed in WebSphere Application Server or copying and pasting Java objects inside the test client.

Connecting to remote servers

The test client can be used to connect to a remote WAS server. This allows you to access beans that are running on an EJB server on another machine.

To connect to a remote server:

1. In the Enterprise Beans pane of the EJB page, select any enterprise bean or group and select *Run Test Client* from the EJB menu. The EJB test client opens and displays the EJB Lookup frame.
2. In the *Provider URL* field, specify the host name (and optionally, the port number) for the remote name service. The provider URL must have the following form (where *hostname* is the remote name server's host name and *port* is the port number on which the remote name server is listening):

`iop://hostname:port/`

3. In the *JNDI name* field, type the name of the enterprise bean you want to test and that is running on the remote machine, for example:

`itso/ejb35/cmp/Customer`

Working with multiple remote interface objects

Using the test client, you can create and work with several different remote objects at the same time. To facilitate this, use the Remote objects pane at the bottom of the test client, as shown in Figure 6-14.

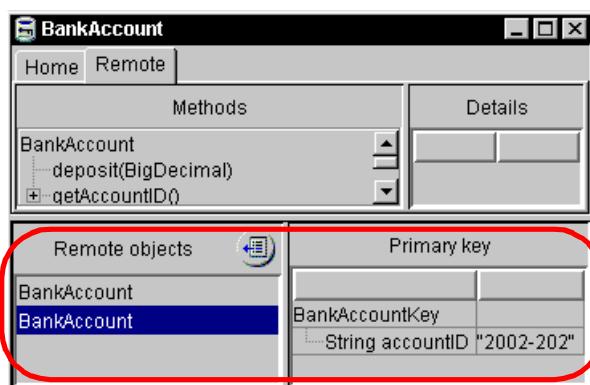


Figure 6-14 Remote objects pane

On the Home interface page, you can create (or find) many different remote objects that are added to the Remote objects pane on the Remote interface page. If the enterprise bean is an entity bean, the bottom half of the page consists of two panes. The left pane contains the remote objects list, and the right pane contains a tree table of the enterprise bean primary key object.

To change the target of the remote method calls, select a different remote object from the list. Subsequent remote method calls will be directed to that object. To remove a remote object from the list, click the *Drop selected* button in the Remote objects pane (not available from a context menu).

Note: This only removes the entry from the list. It is not the same as invoking the remove method on EJBObject. Also note that when you do invoke the remove method, the entry is not automatically removed from the remote objects list.

Displaying additional object features

Sometimes it is useful to view object attributes, such as fields, properties or even methods inherited by superclasses that are not shown by default.

You can modify what is displayed by selecting the root object in the Methods pane and then selecting *Show object features* from the context menu. The Show object features dialog opens (Figure 6-15), which contains check boxes for member fields, normal properties, expert properties, and hidden properties.

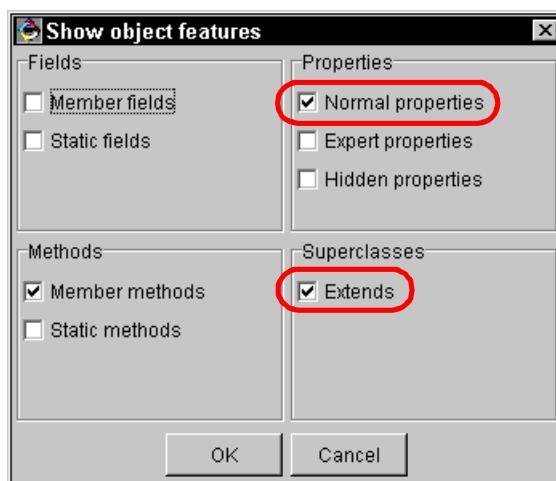


Figure 6-15 Show object features

Here, we have selected the *Normal properties* and *Superclasses* -> *Extends* on the remote object of a BankAccount entity bean. The result is shown in Figure 6-16.

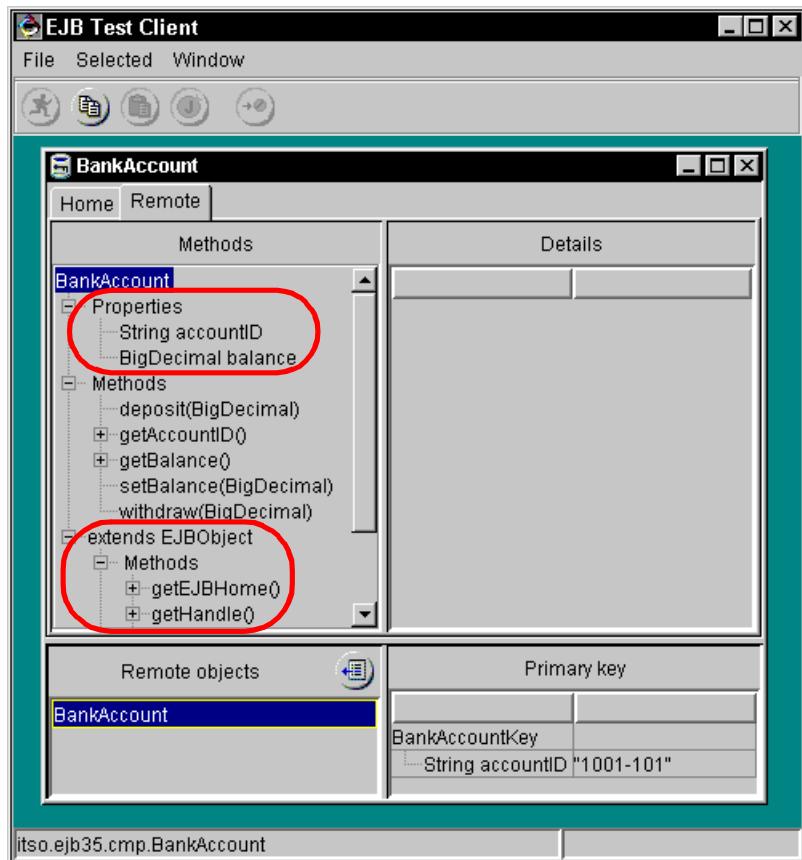


Figure 6-16 Advanced display options

Note: When you use the Show Object Features dialog to display the properties of an enterprise bean, the test client becomes slower, because a remote get method for each property is invoked.

Replacing objects with null references

The test client provides an *Assign null* operation that you can use to replace an object with a null reference. This is sometimes useful for testing how well your implementation deals with null method arguments.

Constructing and inspecting objects

Many times, bean methods use objects as parameters or return types, such as `java.math.BigDecimal`. The EJB test client provides you with two very useful tools, an object constructor and inspector:

- ▶ For example, if you want to set the value to a `java.math.BigDecimal` parameter, select the parameter in the Details pane and from the context menu, select *new(...)*. This opens the Constructor window (Figure 6-17).

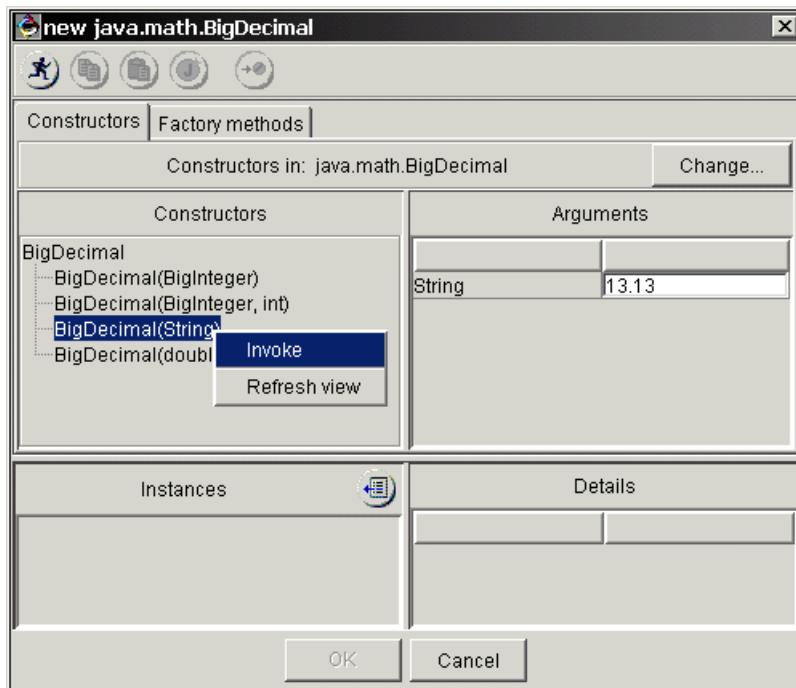


Figure 6-17 Creating a BigDecimal

Select the method in the left pane; complete the parameter in the right pane. Use either `BigDecimal(String)` or `BigDecimal(double)`.

Run the method by selecting *Invoke* in the context menu (or use the running man icon). Click *OK* to close the window and the `BigDecimal` parameter is set.

- ▶ When you want to view the `java.math.BigDecimal` result of a method, select the result in the Details pane and from the context menu, select *Inspect*. This opens the Inspector window where you can see the numeric value (Figure 6-18).

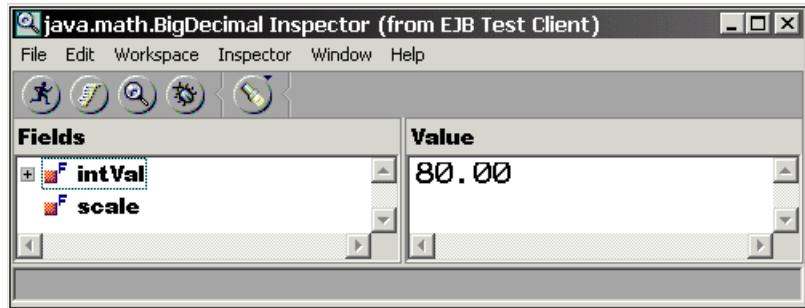


Figure 6-18 Inspecting a result object

Copying and pasting Java objects

Using the *copy*, *paste*, and clipboard tools of the EJB test client, you can reuse the same Java object in different places.

For example, you can easily copy a `BigDecimal` value that you constructed or retrieved in one remote window into another window as a method parameter. Select the `BigDecimal` instance, then from the context menu, select *Copy*. This operation copies the object to the test client clipboard. Select an object parameter in another window of the same test client, and *Paste* from the context menu, and you have copied the complete `BigDecimal` object.

Important: The *Copy* and *Paste* operate on an object reference, not on an object value. Both operations access the same object in memory. Thus, editing an object in one place results in a change of the object in other places.

In another example, the method `remove(Handle)` method on `EJBHome` requires a `javax.ejb.Handle` object. You can obtain an instance of this object by calling the `getHandle` method on the remote object. Select the `getHandle` method in the remote interface page, then from the context menu, select the *Invoke* button.

The result object is an instance of `javax.ejb.Handle` that refers to your enterprise bean object. Select the object instance and *Copy*. Return to the Home interface page and select the `remove(Handle)` method and set the root of the argument tree, `Handle`, (which is initially a null value) and *Paste*.

Tip: This advanced feature of the test client is very useful when testing enterprise beans with associations.

Testing servlets with EJBs

When testing servlets with EJBs in VisualAge for Java you may get error messages after restarting an EJB server:

```
java.rmi.MarshalException: CORBA COMM_FAILURE 3 No; nested exception is:  
org.omg.CORBA.COMM_FAILURE: minor code: 3 completed: No
```

In such cases, it may not be enough to restart the servlet engine. Stop the servlet engine, then start it again. This may solve the problem.

Summary

In this chapter we described how you can use the integrated WebSphere Test Environment inside VisualAge for Java for testing your enterprise Java beans. We focused on how to create data sources, configure and start the persistent name server, the EJB servers, and finally on how to use the EJB test client.

This chapter provides to you all the steps you need to go through before you are ready to start testing your beans. At the end, we discussed some advanced features of the EJB test client that can be very useful when testing real-life scenarios.



Container-managed persistence entity beans

In this chapter, we describe developing container-managed persistence entity beans. Developing a CMP bean is comparably easier than developing a BMP bean. An entity bean with container-managed persistence relies on the container provider's tools to generate Java code that perform data access on behalf of the entity beans. The generated methods transfer data between the entity bean instance variables and the underlying database table columns.

We describe in detail how to use the EJB development environment of VisualAge for Java to develop CMP entity beans.

Persistence basics

An entity bean represents an object view of the data stored in an underlying database. The data access protocol for transferring the state of the object between the entity bean and the database is called *persistence*. As mentioned in “Container-managed persistence (CMP)” on page 26, the EJB container handles the interactions with the data source for a CMP bean. The container uses the `getContainerManagedFields` method of the deployment descriptor, to find out the data members of the CMP bean, that it needs to manage. When we develop a CMP bean, we do not have to be concerned about the persistence mechanism. Therefore, as bean providers, we do not write the database access calls. The advantage of using CMP is that the entity bean can be largely independent of the data source in which the state is stored. It also enables a clear separation of business logic from data access logic.

In this chapter, we develop CMP entity beans using VisualAge for Java SmartGuides and we examine the generated classes and methods.

EJB persistence terms and definitions

Here we list the definitions of some frequently used terms in this chapter:

EJB container	An environment in which the EJBs exist.
Container-managed persistence	An approach where we delegate the access calls required to persist objects in a relational database to the container.
CMP field	The property of an entity object that is to be persisted.
Bean-managed persistence	An approach where we handcraft the code required to persist the EJB state.
Deployment descriptor	A repository of metadata that describes EJBs and enables us to work with EJBs.

Life cycle of EJB instances

Figure 7-1 shows the life cycle of an EJB instance.

The EJB client programs get an `InitialContext` object by specifying the initial context factory and the naming provider URL. Then it performs a lookup using the JNDI name of the bean and gets a reference to a home object of the enterprise bean. Then the client calls the `create` method to create new entity beans. If the bean already exists, the client can use a `finder` method to retrieve a

reference to the remote object of an existing bean. Then the client calls the business methods (for example, `performTask`) on the remote object, which in turn forwards the calls to the bean instance. Then finally the client calls the `remove` method to remove the bean.

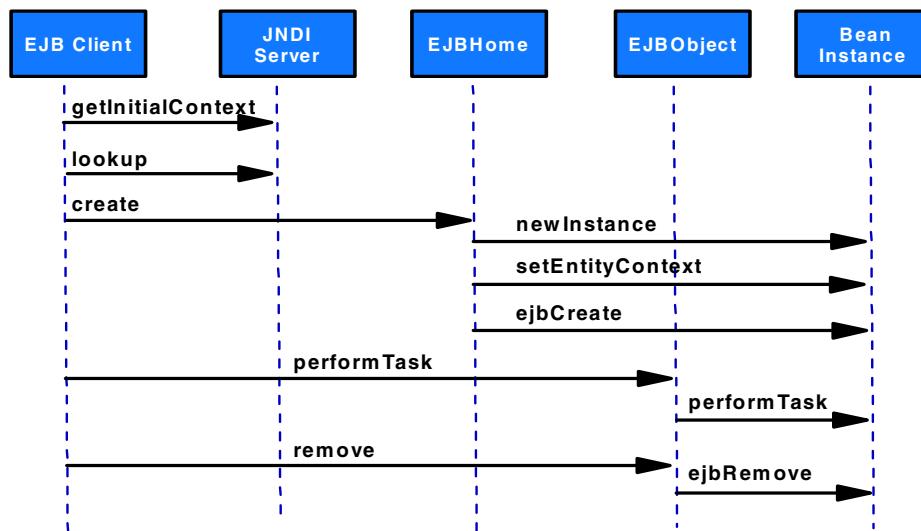


Figure 7-1 Life cycle of an EJB instance in the view of client

In the case of entity beans, removing an entity bean results in deleting a single row or multiple rows in the data source. So generally the `remove` method should not be invoked unless permanent deletion from the data store is desired.

Figure 7-2 shows the life cycle of an entity bean in particular. The term *referenced* in the diagram means that the EJB client has a reference to the remote object of the entity bean.

A client creates an entity object using the home interface, which is implemented by the container. On creating the object, the client obtains a reference to the remote object of the bean.

In a legacy environment, the entity object state may exist in the database even before the bean is deployed.

Similarly, an entity bean may also be created by means of other way, such as creating a new row in the table, and it may be directly deleted from the database without calling the `remove` method.

Multiple clients can access the same entity bean concurrently, but with isolated transactions for each client.

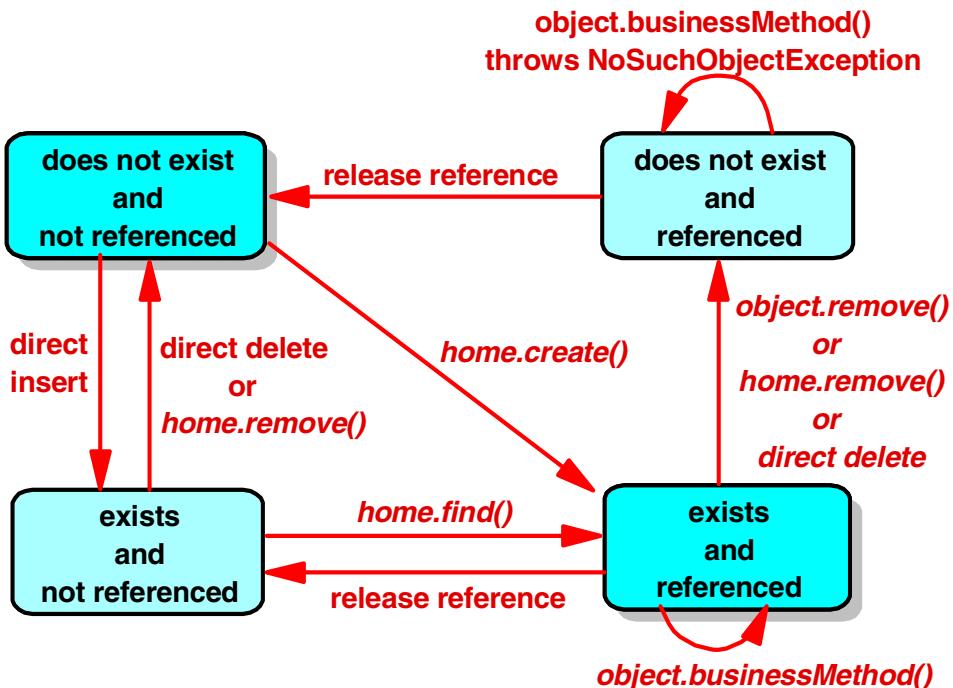


Figure 7-2 Entity bean life cycle

Developing a CMP bean with VisualAge for Java

We first have to set up the EJB development environment as shown in “EJB development environment” on page 466.

Overview

We now create CMP beans using the following steps:

1. Add an EJB group, to hold the enterprise beans.
2. Add or import enterprise beans to the EJB group.
3. Add the home methods and remote methods to the bean class and promote them to the respective home or remote interfaces.
4. Add or define CMP fields.
5. Map the CMP fields to the database schema.
6. Set the deployment descriptor properties.

7. Generate the EJB deployed code
8. Test the EJB beans.

In this chapter we create a CMP entity bean called Customer. There are two other beans, BankAccount and TransactionRecord, which we describe in “Defining the bank model” on page 132. In later chapters, we discuss advanced topics, such as inheritance, associations, collections, and transactions, using all the entity beans.

EJB group

An EJB group is a logical group that allows you to organize enterprise beans. This helps in performing global operations such as exporting the entire group containing all the beans to a JAR file.

There are three ways to add an EJB group in the EJB page.

- ▶ Create a new EJB group and create a mapping to a new database table. This is also called forward engineering or top-down.
- ▶ Create a new EJB group from an existing schema (database table) or Persistence Builder model. This is called backward engineering or bottom-up.
- ▶ Retrieve one or more existing EJB groups from the repository. These groups were developed by another user.

In this section, we will create a new EJB group from scratch. The steps involved are:

1. We will store all the Java code in the *ITSO EJB Redbook* project that was setup for all the examples (see “Using the Web material” on page 492).
2. In the Workbench, click on the EJB tab. The EJB page appears.
3. From the EJB menu, select *Add -> EJB Group*. The Add EJB Group SmartGuide appears (Figure 7-3).
4. For the *Project* field, click the *Browse* button and select the name of the project that you want to contain the EJB group, then click *OK*. In the bottom left portion of the figure, we find **Enter group name*, a reminder to add an EJB group name.
5. There are two ways to add an EJB group. The first option is creating a new EJB group. The second option is to add EJB groups by retrieving them from repository. Here we will use the first option. To add an EJB Group by creating a new EJB group, select the radio button *Create a new EJB group named* and type the name of the EJB group in the text field below this radio button. For our example we type **CMP_Entity**.
6. Click *Finish*. The EJB group is added to the EJB pane.

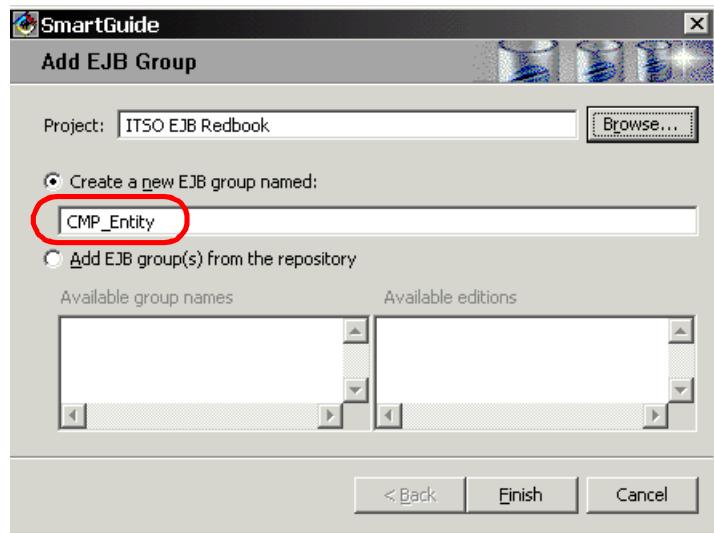


Figure 7-3 Adding an EJB group to the project

Tip: When creating projects, we suggest that you use two packages, one for holding the enterprise beans and related classes, and one for holding the client application code that access the beans.

Creating a CMP bean

Once we have defined the EJB group, we can add enterprise beans to the group in one of the following ways:

- ▶ Creating new enterprise beans
- ▶ Retrieving enterprise beans from the repository
- ▶ Importing enterprise beans from EJB jar files

Here we will create a new CMP bean, named **Customer**. The steps are as follows:

1. In the EJB pane, select the EJB group **CMP_Entity**.
2. In the context menu select *Add -> Enterprise Bean*. The Create Enterprise Bean SmartGuide appears (Figure 7-4).

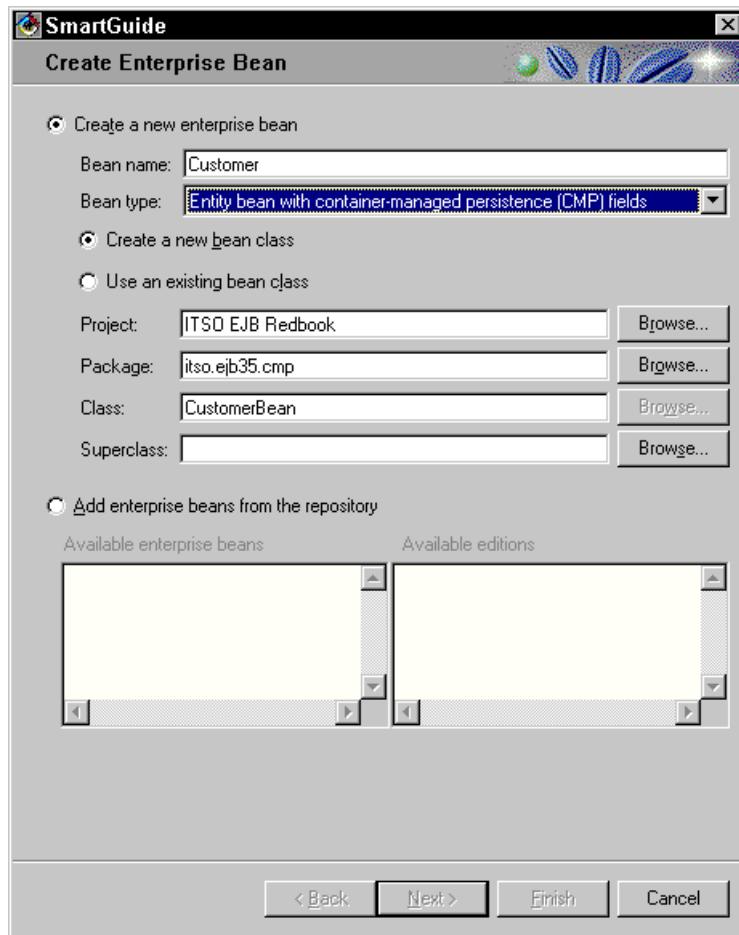


Figure 7-4 Creating a CMP entity bean

3. In the SmartGuide, select the *Create a new enterprise bean* radio button.
4. In the *Bean name* field, type the name of the enterprise bean, for example, Customer.
5. In the *Bean type* drop-down list, select *Entity bean with container-managed persistence (CMP) fields*.
6. Because we are creating a new enterprise bean class, select the *Create a new bean class* radio button.
7. Ensure that in the *Project* field is set to ITSO EJB Redbook; otherwise, click the *Browse* button to select the project.

8. In the *Package* field, type the package name, for example, `itso.ejb35.cmp`.
9. In the *Class* field, accept the default value, which is `CustomerBean`.
10. We do not fill in the *Superclass* field, because our bean does not inherit from another enterprise bean.
11. Click on *Next* to display the Bean Class Attributes and Interfaces *page* (Figure 7-5).

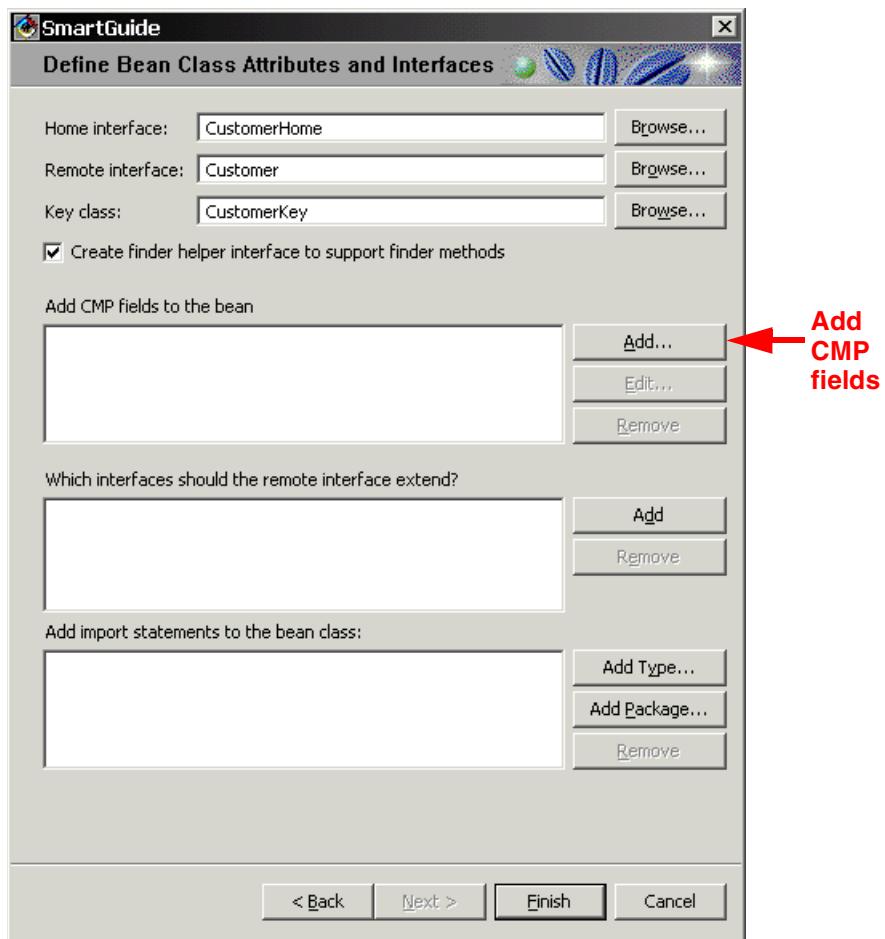


Figure 7-5 CMP bean class attributes and interfaces

12. Accept the names given against the *EJB Home Interface*, *EJB Remote Interface* and *Key class or field*.
13. Select the check box named *Create finder helper interface to support finder methods*. (See Chapter 10. "Custom finder methods" on page 193.)

14. Because we are creating a CMP bean, we can add, edit, or delete the CMP fields in this dialog. Therefore, to add the CMP fields for the CustomerBean, click *Add* next to the *Add CMP fields to the bean* list box. The Create CMP Field SmartGuide appears (Figure 7-6).

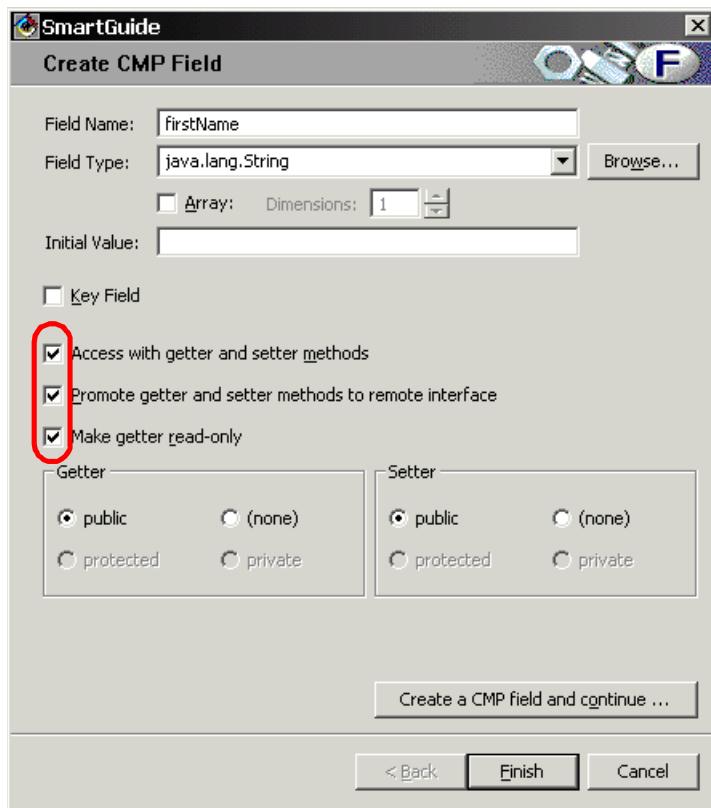


Figure 7-6 CMP bean fields

15. Define each field as described below. For our customer bean refer to Table 7-1 to define all the fields.

- Enter the field name and the field type. You can enter the field type manually, select from the pull-down, or click *Browse* if the field refers to a class.
- Select the *key field* check box, if the field is the data member of the primary key class (CustomerID in our example).
For the key field, you cannot select any of the other options.
- For non-key fields, select *Access with getter and setter methods*, *Promote getter and setter methods to the remote interface* and *Make getter*

read-only check boxes. In most cases you want get and set methods in the remote interface and you want the get method to be read-only.

16. If we have more CMP fields to create, click the *Create a CMP field and continue* button. We can also add any new CMP fields, after creating the bean. We will discuss this later. Similarly, we did not add any methods to the bean except setter and getter methods until now. We will discuss this also in the latter sections.
17. Once finished creating the CMP fields, click on the *Finish* button and we come back to the Define Bean Class Attributes and Interfaces page.
18. In this example, the remote interface is not going to extend any interface. But we have to import some packages for the bean to work correctly. Click on *Add Package* (Figure 7-5 on page 112). The Import statement page appears. For our example, select or type `java.rmi.*`, `javax.ejb.*`, `javax.naming.*`. Click *Close* to come back to the main dialog.
19. Click *Finish* to generate the classes.

Here is the table specifying the CMP fields to be created along with their attributes.

Table 7-1 List of CMP fields for the customer bean

Field name	Type	Key field	Getter/ setter methods	Promote to remote	Getter read-only
customerID	int	Yes	---	---	---
firstName	String	No	Yes	Getter, Setter	Yes
lastName	String	No	Yes	Getter, Setter	Yes
title	String	No	yes	Getter, Setter	Yes
userID	String	No	Yes	Getter, Setter	Yes
password	String	No	Yes	Getter, Setter	Yes
address	Customer Address**	No	Yes	Getter, Setter	Yes

** The CustomerAddress data type has to be developed as a bean. You define typical fields in this bean, such as street, city, state, and zipcode. Declare the class as implementing the `java.io.Serializable` interface, so that it can be stored in the database column. Also implement a constructor that takes four string arguments and initializes the four properties. Use a utility package (`itso.ejb35.util`) that we can use in other EJB groups (Figure 7-7).

```

package itso.ejb35.util;

public class CustomerAddress implements java.io.Serializable {
    private java.lang.String fieldCity = new String();
    private java.lang.String fieldStreet = new String();
    private java.lang.String fieldState = new String();
    private java.lang.String fieldZipcode = new String();
    private final static long serialVersionUID = -4771579797517121057L;

    public CustomerAddress() { super(); }
    public CustomerAddress(String aStreet, String aCity, String aState,
                          String aZipcode) {
        super();
        setStreet(aStreet);
        setCity(aCity);
        setState(aState);
        setZipcode(aZipcode);
    }
    public java.lang.String getCity() { return fieldCity; }
    public java.lang.String getState() { return fieldState; }
    public java.lang.String getStreet() { return fieldStreet; }
    public java.lang.String getZipcode() { return fieldZipcode; }

    public void setCity(java.lang.String city) { fieldCity = city; }
    public void setState(java.lang.String state) { fieldState = state; }
    public void setStreet(java.lang.String street) { fieldStreet = street; }
    public void setZipcode(java.lang.String zipcode) { fieldZipcode = zipcode; }
}

```

Figure 7-7 CustomerAddress class

Adding fields

To add a new CMP field, such as the address (after developing the CustomerAddress class), select the Customer EJB and *Add -> CMP Field* from the context menu. This displays the panel shown in Figure 7-6 on page 113 and you can define the address field.

Tip: To mark a regular field (not a CMP field) as container managed or key field, select the appropriate bean in the Enterprise Beans pane. Then click the *F* icon in the Types pane. This will toggle into the *Properties* pane. Select the field you want to mark and, from the context menu, select the option *Container Managed* or *Key Field* respectively. This can be useful when you add a field to the bean after creating it.

Understanding the generated types

VisualAge for Java generated five types, which we will now discuss:

- ▶ Customer (remote interface)
- ▶ CustomerBean (enterprise bean)
- ▶ CustomerBeanFinderHelper (database queries for custom filters)
- ▶ CustomerHome (home interface)
- ▶ CustomerKey (key class)

Customer interface

The Customer interface is the remote interface for the CustomerBean. The CustomerBean's remote interface provides access to the business methods available in the CustomerBean class. After defining the bean you will find the get and set methods for the CMP fields in the remote interface. In addition, the remote interface inherits methods such as remove and getPrimaryKey.

CustomerBean

The CustomerBean class represents the entity bean. For every field there are generated getter and setter methods. The bean must also implement the ejbCreate methods used to create instances of the Customer bean, and it must implement the business methods used to access and manipulate the data associated with the enterprise bean. In addition to these methods, you will find generated methods such as ejbLoad and ejbStore.

CustomerBeanFinderHelper

The CustomerBeanFinderHelper interface is used by the container to generate the necessary code for querying the database on custom filters. You use the special finder methods to retrieve instances from the database, using other search criteria than the primary key (for example, a search by customer's last name). Initially, there are no fields and methods in this class. Refer to Chapter 10. "Custom finder methods" on page 193 for more information.

CustomerHome

The CustomerHome interface is the home interface for the Customer entity bean. An entity bean home interface defines the methods used by the clients to create new instances of the bean, find and remove existing instances, and obtain metadata information about an instance. After defining the bean you will find the create and findByPrimaryKey methods in CustomerHome. Note that the remove and getEJBMetaData methods are inherited from the javax.ejb.EJBHome superclass.

CustomerKey

The CustomerKey class is used to encapsulate the primary key field(s), in our case the customerID field. The class is public and it must be serializable. It is a unique representation of the Customer, and is used by the findByPrimaryKey method of the CustomerHome interface:

```
findByPrimaryKey(CustomerKey)
```

Understanding the generated methods

VisualAge for Java automatically defines some methods for us. These methods are basically defined in the interfaces we have to implement by default, such as EJBHome and EntityBean. Here we discuss the generated methods for the following types:

- ▶ CustomerBean
- ▶ CustomerHome

CustomerBean

The methods generated into the CustomerBean class are:

public void ejbActivate()

This is a callback method. The container invokes this method when it retrieves an EJB instance and assigns it to a specific EJB object. This method is generally used to load any resource to be used by the bean. For instance, this method can be used for initialization of non-CMP fields. We will not be modifying the implementation of this method for our example.

public void ejbPassivate()

This is also a callback method. The container invokes this method before returning the object to the pool. This method is used for releasing any resources.

public void ejbCreate(int)

This is the default ejbCreate method. We have to initialize the instance variables of the CustomerKey class in this method. It is the container's responsibility to actually create the bean in the permanent storage. Later, we will discuss modifying this method to suit our specific needs.

public void ejbLoad(), public void ejbStore()

If we have only CMP fields in our bean, then we do not have to add any code to the implementation of these callback methods. The container always calls ejbLoad after a bean is loaded from the database, for example, through findByPrimaryKey. You can use ejbLoad to initialize or update the values of any non-CMP fields. The ejbStore method is called before the database is updated with the bean's data, for example, when a transaction is committed.

public void ejbRemove()

This method is invoked by the container, when the client invokes the remove method in the remote object of the bean. The corresponding records are deleted from the database after ejbRemove returns.

CustomerHome interface

The methods generated into the CustomerHome interface are:

public Customer create(int)

A client application uses this method to create an enterprise bean and insert the associated data into the data source. The create method and ejbCreate method must have the same number and type of arguments. But the ejbCreate method returns void, whereas the create method returns a Customer remote reference.

public Customer findByPrimaryKey(CustomerKey primaryKey)

Generally, a finder method is used to find one or more existing entity objects. The findByPrimaryKey method enables a client to locate an EJBObject by the unique primary key.

Adding new methods to enterprise bean

All the business methods for the enterprise bean are created first in the bean class. For this example, we add two methods to the bean:

- ▶ `getCustomerID`, to retrieve the key field with a simple method instead of the more complicated code:
`((CustomerKey)aCustomer.getPrimaryKey()).customerID`
- ▶ `getName`, to retrieve the full name, such as Mr. Firstname Lastname.

To add a new method:

1. In the Types pane of the EJB Page, select the bean class (CustomerBean) and, in the context menu, select *Add -> Method*.
2. In the Create Method SmartGuide, ensure that *Create a new method* radio button is selected. Click *Next*.
3. In the Attributes Page, type the name of the method in the Method Name field, for example, `getName`. Type or browse for the appropriate return type, for example, `String`. Select the appropriate modifier (`public`).
4. Our new method does not have parameters. For methods with parameters, click the *Add* button, and in the Parameters page:
 - Type the name of the parameter. Select *Array* check box if needed and enter the dimension. Type or select the data type of the parameter. Click the *Add* button.

- Repeat this for all the parameters of the method.
5. In the Attributes Page, click *Next*.
 6. Click *Add* to add the exceptions that your method should throw.
 7. Click *Finish*.

For our new methods, complete the body to read:

```
public String getName() {  
    return title.trim() + " " + firstName.trim() + " " + lastName.trim();  
}  
  
public int getCustomerID() {  
    return customerID;  
}
```

Adding methods to the remote interface

After the business methods of an enterprise bean are created in the bean class, they can be added to the remote interface.

The generated get and set methods for the CMP fields of the CustomerBean are already in the remote interface.

For new business methods, select the method in the Members pane and, in the context menu, select *Add To -> EJB Remote Interface*. The method is added to the remote interface and is visible when you select the Customer in the Types pane. Methods that are in the remote interface are marked with the  icon.

If you want to remove a method from the remote interface, select the method and, in the context menu, select *Remove From -> EJB Remote Interface*.

Marking methods read-only

If the new business method does not change the values of any CMP fields in the bean, mark it as *read-only* so that the database is not updated after the method is called.

Select the method and, in the context menu, select *EJB Method Attributes -> Read-only Method*. Read-only method are marked with the  icon.

Customizing the home interface

In this section, we will discuss how to modify the existing methods in the home interface and how to add new ones.

Customizing the ejbCreate method

When creating a new bean by calling the create method of the home interface, the ejbCreate method of the bean is invoked. This method should initialize all the CMP fields of the bean. The generated ejbCreate method only sets the key field. All other fields may get undesired values, because a container reuses storage from other beans when creating a new bean. Therefore, all fields should be set in the ejbCreate method:

```
public void ejbCreate(int argCustomerID) throws javax.ejb.CreateException,
                                              java.rmi.RemoteException {
    _initLinks();
    // All CMP fields should be initialized here.
    customerID = argCustomerID;
    firstName = "";
    lastName = "";
    password = "";
    title = "";
    userID = "";
    address = new itso.ejb35.util.CustomerAddress();
}
```

Alternatively, we can delete the ejbCreate methods and add new tailored methods with parameters that will set all the fields of the bean.

To delete the create method from the home interface, select the ejbCreate method (in the bean) and select *Remove From -> EJB Home Interface*.

Creating a tailored ejbCreate method

Instead of deleting the ejbCreate method, we can change its signature to include parameters for initializing the CMP fields:

1. Select the CustomerBean class in the Types pane and the ejbCreate method in the Members pane.
2. Add parameters for the CMP fields to the method definition and use the parameters to set the CMP fields:

```
public void ejbCreate(int argCustomerID, String aTitle, String aFirstName,
                      String aLastName, String aUserID, String aPassword)
                      throws javax.ejb.CreateException, java.rmi.RemoteException {
    _initLinks();
    // All CMP fields should be initialized here.
    customerID = argCustomerID;
```

```

        title      = aTitle;
        firstName = aFirstName;
        lastName   = aLastName;
        userID     = aUserID;
        password   = aPassword;
        address    = new itso.ejb35.util.CustomerAddress();
    }

```

3. In the context menu, select *Save*. *Save* creates a new method and leaves the original ejbCreate method. *Save Replace* deletes the original ejbCreate method. You can have multiple create methods in the home interface.
4. Select the method in the Members pane and, in the context menu, select *Add To -> EJB Home Interface*. This adds the modified method to the home interface as a new create method with parameters.

Adding methods to the Home interface

The home interface consists of create and find methods. By default we get one create method and one find method (`findByPrimaryKey`) generated. Each create method has a matching ejbCreate method in the bean.

In many cases, we want to add other find methods to the home interface, for example, finding customers by name. This will be discussed in Chapter 10. “Custom finder methods” on page 193.

Database schema and map

In this section we discuss the mapping of entity beans to database tables, as well as how the database schema and map are defined. VisualAge for Java provides a schema browser to define the table layout for an enterprise bean, and a map browser to map CMP fields of a bean to columns of the matching table.

There are three approaches for mapping enterprise beans to tables. We discuss them one by one as an overview; more details on schema and mapping are in Chapter 15. “Advanced mapping for container-manager entity beans” on page 299.

Top-down

This approach takes the existing enterprise beans in the EJB group and generates the database schema and map from the EJB group. The generated schema contains one table for each entity bean and each column in these tables corresponding to a CMP field. The generated mapping maps the fields to the columns. After the schema is created, we can export the schema to the database.

Bottom-up

This approach is used when the database tables already exist. First the schema is imported from the database. Then we can generate enterprise beans, EJB groups, and mappings between them. For each table an enterprise bean is created, and for each table column a CMP field is created in the bean. An example is provided in “Reverse engineering from an existing database” on page 376.

Meet-in-the-middle

In this approach, we can create the enterprise beans and the database schema simultaneously, but independently.

To create the schema, we import the tables from the database using the schema browser. Then we can create a new EJB group map using the map browser. We will use this approach in “Entity model with advanced mapping” on page 315.

Generating the schema and mapping from the EJB group

In our example, we use the top-down approach and follow these steps:

1. In the Enterprise Beans pane of the EJB page, select the EJB Group, for example, `CMP_Entity`.

Generate schema and mapping

2. Select *EJB -> Add -> Schema and Map from EJB Group*. The schema with columns for each CMP field as well as a map of each CMP field to the column is created. Note that you do not automatically see the generated information.
3. Open the schema browser with *EJB -> Open To -> Database Schemas* and the map browser with *EJB -> Open To -> Schema Maps*.
4. Save the schema and mapping. Use a different package than the package used for the EJB code.

In the schema browser, select *Schemas -> Save Schema* and save the `CMP_EntitySchema` in your project into a new package named `itso.ejb35.cmp.schema`.

In the map browser, select *Datastore Maps -> Save Datastore Map* and save the `CMP_EntityCMP_EntityMap` into the same package as the schema.

Edit schema and mapping

5. Study the Customer table in the schema browser (Figure 7-8). Double-click on the table to see the details. Note that the table qualifier is empty by default and the user ID would be used. Set the qualifier to ITSO, so that the actual table name will be ITSO.Customer.

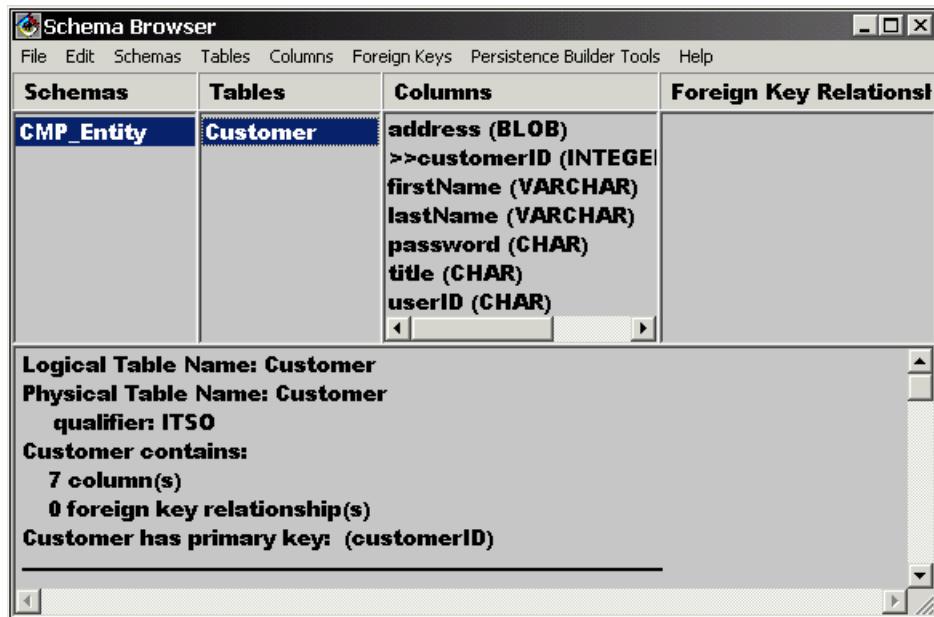


Figure 7-8 Schema browser

6. Study the column definitions. Double-click on a column to see the details. The key column is an INTEGER, the strings are VARCHAR columns of length 30, and the address is a BLOB of 2000 bytes. You can tailor the columns, for example, set the type and length for userID and password to CHAR(8), and change the title to a CHAR(3).
7. Note that the address field is mapped into a BLOB. This only happens if the CustomerAddress class is defined as serializable. The data of such objects is serialized and stored in the BLOB in the table.
The maximum length of the BLOB (default is 2000) can be changed by editing the column. The length should be decided by looking at the maximum data that can be stored for all the fields of an address.
8. Study the mapping in the map browser (Figure 7-9). To see the mapping of the fields to the columns, select the Customer table map and *Table Maps -> Edit Property Maps*. Each class attribute is mapped to a table column.

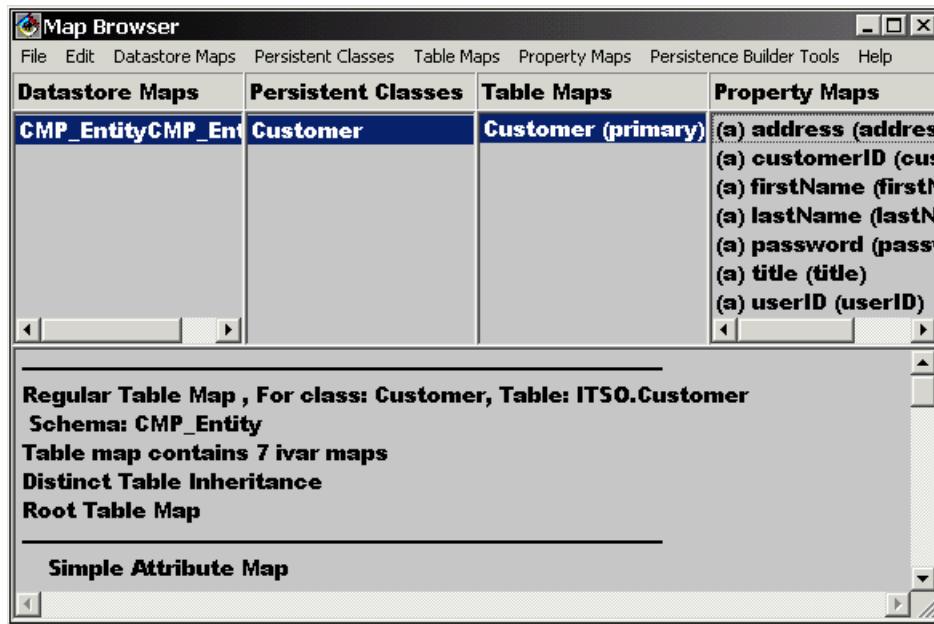


Figure 7-9 Map browser

Create the database table

9. In the schema browser select the schema and *Schemas -> Import / Export Schema -> Export Entire Schema to Database* (or select a single table and *Tables -> Export Tables -> Export Selected Tables with Keys to Database*).
10. In the Database Connection Info dialog (Figure 7-10), select a driver (connection type) and a target database (data source) where the Customer table is created. Generally *Userid* and *Password* are not required. Check with your database access rights.

Note that this is not the driver used at execution time for the enterprise beans. This is the driver used to run the CREATE TABLE statement.

Attention: The JDBC driver classes must be available in the class path, usually set in the *Windows -> Options* dialog. See Chapter 5. "EJB development environment" on page 65 for more information.

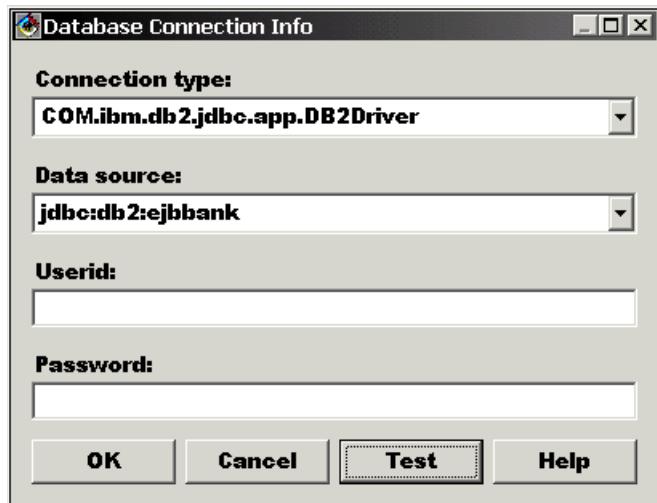


Figure 7-10 Database connection information

11. Click *OK*. The creation of the table can be monitored in the Console window of VisualAge for Java.

```
CREATE TABLE ITSO.Customer (
    CustomerID INTEGER NOT NULL,
    firstName VARCHAR(30),
    lastName VARCHAR(30),
    password CHAR(8),
    title CHAR(3),
    userID CHAR(8),
    address BLOB(2000))
```

12. Save the schema and map if you made any changes. If you select the schema or map itself, the bottom pane will say if the schema or map is saved or is *dirty* and should be saved.

13. Close the schema and map browsers; they are not needed any more.

At this point the enterprise bean is fully defined and mapped to a table in a database. The table itself has been created, but is empty.

You can add some rows manually using DB2 commands, for example:

```
D:\SQLLIB\BIN>db2 insert into itso.customer
              values(1, 'John', 'Smith', 'jspw', 'Mr', 'johnny', null)
D:\SQLLIB\BIN>db2 insert into itso.customer
              (customerid, title, firstname, lastname, userid, password)
              values(2,'Mrs','Jennifer','Mueller','jenny', 'jmpw')
```

Attention: By exporting the customer table to the EJBBANK database you deleted the original customer table for testing. After finishing the testing, recreate the original customer table as described in “Creating the database and tables” on page 62. If you followed the mapping instructions you should be able to run against the original table as well.

Deployment

In this section, we will discuss adjusting the deployment descriptor attributes and generating deployment code

Adjusting deployment attributes

To change any properties of the deployment descriptor, select the enterprise bean and *Properties* (Figure 7-11).

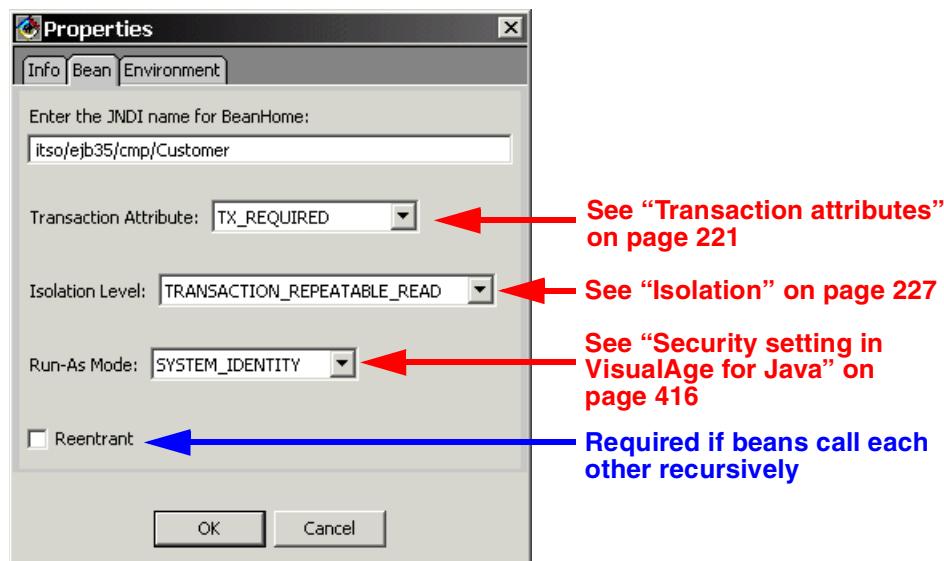


Figure 7-11 CMP enterprise bean properties

Here you can change the value of the properties *JNDI Name*, *Transaction Attribute*, *Isolation Level*, *Run-As Mode*, and *Reentrant*.

For now, we leave the defaults. Transaction attributes and isolation levels are discussed in Chapter 12. “Transaction management” on page 217, and run-as is described in Chapter 20. “Security for enterprise beans” on page 411.

Similarly, if you want to add a control descriptor to an EJB method, select the method of the bean in the Members pane, and select *Members -> EJB Method Attributes -> Add Control Descriptor* to open the Add Control Descriptor dialog (Figure 7-12).

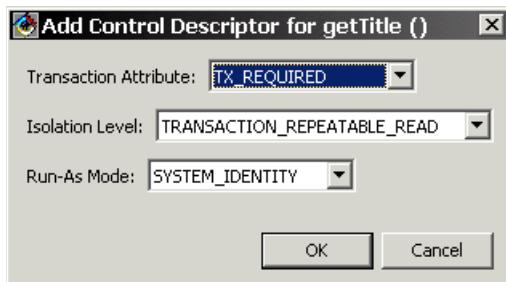


Figure 7-12 Bean method control descriptor

Accept the default values for the descriptors or change them as necessary, then click *OK*. The values set in the method's control descriptor override the values set for the bean.

Generating deployment code

We have to generate the deployed code, before testing or deploying the beans. This helps in analyzing the code for its compatibility with EJB specifications as well as the rules specific to the EJB server.

Select an EJB group (*CMP_Entity*) or an enterprise bean in the Enterprise Beans pane of the EJB page, then select *EJB -> Generate Deployed Code*.

As a result, VisualAge for Java generates all the stubs, ties, and helper classes that are required by the RMI-IIOP communication layer. The generated classes are listed in the Types pane. Click on the rectangular symbol on the top-right of the Types pane to switch between viewing all the classes (including generated classes) or only the original definitions.

Generated classes

The deployed code consists of a number of classes for communication between client and server using RMI-IIOP, and for database access using JDBC. We do not explain this code in detail; we just provide a short description:

- ▶ `_Customer_Stub`, `_Customer_BasteStub`, `EJSRemoteCustomer`, and `_EJSRemoteCustomer_Tie` for communication of customer instance
- ▶ `_CustomerHome_Stub`, `_CustomerHome_BasteStub`, `EJSRemoteCustomerHome`, and `_EJSRemoteCustomerHome_Tie` for communication of the customer home

- ▶ EJSCustomerHomeBean implements the customer home
- ▶ EJSFinderCustomerBean is the interface of customer finder methods
- ▶ EJSJDBCPersisterCustomerBean contains the SQL statements and methods to execute them

Notes on regenerating deployed code

We have to regenerate the deployed code for an enterprise bean in the following situations:

- ▶ A method in the home or remote interface (or base interface) is added, removed, or changed.
- ▶ The method signature is changed for a method that exists in the home or remote interface.
- ▶ A CMP field is added, deleted, or changed.
- ▶ A key field designation is changed.
- ▶ The mapping or schema is changed.
- ▶ An access bean (copy helper or rowset) is added or deleted.
- ▶ A child enterprise bean is added or deleted that exists in an EJB inheritance relationship.
- ▶ An association is added, changed, or deleted.

Test environment

Before deploying the bean, we can test the bean using the EJB test client. In this section, we will discuss testing our created beans, using the VisualAge WebSphere Test Environment.

Creating an EJB server configuration

Once we generate the deployed classes, we can create an EJB server configuration that consists of one or more EJB groups containing enterprise beans.

Select the EJB Group (CMP_Entity) and EJB -> Add to -> Server Configuration. The Server Configuration browser appears (Figure 7-13). See “EJB server configuration” on page 91 for more details.

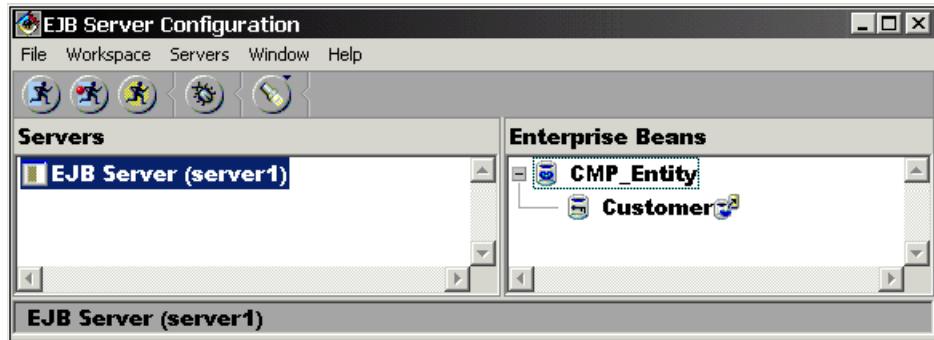


Figure 7-13 EJB server configuration for CMP_Entity group

Setting EJB server properties

Generally, for just testing the bean, we do not have to change the default settings of the server configuration. However, since we are testing the entity bean, we can ensure that the database data source or URL is correctly set.

Select the EJB server and *Servers -> Properties*. In the Properties for EJB Server (Server1) dialog set the correct data source (EJBBANK) and other fields as appropriate. See “Setting the EJB server properties” on page 92 for more information.

Starting the servers

Let's repeat how to start the testing environment:

1. From the Workspace menu, select *Tools > WebSphere Test Environment*. The WebSphere Test Environment Control Center appears.
2. Select the persistent name server and start it. The Console window appears in the background and enables you to monitor the server. Wait until you see the message *Persistent Name Server is started* in the status area of the WebSphere Test Environment Control Center.
3. Start the EJB server in the EJB Server Configuration window. Use the black running man icon or *Servers -> Start Server*. The Console window now also monitors the EJB server. Wait until the EJB server is ready (Server open for business).
4. You can change the source code of any enterprise bean that you are testing and debug it without stopping the EJB server. However, if you change the home or remote interfaces, you must:

- Stop the EJB server (VisualAge for Java may have stopped it already)
- Regenerate the deployed code
- Start the EJB server again

Using the EJB test client

The EJB development environment provides a test client that can be used to test the home and remote interface methods of the enterprise beans. In this section, we explore the basic features of the test client. See “EJB test client” on page 94 for more information.

To run the test client:

1. Start the EJB test client from the EJB Server Configuration window (running man icon with yellow background), or from the Workbench EJB pane (select the enterprise bean group and *EJB -> Run Test Client*). The EJB Test Client window opens.
2. Select the correct JNDI name (`itso/ejb35/cmp/Customer`) and click *Lookup*. When the connection is established to the name server and an instance of the home interface is retrieved from the name server, the Home interface page for the enterprise bean appears (Figure 7-14).

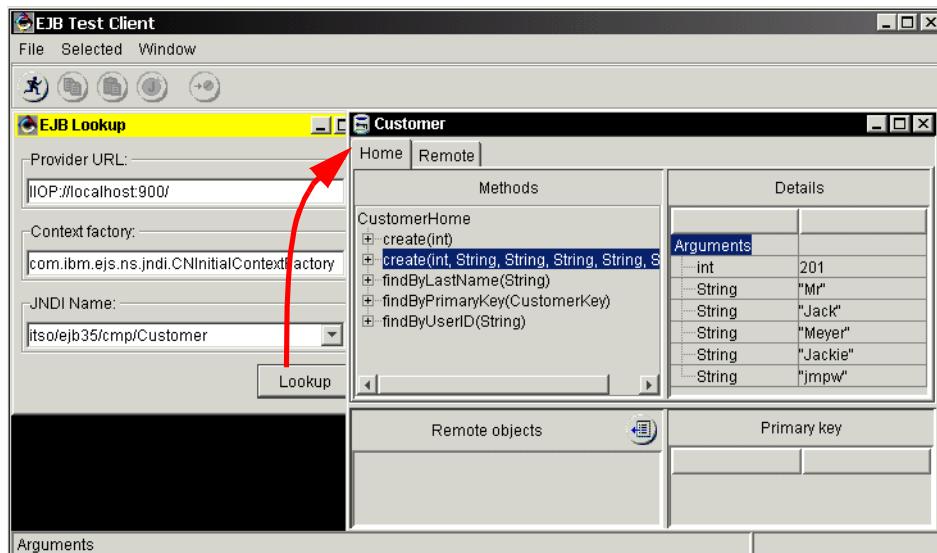


Figure 7-14 EJB test client: creating a customer

Create an instance in the database

3. In the *Methods* pane, select the *create(....)* method, and in the *Details* pane set the arguments, for example, 201, Mr, Jack, Meyer, Jackie, jmpw.
4. Click the *Invoke* button. In our example, the *create* method call causes the EJB server to construct a new enterprise bean and row in the table with the constructor we provided. The resulting remote interface object is used by the test client for testing remote methods.

Retrieving an instance from the database

5. In the *Home* tab, select the *findByPrimaryKey* method. Enter an existing key value in the *Details* pane (103) and click *Invoke* (Figure 7-15, left side).
6. A bean instance is created from the table row and the *Remote interface* is displayed.

Using the remote interface

7. In the *Remote* tab, in the *Methods* pane, select a getter method to retrieve the bean attributes (Figure 7-15, right side).

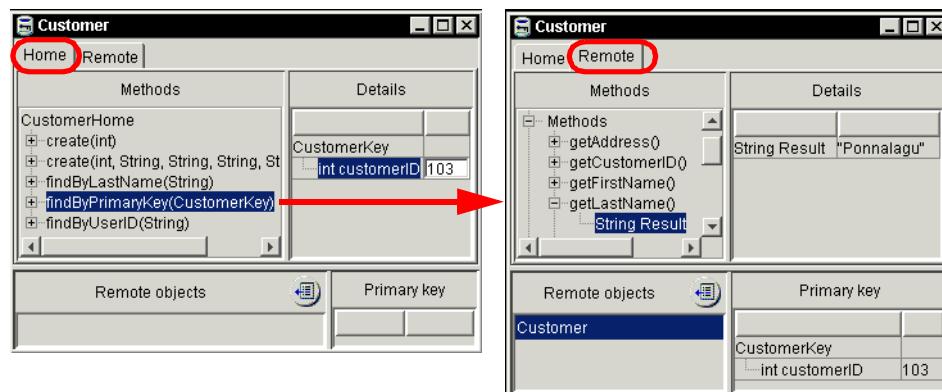


Figure 7-15 Retrieving a customer and invoking methods

8. To update the data, select a setter method, then enter the required parameters using the same approach we used on the Home interface page.
9. Click the *Invoke* button to send the method to the enterprise bean. The result of the method call is displayed as a child node of the method in the *Methods* pane.
10. Close the EJB test client window, once the methods are tested.

Defining the bank model

In the later chapters, we are discussing topics such as session beans, inheritance, and associations. We will be using a typical bank application as our working example (Figure 7-16).

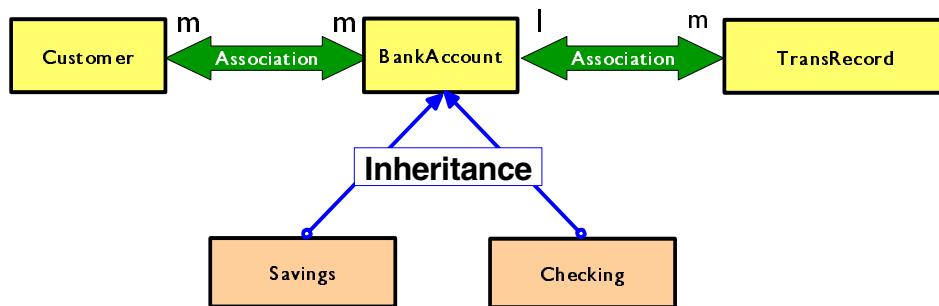


Figure 7-16 Bank application entities

To run the session bean example and to understand associations and inheritance, we create two more CMP beans called *BankAccount* and *TransactionRecord*. The details for creating these two beans are specified in the following sections.

At this point we will not implement the inheritance structure (see Chapter 16. “Inheritance” on page 329) and the associations (see Chapter 17. “Associations” on page 353).

Bank account

The **BankAccount** bean is a generic banking bean. We will use this bean for experimenting with both inheritance and association. Create the bean with finder helper class and primary key class, with their default names.

CMP Fields

Table 7-2 shows the CMP fields of the *BankAccount* bean.

Table 7-2 *BankAccount* CMP fields

Field name	Type	Key field	Getter/setter methods	Promote to remote	Getter read-only
accID	String	Yes	---	---	---
balance	BigDecimal	No	Yes	Getter, Setter	Yes

Mapping

Map the BankAccount bean to the ACCOUNT table in the EJBBANK database. The columns are accid—CHAR(8), and balance—DECIMAL(8,2).

Exception

Create an exception class called InsufficientFundException, which is thrown by the withdraw method, when a customer tries to withdraw an amount greater than the account balance. Create the class in the itso.ejb35.util package, as a subclass of Exception:

```
package itso.ejb35.util;
public class InsufficientFundException extends Exception {
    public InsufficientFundException() {
        super();
    }
    public InsufficientFundException(String s) {
        super(s);
    }
}
```

Tailored methods

The BankAccount bean has a tailored create method:

```
public void ejbCreate(java.lang.String argAccID,
                      java.math.BigDecimal argBalance)
    throws javax.ejb.CreateException, java.rmi.RemoteException {
    _initLinks();
    // All CMP fields should be initialized here.
    accID = argAccID;
    balance = argBalance;
}
```

Replace the generated ejbCreate method with this tailored method and promote it to the home interface.

We will also implement two business methods, deposit and withdraw. These methods will change the balance and create a transaction record, therefore we have to define the enterprise bean for the transaction record first.

Transaction record

The **TransRecord** bean stores a record of each banking transaction (deposit, withdraw) executed on an account. This bean will be used in association with the BankAccount bean. Create the bean with finder helper class and primary key class, with their default names.

CMP Fields

Table 7-3 shows the CMP fields of the TransRecord bean.

Table 7-3 TransRecord CMP fields

Field name	Type	Key field	Getter/setter methods	Promote to remote	Getter read-only
transID	TimeStamp	Yes	---	---	---
accID	String	No	Yes	Getter, Setter	Yes
transamt	BigDecimal	No	Yes	Getter, Setter	Yes
transtype	String	No	Yes	Getter, Setter	Yes

Mapping

Map the TransRecord bean to the TRANSRECORD table in the EJBBANK database. The columns are transid—TIMESTAMP, accid—CHAR(8), transamt—DECIMAL(8,2), and transtype—CHAR(1).

Methods

The TransRecord bean has this tailored create method:

```
public void ejbCreate(String anAccID, java.math.BigDecimal anAmount,
                      String aTranstype)
                      throws javax.ejb.CreateException, java.rmi.RemoteException {
    _initLinks();
    // All CMP fields should be initialized here.
    transID = new java.sql.Timestamp(System.currentTimeMillis());
    accID = anAccID;
    transtype = aTranstype;
    transamt = anAmount;
}
```

Replace the generated ejbCreate method with this tailored method and promote it to the home interface.

Business methods

Now we can implement the two business methods in the BankAccount bean, deposit and withdraw. Both methods must be promoted to the remote interface:

```
public void withdraw(BigDecimal amount)
                     throws itso.ejb35.util.InsufficientFundException;

public void deposit(BigDecimal amount);
```

Deposit method

The deposit method adds the amount to the balance and creates a transaction record. An instance variable (`txRecHomeCMP`) of the class holds the home of the `TransRecord` bean.

```
public void deposit(java.math.BigDecimal amount) {
    balance = balance.add(amount);
    try {
        setTxRecHome();
        txRecHomeCMP.create(accID, amount, "D");
    }
    catch (Exception e) {
        System.out.println(" ==> transaction record failed");
        e.printStackTrace();
    }
}
```

Withdraw method

The withdraw method is similar, but in addition it throws an exception if the balance is too small for the withdraw amount.

```
public void withdraw(java.math.BigDecimal amount)
    throws itso.ejb35.util.InsufficientFundException {
    if (balance.compareTo(amount) == -1) throw
        new itso.ejb35.util.InsufficientFundException("Not enough funds");
    else
        balance = balance.subtract(amount);
    try {
        setTxRecHome();
        txRecHomeCMP.create(accID, amount, "C");
    }
    catch (Exception e) {
        System.out.println(" ==> transaction record failed");
        e.printStackTrace();
    }
}
```

Utility method

The `setTxRecHome` method retrieves the home of the `TransRecord` bean.

```
private itso.ejb35.cmp.TransRecordHome txRecHomeCMP;
protected boolean txRecHome = false; // TxRecHome not set yet

private void setTxRecHome() {
    if (!txRecHome) {
        try {
            //Properties prop = new Properties();
            //prop.put( Context.PROVIDER_URL,"iiop://" );

```

```

//prop.put( Context.INITIAL_CONTEXT_FACTORY,
//           "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
//InitialContext ctx = new InitialContext(prop);
InitialContext ctx = new InitialContext();
txRecHomeCMP = (itso.ejb35.cmp.TransRecordHome)
                javax.rmi.PortableRemoteObject.narrow(
                    ctx.lookup("itso/ejb35/cmp/TransRecord"),
                    itso.ejb35.cmp.TransRecordHome.class );
txRecHome = true;
} catch ( NamingException exc ) {
    System.out.println( "Error retrieving the home" );
    exc.printStackTrace();
}
}

```

Mixing CMP and BMP entity beans

It is possible to use the BankAccount CMP bean together with the TransactionRecord BMP bean, which we develop in Chapter 8. “Bean-managed persistence entity beans” on page 139.

To make this work, change the txRecHomeCMP variable to point to the BMP bean (itso.ejb35.bmp.TransactionRecordHome) and change the setTxRecHome method to retrieve the correct home interface.

Important: When mixing CMP and BMP, it is mandatory to use the JTA driver when defining the data source (see “Data sources” on page 89), because the CMP and the BMP do not share the database connection.

Test the BankAccount and TransRecord beans

Add the EJB group to a server configuration, check the properties of the EJB server, and start the persistent name server and the EJB server according to the instructions in “Testing enterprise Java beans” on page 91.

Use the EJB test client to test the new entity beans. See “Constructing and inspecting objects” on page 101 for instructions on how to handle BigDecimal arguments.

Summary

In this chapter we described how CMP beans can be created and tested. The key points include:

- ▶ An entity bean represents an object view of the data stored in an underlying database.
- ▶ The persistence is based on the information specified in the deployment descriptor. VisualAge for Java can generate the deployed code and deployment descriptor.
- ▶ When you use container-managed persistence, the container invokes a persistence manager class that provides the persistence management business logic.

The EJB development environment provides a test client that can be used to test the home and remote interface methods of the enterprise beans.

Pros and cons

Here are some points that could help you make a decision.

Advantages

The container will automatically synchronize the persistent fields with the database as dictated by the deployer. No code for persistence needs to be written by the developer.

One of the limits of BMP is caching of the object state between method invocations, which has to be implemented by the bean developer. In BMP, the instance must handle the ejbLoad and ejbStore for caching the bean state. When the container invokes the ejbStore method, the instance must push the cached update to the database. When the container invokes the ejbLoad method, the instance must discard the cached state and reload the state from the database. But with CMP, caching of data is handled by the container.

Disadvantages

The main performance problem with CMP is that it can generate a lot of SQL requests to the database. With container-managed persistence, we cannot optimize database queries. (Some optimization is possible through custom finder methods as described in Chapter 10. “Custom finder methods” on page 193.)



Bean-managed persistence entity beans

This chapter discusses how to write bean-managed persistence (BMP) entity beans. We describe in detail how to build a bean-managed entity bean, what methods have to be implemented, and how the SQL statements must be coded.

We will also discuss advantages and disadvantages of BMP beans versus CMP beans.

Bean-managed persistence

Bean-managed persistence or BMP enables you to manage the persistence of your bean instead of delegating it to the container. In order to achieve this, you have to write a number of methods that are usually handled by the CMP layer of your container.

This bean has to implement the same interfaces as the other beans:

- ▶ The bean class has to implement the `javax.ejb.EntityBean` interface. All callback methods in this interface will have to be implemented.
- ▶ The remote interface has to extend the `javax.ejb.EJBObject`. In this interface, you will mainly find business methods as well as accessor methods.
- ▶ The home interface has to extend the `javax.ejb.EJBHome`. This interface may contain create and finder methods.

Why BMP?

So why would you need BMP? Well there are some reasons why you cannot rely on CMP to do the work:

- ▶ The database schema is impossible to map to the EJBs.
- ▶ You have a relational database for entity beans that is not currently supported for CMP or does not have JDBC 2.0 level drivers and therefore you will have to use SQLJ, stored procedures, or some other tooling to access it.
- ▶ You require non-relational persistence method (MQ, CICS, IMS, flat files...).
- ▶ You do not believe in CMP and/or do not trust the quality of generated code.

Creating a BMP entity bean with VisualAge for Java

In this chapter, we will develop a `TransactionRecord` bean to illustrate the use of BMP.

Note: This is the same `TransRecord` bean we developed as a container_managed entity bean, however, we are using a different name so that we can deploy it into the same WebSphere EJB container.

The `TransactionRecord` bean represents a transaction between a customer and his/her bank account. It stores the following information about a particular transaction: `transID` (a unique transaction ID), `accID` (account ID of the customer), `amount` (amount of transaction), and `transType` (type of transaction).

Creating the TransactionRecord entity bean

To create the TransactionRecord bean, perform these steps:

1. In the EJBs pane, select or create an EJB group to contain your EJB bean; we use the **BMP_Entity** group.
2. From the context menu, choose *Add -> Enterprise bean*; this launches the Create Enterprise Bean SmartGuide (Figure 8-1).

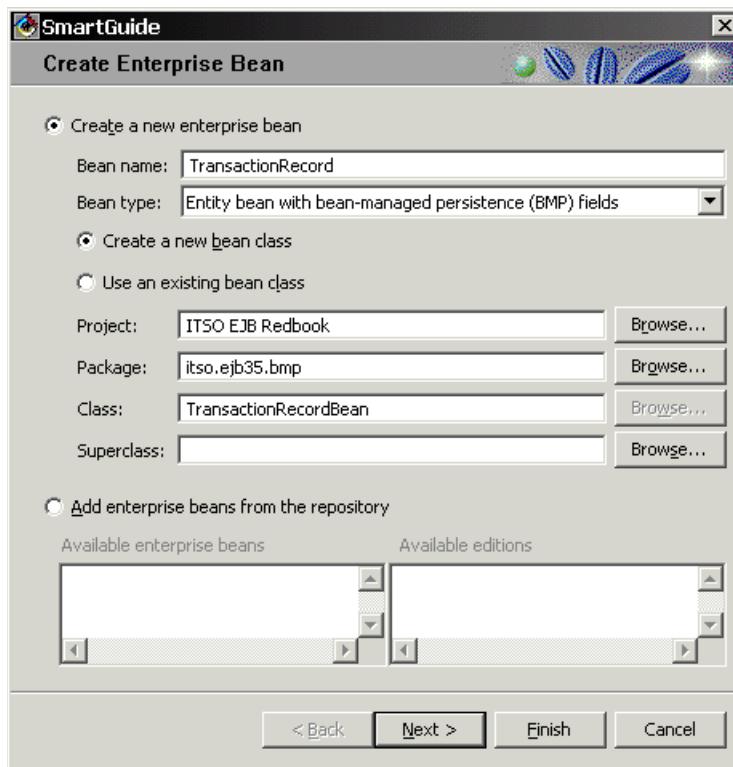


Figure 8-1 Creating a BMP entity bean

3. In the *Bean name* field, type in the name of the new EJB bean, *TransactionRecord*.
4. From the *Bean type* drop-down list, select the *Entity bean with Bean-managed persistence fields (BMP)* item to indicate that the bean is a BMP entity bean.
5. The project should be filled in, otherwise find it using *Browse*. Enter a package name, such as *itso.ejb35.bmp*.

- It is a good idea at this point to click *Next* to define class attributes and interfaces (Figure 8-2). Add some import statements using the *Add Package* button. Select javax.ejb, javax.naming, java.rmi, javax.sql, and java.sql from the list. These packages will be added in the bean class as import statements. It means that the types from these packages will not be fully qualified, making the reading of the code easier. Also add the type java.math.BigDecimal, a class we use for the amount.
- The SmartGuide proposes default names for the EJB home interface, remote interface, and key class. Click on *Finish* to create the bean.

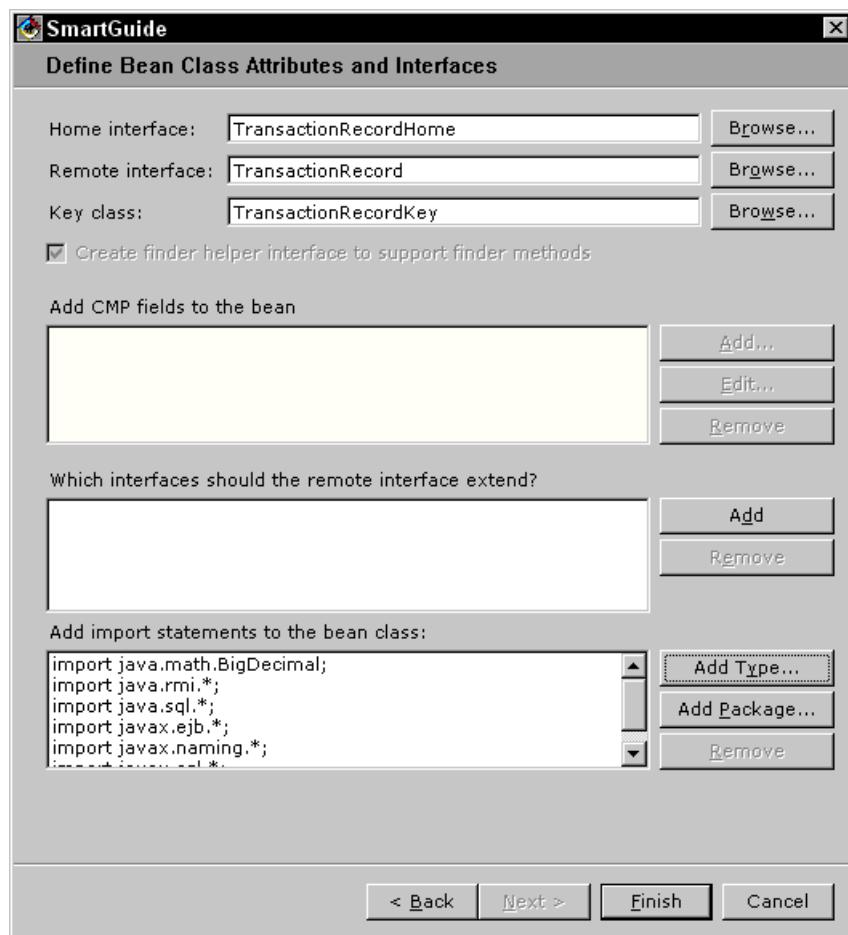


Figure 8-2 BMP class attributes and interfaces

A quick tour of the generated classes

The SmartGuide has generated the following classes for us:

TransactionRecord	The remote interface of the bean used to remotely access the methods on the bean.
TransactionRecordBean	The bean itself which will contain the actual implementation
TransactionRecordHome	The home interface of the bean used to retrieve and create beans.
TransactionRecordKey	The primary key class which will hold all the attributes used to compose the primary key of the bean. In our case, <code>transID</code> will be the key field.

Adding and defining BMP fields

When you create a bean in the EJB development environment, two fields (`entityContext` and `serialVersionUID`) are created automatically in the bean class. This topic explains how to add the bean-specific fields to the `TransactionRecord` bean, namely:

- ▶ `transID` (`java.sql.Timestamp`) - the key field
- ▶ `transtype` (`String`)
- ▶ `transamt` (`java.math.BigDecimal`)
- ▶ `accID` (`String`)

These fields are declared as `private` and are accessible via public getter and setter methods, except for `transID`, where the setter method is `private`.

Tip: In the real world all setter methods would be marked as `private` because a transaction record should not be modified after creation, but for testing our bean it is more practical to have setter methods as well.

To add a field to the `TransactionRecord` bean, you can either add them directly to the class definition and generate accessors afterwards, or:

1. In the *Types* pane of the EJB page, select the `TransactionRecord` bean class.
2. Click on the *Add Field* button to launch the Create Field SmartGuide.
3. Fill in the field name, type name and check the *Access with getter and setter methods* checkbox. Select the *public* radio button for Getter and Setter (*private* for the Setter of `transID`).
4. Click *Finish* to add the field and the accessor methods to the `TransactionRecordBean` class.

Changing the key class

The primaryKey field in the key class, TransactionRecordKey, is by default a string. In our case, however, the key field is a java.sql.Timestamp.

We have to change the primaryKey from String to java.sql.Timestamp:

```
import java.sql.Timestamp;

public class TransactionRecordKey implements java.io.Serializable {
    public Timestamp primaryKey;
    final static long serialVersionUID = .....L;
    .....
    public TransactionRecordKey(Timestamp key) {
        primaryKey = key;
    }
}
```

Delete the original TransactionRecord(String) constructor.

A first look at the bean methods

After the bean has been created, we can examine the methods of the bean. There are two types of methods, the required callback methods for which we have to provide an implementation, and some utility methods that help us to reuse some code and also reduce the complexity of the required methods.

Callback methods

A number of callback methods have to be implemented in order to support the bean-managed persistence.

Note: All these methods have been generated by VisualAge. Most of them are just a skeleton while others have a usable implementation.

ejbLoad	This method has to ensure that the bean data is consistent with the data in the persistent store, whatever it is.
ejbStore	The container invokes this method to allow the bean to update the persistent store with its own data. It is up to the bean to determine if such update is relevant or not.
ejbFindByPrimaryKey	This method has to be implemented to retrieve the bean using the primary key. A FinderException or ObjectNotFoundException can be thrown.
ejbCreate	This method has to be updated with the correct number of parameters and it has to be implemented. It is used to create a new bean in the persistent store.

	Please note that the throws clause includes CreateException. This exception should be thrown if something goes wrong during creation.
ejbRemove	When a client is calling the remove method on a remote interface or is calling the remove(key) method on the home interface, it results in the container invoking the ejbRemove method to delete the matching data in the persistent store and to release held resources.

In addition, there are some generated methods that we do not have to modify:

setEntityContext	This method is called when a bean instance is created in the pool but is not yet associated with a EJBObject. The data received is stored inside an instance variable.
unsetEntityContext)	When this method is called, the bean is about to be removed from the pool and destroyed.
ejbActivate	This method is invoked when a client calls a method on a EJBObject with no associated bean instance. The container then takes an instance from the pool and invokes ejbActivate to allow the instance to acquire whatever resources it needs in order to be in the ready state.
ejbPassivate	When a particular bean instance is not used anymore, the container is passivating it. The result of that operation is a callback to ejbPassivate to allow the bean to release the resources acquired during the activation.

A look at some utility methods

To simplify the implementation of the generated methods, we define some utility methods:

getDatasource	“Lazy initialize” a DataSource class variable, named ds, which allows you to cache the data source between invocations.
getConnection	Open a connection using a user ID and password provided in the bean environment variables.
cleanup	Closes the passed prepared statement and connection.

Implementing the bean methods

Let's first write the utility methods and understand some useful abstractions that we have placed in them. Then we can have a look at the callback methods that we have to implement for the BMP.

Utility methods

To implement the utility methods we need some additional definitions. Change the TransactionRecordBean class to include this statement:

```
private static DataSource ds;
```

Let's begin with the `getDatasource` method (Figure 8-3).

This method is a classic example of *lazy initialization*, a technique which consists of initializing a variable only when it is needed.

You also notice that some data are retrieved from environment variables using `getEntityContext.getEnvironment`. What does it mean? Simply that we have stored environment variables in the deployment descriptor of the bean. The reason behind this is to avoid from recompiling the code if the bean is deployed with a different data source, table name, or using different user ID and password.

```
private DataSource getDatasource() throws SQLException {
    if (ds == null) {
        Properties prp = getEntityContext().getEnvironment();
        String dsName = prp.getProperty("DATASOURCE_NAME");
        String providerURL = prp.getProperty("PROVIDER_URL");
        InitialContext ctx = null;
        Properties prop = new Properties();
        try {
            prop.put(Context.PROVIDER_URL, providerURL);
            prop.put(Context.INITIAL_CONTEXT_FACTORY,
                      "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
            ctx = new InitialContext(prop);
            ds = (DataSource)ctx.lookup(dsName);
        } catch (NamingException exc) {
            System.out.println("error retrieving datasource ");
            exc.printStackTrace();
        }
    }
    return ds;
}
```

Figure 8-3 TransactionRecordBean `getDatasource` method

If you select the bean and the *Properties* context menu (Figure 8-4), you can edit the environment variables in the *Environment* tab.

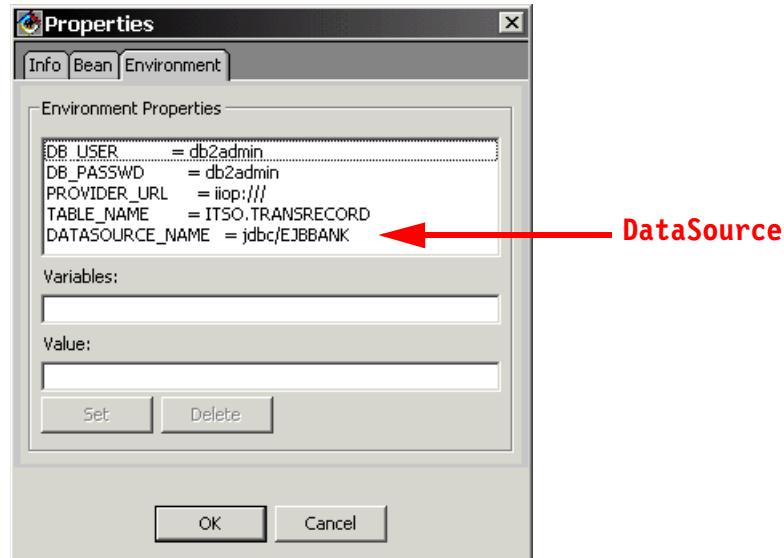


Figure 8-4 TransactionRecord deployment descriptor

The **getConnection** method is shown in Figure 8-5. Note that a user ID and password may be required to create a connection.

```
private Connection getConnection() throws SQLException {
    Properties prp = getEntityContext().getEnvironment();
    String dbUser = prp.getProperty("DB_USER");
    String dbPasswd = prp.getProperty("DB_PASSWD");

    return getDatasource().getConnection( dbUser, dbPasswd );
}
```

Figure 8-5 TransactionRecordBean getConnection method

The **cleanup** method is shown in Figure 8-6. The method closes the prepared SQL statement and the connection that are passed as arguments.

```
private void cleanup(PreparedStatement pstmt, Connection conn) {
    try {
        pstmt.close();
    } catch (SQLException e) {}
    try {
        conn.close();
    } catch (SQLException e) {}
}
```

Figure 8-6 TransactionRecordBean cleanup method

Callback methods

Now we can look at the EJB methods and see how we can implement the persistence using JDBC. We use the TransRecord table in the EJBBank database (see “Bank database” on page 55).

Note that you can use the schema browser to define the table; however, only an empty schema and no mapping will be generated for you when you select the group and *Add -> Schema and Map from EJB Group*. However, you can define the table layout yourself and then export the schema to the database. There is no need for a mapping.

ejbLoad method

We first look at the ejbLoad method (Figure 8-7). The method has first to acquire a connection to the database. Once that connection is available, it is used to create a prepared SELECT statement with the query needed to retrieve the correct bean data. The primary key is retrieved from the entity context and is injected in the query before execution.

When the query is done, the data is extracted from the result set and stored in the instance variables.

Notice that a try/catch/finally structure was used and that the finally block is used to close the prepared statement and connection (actually this is done in the cleanup method).

```

public void ejbLoad() throws java.rmi.RemoteException {
    TransactionRecordKey key =
        (TransactionRecordKey) getEntityContext().getPrimaryKey();
    transID = key.primaryKey;
    PreparedStatement pstmt = null;
    ResultSet rs = null;
    Connection conn = null;
    Properties prp = getEntityContext().getEnvironment();
    String tableName = prp.getProperty( "TABLE_NAME" );

    try {
        conn = getConnection();
        pstmt = conn.prepareStatement( "select accid, transtype, transamt" +
            " from " + tableName + " where transid = ?" );
        pstmt.setTimestamp( 1, transID );

        // execute the query and read the values
        rs = pstmt.executeQuery();
        rs.next();
        accID = rs.getString( "accid" );
        transamt = rs.getBigDecimal( "transamt" );
        transtype = rs.getString( "transtype" );
    } catch (SQLException exc) {
        System.out.println("Exception while loading TransactionRecord #"
            + transID );
    } finally {
        cleanup( pstmt, conn );
    }
}

```

Figure 8-7 TransactionRecord ejbLoad method

ejbStore method

The ejbStore method (Figure 8-8) is very similar to the ejbLoad in the sense that instead of reading the data, it writes it to the database.

The database is updated using a prepared UPDATE statement. The field values are injected into the statement before execution.

```

public void ejbStore() throws RemoteException {
    //TransactionRecordKey key = (TransactionRecordKey)
    //                                         getEntityContext().getPrimaryKey();
    PreparedStatement pstmt = null;
    ResultSet rs = null;
    Connection conn = null;
    Properties prp = getEntityContext().getEnvironment();
    String tableName = prp.getProperty( "TABLE_NAME" );
    try {
        conn = getConnection();
        pstmt = conn.prepareStatement( "update " + tableName +
            " set accid = ?, transtype = ?, transamt = ? where transid = ?" );
        pstmt.setString( 1, accID );
        pstmt.setString( 2, transtype );
        pstmt.setBigDecimal( 3, transamt );
        pstmt.setTimestamp( 4, transID );
        // execute the update statement
        pstmt.executeUpdate();
    } catch (SQLException exc) {
        System.out.println( "Exception while storing TransactionRecord #"
            + transID );
    } finally {
        cleanup( pstmt, conn );
    }
}

```

Figure 8-8 TransactionRecord ejbStore method

Note: If you have been using application servers other than WebSphere, you have probably noticed that the ejbStore method does not check if the bean is dirty (values have been changed) or not. The reason is that VisualAge for Java and JetAce allow you to mark a method as read-only in the deployment descriptor. The WebSphere container will not call ejbStore when a read-only method is invoked.

ejbCreate method

Now we examine the ejbCreate method, which is used to create new beans (Figure 8-9). The first modification is to change the number of parameters. We now pass the account ID (String), the amount (BigDecimal), and the type (String). The key is created using the actual time.

The actual creation of the record in the database is done using a prepared INSERT statement with all the values injected before execution.

```

public TransactionRecordKey ejbCreate( String acctID, BigDecimal amount,
                                      String aTranstype ) throws CreateException, RemoteException {
    java.util.Date date = new java.util.Date();
    TransactionRecordKey key = new TransactionRecordKey( new
                                                          Timestamp( System.currentTimeMillis() ) );
    PreparedStatement pstmt = null;
    Connection conn = null;
    Properties prp = getEntityContext().getEnvironment();
    String tableName = prp.getProperty( "TABLE_NAME" );

    if (acctID == null)
        throw new CreateException("accountID is null");
    else if (amount == null)
        throw new CreateException("amount is null");

    // initialize fields
    accID      = acctID;
    transamt   = amount;
    transtype  = aTranstype;

    try {
        conn = getConnection();
        pstmt = conn.prepareStatement( "insert into " + tableName +
                                      " (transid, accid, transtype, transamt) " +
                                      " values (?, ?, ?, ?)" );
        pstmt.setTimestamp( 1, key.primaryKey );
        pstmt.setString( 2, acctID );
        pstmt.setString( 3, aTranstype );
        pstmt.setBigDecimal(4, amount );
        // execute the insert statement
        pstmt.executeUpdate();
        return key;
    } catch (SQLException exc) {
        System.out.println( "Exception while creating TransactionRecord #"
                           + key.primaryKey );
    } finally {
        cleanup( pstmt, conn );
    }
    return key;
}

```

Figure 8-9 TransactionRecord ejbCreate method

Use *Save Replace* when saving the method. Otherwise, you have two ejbCreate methods; the new one with three parameters, and the generated one with the key as the only parameter. You can also delete the old ejbCreate method explicitly.

ejbRemove method

The ejbRemove method is the most simple of the required methods, it uses a prepared DELETE statement to remove the row corresponding to the primary key (Figure 8-10).

Note that if something goes wrong during the deletion, a RemoveException is thrown to indicate the failure.

```
public void ejbRemove() throws RemoteException, RemoveException {
    TransactionRecordKey key = (TransactionRecordKey)
        getEntityContext().getPrimaryKey();
    Timestamp trID = key.primaryKey;
    PreparedStatement pstmt = null;
    Connection conn = null;
    Properties prp = getEntityContext().getEnvironment();
    String tableName = prp.getProperty( "TABLE_NAME" );
    try {
        conn = getConnection();
        pstmt = conn.prepareStatement( "delete from " + tableName +
            " where transid = ?" );
        pstmt.setTimestamp(1, trID);
        if ( pstmt.executeUpdate() == 0 ) {
            // something went wrong
            throw new RemoveException( "TransactionRecord #" + trID +
                " was not removed from the DB" );
        }
    } catch ( SQLException exc ) {
        System.out.println( "Error occurred while removing TransactionRecord #"
            + trID );
        exc.printStackTrace();
    } finally {
        cleanup( pstmt, conn );
    }
}
```

Figure 8-10 TransactionRecord ejbRemove method

ejbFindByPrimaryKey method

In the BMP world, the finder methods have to be implemented in the bean class and then promoted to the home interface.

The method basically executes a prepared SELECT statement. If the result set has returned some value, the key is returned. In all other cases, a FinderException is created and thrown (Figure 8-11).

```

public TransactionRecordKey ejbFindByPrimaryKey( TransactionRecordKey aKey)
    throws RemoteException, FinderException, ObjectNotFoundException {
    PreparedStatement pstmt = null;
    Connection conn = null;
    ResultSet rs = null;
    Properties prp = getEntityContext().getEnvironment();
    String tableName = prp.getProperty( "TABLE_NAME" );

    try {
        conn = getConnection();
        pstmt = conn.prepareStatement( "select transid from " +
            tableName + " where transid = ?" );
        pstmt.setTimestamp( 1, aKey.primaryKey );
        // execute the query and read the values
        rs = pstmt.executeQuery();
        if (!rs.next()) throw new ObjectNotFoundException
            ("Cannot find TransactionRecord #"+ aKey.primaryKey);
    } catch (SQLException exc) {
        System.out.println( "Exception while querying TransactionRecord #"
            + aKey.primaryKey );
        throw new FinderException( exc.toString() );
    } finally {
        cleanup( pstmt, conn );
    }
    return aKey;
}

```

Figure 8-11 TransactionRecordBean ejbFindByPrimaryKey method

Modifying the home and the remote interfaces

After the methods have been implemented in the bean class, some have to be added to the remote interface. To achieve this, select the methods and, from the context menu, select *Add to -> Remote interface*. The method signatures will be copied to the remote interface, but they will be modified to add `RemoteException` to the throws clause (if it was not yet present).

The methods to be added to the remote interface are `getAccID`, `setAccID`, `getTransamt`, `setTransamt`, `getTranstype`, `setTranstype`, and `getTransID`.

The getter methods should be marked read-only. Select the methods and, from the context menu, select *EJB Method Attributes -> Read-only Method*. This instructs the container that these methods are not changing the bean state and therefore the `ejbStore` method will not be invoked.

The create and finder methods will be made available on the home interface by selecting them and choosing from the context menu *Add To -> Home interface*. Their signatures will be copied to the home interface with the following changes:

- ▶ void ejbCreate(..) becomes TransactionRecord create(..)
- ▶ TransactionRecordKey ejbFindByPrimaryKey(..) and Enumeration ejbFind(..) become TransactionRecord findByPrimaryKey(..) and Enumeration find(..) respectively. Note that we do not have any ejbFind methods that return an Enumeration.

Testing the BMP entity bean

Now that the bean is written, we can generate the deployed code and after that test the enterprise bean.

To generate the deployed code, which means tying your EJB to the container inside VisualAge for Java, select the EJB group containing your EJB and, from the context menu, invoke *Generate deployed code*.

Starting the servers

Start the WebSphere Test Environment and the persistent name server. Wait until the server is ready.

Select *DataSource Configuration* and check that the EJBBANK data source exists. See “Data sources” on page 82 for instructions on how to create a data source.

Add the BMP_Entity group to the server configuration. Select the server and open the Properties window. The connection type should be <DataSource> and point to EJBBANK (see Figure 6-10 on page 92 for more information). Start the EJB server.

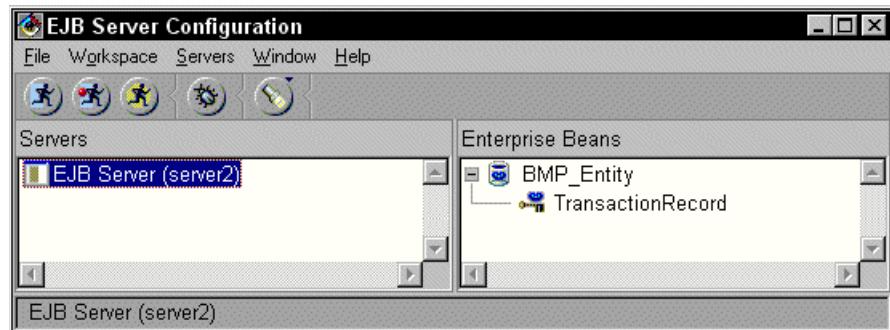


Figure 8-12 EJB server configuration for BMP_Entity group

Testing the EJB with the test client

Once the servers are started, you can launch the EJB test client by selecting the TransactionRecord bean and selecting from the context menu *Run test client*. See “EJB test client” on page 94 for more information.

First click on *Lookup* to locate the home of the TransactionRecord bean:

```
itso/ejb35/bmp/TransactionRecord
```

Once you have a reference to the home, you can create a TransactionRecord (Figure 8-13).

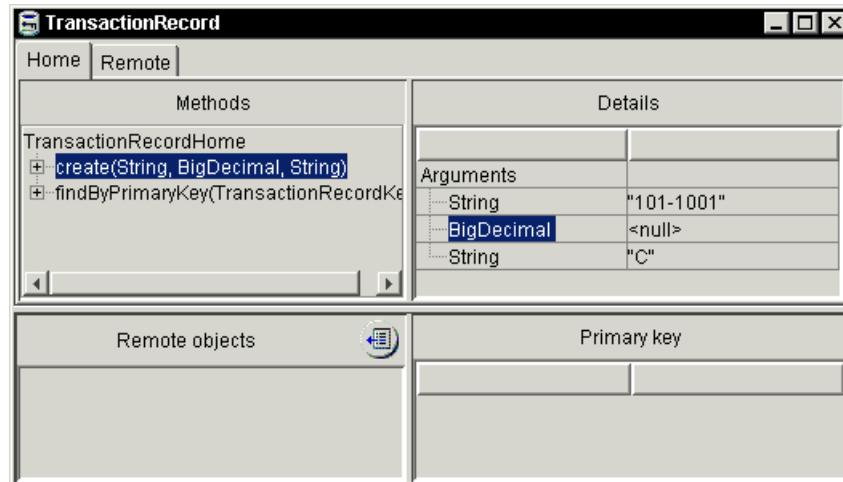


Figure 8-13 Creating a TransactionRecord

- ▶ Select in the left panel (*Methods*), the method `create(String, BigDecimal, String)`.
- ▶ Fill in the simple required parameters, such as the account number (101-1001) and the transaction type (C or D).
- ▶ For complex types, such as `BigDecimal`, follow the instructions in “Constructing and inspecting objects” on page 101 (use the *new constructor*).
- ▶ Run the `create` method. The test client creates a new bean and calls the callback methods to store the bean in the table. The dialog switches to the Remote tab of the window, and there you will be able to execute the get and set methods on the bean.

Retrieving an existing record

This is a little more complex, because you have to create a `Timestamp` object for an existing row in the table. For the `Timestamp` key value, select `new(..)` to invoke a constructor.

Select the `Timestamp(int,int,...)` constructor and enter values that match a row:

```
SQL row display: 2001-05-16-13.48.44.972000
Constructor values: 101, 4, 16, 13, 48, 44, 972000000
(years start at 1900, month is 0 to 11, second fraction is in nanoseconds)
```

Summary

BMP are quite difficult and long to write. But, with some abstractions like those we introduced in the code (using environment variables from the deployment descriptor), they offer quite a good deal of flexibility and can exploit your persistent store, whatever it is.

Pros and cons

After seeing how BMPs are built, here are some points that could help you make a decision.

Advantages

- ▶ Basically access to any persistent store
- ▶ Optimization can be done on each access

Disadvantages

- ▶ You have to write a lot of code
- ▶ Data source dependency: difficulty to isolate the bean from a specific data source (unlike CMP that allows you to switch from one DB to another and eventually from vendor to vendor)



Session beans

Session beans are enterprise beans that represent business processes performed on behalf of a client under a single or multiple interactions. These can contain typically business logic, rules, workflow and algorithms. Examples include tax calculation, currency conversion, interest calculation, and order fulfillment.

This chapter covers the basics about session beans, the two different types, stateless and stateful, and shows how to use VisualAge for Java for developing and testing session beans.

Session bean basics

Every session bean must have a bean class, a remote interface, and a home interface, as described in “An enterprise bean and its interfaces” on page 36.

The bean class of a session bean implements the business methods. In a simplified world, each business method is a user transaction that interacts with entity beans, transactions systems, relational databases, and so forth. However, more complicated scenarios are possible.

Each business method is defined in the remote interface of the session bean. The home interface is simple and is used to create and remove session beans; there are no finder methods.

There is no key class for a session bean.

Session bean lifetime

The lifetime of a session bean is shorter than that of an entity bean. Generally, they have the lifetime equivalent of a client session. A client session lasts as long as the client is connected to the application. After clients have remained inactive for a certain period of time, the EJB container can remove the bean instances from the memory. Session beans do not survive software or hardware crashes.

During the lifetime of a session bean, the bean cycles between different states. The life cycle of session beans is managed by the container. As we will see later, these states are different for the two types of session beans, stateful and stateless.

Conversational state

A typical application holds conversations with clients at various levels. A conversation is the interaction between a client and an application, and it can represent one or more business processes. This interaction requires often that conversational data between the client and the application should be stored and accessed, whenever needed. An example of conversational data is a shopping cart that contains all the products that a customer wants to purchase. This data is also known as session data.

When enterprise beans are used, the conversation between a client and beans can vary, depending on the type of the bean. Session beans behave in a different way, as far as the conversation duration is concerned. A conversation can span a single method call or multiple method calls. Based on that, session beans are divided in two different types, stateless and stateful.

A **stateless** session bean holds conversations that span a single method call. After each method call, a stateless session bean releases any information from the previous method call and becomes available, so that it can serve a new request. When the same client is invoking another method, a different session bean may serve the new request. Therefore, stateless session beans are not allowed to store any conversational state from method to method.

A **stateful** session bean holds conversations that can span many method calls. All the method calls that come from a single client are served by the same bean. During this conversation, the bean holds conversational state for that client. In advanced cases, the conversational state may contain open resources, such as open database connections. The conversational state of a stateful session bean is stored in the bean instance fields.

A stateful session bean can also start a transaction that can be spanned across multiple method calls. This means that a transaction is started in one method and ended in another (see Chapter 12. “Transaction management” on page 217). From a design point of view, this is something that should be generally avoided, because there is no good control over the transaction context. However, certain requirements may lead to the use of this.

Let's now see in more detail the stateful and stateless session beans and their characteristics.

Stateful session bean

Stateful session beans are components that represent business processes that can be performed within multiple method calls. The state of each method call is preserved in the stateful session bean. The limitation of this is that a stateful session bean can only serve one client during its entire life.

In the following sections, we will take a better look to stateful session beans and the way they serve clients as well as how the container manages them.

Conversational session beans

Stateful session beans are components that can keep state across client invocations. Any data passed or generated from one method call to another is saved in the instance variables of the bean instance. Therefore, this unique bean instance acts as an extension to the client.

Because each instance of a stateful bean is dedicated to one client, it cannot be pooled and serve other clients, if the first client is inactive. This problem is solved

by passivating an instance after a certain inactivity time and activating it back, when the client invokes a new request.

Activation and passivation of session beans

The activation/passivation mechanism is the alternative to the pooling mechanism of entity beans and stateless session beans.

As we saw, the conversational state between a client and a session bean is stored in the instance variables of the stateful bean. For a stateful bean instance to be passivated, the bean has to follow the Java object serialization rules. When a container passivates a bean, it saves its conversational state to the hard disk or other storage using object serialization. For an instance variable to be serialized, it has to be a non-transient primitive type or a serializable Java object.

After passivation, the memory occupied by the bean instance can be reclaimed. When the original client invokes a new method call, the passivated conversational state is read from the disk and swapped in to a bean instance. This instance does not have to be as the one before passivation time. When activation is complete, the bean can resume the conversation with the original client.

Passivation can occur when a bean instance has not been used for a specific amount of time. When the number of stateful bean instances reaches the maximum limit, the container will passivate instances that have been called least recently. The container can passivate an instance any time, unless the bean is involved in the method call. The container cannot passivate an instance, if that instance is involved in a transaction. The transaction has to complete first.

Life cycle of a stateful bean

Stateful session beans are neither shared nor reused among multiple clients. Unlike stateless session beans and entity beans, they are not pooled. The life cycle of a stateful session bean has three states:

Does-not-exist This is the same for stateless session beans and entity beans. In this state, the bean has not been instantiated yet and it does not exist in memory.

Method-ready When a client needs to work with a stateful session bean instance, it invokes the create method on the EJB home object provided by the container. To achieve this, the container instantiates a new bean and assigns it to the EJB object. The instance bean is then ready to service client requests. Because each instance is assigned to an EJB object, stateful

session beans consume many more resources than stateless session beans.

Passivated	When a stateful session bean is not actively used (not servicing client requests), the container may decide to passivate it. The container saves the bean's state to the passivation directory specified at the container level properties.
-------------------	---

Life cycle example

Let's see now what happens inside the container, when a client invokes a method on a stateful session bean (Figure 9-1).

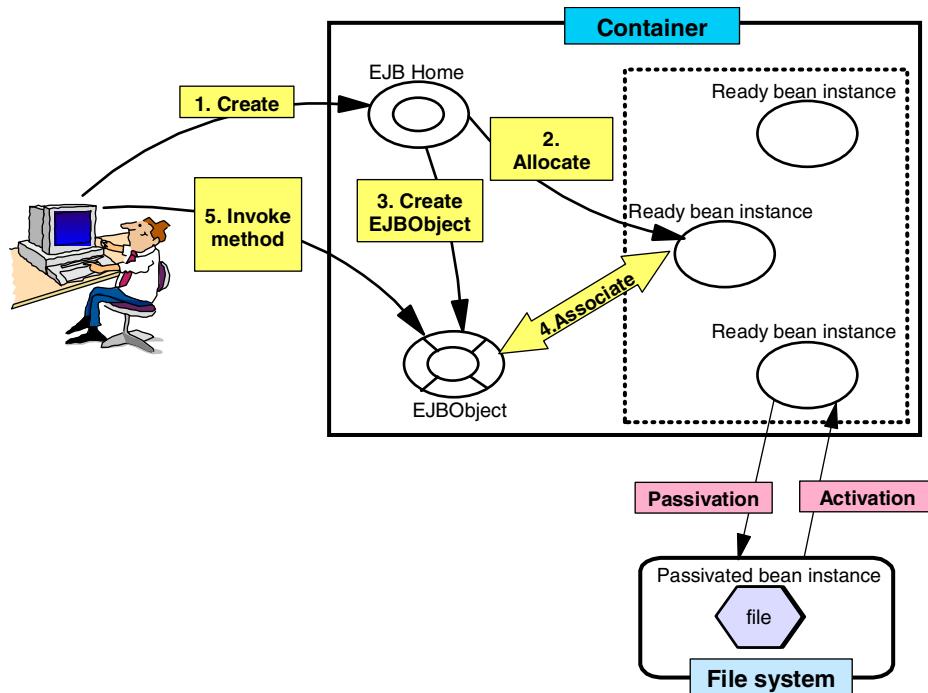


Figure 9-1 Stateful bean - instance pooling

- 1. Create:** The client can call a method on a bean only by using the bean's remote interface, the EJB object. Therefore, a reference to an EJB object is required. This is achieved by invoking the create method on the bean's home interface, the EJB home.
- 2. Allocate:** The EJB home allocates a bean instance from the pool, so that it can associate it with the EJB object that is going to be created.

3. **Create EJB object:** The EJB home creates the EJB object that will make all the business methods of the bean available to the client.
4. **Associate:** After the instance has been taken out from the pool and the EJB object has been created, the EJB home can associate the instance with the EJB object and return the EJB object to the client.
5. **Invoke method:** Now the client is ready to call a business method. The EJB object passes the method call to the instance. After the instance has served the request, the instance is unregistered and returned to the pool.

This description illustrates what happens behind the scenes, when a client interacts with a stateful session bean. Many things have been simplified here for giving a more clear picture. The line between the tasks performed by the EJB home and the container is not always easy to tell. In our description, we use the EJB home as the party that initiates an action. Usually that action is then delegated to the container.

Stateless session bean

Stateless session beans are components that represent business processes that can be performed in a single method call.

In this section, we explore the stateless session beans and take a better look at their attributes and their association with clients.

Non-conversational session beans

Stateless session beans do not hold any conversational state on behalf of clients. This is due to the fact that a stateless session bean is only associated with a client during a single method call. After that method call, another stateless bean will serve the client. The bean instances that serve the same client are swapped freely and do not have to be passivated to a file system and activated back.

Because there is no tight relationship between a bean instance and a client, the client will not be affected, if the bean instance is lost due to a server crash. The client can use any other bean instance.

Stateless session beans usually provide services that are fairly generic and reusable. They need to fulfill a service in one method invocation, because they cannot remember anything from one method call to the other. Therefore, each time a client calls a method, it must pass all the required data that the bean needs. Therefore, stateless beans resemble traditional transaction processing applications, which are executed using a procedure call.

Nevertheless, stateless session beans can have instance variables, therefore some internal state. For example, a stateless bean can have an instance field that tracks the number of times that the bean instance was called. However, this information may not be meaningful to a client, because each bean instance has a different value. Therefore, these variables should be of generic use, like a reference to a home interface of another bean that is used in the session bean methods.

Let's see now, in more detail, how this affects the life cycle of a stateless bean.

Life cycle of a stateless session bean

The stateless session bean life cycle has two states:

Does-not-exist	When a bean instance is in the <i>does-not-exist</i> state, this means that it has not yet been instantiated.
Method-ready pool	When a bean instance is instantiated by the container and is ready to serve client requests, it is in the <i>method-ready pool</i> state.

The container moves a stateless session bean from the does-not-exist state to the method-ready pool state by performing three operations:

- ▶ Invoke the `Class.newInstance` method on the stateless bean class.
- ▶ Invoke the `setSessionContext(SessionContext context)` method on the bean instance.
- ▶ Invoke the `ejbCreate` method on the bean instance.

Life cycle example

Figure 9-2 shows what happens inside the container, when a client invokes a method on a stateless session bean (simplified for illustration purposes).

1. **Create:** The client invokes the `create` method on the EJB home in order to get a reference to an EJB object.
2. **Allocate:** The EJB home attempts to allocate an instance from the ready bean instances that reside in the container. If there already has been a conversational state and the dedicated instance to that client has been passivated, the container will activate it back. This means that the container will read the state from the disk and apply it to a ready bean instance.
3. **Create EJB object:** The EJB home creates the EJB object that will make all the business methods of the bean available to the client.
4. **Associate:** After the instance has been retrieved and the EJB object has been created, the EJB home can associate the instance with the EJB object and return a reference of the EJB object to the client.

- Invoke method:** Now the client is ready to call a business method. The EJB object passes the method call to the instance. If the client inactivity time-out and the maximum number of bean instances have been reached, the container will passivate the bean instance to the disk.

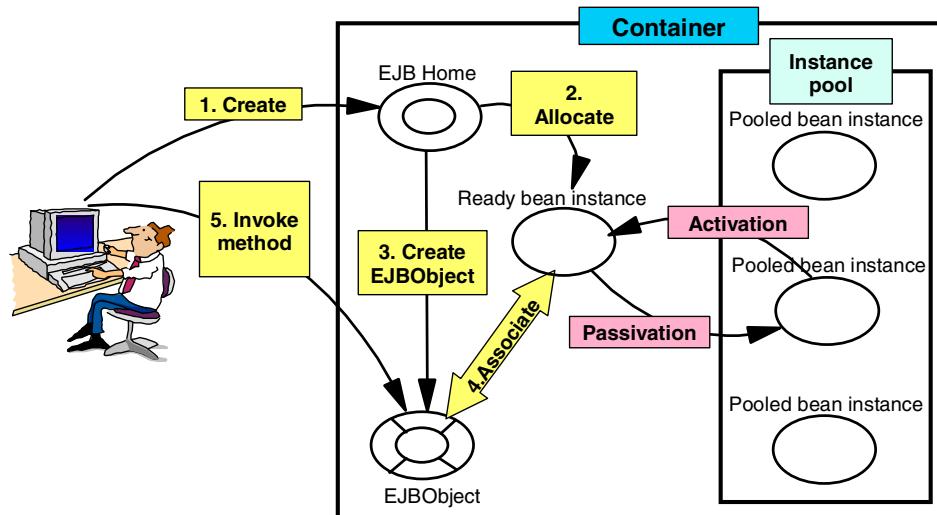


Figure 9-2 Stateless bean - instance pooling

Note: This diagram is valid also for entity beans. An inactive ready bean is released from the EJB object and returned to the pool (passivation). The bean enters the pooled state. At a new request, the pooled bean is associated with another EJB object (activation). Then the bean is again in ready state.

Let's now discuss all the semantics on how to implement a session bean.

Implementing session beans

As we described in “Session bean basics” on page 158, what constitutes a session bean is the bean class, the remote interface, and the home interface.

It is the business methods that implement all the business processes served by the bean. Example of business methods are calculate tax, validate a credit card, or send a notification. All these methods are defined in the bean class and can be part of an order fulfillment process.

On the other hand, it is the management methods that are part of the infrastructure logic, that is used internally by the sever. These methods are callback methods, that the container calls to inform the bean instance of significant events. Clients cannot call them, because these methods are not available in the remote interface. The management methods are defined by the interfaces, that the session bean implements. They are usually prefixed by ejb, such as ejbPassivate.

Let's now see the role these methods play in the session bean class, the remote interface, and the home interface. In the next sections, we use the example of a transfer session bean that can transfer money between two bank accounts. We will actually build this session bean later using VisualAge for Java.

Session bean class

A session bean class implements the javax.ejb.SessionBean interface. The javax.ejb.SessionBean interface extends the more generic javax.ejb.EnterpriseBean interface.

The session bean class defines and implements the business methods of the bean, and implements the management methods used by the container.

By convention, the bean class is named *NameBean*, where *Name* is the name you assign to the bean. The bean class for the example transfer session bean is named **TransferBean**.

Figure 9-3 shows the skeleton of the TransferBean. Notice that there is an instance variable of type BankAccountHome, called bankAcHome. We will see how this variable is used in the next section.

Every session bean class must meet the following requirements:

- ▶ It must be public, it must not be abstract, and it must implement the javax.ejb.SessionBean interface and all of its methods.
- ▶ It must define and implement the business methods that execute the tasks associated with the bean.
- ▶ It must define and implement an ejbCreate method, for each way you want to instantiate the bean class (stateful beans only).

Stateful session beans may need to synchronize their conversational state with the transactional context in which they operate. For example, a stateful session bean may have to reset the value of some of its variables if a transaction is rolled back. If a bean has to synchronize its conversational state with the transactional context, the bean class implements the javax.ejb.SessionSynchronization interface. This interface contains methods to notify the session bean when a transaction begins, when it is about to complete, and when it has completed.

```
package itso.ejb35.session;
import java.math.*;
import java.rmi.*;
import java.security.Identity;
import java.util.Properties;
import javax.ejb.*;
import javax.naming.*;
import itso.ejb35.cmp.*;

public class TransferBean implements javax.ejb.SessionBean {

    private SessionContext mySessionCtx = null;
    final static long serialVersionUID = 3206093459760846163L;
    private BankAccountHome bankAccHome;

    public void ejbActivate() throws RemoteException {
    ...
    }
    public void ejbCreate() throws CreateException, RemoteException {
    ...
    }
    public void ejbPassivate() throws RemoteException {
    ...
    }
    public void ejbRemove() throws RemoteException {
    ...
    }
    public SessionContext getSessionContext() {
    ...
    }
    public void setSessionContext(SessionContext ctx)
        throws RemoteException {
    ...
    }
    public void transferMoney(String fromAccountID, String toAccountID,
        BigDecimal anAmount) throws InsufficientFundException {
    ...
    }
    private BankAccountHome getBankAccountHome() {
    ...
    }
}
```

Figure 9-3 TransferBean class skeleton

Note: The bean class can implement the bean remote interface, but this is not recommended. If the bean class implements the remote interface, it is possible to inadvertently pass the `this` variable as a method argument (VisualAge for Java will not allow you to do that).

As you see in this example, the bean class can contain a lot of code. The developer does not have to write most of this code. VisualAge for Java is generating it, so that the developer can concentrate mainly on business logic.

Implementing the ejbCreate method

A stateless session bean can have only one `ejbCreate` method, which must return `void` and contain no arguments.

For stateful session beans, you must define and implement an `ejbCreate` method for every way in which you want a bean to be instantiated. A stateful session bean can have multiple `ejbCreate` methods. Each `ejbCreate` method must correspond to a `create` method in the bean home interface. Note that there is no `ejbPostCreate` method in a session bean, as there is in an entity bean.

Instead, the client invokes the `create` method in the home interface, it triggers the container to create a bean instance and finally the container invokes the `ejbCreate` method in the bean class. If an `ejbCreate` method is executed successfully, an EJB object is created.

Each `ejbCreate` method must return `void` and contain code to set the values of all variables needed by the EJB object.

Figure 9-4 shows the `ejbCreate` method of the `TransferBean` class.

When a container invokes `ejbCreate` at instance creation time, we get the home interface of the `BankAccount` bean, the `BankAccountHome`. We do that by making a lookup in the initial context and using the naming services. We then store the retrieved home interface of the bank account in the transfer bean field `bankAccHome`. This home interface is required for finding the accounts, before transferring money.

We do that in the `ejbCreate` method to ensure that this task is done only once for each transfer bean instance. However, this approach may not be efficient, if we would subclass the `TransferBean` class. Then we would have to redefine the `ejbCreate` method again, because it will not be inherited.

```

public void ejbCreate() throws javax.ejb.CreateException,
    java.rmi.RemoteException {
    InitialContext ctx = null;
    Properties prop = new Properties();
    try {
        //prop.put( Context.PROVIDER_URL,"iiop://" );
        //prop.put( Context.INITIAL_CONTEXT_FACTORY,
        //          "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
        //ctx = new InitialContext(prop);
        ctx = new InitialContext();
        bankAccHome = (BankAccountHome) javax.rmi.PortableRemoteObject.narrow
            ctx.lookup("itso/ejb35/cmp/BankAccount"),BankAccountHome.class );
    } catch ( NamingException exc ) {
        System.out.println( "Error retrieving the home of BankAccount" );
        exc.printStackTrace();
    }
}

```

Figure 9-4 Transfer ejbCreate method

Implementing the management methods

Every session bean class must implement the methods inherited from the javax.ejb.SessionBean interface. All of these methods must be public, return void, and throw the java.rmi.RemoteException exception. Figure 9-5 shows these methods as they are generated by VisualAge for Java.

```

public void ejbActivate() throws RemoteException {}

public void ejbPassivate() throws RemoteException {}

public void ejbRemove() throws RemoteException {}

public javax.ejb.SessionContext getSessionContext() {
    return mySessionCtx;
}
public void setSessionContext(SessionContext ctx) throws RemoteException {
    mySessionCtx = ctx;
}

```

Figure 9-5 Transfer management methods

ejbActivate

When the container assigns a bean instance to an EJB object, it calls this method. How the bean instance is retrieved varies from stateless to stateful beans.

- Stateless The container selects a bean instance from the pool.
- Stateful The container reads the serialized state of the corresponding passivated instance from the file system. Stateful bean instances are not pooled.

This method must contain any code that you want to execute, when the instance is about to be activated.

ejbPassivate

When the container disassociates an instance from its EJB object, it calls this method. The way the disassociation is performed is different for stateless and stateful session beans.

Stateless The container returns the instance back to the pool.

Stateful The container serializes the state of the instance to the file system, so that it can be retrieved when the client will make another request.

This method must contain any code that you want to execute, when the bean instance is about to be passivated (deactivated).

ejbRemove

When a client invokes the remove method on the home interface (inherited from the javax.ejb.EJBHome interface), the container calls this method. This method destroys the EJB object, that was created earlier by calling the ejbCreate method. The developer should release any resource that has been allocated, for example, open database connections. The ejbRemove method can be also called at any time by the container, when the container determines that the bean life has expired.

setSessionContext and getSessionContext

These methods are used in a conventional way to set or get the value of mySessionCtx. The setSessionContext method is invoked by the container to pass a reference to the javax.ejb.SessionContext interface to a session bean instance. A bean can use the session context to query the container about current transactional state, current security state, and more. This method must contain any code required to store a reference to the context.

Modifying management methods

As shown in Figure 9-5, except for the setSessionContext and getSessionContext methods, all of the callback methods in the TransferBean class are empty.

No additional action is required here for our bean life cycle states associated with these methods. However, these methods may be enhanced with additional code, especially in the case of stateful session beans.

Implementing the business methods

The business methods of a session bean render the business logic that is provided by the bean. These methods implemented in the bean class cannot be directly invoked by a client. Instead, the client invokes the corresponding methods on the EJB object, that is associated with a bean instance.

The EJB object implements the remote interface and defines all the business methods implemented in the bean class. Finally, it is the container that invokes the business methods in the instance. When the bean is requested, an EJB object is created by the container.

Figure 9-6 shows the transferMoney business method for the TransferBean class. This is the only business method implemented in the TransferBean.

```
public void transferMoney(String fromAccountID, String toAccountID,
                           BigDecimal anAmount)
                           throws InsufficientFundException {
    // Initialize the two accounts
    BankAccount fromAccount = null, toAccount = null;
    try {
        // Find the two accounts
        fromAccount = bankAccHome.findByPrimaryKey(
            new BankAccountKey(fromAccountID));
        toAccount = bankAccHome.findByPrimaryKey(
            new BankAccountKey(toAccountID));

        // Perform the money transfer
        fromAccount.withdraw(anAmount);
        toAccount.deposit(anAmount);

    } catch(FinderException exc) {
        exc.printStackTrace();
        System.out.println(
            "FinderException: an account could not be found!");
    } catch(RemoteException exc) {
        exc.printStackTrace();
        System.out.println(
            "RemoteException: an account could not be accessed!");
    }
}
```

Figure 9-6 TransferBean transferMoney business method

This method is used to transfer a given amount between two accounts. These accounts are two entity beans. First the EJB objects of the two accounts are found by using the `findByPrimaryKey` method. Then, the `transferMoney` method calls the `withdraw` method on one account and the `deposit` method on the other. Notice that the two account variables have to be initialized, before attempting to find them.

It is critical for our application that both the `withdraw` and the `deposit` method execute successfully. If one of them fails, the transfer money action must be canceled. For this reason, these two method calls have to run within the same transaction. This is assured by the transaction services provided by the container and the developer does not have to worry about it.

Like all finder methods, `findByPrimaryKey` can throw both `FinderException` and `RemoteException`. The try/catch blocks are set up around invocations of the `findByPrimaryKey` method to handle the entry of invalid account IDs by users. If the user enters an invalid account ID, the `findByPrimaryKey` method cannot locate the corresponding EJB object, and the finder method throws the `FinderException` exception.

The throws clause of the `transferMoney` method contains an `InsufficientFundException`. This exception is thrown by the `withdraw` method of the `AccountBean` for preventing from transferring amounts that exceed the account balance. This exception should be handled by the EJB client that invoked the transfer. The `transferMoney` method should handle only system exceptions.

More about transactions and EJBs can be found in Chapter 12, “Transaction management” on page 217. For more on exceptions and how to manage them, see Chapter 19, “Exception handling” on page 399.

Remote interface

A session bean remote interface provides access to the business methods implemented in the bean class. It also provides methods to remove a bean instance and to obtain a reference to the home interface. The remote interface is defined by the developer and is implemented in the EJB object class, generated by the container tooling during deployment.

By convention, the remote interface is named `Name`, where `Name` is the name you assign to the bean. For example, the `Transfer` bean remote interface is named `Transfer`.

Every remote interface must meet the following requirements:

- ▶ It must extend the `javax.ejb.EJBObject` interface.

- ▶ You must define a corresponding business method for every business method implemented in the bean class.
- ▶ The parameters and return value of each method defined in the interface must be valid for Java RMI-IIOP.
- ▶ Each method throws clause must include the RemoteException.

Figure 9-7 shows the remote interface, Transfer, for the transfer bean. This interface defines the method for transferring money between two accounts. Notice that this method is implemented in the bean class.

```
package itso.ejb35.session;
import java.math.*;
public interface Transfer extends javax.ejb.EJBObject {
    void transferMoney(String fromAccountID, String toAccountID,
                       BigDecimal anAmount) throws
                           itso.ejb35.cmp.InsufficientFundException, java.rmi.RemoteException;
}
```

Figure 9-7 Transfer remote interface

Home interface

A session bean home interface defines the methods used by clients to create and remove instances of the bean and obtain metadata about an instance. The home interface is defined by the bean developer and is implemented in the EJB home class, generated by the container tooling during deployment. The container makes the home interface accessible to clients through the Java Naming and Directory Interface (JNDI).

By convention, the home interface is named NameHome, where Name is the name you assign to the bean. For example, the transfer bean home interface is named TransferHome.

Every home interface must meet the following requirements:

- ▶ It must extend the javax.ejb.EJBHome interface.
- ▶ Every method in the interface must be a create method, corresponding to an ejbCreate method in the bean class. Unlike entity beans, the home interface of a session bean contains no finder methods.
- ▶ The parameters and return value of each method defined in the interface must be valid for Java RMI-IIOP. In addition, each method throws clause must include the RemoteException exception.

Figure 9-8 shows the home interface, TransferHome, for the transfer bean. This interface defines one create method.

```
package itso.ejb35.session;
public interface TransferHome extends javax.ejb.EJBHome {
    Transfer create() throws javax.ejb.CreateException,
                           java.rmi.RemoteException;
}
```

Figure 9-8 Transfer home interface

A create method is used by a client to create a bean instance. A stateful session bean can contain many. However, a stateless session bean can contain only one with no arguments. This restriction on stateless session beans ensures that every stateless session bean instances is the same as every other instance of the same type.

For stateful session beans, each create method must be named create and it must have the same number and types of arguments as a corresponding ejbCreate method in the bean class. The return types of the create method and its corresponding ejbCreate method are always different. Each create method must meet the following requirements:

- ▶ It must return the type of the bean remote interface. For example, the return type for the create method in the TransferHome interface is Transfer.
- ▶ It must have a throws clause that includes the RemoteException exception, the CreateException exception class, and all of the exceptions defined in the throws clause of the corresponding ejbCreate method.

Attention: For most of the container implementations, when calling a create method on a home interface, it creates an EJB object, that is then associated with a bean instance. If the bean instance does not exist, it will be created. Usually bean instances are created at the startup time of an EJB container or after the maximum number of allowed instances has been passed.

When calling a remove method on a home or remote interface, it removes the EJB object but not the bean instance from the memory. Destroying a bean does not necessarily correspond with literally destroying in-memory bean instances. The container is the one that controls the life cycle of a bean instance, in order to provide instance pooling for higher performance.

Developing session beans with VisualAge for Java

In this section, we will show how to create and test a session bean inside VisualAge for Java.

Creating a session bean

In this section we will show to you how you can use VisualAge for Java for developing the transfer session bean that we just presented.

Creating a new bean

Make sure that you are in the EJB page in the workbench. First, you need to create an EJB group that should belong to the *ITSO EJB Redbook* project. We created a new **Stateless_Session** EJB group.

The technique of creating a session bean is very similar to creating an entity bean and is abbreviated in the description here:

1. Select the EJB group and select *Add-> Enterprise Bean*. The Create Enterprise Bean SmartGuide opens (Figure 9-9).

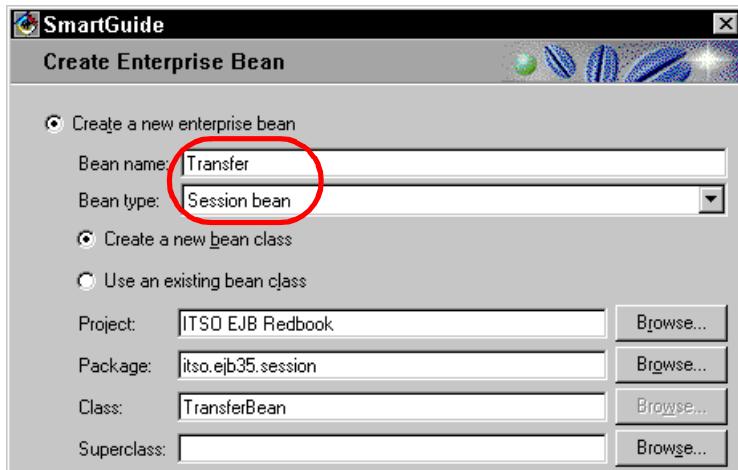


Figure 9-9 Creating a session bean

2. In the Create Enterprise Bean SmartGuide:
 - Select the radio buttons *Create a new enterprise bean* and *Create a new bean class*.
 - Enter **Transfer** as the Bean name.
 - Select *Session bean* as the Bean type.

- The Project field should be filled correctly.
 - Enter *itso.ejb35.session* as the Package.
 - Accept the Class name and leave the Superclass field empty.
3. Click on *Next* and in the Define Bean Class Attributes and Interfaces window:
- Accept the home and remote interface names.
 - Add these packages and class to the import list:
- ```

java.rmi.*
javax.ejb.*
javax.naming.*
itso.ejb35.cmp.* <== access to the entity beans
java.math.BigDecimal

```
4. Click on *Finish* to generate the initial types:
- Transfer (remote interface of the transfer bean)
  - TransferBean (bean class of the transfer bean)
  - TransferHome (home interface of the transfer bean)

You can have a look at what is generated inside each of these classes or interfaces by selecting it in the *Types* pane and looking at the displayed methods in the *Members* pane. As you can see, the remote interface is empty; it is up to you to decide which are the methods you need to put in the Transfer interface.

The home interface contains the create method that our session bean must have. Remember, because the transfer bean is a stateless session bean, we do not implement any additional create methods.

Now we can proceed with building the transfer bean. Go back to the EJB page of the Workbench. We will create a new field for the session bean.

## **Adding a field**

As you may have noticed, the transfer bean already has two fields. These fields were generated automatically in the bean class at the creation time. These are the javax.ejb.SessionContext mySessionCtx and the long serialVersionUID. The first field is used by the setSessionContext and getSessionContext methods. The second field is used for the serialization.

We want to store the home interface of the bank accounts in the Transfer bean as a private field, not accessible to clients. The reason why we declare the field as private is because our session bean is stateless. In a stateful session bean, you would have to declare the field as public, so that the container could take care of its state during passivation.

You can just code the statement for the field yourself:

```
private BankAccountHome bankAccHome;
```

Or, you can use a SmartGuide to create it. In the *Types* pane of the EJB page, select the TransferBean and *Add -> Field* from the context menu. In the Create Field SmartGuide:

- ▶ Enter bankAccHome as *Field Name*.
- ▶ Enter BankAccountHome as *Field Type*.
- ▶ Select private as *Access Modifier*.
- ▶ Deselect *Access with getter and setter methods*.
- ▶ Click *Finish*.

### Updating the ejbCreate method

Now we update the ejbCreate method and set the value of the field that we just created:

- ▶ Select the ejbCreate method in the *Members* pane.
- ▶ Update the method body with the code of Figure 9-4 on page 168.
- ▶ Save the changes.

### Adding a method

To add the transferMoney method, select the TransferBean class and *Add -> Method* to open the Create Method SmartGuide (Figure 9-10).

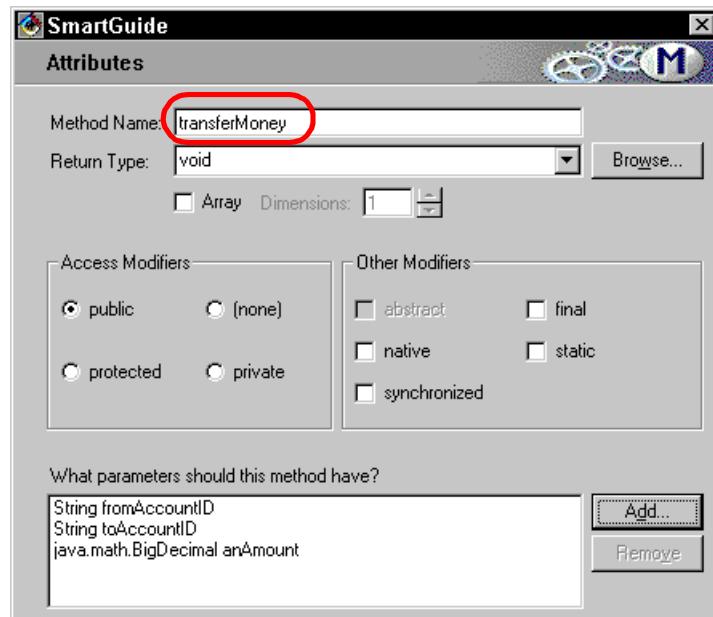


Figure 9-10 Adding a method to the session bean

1. Select the *Create a new method* radio button and click *Next*.
2. Enter transferMoney as method name and void as return type.
3. Select public as *Access Modifiers*; this method should be available to clients.
4. Add these parameters:
  - String fromAccountID
  - String toAccountID, and
  - BigDecimal anAmount
5. Click *Next* to go to the final page.
6. In the *What exceptions may this method throw* list, click on the *Add* button and enter the exception name InsufficientFundException.
7. Click *Finish* to generate the method skeleton.
8. Update the methods with the code of Figure 9-6 on page 170.

Finally, we have to add this method to the remote interface Transfer, so that it can be accessed by clients. Select the transferMoney method and select *Add to -> EJB Remote Interface* from the context menu. Notice the icon that is added as an indicator next to the method name.

You now have completed the development steps for the TransferBean. We will see in the next sections how to deploy it and test it inside VisualAge for Java.

## Deploying the bean

Before generating the deployment code, select the bean and *Properties* (Figure 9-11).



Figure 9-11 Session bean properties

The most important field is the *State Management Attribute*, which we set to `#STATELESS`. For the other fields, we leave the defaults for now.

After setting the deployment properties, generate the deployed code (select the bean and *Generate Deployed Code*).

## Testing the transfer bean inside VisualAge for Java

Now our session bean is ready and can be tested. Make sure that:

- ▶ You have an EJB group that contains all the entity beans that are used by the transfer bean. These beans are the BankAccount and TransRecord (see “Defining the bank model” on page 132).
- ▶ You should have created the EJBBANK database and a corresponding data source (see Figure 6-8 on page 90).

“Testing enterprise Java beans” on page 91 describes all the steps you follow for using the EJB test client in VisualAge for Java:

- ▶ Start the WebSphere Test Environment.
- ▶ Start the persistent name server.
- ▶ Check that the EJBBANK data source exists.
- ▶ Add the EJB groups that contain the entity beans and the session bean to an EJB server.
- ▶ Start the EJB server after checking its properties.

**Important:** Be careful to add both the EJB groups to the same server configuration in order to deploy all your beans inside a single container. If you use two different server configurations (containers), you will not be able to execute the transaction. WTE will try to start a distributed transaction and you will get a `TransactionRolledbackException`. You can avoid this problem by using a JTA enabled driver and data source (select *JTA* instead of *JDBC* when defining the data source), but for now just put all the EJB groups into the same container.

The `transferMoney` method in the transfer bean withdraws an amount from one account and deposits that amount to another one. After each successful transaction, it creates a transaction record in the database. Therefore, we need first to create the two accounts.

Either add two bank account records manually to the ACCOUNT table, or use the EJB test client to create two BankAccount beans, each with a positive balance. Alternatively, use two existing accounts from the EJBBANK database provided (101-1001 and 101-1002).

To test the Transfer session bean:

1. Start the EJB test client.
2. Lookup the home for the Transfer bean (`itso/ejb35/session/Transfer`).
3. Create a session bean (invoke `create` in the home interface).
4. In the remote interface (Figure 9-12), invoke the `transferMoney` method on the remote object, using the two account IDs.

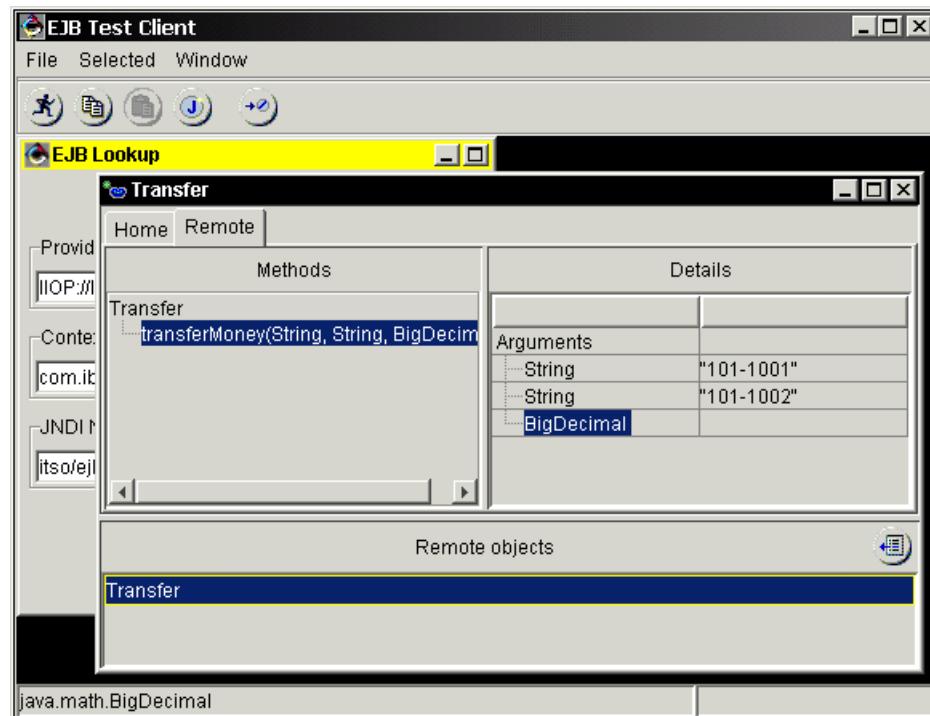


Figure 9-12 Testing the transfer bean

For the amount (`BigDecimal`), use the constructor to create a `BigDecimal` object. Invoke the `new` method on the `BigDecimal` argument. In the constructor window, select either the `BigDecimal(String)` or the `BigDecimal(double)` constructor, enter a value as argument, and invoke the constructor. Close the window by clicking *OK*.

5. To view the results, retrieve the accounts from their home using `findByPrimaryKey`, invoke the `getBalance` method, and then use *Inspect* on the `BigDecimal` result. Alternatively, use a DB2 command window to list the ACCOUNT table.

Test the money transfer with different amounts, once with an amount that triggers the InsufficientFundException. The test client should display the exception in a message window. You should also have at the end two transaction records in the database. You can verify that by querying the TRANSMERGED table in the EJB BANK database.

## Using session beans for database queries

In Chapter 10, “Custom finder methods” on page 193 we will discuss how to implement finder methods in entity beans for database queries. Such finder methods are very practical if the number of objects returned is relatively small. However, finder methods are impractical if a query returns hundreds (or thousands) of rows from a table, because an entity bean will be instantiated for every row when the result set is exhausted by the application.

As an alternative to finder methods in entity beans, queries can be implemented in session beans. A session bean can use JDBC to retrieve a subset (or all) of the columns from a table. Result data from such queries are often presented to a user in a list for selection. Only selected data is used to materialize entity beans for further processing.

There are multiple ways to generate the SQL query in a session bean:

- ▶ **JDBC coding**—the implementation of a JDBC query in a session bean would use coding similar to the ejbFindByPrimaryKey method in a BMP entity bean (Figure 8-11 on page 153), but with different search criteria and probably with a subset of the columns.
- ▶ **Select bean**—instead of JDBC coding a query can be constructed using the Data Access Bean feature of VisualAge for Java. A Select bean can be constructed for any database query and embedded in the session bean.
- ▶ **Stored procedure**—the query can be built as a stored procedure that runs in the database system. A stored procedure can be written in 3GL languages, and also in Java. DB2 and VisualAge for Java provide the Stored Procedure Builder to interactively create Java and SQL stored procedures and install them in the database system. The stored procedure is called from the session bean using JDBC or by embedding a Stored Procedure bean, which is part of the data access beans of VisualAge for Java.

To illustrate these techniques, let’s create a new stateless session bean called **CustomerFinder** in the Stateless\_Session EJB group. Then we use the three techniques to implement the same simple query:

```
select customerid, firstname, lastname from itso.customer
where lastname like '%'||:partialName||'%'
```

## Session bean with JDBC coding

Add a new `findJdbc` method to the `CustomerFinder` bean (Figure 9-13). This method returns a vector with the resulting names. A client could use the vector to display a list, and after selection of a customer the entity bean for that customer could be instantiated.

```
public java.util.Vector findJdbc(String partialName) {
 Vector result = new Vector();
 PreparedStatement pstmt = null;
 Connection conn = null;
 try {
 conn = getDatasource().getConnection();
 pstmt = conn.prepareStatement("select customerid,firstname,lastname"
 + " from itso.customer where lastname like '%'||?|| '%' ");
 pstmt.setString(1, partialName);
 ResultSet rs = pstmt.executeQuery();
 while (rs.next()) {
 result.add(rs.getString("customerid")+" "
 +rs.getString("firstname")+" "+rs.getString("lastname"));
 }
 return result;
 } catch (SQLException exc) {
 System.out.println("Exception names: " + partialName);
 return null;
 } finally { try { pstmt.close(); } catch (Exception e1){};
 try { conn.close(); } catch (Exception e1){} }
 }
}
```

Figure 9-13 Session bean with JDBC coding

The database connection is established using a `DataSource`:

```
private DataSource getDatasource() throws SQLException {
 DataSource ds = null;
 InitialContext ctx = null;
 Properties prop = new Properties();
 try {
 prop.put(Context.PROVIDER_URL, "iiop:///");
 prop.put(Context.INITIAL_CONTEXT_FACTORY,
 "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
 ctx = new InitialContext(prop);
 ds = (DataSource)ctx.lookup("jdbc/EJBBANK");
 } catch (NamingException exc) {
 System.out.println("error ... "); exc.printStackTrace();
 }
 return ds;
}
```

## Session bean with a Select bean

The Select bean of the data access beans is a powerful tool for non-SQL knowledgeable people to construct and run simple or complicated SQL statements. For simple statements JDBC coding is quite easy, but for a complicated join with expressions, the SQL Assist feature of the Select bean can be used to construct the SQL statement and the connection.

In this redbook we do not describe in detail how to use the data access beans. Refer to *VisualAge for Java Enterprise Version 2: Data Access Beans - Servlets - CICS Connector*, SG24-5265, for more details.

In short, to construct a Select bean with our SQL statement:

- ▶ Add the Data Access Bean feature to the Workbench.
- ▶ Start the WebSphere Test Environment and the persistent name server if you want to use a DataSource for the connection.
- ▶ Open the CustomerFinder class and switch to the Visual Composition.
- ▶ Use the Database palette, add a Select bean (it is named *Select1*), and open its properties.
- ▶ Select the *query* property and click the square button.
- ▶ Create a new Database Access class (itso.ejb35.session.DabAccess).
- ▶ Click *Add* for a new connection:
- ▶ Name the connection *conn*, select DataSource, the CNInitialContextFactory, leave the Provider URL empty, enter jdbc/EJBBANK as the DataSource, deselect Auto-commit (it is not supported in the JTA environment), and leave user ID and password empty. Click *Finish* to generate the conn method in the DabAccess class.
- ▶ On the SQL page, click *Add* for a new SQL statement:
- ▶ Name the statement *selectByLastname*, and use the SQL Assist SmartGuide to construct the SQL statement.
- ▶ In the SmartGuide, select the ITS0.Customer table, the condition LASTNAME contains the character(s) :PARTIALNAME, and the columns customerid, firstname, and lastname. Click *Finish* to generate the *selectByLastname* method in the DabAccess class.
- ▶ Click *OK* to close the query dialog.
- ▶ Save the bean. This generates a number of methods in the class, especially a *getSelect1* method that initializes the Select bean with the data from the DabClass methods. Another useful generated method is called *handleException*, and we can use it for exception handling.
- ▶ You can delete the *main* method that is generated.

Add a new `findDab` method to the `CustomerFinder` bean (Figure 9-14). This method returns the same data as the `findJdbc` method, but uses the `Select` bean to execute the SQL statement.

```
public Vector findDab(String partialName) {
 Vector result = new Vector();
 Select sel = getSelect1();
 try {
 sel.setParameterFromString(1,partialName);
 sel.execute();
 for (int i=0; i<sel.getNumRows(); i++) {
 result.add(sel.getColumnValue(0)+" "+sel.getColumnValue(1)+" "
 +sel.getColumnValue(2));
 }
 sel.disconnect();
 } catch (Throwable ex) { handleException(ex); }
 return result;
}
```

Figure 9-14 Session bean with data access bean

## Session bean with a stored procedure

For complicated SQL statements, or even a series of SQL statements, stored procedures provide fast processing, because they run in the database system.

For this example, we assume that a stored procedure with the name `db2admin.CustomerByLastname` exists in the database system and returns a result set of customers that match a partial name.

**Note:** We constructed the stored procedure using the DB2 Stored Procedure Builder. You can run this tool outside VisualAge for Java from the DB2 folder, or you can start it in VisualAge for Java from the *Project -> Tools* menu. The Stored Procedure Builder provides a SmartGuide similar to the Data Access Beans to construct SQL statements. It then installs the stored procedure into the DB2 catalog using a JAR file.

Once a stored procedure is available in the database system, it can be invoked from JDBC or by using the Stored Procedure bean of the data access bean. The Stored Procedure bean is constructed in the same way as the `Select` bean. You can use the same connection and create a new method to invoke the stored procedure.

For our example we used simple JDBC coding to invoke the stored procedure. Add a new findStp method to the CustomerFinder bean (Figure 9-15). This method returns the same data as the findJdbc method, but uses the stored procedure to execute the SQL statement.

```
public Vector findStp(String partialName) {
 Vector result = new Vector();
 CallableStatement pstmt = null;
 Connection conn = null;
 try {
 conn = getDatasource().getConnection();
 pstmt = conn.prepareCall("CALL DB2ADMIN.CUSTOMERBYLASTNAME(?) ");
 pstmt.setString(1, partialName);
 pstmt.execute();
 ResultSet rs = pstmt.getResultSet();
 while (rs.next()) {
 result.add(rs.getString("customerid")+" "
 +rs.getString("firstname")+" "+rs.getString("lastname"));
 }
 return result;
 } catch (SQLException exc) {
 System.out.println("Exception names: " + partialName);
 return null;
 } finally {
 try { pstmt.close(); } catch (Exception e1){};
 try { conn.close(); } catch (Exception e1){};
 }
}
```

Figure 9-15 Session bean with stored procedure

To complete the CustomerFinder session bean, promote the three methods (findJdbc, findDab, and findStp) to the remote interface and generate the deployed code.

To test the session bean, start the persistent name server, an EJB server with the Stateless\_Session group, and use the EJB test client to create an instance of the session bean and invoke the three methods. All three methods should return the same customer instances.

**Note:** VisualAge for Java 3.5.3 ships a technical preview tool that can wrap a stored procedure call for a session bean. This tool is available on the extra CD shipped with VisualAge for Java 3.5.3.

# Enterprise Access Builder session bean

The Enterprise Access Builder for Transactions is a VisualAge for Java tool that encapsulates interactions with other systems into commands using the Common Connector Framework (CCF).

We only describe in a short overview how a command works. For more details, refer to *Connecting Enterprise to the Web by Example, CCF Connectors and database connection*, SG24-5514, and the VisualAge for Java documentation.

## EAB command

A command uses a communication specification, and interaction specification, an input record, and an output record to invoke a transaction on a remote system (Figure 9-16).

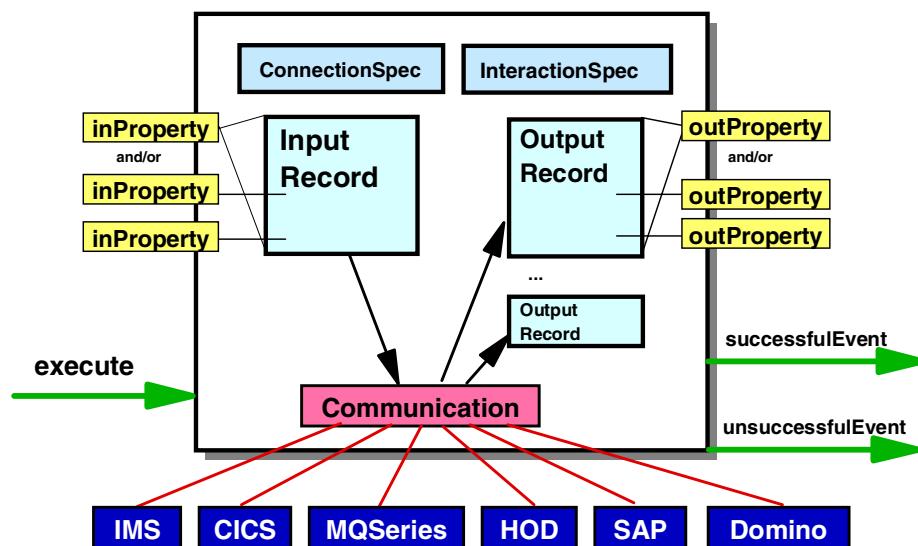


Figure 9-16 EAB command

The components of a command are:

- ▶ ConnectionSpec—specifies the target system and port to connect and what user ID and password to use for logon.
- ▶ InteractionSpec—specifies what program or transaction to invoke on the target system.
- ▶ Input record—is passed to the program or transaction (for example, a CICS communication area).

- ▶ Output record—result of the execution (some systems may provide one or multiple output records).
- ▶ Communication—is constructed from the connection specification and handles the actual connection to the target system.
- ▶ Properties—externalized (promoted) properties of input and output records that can be set (for input) and retrieved (on output).
- ▶ Execute—method to invoke the command.
- ▶ Events—after execution of the command.

ConnectionSpec and InteractionSpec are tailored for each target system.

## EAB tools and runtime

VisualAge for Java provides a set of tools to construct commands:

- ▶ Importer—construct record types and records from existing COBOL structures, BMS definitions, IMS MFS definitions, and 3270 screens.
- ▶ Record editor—create records from scratch or edit records after import.
- ▶ Command editor—create or edit a command (specify components and set properties).
- ▶ Mapper editor—create or edit mapper (a mapper can set input properties from existing business objects, or transfer output record properties into business objects).
- ▶ EAB test client—can test an EAB command (uses same technology as the EJB test client).
- ▶ EAB session bean tool—create a session bean that wraps a command.

WebSphere Application Server provides the runtime environment for executing commands. This includes facilities for connection pooling, coordination (commit processing), RAS (tracing), and logon processing.

## EAB session bean tool

The EAB session bean tool creates a session bean that can be used to execute one or multiple commands. A method of the EAB session bean accepts an input record as parameter, executes a command, and returns an output record. The session bean provides a connection specification that is used to drive the commands (the commands do not require a connection specification).

**Note:** You must add the IBM Enterprise Access Builder Library, IBM Common Connector Framework, IBM Java Record Library, and a suitable Connector to the Workbench, and you must have an existing EAB command.

For this example, we use an existing IMS command (ReadingCommand) from a previous redbook.

The EAB session bean tool is started from the EJB page of the Workbench by selecting an EJB group (for example, **EAB\_Session**) and *Open To -> EAB Session Bean Tool* (Figure 9-17).

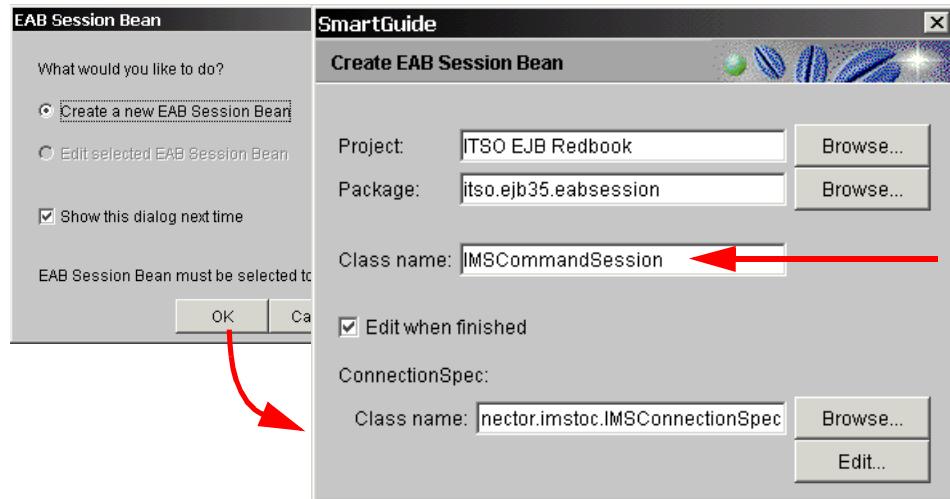


Figure 9-17 EAB session bean tool SmartGuide

- ▶ Enter the project, package, name of the session bean, and select the connection specification for the target system. Click *Next* (Figure 9-18).

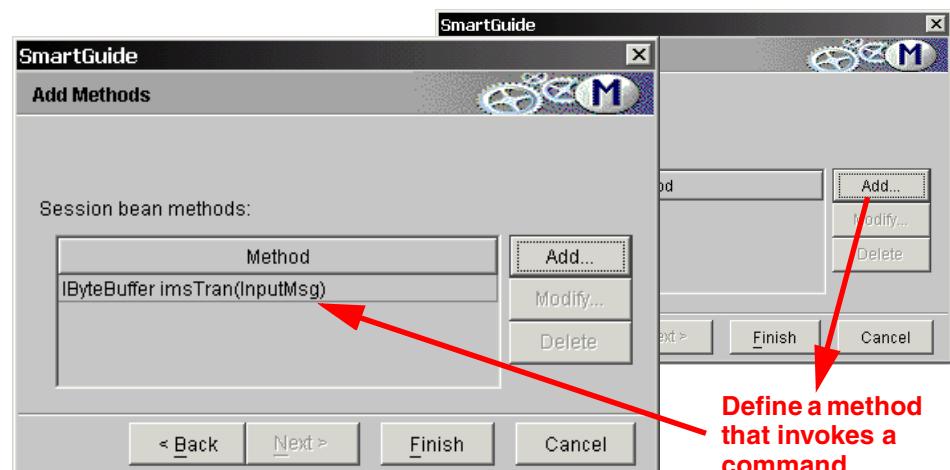


Figure 9-18 Adding methods to the session bean

- ▶ Click on *Add* to define a method that invokes a command. The session bean may contain methods for multiple commands.
- ▶ Each method is defined using another SmartGuide (Figure 9-19).

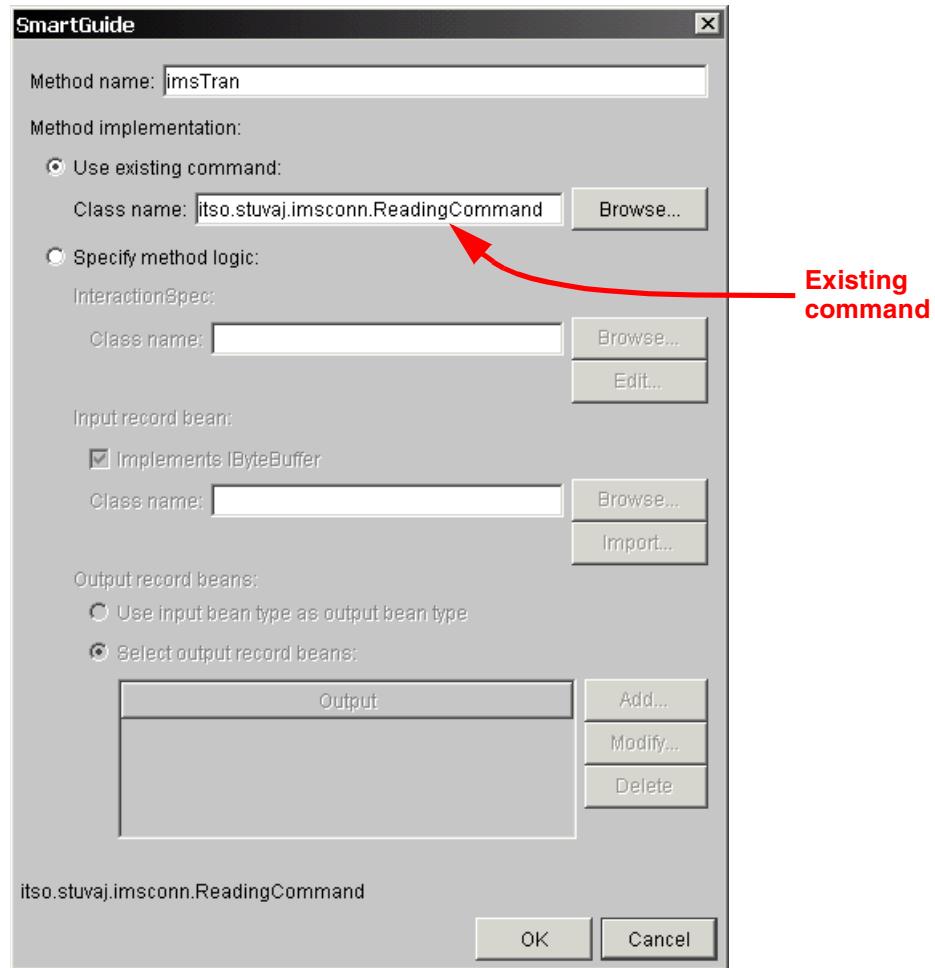


Figure 9-19 Define the methods of the session bean

- ▶ The method can use an existing command (as in Figure 9-19) or a command can be built on the fly using an interaction specification and input and output records. Click *OK* to add the method to the session bean.
- ▶ Click *Finish* when all the methods have been defined.

- ▶ This opens the EAB session editor (Figure 9-20). In this editor you can:
  - Tailor the connection specification
  - Select a method and edit the command components
  - Start the EAB command editor (double-click)

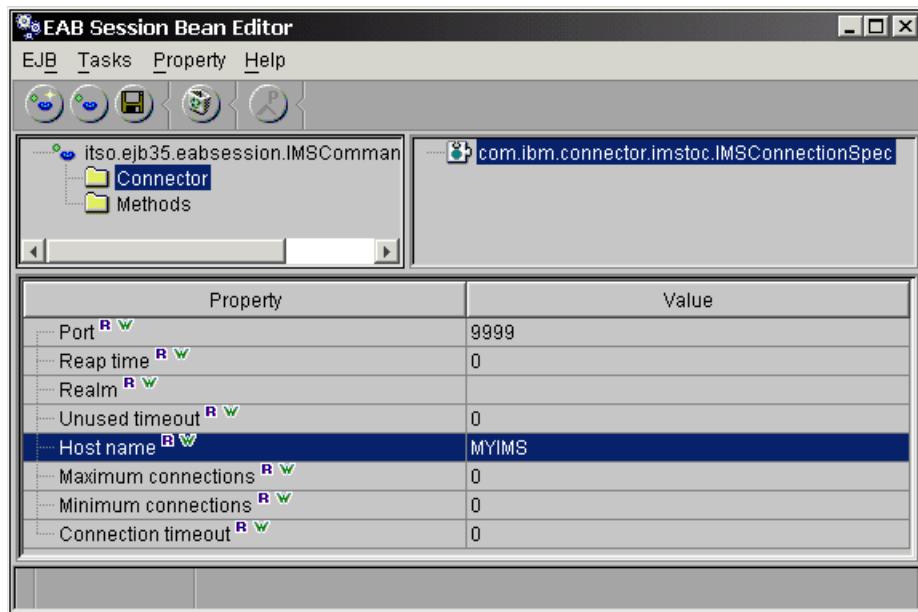


Figure 9-20 EAB session bean editor

- ▶ Save the session bean when finished.

In the EJB page you will find a complete session bean that provides a number of methods:

- ▶ The methods you defined (for example, imsTran). You have to add these methods to the remote interface. This method call the executeInteraction method.
- ▶ beforeInteraction—you have to tailor this method to setup the runtime context, connection manager, coordinator, RAS service, and logon information.
- ▶ afterInteraction—this method commits the interaction.
- ▶ executeInteraction—calls beforeInteraction, executes the command, and calls afterInteraction.
- ▶ initializeConnectionSpec—this method sets the tailored values into the connection specification from the properties of the session bean.

The properties of the session bean contain the tailored values to initialize the connection specification (Figure 9-21).

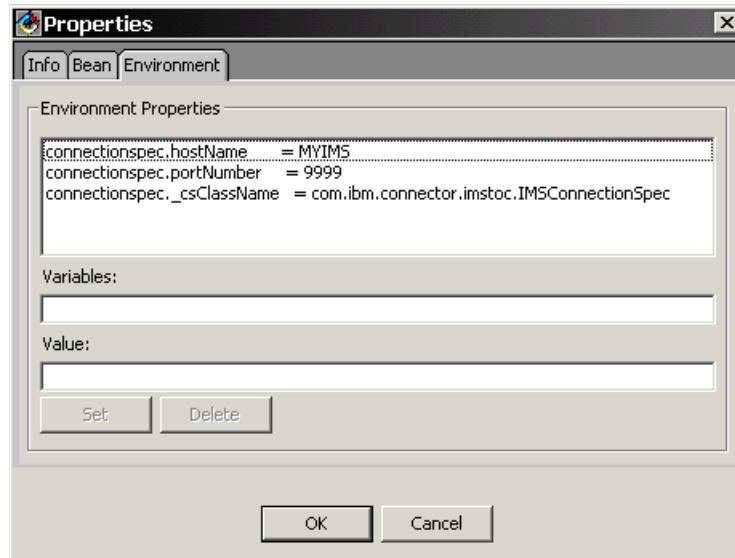


Figure 9-21 EAB session bean properties

An EAB session bean can be tested and deployed as any other session bean.

## Summary

As we have seen in this chapter, stateful and stateless session beans retain a different relationship with clients. A stateful bean is associated only to one client during its entire life.

This tight relationship makes stateful beans very useful to clients but causes a lot of problems to the container. If we assume that an application built with stateful beans has thousands of clients, each client must have its own dedicated bean instance. While there is no limit on the client population, the situation looks a bit different on the server side.

A server has a limited number of available resources, such as memory and database connections. As each bean instance requires a certain amount of system resources, the server can easily run out of resources. This is a known limitation with stateful beans in contrast to stateless beans.

Therefore, using stateful beans has certain impact on performance and scalability.

On the contrary, a stateless bean serves many clients during its lifetime. Because stateless session beans hold no conversational state on behalf of a client, they are all identical to a client. This provides the container with the capability to create a pool of bean instances and efficiently serve hundreds of clients.

All these make stateless beans faster and lightweight, resulting in very good performance and scalability, especially in clustered environments.

A good alternative to using stateful session beans is to use stateless beans, where servlets hold the conversational state with a client in the HttpSession object. In that case, the size of the conversational data that is stored in the HttpSession object should not be very big for performance reasons.

However, stateless session beans cannot replace stateful beans, when a transaction is required to span across multiple methods.

Session beans are mainly used to encapsulate business logic or access to databases and enterprise systems. Session beans can provide performance benefits compared to entity beans when searches are involved and the number of qualifying entity beans is either large or only used for further selection.





# Custom finder methods

This chapter provides in-depth information on how to write custom finder methods using VisualAge for Java. The finder support discussed here is only relevant to CMP beans, because if you are writing BMP beans, you will have to write everything yourself.

We describe the different types of finder methods with examples of implementations.

We also touch on the subject of finder methods in regard to transactions; depending on circumstances, the results of finder methods are instantiated in greedy or lazy mode.

## What are custom finders methods?

Finders are methods designed to help you retrieve entity beans from the persistent store. A finder method can retrieve one or multiple entity beans. For example, `findByPrimaryKey` (a generated method) retrieves one entity bean, and `findAll` (a custom finder method) retrieves multiple entity beans.

If you are dealing with BMP entity beans, you have to implement the finder methods in the bean class.

On the other hand, CMP entity beans custom finder methods are generated at deployment time using the container tools. These tools generate the implementation of the custom finder methods in the bean persister class.

## How to write custom finder methods?

There are two different paths to create custom finder methods:

- ▶ The `BeanFinderHelper` interface (or where custom finders)
- ▶ The `BeanFinderObject` class (or method customer finders)

Let's first examine the `BeanFinderHelper` interface and then the `BeanFinderObject` class.

### BeanFinderHelper interface

This interface, generated when you create a CMP entity bean, is meant to help you easily create simple finder methods. For example, when you define the **BankAccount** enterprise bean, a `BankAccountBeanFinderHelper` interface is generated. In the following section, we present the various usages of the interface.

### How to you use this interface?

The interface enables you to define either where clauses or query strings. Where clauses are the recommended way, whereas query strings are only for compatibility with earlier versions of VisualAge for Java.

### Using where clauses

Define a string constant (remember that variables defined in interfaces are always `static final`) for each custom finder where clause:

```

public interface BankAccountBeanFinderHelper {
 public String findAccountsWithBalanceGreaterWhereClause =
 "T1.balance > ?";
 public String findAllWhereClause = "1 = 1";
 public String findAccountsWithBalanceBetweenWhereClause =
 "T1.balance >= ? AND T1.balance <= ?";
}

```

The table is represented by an alias name, *T1* in our example:

- ▶ This alias should match the alias using in the persister class instance variable `findByPrimaryKeySqlString`. The alias for the table will be the same from one generation to the next unless tables are added to or removed from the mapping. In single-table cases, this may not seem significant (the alias is always *T1*). When multiple tables are used, this is very important.
- ▶ There is one very important restriction when using this form of finder in a mapped hierarchy: The WHERE clause can only reference tables that map the bean in which the finder is defined or tables that map one of the bean's parent beans (inheritance is discussed in Chapter 16. “Inheritance” on page 329).

Define the matching methods in the home interface. These methods must have a parameter for each `?` in the query string, where the number of parameters correspond exactly to the number of injection points (`?`) in the query:

```

import java.util.Enumeration;
import java.math.BigDecimal;
public interface BankAccountHome extends javax.ejb.EJBHome {
 public Enumeration findAccountsWithBalanceGreater(BigDecimal anAmount)
 throws javax.ejb.FinderException, java.rmi.RemoteException;
 public Enumeration findAll()
 throws javax.ejb.FinderException, java.rmi.RemoteException;
 public Enumeration findAccountsWithBalanceBetween(BigDecimal anAmount1,
 BigDecimal anAmount2)
 throws javax.ejb.FinderException, java.rmi.RemoteException;
}

```

## Using query strings

Instead of a where clause you can define a complete SQL select statement as a query string. The table is represented by an alias name (*T1*), and the sequence of column names must match exactly the sequence generated in the `findByPrimaryKeySqlString` string constant.

```

public interface BankAccountBeanFinderHelper {
 public String findAccountsWithBalanceGreaterThanQueryString =
 "select T1.accid, T1.balance from ITSO.Account T1
 where T1.balance > ?";
}

```

Define a matching `findAccountsWithBalanceGreaterThan` method in the home interface. This method must have a parameter for each ? in the query string:

```
public Enumeration findAccountsWithBalanceGreaterThan(BigDecimal anAmount)
 throws javax.ejb.FinderException, java.rmi.RemoteException;
```

**Attention:** This format of this custom finder is provided for backward compatibility only and should not be used any more. Use the where clause approach instead.

## What code is generated?

When the deployed code is generated, code for the finders is generated using the finder method signatures in the home and the strings (where clauses) defined in the finder helper interface.

The `EJSJDBCVerifierBean` class will contain code to support the custom query as shown in Figure 10-1.

```
public class EJSJDBCVerifierBankAccountBean {
 public EJSFinder findAccountsWithBalanceGreater(java.math.BigDecimal arg1)
 throws java.rmi.RemoteException, javax.ejb.FinderException {
 ResultSet rs = null;
 PreparedStatement ps = null;
 EJSJDBCVerifier result = null;
 try {
 preFind();
 ps = getMergedPreparedStatement(itso.ejb35.cmp.BankAccountBeanVerifierHelper.
 findAccountsWithBalanceGreaterWhereClause);
 for (int i=0; i<getMergedWhereCount(); i++) {
 // Injecting "java.math.BigDecimal arg1"
 if (arg1 == null) ps.setNull(1, 3);
 else ps.setBigDecimal(i+1, arg1);
 }
 rs = ps.executeQuery();
 result = new EJSJDBCVerifier(rs, this, ps);
 } catch (Exception exc) {
 try { if (rs != null) rs.close();
 } catch (SQLException sqlExc) {}
 if (ps != null) super.returnPreparedStatement(ps);
 throw new EJSPersistenceException("findAccountsWithBalanceGreater failed:",
 exc);
 }
 return result;
 }
}
```

Figure 10-1 Generated custom finder method

Here are a couple of comments about the generated code:

- ▶ The return type EJSFinder is an interface that extends Enumeration. It provides a specialized API for entity beans. The result of the query is passed to a EJSJDBCFinder class which wraps it in an enumeration of entity beans.
- ▶ The corresponding query string is retrieved from the finder helper interface and passed on to getMergedPreparedStatement to create a prepared statement.
- ▶ The method getMergedWhereCount calculates how many injections (parameters) are required.

## Limitations

Both of the string specifications have limits:

- ▶ Queries must stay simple and have straightforward parameter injections (parameters that are compatible with the column types).
- ▶ If the schema and/or map are changed, the query might have to be changed to reflect the actual configuration.
- ▶ If you use EJB inheritance, the select query strings will not work with the inherited types and the where query strings still have to be simple.

To overcome these limitations you can also write custom finders using the BeanFinderObject class.

## BeanFinderObject class

In order to be able to use any kind of parameters in the signature of the finder methods, you can write the custom finder methods yourself. Note that the example presented here would not require an implementation using the BeanFinderObject class.

First you define the signature of the finder method in the finder helper interface, for example:

```
public interface BankAccountBeanFinderHelper {
 public java.sql.PreparedStatement findGoldAccounts
 (java.math.BigDecimal balance) throws Exception;
}
```

Second, define a matching method (for example, findGoldAccounts) in the home interface. This method must have a parameters matching the custom method signature.

```
public Enumeration findGoldAccounts(java.math.BigDecimal balance)
 throws java.rmi.RemoteException, javax.ejb.FinderException;
```

Third, you have to define and implement a class with the custom finder methods:

- ▶ The class must extend `com.ibm.vap.finders.VapEJSJDBCFinderObject`.
- ▶ The class must implement the `BeanFinderHelper` interface.

The name of the class should be either:

- ▶ `BeannnameBeanFinderObject` and reside in the same package as the bean class (recommended, for example, `BankAccountBeanFinderObject`):

```
public class BankAccountBeanFinderObject
 extends com.ibm.vap.finders.VapEJSJDBCFinderObject
 implements BankAccountBeanFinderHelper { ... }
```

- ▶ Any name, but the fully qualified class name, must be specified in the environment properties of the deployment descriptor using the variable name `CustomFinderClassName`.

**Note:** When you create the `BeannnameBeanFinderObject` within the EJB page, you will not find the class in that page anymore. You have to switch to the Project page to find it in the package of the enterprise bean.

## Implementing the class

Because the class implements the finder helper interface, you have to provide an implementation of each method in the interface.

There is mainly one thing to be careful about: any table reference (or alias) in the handwritten SQL must match the table aliases defined in the `findByPrimaryKeySqlString` instance variable of the persister class.

To ease writing an implementation class that fully uses the CMP layer of the container, the superclass `VapEJSJDBCFinderObject` provides a number of useful methods (Table 10-1).

Table 10-1 *VapEJSJDBCFinderObject utility methods*

| Method Name                             | Description                                                                                                                                                                           |
|-----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>getMergedPreparedStatement</code> | Takes a WHERE clause and returns a PreparedStatement with the WHERE clause merged into the appropriate places. The prepared statement will have the correct result set shape.         |
| <code>getMergedWhereCount</code>        | Returns the number of times the WHERE clause is merged into the prepared statement. This is needed to know how many times to inject the query parameters into the prepared statement. |

| Method Name                | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| getPreparedStatement       | Takes a complete query string and returns a PreparedStatement. This can be used if for some reason you have to do your own WHERE clause merging. This should be a very rare case. The next two methods are provided to help in these extreme cases.                                                                                                                                                                                                         |
| getGenericFindSqlString    | Returns the query string into which WHERE clauses are merged.                                                                                                                                                                                                                                                                                                                                                                                               |
| getGenericFindInsertPoints | Returns an array of integers that defines the point or points in the getGenericFindSqlString returned query string at which the WHERE clause is merged. The first point in the array is the last point in the string. It is usually best to merge from the end of the query string since a merge at the end will not change the location of merges earlier in the string. The size of the array is the same as the value returned from getMergedWhereCount. |

Try to use only the first two methods, because the other three methods are for complex cases.

Now let's look at the implementation of the finder method that you have to write (Figure 10-2).

```
public java.sql.PreparedStatement findGoldAccounts
 (java.math.BigDecimal aBalance) throws Exception {
 java.sql.PreparedStatement pstmt = null;
 int mergedCount = getMergedWhereCount();
 int columnCount = 1; // number of ? in the original query
 pstmt = getMergedPreparedStatement("T1.balance >= ?");
 for (int i=0; i < (columnCount* mergedCount); i+=columnCount)
 pstmt.setBigDecimal(i+1, aBalance);
 return pstmt;
}
```

Figure 10-2 Custom finder method implementation

This is a rather simple example, but it is already prepared to handle subclasses of BankAccount, because we have used the getMergedPreparedStatement method, which returns a query that can handle multiple tables and unions of multiple subselects. The parameters will be injected into each merged where clause.

## What code is generated?

The container tools are generating code to use the custom finder method you have specified in the BeanFinderHelper interface and implemented in the BeanFinderObject. One method per finder will be added at generation time in the persister object.

The generated finder method is quite simple; it retrieves the finder object and executes the corresponding finder method on it.

```
public EJSFinder findGoldAccounts(java.math.BigDecimal arg1)
 throws java.rmi.RemoteException, javax.ejb.FinderException {
 ResultSet rs = null;
 PreparedStatement ps = null;
 EJSJDBCFinder result = null;
 try {
 preFind();
 ps = getFinderObject().findGoldAccounts(arg1);
 rs = ps.executeQuery();
 result = new EJSJDBCFinder(rs, this, ps);
 } catch (Exception exc) {
 try {
 if (rs != null) rs.close();
 } catch (SQLException sqlExc) {}
 if (ps != null) super.returnPreparedStatement(ps);
 throw new EJSPersistenceException("findGoldAccounts failed:", exc);
 }
 return result;
}
```

## When to use which custom finder?

After having described the two options of custom finder, here are the recommendations:

- ▶ You should use the where clause custom finder for simple and straightforward queries with easy parameter injection.
- ▶ You should implement any other query with method custom finder code.
- ▶ You should not use the select custom finder, because they are not recommended for new development.

## Mixing the approaches

You can define both where clause strings and custom method signatures in the BeanFinderHelper interface. For both cases you have to add the finder method signatures to the home interface.

For the where clause string the method implementation is generated for you in the `EJSJDBCVerifierBean` and for the custom method signatures you write the implementation yourself in the `BeanFinderObject` class.

## Alternative to custom finders

Though it is perfectly possible to implement all queries using the methods described above, you should sometimes forget about them and be creative. Let's illustrate this by an example.

You have a product database with millions of items. Each item is categorized and a user can retrieve the items for a specified category using a Web application. If you provide a query that handles this query using EJBs, you will discover that the average response time is not so great.

Why is that? Think for a minute and assume that the query is returning, let's say, a couple of thousands hits. The container has to "hydrate" the results into EJBs whose references are then placed in a collection that your servlet is iterating through.

Instead, a better design would be to implement a simple JDBC query in a session bean to retrieve only the subset of the data that you have to display for the end user to select the requested items. EJBs will then only be instantiated for the items selected by the end user. Using this approach, you avoid:

- ▶ Creating lots of objects that will only be used to read data
- ▶ Slowing down the server and annoying all the other users
- ▶ Possibly crashing the server with out of memory errors

An example of a session bean with JDBC database access is provided in "Using session beans for database queries" on page 180.

## Testing finder methods

After defining the finder methods in the finder helper interface and the home interface, and possibly implementing the custom methods in the `BeanFinderObject` class, you have to (re)generate the deployed code.

Then you can test the finder methods in the usual way:

- ▶ Start the persistent name server
- ▶ Add the EJB group to the server configuration and start the EJB Server
- ▶ Start the EJB test client
- ▶ Lookup the home

A sample test client run with all the customer finder methods is shown in Figure 10-3.

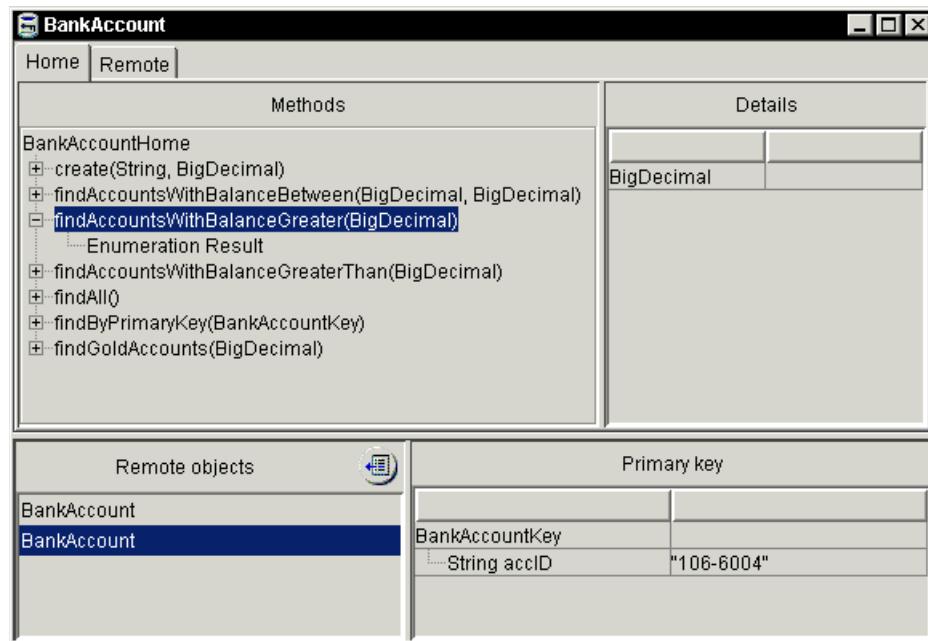


Figure 10-3 EJB test client with custom finder methods

When you execute a finder method, you will usually retrieve multiple remote objects because most finders retrieve an Enumeration.

## Finder methods and transactions

In the WebSphere execution environment, the execution of a finder method returns an enumeration. It can be a greedy or a lazy enumeration, depending on the caller's transaction context.

### Greedy enumeration

A finder will operate in *greedy* mode when called with no active transaction.

In greedy mode, the resulting enumeration will be fully formed at the time the finder method returns. However, the elements are not yet activated. This enumeration may be passed around at will and enumerated at any point in time. There are no guarantees that all members of the enumeration will still exist as some may have been deleted.

## Lazy enumeration

In contrast, a finder will operate in *lazy* mode when there is an active transaction at the time the finder method is invoked. This will typically be the case, for example, if you have a session bean with its transaction attribute set to TX\_REQUIRED that is invoking a finder.

In lazy mode, the resulting enumeration is not fully formed when the finder method returns. The JDBC result set remains open on the server; the enumeration received by the caller will fetch members of the result set in batches from the server as you step through the enumeration.

A lazy enumeration is valid only until the transaction in which the finder was invoked commits. Attempts to use the enumeration after the transaction has committed will result in an IllegalStateException. If you get such an exception you have to start a transaction earlier and keep it alive over the enumeration of the finder method result.

## Finders and complex associations

For a many-to-many (m:m) association an intermediate entity bean is required. Therefore, a client application that follows an m:m association has to deal with this intermediate bean that has no data associated with it.

To relieve the client from accessing the intermediate bean, custom finder methods may be implemented to return the target entity beans.

The following example involves an m:m association between Customer and BankAccount beans, using a intermediary bean (CustAcct) and two 1:m associations (Figure 10-4).



Figure 10-4 Customer - BankAccount m:m association

We can write custom finder methods to hide the intermediate bean in either direction with just one method call. Not only do we hide the intermediate bean, we also do not instantiate EJB objects for the intermediate bean. Using a custom finder we execute a tailored select statement with a table join to retrieve the target entity beans.

For this example, we write a finder method in BankAccount that retrieves all accounts that are associated with a given customer ID.

The BankAccountHome interface contains the appropriate method signature, as follows:

```
public java.util.Enumeration findAccountsForCustomer(String aCustomerId)
 throws RemoteException, FinderException;
```

The BankAccountBeanFinderHelper interface contains the signature for the corresponding finder method:

```
public PreparedStatement findAccountsForCustomer(String aCustomerId)
 throws Exception;
```

The finder object (BankAccountBeanFinderObject) builds and caches the query string for the finder as well as implements the finder method.

```
public class BankAccountBeanFinderObject extends VapEJSJDBCFinderObject
 implements BankAccountBeanFinderHelper {
 private String cachedFindAccountsForCustomerQueryString = null;
 ...
}
```

The implementation of this finder method is described in “Finder method for m:m association” on page 365.

## Summary

Finder methods provide the functionality of finding entity beans by criteria other than by the primary key. In most applications you have to provide different ways to find the data for a client. Finder methods are one way for such queries.

However, be careful with finder methods, because they instantiate the resulting entity beans. If a finder method returns hundreds (or thousands) of results, all these entity beans will be instantiated, probably for no good cause, because the application only requires a subset of the result set.

Always consider alternatives to custom finder methods, such as session beans with direct JDBC access to the database (see “Using session beans for database queries” on page 180).



# Access beans

In this chapter we introduce access beans as a facility to simplify client programming when accessing enterprise beans.

Accessing an enterprise bean requires the use of the initial naming context and basic knowledge about the way the home and remote interfaces are used. The developer also has to provide important parameters, such as JNDI names and PROVIDER\_URL that are not so well-known outside the EJB world.

Considering all these, we need another way to write EJB clients; a way that enables developers with no EJB knowledge to write servlets and JavaServer pages that can access enterprise beans and display their properties. The alternative to this is access beans.

# What are access beans?

Access beans are Java components that adhere to the JavaBeans specification and are meant to simplify the development of EJB clients. An access bean adapts an enterprise bean to the JavaBeans programming model by hiding the home and remote interfaces from the developer. They provide fast access to enterprise beans by letting you maintain a local cache of enterprise bean attributes. Access beans make it possible to use an enterprise bean in much the same way that you would use a JavaBean.

VisualAge for Java provides a SmartGuide for creating or editing access beans. Access beans are designed to simplify the development of the EJB clients. Client programming becomes as simple as:

- ▶ Invoke an access bean constructor
- ▶ Set the initial arguments required for creating or finding an enterprise bean
- ▶ Invoke the business methods on the access bean

Figure 11-1 summarizes how access beans are used. In this diagram we show the Customer enterprise bean, enhanced with a custom finder method (findByLastName).

## CustomerAccessBean

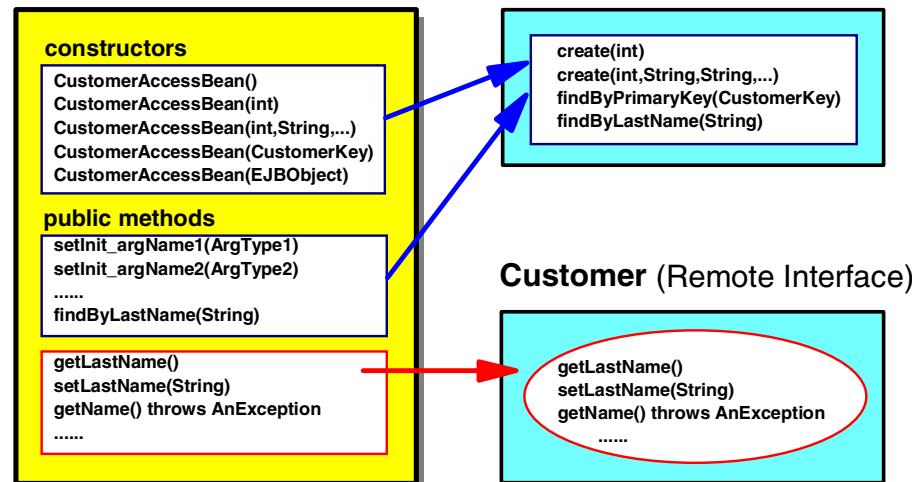


Figure 11-1 Access bean characteristics

Let's describe the general characteristics of access beans:

- ▶ Create methods of the home interface become constructors of the access bean.
- ▶ The `findByPrimaryKey` method of the home interface becomes a constructor of the access bean; either the default constructor or a constructor with the key class as argument.
- ▶ The default constructor (no arguments) of the access bean maps either to any of the create methods or to a finder method that retrieves one instance, for example, `findByPrimaryKey`. This is a choice when generating the access bean with the SmartGuide provided by VisualAge for Java. After instantiating the access bean you must set the arguments used in the home method by invoking the special `setInit_argument` methods. The arguments are stored as instance variables in the access bean.
- ▶ Finder methods in the home interface are mapped to a finder method in the access bean. You must first instantiate the access bean and then invoke the appropriate finder method, which returns either a collection of access bean instances or a single access bean.
- ▶ Remote interface methods are mapped to methods in the access bean.
- ▶ To instantiate an enterprise bean, the access bean invokes a create or finder method defined in the enterprise bean home interface. If a no-argument constructor is used, the access bean only instantiates the actual enterprise bean when the first business method is called. It uses the arguments set through the `setInit_argument` methods to invoke the create or find method.
- ▶ Access beans may employ copy helper objects that are basically caches of user-selected entity bean attributes that are stored inside the access bean. The getter and setter methods for these attributes deal directly with the local cache rather than calling straight through to the remote getter and setter call. Methods are provided to flush the cache to the actual enterprise bean database and to refresh the cache from the actual enterprise bean. This can improve performance significantly for entity enterprise beans that have a large number of attributes, where issuing one remote call to get and set a large number of attributes is faster than issuing a single remote call for each attribute.
- ▶ Access beans are subclasses of the `AbstractAccessBean`. This class stores the default constants for JNDI access to retrieve the home classes:

```
public static final String DEFAULT_NAMESERVICE_TYPE =
 "com.ibm.ejs.ns.jndi.CNInitialContextFactory";
public static final String DEFAULT_NAMESERVICE_PROVIDER_URL = "iiop:///";
private static final String NAMESERVICE_TYPE_PROPERTY =
 "java.naming.factory.initial";
private static final String NAMESERVICE_PROVIDER_URL_PROPERTY =
 "java.naming.provider.url";
```

- ▶ The access bean contains its JNDI name:
 

```
protected String defaultJNDIName() { return "itso/ejb35/cmp/Customer"; }
```
- ▶ Methods are provided to change the JNDI service and provider classes and the beans JNDI name:
 

```
public void setInit_NameServiceTypeName(String name)
public void setInit_NameServiceURLName(String name)
public void setInit_JNDIName(String name)
```
- ▶ The getHome method is used by the access bean to retrieve the home instance at the first usage of the enterprise bean. The home instance is stored in a private variable of the AbstractAccessBean:
 

```
transient private Object myHome;
```
- ▶ Multiple instances of an access bean use the same home. The access bean retains a class-level cache (globalHome variable) to improve performance when instantiating enterprise beans.

## Access bean types

There are three types of access beans, which are listed in ascending order of complexity:

- ▶ JavaBean wrapper (for a session or entity bean)
- ▶ Copy helper (for an entity bean)
- ▶ Rowset (for multiple entity bean instances)

## JavaBean wrapper

Of the three types of access beans, a JavaBean wrapper is the simplest to use. It is designed to allow either a session or entity enterprise bean to be used like a standard JavaBean and it hides the enterprise bean home and remote interfaces from the developer. Each JavaBean wrapper that you create extends either `AbstractEntityAccessBean` or `AbstractSessionAccessBean`.

A JavaBean wrapper access bean has the following characteristics:

- ▶ It contains a no-argument constructor.
- ▶ When the SmartGuide prompts you to map one of the create or finder methods defined in the home interface to the no-argument constructor of the access bean, the access bean subsequently contains one `init_xxx` property for each parameter (for example, `private int init_argCustomerID`).

To simplify JSP coding, you can choose to have your access beans expose the `init_xxx` properties as `String` types. However, you can also select your own converters for the `init_xxx` properties.

- ▶ When a key class is used in the create and finder methods for a CMP entity bean, the key fields are used as the `init_xxx` properties instead of the key class. A key field is normally declared as a simple type. This makes it easier for visual construction tools, such as the Visual Composition Editor, to use an access bean.
- ▶ When the no-argument constructor is used, the `init_xxx` properties must be set first before any other calls to the access bean.
- ▶ The access bean may contain several multiple-argument constructors, each corresponding to one of the create or finder methods defined in the bean home interface.
- ▶ The access bean performs lazy initialization when the no-argument constructor is used. When the access bean is instantiated, it does not instantiate the enterprise bean. On the first remote method call, the access bean instantiates the remote enterprise bean.
- ▶ A default JNDI name is generated into each access bean class. The code generator reads the deployment descriptor and passes the JNDI name to the access bean. You can change the JNDI name using the `setInit_JNDIName` method. It is not expected that you will have to change the JNDI name. However, in the event that an enterprise bean is deployed into a different home, the administrator may add a prefix to the JNDI name to indicate the different home.
- ▶ To look up a home, an access bean has to obtain a name service context. This context can be constructed if you know the name service URL and the name service type. Access beans provide two APIs that allow you to define a customized context, `setInit_NameServiceTypeName` and `setInit_NameServiceURLName`.
- ▶ An enterprise bean remote interface method can return an enterprise bean object. When this kind of method is generated in the access bean class, the return type is changed to the corresponding access bean type. This allows your user program to deal with only the access bean type and inherit the benefits provided by the access bean.

## Copy helper

A copy helper access bean has all of the characteristics of a JavaBean wrapper, but it also incorporates a single copy helper object that maintains a local copy of attributes from a remote entity bean. A user program can retrieve the entity bean attributes from the local copy helper object that resides in the access bean, which eliminates the need to access the attributes from the remote entity bean.

When you create a copy helper access bean, the remote interface of the enterprise bean is changed to extend the `CopyHelper` interface as well as the

EJBObject interface. You can select all of these attributes or only a subset in creating the copy helper object. The selected attributes are saved in the enterprise bean meta model. These selections are re-displayed if you decide you want to change the selection.

The copy helper object is stored inside the access bean. A get and set method is delegated to the local copy helper object instead of the remote enterprise bean object. The local copy of bean properties can be synchronized with the remote instance properties by using the following two methods.

- ▶ The refreshCopyHelper method refreshes the local copy helper data from the remote enterprise bean.
- ▶ The commitCopyHelper method commits the changes in the local copy helper data to the remote enterprise bean. Notice that this method is smart enough to update only the modified attributes in the bean instance.

Notice that when you create a copy helper access bean, your enterprise bean class is modified by VisualAge for Java. A copy helper interface with two methods defined is added to the corresponding class and promoted to the remote interface. These two methods are:

```
_copyToEJB
_copyFromEJB
```

**Note:** After adding a copy helper access bean to an entity bean, you need to regenerate the deployed code, because of the changes made to the bean.

## Rowset

A rowset access bean has all of the characteristics of both the JavaBean wrapper and copy helper access beans. However, instead of a single copy helper object, it contains multiple copy helper objects. Each copy helper object corresponds to a single enterprise bean instance.

When generating a rowset access bean, you actually get a copy helper access bean (for example, CustomerAccessBean) and a rowset access bean (for example, CustomerAccessBeanTable) that holds a collection of copy helper access beans.

A rowset access bean contains a collection of copy helper objects. In turn, a copy helper object contains the primary key for each entity bean instance, but it does not contain the proxy object (the EJB object in the EJB server) itself for the entity bean.

When a bean finder method returns multiple entity beans you can use a rowset access bean to store the result set. Only the attributes of the entity beans are copied to the client space. The proxy objects are not copied. This is because copying a large number of enterprise bean proxy objects from the server space to the client space can cause performance problems. A JSP can read from a rowset access bean immediately, without invoking a remote call. On an update call, for example, using the `commitCopyHelper` method, the access bean constructs the enterprise bean proxy object using the key object saved in the copy helper.

The main methods generated into the customer rowset access bean are:

- ▶ `setCustomerAccessBean(Enumeration)`—fill the rowset access bean with the results of a finder method.
- ▶ `getCustomerAccessBean(int)`—retrieve a copy helper access bean from the collection stored in the rowset; this method returns a `CustomerAccessBean`.
- ▶ `getCustomerAccessBean()`—retrieve all copy helper access bean from the collection stored in the rowset; this method returns an `Enumeration`.
- ▶ `setCustomerAccessBean(int,CustomerAccessBean)`—replace one access bean in the collection.
- ▶ `numberOfRows()`—number of access beans in the collection; this method is inherited from the `AbstractAccessBeanTable` superclass.

## Access beans and associations

Associations are described in Chapter 17, “Associations” on page 353.

If you create an access bean for an enterprise bean involved in an association and the association has been made navigable, the navigation method of the access bean returns an access bean corresponding to the enterprise bean at the other side of the association.

Therefore, you have to generate access beans for all enterprise beans related through associations, otherwise you encounter errors when following associations through access bean methods.

## Creating an access bean with VisualAge for Java

For this example we create a rowset access bean (and therefore also a copy helper) using the SmartGuide provided with VisualAge for Java:

1. In the EJB page, select the **Customer** bean and *EJB -> Add -> Access Bean*.
2. The Create Access Bean SmartGuide opens (Figure 11-2).

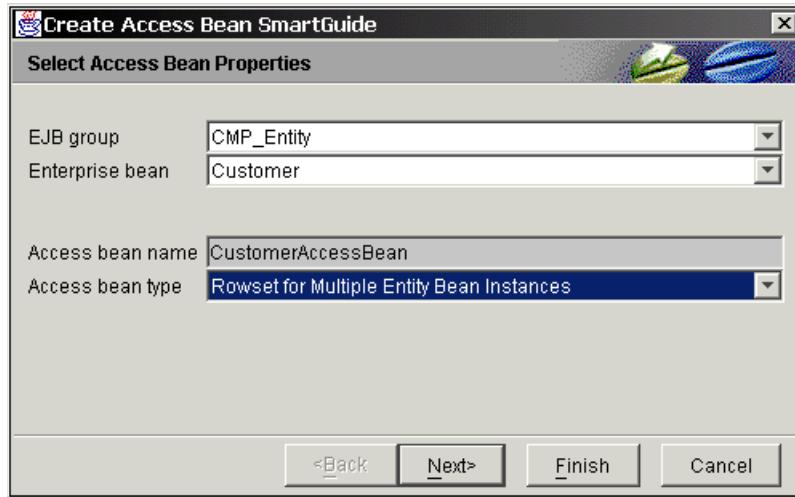


Figure 11-2 Creating an access bean

3. Ensure that the correct enterprise bean is selected.
4. The name of the access bean (*CustomerAccessBean*) cannot be changed.
5. In the *Access bean type* field, select *Rowset for Multiple Entity Bean Instances*.
6. Click *Next*. The Define Zero Argument Constructor page opens. Here you specify the home interface method to correspond to the zero argument constructor of the access bean (Figure 11-3):
  - You can select between any of the create methods, the *findByPrimaryKey* method, or a finder method that returns a single instance (not an *Enumeration*).
  - Depending on your selection, the initial properties are displayed; one for each argument of the selected method.

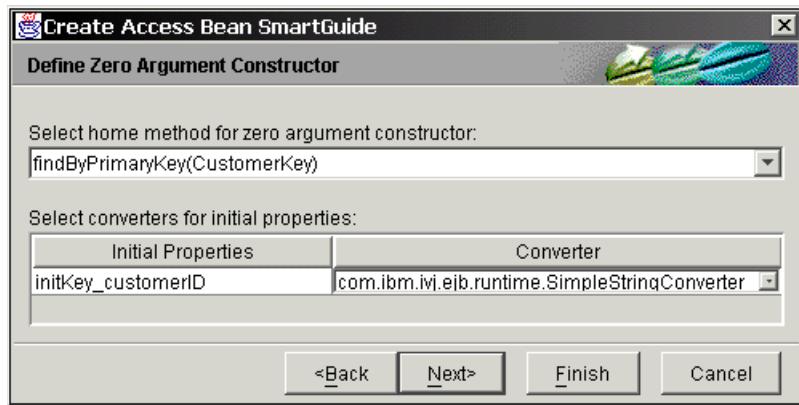


Figure 11-3 Zero argument constructor selection

7. For our example, select the *findByPrimaryKey(CustomerKey)* method; we want the zero argument constructor to find an existing instance. The *initKey\_customerID* is displayed in the Initial Properties pane.

When access beans are used in the Visual Composition Editor or in JSPs, you can convert the arguments in the home interface method to strings for simplifying visual development or JSP code. Select the default string converter for the customerID key field. Note that this string converter only supports the Java primitive types and their object wrappers.

8. Click *Next*. The Select and Customize Bean Properties for Copy Helper page appears (Figure 11-4).

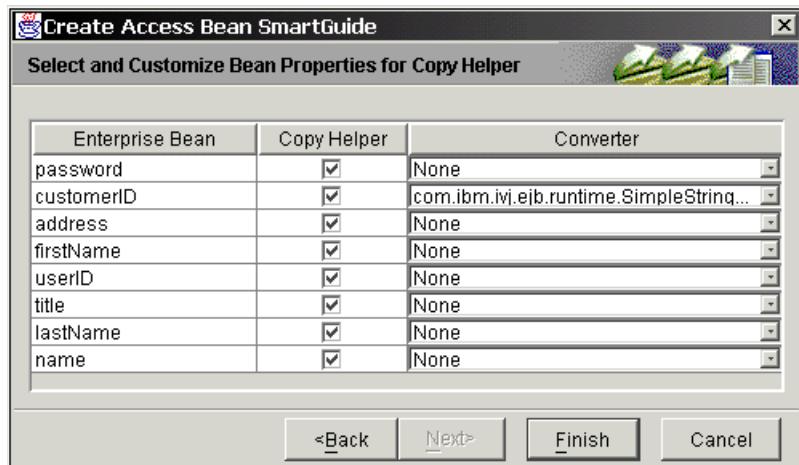


Figure 11-4 Copy helper field selection and conversion

9. Select the bean properties for which you would like to have copy helper support. For our example select all the CustomerBean properties. Also select the string converter for the customerID property. Notice that the tailored getName method is listed as a name property of the customer bean.
10. Click *Finish* to generate the access bean. After the copy helper is generated, you have to re-generate the deployed code for the customer bean, because VisualAge for Java has modified the bean class and the remote interface (see “Copy helper” on page 209).

**Tip:** You can use the same SmartGuide to modify an access bean you have created.

## Using access beans in client programs

We describe the usage of access beans in client programs in “Using access beans” on page 254.

Here is an extract of typical code to use a copy helper access bean to retrieve and update an entity bean:

```
CustomerAccessBean cab = new CustomerAccessBean();
cab.setInitKey_customerID("101"); // prepare key value
String password = cab.getPassword(); // lazy initialization performed
password = password + "X";
cab.setPassword(password); // change password in copy helper
cab.commitCopyHelper(); // commit changes to entity bean
```

**Note:** If we wanted to use the copy helper for creating a new customer instead, we would have to use another constructor and set all the initial properties of the customer:

```
new CustomerAccessBean(customerID, title, firstName, lastName, uid, pwd);
```

However, this would be different, if we had created a copy helper with a zero-argument constructor calling the create method on the home interface, instead of the findByPrimaryKey, as we did in our example. After instantiating the CustomerAccessBean, we would then have to set all the initial properties for the customer one by one using multiple setInit\_xxxx methods.

Always after setting new values to the properties on the copy helper, you have to call the methods to refresh and commit the copy helper cache:

```
commitCopyHelper and refreshCopyHelper
```

## Summary

Now that we have described access beans, the question is when should we use access beans?

Access beans have been designed to improve performance when enterprise beans and their clients are separated by a network. They also provide to the client a caching mechanism for accessing homes.

We can use access beans in servlets and JSPs. Especially in JSPs access beans can simplify coding because of the caching of properties and automatic conversion of values to strings.

We can use access beans between session and entity beans. If entity beans are running in different clones, then using access beans may still improve the performance.





# Transaction management

In this chapter we describe how transactions are defined in the context of J2EE and how transactions are managed around enterprise beans.

We describe the transaction attributes and isolation levels that can be set for each enterprise bean, and the effect they have on commit, rollback, concurrency, and locking.

We then describe ways of starting and ending transactions by enterprise beans, the container, and client applications.

Finally, we provide guidelines for transaction management.

# What is a transaction?

In this section, we describe the basics of transactions, study their four main characteristics, and provide a real-life example.

## Overview

The concept of transaction is not a new one. It has existed since human beings started trading goods with each other, even before money was used. Today the main principles remain basically the same. In business, a transaction involves a commercial exchange and is usually done between two parties. For example, when you go and buy a book from a bookstore, you give first money and then you get the book in return.

In the computer world, a transaction is the execution of a *unit-of-work* that accesses one or more shared resources and does the actual exchange. A unit-of-work is a set of activities that relate to each other and must be completed together. If any of these activities fail, the entire unit-of-work must be undone. This can be better understood by using an example.

Let's say that we want to transfer money from one account to the other, as we do in the `transferMoney` method of the transfer session bean. If you remember, the activities that must be performed are:

1. Withdraw the money from one account.
2. Create a transaction record for that account.
3. Deposit the money to another account.
4. Create a transaction record for that account.

Just a reminder here for the transaction record use and importance. In banks, it is very common that the balance of an account is verified based on all the recorded transactions. So, if there were a transaction that had not been recorded, then the account balance would not be consistent. It would be either higher or lower, depending on if the transaction was a withdraw or a deposit respectively.

Updating the balance for each account after withdrawing or depositing money is also important for the success of our operation. A money transfer from one account to another would not be complete, if any of these activities failed.

Let's suppose that somehow a money transfer is done, without actually withdrawing the money from the first account. In that case, the first account would have the same balance and the second account an increased one. So, money would not have come out from an account, but would have been

deposited to another account. This would have caused a money loss on the bank which would not be able to claim the money back from the second account.

Therefore, all these activities have to be completed together. Only then a successful money transfer is done. This action is known as *commit*. This is what makes the set of these activities a unit-of-work (Figure 12-1).

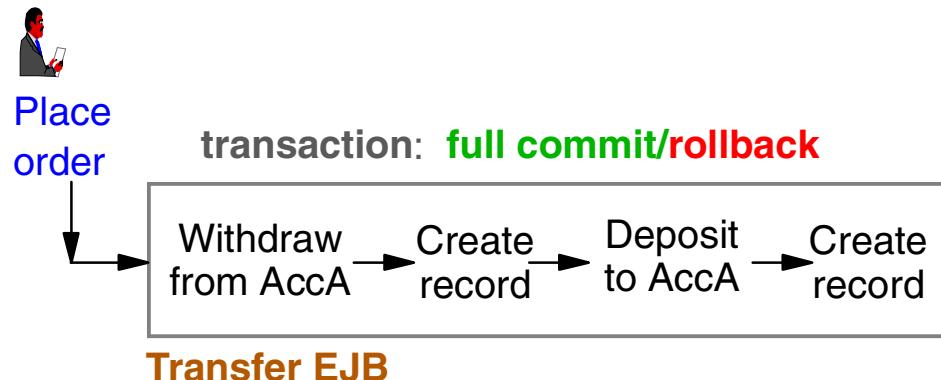


Figure 12-1 The transfer money transaction

But, what happens if any of these activities do not complete? This will cause all the other completed activities to be undone. This action is known as *rollback*.

It is not so unusual that problems like this occur in real life. Usually, systems that are running such business applications are pretty complex and distributed on several machines (multi-tier architecture). The probability of a network connection or a machine going down, while updating a data source is fairly likely. A similar problem can also occur to a database server. No system or network can be 100% available.

Apart from availability, there is also a concurrent data access aspect. Activities like reading or updating an account balance usually requires access to an underlying database where all the account data is stored. According to our implementation, the method that withdraws money from an account, checks first to see if the balance is not exceeded by the amount to transfer. What happens if during the transfer transaction, another application is attempting to reduce the balance of the first account, after the withdraw method has committed?

What guarantees that our applications and data are protected in such occasions is the four main properties of a transaction, the *ACID* properties. The word ACID stands for *Atomicity*, *Consistency*, *Isolation*, and *Durability*.

## The ACID properties

A transaction has the following four properties:

- |                   |                                                                                                                                                                                                                                                                                                                                           |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Atomic</b>     | A transaction must execute completely or not at all. Every activity in a unit-of-work must execute successfully. If any activity fails, the entire transaction is aborted and all the data changes are rolled back. If all activities execute without an error, the transaction completes and all data changes are committed.             |
| <b>Consistent</b> | A transaction must not leave a system inconsistent after it completes. Consistency refers to the integrity of the underlying data store. For example, in our money transfer method, we should not have a negative balance in an account, after the transfer is done. Notice that consistency should usually be enforced by the developer. |
| <b>Isolated</b>   | A transaction must be allowed to execute without interference from other processes or transactions. Any intermediate states are transparent to other transactions, allowing multiple transactions to execute serially.                                                                                                                    |
| <b>Durable</b>    | All data changes committed during a transaction must be written to a persistent data store and should survive hardware or software failures. If a failure occurs, the data can be recovered by using transactional logs.                                                                                                                  |

## Transaction support in J2EE

The transactions in J2EE are based on the Java Transaction Service (JTS) and Java Transaction API (JTA).

Let's first define a few concepts:

- |                             |                                                                                                                                                                                                                                                   |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Transactional object</b> | Component involved in a transaction. In our example the bank account is a transactional object.                                                                                                                                                   |
| <b>Transaction manager</b>  | Component or system responsible to manage the transactional operations of transactional components.                                                                                                                                               |
| <b>Resource</b>             | Any persistent store on which you can read or write, it could be a database or a message queue.                                                                                                                                                   |
| <b>Resource manager</b>     | Responsible for managing a resource. Typically resource managers are relational database or message queue products. Resource managers are very often based on the X/Open XA resource manager interface to allow interoperability between vendors. |

X/Open XA is a standard supported by most database vendors. It envisions the software components mentioned above in a distributed transactional system. This specification defines the interface between the transaction manager (or coordinator) and the local resource manager.

## Java Transaction Service

JTS defines a low level API that is meant to be used by the application server provider. JTS is the Java implementation of the OMG CORBA *Object Transaction Service* (OTS). This API is not meant to be used by EJB developers, because it is rather complex. EJB developers should rather use JTA.

## Object Transaction Service

OTS is an optional CORBA service that provides transaction support involving one or more parties. It has a number of interfaces that the transaction manager, the resource manager, and the transactional objects use to collaborate.

One of the major advantages of OTS is that it has transaction context propagation built-in. It allows to propagate transaction contexts on multiple servers as long as they are using RMI-IIOP; this feature is what makes distributed transaction possible.

## Java Transaction API

JTA is a simple transaction API that EJB developers can use to demarcate transactions on the client side or in the session beans. This API is defined in the `javax.transaction.UserTransaction` interface. It allows to programmatically create, commit, or rollback transactions.

Another interface provided by JTA is the interface used by X/Open XA resource managers; it enables the use of distribution transactions. This interface has to be implemented by JDBC drivers to enable them to participate in distributed transactions. When defining a `DataSource`, you can choose between a regular JDBC driver and a JTA-enabled driver (see Figure 6-8 on page 90).

## Transaction attributes

In “Developing session beans with VisualAge for Java” on page 174, we did not code any transaction logic in the transfer bean. However, we executed the `transferMoney` method in a transaction context.

Any enterprise Java bean that is deployed in a container enjoys the transaction services provided by the container. But how does the container know what the transaction requirements of a deployed bean are?

This is determined by attributes in the deployment descriptor. Let's see how these attributes define the transactional behavior of a bean and guide the container to manage the transaction on behalf of the bean.

The transaction attributes are declarative elements that tell the container when to start and commit a transaction. These attributes can be set either on the bean or the method level. An attribute set at the method level overrides the setting on the bean level, if any.

The bean-level attributes are set in the Properties dialog of an enterprise bean (Figure 12-2).

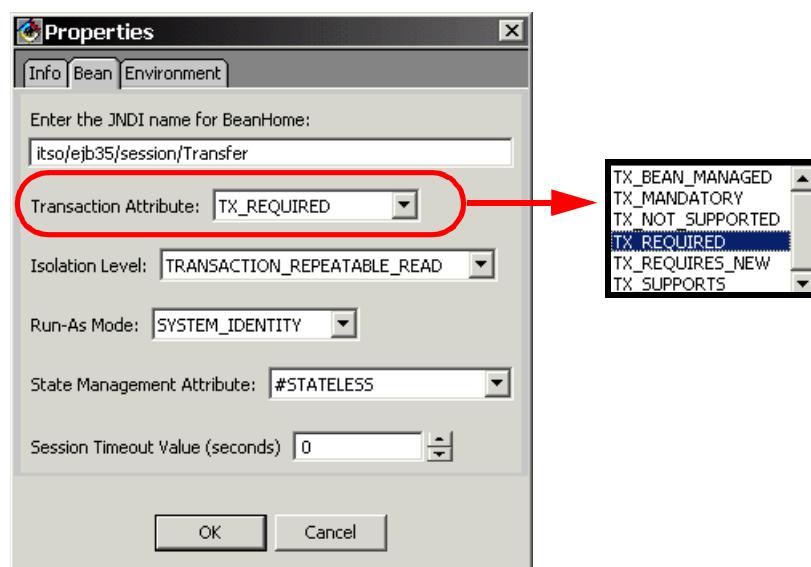


Figure 12-2 Bean properties for transaction management

The values that these attributes can have are:

|              |                                                                                                                                                                                                                                                                                      |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TX_MANDATORY | When this value is used, a transaction must be already running when the bean method is called. The object will participate in the existing transaction initiated by the caller. If no transaction context is present, the TransactionRequiredException is thrown back to the caller. |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TX_NOT_SUPPORTED | This value means that the bean or a method cannot be involved in a transaction at all. If a client has started a transaction, it is ignored, and methods will always execute outside the transaction context. The initial transaction will be resumed, after the end of those methods.                                                                                                                                                                                                                         |
| TX_REQUIRED      | This value means that the bean methods must always execute in a transaction context. If there is a transaction already running, the bean participates in that transaction. If there is no transaction, the EJB container will start a new one on behalf of the bean.                                                                                                                                                                                                                                           |
| TX_REQUIRES_NEW  | Here the bean requires that a new transaction is always started, when any of its methods is called. If there is a transaction running, it is suspended. The container will start a new transaction and at the end of the method execution, it will commit or abort it. After that, the container will resume the client transaction.                                                                                                                                                                           |
| TX_SUPPORTS      | This value means that the bean participates in a running transaction but does not require it. So, if there is no transaction, the method executes without a transaction.                                                                                                                                                                                                                                                                                                                                       |
| TX_BEAN_MANAGED  | Here, the enterprise bean programmatically controls its own transaction boundaries. In WebSphere, only session beans may be marked with this value, as specified by EJB 1.1. When this value is used, it is the bean that demarcates the transaction and not the container, as with all the previous values. We will see more about the different demarcation types in the section that follows. Note that when using TX_BEAN_MANAGED, you cannot mix transaction attributes on different methods in the bean. |

In most cases, TX\_REQUIRED (the default) is the best choice:

- ▶ If a transaction does not exist, one is created for the duration of the method of the bean that is invoked. At the end of the method the changes are committed and the database is updated (except for a read-only method where no database update is required).
- ▶ If a transaction already exists, it is used by the methods of the bean. For example, a transaction is created by executing a method in a session bean, and automatically the methods in entity beans that are called from the session bean run under the same transaction.

Table 12-1 summarizes the transaction attributes.

Table 12-1 Transaction attributes and their use

| Transaction attribute | Client transaction | Transaction associated with bean method |
|-----------------------|--------------------|-----------------------------------------|
| TX_NOT_SUPPORTED      | None               | None                                    |
|                       | T1                 | None                                    |
| TX_REQUIRED           | None               | T2                                      |
|                       | T1                 | T1                                      |
| TX_SUPPORTS           | None               | None                                    |
|                       | T1                 | T1                                      |
| TX_REQUIRES_NEW       | None               | T2                                      |
|                       | T1                 | T2                                      |
| TX_MANDATORY          | None               | ERROR                                   |
|                       | T1                 | T1                                      |

**Note:** The values of these attributes have changed in EJB 1.1. The prefix TX is removed as well as the space between words. For example, TX\_REQUIRES\_NEW has become *RequiresNew*. A new value has also been added: *Never*.

## How to read the table

- ▶ The first column contains all the transaction attributes, except TX\_BEAN\_MANAGED.
- ▶ The second column shows whether there is a transaction running already or not, when the client calls a bean method. If there is a transaction, it is named as *T1*, otherwise it is marked as *None*. A client could be a session bean calling an entity bean.
- ▶ The third column shows what happens when the method is executed on the bean. If the bean participates in the existing transaction, this is marked as *T1*. If the bean starts a new transaction, then *T2* is used. If the bean ignores the running transaction, it is marked as *None*. If the bean requires a transaction and none is running, it is marked with *ERROR*.

## Method-level attributes

The bean-level attributes can be overwritten for individual methods of the bean. Select a method and *EJB Method Attributes -> Add Control Descriptor* (Figure 12-3).

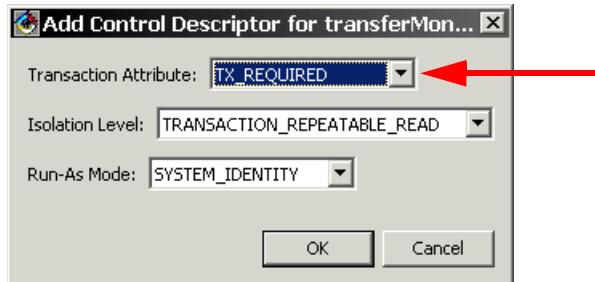


Figure 12-3 Method-level attributes

### Example

Let's see now how we would use these transaction attributes if we wanted to register to a conference event. We assume that the interested person is located in Europe and that the event is taking place in US. We also assume that the event holder provides all the travel arrangement services to the participants, apart from the registration.

Let's now say that we have a session bean that provides all the required services in separate methods. These methods will provide:

- ▶ Flight booking
- ▶ Car reservation
- ▶ Hotel reservation
- ▶ An e-mail notification
- ▶ Registration

When a person wants to register to the conference, he or she must first have a confirmed seat. The next requirement is that this person is booked on a flight and has a room reserved in a local hotel. Also a car is useful when visiting the USA, especially when the hotel is not in the neighborhood of the conference location. However, this is not absolute and the participant would manage without a car as well. Finally, the participant should be notified of his or her registration by e-mail.

In this approach, we set the transaction attributes at the method level. Notice that it is important that the bean itself is not marked as TX\_BEAN\_MANAGED. We can leave the default setting in VisualAge for Java, that is TX\_REQUIRED. The main method of our session bean could be implemented as shown in Figure 12-4.

```

void newParticipant() { // T1 starts here

 doRegistration(); // T1 is used

 doFlightBooking(); // T1 is used

 doRoomReservation(); // T1 is used

 doCarReservation(); // T2 starts here

 sendEmailNotification(); // T1 is used
}

```

*Figure 12-4 Session bean main method*

Based on the requirements listed above, we would set the transaction attributes for these methods as:

|                       |                 |
|-----------------------|-----------------|
| newParticipant        | TX_REQUIRED     |
| doRegistration        | TX_MANDATORY    |
| doFlightBooking       | TX_MANDATORY    |
| doRoomReservation     | TX_MANDATORY    |
| doCarReservation      | TX_REQUIRES_NEW |
| sendEmailNotification | TX_SUPPORTS     |

When the newParticipant method is called, a new transaction is started, unless a transaction is provided by the caller (and the method will participate in that transaction). Let's call this transaction *T1*.

Now, when the methods doRegistration, doFlightBooking and doRoomReservation are called, there must be a transaction running, so that they can execute under the same transaction context. Because it is the newParticipant method that calls all these methods, the transaction T1 is provided to them. If there was no transaction, these three methods would not execute, but instead throw a TransactionRequiredException exception.

When the doCarReservation method is called, this will start a new transaction, *T2*. We want to have a new transaction started here, so that if this method fails, we are still able to finish the registration. Finally, the last method, sendEmailNotification, will run under the T1 transaction. We do not use here the TX\_MANDATORY attribute, because the e-mail notification is not so critical as the registration, flight booking and room reservation.

In this method, we use transaction types for isolating non-critical transactions from critical. In the transferMoney method, we used one flat transaction, because we want all the operations to run under the same transaction context.

You can see in Figure 12-1, where the transactions T1 and T2 are started. Remember that when we use transaction attributes, we allow the container to set the boundaries (demarcation) and manage all our transactions. This type of transaction is called *container-managed transaction*.

There are three types of transaction demarcation. We will describe these in “Transaction demarcation” on page 232.

## Isolation

Isolation protects transactions that are concurrently executing from interfering with each other's results. It allows multiple transactions to read or write to a database without knowing about each other. Isolation ensures the consistency of the application state. In general, it should ensure that the effect of concurrent transactions should be the same as that of executing the transactions sequentially. Isolation is achieved by using low level synchronization protocols on data in the permanent store. This synchronization ensures the isolation of one transaction from another.

The EJB specification defines some isolation levels. Choosing the right one ensures the robustness and scalability of an application. Isolation is achieved by performing concurrency control.

## Concurrency control

Concurrency control can be better explained with an example. Let's assume that two different applications want to update a shared database table, for example, the Account table of bank accounts.

In this example, each application represents an account holder of a joint account, and performs a deposit transaction:

1. Read the balance field from a row representing an account in the table.
2. Add a dollar amount to the balance.
3. Write the new value of balance to the table.

If all the three steps execute together, one application cannot interfere with the operations of another. But the system controller does not guarantee this. Therefore, the operations could be interleaved. In a worst case, this scenario is possible:

1. Application X reads the balance from the table. Assume the value is zero.
2. Application Y also reads the zero balance from the table.

3. Application X adds 500 dollars to the balance and updates the table. The value of the balance in the table is now 500 dollars.
4. Application Y adds 200 dollars to its local copy of balance and updates the table. The value of the balance in the table is now 200 dollars.

Due to the interleaving of operations, application Y is working with an obsolete copy of the balance data. That is, before application X updates the table, application Y reads data from the table. Therefore, the operation of X is never recorded. This scenario is known as *lost update*.

## Data locking

The solution to the above mentioned scenario is *data locking*. Locking is a database operation that restricts a user from accessing a table or record(s). By locking the data, it is assured that only one application has access to the data, until the lock is released. This prevents the interleaving of operations. Now revisiting the scenario discussed above, we have these operations:

1. Request a lock on the table row and receive the lock.
2. Read the balance from the table.
3. Add some amount to the balance.
4. Update the balance in the table.
5. Release the lock on the table row.

If there are concurrent transactions and one application acquires a lock of the data, all the other applications have to wait, until the lock is released.

To improve performance, transactions use two main types of data locks. They are *read locks* and *write locks*. Read locks are non-exclusive. That is, any number of transactions can acquire the lock, concurrently. But write locks are exclusive. Only one transaction can hold a write lock at a time.

## Limits

Because locking physically prevents other concurrent transactions from accessing the data, major performance problems may arise. In addition, deadlock of transactions can also occur, which may cause stability concerns to the applications. An example of deadlock is when two concurrent transactions are waiting for each other to release a lock.

## Problems of concurrent transactions

Now, we discuss some problems that occur due to concurrent transactions.

## **Dirty read**

A *dirty read* occurs when an application reads data from a database that has not been committed to permanent storage. Consider the following scenario:

- ▶ Application X reads the balance from a table, for example, 100.
- ▶ X adds 500 to balance and updates the table. The value of balance in the database now is 600. But X does not commit the update yet.
- ▶ Y reads the balance from the table; the value is 600.
- ▶ X aborts the transaction. The balance is restored to 100.
- ▶ Y withdraws 300 from the account and updates the table; the balance is now 300. This is obviously wrong.

What happened is that Y reads the update of X, before X has committed. Therefore, the database has been wrongly updated. This problem of reading uncommitted data is called *dirty read*.

## **Nonrepeatable read**

*Nonrepeatable read* occurs when data has been changed between two consecutive reads of the same data. For example consider the following situation:

- ▶ Application X reads the balance from a table, for example, 100.
- ▶ Application Y updates the balance value to 20.
- ▶ Application X again reads the table and gets a different value.

This can arise when a component modifies the data being read inbetween the two reads of another component.

## **Phantom read**

*Phantom read* occurs when a new set of data is inserted into a table between two read operations. Consider the following case:

- ▶ Application X queries the table and retrieves a result set of five records.
- ▶ Application Y updates the table with a new row.
- ▶ Application X again performs the query and retrieves a result set of six records.

Therefore, unrepeatable reads occur when existing data is changed, while phantom reads occur when new data is inserted.

Here is the summary of these three problems (Table 12-2).

Table 12-2 Summary of errant behaviors

| Problem             | Description                                                                                                                                                                                                                                                     |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Dirty reads         | User 1 modifies a row. User 2 reads the same row before user 1 commits. User 1 performs a rollback. User 2 has read data that has never really existed in the database.                                                                                         |
| Nonrepeatable reads | User 1 reads a row but does not commit. User 2 modifies or deletes the same row and then commits. User 1 rereads the row and finds it has changed (or has been deleted).                                                                                        |
| Phantom reads       | User 1 uses a search condition to read a set of rows but does not commit. User 2 inserts one or more rows that satisfy this search condition, then commits. User 1 rereads the rows using the search condition and discovers rows that were not present before. |

Now, let's see the various isolation levels that address these problems.

## Isolation levels

An isolation level represents a particular locking strategy employed in the database system to improve data consistency. Isolation levels are specified for enterprise java beans similar to transaction attributes. We can either specify a strict isolation or a relaxed isolation. It is a trade off between concurrency control and performance. That is, a strict isolation level can only be achieved at the expense of performance.

Isolation levels for enterprise beans are specified in the Properties dialog (Figure 12-2 on page 222). The isolation level can be overwritten for individual methods (Figure 12-3 on page 225).

There are four isolation levels:

- ▶ TRANSACTION\_READ\_UNCOMMITTED
- ▶ TRANSACTION\_READ\_COMMITTED
- ▶ TRANSACTION\_REPEATABLE\_READ
- ▶ TRANSACTION\_SERIALIZABLE

### **TRANSACTION\_READ\_UNCOMMITTED**

This is the weakest isolation level. When this isolation level is used, all the above mentioned isolation problems occur, regardless of the isolation levels of other transactions. Therefore, this isolation level should not be used for mission critical applications. This level is more suited if we know that only one application will be

running at a given time and there are no other concurrent transactions. The advantage of this isolation level is good performance.

### **TRANSACTION\_READ\_COMMITTED**

This level is very similar to TRANSACTION\_READ\_UNCOMMITTED. The only difference is that this isolation level addresses *dirty read*. Therefore, transactions will read committed data only, when running in this level. However, this isolation level does not address unrepeatable reads and phantoms. This level guarantees that the data we read is always consistent. This mode is useful for report-generating programs that use the snapshot of the database at the time of report generation.

### **TRANSACTION\_REPEATABLE\_READ**

This mode guarantees that repeated reads of the database results in the same data values. Therefore, it addresses both *dirty read* as well *unrepeatable read*. This mode is useful, when we have to update the database records often. This prevents data from being modified by other concurrent transactions. However, phantom reads may occur.

### **TRANSACTION\_SERIALIZABLE**

This is the strictest isolation level. This mode enforces all the ACID properties and guarantees fully independent transactions. This is useful for mission critical applications. It ensures that no intermediate transaction results can appear. Therefore, even if transactions occur concurrently, users will view their effects only successively. However, database access performance may suffer when this isolation level is used.

A summary of the isolation levels is shown in Table 12-3. The value of *NO* indicates that the problem does not occur; *YES* indicates that the problem may occur.

Table 12-3 Isolation levels

| Isolation level              | Phantom read | Dirty read | Nonrepeatable read |
|------------------------------|--------------|------------|--------------------|
| TRANSACTION_SERIALIZABLE     | NO           | NO         | NO                 |
| TRANSACTION_REPEATABLE_READ  | YES          | NO         | NO                 |
| TRANSACTION_READ_COMMITTED   | YES          | NO         | YES                |
| TRANSACTION_READ_UNCOMMITTED | YES          | YES        | YES                |

## Isolation levels in JDBC

JDBC also deals with transaction and supports its own set of isolation levels. They correspond exactly to the isolation levels supported by enterprise beans. The only exception is the TRANSACTION\_NONE isolation level in JDBC, which is not supported by enterprise beans. The equivalent of this isolation level can be achieved by specifying the bean transaction attribute as TX\_NEVER.

**Important:** In a typical scenario, in which a client invokes methods of multiple beans in a single transaction, ensure that all of the beans methods are set to the same isolation levels. A specification that changes the isolation level is not honored after a transaction has started, and results in an IsolationLevelChangeException (which triggers a RemoteException).

## Mapping JDBC isolation levels to DB2

DB2 accepts a set of isolation levels when you bind an application. The DB2 specification is a little different than the JDBC specification, but there is a perfect match between the two sets (Table 12-4).

*Table 12-4 DB2 isolation levels*

| Isolation level in JDBC      | Isolation level in DB2 | Abbreviation |
|------------------------------|------------------------|--------------|
| TRANSACTION_SERIALIZABLE     | Repeatable Read        | RR           |
| TRANSACTION_REPEATABLE_READ  | Read Stability         | RS           |
| TRANSACTION_READ_COMMITTED   | Cursor Stability       | CS           |
| TRANSACTION_READ_UNCOMMITTED | Uncommitted Read       | UR           |
| TRANSACTION_NONE             | Not supported in DB2   |              |

## Transaction demarcation

In this section, we describe the three different types of transaction demarcation that can be used when programming with enterprise Java beans.

### Container-managed transaction

The simplest solution is when the container manages the transactions. There is no need to write any transaction logic, because the required transaction logic is handled by the container at runtime.

Before and after the execution of any method, the container is calling two special methods, `pre_invoke` and `post_invoke`, respectively. These methods contain all the necessary logic that is required for the method execution. When we are using a container-managed transaction, it is in these methods, where the container actually starts, commits, or aborts the transaction.

Depending on the transaction attributes we set for a bean and/or its methods in the deployment descriptor, the container uses or ignores an existing transaction, or starts a new one.

This was the way we implemented the transfer bean, “Developing session beans with VisualAge for Java” on page 174. The transaction attribute of our bean was set to `TX_REQUIRED` (Figure 12-2 on page 222).

You can also set the transaction attribute at the method level. If different from the transaction attribute at the bean level, the method attribute will overwrite it (Figure 12-3 on page 225).

**Note:** When a problem occurs, use the `setRollbackOnly` method of the `EJBContext` to mark the transaction for rollback. This is usually done before throwing an application exception. Application exceptions do not automatically cause the container to rollback the transaction.

## Bean-managed transaction

A session bean can be marked as `TX_BEAN_MANAGED`. This attribute enables the bean to programmatically control its own transactions. This is done by using the Java Transaction API (JTA) and more specifically the methods defined in the `javax.transaction.UserTransaction` interface.

The `UserTransaction` is retrieved from the `EJBContext`. Figure 12-5 shows an example where a bean method manages the transaction itself.

Programming transaction control is the more traditional way of adding transaction logic to your code. The developer has full control of where a transaction is started and ended. However, programming this way mixes infrastructure logic with business logic in an application and should generally be avoided.

After using the transaction attributes provided by enterprise beans, you will realize that you have a lot of freedom to implicitly control the transactions in an application. This approach enables you to solve most of the problems and keeps your code clear from infrastructure logic, because it is handled by the container.

```
beanMethod() {
 // Use the context to start a user transaction
 UserTransaction tranCtx = getSessionContext().getUserTransaction();

 // Begin the transaction
 tranCtx.begin();

 Perform transactional work.....

 //Commit the transaction
 tranCtx.commit();

 // Begin another transaction
 tranCtx.begin();

 tranCtx.commit();
}
```

Figure 12-5 Bean-managed transaction example

In certain cases, however, you may be forced to programmatically control transactions. One such case is when it is required that you start a transaction in one method and end it in another. From a design point of view, this is not the best thing you can do, because all the accessed tables will be locked for a longer time.

**Attention:** Spanning a transaction across many methods is only allowed in stateful session beans (see “Conversational state” on page 158). Stateless session beans must end or abort all transactions they start in one method call.

## Client-managed transaction

This is the case where the EJB client is explicitly controlling the transaction. For example, this can be a Java servlet that updates some records in a database and starts, ends, or aborts a transaction based on some external criteria.

Again, it is the javax.transaction.UserTransaction interface of the JTA that is used in the client program. The UserTransaction is retrieved through the naming service. Figure 12-6 shows an example where a client manages the transaction itself.

```

// Use JNDI to locate the UserTransaction object
Context ctx = new InitialContext();
UserTransaction t1 = (UserTransaction)ctx.lookup("jta/usertransaction");

// Begin the transaction
t1.begin();

 Perform transactional work.....

//Commit the transaction
t1.commit();

```

*Figure 12-6 Client-managed transaction example*

When the client is demarcating a transaction, it is recommended that the bean methods that the client calls are enabled for container-managed transaction. This simplifies the application and protects the developer from many problems.

**Note:** Be careful when using JTA to demarcate transactions in your client. Once you have called a method on a bean and that bean is running in a transaction, you must first complete the original transaction with a commit or abort, before you invoke a method of the bean that requires its own or no transaction. Otherwise, you will get a `java.rmi.RemoteException`.

## Recapitulation

Table 12-5 summarizes the available types of transaction demarcation.

*Table 12-5 Transaction demarcation types*

| Demarcation type              | Description                                                                                                                          |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| Container-managed transaction | Let the container decide when to start, commit, or rollback transactions, based on deployment descriptor (bean properties).          |
| Bean-managed transaction      | You explicitly start, commit, or rollback transactions in enterprise bean methods, using the <code>UserTransaction</code> interface. |
| Client-managed transaction    | You explicitly start, commit, or rollback transactions in the EJB client code, using the <code>UserTransaction</code> interface.     |

The best and easiest way is to allow the container to manage the transactions on behalf of the bean, so that you can focus on the business logic of the application.

There are a number of attributes that you can use for implicitly controlling transactions and for dealing with most scenarios. Use bean-managed transactions only when it is the only way to solve a problem.

You should keep in mind that EJBs were designed for delegating all the middleware logic to the container, so that the developer does not have to worry about any of this.

## Distributed transactions

When a transaction updates data across more than one data source, either on the same database server or across many database servers on different physical machines, it is called a *distributed transaction*. Providing a transaction across multiple systems requires more coordination.

### XA protocol

XA is the standard interface between a transaction manager and a resource manager. The transaction manager coordinates a distributed transaction. It typically uses the XA protocol to interact with the database backends. The databases has to understand the XA protocol for distributed transactions. There is a resource manager for each participant in the distributed transaction, and the resource manager is the communication point for the transaction manager. To support the XA protocol, a number of extra remote procedural calls must be sent to the database both before and after a given connection is used.

### Two-phase commit

Two-phase commit is a protocol complying to the XA interface. It ensures that the result of a transaction is consistent across all resource managers participating in the transaction. It is used only in distributed transactions. The protocol operates in distinct phases to ultimately commit or abort a transaction.

*Phase one* evaluates the status of each resource manager. The transaction manager checks with each local resource manager, whether they are ready to commit the transaction. Each resource manager responds that they are ready or not. A transaction can commit only when all participating resource managers agree during this phase one. This phase is called the *prepare* phase.

*Phase two* concludes the transaction. Based on the response from each resource manager, the transaction manager instructs all resource managers to commit the transaction if all agree or to roll back the transaction if at least one disagrees. This phase is called *commit* phase.

## Transaction context

Transaction context is sent over the transactional communications protocol, such as Internet Inter-ORB Protocol (IIOP). It is an object that holds information about the system transactional state. It is passed between different nodes participating in the same transaction. For any component to participate in a transaction, it has to associate with a transaction context.

## Unsuccessful prepare

When a resource manager fails to prepare, it performs these actions:

- ▶ It releases any resources held by the transaction and rolls back the local portion of the transaction.
- ▶ It responds with an abort message when it is referenced in the distributed transaction.

These actions then propagate to the other transaction managers involved in the distributed transaction to rollback the transaction and guarantee data integrity. Again, this enforces that all nodes involved in the transaction either commit or rollback the transaction at the same logical time.

## Real world scenario

In the real world, we often have to carry out transacted operations that span different databases, which may be on different servers and even at different sites. The distributed transaction coordinator or manager is a service, that coordinates transactions spanning separate data stores or resources. To understand how a distribution transaction works, see Figure 12-7.

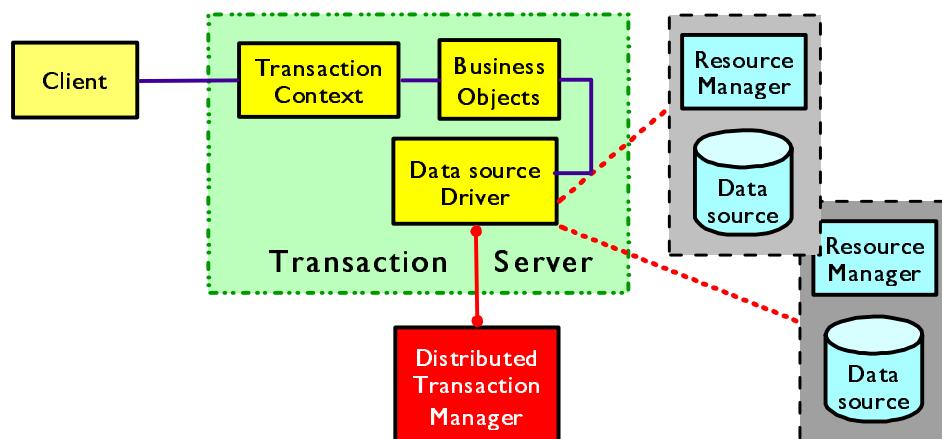


Figure 12-7 Distributed transaction

Consider the case, where we have two data sources to be updated within the same transaction:

- ▶ When an application requests access to the first data source, the data source driver (usually JDBC or ODBC) checks with the appropriate transaction context object to see if a transaction is required. If it is, it informs the coordinator, and then contacts the resource manager for the data store and tells it to start a transaction within that data source. In other words, it automatically starts an integral database transaction for this operation.
- ▶ Now the component opens a connection to the second data source. The coordinator and the driver software contact this resource manager and instruct it to start a new transaction for the updates that follow. Now both data sources are holding active transactions with the updates.
- ▶ Once the component concludes with either aborting or committing the operations, the coordinator contacts both resource managers and instructs them to commit or abort their current transaction. In this way, the transaction will be expanded to include the remote data sources as well. Either all the updates on all the data sources will be committed or else all will be rolled back.

## Distributed transaction with DB2

You must use a DataSource with the JTA driver to enable distributed transactions involving DB2 tables (see Figure 6-8 on page 90).

## New transaction support in WebSphere 3.5.3

Fixpack 3 of WebSphere Application Server 3.5 introduced:

- ▶ Transactional JMS support and more important distributed transaction support across JMS and EJBs.
- ▶ Distributed transaction support for Oracle and Microsoft SQL Server using the Merant drivers; in addition to the existing support for DB2 and Sybase, plus support across multiple database vendors.

## Troubleshooting

The most common problems encountered in the transaction world with databases are deadlocks. Therefore, enterprise beans suffer from the same problems.

## Deadlocks

Deadlocks occur when two concurrent transactions place a shared lock on the same resource (table or row) when they read it—then attempt to update the information at commit time. This can happen when the same enterprise bean is accessed and updated by two or more clients at the same time. When using DB2, a deadlock results in one of the clients getting a rollback exception.

You can verify when this happens by activating an application server trace on the com.ibm.ejs.cm.portability component.

## Find for update

Unfortunately, the recovery from exceptions due to conflicts can be non-trivial. What is needed is a way to indicate that the enterprise bean is being accessed with the intent to update the data so that deadlock conditions can be avoided. That is exactly what the *Find for update* setting does. It basically instructs the runtime to use a write lock or write intent when using the findByPrimaryKey method to retrieve an enterprise bean. The main effect results in the serialization of concurrent requests for the same bean.

In VisualAge for Java you can set *Find For Update* in the Properties for an EJB server from the EJB Server Configuration window (Figure 12-8).



Figure 12-8 Find for update setting in VisualAge for Java

In WebSphere you can set this property individually for each bean in the Administrators Console dialog (Figure 12-9).

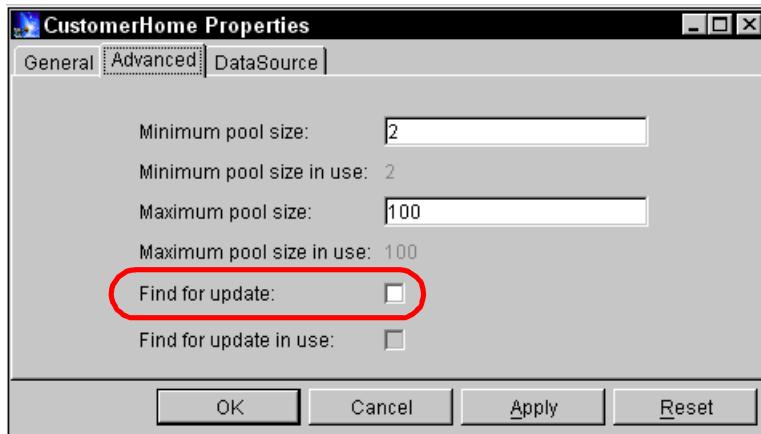


Figure 12-9 Find for update setting in WebSphere

## Summary and guidelines

In this section we give you some guidelines for handling transactions and for application programming.

### Guidelines for transactions

Here are some guidelines for transaction demarcation and transaction attributes:

- ▶ Use long-running read only transactions to cache static data only.
- ▶ Avoid referring to dynamic data in long running read-only transactions, because the repeatable-read transaction isolation prevents detecting changes committed by other top-level transactions.
- ▶ Properly demarcate transaction boundaries and complete transactions quickly.
- ▶ The recommended way to manage transactions in EJB is through container-managed demarcation.
- ▶ Entity beans should always use container-managed transaction demarcation. Session beans can use either container-managed or bean-managed transaction demarcation.
- ▶ Bean-managed transaction demarcation should be used only when it is the only way to address a problem.

- ▶ Stateless session beans should always either commit or rollback a transaction before the business method returns.
- ▶ The default choice for a transaction attribute should be `TX_REQUIRED`. Enterprise beans with the `TX_REQUIRED` transaction attribute can be easily composed to perform work under the scope of a single JTA transaction.
- ▶ The `TX_REQUIRE_NEW` transaction attribute is useful when the bean method has to commit its results independent of other transactions, enabling normal transactions to be isolated from critical transactions.
- ▶ Session beans that implement the interface `SessionSynchronization` must have either the `TX_REQUIRED`, `TX_REQUIRE_NEW`, or `TX_MANDATORY` transaction attribute.
- ▶ The transaction attributes `TX_MANDATORY` and `TX_NEVER` reduce composition of a component by putting constraints on the calling client's transaction context. These attributes can be used when it is necessary to verify the transaction association of the calling client.
- ▶ The transaction attribute `TX_SUPPORTS` is not recommended. It has transactional behavior depending on the client association with a transactional context, which is a violation of ACID properties.
- ▶ Try avoiding the use of `TX_SUPPORTS` and `TX_NOT_SUPPORTED`, because if the application accesses multiple entities with container-managed persistence in a single high-level business operation, such as a servlet invocation, you may experience unexpected results. The bean methods for these cases are run under an *unspecified transaction context*, where the container determines the semantics. In the absence of a transaction, the current implementation of the container executes the database accesses in a local transaction. If those local transactions use different connections to the same database, deadlocks can occur. This is true regardless of whether there is a single or multiple application servers, and whether JTA is enabled.
- ▶ Avoid mixing methods with different isolation levels in the same transaction. It causes an `IsolationLevelChangeException`, because a connection cannot change its isolation level.

## Guidelines for applications

Here are some guidelines for application programming:

- ▶ Application beans should explicitly make the decision to cause a rollback and not leave it to the container. They can do this by calling the `EJBContext.setRollbackOnly` method. Remember, application exceptions result in a commit unless you explicitly call the `setRollbackOnly` method.
- ▶ When calls are returned from downstream EJBs, application beans should check if their transaction has been marked for rollback using the

`EJBContext.getRollbackOnly` method, and act accordingly. They should not just rely on getting this notification as a result of a rollback exception.

- ▶ An application can set `RollbackOnly` without necessarily throwing an exception, although this should be avoided.
- ▶ Application exceptions should follow the EJB 1.1 specification and no longer extend `RuntimeException` or `RemoteException` or any of its subclasses.
- ▶ Application beans should throw an exception that extends `RuntimeException` if it wants to cause a rollback and indicate a system level error. In the EJB 1.1 specification, the application would have been required to throw an `EJBException`, but this has not yet been implemented in WebSphere V3.5.
- ▶ When a call to a bean results in a thrown `RemoteException`, you cannot know that the target bean rolled back, or was even called. In this case (unless you really know what you are doing), you should call `EJBContext.setRollbackOnly` and throw an appropriate application or `RemoteException` and cause the entire transaction to roll back. Leave it to the next level up to retry the transaction if desired.
- ▶ If an application bean wants to report a system type problem and cause a rollback, it should continue to throw a `RemoteException` until WebSphere fully supports EJB 1.1.
- ▶ If an application bean wants to ensure that its work is committed regardless of the outcome of calls to other downstream beans, the downstream bean's transaction attribute should be marked as `TX_REQUIRE_NEW` or `TX_NOT_SUPPORTED`, or the calling bean should manage its own transaction (`TX_BEAN_MANAGED`). This will cause the caller's transaction to be suspended when the downstream bean is called.
- ▶ Note that a transactional attribute of `TX_REQUIRE_NEW` can set the stage for a deadlock, if the new and the suspended transactions try to use the same resources, depending on their transaction isolation levels.
- ▶ It is generally stated in EJB books and articles that bean-demarcated transaction management should be avoided. While this is true in most cases, it should not be dismissed as a bad thing. In the case where a session bean is coordinating a complex process or set of activities, it may be more practical to use bean-demarcated transactions. This will allow you to split the processing into multiple transactions, recover from downstream beans that cause rollback, and explicitly ensure that a downstream bean will not affect your processing. If you find yourself fighting transaction problems, consider using bean-demarcated transactions in your main session bean. Alternatively, instead of a session bean, you can resort to a non-EJB caller that will demarcate transactions explicitly, but that requires writing even more code.



# Client programming

In this chapter, we will describe how clients can access and use enterprise Java beans in distributed applications. After introducing some basic concepts, we will cover the main alternatives a developer has: direct access to entity beans, using access beans, and using a session facade.

We first describe simple applications to illustrate the concepts, but then we focus on using servlets as EJB client applications and give examples for each approach we present. At the end of the chapter, we will compare these different approaches and provide guidelines for using them.

# Model-view-controller

Before presenting the different possible ways about how a client can access enterprise beans, we will describe the programming model that we use in our examples.

The model-view-controller, also known as MVC, is a programming paradigm that was first introduced by Smalltalk. It is a design pattern that aims at separating the business logic from interaction and presentation logic.

In a typical Web application, a user interacts with a Web site using a browser. The user makes a request by following a link or submitting a form. The application, after processing the request, returns a response to the user. The response is presented to the user in HTML. A typical example is when a user tries to get product or account information by using the product or account ID respectively.

The MVC model enforces the application to be split to three main parts, the *model*, the *view* and the *controller* (Figure 13-1).

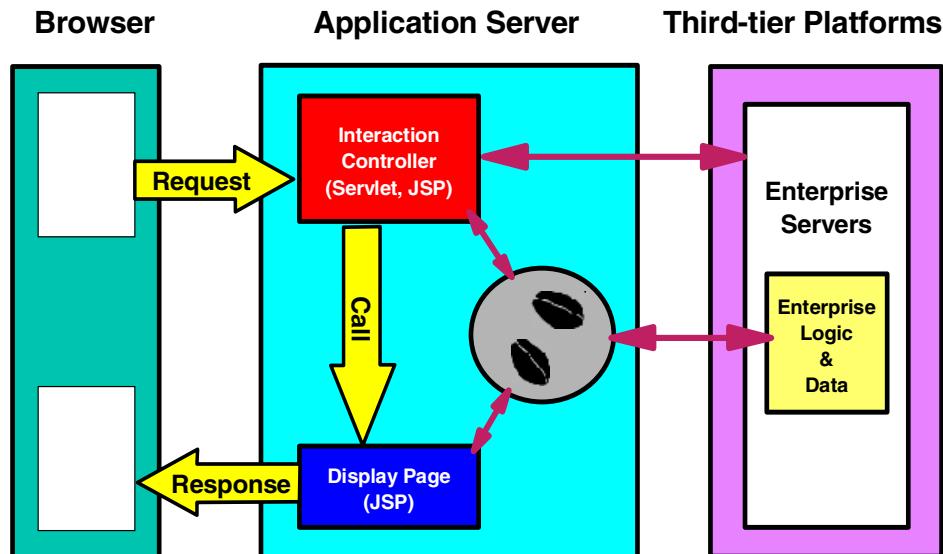


Figure 13-1 MVC programming paradigm

|              |                                                                                                                                    |
|--------------|------------------------------------------------------------------------------------------------------------------------------------|
| <b>Model</b> | It encapsulates all the business logic and rules and does the business processing. It is usually implemented by JavaBeans or EJBs. |
|--------------|------------------------------------------------------------------------------------------------------------------------------------|

|                   |                                                                                                                                                                                                                                                                       |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>View</b>       | It uses the result of the business processing and constructs the response that is presented to the user. It is usually implemented by JavaServer pages.                                                                                                               |
| <b>Controller</b> | It manages and controls all the interaction between the user and the application. Usually, it is a servlet that receives the user request and passes all the input parameters to the model that does the actual work. Finally the JSP is called to return the output. |

This approach gives us many advantages, for example:

- ▶ It provides clear separation of the business logic from presentation and interaction logic.
- ▶ It allows the use of three-tier architectures where business logic and presentation logic are placed on different servers with different security models applied.
- ▶ It enables role-separation in a development team, where the work of a page designer smoothly integrates with the work of a business developer.

We will use this model in our examples throughout this chapter.

When applying this programming model in Web applications, the vehicle that is used for carrying data or objects from the servlet to the JavaServer page is often the `HttpServletRequest` or `HttpSession`. The request object contains all the information that a Web browser sends to a Web server, when a user requests a resource (URL) on the server.

## How to access EJBs

Now we are ready to start writing our client application for accessing EJBs. In our examples we will write servlets that will create and find `Customer` entity beans. These servlets will be called `CreateCustomer` and `FindCustomer` respectively. The results of a find or a create method will be displayed by a JSP applying the MVC model that we described.

### The basics

We will first see how we can write a simple EJB client that can find and use enterprise beans in an EJB container.

There are certain steps a client has to follow before accessing enterprise beans on an EJB server. These steps are presented here.

## Obtain an initial naming context

When you are writing an EJB client, you are responsible for creating a JNDI InitialContext that will be hooked to the naming service running in the application server. Two parameters must be given:

- ▶ The location, host name and port of the naming service (PROVIDER\_URL)
- ▶ The name of the initial context factory (INITIAL\_CONTEXT\_FACTORY)

This information is provided to the constructor of the InitialContext as a java.util.Properties. Here is an example:

```
java.util.Properties properties = new java.util.Properties();
properties.put(javax.naming.Context.PROVIDER_URL, "iiop:///");
properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
 "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
javax.naming.InitialContext initialContext =
 new javax.naming.InitialContext(properties);
```

Notice that we use the string “iiop://” for the PROVIDER\_URL property. This string indicates to the runtime environment to find the naming service at the standard port on the local machine.

The general form of the PROVIDER\_URL string is “iiop://hostname:port/”. The value of the second property, INITIAL\_CONTEXT\_FACTORY, is the class name of the naming service factory. For WebSphere Application Server or VisualAge for Java, use com.ibm.ejs.ns.jndi.CNInitialContextFactory.

**Attention:** Since Version 3.5 of the toolset, you can abbreviate the coding and default values are taken for INITIAL\_CONTEXT\_FACTORY and PROVIDER\_URL:

```
javax.naming.InitialContext initialContext =
 new javax.naming.InitialContext();
```

## Looking up an EJB home from the initial context

Using the initial naming context, we can retrieve the EJB home of the entity bean we want to access, for example, the Customer entity bean:

```
Object objHome = initialContext.lookup("itso/ejb35/cmp/Customer");
CustomerHome customerHome = (CustomerHome)
 javax.rmi.PortableRemoteObject.narrow(objHome,CustomerHome.class);
```

Notice the JNDI name of the customer entity bean, itso/ejb35/cmp/Customer. This is the default name that VisualAge for Java generated for us, when we created the customer bean.

In the second line we use the utility class PortableRemoteObject to “narrow” the reference that was obtained from the InitialContext. This can be seen as equivalent to casting; it takes a generic RMI-IIOP reference and returns an instance of the proper class.

## Using the EJB home

With the EJB home, we can create new instances, find, and remove entity beans. Note that most methods can throw a FinderException and a RemoteException; these have to be handled in a try/catch block.

- ▶ Creating a new customer bean:

```
Customer customer = customerHome.create(customerID, title, firstName,
 lastName, userid, password);
```

Here we use one of the available constructors of the Customer bean that does not take the CustomerAddress as an argument. This property can be set later by using the setAddress method.

- ▶ Finding a customer bean:

```
Customer customer = customerHome.findByPrimaryKey(new CustomerKey(id));
```

The findByPrimaryKey method takes a parameter of type CustomerKey that we initialize with the an ID that was entered by the user as an input parameter. Custom finder methods are also invoked from the home.

- ▶ Removing a customer bean:

```
customerHome.remove(customer);
```

You can also use the javax.ejb.Handle of the customer as a parameter. This method results to the deletion of the corresponding row in the customer table.

**Note:** If you go to the Members pane of the EJB page of the customer bean, you may not see the remove methods. These methods are inherited from the EJB home and can be viewed by setting the appropriate view filter. Select from the context menu in the Members pane *Inheritance Filters* and *All Inherited Methods*, or use the *Root Minus One* icon on top of the pane.

## Manipulating EJB references

After acquiring the customer bean object, we can invoke any of the methods available in the remote interface, for example, we can get or set the customer properties. We can also invoke the remove method on the EJB object.

```
customer.getName();
customer.setLastName("Newname");
customer.remove();
```

In our example, we are passing the customer object to the result JSP and invoke the getter methods from there instead.

## Simple test application

Figure 13-2 shows a simple main application that uses the basic code to find the home, create and retrieve entity beans, and delete a bean.

```
package itso.ejb35.cmp.client;
import itso.ejb35.cmp.*;
public class SimpleApp {
public static void main(java.lang.String[] args) {
 try {
 //java.util.Properties properties = new java.util.Properties();
 //properties.put(javax.naming.Context.PROVIDER_URL, "iiop:///");
 //properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
 // "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
 //javax.naming.InitialContext initialContext =
 //new javax.naming.InitialContext(properties);
 javax.naming.InitialContext initialContext =
 new javax.naming.InitialContext();
 Object objHome = initialContext.lookup("itso/ejb35/cmp/Customer");
 CustomerHome customerHome = (CustomerHome)
 javax.rmi.PortableRemoteObject.narrow(objHome, CustomerHome.class);

 Customer cust2 = customerHome.create(201,"Mr","John","Smith","JS","JS");
 System.out.println("Customer 201 created");
 Customer cust1 = customerHome.findByPrimaryKey(new CustomerKey(101));
 System.out.println("Customer 101: name="+cust1.getName());
 System.out.println("Customer 201: name="+cust2.getName());
 cust2.setLastName("Keller");
 System.out.println("Customer 201: name="+cust2.getName());
 cust2.remove();
 System.out.println("Customer 201 removed");
 } catch(Exception ex) { ex.printStackTrace(); }
}
}
```

Figure 13-2 Simple EJB client application

**Important:** When calculating the class path for this application be sure to include these projects:

IBM EJB Tools

IBM Enterprise Extension Libraries

IBM WebSphere Test Environment

<== not included by calculate

## Using a finder method

Many finder methods return an enumeration. You can loop through the enumeration to access the returned entity beans. Note that you have to use the *narrow* method to convert a returned object into the desired remote object. Figure 13-3 shows an extract of the main method.

```
package itso.ejb35.cmp.client;
import itso.ejb35.cmp.*;
public class UsingFinder {
 public static void main(java.lang.String[] args) {
 try {
 getInitialContext.....
 Object objHome = initialContext.lookup("itso/ejb35/cmp/BankAccount");
 BankAccountHome bankHome = (BankAccountHome)
 javax.rmi.PortableRemoteObject.narrow(objHome,BankAccountHome.class);
 System.out.println("Calling BankAccount finder");
 BankAccount acct;
 java.util.Enumeration enum = bankHome.findAccountsWithBalanceGreater
 (new java.math.BigDecimal(20.00));
 while (enum.hasMoreElements()) {
 acct = (BankAccount)javax.rmi.PortableRemoteObject.narrow
 (enum.nextElement(),BankAccount.class);
 System.out.println("Account: id="
 + ((BankAccountKey)acct.getPrimaryKey()).accID
 + " balance=" + acct.getBalance());
 acct.setBalance(acct.getBalance().add(new java.math.BigDecimal("1")));
 }
 System.out.println("End of list");
 } catch(Exception ex) { ex.printStackTrace(); }
 }
}
```

Figure 13-3 Using a finder method that returns a list of entity beans

## Coding technique

Note that this coding technique is not very efficient. Every get method retrieves the entity bean from the database table, and every set method retrieves the bean again and then updates the table. Although the data will be in the database buffer pool, it is still a big overhead.

Using access beans or an intermediate stateless session bean improves performance and reduced the number of database accesses.

## Simple servlet

In this example we develop a simple Web page to enter a customer number. The HTML input page schedules a servlet for processing. The servlet accesses the entity bean to retrieve the data. The entity bean is passed to the JSP to display the HTML output page. Figure 13-4 shows the output of a sample run.

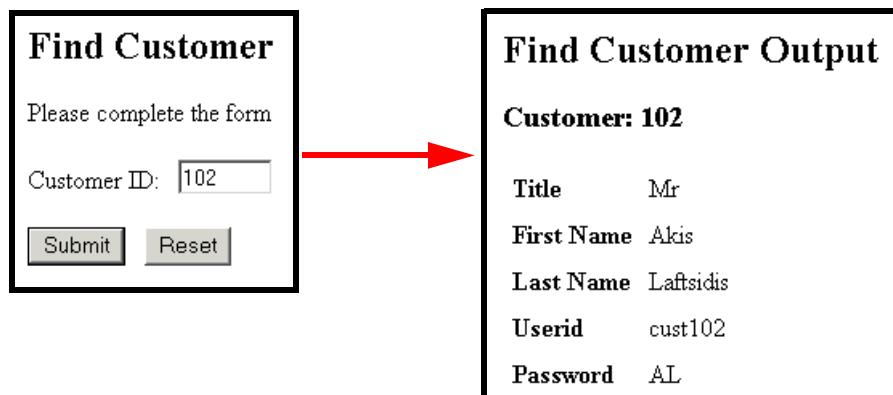


Figure 13-4 Customer find servlet run

### HTML input page

The HTML page captures the user input of the customer ID and invokes the servlet (Figure 13-5).

```
<HTML>
 <HEAD><TITLE>Find Customer</TITLE></HEAD>
<BODY>
 <H2>Find Customer</H2>
 <FORM METHOD="post" ACTION="/servlet/itso.ejb35 cmp.servlet.CustomerFind">
 <P>Please complete the form</P>
 <P>Customer ID:
 <INPUT TYPE="text" NAME="CustomerID" ID="CustomerID" SIZE="6"
 MAXLENGTH="6">
 <P> <INPUT TYPE="submit" NAME="Submit" ID="Submit" VALUE="Submit">
 <INPUT TYPE="reset" NAME="Reset" ID="Reset" VALUE="Reset">
 </FORM>
</BODY>
</HTML>
```

Figure 13-5 Customer find HTML input page: CustomerFind.html

## Creating the servlet

The servlet serves as the EJB client in our example. After reading the customer ID from the input page, it gets the initial naming context, retrieves the home for the customer bean, finds the given customer, stores the bean in the request block, and calls the JSP (Figure 13-6).

```
package itso.ejb35.cmp.servlet;
import itso.ejb35.cmp.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class CustomerFind extends HttpServlet {
 public void doPost(HttpServletRequest request, HttpServletResponse
 response) throws ServletException, IOException {
 performTask(request, response);
 }
 public void performTask(HttpServletRequest request, HttpServletResponse
 response) throws ServletException, IOException {
 // Read the input parameter from the HTML Form
 String id = request.getParameter("CustomerID");
 int custId = (new Integer(id)).intValue();
 // Set the results page URL
 String url = "/ejb/cmpservlet/CustomerFind.jsp";
 try {
 //.....getInitialContext.....
 javax.naming.InitialContext initialContext =
 new javax.naming.InitialContext();
 Object objHome = initialContext.lookup("itso/ejb35/cmp/Customer");
 CustomerHome customerHome = (CustomerHome)
 javax.rmi.PortableRemoteObject.narrow(objHome, CustomerHome.class);
 // Find the customer
 Customer customer = customerHome.findByPrimaryKey
 (new CustomerKey(custId));
 // Forward to the results JSP
 request.setAttribute("customer", customer);
 getServletConfig().getServletContext().getRequestDispatcher(url).
 forward(request, response);
 } catch (Exception e) {
 System.out.println("Exception thrown for customer with id:" + id);
 e.printStackTrace(); }
 }
}
```

Figure 13-6 Customer find servlet

## Result JSP

The result page is a JSP that receives the customer object reference in the request object and displays the main properties of the customer. For simplicity, the customer address is not displayed (Figure 13-7).

```
<HTML>
 <HEAD><TITLE>Find Customer Output</TITLE></HEAD>
<BODY>
 <H2>Find Customer Output</H2>
 <jsp:useBean id="customer" type="itso.ejb35 cmp.Customer"
 scope="request"/>
 <H3>Customer: <%= customer.getCustomerID() %></H3>
 <TABLE border="0" cellpadding="4">
 <TR>
 <TD>Title </TD> <TD><%= customer.getTitle() %></TD>
 </TR><TR>
 <TD>First Name</TD> <TD><%= customer.getFirstName() %></TD>
 </TR><TR>
 <TD>Last Name</TD> <TD><%= customer.getLastName() %></TD>
 </TR><TR>
 <TD>Userid </TD> <TD><%= customer.getUserID() %></TD>
 </TR><TR>
 <TD>Password </TD> <TD><%= customer.getPassword() %></TD>
 </TR>
 </TABLE>
</BODY>
</HTML>
```

Figure 13-7 Customer find JSP output page: CustomerFind.jsp

## Testing the servlet in VisualAge for Java

For testing in VisualAge for Java we have to place the HTML input page (CustomerFind.html) and the JSP output page (CustomerFind.jsp) into a suitable directory. Notice in the servlet code that we called the JSP as /ejb/cmpservlet/CustomerFind.jsp, therefore we have to put the files into a matching subdirectory structure:

```
<IBMVJava>\ide\project_resources\IBM WebSphere Test Environment\hosts\
 \default_host\default_app\web\ejb\cmpservlet\
```

Start the WebSphere Test Environment. Configure the class path of the servlet engine with your project and the IBM EJB Tools project (see Figure 6-3 on page 84). Optionally set *Load generated servlet externally* to bypass importing of the JSP (see Figure 6-2 on page 81). Start the servlet engine.

Start the persistent name server and the EJB server containing the CMP\_Entity group with the customer bean.

Now open a browser and request the page:

<http://localhost:8080/ejb/cmpservlet/CustomerFind.html>

Enter a customer number and click *Submit*. You should receive the output as shown in Figure 13-4 on page 250.

## Externalizing strings

Some of the strings we use in the servlet `performTask` method contain important information, such as the PROVIDER\_URL, the INITIAL\_CONTEXT\_FACTORY, the URL of the results page, and the JNDI name of the customer bean.

These strings would rather be placed into a properties file, making it easier for a developer to maintain the code. This is easily done in VisualAge for Java by selecting the servlet class and from the context menu invoking *Externalize Strings*. Select which strings to externalize, change the key name, and enter the name of a resulting properties file (Figure 13-8).

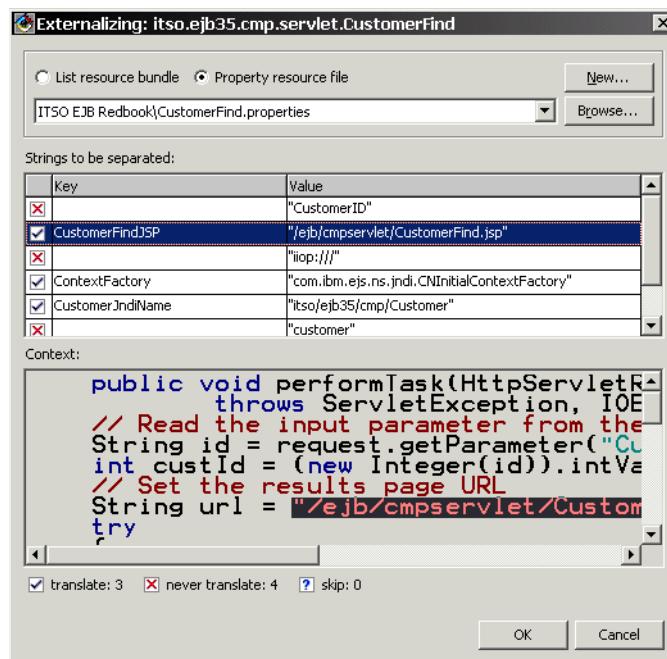


Figure 13-8 Externalizing strings into a properties file

This process updates the servlet code and replaces the references to the externalized strings to look up the string in the properties file.

The CustomerFind.properties file has this content:

```
CustomerFindJSP = /ejb/cmpservlet/CustomerFind.jsp
ContextFactory = com.ibm.ejs.ns.jndi.CNInitialContextFactory
CustomerJndiName = itso/ejb35/cmp/Customer
```

The properties file is stored in the resources directory of the project. Copy it to the servlets directory of the Web application:

```
from: <IBMVJava>\ide\project_resources\ITSO EJB Redbook
to: <IBMVJava>\ide\project_resources\IBM WebSphere Test Environment\
 hosts\default_host\default_app\servlets
```

You have to stop and start the servlet engine to make the new code work. If the file is not found then the static variable in the servlet is not initialized. Adding this line of code to the servlet's performTask method may help:

```
if (resCustomerFind == null) resCustomerFind =
 java.util.ResourceBundle.getBundle("CustomerFind");
```

## Using access beans

We saw in the previous sections the basic facilities to access enterprise beans from applications and servlets. However, you may have noticed that the code that has to be written is not simple for a developer who is not familiar with EJBs.

In Chapter 11. “Access beans” on page 205, we introduced the concept of access beans as a means to simplify coding and to increase performance by caching the data of an enterprise bean in a client JavaBean. Let’s explore now how client coding is changed when using access beans.

### Customer access bean

To simplify our examples we will use the customer access bean created in “Creating an access bean with VisualAge for Java” on page 212.

We want to use a copy helper bean for individual access, and the rowset for the finder method with multiple result beans. Therefore, we generated the rowset access bean, which includes the copy helper.

## Simple test application with access bean

Rewriting the simple test application with the customer access bean produces code as shown in Figure 13-9. To create a customer entity bean we use the constructor with parameters, to retrieve an existing bean we use the default constructor and set the parameter afterwards. The bean is only instantiated when we use the first method.

```
package itso.ejb35.cmp.client;
import itso.ejb35.cmp.*;
public class SimpleAppAB {
 public static void main(java.lang.String[] args) {
 try {
 CustomerAccessBean cust2 =
 new CustomerAccessBean(202, "Mr", "Mike", "Vaughn", "MV", "MV");
 System.out.println("Customer 202 created");
 CustomerAccessBean cust1 = new CustomerAccessBean();
 cust1.setInitKey_customerID("101");
 System.out.println("Customer 101: name="+cust1.getName());
 System.out.println("Customer 202: name="+cust2.getName());
 cust2.setLastName("VaughnX");
 System.out.println("Customer 202: name="+cust2.getName());
 cust2.commitCopyHelper();
 cust2.refreshCopyHelper();
 System.out.println("Customer 202: name="+cust2.getName());
 (cust2.getEJBRef()).remove();
 System.out.println("Customer 202: deleted");
 } catch(Exception ex) {
 ex.printStackTrace();
 }
 }
}
```

Figure 13-9 Simple test application using access bean

Note the call to the getName method, which concatenates the title, first name, and last name fields. We did add this method to the access bean when we created the access bean (see Figure 11-4 on page 213).

However, calling this method retrieves the value that was copied when the access bean is instantiated. So after updating the last name, the concatenated name is not changed. Only after commit and refresh of the access bean is the name attribute changed.

**Tip:** Be careful with the attributes of the access bean, only real properties of the bean should be copied to the access bean.

## Using a rowset access bean for a finder method

This example assumes that we have a custom finder in the customer entity bean that retrieves customer by partial last name (we also defined a custom finder to retrieve a customer by user ID):

```
public interface CustomerBeanFinderHelper {
 public String findByLastNameWhereClause = "T1.lastname like ?";
 public String findByUserIDWhereClause = "T1.userid = ?";
}
```

The matching home interface is:

```
public Enumeration findByLastName(String aName)
 throws javax.ejb.FinderException, java.rmi.RemoteException;
public Customer findByUserID(String aUserID)
 throws javax.ejb.FinderException, java.rmi.RemoteException;
```

If you add custom finders you have to regenerate the access bean and the code.

We can write a sample client that uses the rowset access bean, filled with the results of the findByLastName method (Figure 13-10).

```
package itso.ejb35.cmp.client;
import itso.ejb35.cmp.*;
public class UsingFinderAB {
public static void main(java.lang.String[] args) {
 try {
 CustomerAccessBeanTable custrows = new CustomerAccessBeanTable();
 CustomerAccessBean cust = new CustomerAccessBean();
 System.out.println("Calling Customer finder");
 custrows.setCustomerAccessBean(cust.findByLastName("%d%"));
 try {
 for (int i=0; i < custrows.numberofRows(); i++) {
 cust = custrows.getCustomerAccessBean(i);
 System.out.println("Customer " + cust.getCustomerID() + " "
 + cust.getLastName());
 }
 }
 catch (IndexOutOfBoundsException o) {}
 System.out.println("End of list");
 } catch(Exception ex) { ex.printStackTrace(); }
}
```

Figure 13-10 Client using a rowset access bean

Notice the coding technique. You have to instantiate both the rowset bean (*CustomerAccessBeanTable*) and the copy helper bean (*CustomerAccessBean*).

The finder method of the home interface becomes a method of the copy helper. The resulting enumeration is used to fill the rowset. After filling the rowset you can loop through access beans until an *IndexOutOfBoundsException* is thrown. It is better to end the loop with the number of rows as coded.

## Using a copy helper in a servlet

Let's now go back to our servlet example and see how the code can be simplified by using access beans.

### Changing the servlet

Instead of using the remote reference of the customer bean we use the customer access bean. To retrieve the customer, we code:

```
CustomerAccessBean customerAccessBean = new CustomerAccessBean();
customerAccessBean.setInitKey_customerID(id);
```

We access all the properties of the customer bean through the access bean and we place the access bean into the request block for the JSP:

```
request.setAttribute("customer", customerAccessBean);
```

Figure 13-11 shows the revised servlet code using the copy helper.

```
public class CustomerFindAB extends HttpServlet {
 public void performTask(HttpServletRequest request, HttpServletResponse
 response) throws ServletException, IOException {
 // Read the input parameter from the HTML Form
 String id = request.getParameter("CustomerID");
 // Set the results page URL
 String url = "/ejb/cmpservlet/CustomerFindAB.jsp";

 // Instantiate the access bean and initialize it with the customerID
 CustomerAccessBean customer = new CustomerAccessBean();
 customer.setInitKey_customerID(id);

 // Forward to the results JSP
 request.setAttribute("customer", customer);
 getServletConfig().getServletContext().getRequestDispatcher(url).
 forward(request, response);
 }
}
```

Figure 13-11 Servlet using a copy helper access bean

As you can see, the `performTask` method looks much simpler now than in the previous example. There is no initial naming context, home, or remote object references. The developer does not have to know about the enterprise bean model. This is the main advantage when using access beans.

## Changing the JSP

The only change in the JSP is in the `useBean` tag, which has to refer to the access bean instead:

```
<jsp:useBean id="customer" type="itso.ejb35.cmp.CustomerAccessBean"
scope="request"/>
```

All the get and set methods are identical for the remote reference as for the access bean.

## Testing the servlet with the copy helper

Create copies of the HTML file (`CustomerFindAB.html`) to invoke the new servlet (`itso.ejb35.cmpservlet.CustomerFindAB`) and of the JSP (`CustomerFindAB.jsp`) to use the access bean.

Test the servlet using the instructions in “Testing the servlet in VisualAge for Java” on page 252, but using the new HTML file:

```
http://localhost:8080/ejb/cmpservlet/CustomerFindAB.html
```

Also it may not be noticeable in such a small test, but the servlet and JSP do execute faster, because there are fewer remote method calls to the entity bean. Remote method calls are reduced greatly when using copy helpers instead of directly accessing the enterprise bean. We will see this in more detail in “Client comparison” on page 271.

## Using access beans in an applet or application

Although most Web applications are done with servlets and JSPs, it is possible to connect to enterprise beans from applications and applets.

Access beans are very well suited for visual programming, because they behave like JavaBeans. Also the programmer does not have to deal with EJB semantics, but connects access beans to GUI components. All EJB access is encapsulated in the access beans and makes GUI programming relatively simple.

Figure 13-12 shows the visual composition of a customer maintenance applet (`itso.ejb35.gui.CustomerApplet`).

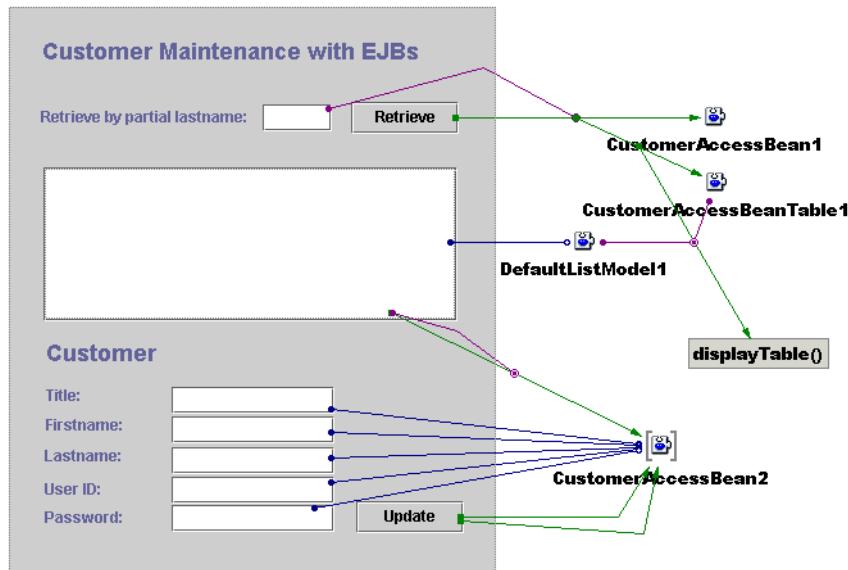


Figure 13-12 Applet design with access beans

This applet uses a `CustomerAccessBean` to invoke the `findByLastName` finder method, which returns an `Enumeration` of customer access beans. This result is stored in a `CustomerAccessBeanTable`, which is the rowset created for the customer entity bean.

The list of customers stored in the `CustomerAccessBeanTable` is displayed in a Swing `JList`. Because there is no direct attribute that can be connected to the `JList` default model, a tailored method named `displayTable` is used to add the customer access beans to the `JList` model:

```

public void displayTable(java.util.Enumeration enum,
 javax.swing.DefaultListModel model) {
 model.clear();
 while (enum.hasMoreElements()) {
 model.addElement(enum.nextElement());
 }
 return;
}

```

The `JList` bean calls the `toString` method on each object to display a list entry. We added a tailored `toString` method to the `CustomerAccessBean`:

```

public String toString() {
 try { return getCustomerID() + " " + getName(); }
 catch (Exception e) { return "aCustomerObject"; }
}

```

When an entry in the list is selected, that customer is stored in the second CustomerAccessBean, which is a variable. The properties of that bean are displayed in appropriate Swing text fields to show the customer's details.

The data in the text fields can be changed. It is propagated back to the CustomerAccessBean through the keyReleased event of each text field.

Clicking the *Update* button invokes both the commitCopyHelper and refreshCopyHelper methods of the CustomerAccessBean. This action updates the database tables and refreshes the data for the display in the list.

A sample run of the applet is shown in Figure 13-13.

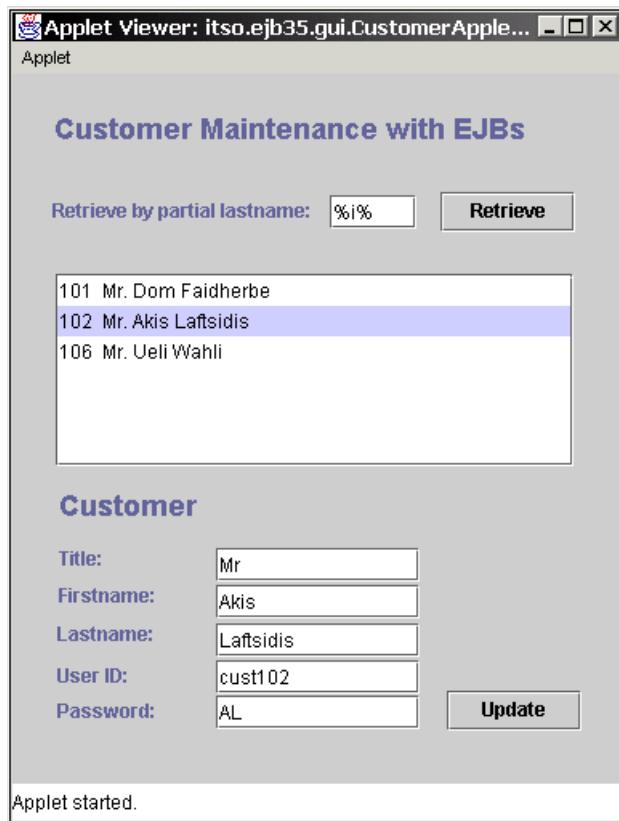


Figure 13-13 Applet sample run

## Java applets and security

JDK 1.2 introduces higher security requirements than JDK 1.1. You will get an error message when an applet tries to access an enterprise bean. This is controlled by a file named `java.policy`, which resides in:

```
<VAJ-HOME>\ide\program\lib\security\java.policy
<WAS-HOME>\jdk\jre\lib\security\java.policy
```

To make the applet work in VisualAge for Java update the `java.policy` file with these additional permissions:

```
// EJB access
permission java.util.PropertyPermission "*", "read,write";
permission java.net.SocketPermission "xxxxx", "connect"; <== hostname
permission java.net.SocketPermission "LOCALHOST", "connect";
permission java.lang.RuntimePermission "modifyThreadGroup";
permission java.lang.RuntimePermission "modifyThread";
```

## Using a session facade to entity beans

We have seen so far how clients can access enterprise beans directly or using access beans. We saw that using access beans simplifies the client code and accelerates its execution.

The trade-off of simplicity and speed is freedom. Using access beans limits the flexibility of the design and restricts the developer from using other techniques. In this section, we show how EJB client applications can use session beans instead.

### Facade session beans

A common approach when designing EJB applications with entity and session beans is to use a facade in front of entity beans that protects the persistent data layer and controls all the client access. Such a facade can be built using a stateless session bean that provides:

- ▶ In the remote interface: the business methods of the entity bean
- ▶ In the home interface: the create and finder methods of the entity home

The advantages of using facade session beans are:

- ▶ The persistence model is hidden from the client. Entity beans are viewed as general-purpose data sources.
- ▶ Session beans can tie together multiple data sources and act as single point of entry to the business data layer.

- Centralized control of transactions and security is delegated to the session facade layer and managed more efficiently for the application.

Figure 13-14 illustrates how session beans can act as a facade layer to entity beans in front of clients.

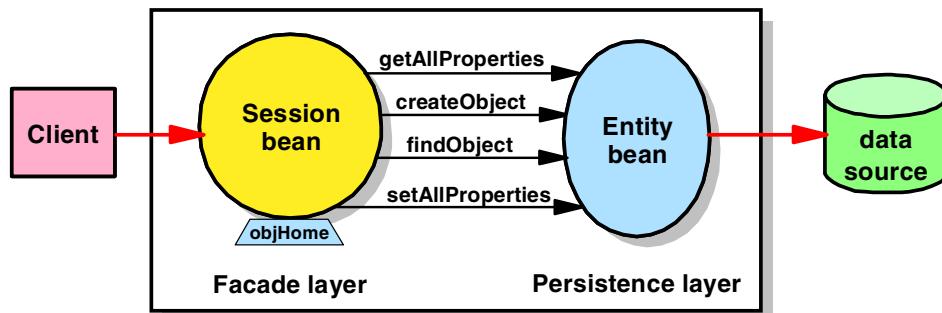


Figure 13-14 Session facade to entity beans

## Facade session design

The facade session bean can be easily built using the EJB development tools in VisualAge for Java. We recommend that stateless session beans are used, because they are lightweight and have better performance.

For a session bean to serve as a facade to entity beans, it should have these members:

### ***objHome***

This is a private field that caches the home of the entity bean that the session bean will serve (instance variable). Caching the home object reference improves the performance. The session bean should also have a `getObjHome` method that performs a lazy initialization and returns the home object.

There is no field caching the entity bean. This is provided by the client instead as a method argument.

### ***createObject***

This is a public method that calls a `create` method on the home of the entity bean, such as `objHome.create(...)`.

- There must be one `createObject` method for each `create` method defined in the home interface of the entity bean.
- It must be promoted to the remote interface of the facade session bean.
- It must return the remote entity object.

### ***findObject***

This is a public method that calls a finder method on the objHome.

- ▶ There must be one `findObject` method for each finder method defined in the home interface of the entity bean.
- ▶ It must be promoted to the remote interface of the session bean.
- ▶ The method calling the `findByPrimaryKey` returns the remote entity object. A finder method may return an enumeration or a single object.

### ***getAllProperties***

This is a public method that is used for retrieving all the properties of the entity bean. It calls the getter methods for all the entity bean properties and stores them in a *data container* that is returned to the client.

- ▶ It takes a remote entity object as argument.
- ▶ It can return either a Hashtable or a JavaBean containing all the properties (data container). JavaBeans can be used easily in JSPs.
- ▶ It must be promoted to the remote interface of the session bean.
- ▶ It must throw a `RemoteException` on failure.

### ***setAllProperties***

This is a public method that is used for setting all the properties of the entity bean. It reads all the entity bean properties from a *data container* sent by the client and calls the corresponding setter methods on the entity bean.

- ▶ It takes two arguments, the remote entity object, and the data container (Hashtable or JavaBean)
- ▶ It must be promoted to the remote interface of the session bean.
- ▶ It must throw a `RemoteException` on failure.

**Attention:** We do not store the remote entity object, as we do with the home. This is done because the facade session bean is stateless, and a bean instance will not always serve the same client. All the bean instances serving the same entity bean will always use the same home but not the same entity bean instance (remote entity object).

Therefore, the client uses the remote entity object that is returned by a `createObject` or a `findObject` method and sends it back to the session bean every time it invokes a method.

## Servlet example using a session facade

We now implement the servlet example by defining a session facade and then use the session bean in the servlet.

### Create the session facade bean

We create the session bean the same way we did in “Creating a session bean” on page 174. The steps are given below:

1. In the EJB page, select the EJB group containing the stateless session bean, for example, **Stateless\_Session**. Usually session beans should not be in the same group as entity beans.
2. Select *Add -> Enterprise Bean*. Create a new session bean named **CustomerManager**, and store it in the `itso.ejb35.session` package.
3. Click on the *Next* button. Add some useful packages in the import section:  
`java.rmi.*`, `javax.rmi.*`, `javax.ejb.*`, `javax.naming.*`, `java.util.*`,  
`itso.ejb35.cmp.*` (where the customer entity bean is).
4. Click *Finish* to create the bean.
5. Select the newly created bean class in the Types page and add the fields and methods discussed in session facade design.

#### ***customerHome* field**

The `customerHome` field is defined as:

```
private CustomerHome customerHome;
```

#### ***getCustomerHome* method**

This method is used for retrieving the home of the customer bean. It employs the lazy initialization technique.

We define the parameters used for retrieving the initial naming context in the deployment descriptor as environment variables by selecting *Properties* for the `CustomerManager` bean in the Enterprise Beans pane (Figure 13-15).

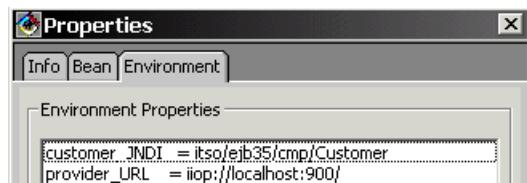


Figure 13-15 Setting the environment variables for the `CustomerManager`

The code of the `getCustomerHome` method is shown in Figure 13-16.

```

private itso.ejb35.cmp.CustomerHome getCustomerHome() {
 if (customerHome == null) {
 try {
 // Get initial context
 Properties prp = getServletContext().getEnvironment();
 String sProviderURL= prp.getProperty("provider_URL");
 String sNameService= "com.ibm.ejs.ns.jndi.CNInitialContextFactory";
 Hashtable htEnv = new Hashtable();
 htEnv.put(javax.naming.Context.PROVIDER_URL, sProviderURL);
 htEnv.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
 sNameService);
 InitialContext ctx = new InitialContext(htEnv);
 //InitialContext ctx = new InitialContext();
 // Get EJBHome for Customer
 Object objHome = ctx.lookup(prp.getProperty("customer_JNDI"));
 customerHome = (CustomerHome) PortableRemoteObject.narrow(objHome,
 CustomerHome.class);
 } catch (Exception e) {
 System.out.println("Cannot retrieve the EJBHome for Customer");
 }
 return customerHome;
 }
}

```

Figure 13-16 Session facade *getCustomerHome* method

### ***createCustomer* method**

This method calls the create method on the customer home passing all the parameters entered by the user (Figure 13-17).

```

public Customer createCustomer(int id, String title, String firstName,
 String lastName, String userid, String password) {
 Customer customer = null;
 try {
 customer = getCustomerHome().create(id, title, firstName,
 lastName, userid, password);
 } catch (Exception e) {
 System.out.println("Cannot create a new EJB for customer" + id);
 }
 return customer;
}

```

Figure 13-17 Session facade *createCustomer* method

### ***findCustomer method***

This method calls the findByPrimaryKey method on the customer home passing all the parameters entered by the user (Figure 13-18).

```
public Customer findCustomer(int id) {
 Customer customer = null;
 try {
 customer = getCustomerHome().findByPrimaryKey(new CustomerKey(id));
 } catch (Exception e) {
 System.out.println("Cannot find the customer: " + id);
 }
 return customer;
}
```

Figure 13-18 Session facade *findCustomer* method

### ***getAllProperties method***

This method reads all the properties of the customer bean and places them in a hash table. As you can see, it does not read the address field. The client should invoke the getAddress method on the customer object instead. This is done for simplicity (Figure 13-19).

```
public Hashtable getAllProperties(Customer customer) throws RemoteException{
 Hashtable ht = new Hashtable();
 ht.put("customerID", new Integer(customer.getCustomerID()));
 ht.put("title", customer.getTitle());
 ht.put("firstName", customer.getFirstName());
 ht.put("lastName", customer.getLastName());
 ht.put("userID", customer.getUserID());
 ht.put("password", customer.getPassword());
 return ht;
}
```

Figure 13-19 Session facade *getAllProperties* method

### ***setAllProperties method***

This method reads first all the parameters sent by the client in a hash table and then sets the corresponding properties. Again here, the method does not deal with the address field. The client should invoke the setAddress method on the customer object instead (Figure 13-20).

```

public void setAllProperties(Customer customer, Hashtable ht)
 throws RemoteException {
 String localTitle = (String) ht.get("title");
 String localFirstName = (String) ht.get("firstName");
 String localLastName = (String) ht.get("lastName");
 String localPassword = (String) ht.get("password");
 String localUserID = (String) ht.get("userID");
 // Set the property, only if parameter not null
 if (localTitle != null) customer.setTitle(localTitle);
 if (localFirstName != null) customer.setFirstName(localFirstName);
 if (localLastName != null) customer.setLastName(localLastName);
 if (localUserID != null) customer.setUserID(localUserID);
 if (localPassword != null) customer.setPassword(localPassword);
}

```

Figure 13-20 Session facade setAllProperties method

### Promote

Add the last four methods to the remote interface (*Add To -> EJB Remote Interface*). Now, the session facade bean is ready.

### Create the servlet

We can copy the existing CustomerFind servlet as **CustomerFindSF** and recode the performTask method using the CustomerManager session facade.

Figure 13-21 shows the performTask method of the servlet.

We first create an initial naming context and use it for retrieving the home interface of the CustomerManager. Then, we will create a bean instance and get the remote interface of the facade bean.

The next step is to invoke the findCustomer method on the remote interface by passing the customer ID entered by the user. We use the returned customer object as an argument and invoke the getAllProperties method. At the end, we place the returned hash table within the request object and forward the call to the results JSP.

In a real application the URL of the results page, the PROVIDER\_URL and the JNDI name of the CustomerManager bean should be externalized to a properties file.

The performTask method we use here is not so simple as in the previous example. We have to use an initial naming context as well as the home and remote object references of the session bean. However, this approach gives us better control over the EJB model and more freedom to modify and enhance our application design.

```

public class CustomerFindSF extends HttpServlet {
 public void performTask(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException {
 // URL for the results page
 String url = "/ejb/cmp servlet/CustomerFindSF.jsp";
 // Read the input parameters from the HTML Form.
 String id = request.getParameter("CustomerID");
 int custId = (new Integer(id)).intValue();
 try {
 // Create a CustomerManager instance
 //java.util.Properties properties = new java.util.Properties();
 //properties.put(javax.naming.Context.PROVIDER_URL, "iiop:///");
 //properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
 // "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
 //javax.naming.InitialContext initialContext =
 //new javax.naming.InitialContext(properties);
 javax.naming.InitialContext initialContext =
 new javax.naming.InitialContext();
 Object objHome = initialContext.lookup
 ("itso/ejb35/session/CustomerManager");
 CustomerManagerHome customerMgrHome = (CustomerManagerHome)javax.rmi.
 PortableRemoteObject.narrow(objHome,CustomerManagerHome.class);
 CustomerManager customerManager = customerMgrHome.create();
 // Find the customer
 Customer customer = customerManager.findCustomer(custId);
 java.util.Hashtable ht = customerManager.getAllProperties(customer);
 // Store hashtable and call JSP
 request.setAttribute("custht", ht);
 getServletConfig().getServletContext().getRequestDispatcher(url).
 forward(request, response);
 }
 catch (Exception e) {
 System.out.println("Cannot find customer with id: " + id);
 }
 }
}

```

*Figure 13-21 Servlet using a session facade bean*

## Create the JSP

The JSP to display the customer has to work with the hash table instead of a customer bean or customer access bean. The coding is very similar, however (Figure 13-22).

```

<HTML>
 <HEAD><TITLE>Find Customer Output</TITLE></HEAD>
<BODY>
 <H2>Find Customer Output with Session Facade</H2>
 <jsp:useBean id="custht" type="java.util.Hashtable" scope="request"/>
 <H3>Customer: <%= custht.get("customerID") %></H3>
 <TABLE border="0" cellpadding="4">
 <TR>
 <TD>Title </TD> <TD><%= custht.get("title") %></TD>
 </TR><TR>
 <TD>First Name</TD> <TD><%= custht.get("firstName") %></TD>
 </TR><TR>
 <TD>Last Name </TD> <TD><%= custht.get("lastName") %></TD>
 </TR><TR>
 <TD>Userid </TD> <TD><%= custht.get("userID") %></TD>
 </TR><TR>
 <TD>Password </TD> <TD><%= custht.get("password") %></TD>
 </TR>
 </TABLE>
</BODY>
</HTML>

```

Figure 13-22 JSP using a session facade bean: CustomerFindSF.jsp

**Tip:** For better performance, we recommend that you cache the home of the CustomerManager bean in a private static variable in the FindCustomer servlet:

```
private static CustomerManagerHome customerMgrHome;
```

The initialization of this class variable should be done in the `init` method of the servlet:

```
public void init() throws ServletException {
 getCustomerMgrHome();
}
```

Where, the `getCustomerMgrHome` performs a lazy initialization of the home, similar to Figure 13-16 on page 265. This would simplify the body of the `performTask` method.

This technique ensures that none of the servlet threads will have to find and initialize the home interface of the facade bean; instead they will access it in memory and reuse it.

## Using an access bean for the session facade

We can combine the session facade model with access beans by using a Java wrapper for the CustomerManager bean. (This approach is questionable and there is hardly a reason for the session facade when using access beans.) The performTask method becomes much simpler; however, this requires changes to our approach:

- ▶ The find and create methods in the generated CustomerManagerAccessBean return a CustomerAccessBean, and not a Customer.
- ▶ The getAllProperties and setAllProperties methods of the session bean expect a remote customer object.

Therefore, we have to change the servlet logic to deal with the CustomerAccessBean. The getAllProperties and setAllProperties methods of the session bean can be omitted completely, because the data is now local in the CustomerAccessBean (Figure 13-23).

```
public class CustomerFindSFAB extends HttpServlet {
 public void performTask(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException {
 // Read the input parameter from the HTML Form
 String id = request.getParameter("CustomerID");
 int custId = (new Integer(id)).intValue();
 // Set the results page URL
 String url = "/ejb/cmpservlet/CustomerFindAB.jsp";

 // Instantiate the access bean and initialize it
 CustomerAccessBean customer = null;
 CustomerManagerAccessBean customerMgr = new CustomerManagerAccessBean();
 try {
 customer = customerMgr.findCustomer(custId);
 } catch (Exception e) {
 System.out.println("Cannot find customer with id: "+id);
 }

 // Forward to the results JSP
 request.setAttribute("customer", customer);
 getServletConfig().getServletContext().getRequestDispatcher(url).
 forward(request, response);
 }
}
```

Figure 13-23 Servlet using access beans and a session facade bean

Note that we can use the same JSP we developed for the access bean solution, because we store the CustomerAccessBean in the request block for the JSP.

## Testing the servlet with the facade bean

You can test the session facade bean by itself using the EJB test client. Find the home of the facade bean, instantiate it using the create method, and then execute createCustomer and findCustomer methods. In the remote customer objects execute the getAllProperties method and inspect the resulting hash table.

To test the servlet, create copies of the HTML file (`CustomerFindSF.html`) to invoke the new servlet (`itso.ejb35.cmpservlet.CustomerFindSF`) and of the JSP (`CustomerFindSF.jsp`) to use the access bean.

Test the servlet using the instructions in “Testing the servlet in VisualAge for Java” on page 252, but use the new HTML file:

```
http://localhost:8080/ejb/cmpservlet/CustomerFindSF.html
```

You will notice that the servlet executes as fast as when we used access beans instead. Again, the number of remote method calls is reduced greatly by the use of the facade bean. The servlet performance can further be improved by caching the EJB home of the customer manager bean, as discussed in the tip given in the previous section.

After trying these three different approaches, you may wonder which one to use when accessing enterprise beans from a client. In the next section, we will compare these different models and give some guidelines that you can use, when designing your application.

## Client comparison

After presenting the different ways of accessing enterprise beans from a client application (a servlet in our examples), let's now see the advantages and disadvantages of using them.

### Direct access

This is the straight forward way for a client to access an enterprise bean. The home of the bean should be cached in a static variable and be initialized in the init method of the servlet. This gives good performance. However, the result of each method invocation performed by the client is a remote call. Even if the client is a servlet running in the same application server as the EJB container (same JVM), all the method calls are routed through the TCP/IP stack. This of course has an impact on the performance of the application.

## Pros and cons

The advantages of using this approach are:

- ▶ Fast to code
- ▶ No need to create extra components
- ▶ No extra load generated on the client or server side
- ▶ The enterprise bean implementation is not modified

The disadvantages are:

- ▶ Poor performance
- ▶ Requires EJB programming in the client code
- ▶ The client developer has to know how to use EJBs

## Access beans

This is a better way to access enterprise beans. The home is cached in the access bean, and for copy helpers and rowsets the properties of the bean are cached as well. This improves the performance by eliminating many remote calls. Though, the developer should be careful using always the local cache of the bean properties. It is not certain, if they have not been updated in the meantime by another application.

The number of the remote method calls over the network is decreased greatly, as the access beans perform all the get or set methods in a single call. For example, if we assume that a customer bean has ten properties, retrieving them generates only one remote method call instead of ten. However, this requires the modification of the enterprise bean implementation; the methods `_copyToEJB` and `_copyFromEJB` are added to the bean class.

Access beans have also some limitations when used together with associations. If you create an access bean for an enterprise bean involved in an association and the association has been made navigable, the navigation method returns an access bean which does not exist, corresponding to the enterprise bean at the other side of the association.

## Pros and cons

The advantages of using access beans are:

- ▶ Very good performance
- ▶ Simple to use
- ▶ The client developer uses EJBs in the same manner as JavaBeans

The disadvantages are:

- ▶ Extra components have to be created
- ▶ The enterprise bean implementation is modified

- ▶ Very close design that does not allow use of other models
- ▶ You have to use access beans for all the beans in an application

## Facade beans

This is an alternative approach to access beans. It is applicable when accessing entity beans. The entity bean is accessed via a stateless session bean that caches the home and provides access to all the properties of the bean in a single method. This also does not require the modification of the bean class, as required when using access beans.

The performance is also very good here, as the number of the remote method calls is limited. Actually, all the method calls between the facade bean and the entity bean are made locally (through the local pipes), as both of them are in the same container.

However, the facade approach requires the creation of a session bean for each set of entity beans we access in one application. This requires more development time and system resources on the application server, because more beans are running in the container. Additionally, the client code is not so simple as when using access beans, because a session bean instance must be used for each method call.

However, the code here can be simplified by using an external Java class (a JavaBean) that retrieves and caches the home interface in a class variable instead. This increases the performance and enables code reuse, simplifying the development of all the facade beans that are needed for the entity beans.

### Pros and cons

The advantages of using facade beans are:

- ▶ Very good performance
- ▶ Open design to other models
- ▶ The enterprise bean implementation is not modified

The disadvantages are:

- ▶ Extra development time
- ▶ More load in the EJB container
- ▶ Requires EJB programming in the client code

In Chapter 23. “Best practices for EJBs” on page 443, we describe that using facade session beans to entity beans is a better architecture approach than using access beans. This way we have better control over the persistence data layer and we manage all the security and the transaction issues in a more efficient and organized manner.

**Attention:** The EJB specification restricts a session bean to declare static variables. The reason is that static variables (class variables) are maintained within the current JVM. If another EJB of the same type uses the same static field inside another container running in a different machine, then the value of the static variable will be different. Therefore, we are forced to use instance variables for caching the home object.

When using an external JavaBean as a utility class, we can then cache the home in a static variable. Therefore, the home is retrieved by the utility bean only once in each JVM and all the facade bean instances use the same reference. In our example, each bean instance has to retrieve a separate reference, the first time the client invokes a method.

## Recapitulation of approaches

Figure 13-24 demonstrates what happens under the scenes, when a client accesses an entity bean:

- ▶ Directly
- ▶ Through an access bean
- ▶ Through a session facade bean

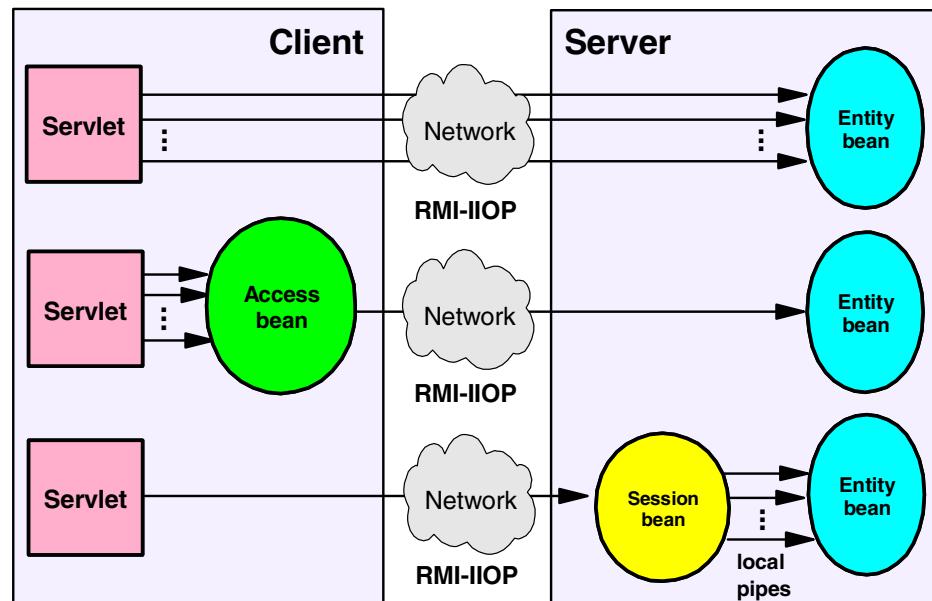


Figure 13-24 Client comparison

The client side in the figure represents the application server where the servlet engine resides. On the other hand, the server side represents the application server where the EJB container resides. Although these two components should be in the same application server for performance reasons, here we assume a more general case that can occur when, for example, a firewall is placed between the presentation and business logic for security reasons.

**Note:** This comparison applies also when the servlet engine resides in the same application server with the EJB container. As we mentioned before, all method calls between a servlet and an EJB are routed through TCP/IP.

## Summary

In this chapter we presented the possible ways a client application can access and use enterprise beans. We saw all the steps that have to be followed and we gave an example for each different approach.

Use the guidelines given in Table 13-1 when deciding how to design your client application.

*Table 13-1 What client type to choose*

Client type	Yes, when	No, when
<b>Direct access</b>	Simple EJB model is used	Entity bean contains many properties
	Performance is not critical	
<b>Access beans</b>	Simple and fast development is wanted	More complex design model is required
	The application server has limited system resources	Clustered environment with many application servers is used
	Java wrapper is used for accessing session beans	
<b>Facade beans</b>	Scalability is critical	Client has to access session beans
	Open design is applied	Development time is critical
	Better overall performance is required	





# Deployment to WebSphere Application Server Advanced

This chapter provides some useful information and practical guidelines to deploy enterprise beans from VisualAge for Java to WebSphere Application Server.

First, we describe the tasks in VisualAge for Java to generate the deployed code to be installed in WebSphere. Then, we describe how to define and install EJBs in a WebSphere EJB container. Finally, we describe the preparation for client programming and how to run the examples.

**Attention:** To follow the directions given in this chapter, WebSphere Application Server must be installed as described in Appendix A. “Setting up the environment” on page 453 and configured as described in “Setting up the WebSphere execution environment” on page 456.

**See Appendix B. “Early information: deployment to WebSphere Version 4” on page 469 for guidelines on deploying EJBs and applications to WebSphere Version 4.**

# Code deployment from VisualAge for Java

VisualAge for Java, being the development arm of WebSphere, offers you a powerful but simple tooling to deploy enterprise beans and export them to WebSphere.

## Preparation before going any further

Before going any further, we have to prepare the enterprise beans and associated classes to ensure smooth deployment, installation, and testing:

- ▶ Make sure all the getters are marked as read-only methods (as explained in “Actions for members” on page 73).
- ▶ Have the deployed code generated.
- ▶ For all Java classes you created (except those directly linked to the EJBs), generate a serial UID using the class context menu *Generate -> Serial UID*. An example would be the CustomerAddress class.
- ▶ Define the underlying database tables and keys on the target WebSphere system.

The reason behind these additional steps will be explained later, when necessary. They will ensure that you do not have to restart exporting from the beginning.

## Enterprise bean JAR files

There are four different types of JAR files that VisualAge for Java can generate:

<b>EJB JAR</b>	A JAR file with non-deployed beans with their deployment descriptors. The deployed code has to be regenerated by WebSphere.
<b>Deployed JAR</b>	A JAR as above, but with the deployed code. WebSphere will use the deployed code generated by VisualAge for Java. This is the more practical than the EJB JAR, and this option must be used for enterprise beans with inheritance or associations.
<b>Client JAR</b>	A JAR file containing all the classes a client requires to use the enterprise beans.
<b>EJB JAR for CB</b>	A JAR file with non-deployed beans with their deployment descriptors for the Component Broker (CB) component of WebSphere Enterprise. We do not further discuss deployment to CB in this redbook.

## Creating a deployed JAR file

In general, the most practical way to deploy enterprise beans to WebSphere is to have VisualAge for Java generate a deployed JAR:

- ▶ Select the beans (or EJB groups) you want to export, for example, the CMP\_Entity group.
- ▶ Ensure that the deployed code has been generated (use the context menu *Generate Deployed Code*).
- ▶ From the context menu, select *Export -> Deployed JAR* and the Export SmartGuide opens (Figure 14-1).

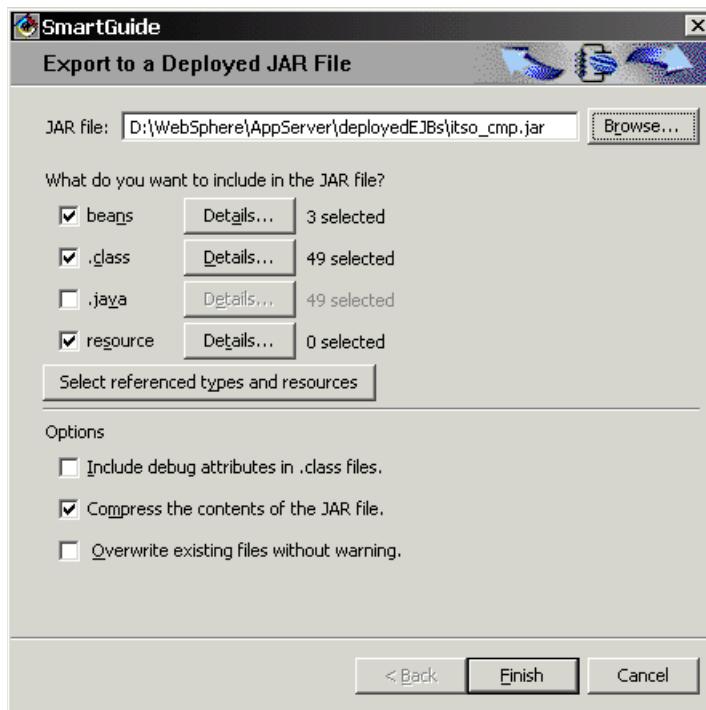


Figure 14-1 Creating a deployed JAR file

- ▶ Specify a location and a JAR file to be created. A suggested directory for deployment to WebSphere is:  

```
<%WAS_HOME%>\deployedEJBs
==> d:\WebSphere\AppServer\deployedEJBs\itso_cmp.jar
```

Where, `<%WAS_HOME%>` is the WebSphere installation directory.
- ▶ The beans, classes, and resources should be automatically selected. Exporting the Java source code is optional.

- ▶ Click on the *Select referenced types and resources* button and let VisualAge for Java search for types and resources used in your code for including them in the exported file:
  - In our case additional classes (for example, CustomerAddress, and InsufficientFundException) and one resource file (CustomerFind.properties, from client programming) are found.
- There may still be classes that are not found automatically:
  - Types referenced to as strings (for example, a class created dynamically and on which you use introspection and reflection).
  - Composers and converters—these are referred to by strings.
- ▶ Click on *Details* for classes and resources to ensure that no type or important resources such as properties file are left out. Failure to include these could prevent deployment.
- ▶ If you do not intend to debug the code using the distributed debugger, uncheck the *Include debug attributes in .class files* check box.
- ▶ Leave the other check boxes to their default values.
- ▶ Click on *Finish* to create the JAR file.

## Non-deployed JAR file

WebSphere can also generate the deployed code, therefore it is possible to create a regular JAR file with VisualAge for Java and let WebSphere do the work.

However, the mapping of the enterprise beans to database tables will be redone by WebSphere and will not match the tailored mapping produced in VisualAge for Java. The names of tables and columns will be based on the entity and property names.

When generating a non-deployed JAR file, you should use this target directory:

```
<%WAS_HOME%>\deployableEJBs
====> d:\WebSphere\AppServer\deployableEJBs\itso_bmp_n.jar
====> d:\WebSphere\AppServer\deployableEJBs\itso_cmp_n.jar
```

WebSphere does not generate deployed code for enterprise beans with inheritance or associations; such beans must be deployed in VisualAge for Java.

The WebSphere deployment tool is most useful for enterprise beans that have been produced by development tools other than VisualAge for Java.

To deploy beans without VisualAge for Java, you have to use JetAce to edit the deployment descriptors and the WebSphere Administrative Console to deploy the beans before installing them.

## Editing the deployment descriptors using JetAce

If you have exported enterprise beans to a EJB jar file or received them in that format, you may deploy the beans using *JetAce*, the deployment editor of WebSphere:

- ▶ Open a command line window and change to <%WAS\_HOME%>\bin. Execute the jetace.bat script and the JetAce window opens.
- ▶ Using the menu *File -> Load*, select the non-deployed JAR file, for example, itso\_cmp\_n.jar. JetAce processes the file and displays the deployment descriptors of the beans (Figure 14-2).

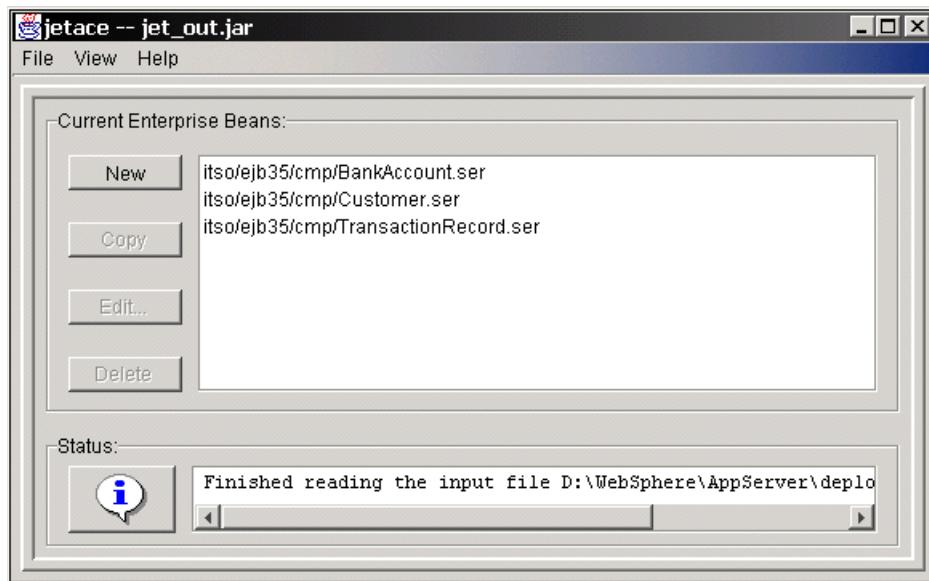


Figure 14-2 WebSphere JetAce tool

- ▶ You will encounter this error message if an enterprise bean was generated with copy helpers:

```
Error: The class itso.ejb35.cmp.Customer could not be loaded
An error: "java.lang.NoClassDefFoundError:
com/ibm/ivj/ejb/runtime/CopyHelper" occurred
```

By default the class path used by JetAce does not include the VisualAge for Java EJB tools. Edit jetace.bat and add this line to the class path setting:

```
set CP=%CP%;%WAS_HOME%\lib\ivjejb35.jar
```

- ▶ From the list of deployment descriptors you can create a new descriptor, or edit, delete, and copy a selected descriptor.

## Editing a deployment descriptor

Click on *Edit* to edit a deployment descriptor (Figure 14-3).

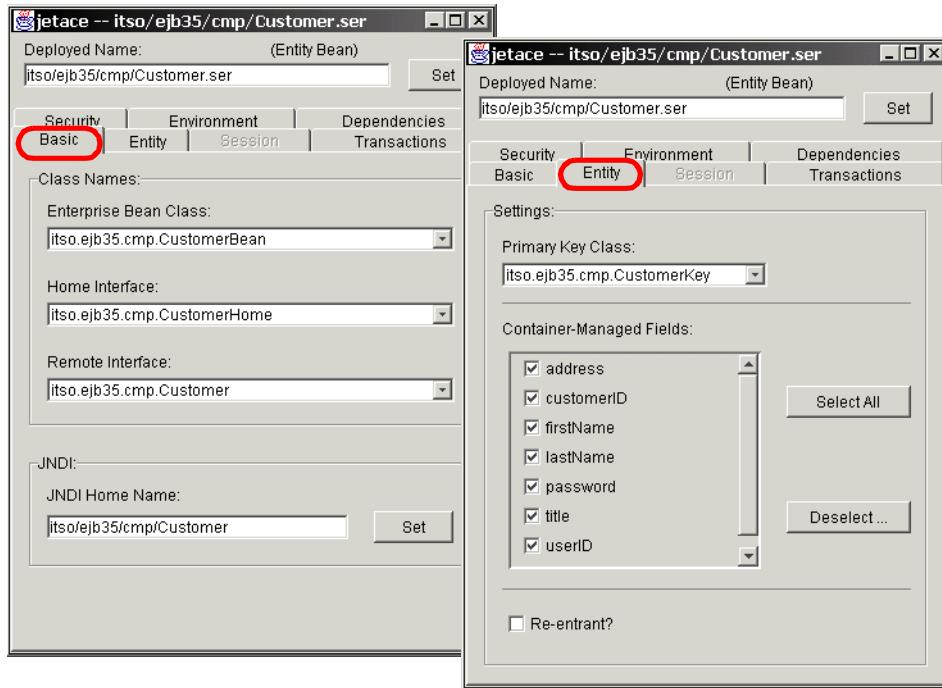


Figure 14-3 Deployment descriptor editor

Many of the values displayed on the other pages come from the settings done in VisualAge for Java using the *Properties* dialog (Figure 7-11 on page 126):

- ▶ On the Transactions page you can set the transaction attribute (for example, TX\_REQUIRED), the isolation level (REPEATABLE\_READ), as well as overwrites of these values for individual methods.
- ▶ On the Security page you set the run-as mode (SYSTEM\_IDENTITY).
- ▶ On the Environment page you can set environment variables.
- ▶ The Dependencies page shows the names of the bean class, and the home and remote interface.

Do not forget to save the JAR file after changing deployment descriptors using *File -> Save*.

For installation of a non-deployed code see “Installing a non-deployed JAR file” on page 286.

# Installation of the deployed code in WebSphere

Once the beans are packaged in a JAR file, you can install them in an EJB container. Make sure WebSphere is started and prepared according to the instructions in “Setting up the WebSphere execution environment” on page 456.

## Set data source for the EJB container

Select the EJB container in the Administrative Console and select the DataSource tab. Click *Change* and select a data source from the list displayed. Click *OK* to close the selection dialog. Click *Apply* to make the change permanent (Figure 14-4).

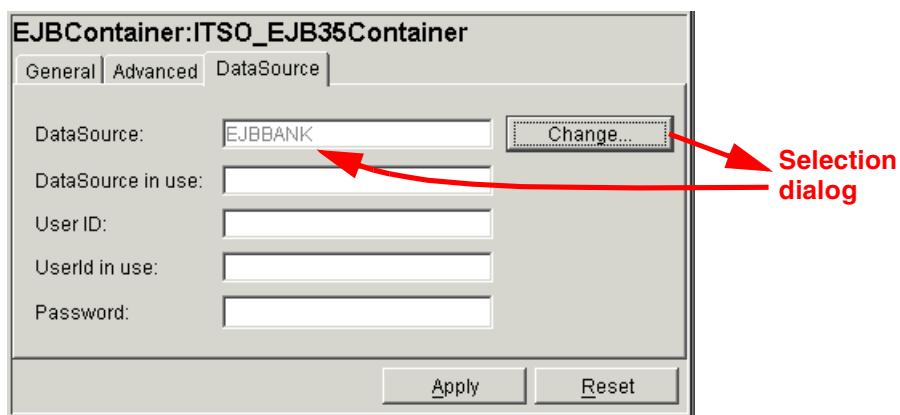


Figure 14-4 Setting a data source for the EJB container

## Installing a JAR file

Let's install the CMP\_Entity group in WebSphere. We generated the deployed JAR file as described in “Creating a deployed JAR file” on page 279.

The steps for EJB installation in WebSphere are:

- ▶ Select the EJB container and, from the context menu, select *Create -> EnterpriseBean*.
- ▶ In the SmartGuide you have to locate the JAR file using *Browse*, for example, locate the `deployedEJBs\itso_cmp.jar` file (Figure 14-5). Click *Select* and a conformation dialog tells you about the beans located in the JAR file.

Click *Yes* to select all the beans, or click *No* and double-click on the JAR file to select individual beans in the JAR file.

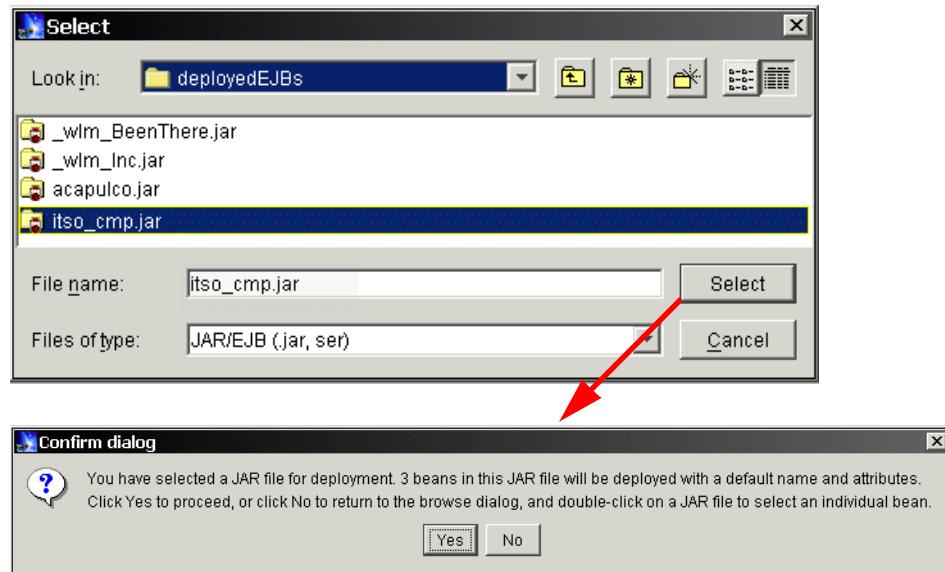


Figure 14-5 Create enterprise bean: confirmation dialog

- ▶ The SmartGuide is now filled with the names of the beans to deploy (Figure 14-6).

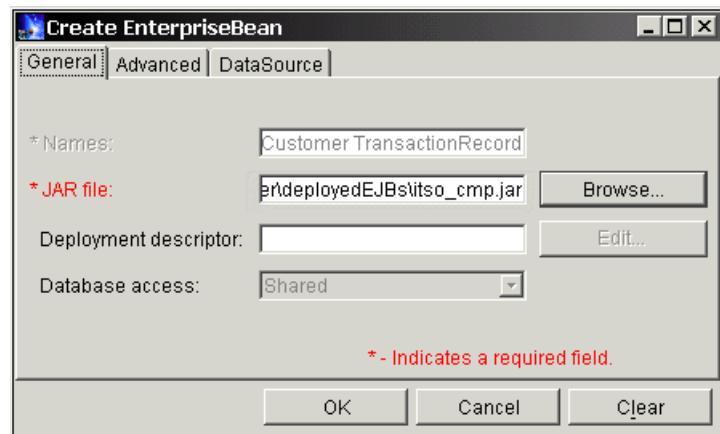


Figure 14-6 Creating an enterprise bean in WebSphere

- ▶ On the Advanced page you can set the size of the connection pool, and if *Find for update* should be used for these beans (Figure 14-7).

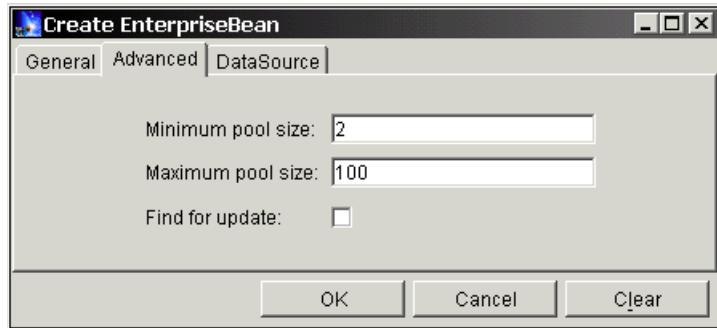


Figure 14-7 Enterprise bean connection pool

- ▶ On to the DataSource tab, nothing can be set for these beans.
- ▶ Click *OK* to create the beans, you will get a confirmation dialog box per bean.

## Setting EJB properties

The EJBs are added to the container. Select an EJB to see its properties (Figure 14-8). On the DataSource tab, deselect *Create Table*, otherwise WebSphere tries to recreate the table in the database (and will fail upon starting the server).

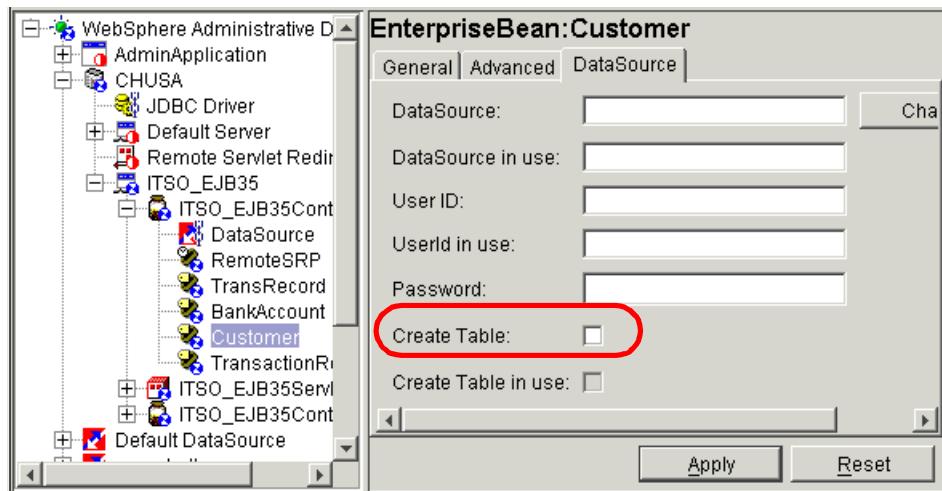


Figure 14-8 Modifying EJB properties

The data source is picked up from the container, but could be overwritten here.

## Installing a non-deployed JAR file

If you install a JAR file containing non-deployed EJBs, WebSphere will deploy them and then ask you if you wish to enable WLM.

As an example, we can deploy the BMP\_Entity group with the TransactionRecord bean:

- ▶ Create the EJB JAR file `deployableEJBs\itso bmp_n.jar` in VisualAge for Java (see “Non-deployed JAR file” on page 280).
- ▶ Select the container and *Create -> Enterprise bean*. In the dialog, click on *Browse* to locate the JAR file (`deployableEJBs\itso bmp_n.jar`) and click *Select*. The confirmation message box appears. Click *OK* and you are asked to deploy the JAR file with or without WLM (Figure 14-9).

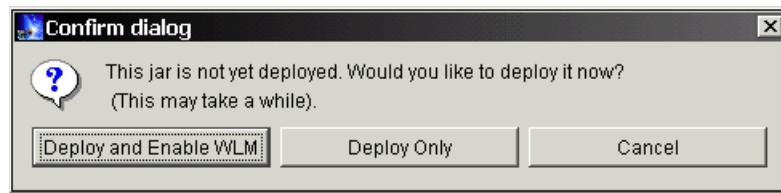


Figure 14-9 Deploying a non-deployed JAR file

- ▶ Click *Deploy Only* to generate the deployed code and install the bean. The deployed JAR file is added to the `deployedEJBs` subdirectory:  
`..\deployedEJBs\Deployeditso bmp_n.jar`

Restart the container to make the enterprise beans active.

## Installing session beans

There is no difference between installing entity or session beans. Create the deployed JAR file for the Stateless\_Session EJB group, for example, `itso_session.jar`.

Note that when clicking the *Select referenced types and resources* button, many classes from the data access beans project are included, if you implemented the CustomerFinder session bean with data access beans.

Then use the WebSphere Administrative Console to install the JAR file into the container. Note one difference: session beans show up in the Console with their home name, not with their bean name (Figure 14-11 on page 288).

## Testing with the VisualAge EJB test client

To verify successful installation of the code, we can test the EJBs using the VisualAge for Java EJB test client.

Make sure the application server (ITSO\_EJB35) and the container are running. Do not start the VisualAge for Java WebSphere Test Environment (it is not needed) and certainly do not start the persistent name server if you are testing on the same machine (start would fail because WebSphere occupies port 900).

Select the EJB group on the EJB page of VisualAge for Java and start the test client using *EJB -> Run Test Client*. The test client connects to the enterprise beans in WebSphere and you can operate it in the same way as with VisualAge for Java (Figure 14-10).

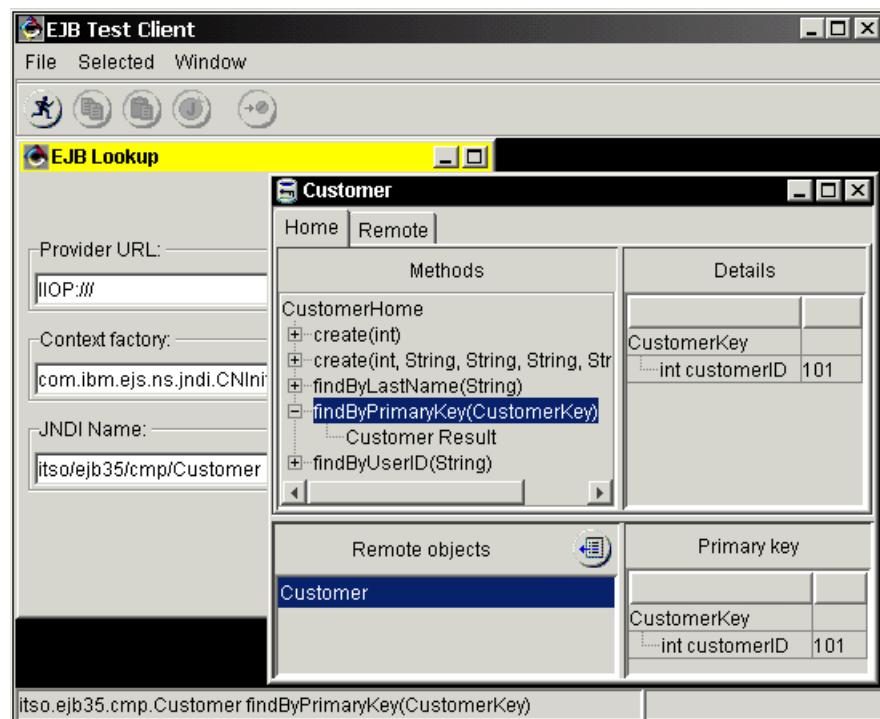


Figure 14-10 Using the EJB test client with WebSphere

# Deploying and running the client applications

Now let's look at how to export, configure, and run the client applications and servlets against the enterprise beans running in WebSphere.

We assume that the CMP and BMP entity beans, as well as the session beans, have been installed in an EJB container (Figure 14-11).

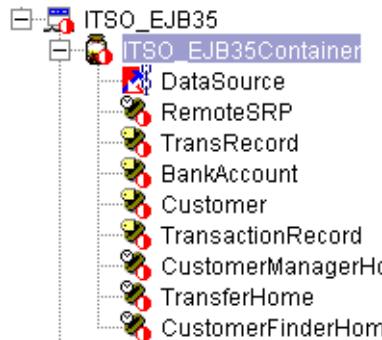


Figure 14-11 EJBs installed in the EJB container

Create a directory for testing the client applications, for example:

```
e:\SG246144\sampcode\ejbclient
```

## Create client JAR files

The first step is to package the code required by clients to access enterprise beans in a JAR file. VisualAge for Java provides an export facility to create client JAR files:

- ▶ Select the EJB group or multiple groups (CMP\_Entity, BMP\_Entity, and Stateless\_Session) and *Export -> Client JAR*.
- ▶ Select the location and name the client JAR file:  
`e:\SG246144\sampcode\ejbclient\itso_client.jar`
- ▶ Click on the *Select referenced types and resources* button and let VisualAge for Java search for types and resources used in your code for including them in the exported file.

## Java JDK and the class path

Make sure that the IBM JDK is available in the system PATH. In a command window, type `java -version`.

If you do not get a reply similar to:

```
java version "1.2.2"
Classic VM (J2RE 1.2.2 IBM build cn122-20001026b (JIT enabled: jitc))
you should add the JDK to the path:
set path=%path%;d:\WebSphere\AppServer\jdk\bin;
```

Client applications that use enterprise beans must have access to a number of classes that are contained in WebSphere JAR files and in the client JAR file we generated.

- ▶ ujc.jar is required for EJB classes (javax.ejb.EJBObject)
- ▶ ivjejb35.jar is required for access beans, inheritance, associations

The easiest way to set the correct class path is to write a small command file (setCP.bat):

```
set classpath=d:\WebSphere\AppServer\lib\ujc.jar;%CLASSPATH%
set classpath=d:\WebSphere\AppServer\lib\ivjejb35.jar;%CLASSPATH%
set classpath=itso_client.jar;%CLASSPATH%
```

## Test the applications

Export the class files of your applications (itso.ejb35 cmp client and itso.ejb35 gui packages, not the servlets) from VisualAge for Java into the test directory. This should create a directory structure:

```
e:\SG246144\sampcode\ejbclient\itso\ejb35\...pkcname...\...classname...
```

To test the stand-alone applications, open a command window and switch to the test directory. Then set the class path and run the applications:

```
cd e:\SG246144\sampcode\ejbclient
setCP.bat
java itso.ejb35 cmp client SimpleApp
java itso.ejb35 cmp client SimpleAppAB
java itso.ejb35 cmp client UsingFinder
java itso.ejb35 cmp client UsingFinderAB
java itso.ejb35 gui CustomerApplet
```

## Test the servlets

Export the servlet class files (itso.ejb35 cmp servlet package) into the WebSphere Web application directory:

```
d:\WebSphere\AppServer\hosts\default_host\itsoejb\servlets
==> ...itsoejb\servlets\itso\ejb35\cmp\servlet\...classname...
```

Copy the HTML and JSP files into the WebSphere Web application directory:

```
d:\WebSphere\AppServer\hosts\default_host\itsoejb\web
==> ...\\itsoejb\\web\\ejb\\cmpservlet\\...filename...
```

## Calling a servlet in a Web application

Modify all the HTML files. Currently they call the servlets without the Web application name, which is required under WebSphere for a Web application:

```
old: <FORM ... ACTION="/servlet/itso.ejb35.cmp.servlet.CustomerFind">
new: <FORM ... ACTION="/itsoejb/servlet/itso.ejb35.cmp.servlet.Xxxx">
```

For the CustomerFind servlet, which we defined in the Web application (see “Adding a servlet to the Web application” on page 464) we can also call the servlet using its alias name:

```
<FORM ... ACTION="/itsoejb/customerFind">
```

## Adding the client JAR file to the class path

The servlets require access to the access beans and potentially other classes. All these classes are assembled in the client JAR file. To add a JAR file to the class path of a Web application just copy the JAR file to the servlets directory:

```
d:\WebSphere\AppServer\hosts\default_host\itsoejb\servlets\itso_client.jar
```

## Running the servlets

Open a browser and display the HTML files that invoke the servlets:

```
http://localhost/itsoejb/ejb/cmpservlet/CustomerFind.html
http://localhost/itsoejb/ejb/cmpservlet/CustomerFindAB.html
http://localhost/itsoejb/ejb/cmpservlet/CustomerFindSF.html
http://localhost/itsoejb/ejb/cmpservlet/CustomerFindSFAB.html
```

Enter a customer number and invoke the servlet. Note that if you get error messages and you have to change the class path or servlet code, you have to restart the Web application (*Restart Web App* from the context menu), or in some cases you even have to restart the application server (ITSO\_EJB35).

## Deployment using the XML configuration tool

WebSphere provides the XMLConfig batch utility to modify the configuration of the application server. Using XMLConfig we can define an EJB container and EJBs by providing an XML file with the necessary configuration information. We describe this facility when we deploy advanced EJBs with inheritance and associations in “Using XMLConfig for deployment” on page 373.

## Setting up security in WebSphere

WebSphere provides facilities to secure Web files (HTML, JSP) and servlets. Servlets must be defined in WebSphere if you want to restrict their access, and you should not allow servlets being called by their full name.

Remove the Auto-Invoker servlet from the Web application to force the use of alias names for all servlets. Note that you have to code the HTML and JSP files accordingly, so that they call all servlets using the alias name.

Protecting the resources using WebSphere security goes beyond the content of this book. See Chapter 20. “Security for enterprise beans” on page 411 for an overview and the *WebSphere V3.5 Handbook* for more details.

## Tracing and debugging

For this section you must have installed the IBM Distributed Debugger 9.1 with the Object Level Trace (OLT) facility.

Debugging and OLT is enabled for an application server. In the Administrative Console select the server, and on the general page, enter these command line arguments, all on one line (Figure 14-12):

```
-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7777
-Xnoagent -Dcom.ibm.debug.jdwpport=7777 -Djava.compiler=NONE
-Xbootclasspath/a:%JAVA_HOME%\lib\tools.jar;
```

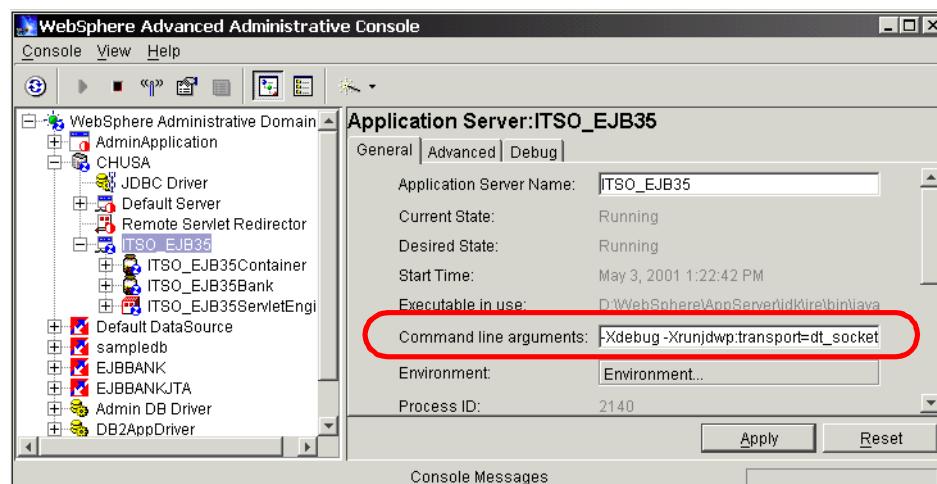


Figure 14-12 Tracing and debugging for an application server

Click *Apply*, then switch to the Debug page and enable debugging and object level trace (Figure 14-13).

Note the setting of the source path so that the debugger can find the source code of the servlets and EJBs:

```
e:\sg246144\sampcode\zsource
```

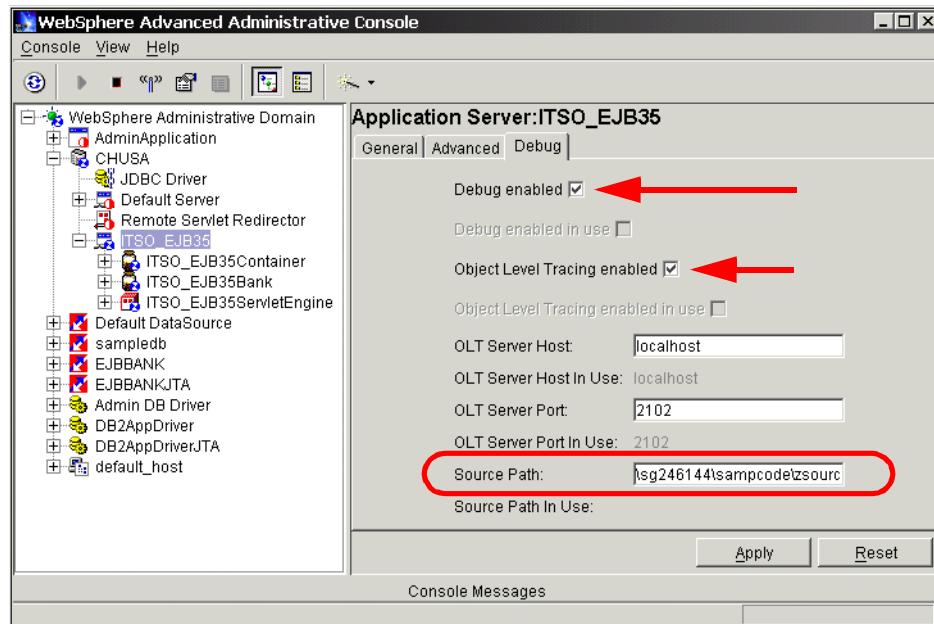


Figure 14-13 Enabling tracing and debugging

After setting the options, click *Apply*, and then stop and restart the application server.

## OLT tracing

Start OLT from a command window by typing `olt`, and wait for the OLT window to appear. Then run a servlet, for example, the `CustomerFind` servlet from its HTML page (`http://localhost/itsoejb/ejb/cmpservlet/CustomerFind.html`).

The OLT window displays the graph of calls between the servlet, the container, the JSP, and the EJB (Figure 14-14).

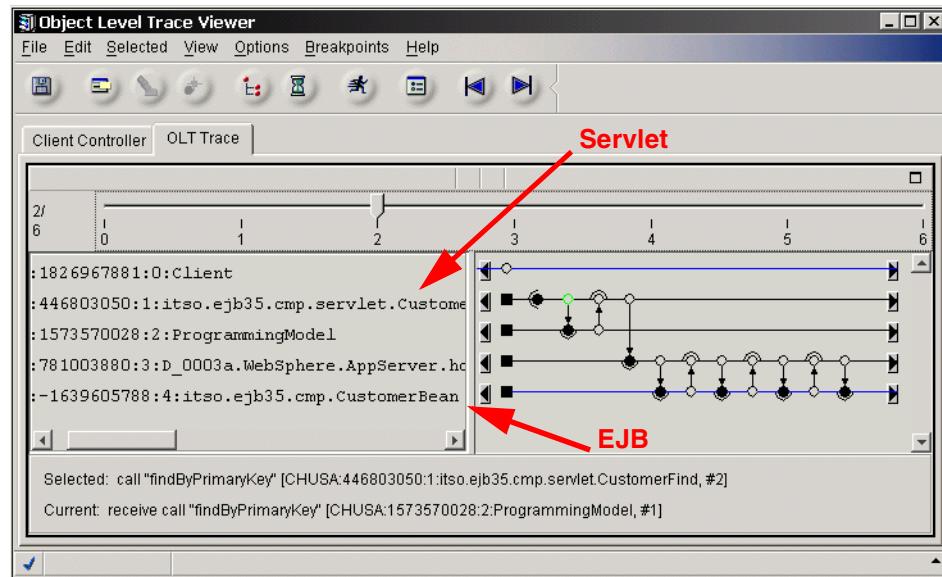


Figure 14-14 OLT trace diagram

## Breakpoints

You can set breakpoint at the start of method calls by selecting a filled circle point and *Add to Method Breakpoint List* from the context menu.

## Enable the debugger

Switch to the Client Controller page of the OLT window, change the execution mode to *Trace and debug* and click *Apply* (Figure 14-15).

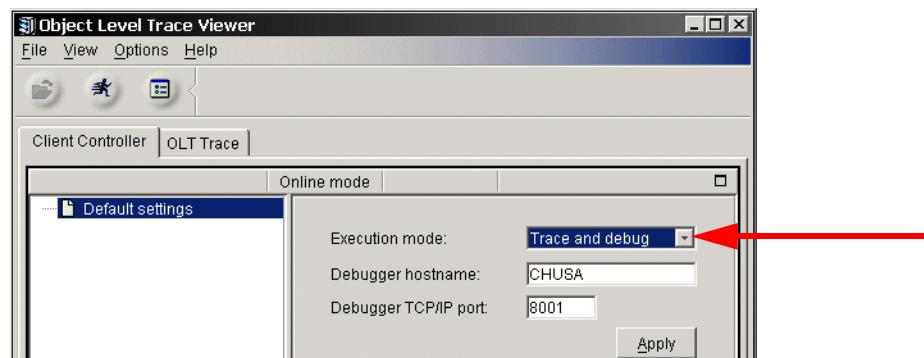
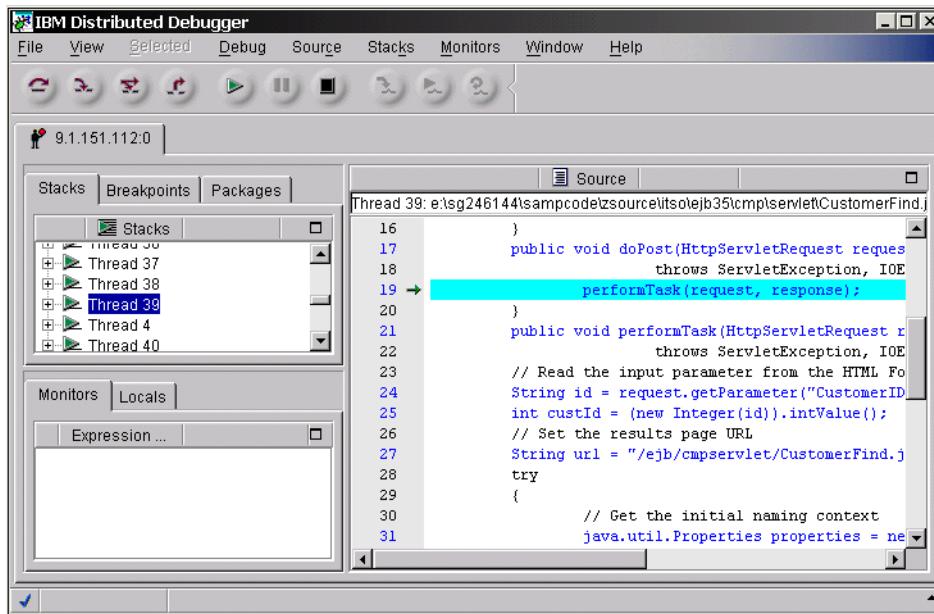


Figure 14-15 Enabling trace and debug in OLT

Rerun the servlet and the Distributed Debugger window opens. Be patient, the connection to WebSphere can take quite a long time.

# Distributed Debugger

The Distributed Debugger connects to the WebSphere execution environment and displays the source code of the servlet at the first breakpoint (Figure 14-16).



*Figure 14-16 Distributed Debugger*

You can now step through the code and debug servlet, JSP, and EJB code using the buttons on the toolbar. Note that some system classes cannot be debugged, because the source code is missing. Use the third toolbar icon to step only into code that can be debugged.

In the left-bottom pane you can see the local variables of the selected thread and you can add variables to be monitored.

It is good practice to have many breakpoints set, so that you can click the *Run* icon to forward to the next breakpoint.

## Debugging an application

Use a command file to set the class path and start the debugger (we provide this file as DebugApp.bat in sg246144\sampcode\ejbclient).

```
set DER_DBG_PATH=E:\sg246144\sampcode\zsource
set classpath=E:\sg246144\sampcode\ejbclient\itso_client.jar;%CLASSPATH%
set classpath=E:\sg246144\sampcode\ejbclient;%CLASSPATH%
set classpath=d:\WebSphere\AppServer\lib\ivjejb35.jar;%CLASSPATH%
set classpath=d:\WebSphere\AppServer\lib\ujc.jar;%CLASSPATH%

idebug -qlang=java -quiport=8001 -qjvmargs="-Xdebug -Xnoagent
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=8888
-Djava.compiler=None
-Xbootclasspath/a:"d:\WebSphere\AppServer\jdk\lib\tools.jar;d:\WebSphere\Ap
pServer\lib\derdbpw.jar"
-classpath "d:\WebSphere\AppServer\lib\dertrjrt.jar;%CLASSPATH%"
-Dcom.ibm.debug.jdwport=8888 -Dcom.ibm.CORBA.requestTimeout=0"
itso.ejb35.getCmp.client.SimpleApp
```

Note that the **idebug** command with all its parameters has to go onto one line! The debugger starts with the application source code loaded and you can step through the code (Figure 14-17).

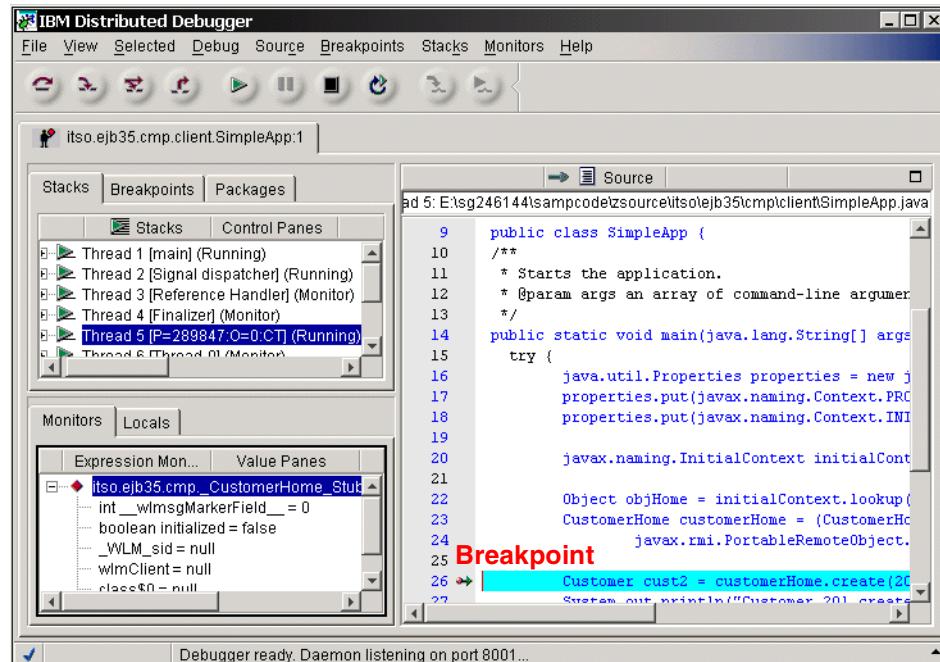


Figure 14-17 Debugging an application with EJB access

You can also activate OLT before starting the application. Additional parameters in the `idebug` command are used to communicate with OLT:

```
idebug -qlang=java -quiport=8001 -qjvargs="-Xdebug -Xnoagent
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=8888
-Djava.compiler=NONE
-Xbootclasspath/a:"d:\WebSphere\AppServer\jdk\lib\tools.jar;d:\WebSphere\Ap
pServer\lib\derdbpw.jar"
-classpath "d:\WebSphere\AppServer\lib\dertrjrt.jar;%CLASSPATH%"
-Dcom.ibm.debug.jdwport=8888 -Dcom.ibm.CORBA.requestTimeout=0
-Dcom.ibm.CORBA.EnableApplicationOLT=true
-Dcom.ibm.CORBA.OLTAapplicationHost=localhost
-Dcom.ibm.CORBA.OLTAapplicationPort=2102
-DOLTCClient=true"
itso.ejb35.cmp.client.SimpleApp
```

You step through the code in the debugger and you can see the object invocation diagram in the OLT window.

## Summary

The easiest way to deploy enterprise beans into WebSphere is to generate a deployed JAR file with VisualAge for Java and install the JAR file into a WebSphere container.

This approach is mandatory when using entity beans with inheritance or associations, or when a tailored schema and mapping was specified in VisualAge for Java. Be careful with the properties of the enterprise beans, especially if the database tables should be created or are already installed on the server machine.

Non-deployed JAR files are installed using the JetAce tool of WebSphere. Note that this tool recreates the mapping of the entity beans to database tables.

# Advanced topics

In this part we describe advanced topics for enterprise beans. This includes the mapping of entity beans into multiple tables, mapping using composers and converters, entity beans with inheritance structures and associations, exception handling, and security.

We also provide sample client programs using these advanced facilities.





# Advanced mapping for container-manager entity beans

This chapter provides some insights on the advanced features of the mapping support for enterprise beans in VisualAge for Java and WebSphere.

Advanced mapping features include bottom-up and tailored mappings, mapping of an entity into multiple tables, and mapping using conversion programs (composers and converters).

# CMP architecture

We first review the CMP architecture in an overall picture (Figure 15-1) and then detail each section.

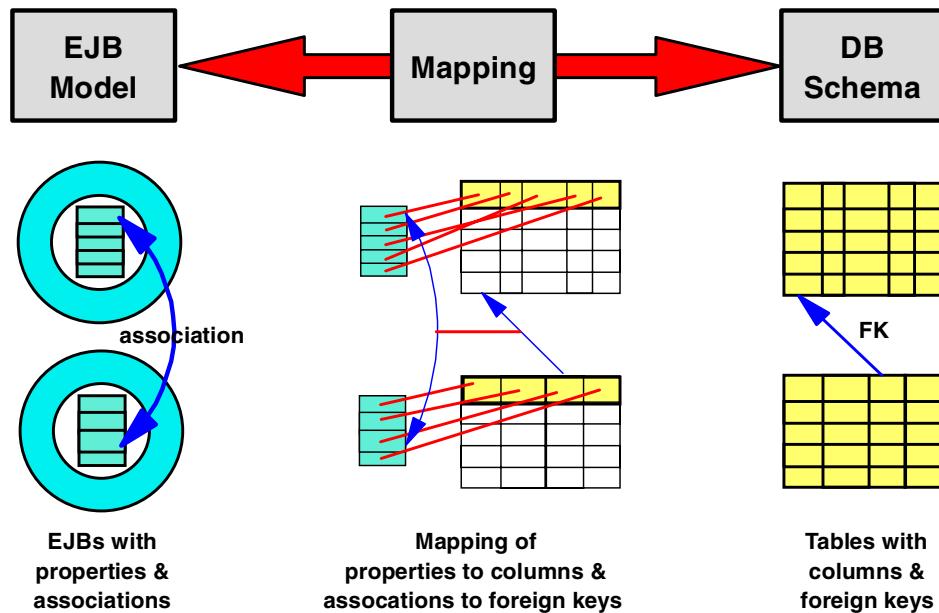


Figure 15-1 CMP architecture

The architecture is based on three components:

- ▶ EJB model of entity beans with associations
- ▶ Database schema with underlying tables
- ▶ Mapping between entity beans and tables

We are going to examine each separately and, at the end, see how they relate to each other.

## EJB model

The *EJB model* defines the object model. It contains EJBs, with or without inheritance and associations between them. It is the implementation of your design.

The tool used to define the EJB model is the EJB page of the VisualAge for Java Workbench (Figure 15-2).

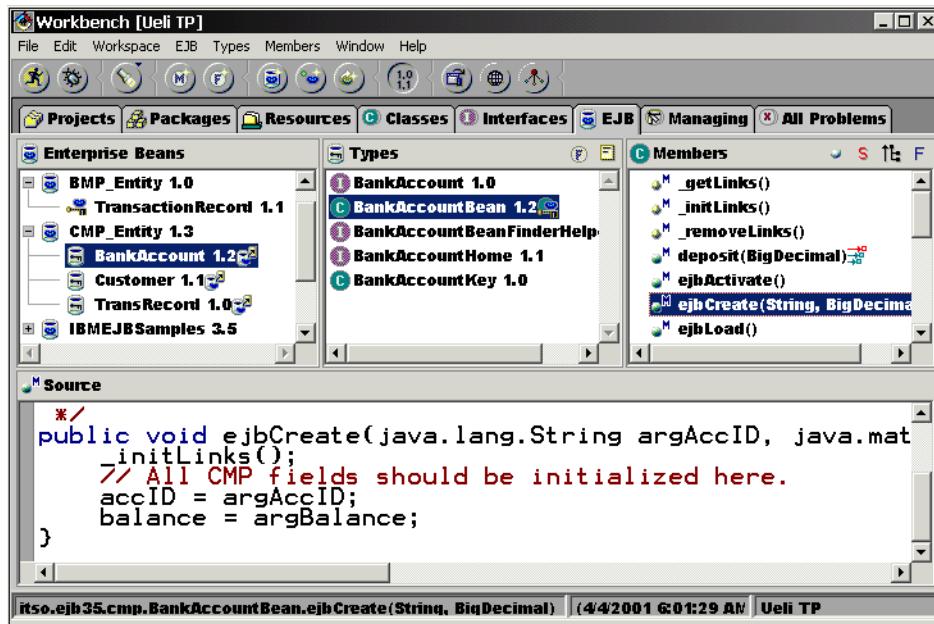


Figure 15-2 VisualAge for Java EJB page

We described how to create entity beans in Chapter 7. “Container-managed persistence entity beans” on page 105. We will discuss entity beans with inheritance in Chapter 16. “Inheritance” on page 329, and entity beans with associations in Chapter 17. “Associations” on page 353.

## Database schema

The *schema* defines a set of table definitions and foreign key relationships. Each table is composed of columns and each column has some characteristics.

The tool used to define the schema is the *schema browser* (Figure 15-3). Each EJB group should have a corresponding schema, named the same as the EJB group. When you select a schema, the list of table and foreign key definitions are displayed. Table and foreign definitions are detailed below.

The bottom pane of the window displays informational messages about the current selection and it can also display the DDL that is used to generate the currently selected tables, if you choose to export the definitions to a database system.

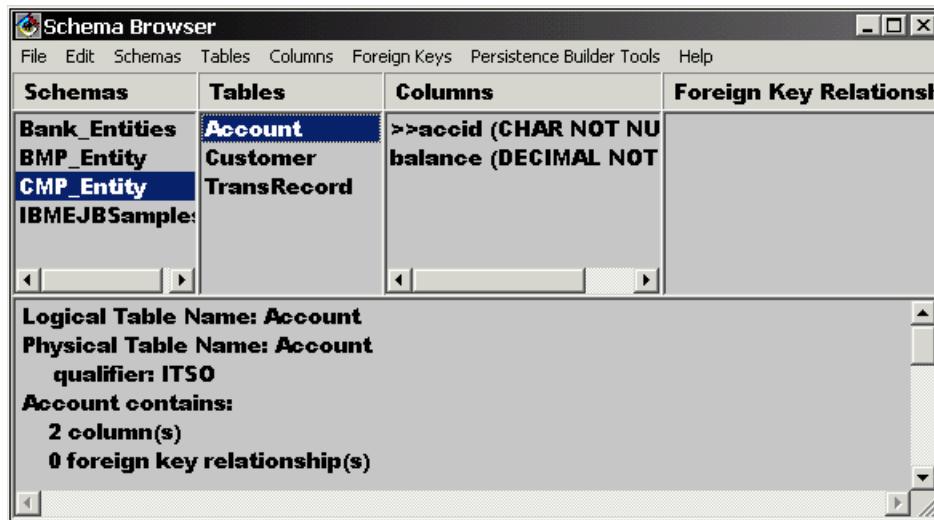


Figure 15-3 VisualAge for Java schema browser

There are four panes in the top half and one pane at the bottom:

- ▶ Schemas—a list of schemas loaded from the repository
- ▶ Tables—the list of table covered by the selected schema
- ▶ Columns—the column definitions of the selected table
- ▶ Foreign Key Relationships—relationships where the selected table is involved
- ▶ The bottom pane of the window contains information depending on the selected item in the top half.

### Table and column definitions

In the schema, you define tables, their columns, and the primary key field (Figure 15-4).

The table editor is used to define the table with its name (logical and physical), qualifier, columns, and primary key column(s). In this example the real table name is ITSO.ACCOUNT, composed of qualifier and physical name.

The columns are defined using the *New* button in the table editor or they are edited using the *Edit* button. These actions open the Column Editor window where you can edit basic properties, such as name, physical name, and SQL type with length and scale.

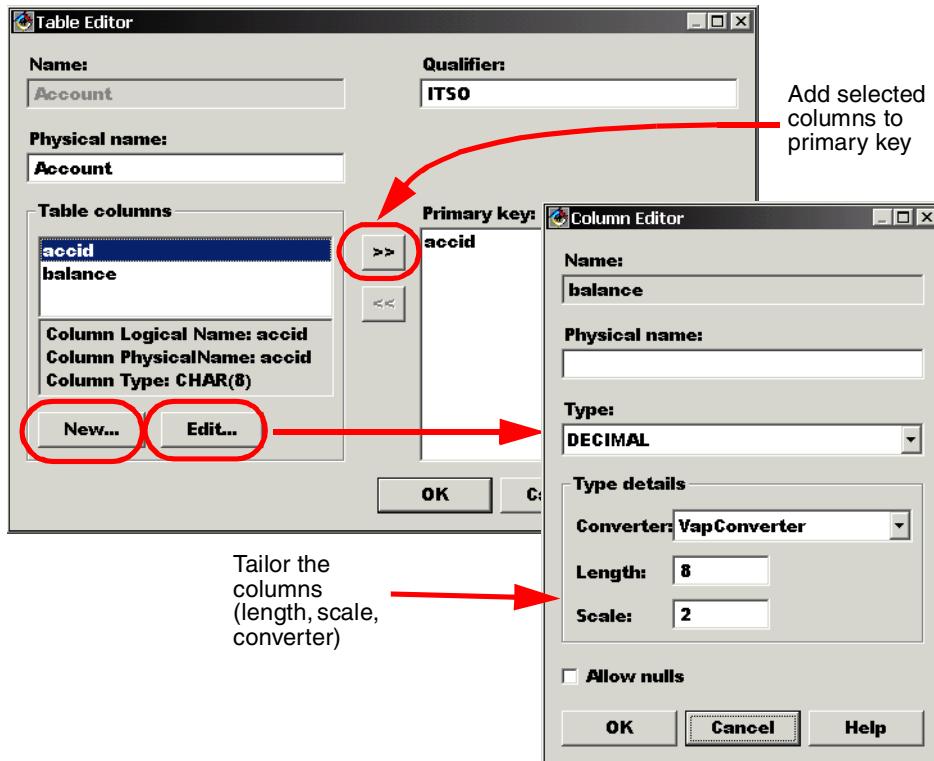


Figure 15-4 Table and column editor

## Type conversion

The *Converter* property of the column defines a two-way converter between Java types (primitives types and simple classes) and SQL types. VisualAge for Java ships with a number of converters, the most common are listed in Table 15-1.

Table 15-1 Example of useful converters

Converter Name	Role
VapTrimStringConverter	Convert String to fixed size CHAR column
VapBinaryStreamToSerializableConverter	Convert Java serializable object to/from BLOB column
VapConverter	Convert most Java primitive types to their SQL equivalents

Table 15-2 shows some of the Java types and how they are mapped to SQL types by the default VapConverter.

Table 15-2 Types associated with VapConverter and their default mapping

Java type	SQL type
short	SMALINT
int	INTEGER
long	BIGINT
double	DOUBLE
float	REAL
BigDecimal	DECIMAL(20,2)
String	VARCHAR(30)
boolean	BINARY or BIT
Other (non serializable)	VARCHAR(30)

The default mapping for Serializable types is a BLOB of size 2000 with a VapBinaryStreamToSerializableConverter.

## Tailoring the columns

If the schema is generated from the entity bean, the default columns are often not suitable. In many cases you want to tailor:

- ▶ CHAR(n) instead of VARCHAR(30) for short strings, and use of the VapTrimStringConverter that removes blanks when the character value is converted into a Java string.
- ▶ The maximum length of VARCHAR columns; the default of 30 may not be correct.
- ▶ The precision of decimal fields, for example DECIMAL(8,2) instead of (20,2).

## Foreign key relationships

The schema browser allows you to create and edit foreign key relationships between tables. In the Foreign key pane, select *Foreign Keys -> New Foreign Key Relationship* or *Foreign Keys -> Edit Foreign Key Relationship*. In both cases, the Foreign Key Relationship Editor dialog is opened (Figure 15-5).

In this dialog you specify the name (logical and physical), if the constraint exists in the database, the primary and the foreign key table, and the field in the foreign key table that serves as a foreign key.

The primary key field (or fields) is always filled in and cannot be changed.

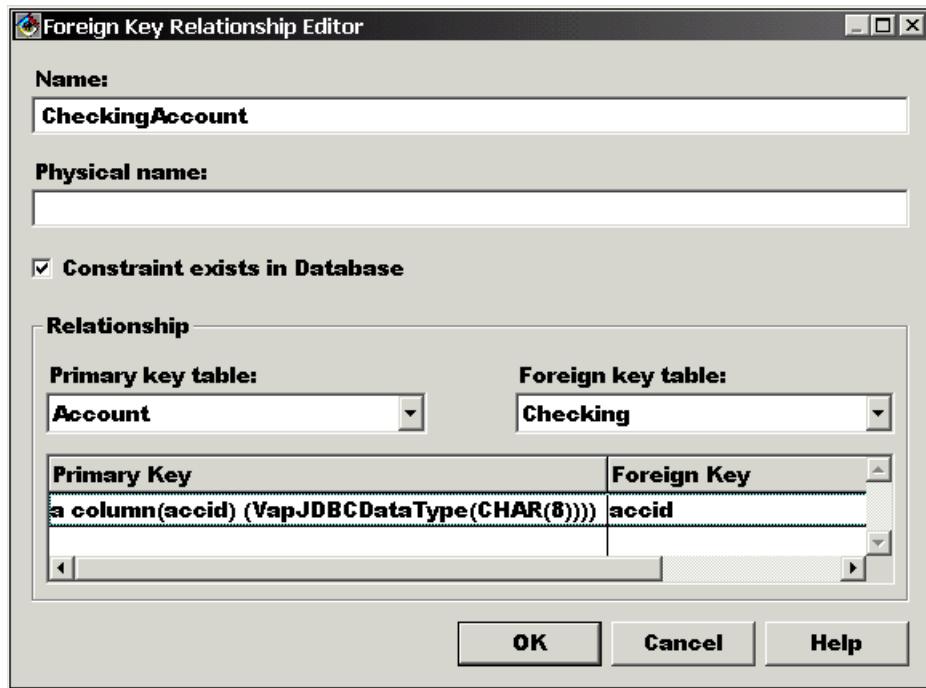


Figure 15-5 Editing foreign key relationships

For entity beans, foreign keys are used to model:

- ▶ Dependent tables, where one entity is mapped into multiple tables  
See “Using secondary tables” on page 321 for an example.
- ▶ Inheritance structures, where each subclass entity has its own table  
See “Implementing the inheritance with VisualAge for Java” on page 336 for an example.
- ▶ Associations between entities  
See “1:m association” on page 355 for an example.

## Mapping

The *mapping* defines how the objects and associations in the EJB model connect with the tables and foreign key relationships of the schema.

You define the mapping using the map browser of VisualAge for Java (Figure 15-6).

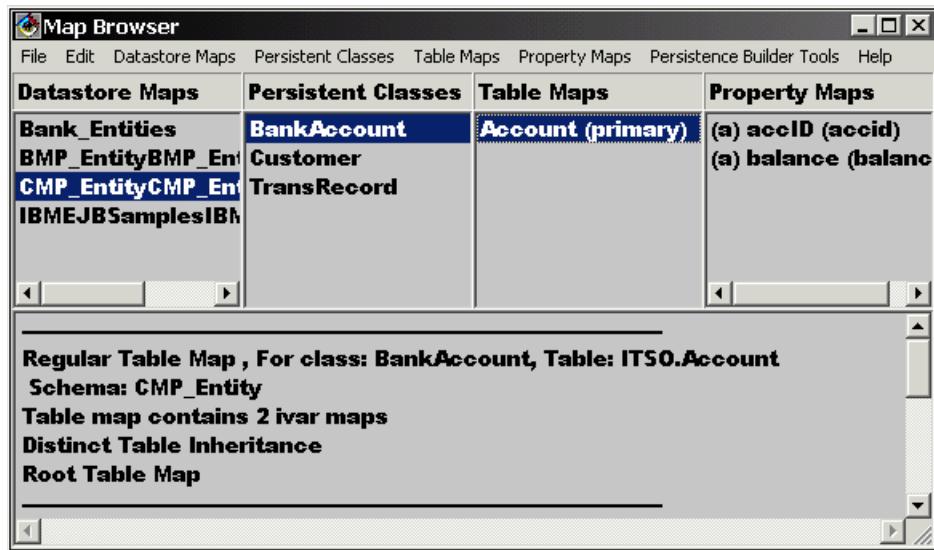


Figure 15-6 VisualAge for Java map browser

There are four panes in the top half and one pane at the bottom:

- ▶ Datastore Maps—a list of maps (usually, the name of the EJB group repeated, for example `CMP_EntityCMP_Entity`).
- ▶ Persistent Classes—the entity beans covered by this mapping.
- ▶ Table Maps—the table definitions used for the selected bean.
- ▶ Property Maps—the mapping of entity properties to table columns.
- ▶ The bottom pane of the window contains information depending on the selected item in the top half.

You can open the table maps and the property maps:

- ▶ The table map specifies which table or tables are used for a bean. An entity can be mapped into one or multiple tables. Table maps are also used for inheritance, where each subclass can have its own table.
- ▶ Property maps specify how each attribute of the entity is mapped to a column of a table (Figure 15-7). The map types are `<Not mapped>`, Simple, or Complex.

Most mappings are simple. Attributes that are not mapped occur when an entity is mapped into multiple tables, or with inheritance (where some attributes are inherited from the superclass and are mapped into another table). Complex mapping involves writing composer classes that perform the operation (see “Using composers” on page 312).

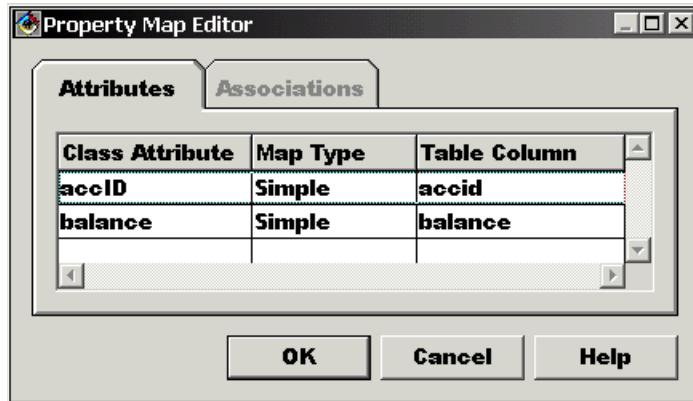


Figure 15-7 Property map editor

## Design and mapping approaches

While designing your entity beans and application, you should decide up front how your mapping to database will be handled, because it will have important consequences on your design and implementation.

Three approaches are common:

- |                    |                                                                                                                                |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------|
| Top-down           | The entity design is mapped as-is to the database. Some tailoring of the tables will be performed.                             |
| Bottom-up          | The database tables are imported as a schema and matching are generated. Some tailoring of the entity beans will be performed. |
| Meet-in-the-middle | The database tables are imported as a schema. The entities are designed and then hand-mapped to the database tables.           |

In most real applications you start either top-down or bottom-up, and then use the meet-in-the-middle approach to tailor the implementation.

### Top-down

The *top-down* approach is the easiest to handle, because no legacy database is imposed to you (Figure 15-8). You have full freedom to design the entity beans.

The top-down approach is also called *forward engineering*.

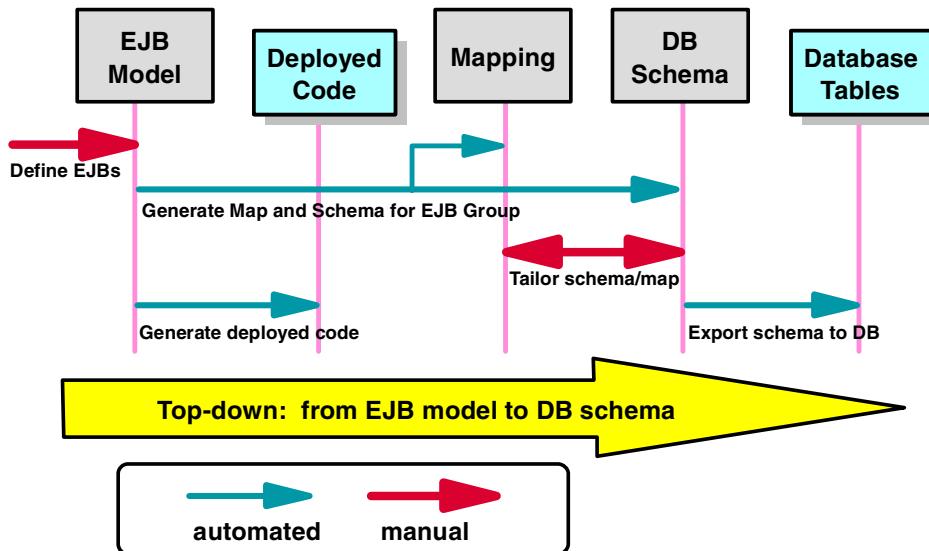


Figure 15-8 EJB mapping: top-down approach

## Approach

The steps in the top-down approach are:

- ▶ Create an EJB group, add entity beans to the group, and define the required associations for the application.
- ▶ Once the object model is created inside the EJB page, you can generate the schema and mapping by selecting *Add -> Schema and Map from EJB Group* from the EJB group context menu.
- ▶ Check the schema and map for errors. Typical errors are due to serialized objects (EJB handles, for example) being mapped to VARCHAR columns instead of BLOB. Fix them and you are done.
- ▶ Tailor the schema: table and column names, column characteristics (CHAR or VARCHAR, maximum length of strings, precision of DECIMAL columns).
- ▶ Save the schema and map.
- ▶ Export the tables to the database with the keys (*Import/Export Schema -> Export Entire Schema to Database*).

**Attention:** You should avoid designing object models that lead to unrealistic database design. Always have a database administrator check the resulting database design.

## Bottom-up

The *bottom-up* approach is the exact opposite of the top-down approach. Basically, you create an EJB layer on top of an existing database design (Figure 15-9). This approach is also called *reverse engineering*.

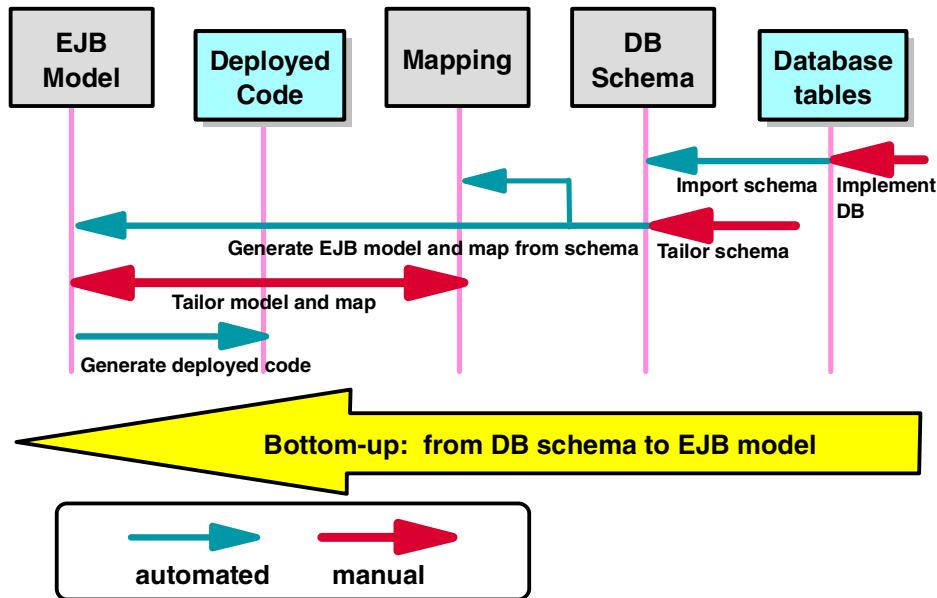


Figure 15-9 EJB mapping: bottom-up approach

Here your freedom of development is on top of the EJB layer. You have to add session beans to provide all the business logic that the entity beans do not provide.

### Approach

The steps in the bottom-up approach are:

- ▶ Implement a database (or have an existing database).
- ▶ Import a schema from the database into the schema browser (*Import/Export Schema* -> *Import schema from Database*). Tailor the names of tables and columns in the schema browser because these names will be used for the entity beans.
- ▶ In the EJB page, select *Add EJB group from Schema or Model* and select the schema to use. This action creates the EJBs and the corresponding mapping from the schema. Tailor the model (Java types) and adjust the mapping if necessary.

## Meet-in-the-middle

*Meet-in-the-middle* is the most common approach used in the e-business projects. It combines an existing database and an entity model design (Figure 15-10).

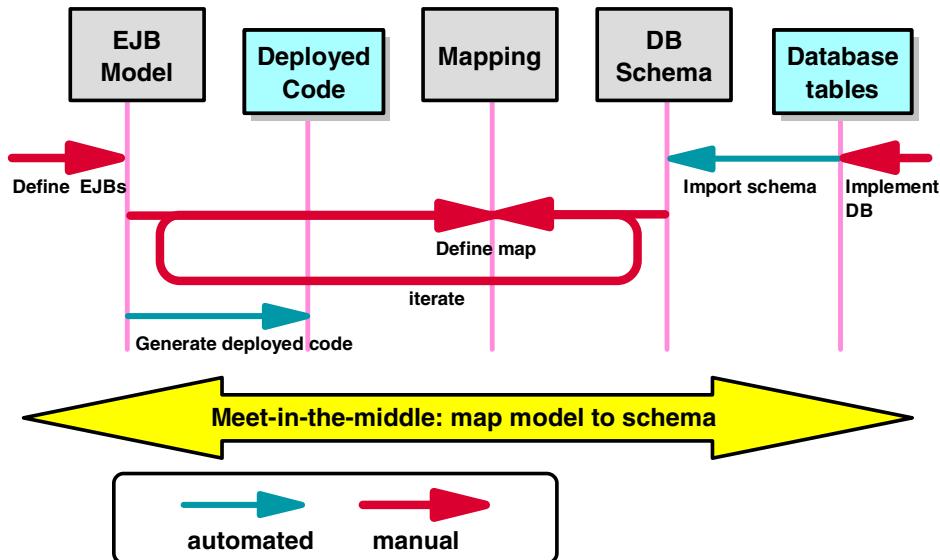


Figure 15-10 EJB mapping: meet-in-the-middle approach

The real difficulty is to find a good balance between the design and the mapping. The entity model must match the database design in enough commonality so that a mapping is feasible.

The schema has a very limited role here, because it is imposed and cannot be changed. The development life cycle mainly will be a cycle between the EJB model and the mapping where one has to accommodate changes to make the other viable.

## Approach

The steps in the meet-in-the-middle approach are:

- ▶ Implement a database (or have an existing database).
- ▶ Import a schema from the database into the schema browser (*Import/Export Schema -> Import schema from Database*).
- ▶ Create an EJB group, add entity beans to the group, and define the required associations for the application.

- ▶ Use the map browser to map the entity beans to the schema tables by hand. For each entity add one or multiple tables to the mapping, and then map the attributes to the columns.

The real problem is how can the design and the legacy tables match! Here you will have to deploy all your knowledge of composers, converters, as well as compromising skills to have the two coexist. It may happen that your design is impossible to map in its entirety and, therefore, redesign will have to be done.

In most cases this is an iterative process that takes several cycles to result in a good entity bean design and also good performance.

## Mapping examples

We showed the top-down approach in “Database schema and map” on page 121 for the Customer CMP entity bean. We did tailor the tables to match with our existing database, but such tailoring of the length of string fields is almost always necessary.

We describe an example of the bottom-up approach in “Reverse engineering from an existing database” on page 376.

We describe an example of the meet-in-the-middle approach in “Implementing the inheritance with VisualAge for Java” on page 336, where we map an inheritance structure to existing tables.

## Converters and composers

Converters and composers are used to perform the mapping operation between an entity attribute and one or more table columns:

- ▶ A **converter** maps one attribute to one column. Many converters are provided by VisualAge for Java, but you can write your own as well. A converter is specified in the schema browser and is used for simple mapping.
- ▶ A **composer** maps one attribute (which can be a complex class) to multiple table columns. A composer is specified in the map browser and is used for complex mapping.

The coding for converters and composers is very similar; both must provide methods for both ways of the mapping; from an entity attribute to column(s) and from column(s) to an entity attribute.

## Using composers

The map browser allows you to specify a “complex mapping” for a CMP field. When you use this option, you have to provide a composer class. You typically use this option for complex attributes, such as the CustomerAddress object used in the Customer bean.

A composer is designed to split a complex attribute into multiple columns.

### How to write composers

There are a number of rules to follow when implementing a composer

- ▶ Give the composer class a unique class name, because the package information will not be stored in the persistence metadata. The package therefore can be chosen at will.
- ▶ The composer is a singleton object (only one instance will exist) so you have to make sure it behaves correspondingly:
  - Declare a static variable named `singleton` of the same type as the composer class itself.
  - Provide a `singleton` method returning the object stored in the static variable.
  - Provide a `reset` method to assign `null` to the singleton.
- ▶ A composer is a subclass of `com.ibm.vap.composers.VapAttributeComposer` and must provide implementations for these methods:
  - `getTargetClassName`—returns the name of the class associated with the composer (same as the instance created as a result of the `objectFrom(Object)` method).
  - `getAttributeNames`—returns an array of the attribute names of the target class.
  - `getSourceDatatype`—returns an array of class names, one for each attribute of the target class (matches `getAttributeNames` in size).
  - `objectFrom`—converts an array of data objects (from the database) into attributes values of the target class (in the order of `getAttributeNames`) and returns an instance of the target class.
  - `dataFrom`—converts an instance of the target class into an array of data objects (for the database) and returns that array.
- ▶ The composer is specified in the mapping of an attribute. It shows up automatically in the pull-down, because it is a `VapAttributeComposer`.

Figure 15-11 shows the implementation of these methods for the `AddressComposer`.

```

package itso.ejb35.bank;
import itso.ejb35.util.CustomerAddress;

public class AddressComposer
 extends com.ibm.vap.composers.VapAttributeComposer {

 private static AddressComposer singleton;

 public static String getTargetClassName() {
 return CustomerAddress.class.getName();
 }
 public static String[] getAttributeName() {
 String[] attributes = {"street", "city", "state", "zipcode"};
 return attributes;
 }
 public static String[] getSourceDatatype() {
 String[] types = {"String", "String", "String", "String"};
 return types;
 }
 public Object objectFrom (Object[] anArray) {
 String name, street, city, state, zipcode;
 street = (String) anArray[0];
 city = (String) anArray[1];
 state = (String) anArray[2];
 zipcode = (String) anArray[3];
 return new CustomerAddress(street, city, state, zipcode);
 }
 public Object[] dataFrom (Object anObject) {
 Object[] anArray = new Object[] {null, null, null, null};
 if (anObject != null) {
 CustomerAddress address = (CustomerAddress) anObject;
 anArray[0] = address.getStreet();
 anArray[1] = address.getCity();
 anArray[2] = address.getState();
 anArray[3] = address.getZipcode();
 }
 return anArray;
 }
 public static AddressComposer singleton() {
 if (singleton == null) singleton = new AddressComposer();
 return singleton;
 }
 public static void reset() { singleton=null; }
}

```

Figure 15-11 AddressComposer class

### **Remarks about the code**

- ▶ The `getTargetClassName` method is using `class.getName()` on the `CustomerAddress`. It avoids hardcoding string in the code which are difficult to maintain and error-prone.
- ▶ The singleton is using lazy-initialization on the singleton variable.
- ▶ Make sure that the order of the attributes is identical in the `getAttributeNames`, `objectFrom`, and `dataFrom` methods.

### **How to use composers**

Let's delay the use of composers. We will show how to use the address composer in the mapping developed in "Using secondary tables" on page 321.

## **Using converters**

As you have seen in "Table and column definitions" on page 302, a *converter* is always associated with a column definition. The converter has the responsibility to convert a Java type to an SQL type.

We could, for example, use a converter to map the `CustomerAddress` attribute (see Table 7-1 on page 114) to some string representation that is stored as a `VARCHAR` in the table.

Another example would be to map a bean that contains a collection of strings. If you let the default mapping handle it, the collection is serialized in a `BLOB`. The solution is to write a simple converter that reads the collection and stores it in `VARCHAR` or `CLOB` column as an XML stream. Of course the converter must be able to handle the conversion in the other direction also, reading the XML and parsing it to recreate the collection.

We do not provide an example of a converter. The coding is similar to that of a composer. You would create a subclass of `VapConverter` and implement the same methods as for a composer (Figure 15-11 on page 313), with the exception that you deal with one object instead of an array.

The best approach for building your own converter would be to study the implementation of the converters provided with VisualAge for Java in the `com.ibm.vap.converters` package.

# Entity model with advanced mapping

For the section that follows and for the chapters on inheritance and associations, we want to implement the full bank model shown in Figure 4-1 on page 54.

We perform these tasks in a new EJB group and a new package, so that we do not disturb the current implementation of the `CMP_Entity` group.

## Setting up the structure

We use the `Customer`, `BankAccount` and `TransRecord` beans developed in Chapter 7. “Container-managed persistence entity beans” on page 105. Here we discuss setting up the basic structure for building the hierarchy and the associations.

Instead of redefining the entity beans, we can copy the underlying classes to a new package and add the beans to a new EJB group. The copy operation itself is simple, but we have to fix some of the references in the new classes.

The steps are as follows:

1. In the EJB pane of the Workbench add a new EJB group to the ITSO EJB Redbook project, for example, `Bank_Entities`. We will build the complete model in this group.
2. Create a new package, under the project ITSO EJB Redbook. For our example, name it `itso.ejb35.bank`.

### Copy the Customer bean

3. Select the `CustomerBean` related group of classes and interfaces, but not the access bean and deployed code (Figure 15-12):

```
Customer
CustomerBean
CustomerBeanFinderHelper
CustomerHome
CustomerKey
```

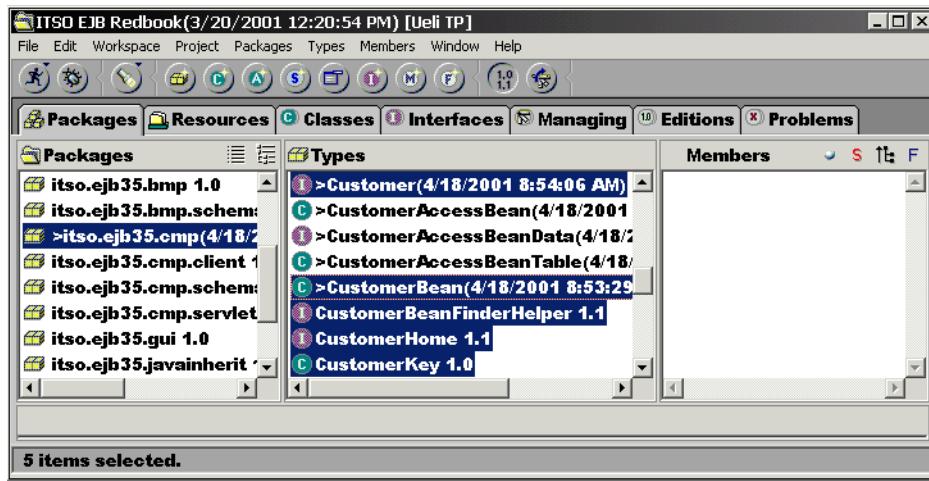


Figure 15-12 Copying classes to a new package

4. Invoke the context menu and select *Reorganize -> Copy*. A dialog named *Copying Types* appears. Specify the package `itso.ejb35.bank`. Ensure that *Rename the copy* check box is unchecked. Then click *OK* and the selected classes are copied.
5. The `CustomerBean` has errors in the new package that must be fixed. The bean, and the home and remote interfaces have references to the old package name and must be changed.

Change all references to:

```
itso.ejb35.cmp.Customer ==> itso.ejb35.bank.Customer
itso.ejb35.cmp.CustomerKey ==> itso.ejb35.bank.CustomerKey
```

6. In the Workbench EJB page, invoke the context menu on the EJB Group and select *Add -> Enterprise Bean*. The Create Enterprise Bean SmartGuide appears. Select *Use an existing bean class* option. Select the appropriate project and package (`itso.ejb35.bank`), click *Browse* for the class, and select the `CustomerBean` (Figure 15-13).
7. Click *Finish* and the existing bean is added to the new group.

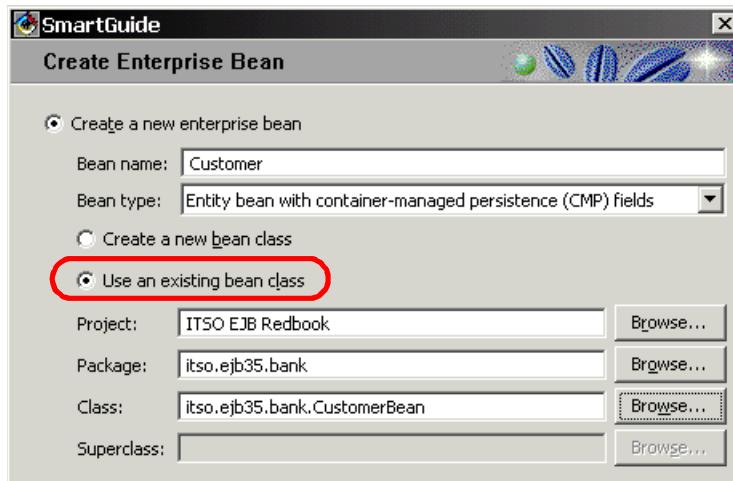


Figure 15-13 Adding an existing bean to an EJB group

### Copy the TransRecord bean

Perform the same process for the TransRecord bean. Select these classes for the copy operation:

```
TransRecord
TransRecordBean
TransRecordBeanFinderHelper
TransRecordHome
TransRecordKey
```

Change the home interface (TransRecordHome):

- ▶ Change the package name for the return types in the create and findByPrimaryKey methods (from itso.ejb35.cmp to itso.ejb35.bank).
- ▶ Change the package name of the TransRecordKey parameter in the findByPrimaryKey method in the same way.

Add the TransRecord bean to the Bank\_Entities group in the EJB page of the Workbench. (Use same process as for the Customer bean in Figure 15-13.)

### Copy the BankAccount bean

Perform the same process for the BankAccount bean and add the bean to the Bank\_Entities group in the EJB page.

Select these classes for the copy operation:

BankAccount  
BankAccountBean  
BankAccountBeanFinderHelper  
BankAccountBeanFinderObject  
BankAccountHome  
BankAccountKey

Change the home interface (BankAccountHome):

- ▶ Change the package name (from itso.ejb35.cmp to itso.ejb35.bank) for the return types in the create and findByPrimaryKey methods and for the BankAccountKey parameter in the findByPrimaryKey method.

Change the BankAccountBean so that it creates transaction records of the correct bean class:

- ▶ Change the home interface variable:

```
protected itso.ejb35.bank.TransRecordHome txRecHomeCMP;
```

- ▶ Change the setTxRecHome method:

```
txRecHomeCMP =
 (itso.ejb35.bank.TransRecordHome) javax.rmi.PortableRemoteObject.
 narrow(ctx.lookup("itso/ejb35/bank/TransRecord"),
 itso.ejb35.bank.TransRecordHome.class);
```

Add the BankAccount bean to the Bank\_Entities group in the EJB page of the Workbench. (Use same process as for the Customer bean in Figure 15-13.)

## Completing the EJBs

Switch to the Properties view in the Types pane. Notice that only the key field carries the container-managed icon. You have to manually set the other fields to be container-managed, ***in all three beans*** (Figure 15-14 shows the Customer).

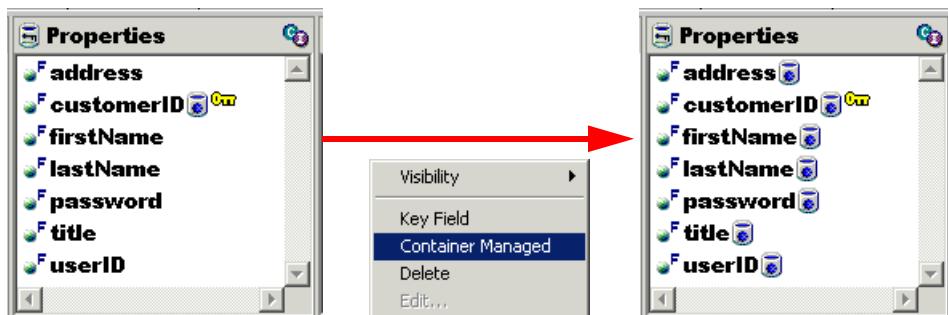


Figure 15-14 Setting fields as container-managed

## Mapping the beans

The three new beans are not mapped to tables. Copying the beans to a new EJB group requires a new schema and mapping for this entity group:

- ▶ First we map the TransRecord bean into the existing TransRecord table.
- ▶ Then we map the Customer bean into two tables, one table for all the properties except the address, and one table for the address property. See “Using secondary tables” on page 321.
- ▶ Finally, we will map the BankAccount bean in Chapter 16. “Inheritance” on page 329. We will define two subclasses (Checking and Savings) as two types of bank accounts and map all types of accounts into a set of tables.

### Create the schema

Open the schema browser and select *Schemas -> New Schema* and use the same name as the EJB group, *Bank\_Entities*.

### Create the map

To create the data map for this schema open the map browser (*EJB -> Open To -> Schema Maps*) and:

1. Select *Datastore\_Maps -> New EJB Group Map*.
2. Enter a name (usually the same name as for the schema) and select the EJB group and the schema we created (Figure 15-15). Click *OK*.

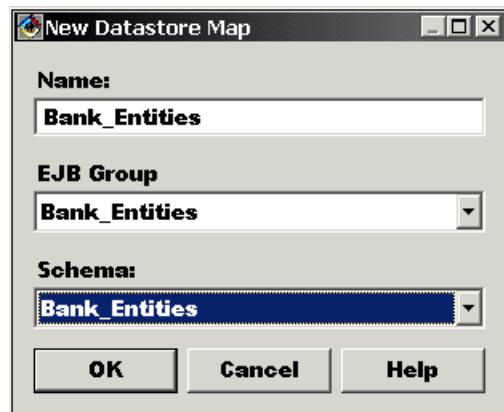


Figure 15-15 Creating a new map

## Mapping the TransRecord bean

To create the table and mapping for the TransRecord bean:

1. In the schema browser, Tables section, select *New Table* and define the ITS0.TransRecord table (Figure 15-16).

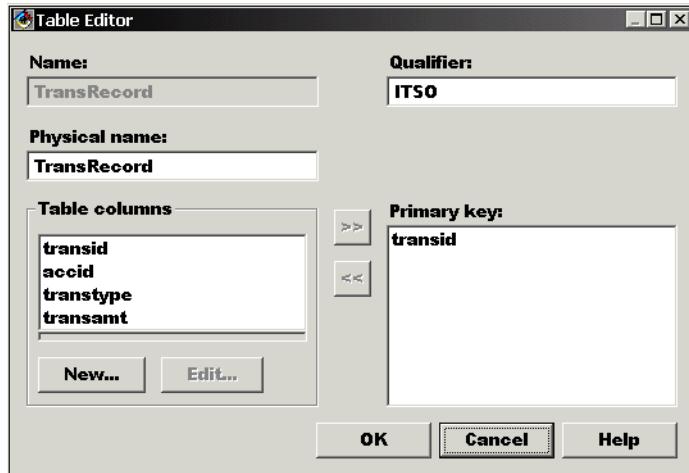


Figure 15-16 Table for the transaction record bean

2. Define columns that match our existing table:

transid	TIMESTAMP not null (key)
accid	CHAR(8) not null
transtype	CHAR(1) not null
transamt	DECIMAL(8,2) not null

3. In the map browser, *Persistent Classes section*, the list of classes are displayed. We map the TransRecord class to the TransRecord table by selecting the class and *Table Maps -> New Table Map -> Add Table Map with No Inheritance*. Select the TransRecord table from the pull-down.
4. Edit the Property Map and map each attribute to the matching column.

### Generate deployed code

In the EJB page, generate the deployed code for the TransRecord bean.

## Using secondary tables

The map browser enables you to map a bean to several tables. The main table, called the *primary table*, holds the primary key and some attributes, whereas the *secondary* tables have a copy of that primary key as a foreign key and the rest of the attributes.

A typical use of these secondary tables is when the primary table contains too many attributes or some very long attributes (such as BLOBs), and the database administrator decides to split it in several tables.

**Attention:** Usually, in the database world, less important attributes are relegated to secondary tables to improve performance. In the EJB world, a bean mapped to several tables always reads all of its attributes when accessed. A better solution would be to place the less important attributes into another bean and map it to the secondary table.

## How to use secondary tables

We will now use the address composer that was created in “How to use composers” on page 314 and map the address field to a secondary table. The process below describes the general steps, then we apply the process to our case:

- ▶ Identify the candidates attributes to be placed on the secondary table and remove the corresponding columns from the primary table.
- ▶ Create a new table with the remaining columns. Do not forget to add a column with the same type and size as the primary key of the primary table, because it will be used for the foreign key. That column will also be the primary key for the secondary table.

## Creating a two-table schema and mapping for the Customer bean

Because we copied the Customer bean to a new entity group, there is no table and mapping defined for it. We create a new table and mapping for this entity.

## Create the schema table

Open the schema browser and:

1. Select the Bank\_Entities schema.
2. In the *Tables* section, select *New Table*. Define the ITSO.Customer table with columns that match our existing table (Figure 15-17):

customerID	INTEGER not null (key)
firstName	VARCHAR(30) not null
lastName	VARCHAR(30) not null
title	CHAR(3) not null
userID	CHAR(8)
password	CHAR(8)

Note: Do not define the address column (BLOB 2000)

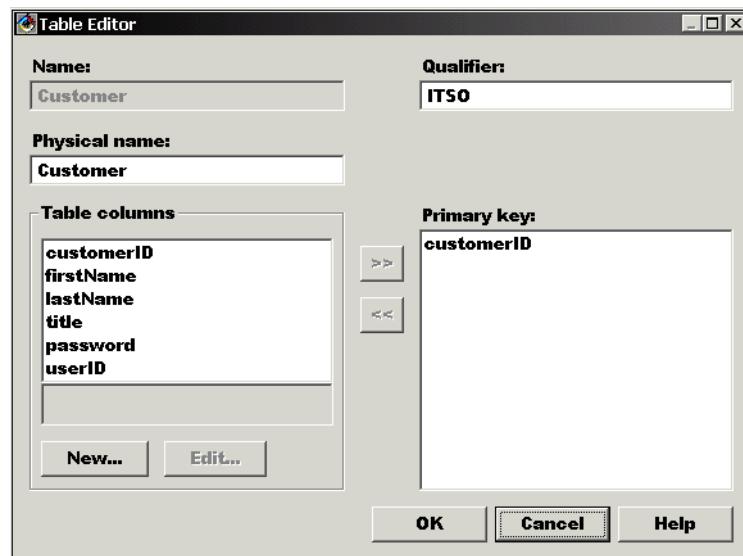


Figure 15-17 Primary table for customer

3. Define the secondary ITSO.Address table in the same way (Figure 15-18):

customerID	INTEGER not null (key)
street	CHAR(20)
city	CHAR(12)
state	CHAR(12)
zipcode	CHAR(10)

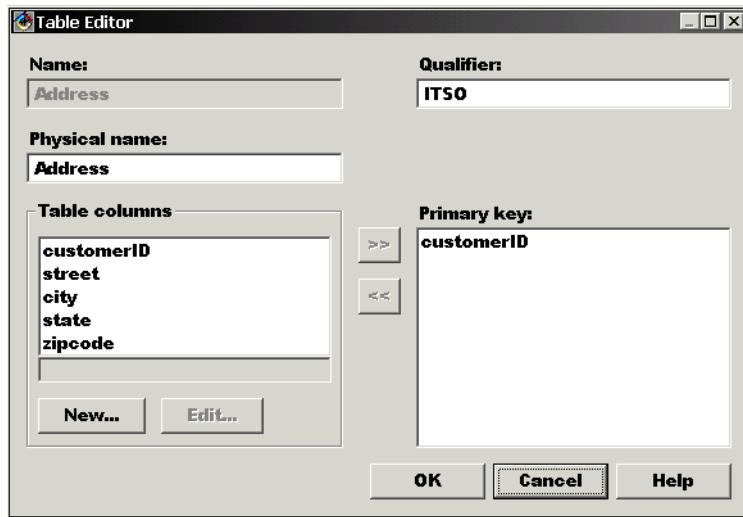


Figure 15-18 Secondary table for customer address

- Select *Foreign Keys -> New Foreign Key Relationship* to define the relationship between the customer and address tables (Figure 15-19).



Figure 15-19 Foreign key relationship for customer address

## Create the map

To create the map for this schema, open the map browser (*EJB -> Open To -> Schema Maps*) and:

1. Select the Customer class and *Table Maps -> New Table Map -> Add Table Map with No Inheritance*. Select the Customer table and click *OK*.
2. Define the secondary table using *Table Maps -> New Table Map -> Add Secondary Table Map* and select the Address table and the CustomerAddress foreign key relationship (Figure 15-20).



Figure 15-20 Secondary table map

3. Select the Customer table map and *Edit Property Map*. Map all the attributes of the Customer class, except the address, to the primary table columns (Figure 15-21).

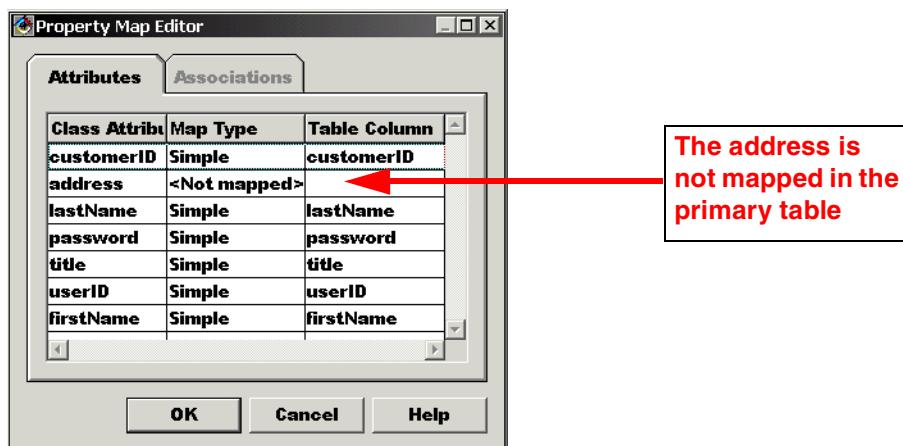


Figure 15-21 Property map for the primary table

- Map only the address attribute to the secondary table using a *complex* mapping type.

Click on the square button to open the Complex Attribute Editor, where you select the composer type from the pull-down (the AddressComposer should appear) and then map the attributes to the columns (Figure 15-22).

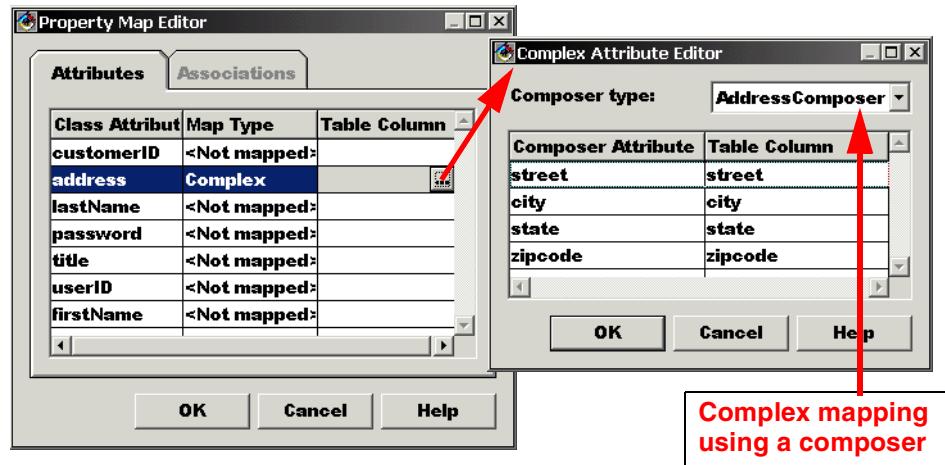


Figure 15-22 Property map for the secondary table

- The final mapping in the map browser is shown in Figure 15-23.

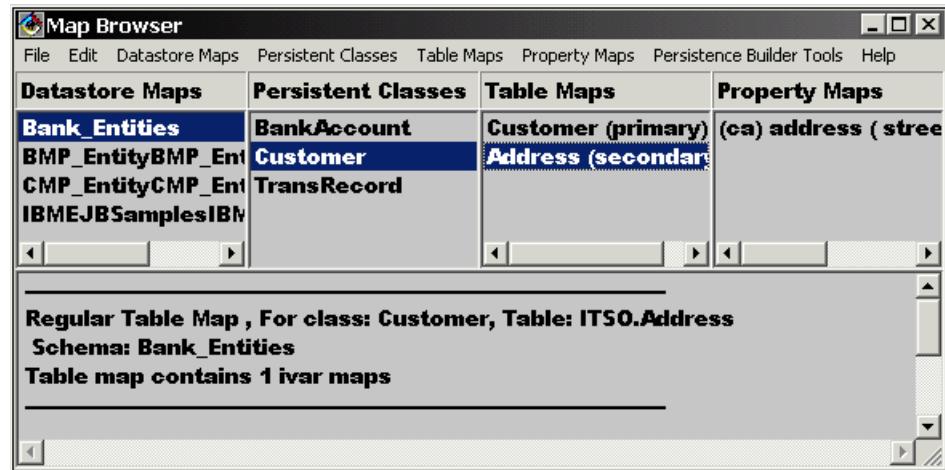


Figure 15-23 Map browser with mapping into two tables

- Save the schema and map, for example, to `itso.ejb35.bank.schema`.

In a top-down approach you could export the schema of the new table definitions and foreign key relationship to the database to implement the tables. In our example the tables already exist.

## Generate the deployed code and test

Generate the deployed code for the Customer bean. To test the approach, start the Persistent Name Server, add the EJB group to the server configuration, start the EJB server, and start the EJB test client

Retrieve the customer home, then retrieve a customer (key 106). Execute the getAddress method and display its address (Figure 15-24).

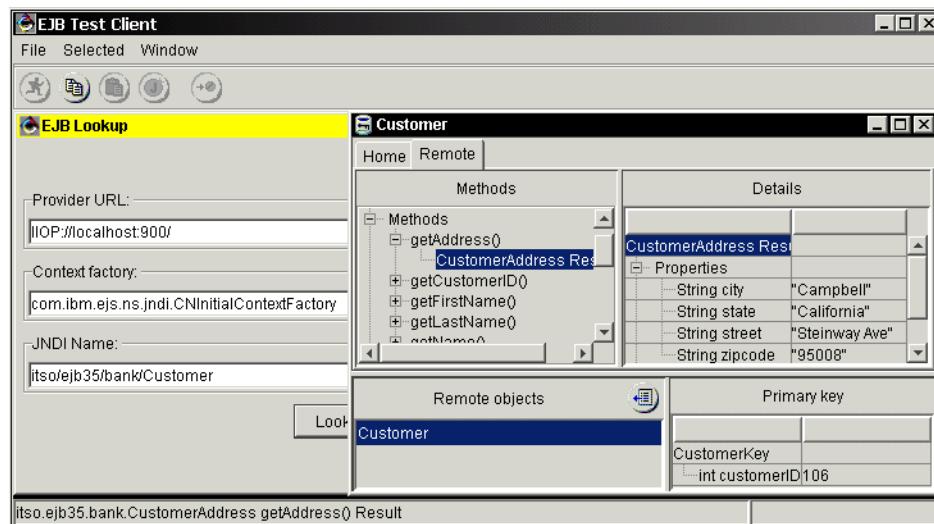


Figure 15-24 EJB test client with customer and address

When designing an entity using a primary and secondary table, each entity must have matching records in both tables. If a record only exists in the primary table, you get an exception (Figure 15-25).



Figure 15-25 Exception when address data is missing

You can also create a new customer. In our implementation we did not provide a constructor that includes address data. An empty address record is created in the table, either with null or empty strings, dependent on the coding of the CustomerAddress class.

You can use the setAddress method to update the address data.

## Summary

In this chapter we explored some of the approaches when implementing entity beans and their underlying database implementation.

For completely new applications the top-down approach is very effective. You design the entity model, and from there create the schema and mapping. You would usually tailor the schema with the help of a database administrator.

In many installations databases already exist, and you want to create new-style Web applications using enterprise beans. In such a case, you can use the bottom-up approach to have the system generate entity beans from an existing set of tables. Most of the time you then tailor the entity beans, because the generated classes might not be suitable for your application.

In some cases a manual design of the entity beans is the best. You would have the underlying database tables in your mind, while you layout the entity model. Then you perform a manual mapping between the model and the schema.





# Inheritance

Defining an enterprise bean as a subclass of another enterprise bean is not part of the Enterprise JavaBeans specification. However, in real life applications, we need this essential feature of object-oriented programming. Direct inheritance is specific to WebSphere Application Server, Advanced Edition and VisualAge for Java Enterprise.

VisualAge for Java provides different mapping approaches for an inheritance hierarchy: a single table approach, a root/leaf approach with foreign key related tables, and individual tables.

In addition, entity beans can inherit from regular Java classes.

## Bank example

In this chapter, we use the BankAccount entity bean created in the “Defining the bank model” on page 132. This entity bean describes a generic bank account. The Checking and Savings enterprise beans that we create in this chapter inherit from this generic entity bean (Figure 16-1).

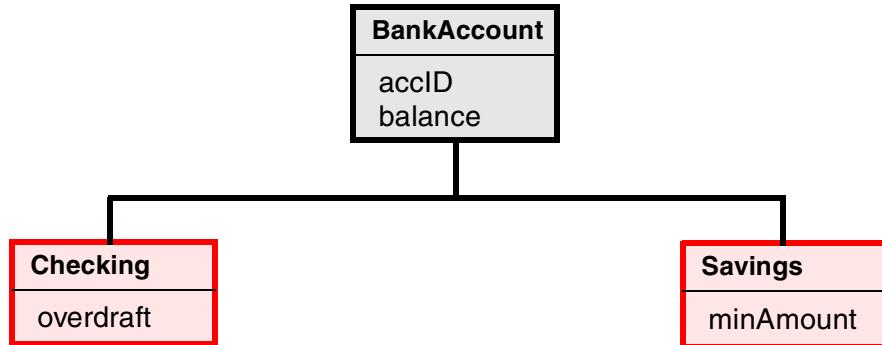


Figure 16-1 BankAccount with inheritance: Checking and Savings

## Inheritance overview

In this section, we describe the different types of inheritance, developing EJBs based on these types with examples, and the pros and cons of these different approaches. The EJB development environment permits two forms of inheritance: EJB inheritance and Java inheritance.

### EJB inheritance

In EJB inheritance, an enterprise bean inherits properties, such as CMP fields and association roles, methods, and method-level control descriptor attributes from another enterprise bean that resides in the same EJB group. Because EJB inheritance is possible only within the same EJB group, the Savings and Checking beans must be created in the same group in which we created the BankAccount bean.

### Java inheritance

In Java inheritance, an enterprise bean class inherits properties and methods from a super class that is not itself enterprise bean class, and the remote interface extends from an interface which defines the business methods and is implemented by the super class.

## Mapping schemes for inheritance model

In this section, we describe three mapping schemes related to an inheritance hierarchy.

### Single-table mapping

All the classes in the inheritance model map to a single table (Figure 16-4). A discriminator column is used to specify the type of the instance in the hierarchy. The discriminator column is not an attribute of any enterprise bean; it is only used for mapping purposes.

In this mapping model, the table is not fully normalized, that is the columns specific to one bean will be unnecessarily tied up with other beans and remain not utilized. But this model provides easy access to the tables. One practical limitation to this mapping is, when we create a new enterprise bean by extending the existing beans of the model, we need to alter the definition of the table. This is to accommodate the new columns specific to the bean.

Therefore, single table mapping should only be used when there are more CMP fields in the parent enterprise bean than in the child beans.

Table ACCOUNT

accID	balance	accType	minAmount	overdraft
101-1001	1495.00	SAVINGS	100.00	
102-2003	223.57	CHECKING		50.00
103-3001	675.82	CHECKING		50.00

 Discriminator column       unused space in table

Figure 16-2 Single table inheritance mapping

### Root/leaf table mapping

In root/leaf table mapping, you create one table per enterprise bean in the inheritance hierarchy (Figure 16-3).

The *root* table contains the parent bean specific columns and the discriminator column. As some of these columns are related to properties inherited by the child class beans, this table will also contain the state of these properties of the child beans.

A *leaf* table is created for each child bean. Each child bean's *leaf* table has a foreign key pointing to the primary key of the root table and columns for fields specific of the child bean. Using root/leaf mapping is more efficient, because it does not waste space in the tables. On the other hand two tables must be accessed to instantiate an entity bean.

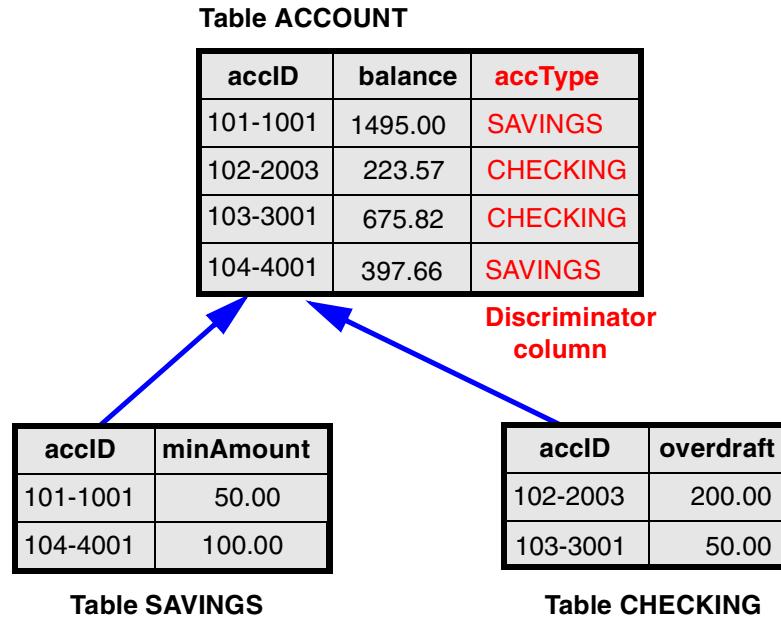


Figure 16-3 Root/leaf table inheritance mapping

### Distinct table mapping

Each class in the inheritance hierarchy maps to a different table. In this model, the tables are not likely to be normalized, because each table contains all the columns. Because of data consistency problems between the unrelated tables, it is difficult to automate the data access for this model.

## Characteristics of EJB inheritance

EJB inheritance, in respect to developing EJBs has the following characteristics:

- ▶ For CMP entity beans, there is support for single table and root/leaf table mapping.
- ▶ The deployment descriptor will list all of the CMP fields, including those fields that have been inherited.

- ▶ Home interface classes cannot inherit from other home interface classes. Therefore, we have to promote all the related methods ourselves.
- ▶ Remote interface classes can inherit from other remote interface classes. This means, we can inherit directly all the business methods defined in the parent bean in the remote interface of the child beans. In addition, we need to promote the business methods written specifically for the child bean.
- ▶ The bean class of the new child enterprise bean will extend the bean class of the parent enterprise bean.
- ▶ Key classes are shared among all enterprise beans in the inheritance model. This means that the key class in the child enterprise bean is identical to the key class of the parent enterprise bean.
- ▶ For BMP entity beans you can use any mapping model, because you write the JDBC access code yourself.

## Characteristics of Java inheritance

The general characteristics of Java inheritance, with respect to the EJB development are:

- ▶ Only distinct table mapping is supported in Java inheritance.
- ▶ The bean class extends from a superclass A, which implements the interface B. The remote interface can extend from the interface B, which defines the business methods.
- ▶ The derived classes from the same super class are entirely different beans and they have different mappings and schemas respectively.
- ▶ The derived bean classes have different key classes characterizing them.
- ▶ If the superclass contains state, make sure that the state is persisted.

## EJB inheritance

The EJB inheritance model is specific to the IBM EJB development environment of VisualAge for Java. This is an extension of the EJB specification. In this section we describe EJB inheritance as implemented in the IBM tool set.

### Requirements for EJB inheritance

The inheritance implementation for entity beans implies providing the following functionalities:

- ▶ Specifying an is-a relationship among entity beans, such as a Savings is a BankAccount.

- ▶ Implementing the `findByPrimaryKey` method in the home interface of the parent bean in such a way that the target base class along with its subclass is returned. For example, a search for a `BankAccount` returns either a `Savings` beans or a `Checking` bean.
- ▶ Implementing custom finder methods so that they return collections of `Savings` and `Checking` bean instances.
- ▶ Relationship finders also return heterogeneous results, containing instances of the related class and its subclasses.

## Limitations

There are some practical limitations on EJB inheritance:

- ▶ Session beans cannot inherit from entity beans and vice versa.
- ▶ Stateless session beans cannot inherit from stateful session beans and vice versa.

## Home interface specifics

The *home* interface of the child bean, for example, `SavingsHome`, must not extend the parent home interface, `BankAccountHome`. However, the relationship between the methods of these two interfaces have to be maintained. When a method is added to the home interface of the parent bean, it is automatically copied to the child beans.

A create method of the parent *home* interface can only create the parent bean instance. However, a create method of the child *home* interface creates both the parent bean instance and the child bean instance.

A client may not work directly with an instance of the parent bean but only with the child bean, which is more specific to the client's requirements. (This assumes that there are no instances of the parent bean that are not instances of one of its children.)

## Developing the inheritance hierarchy

In this section, we develop two enterprise beans, `Savings` and `Checking`. Both beans will be inherited from the `BankAccount` bean that we developed in “Entity model with advanced mapping” on page 315. In this section, we will discuss developing the inheritance hierarchy using the EJB inheritance model.

### Remote interface

The remote interfaces of the child beans define the business methods specific to them. They also extend the remote interface of the `BankAccount`.

## BankAccount bean

The BankAccount bean, which is the super class for our example hierarchy, is an entity bean. This bean does not have a create and the associated ejbCreate methods, because we assume that each account must be either a checking or a savings account. (In real applications there can be models where instances of the superclass exist.) The BankAccount bean does have finder methods. Using these finder methods, we can retrieve the corresponding subclass beans. This class also includes business logic method that is independent of the account type, for example, deposit of funds.

## Beans and interfaces

In the bank application scenario, we implement two beans, Savings and Checking, inherited from the BankAccount bean (Figure 16-4).

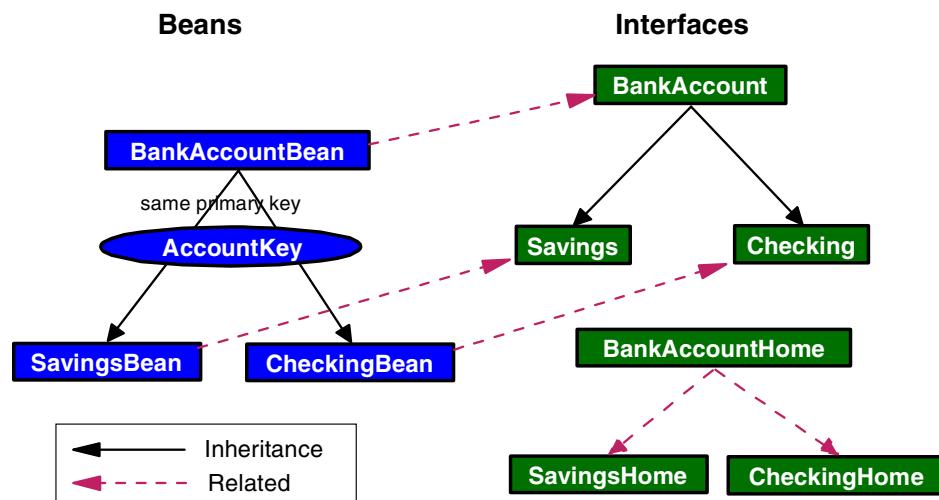


Figure 16-4 EJB inheritance hierarchy example

## Mapping scheme

For our example, we employ root/leaf inheritance mapping model. In this mapping scheme, we have one table for all common attributes, and one table for each subclass.

The root table contains the columns corresponding to the CMP fields defined in the parent class, whereas leaf tables have columns corresponding to the CMP fields defined in their respective subclasses, along with a foreign key matching the primary key of the root table.

We choose root/leaf model because:

- ▶ The tables are compact.
- ▶ There is an exclusive logical association of a bean with its table, as the table exactly represents the bean state of the model.
- ▶ It enables more subclasses to be derived and added to the class hierarchy, without any modification to the existing table structures.

## Implementing the inheritance with VisualAge for Java

In this section we implement the entity beans of the bank inheritance structure with VisualAge for Java.

### Setting up the structure

We use the **BankAccount** bean of the Bank\_Entities EJB group. Here we will discuss setting the basic structure for building the hierarchy.

**Important:** This section assumes that you have defined the Bank\_Entities EJB group and copied the BankAccount related classes to the `itso.ejb35.bank` package, as described in “Entity model with advanced mapping” on page 315.

The Workbench EJB page should include the BankAccount bean, with the related types (interface, home interface, finder helper, key class).

Check the CMP fields. The accID and balance fields should be container-managed.

### Creating the subclass beans

Savings and checking accounts are subclasses of the bank account. We define these accounts as enterprise beans with inheritance.

#### Savings bean

To define the savings account as an enterprise bean in an inheritance structure, follow these steps in the EJP page:

1. Select the parent enterprise bean that you want your new child enterprise bean to inherit from, in our example, BankAccount.

- From the *EJB* menu, select *Add -> Enterprise Bean with Inheritance*. The *Create Enterprise Bean with Inheritance* SmartGuide appears (Figure 16-5).

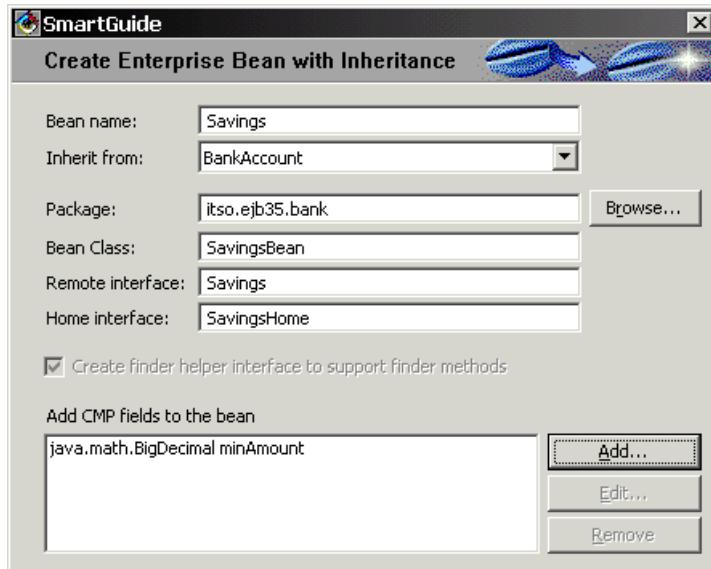


Figure 16-5 Inherit from an existing bean

- Enter the name of the new bean (*Savings*).
- The subclass can have its own CMP fields. Click on *Add* and define *minAmount* (*BigDecimal*) as a new CMP field with getter/setter methods, promote, and getter read-only.

### Checking bean

Create the *Checking* enterprise bean in the same way as a subclass of *BankAccount*. Define *overdraft* (*BigDecimal*) as a CMP field.

## What is inherited?

Let's review what is inherited. The *Savings* bean inherits the interface of the *BankAccount* bean, that is, the methods:

```
getBalance()
setBalance(BigDecimal)
deposit(BigDecimal)
withdraw(BigDecimal)
```

The home interface is not inherited, but all the methods of the *BankAccount* home are copied to the *Savings* home:

```
create(String, BigDecimal)
findByPrimaryKey(BankAccountKey)
findAccountsWithBalanceBetween(BigDecimal, BigDecimal)
findAccountsWithBalanceGreater(BigDecimal)
findAccountsWithBalanceGreaterThan(BigDecimal)
findAll()
findGoldAccounts(BigDecimal)
```

Notice that all the custom finder methods have been copied, in addition to the standard create and findByPrimaryKey methods.

## Schema and map for a model with inheritance

We can create the database schema and map specific to the inheritance hierarchy in two ways: single table or root/leaf tables.

We first show how to define the single table model, but for our example, we will actually implement the root/leaf table model.

## Generating a single table database schema and map

Let's see what VisualAge for Java generates for an inheritance model.

**Attention:** If you want to perform the schema mapping to a single table model for illustration, first version the schema, model, entity beans, and packages, because this operation destroys the schema and mapping done for the Customer bean in “Using secondary tables” on page 321.

Version the existing implementation:

- ▶ Save the schema and mapping for the Bank\_Entities into itso.ejb35.bank.schema.
- ▶ Version the Customer entity bean in the EJB page.
- ▶ Version the itso.ejb35.bank and itso.ejb35.bank.schema packages.
- ▶ Delete the Bank\_Entities schema and map from the schema and map browser (they still exist in the repository and can be loaded again).

## Generate the single table model

To have VisualAge for Java generate the inheritance schema and mapping:

- ▶ Select the EJB group and *Add -> Schema and Map from EJB Group*.
- ▶ The generated schema and map appear in the schema and map browser. (If you closed the browsers, select the EJB group and *Open To -> Database Schemas* and *Open To -> Schema Maps*.)

## Schema

The generated schema is shown in Figure 16-6. Note that the table contains the fields for the bank account, as well as for the checking and savings account. In addition, there is a column named Bank\_Entities, the discriminator column that distinguishes between checking and savings accounts.

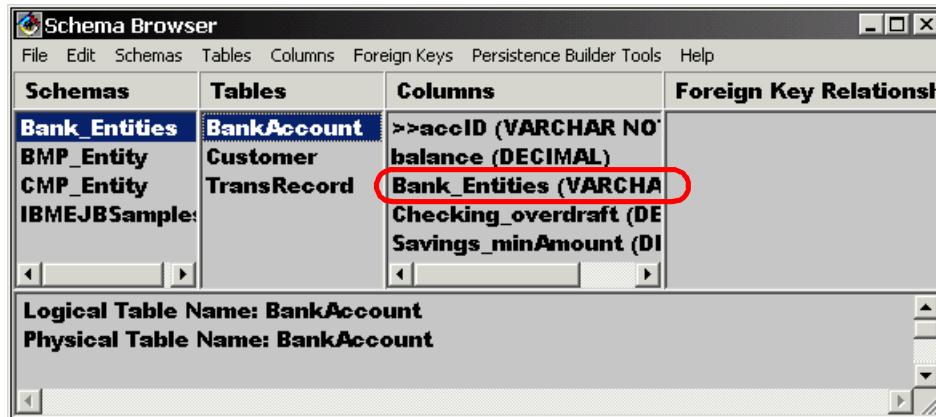


Figure 16-6 Single table schema

## Map

Figure 16-7 shows the generated mapping. All three classes map to the same table. If the parent class shows as BankAccount..., double-click on the parent class to see the subclasses.

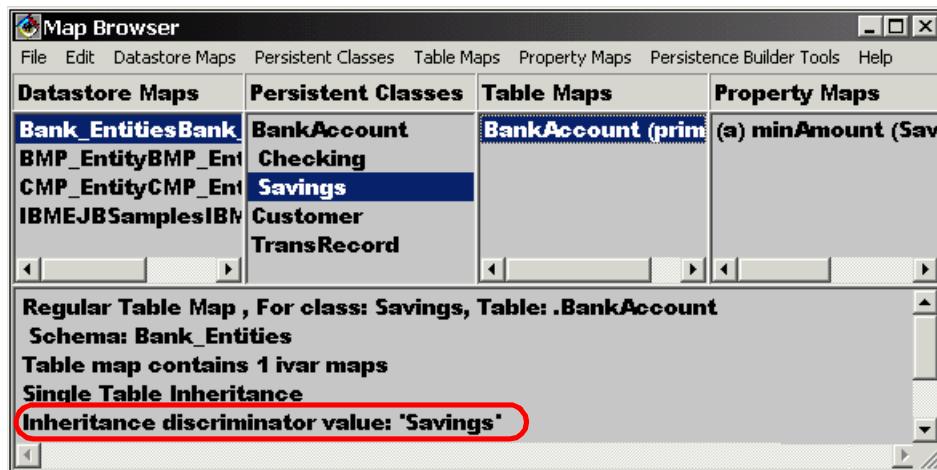


Figure 16-7 Single table mapping

The table map (*Table Maps -> Edit Table Map*) shows the table, the discriminator column, and the discriminator value. Each property map (*Table Maps -> Edit Property Map*) maps the CMP fields of its class, and also shows the parent's properties in gray (Figure 16-8).

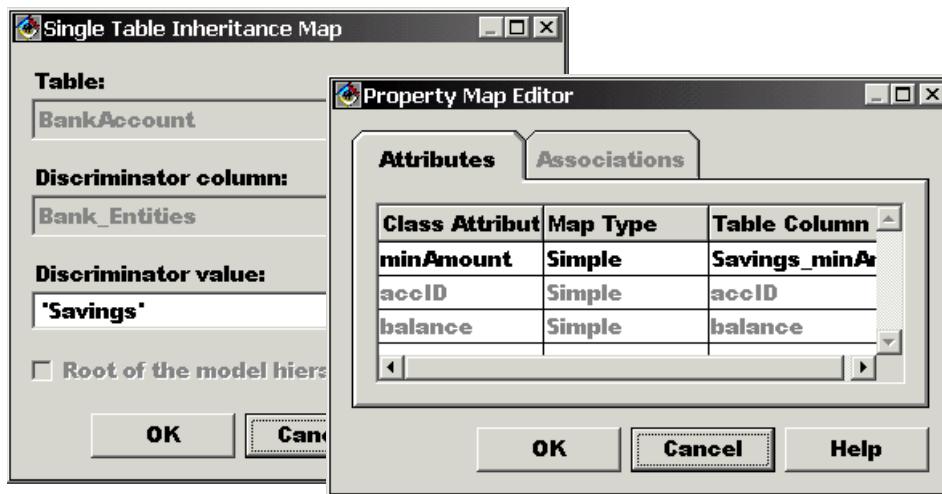


Figure 16-8 Single table and property maps

To map an inheritance structure to an existing table, you have to define the schema and map manually, similar to what we will describe now for root/leaf table maps.

### ***Restore the old version of schema and map***

To implement our own schema and mapping using root/leaf tables, we have to delete the generated schema and map, and reload the old version of the schema and map.

- ▶ Delete the Bank\_Entities schema and map from the schema and map browser.
- ▶ Reload the old schema: Select *Schemas -> Load Available Schemas*, select the Bank\_EntitiesSchema and click on **>>** to move the entry to the Selected Storage Classes pane. Click **OK**.
- ▶ Reload the old map: Select *DataStore Maps-> Load Available Maps*, select the Bank\_EntitiesMap and click on **>>** to move the entry to the Selected Storage Classes pane. Click **OK**.

## Creating a root/leaf table model

We have to create the root/leaf table model manually. We have already discussed that the root/leaf model has one table per class in the hierarchy. All the common fields are stored in the root table, and the specific fields of subclasses are stored in their individual tables.

### Schema

First, we create the tables for the account classes:

1. In the schema browser select the Bank\_Entities schema.
2. In the *Tables* section, invoke the context menu and select *New Table*. Define the ITS0.Account table with columns that match our existing table (Figure 16-9). Click *OK* when done.

accid           CHAR(8) not null (key)  
balance        DECIMAL(8,2) not null  
acctype        CHAR(10) not null

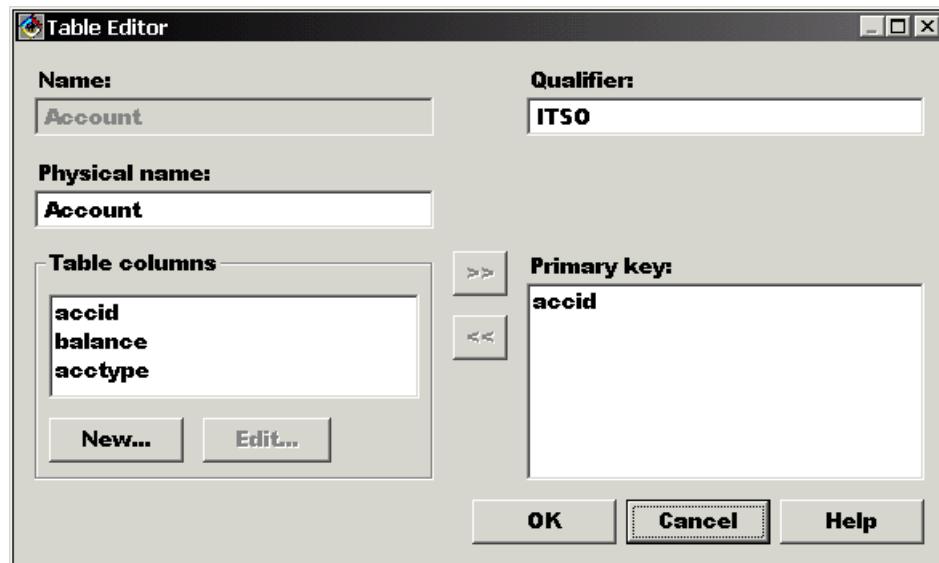


Figure 16-9 Root/leaf table definition

3. Create two more tables for Savings and Checking (Figure 16-10):
  - The ITS0.Savings table has columns accid — key CHAR(8) not null, and minamount — DECIMAL(8,2) not null.
  - The ITS0.Checking table has columns accid — key CHAR(8) not null, and overdraft — DECIMAL(8,2) not null.

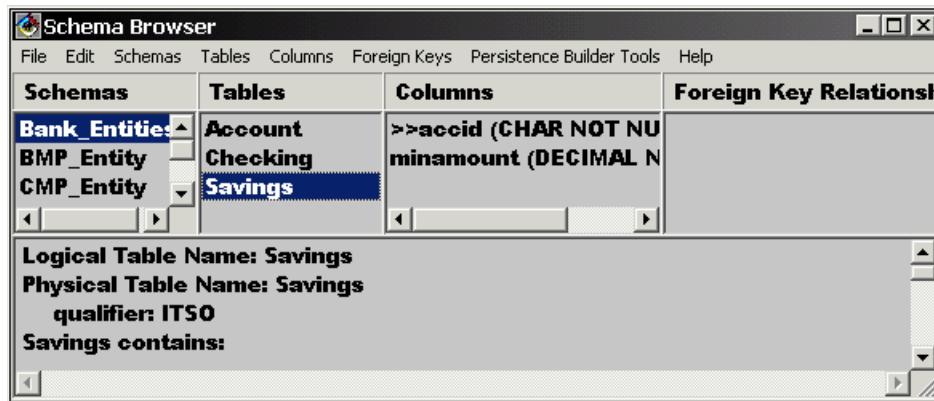


Figure 16-10 Root/leaf subclass tables

4. We have to specify the foreign key relationships for the Checking and Savings tables with the Account table. Select the Savings table and *Foreign Keys -> New Foreign Key Relationship* (Figure 16-11).



Figure 16-11 Checking account foreign key relationship

- Enter a name, select the primary table (Account) and the foreign key table (Savings). Click on the Foreign Key field and select the foreign key column (accid) from the pull-down.
  - Ensure that the *Constraint exists in Database* check box is selected, if your tables have been set up with foreign key constraints.
5. Create the foreign key relationship between the Account and Checking tables in the same way. The foreign keys appear now in the schema browser.

## Map

To create the data map for the account tables:

1. Select the Bank\_Entities map. In the *Persistent Classes section*, the list of classes are displayed.
2. Select the BankAccount class and *Table Maps -> New Table Map -> Add Root/leaf Inheritance Table Map*. The Root/Leaf Inheritance Table Map dialog appears (Figure 16-12). Select the table, discriminator column and enter a discriminator value. This is necessary even for the parent table.

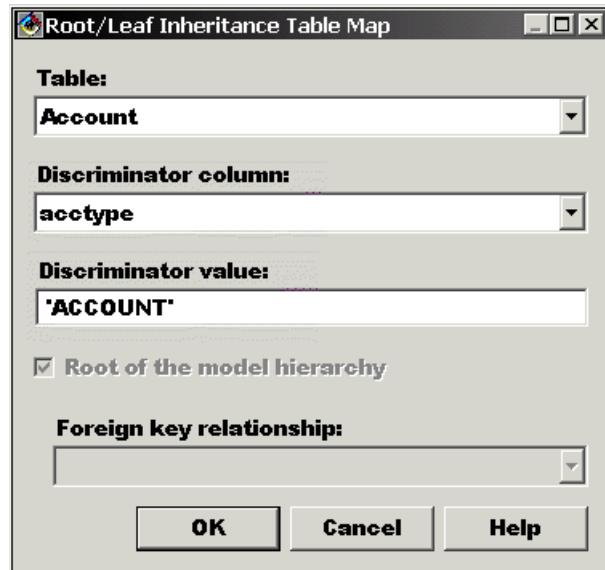


Figure 16-12 Root/leaf inheritance table map

3. Perform the same step for the subclasses. The discriminator column is grayed (cannot be changed), but you have to select the correct foreign key relationship (Figure 16-13).



Figure 16-13 Root/leaf inheritance table map for a subclass

4. Select each table map and *Table Maps -> Edit Property Maps*. Map each field to the matching table column (Figure 16-14). Note that the discriminator column is not mapped; it is not a CMP field of the bean.

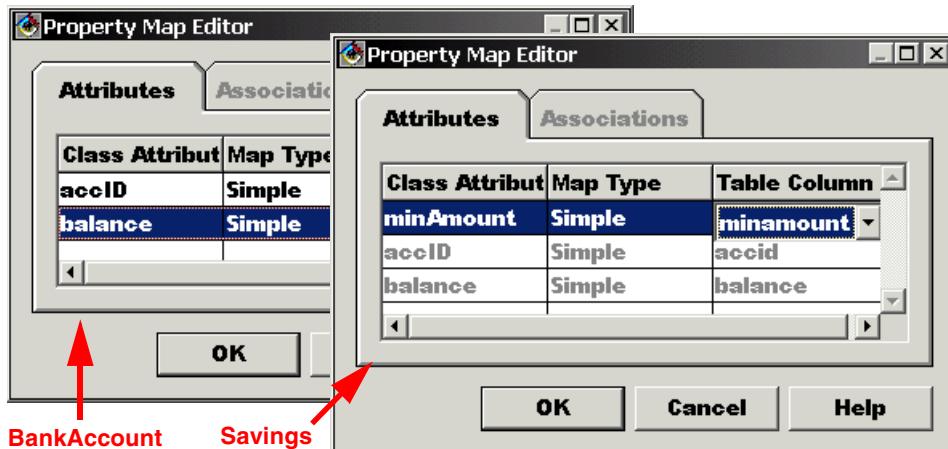


Figure 16-14 Root/leaf inheritance property map

Save the schema and the map into the `itso.ejb35.bank.schema` package and close the browsers.

## Tailoring the methods

The subclasses inherit the remote interface, and the parent's home methods were copied to the subclass home methods. In many cases we have to overwrite some of the methods.

### Create methods

Checking and savings account beans inherit the ejbCreate method from the parent. However, this method does not initialize the fields of the subclasses. You can overwrite the create methods and initialize the subclass fields. This is required if the database does not allow null values and no default was specified.

Here is a tailored method for the Checking bean:

```
public void ejbCreate(java.lang.String argAccID,
 java.math.BigDecimal argBalance)
 throws javax.ejb.CreateException, java.rmi.RemoteException {
 super.ejbCreate(argAccID,argBalance);
 _initLinks();
 // All CMP fields should be initialized here.
 overdraft = new java.math.BigDecimal(200.00);
}
```

Note that if you change the signature you have to promote the new method to the home interface.

### Business methods

Checking and savings account beans inherit the deposit and withdraw methods. The deposit method is universal, independent of the account type.

#### ***withdraw method***

The withdraw is different for the two account types. For a checking account the method should check the withdraw amount against the balance plus the overdraft amount, and for a savings account the check is against the balance minus the minimum amount.

Here is an extract of the tailored withdraw method for the checking account:

```
public void withdraw(java.math.BigDecimal amount)
 throws InsufficientFundException {
 if (balance.add(overdraft).compareTo(amount) == -1)
 throw new InsufficientFundException("Not enough funds");
 else
```

Similarly, the withdraw method of the savings account starts with:

```
public void withdraw(java.math.BigDecimal amount)
 throws InsufficientFundException {
 if (balance.subtract(minAmount).compareTo(amount) == -1)
 throw new InsufficientFundException("Not enough funds");
 else
```

### ***Finder methods***

The finder helper interface has been generated for the subclasses, for example, CheckingBeanFinderHelper, and the method signatures have been added to the home interface.

For the custom finder methods you have to implement the BeanFinderObject also for the subclasses, for example, CheckingBeanFinderObject and SavingsBeanFinderObject. Simply implement the classes as subclasses of the BankAccountBeanFinderObject:

```
public class CheckingBeanFinderObject extends BankAccountBeanFinderObject
 implements CheckingBeanFinderHelper {}
```

Note that the subclass finder object must implement the subclass finder helper interface.

You do not have to copy or tailor the implementation methods, in our case the findGoldAccounts method. The method of the parent class will be used and it works for the subclasses as well.

If you do not require the custom finder on the subclass, delete the method signatures from the home interface.

## **Testing the Checking and Saving beans**

To test the new bank account with its subclasses:

- ▶ Add the Bank\_Entities to a server configuration.
- ▶ Make sure the deployed code is generated for the BankAccount, Checking, Savings, and TransRecord beans.

The TransRecord bean was mapped and generated in “Mapping the TransRecord bean” on page 320.

- ▶ Start the persistent name server, the EJB server, and the EJB test client.

Figure 16-15 shows the EJB test client displaying a savings account. Note that you can access the methods of the savings class as well as the methods of the superclass.

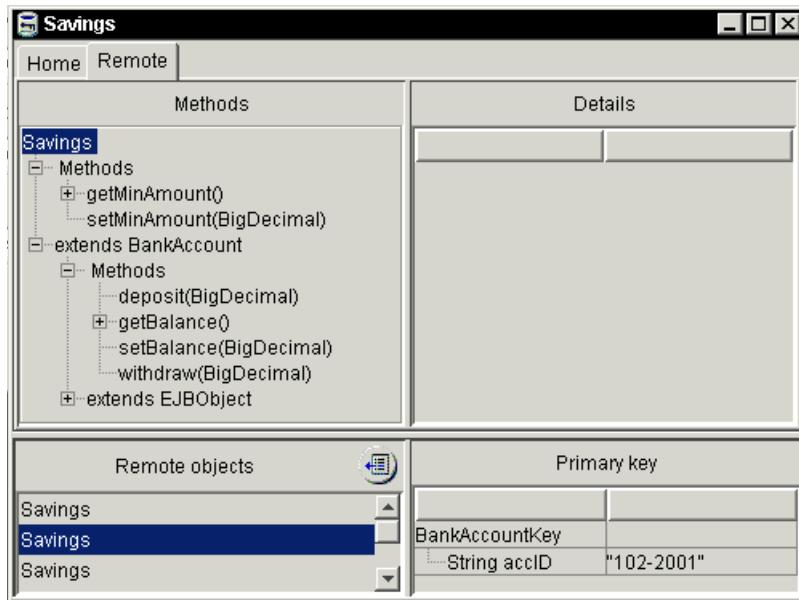


Figure 16-15 EJB test client with a savings account

Test the withdraw method to verify that the tailored method of the savings account is invoked.

Use the finder methods in the home interface. They are available in the subclasses and in the superclass.

### ***Checking or savings account?***

When accessing an account through the BankAccount bean, you cannot immediately see if it is a checking or savings account. Select the BankAccount and *Selected -> View as actual type* to see the object as a checking (or savings) account (Figure 16-16). This also enables you to invoke methods specific to the subclass.

However, even if you invoke the withdraw method on the BankAccount object, the tailored withdraw method of the checking or savings account is invoked.

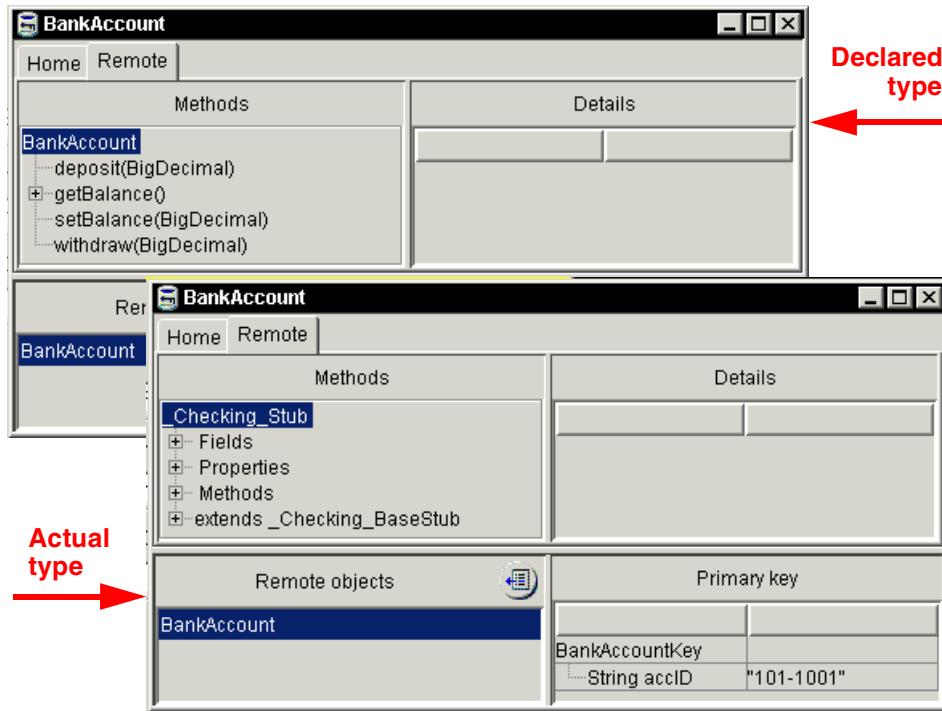


Figure 16-16 BankAccount or Checking object

## Java inheritance

In this section we discuss developing or creating enterprise beans using the standard Java inheritance model and we show how VisualAge for Java supports this.

### Developing using Java inheritance

Assume that we have an Account class that implements the AccountInterface, in this small example a calculateInterest method (Figure 16-17).

We want to implement CMP entity beans that are subclasses of account.

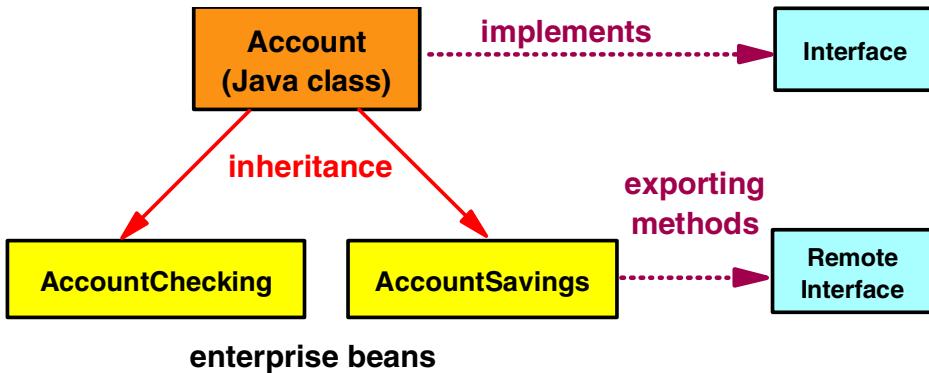


Figure 16-17 Java inheritance hierarchy example

To implement Java inheritance follow these steps:

1. Define the superclass Account, its interface AccountInterface, and implement the calculateInterest method:

```

package itso.ejb35.javainherit;

import java.math.BigDecimal;
public interface AccountInterface {
 BigDecimal calculateInterest(BigDecimal balance);
}

import java.math.BigDecimal;
public class Account {
 private fieldInterestRate = new BigDecimal(0.05);
 public Account() { super(); }
 public BigDecimal calculateInterest(BigDecimal balance) {
 /* Perform the calculateInterest method. */
 return balance.multiply(getInterestRate());
 }
 public BigDecimal getInterestRate() { return fieldInterestRate; }
 public void setInterestRate(BigDecimal interestRate) {
 fieldInterestRate = interestRate;
 }
}

```

2. Create an EJB group named Java\_Inherit and add an enterprise bean named AccountChecking (Figure 16-18).

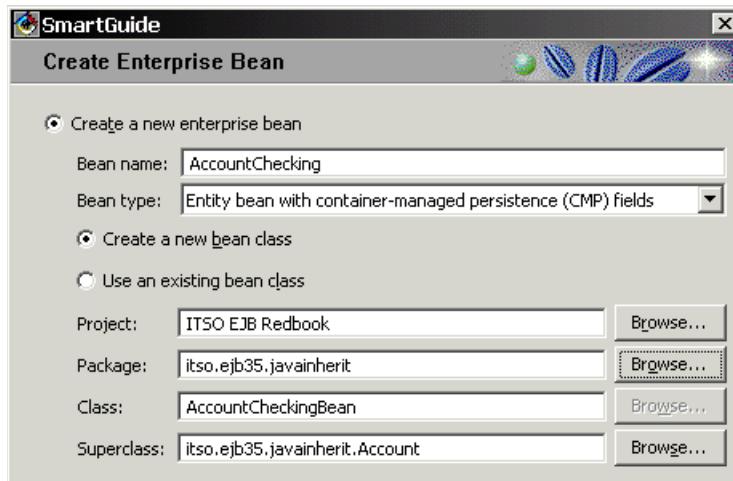


Figure 16-18 Entity bean with Java inheritance

3. Be sure to select the Account superclass. Click *Next*.
4. Define the CMP fields *accID* (string, key), *balance* (BigDecimal), and *accType* (string). Click *Finish* and the bean classes are created.
5. For the AccountChecking bean promote the *calculateInterest* method, which was inherited from the Account class, to the remote interface. Use the icon to show the inherited methods (or *Members -> Inheritance Filters*).  
This enables us to invoke the *calculateInterest* method for bean instances.
6. Create a schema (table) and a mapping (*Add -> Schema and Map from EJB Group*). Note that with Java inheritance each bean must have its own table; single-table or root/leaf inheritance is not supported.

We did not further develop this example.

# Summary

In this chapter we discussed creating enterprise beans with inheritance. Inheritance is not part of EJB specification; it is a VisualAge for Java feature, which helps in realizing the object reusability model in enterprise beans, because EJB as an object-oriented technology may use object inheritance and code reuse to reduce development cost.

EJB groups are used to group related EJBs, such as beans created using inheritance or association.

## Pros and cons

The Java inheritance model is useful when we have a Java class which already implements the logic of the bean. So, as an general merit of inheritance, it helps in code reusability. But this model is not useful for the database mapping point of view, because the underlying tables do not reflect the inheritance. This lacks the advantage of the root/leaf or single table model which can make use of an existing database schema with one table or foreign key relationships.





# Associations

In this chapter we describe how to implement associations (relationships) between entity beans using VisualAge for Java.

This support is an IBM extension of the EJB specification. In VisualAge for Java we can define the associations and have the deployed code generated for installation of the entity beans in a WebSphere container. The generated code provides special methods to retrieve related entity beans, that is, to follow an associations from one bean to a related bean.

A good source of information about entity bean association support is Chapter 7. Associations in the redbook *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*, SG24-5754.

## Association support in VisualAge for Java

The association support for entity beans has these restrictions:

- ▶ All entities must be in one entity group
- ▶ All entities must be container-managed (no bean-managed entities)

VisualAge for Java supports 1:1, 1:m, and m:m associations. Foreign keys are used in the underlying database tables to implement the association.

For an m:m association, an intermediate table, and also an intermediate entity bean, is required. This means that the entity model must be expanded and the client applications have to deal with an additional entity that is not really part of the design. Custom finders can help for this situation (see “Finder method for m:m association” on page 365).

## Reviewing the bank model

The bank model (Figure 17-1) requires two associations: an m:m association between Customer and BankAccount, and a 1:m association between BankAccount and TransRecord.

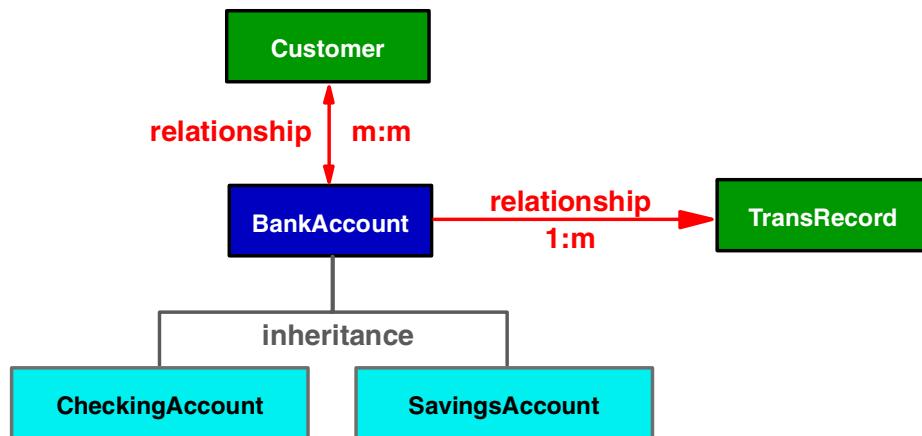


Figure 17-1 Bank model associations

In Chapter 16. “Inheritance” on page 329, we implemented the inheritance hierarchy; now we are implementing the associations between the entity beans.

# 1:m association

A 1:m association between two entity beans (BankAccount and TransRecord) means that there can be many instances of one entity bean (TransRecord) related to one instance of the other entity bean (BankAccount).

An association requires a foreign key in the underlying table. In our case, the TransRecord table carries the bank account number (accID) as a foreign key.

## Optional or mandatory

A transaction record cannot exist without a bank account. Such a rule can easily be implemented in the database: the foreign key is defined as *not null*. Any attempt to create a transaction record without an associated bank account will fail with an exception. The client application should handle this properly, the system can hardly generate the correct associated entity. (It is possible to modify the generated code to create the associated entity, but will it have reasonable attribute values?)

Are transaction records mandatory, or can a bank account exist without transaction records? We can set a rule that when a bank account is created, a transaction record for the original balance must be created. VisualAge for Java does not generate code to automatically add associated entities; such extensions must be added manually. Again, this could be handled by the client applications; but in this case it may be better to let the system perform the operation for all clients that create bank accounts.

**Attention:** Mandatory associations require user code in most cases.

## Association editor

Associations between entities are defined on the EJB page in the Workbench. For existing entities with deployed code, we suggest to delete the deployed code first, it has to be regenerated later. Select the beans (BankAccount, Checking, Savings, TransRecord, and Customer) and select *EJB -> Delete*, then click on *Deployed Code*.

Select the BankAccount bean and *EJB -> Add -> Association*. This action opens the Association Editor (Figure 17-2), where we specify the characteristics of the relationship between bank account and transaction records.

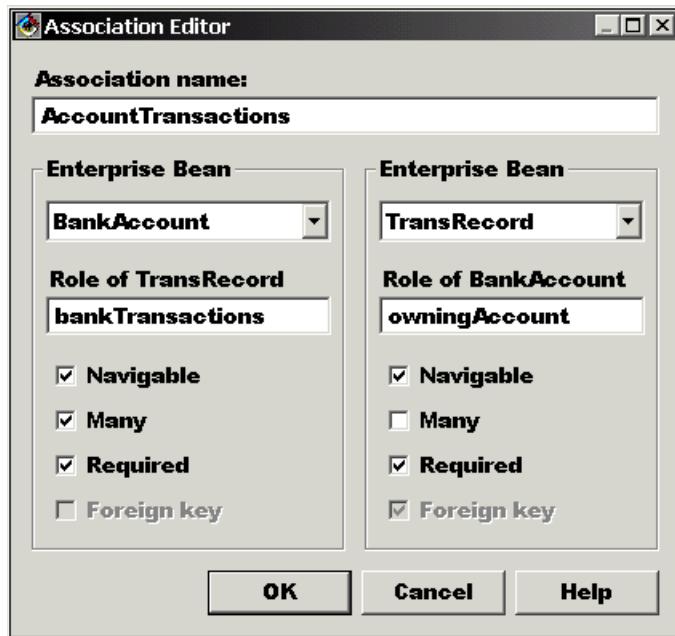


Figure 17-2 Association editor

The fields in the Association Editor are:

- ▶ Association name—shows up as a property of the entity group.
- ▶ Enterprise bean—the two beans that participate in the association.
- ▶ Role—will be used to generate the methods in the entity beans:
  - For BankAccount, TransRecord instances are bankTransactions; a getBankTransactions method will be generated.
  - For TransRecord, the BankAccount instance is the owningAccount; a getOwningAccount method will be generated.
- ▶ Navigable—a method to retrieve the related instances is generated.
- ▶ Many—there can be many related instances (a BankAccount can have many TransRecord, a TransRecords can only have one BankAccount).
- ▶ Required—there must be a related instance, association method does not return *null* (a BankAccount must have at least one TransRecord, and there must be a BankAccount for a TransRecord).
- ▶ Foreign key—which entity table carries the foreign key (TransRecord). This field is set automatically on the other side when you select *Many*. Only in a 1:1 association can you decide where the foreign key is located.

Notes:

- ▶ Do not select *Many* on both sides, this is not supported. m:m associations must be defined as two 1:m associations.
- ▶ The simplest *Roles* would be the names of the other entities in lower case and plural where there are many; for example, *transRecords* (*getTransRecords*) and *bankAccount* (*getBankAccount*).

Click *OK* to generate the association. The entities will carry red flags, because the schema/mapping are invalid at this point.

## Entity properties

The associations show up in the Properties pane of the EJB page (Figure 17-3).

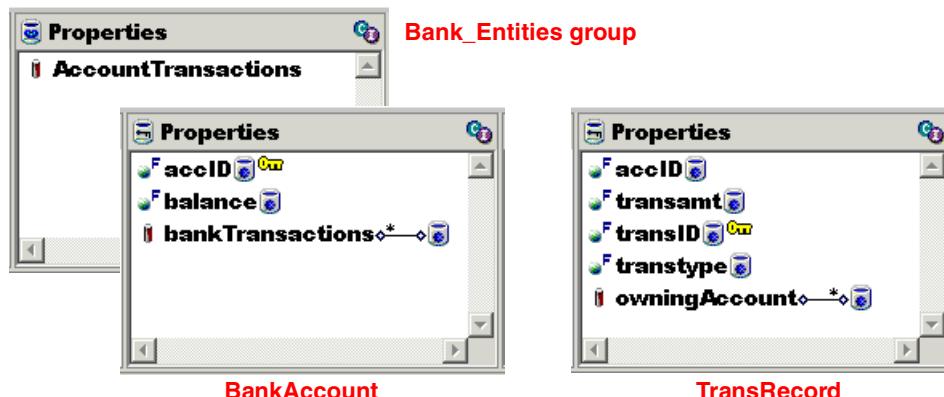


Figure 17-3 Associations as properties

## Generated methods and classes

Defining an association generates the access methods into the beans and interfaces.

### BankAccount methods

These methods are generated into the BankAccount bean and remote interface:

- ▶ The *getBankTransactions* method returns an Enumeration of related TransRecord beans.
- ▶ The *addBankTransactions(TransRecord)* method adds a transaction record to the account. This action sets the foreign key field.
- ▶ The *getBankTransactionsLink* method retrieves a utility class that maintains the association.

- ▶ There are two utility methods (`secondaryAddBankTransactions` and `secondaryRemoveBankTransactions`) used for inverse maintenance of the association.

## TransRecord methods

These methods are generated into the TransRecord bean and remote interface:

- ▶ The `getOwningAccount` method retrieves the related BankAccount bean.
- ▶ The `getOwningAccountKey` method retrieves the related BankAccountKey.
- ▶ The `setOwningAccount(BankAccount)` methods sets the account the owns the transaction record.
- ▶ The `getOwningAccountLink` method retrieves a utility class that maintains the association.
- ▶ There are two utility methods (`privateSetOwningAccountKey` and `secondarySetOwningAccount`) used for inverse maintenance of the association.

## Helper classes

Two helper classes, `BankAccountToBankTransactionsLink` and `TransRecordToOwningAccountLink`, are used to maintain the association.

## Helper methods

There are other helper methods in the entity beans, such as `_getLinks`, `_initLinks`, and `_removeLinks`. These methods already exist in beans without associations, but they are modified now and include references to the association that we defined.

## Schema and mapping

We already defined the underlying tables, but without the foreign key relationships that support the associations.

## Schema

Open the schema browser, select the `Bank_Entities` group and *Foreign Keys -> New Foreign Key Relationship*.

In the Foreign Key Relationship Editor, enter a name (`AccountTransrecord` matches our DDL), the two related tables, and the foreign key field (`accid`) that points to the primary table (Figure 17-4).

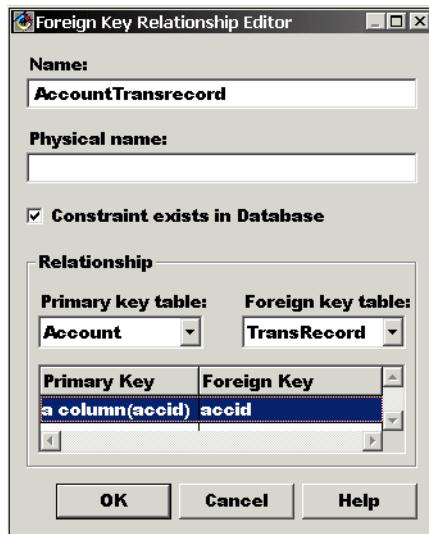


Figure 17-4 Foreign key relationship in the transaction record

All foreign key relationships are listed in the schema browser when no table is selected. With a table selected, only the foreign key relationships that involve the selected table are listed.

### Mapping the BankAccount entity

Open the map browser, select the Bank\_Entities group. Select the BankAccount class, the Account table map, and *Table Maps -> Edit Property Maps*.

Switch to the *Associations* tab and map the bankTransactions associations to the AccountTransrecord foreign key (Figure 17-5).

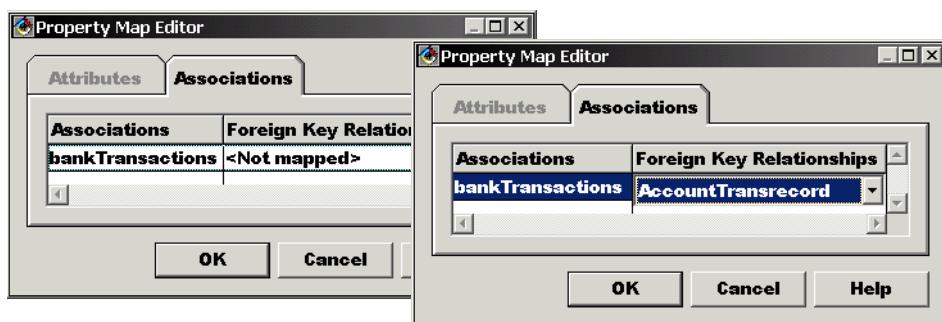


Figure 17-5 Mapping the BankAccount - TransRecord association

## Mapping the TransRecord entity

In the map browser select the TransRecord class and map. Edit the property map and on the Associations tab map the owningAccount association to the AccountTransrecord foreign key (same as for the Account map).

This action results in an error message in the bottom pane:

```
VapValidationException: in a VapRelationshipMap(TransRecord.owningAccount:
'AccountTransactions' <-> 'AccountTransrecord') at (4/17/2001 9:56:00 AM)
'Columns of the map from role owningAccount of TransRecord to
AccountTransrecord overlap with the map from attribute accID of TransRecord
to accid'
```

We are now using the accid field for the association, therefore we have to remove it from the entity bean.

## Fixing the TransRecord entity

In the EJB page, in the Properties pane, select the accid property and delete it (*EJB -> Delete*). This leaves an error in the ejbCreate method. Change the ejbCreate method to set the owningAccount association:

```
public void ejbCreate(String anAccid, java.math.BigDecimal anAmount,
 String aTranstype)
 throws javax.ejb.CreateException, java.rmi.RemoteException {
 _initLinks();
 // All CMP fields should be initialized here.
 transID = new java.sql.Timestamp(System.currentTimeMillis());
 transtype = aTranstype;
 transamt = anAmount;
 privateSetOwningAccountKey(new BankAccountKey(anAccid));
}
```

Add another ejbCreate method that uses a BankAccount as parameter:

```
public void ejbCreate(BankAccount acct, java.math.BigDecimal anAmount,
 String aTranstype)
 throws javax.ejb.CreateException, java.rmi.RemoteException {
 _initLinks();
 // All CMP fields should be initialized here.
 transID = new java.sql.Timestamp(System.currentTimeMillis());
 transtype = aTranstype;
 transamt = anAmount;
 setOwningAccount(acct);
}
```

These changes ensures that every new transaction record belongs to an account. Add the new ejbCreate method to the home interface (*Add To -> EJB Home Interface*).

Now the TransRecord bean shows a broken schema or map, and errors appear for the account beans (BankAccount, Checking, Savings).

### **Fixing the map of the TransRecord bean**

In the map browser the table map is now broken. Select the broken property map (for the accid property) and *Property Maps -> Delete Property Map*.

The TransRecord bean is now complete, it does not show any errors.

### **Changing the BankAccount and its subclasses**

The deposit and withdraw methods call the create method of the TransRecord bean. We could change these methods to use the new constructor, for example:

```
BankAccount: deposit
==> //txRechomeCMP.create(accID, amount, "D"); // old code, works as well
==> txRechomeCMP.create((BankAccount)(getEntityContext().getEJBObject()),
 amount, "D");
```

(Similar code in the withdraw methods of BankAccount, Checking, Savings)

## **Generating the deployed code**

Always save the schema and map when finished with one step. If anything brakes, it is much easier to restore VisualAge for Java.

We could generate the deployed code now, but let's first define the m:m association between Customer and BankAccount.

## **m:m association**

An m:m association between two entity beans means that there can be many instances of an entity bean on one side related to one instance of the other entity bean. In our example, a customer can own many bank accounts, and a bank account can be owned by many customers (shared accounts).

In the current implementation an m:m association must be implemented by an intermediate entity bean and two 1:m associations. The intermediate entity bean does not carry any attributes (Figure 17-6).



Figure 17-6 m:m relationship implementation

Each association requires a foreign key in the underlying table. In our case, the CustAcct table carries the customer ID and the bank account number as foreign keys, and together they form the primary key.

Both associations are required, that is, a CustAcct instance cannot exist without both a Customer and a BankAccount instance.

## Define the intermediate entity and two 1:m associations

In the EJB page, define a new entity bean:

- ▶ Name: CustAcct
- ▶ Type: entity bean, container-managed
- ▶ Package: itso.ejb35.bank
- ▶ No CMP fields (there will be associations)

This entity bean shows errors, because there is no key field defined yet.

## Define the two associations

In the EJB page, define two 1:m associations (Figure 17-7).

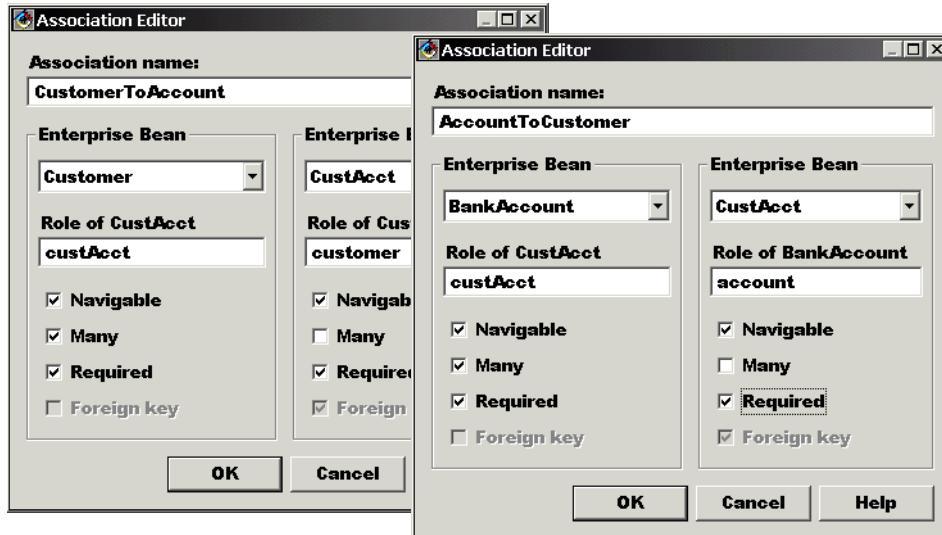


Figure 17-7 m:m as two 1:m associations

Notice that the role names become method names:

- ▶ Customer: getCustAcct
- ▶ Account: getCustomer
- ▶ CustAcct: getCustomer and getAccount

## Complete the intermediate entity

Select the *CustAcct* entity bean and switch to the Properties pane. For each association role, select *Properties* -> *Add Role To Key* to add the two foreign key fields to the primary key (Figure 17-8).



Figure 17-8 Associations as primary key

## Define the schema for an m:m association

In the schema browser define a new table for the intermediate *CustAcct* entity (Figure 17-9). Note that both fields form the primary key.

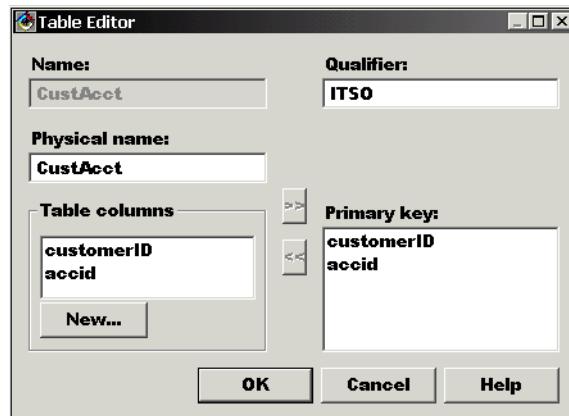


Figure 17-9 Table for intermediate entity

Define two foreign key relationships (Figure 17-10). Make sure that the names are clear:

- ▶ CAtoCustomer—from the intermediate *CustAcct* table to the *Customer* table
- ▶ CAtoAccount—from the intermediate *CustAcct* table to the *Account* table

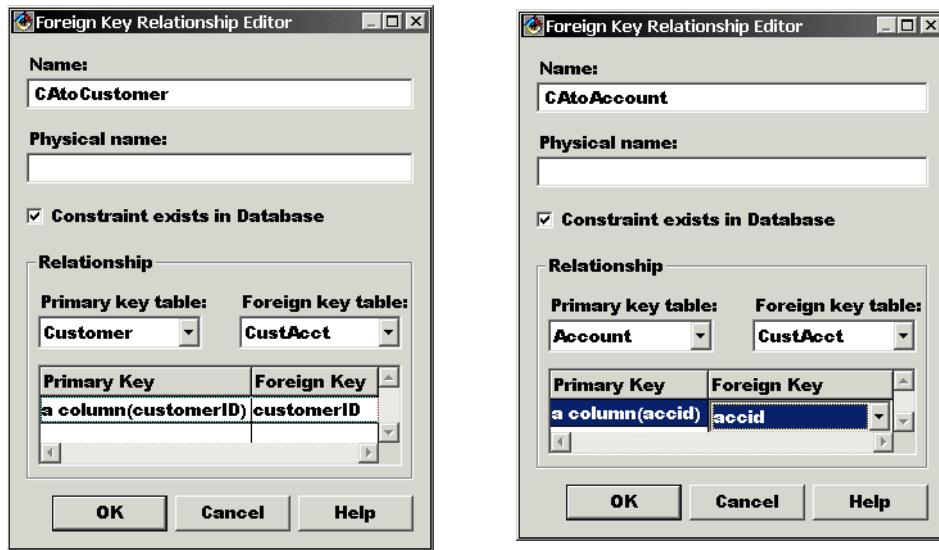


Figure 17-10 Foreign keys for m:m association

## Define the map for an m:m association

In the map browser select the new CustAcct class. Define a new table map (*New Table Map -> Add Table Map with No Inheritance*) and map the class to the CustAcct table. There are no attributes to map, only two associations (Figure 17-11).



Figure 17-11 Association mapping for the intermediate entity

Select the Account table (for the BankAccount) and *Edit Property Maps*. Map the custAcct association to the CAtoAccount foreign key. Select the Customer table and map the custAcct association to the CAtoCustomer foreign key.

Save the schema and map. There should be no mapping errors in the EJB page.

## Generated methods for m:m association

The Customer and the BankAccount beans have a getCustomer method to retrieve the intermediate beans. Both methods return an Enumeration.

The CustAcct bean has getCustomer and getAccount methods to retrieve the target entities, as well as setCustomer and setAccount methods to set the related entities when creating (or updating) an intermediate object.

To traverse from a Customer instance to all the BankAccount instances, a client program has to retrieve the intermediate objects using getCustomer, then loop through the returned Enumeration and retrieve each account using the getAccount method. We will simplify this process with a tailored finder method.

## Finder method for m:m association

We introduced finder methods in Chapter 10. “Custom finder methods” on page 193, and after explaining simple finder methods, we started to elaborate on a finder method for a complex association in “Finders and complex associations” on page 203.

We now describe how to implement this complex finder method in the BankAccount bean. Notice that the finder method to traverse from a Customer object to its BankAccount objects is implemented in the BankAccount bean, and not in the Customer bean. Finder methods are implemented in the target bean.

## Home and finder helper interfaces

The BankAccountHome interface contains the appropriate method signature:

```
public java.util.Enumeration findAccountsForCustomer(int aCustomerId)
 throws java.rmi.RemoteException, javax.ejb.FinderException;
```

Add the same method signature to the CheckingHome and SavingsHome.

The BankAccountBeanFinderHelper interface contains the signature for the corresponding finder method:

```
public java.sql.PreparedStatement findAccountsForCustomer
 (int aCustomerId) throws Exception;
```

There is no need to add this signature to the CheckingBeanFinderHelper and SavingsBeanFinderHelper interfaces, because those inherit from the BankAccountBeanFinderHelper.

## Implementation in the finder object

The finder object (BankAccountBeanFinderObject) builds and caches the query string for the finder as well as implements the finder method.

### Define the cached query string

```
public class BankAccountBeanFinderObject extends VapEJSJDBCFinderObject
 implements BankAccountBeanFinderHelper {
 private String cachedFindAccountsForCustomerQueryString = null;
 ...
}
```

### Build the cached query string

Through lazy initialization in the finder object, the accessor method for the query string field (Figure 17-12) builds up the query string by first merging the where condition into the query template and then adding a reference to the intermediate table into the from clause.

```
public class BankAccountBeanFinderObject {
.....
protected String getFindAccountsForCustomerQueryString() {
 if (cachedFindAccountsForCustomerQueryString == null) {
 // Do the WHERE first
 // so that the genericFindInsertPoints are correct.
 int i;
 int[] genericFindInsertPoints = getGenericFindInsertPoints();
 StringBuffer sb = new StringBuffer(getGenericFindSqlString());
 for (i = 0; i < genericFindInsertPoints.length; i++) {
 sb.insert(genericFindInsertPoints[i],
 "(T1.accID = T4.accID) AND (T4.CustomerID = ?)");
 }
 // Make sure to update every FROM clause.
 String soFar = sb.toString();
 int fromOffset = soFar.indexOf(" FROM ");
 while (fromOffset != -1) {
 sb.insert((fromOffset+5), " ITSO.CustAcct T4, ");
 soFar = sb.toString();
 fromOffset = soFar.indexOf(" FROM ", (fromOffset+5));
 }
 cachedFindAccountsForCustomerQueryString = sb.toString();
 }
 return cachedFindAccountsForCustomerQueryString;
}
}
```

Figure 17-12 Method to create the merged select statement

The first half of the accessor method uses a genericFindInsertPoints array to locate and update each where clause. Then, the second half of the method counts forward from the beginning of each from clause, inserts the reference to the intermediate table into the query string as needed, and updates the query-string field.

After this method call, a simple query string would look like:

```
SELECT <columns> FROM CUSTACCT T4, ACCOUNT T1
WHERE ((T1.accID = T4.accID) AND (T4.customerID = ?))
```

But, because of inheritance it becomes much more complex:

```
SELECT T1.acctype, T1.balance, T1.accid, -1, T2.overdraft FROM
ITSO.CustAcct T4, ITSO.Account T1, ITSO.Checking T2 WHERE ((T1.accid =
T2.accid) AND (T1.acctype IN ('CHECKING'))) AND ((T1.accID = T4.accID) AND
(T4.CustomerID = ?))
UNION ALL (SELECT T1.acctype, T1.balance, T1.accid, -1, -1 FROM
ITSO.CustAcct T4, ITSO.Account T1 WHERE (T1.acctype IN ('ACCOUNT')) AND
(T1.accID = T4.accID) AND (T4.CustomerID = ?))
UNION ALL (SELECT T1.acctype, T1.balance, T1.accid, T3.minamount, -1 FROM
ITSO.CustAcct T4, ITSO.Savings T3, ITSO.Account T1 WHERE ((T1.accid =
T3.accid) AND (T1.acctype IN ('SAVINGS'))) AND ((T1.accID = T4.accID) AND
(T4.CustomerID = ?)))
```

Note that you have to choose an alias name (T4) that is not used already in the query because of inheritance.

In the finder object, the finder method implementation uses the query string to create a PreparedStatement. Finally, the customer ID value is added into each where clause by using the superclass method getMergedWhereCount in the iteration loop (Figure 17-13).

```
public class BankAccountBeanFinderObject {
 public java.sql.PreparedStatement findAccountsForCustomer(int customerID)
 throws Exception {
 // Get the full query string and make a PreparedStatement.
 java.sql.PreparedStatement ps = getPreparedStatement(
 getFindAccountsForCustomerQueryString());
 // Inject the product id parameter into each merged where clause.
 for (int i = 0; i < getMergedWhereCount(); i++) {
 ps.setInt(i+1, customerID);
 //ps.setString(i+2,nextStringArgument); // if more arguments
 //ps.setDouble(i+3,nextDoubleArgument); }
 return ps;
 }
```

Figure 17-13 Finder method to traversing m:m associations

## Banking session bean

To implement simple banking transactions we create a stateless session bean, similar to the Transfer session bean described in “Creating a session bean” on page 174.

Create a new session bean in the **Bank\_Entities** group (in a real installation you should use a different group):

- ▶ Enter **Banking** as the name, and `itso.ejb35.bank` as package.
- ▶ Import the packages `javax.naming.*` and `java.rmi.*`, and the type `java.math.BigDecimal`. Click *Finish*.
- ▶ Create a private field for the home of bank accounts:

```
private BankAccountHome bankAccHome;
```

- ▶ Copy the `ejbCreate` method from the Transfer session bean, but change the JNDI name to `itso/ejb35/bank/BankAccount`.
- ▶ Copy the `transferMoney` methods from the Transfer session bean.
- ▶ Create three new methods: `getBalance`, `deposit`, and `withdraw`:

```
public BigDecimal getBalance(String anAccountID) {
 BankAccount anAccount = null;
 try { anAccount = bankAccHome.findByPrimaryKey
 (new BankAccountKey(anAccountID));
 return anAccount.getBalance();
 } catch(Exception exc) {
 exc.printStackTrace();
 System.out.println("Exception: an account could not be found!");
 return null;
 }
}

public BigDecimal deposit(String anAccountID, BigDecimal anAmount) {
 BankAccount anAccount = null;
 try { anAccount = bankAccHome.findByPrimaryKey
 (new BankAccountKey(anAccountID));
 anAccount.deposit(anAmount);
 return anAccount.getBalance();
 } catch(Exception exc) {
 System.out.println("Deposit on the account failed!");
 return null;
 }
}
```

```

public BigDecimal withdraw(String anAccountID, BigDecimal anAmount)
 throws itso.ejb35.util.InsufficientFundException {
 BankAccount anAccount = null;
 try { anAccount = bankAccHome.findByPrimaryKey
 (new BankAccountKey(anAccountID));
 anAccount.withdraw(anAmount);
 return anAccount.getBalance();
 } catch(Exception exc) {
 System.out.println("Withdraw on the account failed!");
 return null;
 }
}

```

- ▶ Promote the four business to the remote interface.
- ▶ Generate the deployed code.

## Create access beans for client programming

To simplify client programming, generate access beans for all entity beans in the `Bank_Entities` group.

Select each entity and *Add -> Access Bean*:

- ▶ Select to generate a rowset (which includes the copy helper)
- ▶ Select the `findByPrimaryKey` method for the no-argument constructor for `Customer`, `BankAccount`, `Checking`, and `Savings`.
- ▶ Select the `create` method for the no-argument constructor for `CustAcct` and `TransRecord`.
- ▶ Use no converters (do not convert `customerID` to a string).
- ▶ Select all attributes for the copy helper.

Depending on the sequence of operation you can get warning messages if an access bean for a related class does not exist yet. In a model with associations and inheritance you have to create access beans for all entity beans or for none.

Generate (or regenerate) the deployed code; copy helper beans change the remote interface.

Also create an access bean for the Banking session bean. For a session bean only the Java wrapper bean can be generated. The deployed code must not be regenerated.

## Entity properties

In the EJB page, open the properties for the BankAccount bean (Figure 17-14).

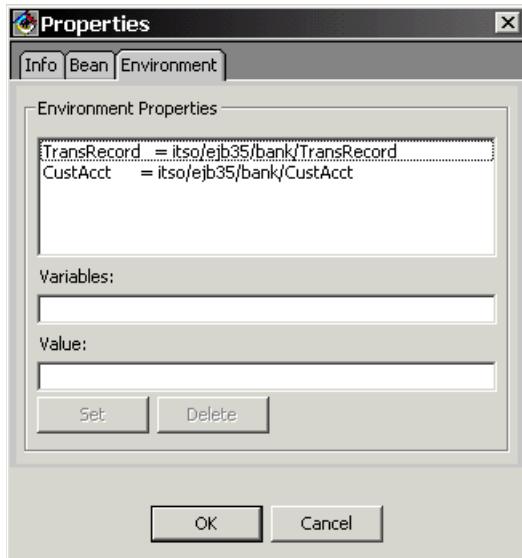


Figure 17-14 Entity properties with related home JNDI names

For the entity beans Customer, BankAccount, CustAcct, and TransRecord, the related home JNDI names are specified as environment variables.

However, this is not true for Checking and Savings bean; the environment is empty. When following a relationship to CustAcct or TransRecord from a checking or savings account, the home of the related entity is not found!

- ▶ For example, the `getTargetHome` method in the `BankAccountToBankTransactionsLink` class returns the wrong JNDI name:

```
public class BankAccountToBankTransactionsLink extends SingleToManyLink {
 private static TransRecordHome targetHome = null;
 private final static java.lang.String targetHomeName = "TransRecord";

 protected static synchronized TransRecordHome getTargetHome
 (com.ibm.ivj.ejb.associations.links.Link aLink)
 throws javax.naming.NamingException {
 if (targetHome == null) {
 String homeName = aLink.getEntityContext().getEnvironment()
 .getProperty(targetHomeName);
 if (homeName == null)
 homeName = targetHomeName;
 targetHome = (itso.ejb35.bank.TransRecordHome) lookupTargetHome
```

```

 (homeName, itso.ejb35.bank.TransRecordHome.class);
 }
 return targetHome;
}
....
```

Because the targetHome variable is initialized as "TransRecord" and the environment property is missing, the method fails with a NamingException.

- ▶ To fix this problem, open the properties of the Checking and Savings beans and set the same environment variables as for the BankAccount bean.

## Testing associations with the EJB test client

Bring up the WebSphere Test Environment and the persistent name server. Open a server configuration with the Bank\_Entities group, and start the server. Open the EJB test client. Then execute scenarios to go from entity to entity following the associations:

- ▶ Retrieve the home of the Customer bean.
- ▶ Retrieve a customer, for example, 101.
- ▶ Invoke the getCustomerAcct method; this should open the remote interface for the CustAcct bean and find two instances.
- ▶ Select an instance (101-1002,101) and invoke the getAccount method; this opens the remote interface for the target savings account.
- ▶ From any account instance invoke the getBankTransactions method; this opens the remote interface for the target TransRecord beans.
- ▶ In the home interface of the BankAccount, invoke the findAccountsForCustomer method with a customer ID of 101 or 106. This finder method should retrieve all the accounts for the given customer.
- ▶ Run the same method from the Checking home interface; it should work as well.

## Mandatory association

Earlier we decided that a new bank account would require a transaction record with the amount of the original balance. What we need to implement is to create a TransRecord instance during creation of the account instance.

You cannot perform this operation in the create method of the account, because the instance does not exist yet to connect a transaction record.

You have to implement the logic in the ejbPostCreate method that has the same signature as the create method. In the BankAccountBean, modify or create this method:

```
public void ejbPostCreate(java.lang.String argAccID,
 java.math.BigDecimal argBalance) throws java.rmi.RemoteException {
 try {
 setTxRecHome();
 txRecHomeCMP.create(accID, balance, "C");
 } catch (Exception e) {
 e.printStackTrace();
 }
}
```

You need a second method (in the BankAccountBean) for the create of a Savings instance, that has three parameters:

```
public void ejbPostCreate(java.lang.String argAccID,
 java.math.BigDecimal argBalance, java.math.BigDecimal minAmount)
 throws java.rmi.RemoteException {
 try {
 setTxRecHome();
 txRecHomeCMP.create(accID, balance, "C");
 } catch (Exception e) {
 e.printStackTrace();
 }
}
```

Note that we implement the ejbPostCreate methods in the BankAccountBean, one for every create method in the BankAccount and its subclasses.

## Deployment to WebSphere

Deployment of the entities with associations and inheritance can be done in the same way as described in Chapter 14. “Deployment to WebSphere Application Server Advanced” on page 277.

**Note:** We have to create a new EJB container, because the short names of the EJBs in the Bank\_Entities group are the same as in the CMP\_Entity group.

The first step is to create the deployed JAR file for the Bank\_Entities EJB group. Name the deployed JAR file `itso_bank.jar` and place it into the directory

```
<%WAS_HOME%>\deployedEJBs
==> d:\WebSphere\AppServer\deployedEJBs\itso_bank.jar
```

Be sure to click on the *Select referenced types and resources* button and let VisualAge for Java include referenced types.

**Attention:** Composers are not automatically included in the JAR file.

Click on *Details* for .class files, and manually add the AddressComposer class of the itso.ejb35.bank package. If you forget this and try to instantiate a Customer EJB, you will get this notice in the EJB test client (Figure 17-15).

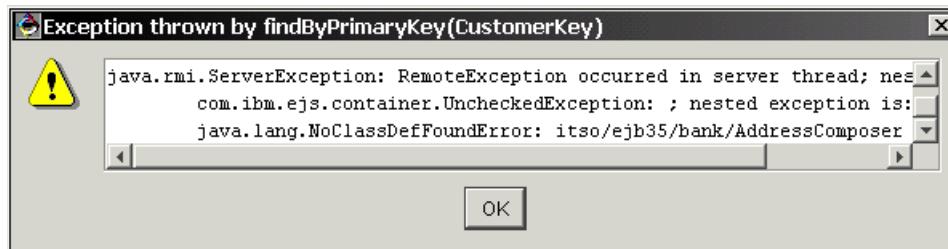


Figure 17-15 Exception when a composer is missing

## Using XMLConfig for deployment

Instead of installing the JAR file using the Administrative Console, we will use the XMLConfig tool to create an EJB container and install the EJBs.

XMLConfig is a WebSphere tool to configure the WebSphere Application Server using a batch utility instead of bringing up the Administrative Console. Configurations can be exported into XML files, and changes can be made to the configuration using XML import.

To see the format of an XML file, you can use the Administrative Console and export the current configuration. Then use an editor to extract the definitions of an EJB container and EJBs, and modify the file to create a new container and install EJBs into the container.

### Running the XMLConfig tool

The tool is started using the XMLConfig.bat file in the WebSphere BIN directory. The command has a number of options, but we will only use a simple form to import our EJBs:

```
set PATH=d:\WebSphere\AppServer\bin;%PATH%
xmlconfig -import itsoejb35bank.xml -adminNodeName CHUSA <===== tailor
```

The file is provided in sg246144\sampcode\was\xmlconfig. Figure 17-16 shows an extract of the XMLConfig file to define a new EJB container and the EJBs in the container.

```
<?xml version="1.0"?>
<!DOCTYPE websphere-sa-config SYSTEM
 "$server_root$$dsepbindsep$xmlconfig.dtd" >
<websphere-sa-config>
 <node name="CHUSA" action="locate"> <===== must tailor
 <application-server name="ITSO_EJB35" action="locate">
 <container name="ITSO_EJB35Bank" action="update">
 <user-id></user-id>
 <password></password>
 <cache-config>
 <size>2047</size>
 <soft-limit>2000</soft-limit>
 <hard-limit>2047</hard-limit>
 <sweep-interval>1000</sweep-interval>
 <passivation-directory></passivation-directory>
 </cache-config>
 <data-source name="EJBBANK"/>
 <ejb name="Customer" action="update">
 <jar-file>D:\WebSphere\AppServer\deployedEJBs\itso_bank.jar
 </jar-file>
 <home-name>itso/ejb35/bank/Customer</home-name>
 <user-id></user-id>
 <password></password>
 <create-db-table>false</create-db-table>
 <find-for-update>false</find-for-update>
 <minimum-pool-size>2</minimum-pool-size>
 <maximum-pool-size>100</maximum-pool-size>
 <primary-key-check>false</primary-key-check>
 <db-exclusive-access>false</db-exclusive-access>
 </ejb>
 <ejb name="BankAccount" action="update">
 <jar-file>D:\WebSphere\AppServer\deployedEJBs\itso_bank.jar
 </jar-file>
 <home-name>itso/ejb35/bank/BankAccount</home-name>

 </ejb>
 more EJBs
 </container>
 </application-server>
 </node>
</websphere-sa-config>
```

Figure 17-16 XMLConfig file to define a container and EJBs: itsoejb35bank.xml

### Notes:

- ▶ The WebSphere Admin Server must be running (start in Windows Services).
- ▶ The adminNodeName in the command is case sensitive and must match exactly the node name that is recorded in WebSphere; if it does not match, the utility gives an error message before doing any work.
- ▶ The structure of the XML tags must match the structure that is generated when the configuration is exported.
- ▶ The <node> and the <application-server> tags locate the correct application server. (One administration facility can administer multiple nodes.)
- ▶ The <container> tag creates a new EJB container.
- ▶ The <ejb> tags install the EJBs into the container. The tags point to the JAR file with the classes and define the JNDI name of the EJB.

After running the XMLConfig tool and opening the Administrative Console the new EJB container and its EJBs show up under the application server (Figure 17-17).

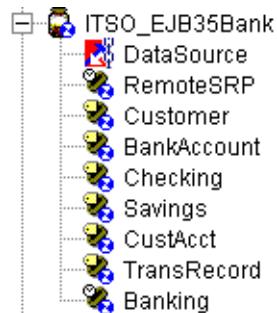


Figure 17-17 EJB container with EJBs after XMLConfig run

## Testing the EJBs in WebSphere

You can use the XMLConfig tool to start and stop the application server with its EJB containers and servlet engine. We provide sample XMLConfig files (see “Using the Web material” on page 492).

To test the deployed EJBs, start the EJB test client from VisualAge for Java and connect to the WebSphere system. You can use the test client in the same way as when working with the WebSphere Test Environment of VisualAge for Java.

For real client applications, see Chapter 18. “Client programming for inheritance and associations” on page 381.

## Reverse engineering from an existing database

In many installations you have existing databases that you want to access using modern technology, such as enterprise beans. VisualAge for Java provides the facility to import existing tables into the schema browser, and then converts the schema into entity beans. This facility supports foreign key relationships and they are converted into entity associations.

Let's see what entity beans we would get if we imported the EJBBANK database and converted it into an EJB model.

### Importing existing tables into a schema

Open the schema browser and select *Schemas -> Import/Export Schema -> Import Schema from Database*. A dialog prompts you for a schema name, for example, Bank\_Reverse.

In the Database Connection Info dialog, select the DB2AppDriver and enter the data source as jdbc:db2:ejbbank. Click *OK*.

Select the tables that you want to import (Figure 17-18). First select the qualifier (ITSO) and click on *Build Table List*; then select all the tables (or a subset) and click *OK*.

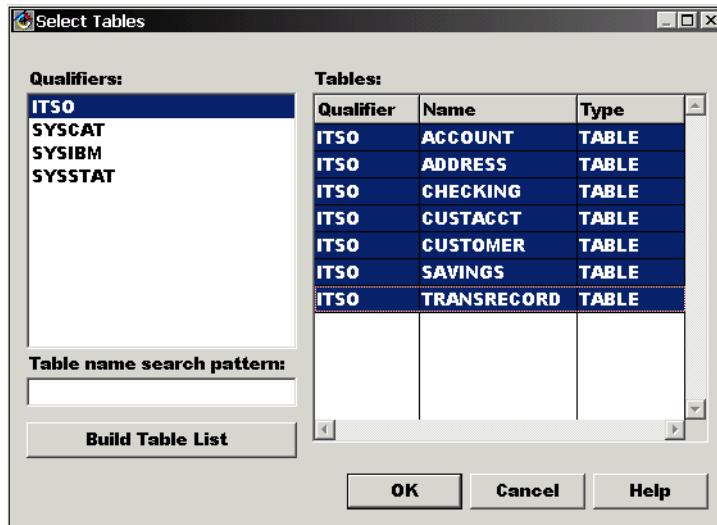


Figure 17-18 Import existing tables into a schema

The resulting schema appears in the schema browser with tables, columns, and foreign key relationships (Figure 17-19).

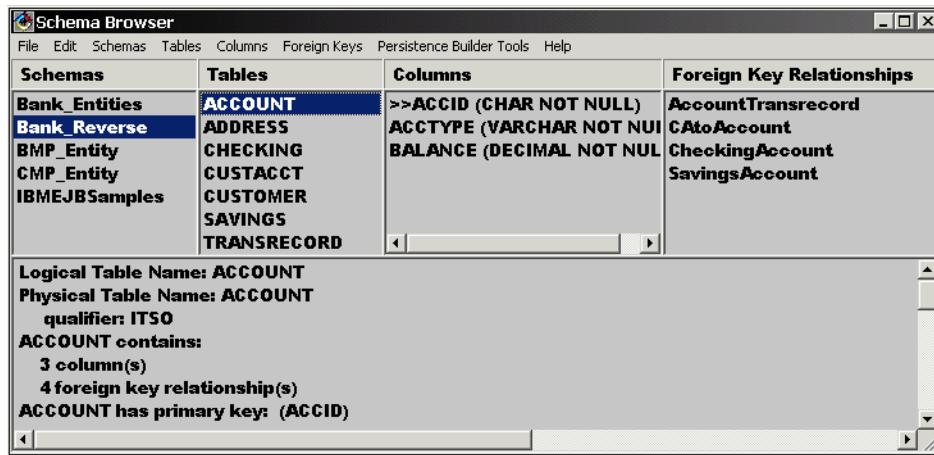


Figure 17-19 Imported schema

At this point the column and foreign key names could be tailored, but let's just see what kind of entity model we will get with the schema unchanged.

## Creating an EJB model from the schema

In the EJB page, select *EJB -> Add -> EJB Group from Schema or Model* and the Create EJB Group from Schema or Model opens (Figure 17-20).

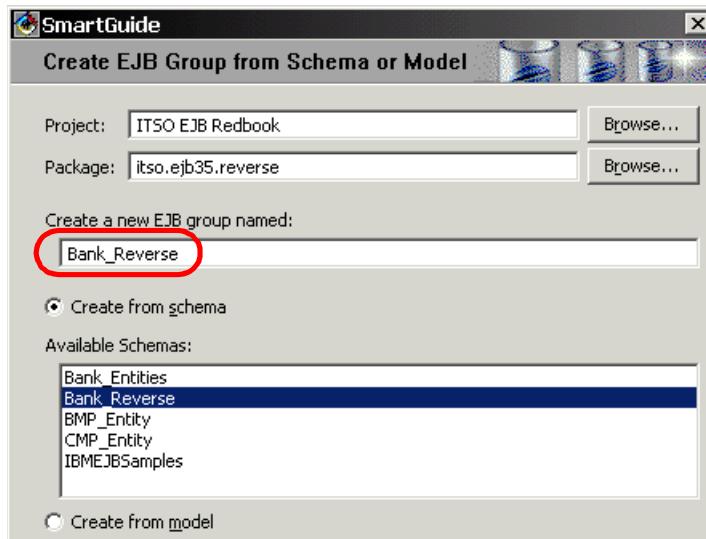


Figure 17-20 Create an EJB group form a schema

Select a project, enter a new package name (`itso.ejb35.reverse`) and a new EJB group name (`Bank_Reverse`) and select *Create from schema* (the other choice is to create an EJB group from a Persistence Builder model). Click on *Finish* and the EJB group is generated.

## EJB model

The EJB model consists of seven entities and six associations. There is one entity per table, and one association per foreign key.

The Customer entity reports: *Database schema or schema map has errors*. Opening the map reveals that the problem occurs because Customer has an address column, and an address foreign key. This can be fixed by:

- ▶ Change the ADDRESS column to ADDRESSX (the physical name is unchanged).
- ▶ Open the Address CustAddress Customer association and change the role of Address from address to theAddress (for example).

This process adjusts the mapping and the error disappears.

## Observations

The EJB model that is generated may work, but may not be very usable:

- ▶ Address is its own entity, associated with the Customer entity through a 1:1 association. This may be a good design if the address information is used infrequently.

In our tailored model the Customer entity mapped to both Customer and Address tables.

- ▶ The 1:m and m:m associations are generated perfectly.
- ▶ Inheritance is not generated from the schema; Checking and Savings are independent entities, associated with the customer through a 1:1 association.

We would have to delete the Checking and Savings entities, redefine them with inheritance, and redo the mapping of the inheritance structure (including BankAccount) to the tables.

- ▶ The names of the associations are very long, such as Address CustAddress Customer, which is composed of the entity names and the foreign key relationship name.
- ▶ The names of the association methods are generated from the entity name, such as, `getAccount`, and `getCustAcct`.
- ▶ There are no tailored create methods (methods with parameters to initialize the attributes).

## Summary

The association support for entity beans in VisualAge for Java is a powerful facility to generate methods for traversing foreign key relationships into the remote interface.

Any foreign key relationship in the underlying tables can be implemented as an association between entities. Depending on the table implementation this translates into 1:1, 1:m, or m:m associations.

Because m:m associations require an intermediate table in the database, an intermediate entity bean is created as well. Note that in most practical examples, the intermediate table also carries some data columns, and therefore it does make sense to also have an entity for that table.

VisualAge for Java provides the complete deployed code that can be installed into a WebSphere EJB container.





# Client programming for inheritance and associations

In this chapter we provide a few examples of client programs that use the inheritance structure of bank accounts and follow associations between entities.

The task with inheritance is to analyze what type of subclass a bank account is, checking or savings. Depending on how you retrieve the bank accounts, for example, by primary key, through a finder, or by following associations, and if you use the entity beans or access beans, the task is easy or not so easy.

**Important:** When running the sample programs be sure to calculate the class path, and manually add the WebSphere Test Environment project.

# Entity bean types with inheritance

When we access bank accounts through finder methods, or when we follow associations from other beans to bank accounts, we want to find out what kind of an account the resulting bean is.

## Listing the type of bank accounts

A first program lists all the bank accounts with their type (checking or savings) by using the findAll custom finder (Figure 18-1).

Notes:

- ▶ The findAll method returns an Enumeration of bank accounts.
- ▶ To get the account we have to *narrow* the object to a BankAccount.

```
acct = (BankAccount)PortableRemoteObject.
 narrow(eAcct.nextElement(),BankAccount.class);
```

- ▶ From this variable we cannot find out its type, casting or using the narrow method does not reveal if the account is a checking or savings account.
- ▶ To analyze the account type we have to retrieve the account by its key:  

```
acct2 = (BankAccount)acctHome.findByPrimaryKey(acctKey);
```
- ▶ Now we can use casting or narrow to look for checking or savings account, either code works:

```
acctChk = (Checking)acct2;
acctChk = (Checking)PortableRemoteObject.narrow(acct2,Checking.class);
```

- ▶ We can also check the type using code such as:

```
if (acct2 instanceof Checking) {
 acctChk = (Checking)acct2;

}
```

- ▶ Typical output is sorted by type because of the way the underlying SQL statement is structured with UNION clauses selecting the types:

```
Checking 101-1001: 80.00 200.00
Checking 102-2002: 75.50 200.00
Checking 105-5001: 0.00 200.00
Checking 106-6001: 1000.00 300.00
Savings 101-1002: 375.26 100.00
Savings 102-2001: 9375.26 100.00
Savings 103-3001: 100.00 150.00
Savings 106-6002: 2000.00 100.00
Savings 106-6003: 3000.00 250.00
```

```

package itso.ejb35.bank.client;
import itso.ejb35.bank.*;
import java.util.*;
import javax.rmi.*;
public class ListAccountTypes {
public static void main(java.lang.String[] args) {
try {
BankAccountHome acctHome;
BankAccount acct, acct2;
BankAccountKey acctKey;
Checking acctChk;
Savings acctSav;
String acctID;
Enumeration eAcct;
//Properties properties = new Properties();
//properties.put(javax.naming.Context.PROVIDER_URL, "iiop:///");
//properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
// "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
//javax.naming.InitialContext initialContext =
//new javax.naming.InitialContext(properties);
javax.naming.InitialContext initialContext =
 new javax.naming.InitialContext();
Object objHome = initialContext.lookup("itso/ejb35/bank/BankAccount");
acctHome = (BankAccountHome)PortableRemoteObject.
 narrow(objHome,BankAccountHome.class);
eAcct = acctHome.findAll();
while (eAcct != null && eAcct.hasMoreElements()) {
 acct = (BankAccount)PortableRemoteObject.
 narrow(eAcct.nextElement(),BankAccount.class);
 acctKey = (BankAccountKey)acct.getPrimaryKey();
 acctID = acctKey.accID;
 acct2 = (BankAccount)acctHome.findByPrimaryKey(acctKey);
 if (acct2 instanceof Checking) {
 acctChk = (Checking)acct2;
 System.out.println("Checking "+acctID+": "+acct.getBalance()
 +" "+acctChk.getOverdraft());
 } else if (acct2 instanceof Savings) {
 acctSav = (Savings)acct2;
 System.out.println("Savings "+acctID+": "+acct.getBalance()+"...");
 } else {
 System.out.println("Account "+acctID+": "+acct.getBalance());
 }
}
} catch(Exception ex) { ex.printStackTrace(); }
}}

```

Figure 18-1 Listing all bank accounts with their type

## Using access beans with inheritance

Figure 18-2 shows a sample program to list the bank accounts with access beans. Access beans do not provide an easy way to check for the type of accounts. You would have to use a new access bean of each type for each instance in the rowset and try to retrieve the correct type.

However, this example does show that finder methods can be used on subclasses.

```
public class ListAccountTypesAB {
 public static void main(java.lang.String[] args) {
 try {
 BankAccountAccessBeanTable accttab = new BankAccountAccessBeanTable();
 BankAccountAccessBean acct = new BankAccountAccessBean();
 CheckingAccessBeanTable chcktab = new CheckingAccessBeanTable();
 CheckingAccessBean chck = new CheckingAccessBean();
 System.out.println("All Accounts:");
 accttab.setBankAccountAccessBean(acct.findAll());
 for (int i=0; i < accttab.numberOfRows(); i++) {
 acct = accttab.getBankAccountAccessBean(i);
 String accid = ((BankAccountKey)acct._getKey()).accID;
 System.out.println("Account "+accid+": "+ acct.getBalance());
 }
 System.out.println("Checking Accounts:");
 chcktab.setCheckingAccessBean(chck.findAll());
 for (int i=0; i < chcktab.numberOfRows(); i++) {
 chck = chcktab.getCheckingAccessBean(i);
 String accid = ((BankAccountKey)chck._getKey()).accID;
 System.out.println("Checking "+accid+": "+ chck.getBalance() +.....);
 }
 } catch(Exception ex) { ex.printStackTrace(); }
 }
}
```

Figure 18-2 Listing bank accounts with access beans

## Programming with associations

In this section we explore the associations between entities in sample programs.

Figure 18-3 shows a program that starts with a customer, follows the association to the intermediate CustAcct beans, retrieves the related bank account for each intermediate bean, and then follows the association to list the transactions of the account. The calls to association methods are highlighted in bold.

```

package itso.ejb35.bank.client;
import itso.ejb35.bank.*;
import;
public class ListCustomerAccounts {
public static void main(java.lang.String[] args) {
try {
CustomerHome custHome; Customer cust; CustAcct ca;
BankAccount acct; Checking acctChk; Savings acctSav;
TransRecord trec;
Enumeration eCustAcct, eAcct;
UserTransaction transact; InitialContext ctx = new InitialContext();
int custKey = 106;
if (args.length > 0) custKey = (new Integer(args[0])).intValue();
// ... standard code to retrieve customer home
// ... custHome = (CustomerHome)PortableRemoteObject.narrow(...);
transact = (UserTransaction)ctx.lookup("jta/usertransaction");
transact.begin();
cust = custHome.findByPrimaryKey(new CustomerKey(custKey));
System.out.println("Customer "+cust.getCustomerID()+" "+cust.getName());
eCustAcct = cust.getCustomerAccts();
while (eCustAcct != null && eCustAcct.hasMoreElements()) {
ca = (CustAcct)PortableRemoteObject.narrow
(eCustAcct.nextElement() ,CustAcct.class);
acct = ca.getAccount();
String accid = ((BankAccountKey)acct.getPrimaryKey()).accID;
if (acct instanceof Checking) {
System.out.println("- Checking "+accid+": "+ acct.getBalance());
} else if (acct instanceof Savings) {
System.out.println("- Savings "+accid+": "+ acct.getBalance());
} else {
System.out.println("- Account "+accid+": "+ acct.getBalance());
}
try { eAcct = acct.getBankTransactions(); }
catch (Exception e) {eAcct = null; }
while (eAcct != null && eAcct.hasMoreElements()) {
trec = (TransRecord)PortableRemoteObject.narrow
(eAcct.nextElement(),TransRecord.class);
java.sql.Timestamp trecKey = ((TransRecordKey)
trec.getPrimaryKey()).transID;
System.out.println(" - Transaction: "+trecKey + " " +trec
.getTranstype() + " " + trec.getTransamt());
}
}
transact.commit();
} catch(Exception ex) { ex.printStackTrace(); }
}}}
```

*Figure 18-3 Following associations: customer - accounts - transactions*

#### Notes:

- ▶ Whenever you retrieve an enumeration by following an association you have to use the *narrow* method to cast to the target class (for example, *CustAcct* from *Customer* and *TransRecord* from *BankAccount*).
- ▶ If the target of the association is a single bean, then you do not need to cast at all (for example, *BankAccount* from *CustAcct*).
- ▶ After retrieving the *BankAccount* instance, you can use instance checking or simple casting to analyze the type of account.
- ▶ Without the *UserTransaction* you may get an *IllegalStateException*.

Here is a sample output:

```
Customer 106 Mr. Ueli Wahli
- Checking 105-5001: 0.00
- Checking 106-6001: 1000.00
 - Transaction: 2001-04-18 11:02:52.584 D 66.66
- Savings 106-6002: 2000.00
 - Transaction: 2001-04-18 11:02:52.594001 C 66.66
- Savings 106-6003: 3000.00
```

## Following associations with access beans

When you analyze the code generated for associations in the access beans you notice that the result of the relationship methods is the same Enumeration that is retrieved in the entity beans.

For example:

- ▶ The *getCustAcct* method in the *Customer* bean retrieves an Enumeration of related *CustAcct* beans.
- ▶ The *getCustAcct* method in the *CustomerAccessBean* class forwards the call to the remote instance and, therefore, gets the same result.

When using access beans, we would want the result of an association method to be a rowset, that is a set of access beans, of related beans.

Luckily access beans provide a method to convert an Enumeration of beans into a special *AccessBeanEnumeration*, which can be assigned to a rowset:

```
catab.setCustAcctAccessBean(new AccessBeanEnumeration
 (cust.getCustAcct(), ca.getClass()));
```

Where, *cust.getCustAcct* retrieves the related instances, which are then converted into a rowset (*catab* is a *CustAcctAccessBeanTable*, and *ca* is a *CustAcctAccessBean*).

Figure 18-4 shows an extract of the sample program that:

- ▶ Starts with a CustomerAccessBean
- ▶ Retrieves the intermediate CustAcctAccessBean objects into a rowset
- ▶ Retrieves the BankAccountAccessBean related to the CustAcctAccessBean
- ▶ Retrieves the TransRecordAccessBeans related to the bank account

The calls to association methods are highlighted in bold.

```
public class ListCustomerAccountsAB {
 public static void main(java.lang.String[] args) {
 try {
 CustomerAccessBean cust;
 CustAcctAccessBean ca = new CustAcctAccessBean();
 CustAcctAccessBeanTable catab = new CustAcctAccessBeanTable();
 BankAccountAccessBean acct;
 TransRecordAccessBean trec = new TransRecordAccessBean();
 TransRecordAccessBeanTable trectab;
 InitialContext ctx = new InitialContext();
 UserTransaction transact =
 (UserTransaction)ctx.lookup("jta/usertransaction");
 transact.begin();
 cust = new CustomerAccessBean();
 cust.setInitKey_customerID(106);
 System.out.println("Customer "+cust.getCustomerID()+" "+cust.getName());
 catab.setCustAcctAccessBean(new AccessBeanEnumeration
 (cust.getCustAcct(), ca.getClass()));
 for (int i=0; i < catab.numberofRows(); i++) {
 ca = catab.getCustAcctAccessBean(i);
 acct = ca.getAccount();
 String accid = ((BankAccountKey)acct._getKey()).accID;
 System.out.println("- Account "+accid+": "+ acct.getBalance());
 trectab = new TransRecordAccessBeanTable();
 trectab.setTransRecordAccessBean(new AccessBeanEnumeration
 (acct.getBankTransactions(), trec.getClass()));
 for (int j=0; j < trectab.numberofRows(); j++) {
 trec = trectab.getTransRecordAccessBean(j);
 java.sql.Timestamp trecKey = ((TransRecordKey)trec._getKey()).
 transID;
 System.out.println(" - Transaction: "+trecKey + " " +);
 }
 }
 transact.commit();
 } catch(Exception ex) { ex.printStackTrace(); }
 }
}
```

Figure 18-4 Using access beans to follow associations

## Using a custom finder method to traverse an association

In “Finder method for m:m association” on page 365, we implemented a custom finder method (`findAccountsForCustomer`) to retrieve all BankAccount beans for a Customer.

Figure 18-5 shows how this method can be used to bypass the intermediate `CustAcct` beans. After retrieving the customer, we directly use the customer finder methods (highlighted in bold) to retrieve the related bank accounts.

Notice that the `findAccountsForCustomer` method was propagated to the `BankAccountAccessBean`, and retrieves an Enumeration of access beans that can be directly assigned to the `BankAccountAccessBeanTable`.

```
public class ListCustomerAccountsFinderAB {
 public static void main(java.lang.String[] args) {
 try {
 CustomerAccessBean cust;
 BankAccountAccessBean acct = new BankAccountAccessBean();
 BankAccountAccessBeanTable accttab = new BankAccountAccessBeanTable();
 TransRecordAccessBean trec = new TransRecordAccessBean();
 TransRecordAccessBeanTable trectab;
 InitialContext ctx = new InitialContext();
 UserTransaction transact =
 (UserTransaction)ctx.lookup("jta/usertransaction");
 transact.begin();
 cust = new CustomerAccessBean();
 cust.setInitKey_customerID(106);
 System.out.println("Customer "+cust.getCustomerID()+" "+cust.getName());
 accttab.setBankAccountAccessBean(acct.findAccountsForCustomer(106));
 for (int i=0; i < accttab.numberOfRows(); i++) {
 acct = accttab.getBankAccountAccessBean(i);
 String accid = ((BankAccountKey)acct._getKey()).accID;
 System.out.println("- Account "+accid+": "+acct.getBalance());
 trectab = new TransRecordAccessBeanTable();
 trectab.setTransRecordAccessBean(new AccessBeanEnumeration
 (acct.getBankTransactions(), trec.getClass()));
 for (int j=0; j < trectab.numberOfRows(); j++) {
 trec = trectab.getTransRecordAccessBean(j);
 java.sql.Timestamp treckey = ((...))trec._getKey().transID;
 System.out.println(" - Transaction: "+treckey + " " +.....);
 }
 }
 transact.commit();
 } catch(Exception ex) { ex.printStackTrace(); }
 }
}
```

Figure 18-5 Using a custom finder method for m:m associations

## Using associations in a servlet and JSP application

This sample consists of two servlets and JSPs:

- ▶ The first servlet (CustAccounts) retrieves a customer bean and its bank account beans and displays the information using the first JSP.
  - ▶ The second servlet (AccountMgr) performs simple banking transactions
    - Deposit an amount into one account
    - Withdraw an amount from one account
    - Transfer an amount from one account to another
- and displays the resulting balance(s) using the second JSP.

Figure 18-6 shows a sample run of the application.

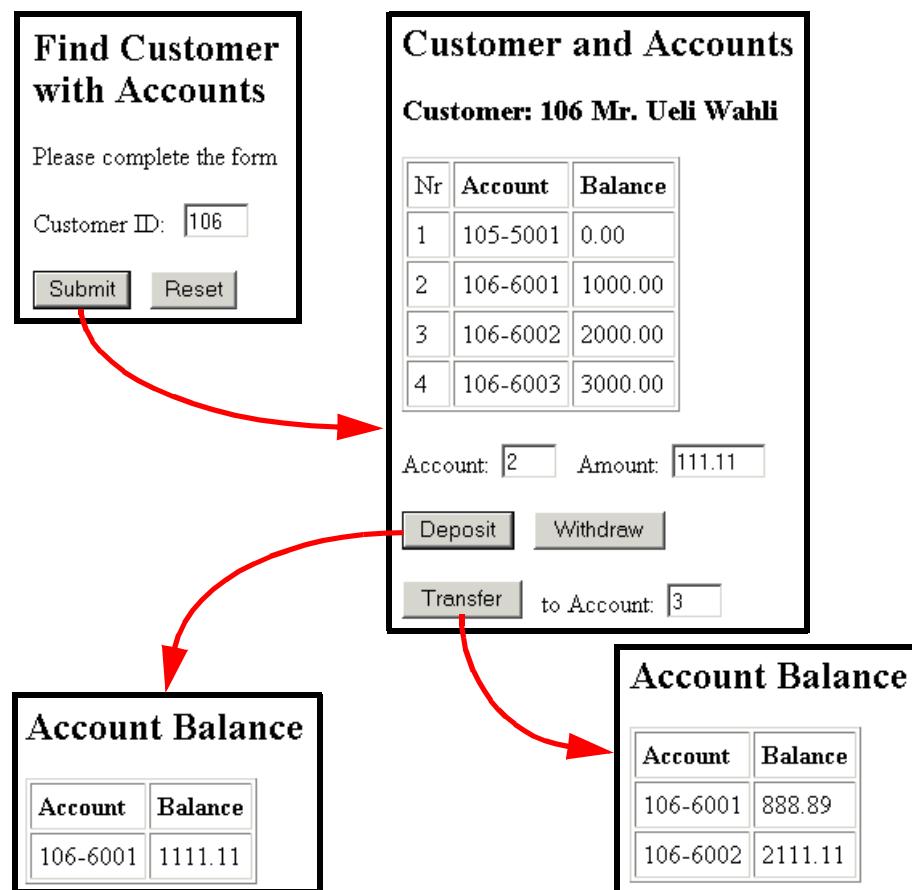


Figure 18-6 Simple banking application using associations

## Servlet to retrieve customer and bank accounts

Figure 18-7 shows an extract of the `itso.ejb35.bank.servlet.CustAccounts` servlet that retrieves the customer for a given customer ID, and then the associated bank accounts, using access beans and the custom finder method.

```
package itso.ejb35.bank.servlet;
public class CustAccounts extends HttpServlet {
 public void performTask(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException {
 // Set the results page URL
 String url = "/ejb/bankservlet/CustAccounts.jsp";
 String urlerr = "/ejb/bankservlet/CustAccountsError.html";

 try {
 // Read the input parameter from the HTML Form
 String id = request.getParameter("CustomerID");
 int custId = (new Integer(id)).intValue();

 // variables
 CustomerAccessBean customer = new CustomerAccessBean();
 BankAccountAccessBean acct = new BankAccountAccessBean();
 BankAccountAccessBeanTable accounts = new BankAccountAccessBeanTable();

 // get the customer and bank accounts
 customer.setInitKey_customerID(custId);
 String custName = customer.getName();
 accounts.setBankAccountAccessBean(acct.findAccountsForCustomer(custId));

 // Forward to the result JSP
 request.setAttribute("customer", customer);
 request.setAttribute("accounts", accounts);
 getServletConfig().getServletContext().getRequestDispatcher(url).
 forward(request, response);
 } catch (Exception e) {
 getServletConfig().getServletContext().getRequestDispatcher(urlerr).
 forward(request, response);
 }
 }
}
```

Figure 18-7 Servlet to retrieve customer and bank accounts

The code is similar to “Using a custom finder method to traverse an association” on page 388. The customer access bean and the bank access bean rowset are stored in the request block for the JSP.

## JSP to display customer and bank accounts

The CustAccounts JSP retrieves the access beans from the request block and displays the customer name and the list of bank accounts (Figure 18-8).

```
<HTML>
 <HEAD><TITLE>Banking Application</TITLE></HEAD>
 <BODY>
 <H2>Customer and Accounts</H2>
 <jsp:useBean id="customer"
 type="itso.ejb35.bank.CustomerAccessBean" scope="request"/>
 <jsp:useBean id="accounts"
 type="itso.ejb35.bank.BankAccountAccessBeanTable" scope="request"/>
 <%! itso.ejb35.bank.BankAccountAccessBean acct;
 String accid; %>
 <H3>Customer: <%= customer.getCustomerID() %> <%= customer.getName() %></H3>
 <FORM METHOD="post" ACTION="itso.ejb35.bank.servlet.AccountMgr">
 <TABLE border="1" cellpadding="4">
 <TR> <td>Nr</td><td>Account</td><td>Balance</td> </TR>
 <% for (int i=0; i < accounts.numberOfRows(); i++) {
 acct = accounts.getBankAccountAccessBean(i);
 accid = ((itso.ejb35.bank.BankAccountKey)acct._getKey()).accID; %>
 <TR>
 <td> <%= i+1 %>
 <INPUT TYPE="hidden" NAME="Acct<%= i+1 %>" ID="Acct<%= i+1 %>"
 SIZE="8" MAXLENGTH="8" VALUE="<%= accid %>"> </td>
 <td> <%= acct.getBalance() %> </td>
 </TR>
 <% } %>
 </TABLE>
 <P> Account:
 <INPUT TYPE="text" NAME="Account" ID="Account" SIZE="2" MAXLENGTH="2">
 Amount:
 <INPUT TYPE="text" NAME="Amount" ID="Amount" SIZE="6" MAXLENGTH="6">
 <P>
 <INPUT TYPE="submit" NAME="Deposit" ID="Submit" VALUE="Deposit">
 <INPUT TYPE="submit" NAME="Withdraw" ID="Submit" VALUE="Withdraw">
 <P>
 <INPUT TYPE="submit" NAME="Transfer" ID="Submit" VALUE="Transfer">
 to Account:
 <INPUT TYPE="text" NAME="Account2" ID="Account2" SIZE="2" MAXLENGTH="2">
 </FORM>
 </BODY></HTML>
```

Figure 18-8 JSP to display customer and bank accounts: CustAccounts.jsp

JSP processing:

- ▶ The two beans are retrieved from the request block.
- ▶ The customer information is formatted.
- ▶ A form is started to be input for the second servlet.
- ▶ The form consists of a table with the account number and balances of the bank accounts.
- ▶ The first column of the table is a sequential number of the accounts that is used to identify the account for the deposit, withdraw, and transfer requests.
- ▶ A hidden field in the first column carries the bank account number.
- ▶ At the bottom of the form are three push buttons for deposit, withdraw, and transfer requests, as well as input fields for the account selection and the amount.

## Servlet with simple bank transactions

Figure 18-9 and Figure 18-10 show an extract of the AccountMgr servlet that retrieves the bank accounts using access beans and then performs the requested deposit, withdraw, or transfer operation.

```
public class AccountMgr extends HttpServlet {
 public void performTask(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException {
 // Set the results page URL
 String url = "/ejb/bankservlet/AccountMgr.jsp";
 String urlerr = "/ejb/bankservlet/AccountMgrError.html";
 try {
 // Read the input parameter from the HTML Form
 String acctnr1 = request.getParameter("Account");
 String acctnr2 = request.getParameter("Account2");
 String amountx = request.getParameter("Amount");
 String deposit = request.getParameter("Deposit");
 String withdraw = request.getParameter("Withdraw");
 String transfer = request.getParameter("Transfer");
 String acctid1 = request.getParameter("Acct"+acctnr1);
 String acctid2 = request.getParameter("Acct"+acctnr2);
 // variables
 java.math.BigDecimal amount = new java.math.BigDecimal(amountx);
 BankAccountAccessBean acct1 = new BankAccountAccessBean();
 BankAccountAccessBean acct2 = null;
 UserTransaction transact;
 javax.naming.InitialContext ctx = new javax.naming.InitialContext();
```

Figure 18-9 Servlet with simple bank transactions (part 1)

```

//.....continuation.....
// check input
if (acctid1 == null || (transfer != null && acctid2 == null)) {
 getServletConfig().getServletContext().getRequestDispatcher(urlerr).
 forward(request, response);
 return;
}
// get the accounts
acct1.setInitKey_acctID(acctid1);
if (transfer != null) {
 acct2 = new BankAccountAccessBean();
 acct2.setInitKey_acctID(acctid2);
}
// processing
transact = (UserTransaction)ctx.lookup("jta/usertransaction");
transact.begin();
if (deposit != null) acct1.deposit(amount);
if (withdraw != null) acct1.withdraw(amount);
if (transfer != null) {
 acct1.withdraw(amount);
 acct2.deposit(amount);
 acct2.commitCopyHelper();
 acct2.refreshCopyHelper();
}
acct1.commitCopyHelper();
acct1.refreshCopyHelper();
transact.commit();
// Forward to the results JSP
request.setAttribute("account1", acct1);
if (acct2 != null)
 request.setAttribute("account2", acct2);
else
 request.setAttribute("account2", acct1);
getServletConfig().getServletContext().getRequestDispatcher(url1).
 forward(request, response);
} catch (Exception ex) {
 getServletConfig().getServletContext().getRequestDispatcher(urlerr).
 forward(request, response);
}
}
}

```

*Figure 18-10 Servlet with simple bank transactions (part 2)*

Notice the use of a user transaction to make sure that deposit and withdraw complete for a transfer operation.

The account access beans are stored in the request block for the result JSP.

## JSP to display the result balances

The AccountMgr JSP retrieves the account access beans and displays the account number and balance information. For deposit and withdraw operations only one account is displayed (the servlet stores the same account twice in the request block).

```
<HTML>
 <HEAD><TITLE>Banking Application</TITLE></HEAD>
<BODY>
 <jsp:useBean id="account1" type="itso.ejb35.bank.BankAccountAccessBean"
 scope="request"/>
 <jsp:useBean id="account2" type="itso.ejb35.bank.BankAccountAccessBean"
 scope="request"/>
<H2>Account Balance </H2>
 <TABLE border="1" cellpadding="4">
 <TR> <td>Account</td><td>Balance</td> </TR>
 <TR>
 <td> <%= ((itso.ejb35.bank.BankAccountKey)account1.__getKey()).accID %>
 </td>
 <td> <%= account1.getBalance() %> </td>
 </TR>
 <% if (account2 != account1) { %>
 <TR>
 <td> <%= ((itso.ejb35.bank.BankAccountKey)account2.__getKey()).accID %>
 </td>
 <td> <%= account2.getBalance() %> </td>
 </TR>
 <% } %>
 </TABLE>
</BODY>
</HTML>
```

Figure 18-11 JSP to display result balances: AccountMgr.jsp

The JSP accesses the two bank account access beans and gets the balance information from the values stored in the copy helper beans.

The only logic required is to test if the operation was a transfer, and in that case display two rows in the table. This test could also have been performed by testing which button was used to invoke the servlet:

```
<% if (request.getParameter("Transfer") != null) { %>
```

The JSP does have access to all the parameters of the form that invoked the servlet; the values are still in the request block.

## Servlet with session bean

We can rewrite the AccountMgr servlet to use the Banking session bean, through its BankingAccessBean, instead of accessing the bank accounts.

The checking of the parameters remains the same, but the main logic becomes quite simpler and no user transaction is required because the session bean will handle the transaction management (Figure 18-12).

```
public class AccountMgrSess extends HttpServlet {
 public void performTask(.....) throws ServletException, IOException {
 // Set the results page URL
 String url = "/ejb/bankservlet/AccountMgrSess.jsp";
 String urlerr = "/ejb/bankservlet/AccountMgrError.html";
 try {
 // Read the input parameter from the HTML Form
 same.....
 // variables
 java.math.BigDecimal amount = new java.math.BigDecimal(amountx);

 // check input
 same.....

 // processing
 BankingAccessBean banking = new BankingAccessBean();
 if (deposit != null) banking.deposit(acctid1,amount);
 if (withdraw != null) banking.withdraw(acctid1,amount);
 if (transfer != null) banking.transferMoney(acctid1,acctid2,amount);

 // Forward to the results JSP
 request.setAttribute("banking", banking);
 getServletConfig().getServletContext().getRequestDispatcher(url).
 forward(request, response);
 } catch (Exception ex) {same..... }
 }
}
```

Figure 18-12 Servlet using a session access bean

We have to use a different JSP for the output, because we do not have any bank account access beans in the servlet.

The easiest solution is to pass the session access bean to the JSP, from where it can retrieve the balance information. The account IDs are still available in the request block (parameters) and can be obtained by the JSP code as well.

Figure 18-13 shows the JSP code that works with the session access bean.

```

<HTML>
 <HEAD><TITLE>Banking Application</TITLE></HEAD>
<BODY>
 <jsp:useBean id="banking" type="itso.ejb35.bank.BankingAccessBean"
 scope="request"/>
 <%! String acctnr1, acctnr2, acctid1, acctid2; %>
 <% String acctnr1 = request.getParameter("Account");
 String acctnr2 = request.getParameter("Account2");
 String acctid1 = request.getParameter("Acct"+acctnr1);
 String acctid2 = request.getParameter("Acct"+acctnr2); %>
 <H2>Account Balance </H2>
 <TABLE border="1" cellpadding="4">
 <TR> <td>Account</td><td>Balance</td> </TR>
 <TR>
 <td> <%= acctid1 %> </td>
 <td> <%= banking.getBalance(acctid1) %> </td>
 </TR>
 <% if (request.getParameter("Transfer") != null) { %>
 <TR>
 <td> <%= acctid2 %> </td>
 <td> <%= banking.getBalance(acctid2) %> </td>
 </TR>
 <% } %>
 </TABLE>
</BODY>
</HTML>

```

*Figure 18-13 JSP using a session access bean: AccountMgrSess.jsp*

To implement the change you can copy the *CustAccounts* servlet as *CustAccountsSess*, which calls a *CustAccountsSess.jsp* file.

The *CustAccountsSess.jsp* file is a copy of the *CustAccounts.jsp* file, but invokes the new *AccountMgrSess* servlet, which calls the *AccountMgrSess.jsp* file (Figure 18-13).

Finally, to invoke the whole new chain, copy the *CustAccounts.html* file as *CustAccountsSess.html* and invoke the *CustAccountsSess* servlet.

# Running the advanced examples

You can run these sample programs and servlets in the VisualAge for Java WebSphere Test Environment or against a real WebSphere Application Server.

## WebSphere Test Environment

Start all the servers in the test environment:

- ▶ Persistent name server
- ▶ EJB server (use a configuration that includes the Bank\_Entities EJB group)
- ▶ Servlet engine (check the class path, it must include the IBM EJB Tools)

Place all the HTML and JSP files into the correct directory:

```
<IBMVJava>\ide\project_resources\IBM WebSphere Test Environment\
hosts\default_host\default_app\web\ejb\bankservlet
```

For the sample applications, calculate the class path, and then manually add the IBM WebSphere Test Environment. Then run the sample application in VisualAge for Java.

For the servlets, point a browser at:

```
http://localhost:8080/ejb/bankservlet/CustAccounts.html
http://localhost:8080/ejb/bankservlet/CustAccountsSess.html
```

## WebSphere Application Server

Follow the same process as outlined in “Deploying and running the client applications” on page 288.

### Client applications

To test the client applications against WebSphere:

- ▶ Create a client JAR file from the Bank\_Entities EJB group (`itso_client.jar`) and place it into a suitable directory:

```
e:\sg246144\sampcode\ejbclient\itso_client.jar
```

Be sure to click on *Select referenced types and resources*.

- ▶ Set the class path to include the client JAR file.
- ▶ Export the class files of your applications (`itso.ejb35.bank.client package`) from VisualAge for Java into the test directory:  

```
e:\SG246144\sampcode\ejbclient\itso\ejb35\bank\client\...classname...
```
- ▶ To test the stand-alone applications, open a command window and switch to the test directory. Then set the class path and run the applications:

```
cd e:\SG246144\sampcode\ejbclient
setCP.bat
java itso.ejb35.bank.client.ListAccountTypes
java itso.ejb35.bank.client.ListAccountTypesAB
java itso.ejb35.bank.client.ListCustomerAccounts
java itso.ejb35.bank.client.ListCustomerAccountsAB
java itso.ejb35.bank.client.ListCustomerAccountFinderAB
```

## Servlets

To test the servlets in WebSphere:

- ▶ Install the EJBs in an EJB container as described in “Deployment to WebSphere” on page 372.
- ▶ Export the servlet class files (itso.ejb35.bank/servlet package) into the WebSphere Web application directory:

```
d:\WebSphere\AppServer\hosts\default_host\itsoejb\servlets
==> ...itsoejb\servlets\itso\ejb35\bank\servlet\...classname...
```

- ▶ Copy the HTML and JSP files into the WebSphere Web application directory:  
  
d:\WebSphere\AppServer\hosts\default\_host\itsoejb\web  
==> ...itsoejb\web\ejb\bankservlet\...filename...
- ▶ Modify the HTML files to call the servlet with the Web application prefix:  
  
<FORM ... ACTION="/**itsoejb**/servlet/itso.ejb35.bank.servlet.CustAccounts">
- ▶ Add the client JAR file to the class path:  
  
d:\WebSphere\AppServer\hosts\default\_host\itsoejb\servlets\itso\_client.jar
- ▶ Restart the application server (or at least the Web application).
- ▶ Run the servlets by pointing a browser to:

```
http://localhost/itsoejb/ejb/bankservlet/CustAccounts.html
http://localhost/itsoejb/ejb/bankservlet/CustAccountsSess.html
```

## Summary

Client programming with inheritance and associations provides some challenges over simple access to entities. The usage of access beans becomes a little more complicated and does not handle inheritance very well.

The examples provided in this chapter illustrate the principles and can be used as code guidelines for your own applications.



# Exception handling

In this chapter, we describe the different types of exceptions and how they are handled by the client, beans and the container. We also present guidelines on handling exceptions and the benefits of exception handling.

Exception handling makes EJB more reliable and highly communicative with the clients. EJBs come with some built-in exceptions and it allows us to define our own exceptions.

# Introduction

Exceptions are the customary way in Java to indicate to a calling method that an abnormal condition has occurred. When a method encounters an abnormal condition (an exception condition) that it cannot handle itself, it may throw an exception. Exceptions are thrown at many different levels in large applications. Each exception has a specific meaning and is handled differently by the application.

## Checked and unchecked exceptions

There are two kinds of exceptions in Java, checked and unchecked (Figure 19-1). Only checked exceptions have to appear in throws clauses. Checked exceptions that may be thrown in a method must either be caught or declared in the throws clause.

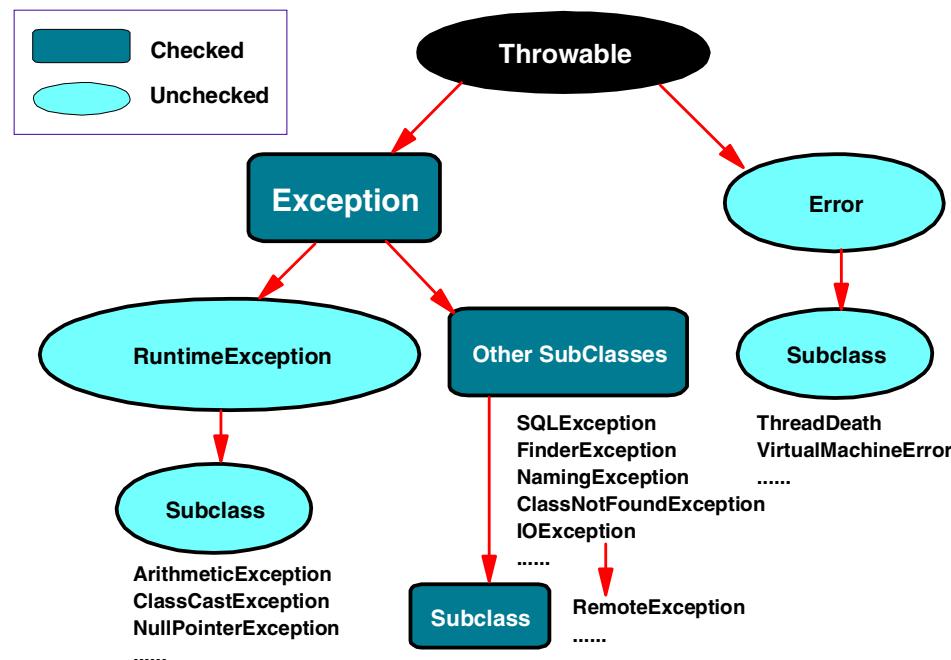


Figure 19-1 Checked and unchecked exceptions

The difference between checked and unchecked exceptions is that checked exceptions signal abnormal conditions that the client programmers can deal with. Most unchecked exceptions are problems that will be detected by the Java virtual machine at runtime; they are subclasses of **Error** and **RuntimeException**.

# Exception guidelines

In this section, we discuss some general guidelines on exceptions.

- ▶ If your method encounters an abnormal condition that it cannot handle, it should throw an exception.
- ▶ Avoid using exceptions to indicate conditions that can reasonably be expected as part of the normal functioning of the method.
- ▶ If your method discovers that the client has breached its contractual obligations (for example, by passing in bad input data), throw a runtime exception.
- ▶ If your method is unable to fulfill its contract, throw either a checked or unchecked exception.
- ▶ If you are throwing an exception for an abnormal condition that you feel client programmers should consciously decide how to handle, throw a checked exception.
- ▶ Define or choose an already existing exception class for each kind of abnormal condition that may cause your method to throw an exception.

`Throwable` serves as the base class for an entire family of classes, declared in `java.lang`, that your program can instantiate and throw.

## Benefits

Exceptions allow us to separate error handling code from normal code. We can wrap the code that we expect to execute most of the time with a `try` block, and then place error handling code in `catch` clauses. That is the code that we do not expect to get executed often, if ever.

If we feel that a method does not know how to handle a particular error, we can throw an exception from the method and let the caller of the method deal with it. If we throw a checked exception, we utilize the help of the Java compiler to force client programmers to deal with the potential exception, either by catching it or declaring it in the `throws` clause of their methods. The fact that Java compilers make sure checked exceptions are handled helps making Java programs more robust.

# Exceptions and EJB

The EJB 1.1 specification groups exceptions into two categories to enable bean developers, client developers, and container providers to handle exceptions and transactions more effectively. They are system and application exceptions. Each type of exception is handled differently by the container. These exceptions are propagated to the client across remote invocation, so that the client will be able to catch it.

## Application exceptions

Application exceptions are checked exceptions. They do not subclass `RuntimeException` or `RemoteException`. They represent errors in business logic or abnormal application-level conditions specific to a use case or business service provided by your EJBs, for example, `InSufficientFundException` in a bank application.

Clients typically can recover from application exceptions. Therefore, application exceptions are not intended for reporting system-level problems.

The EJB specification even provides some basic application exceptions for use by bean providers, known as standard application. These exceptions include:

- ▶ `CreateException`
- ▶ `DuplicateKeyException`
- ▶ `FinderException`
- ▶ `ObjectNotFoundException`
- ▶ `RemoveException`

Furthermore, these exceptions do not necessarily cause a transaction to rollback, giving the client the opportunity to retry the operation.

These exceptions may be thrown by the beans themselves. In the case of container-managed persistence (CMP), the container can also throw any of these exceptions while handling the `ejbCreate`, `ejbFind`, and `ejbRemove` methods.

A bean developer can always choose to throw a standard application exception from the appropriate method regardless of how persistence is managed. They are always delivered directly to the client, without being repackaged as `RemoteException`.

Business methods defined in the remote interface can throw any kind of application exception. Let's explain the standard application exceptions in detail.

## CreateException

The CreateException is thrown by the create method in the home interface.

This exception indicates that an application error has occurred (for example, invalid arguments) while the bean was being created. If the container throws this exception, it may or may not rollback the transaction.

Bean developers should rollback the transaction before throwing this exception only if data integrity is a concern.

## DuplicateKeyException

The DuplicateKeyException is a subclass of the CreateException. It can be thrown by the create method in the home interface.

This exception indicates that the primary key already exists in the database.

## FinderException

The FinderException is thrown by the finder methods in the home interface. This exception indicates that an application error occurred, example invalid arguments, while the container attempted to find the beans.

**Important:** Multiple-entity finder methods return an empty collection if no entities are found; single-entity finder methods throw an ObjectNotFoundException to indicate that no object was found.

## ObjectNotFoundException

The ObjectNotFoundException is thrown from a single-entity finder method to indicate that the container could not find the requested entity. This exception should not be thrown to indicate a business logic error (for example, invalid arguments).

The transaction is typically not rolled back by the bean provider or container before throwing this exception.

## RemoveException

The RemoveException can be thrown by the remove methods in the remote and home interfaces.

This exception indicates that an application error has occurred while the bean was being removed. The transaction may or may not have been rolled back by the container before throwing this exception.

## Client view

The client can continue working with the enterprise bean even after receiving an application exception. Similarly, the client can continue the transaction, with which it is associated. The container does not automatically rollback the transaction for an application exception. But the transaction might have been marked for rollback by the bean itself before an application exception is thrown. It can be checked in either of the following ways:

- ▶ We can check in the documentation of the bean. The bean provider may have specified the application exceptions for which transaction rollbacks are marked.
- ▶ If the client is an EJB with container-managed transaction, it can use the `getRollbackOnly` method of `EJBContext` to learn whether the current transaction is marked for roll back or not.
- ▶ Other clients, that are not using container-managed transaction, can use `getStatus` method on `UserTransaction` obtain the transaction status.

## System exception

System exceptions include `RuntimeException`, `RemoteException`, and all exceptions that inherit from them. Therefore, it includes both checked and unchecked exception. System exceptions cause the container to automatically rollback the transaction if existing, while any application exception is passed to the client for handling.

`EJBException` is a subclass of the `RuntimeException` and is considered a system exception. In Java, `RuntimeException` types do not need to be declared in the `throws` clause of the method signature or handled using `try/catch` blocks; they are automatically thrown from the method. They usually reflect a violation of contract, that is a bug on the software.

If the method in which the exception occurred has started a transaction, the transaction is rolled back. If the transaction started from a client that invoked the method, the client transaction is marked for rollback and cannot be committed.

When a system exception occurs, the bean instance is discarded, which means that it is disassociated and garbage collected or returned to the pool.

The impact of discarding a bean instance depends on the bean type. In the case of stateless session beans and entity beans, the client does not notice that the instance was discarded. With stateful session beans, however, the impact on the client is severe. Discarding a stateful bean instance destroys the instance conversational state and invalidates the client reference to the bean.

When a system exception occurs and the instance is discarded, a RemoteException is always thrown to the client. If the client started a transaction, which was then propagated to the bean, a system exception (thrown by the bean method) would be caught by the container and rethrown as a TransactionRolledbackException.

In all other cases, whether the bean is container-managed or bean-managed persistent, a system exception thrown from within the bean method will be caught by the container and rethrown as a RemoteException.

### NoSuchEntityException

The NoSuchEntityException is a subclass of EJBException. It should be thrown by the entity bean methods to inform the clients that the required entity bean instance has been removed from the database. This method is typically thrown from the ejbLoad and ejbStore methods.

## Application exceptions versus system exceptions

In EJB 1.1, an application exception is any exception that does not extend RuntimeException or the RemoteException.

Transactions are automatically rolled back if a system exception is thrown from a bean method. Transactions are not automatically rolled back if an application exception is thrown.

### Checked subsystem exceptions

Checked subsystem exceptions are exceptions thrown from various components for a J2EE server, including JNDI, JDBC, and JMS.

Typically, the approach is to look for checked exceptions from J2EE subsystems, such as JNDI, JDBC, JMS, and RMI, and nest them in an EJBException. Then throw the EJBException to indicate to the container that it should mark the current transaction for rollback. Naming and SQL exceptions are examples of the checked subsystem exceptions.

### Remote exception

RMI is used to distribute both the remote and home interfaces. As a result, all methods on these interfaces must throw RemoteException. Remote exceptions indicate a system-level failure that originates in the communication layer between the client and the server, either in the server, or in a bean.

A bean can throw this exception to indicate a database error. In addition to remote exceptions, a method throws clause can include any number of application-specific exceptions.

## Transaction exceptions

The server rolls back a transaction and throws an exception to the client only if one of the following conditions is true:

- ▶ The exception occurred in a container-managed transaction context.
- ▶ The exception occurred in a container-managed transaction nested in a client-managed transaction (in this case the outer transaction is marked for rollback).
- ▶ The exception occurred in a bean-managed transaction or a stateless session bean.

To mark a transaction for rollback:

- ▶ Use the `setRollbackOnly` method on the `UserTransaction` for bean-managed transactions.
- ▶ Use the `setRollbackOnly` method on the `EJBContext` for container-managed transactions.

This ensures that the transaction is never committed after your code throws an exception.

### TransactionRolledBackException

Clients that use explicit transactions must be prepared to handle both the `RemoteException` and `TransactionRolledBackException` exception. If a client program receives the `TransactionRolledBackException`, the client should rollback the transaction, because it will not be able to commit.

### TransactionRequiredException

This occurs when a client without first starting a transaction, invokes a bean method that is marked as `TX_MANDATORY`. (See “Transaction attributes” on page 221.)

### Tracing rollback exceptions

`TransactionRolledBackException` masks a variety of failures in the server side code. To trace the actual exception or failure, you have to follow these steps in VisualAge for Java:

1. In the Enterprise Beans pane, select the bean which you want to trace.
2. Invoke the context menu and select *Open to -> Server Configuration*.
3. In the Server Configuration window, select the server and *Properties*.
4. In the Properties dialog, select the option *high* for Trace Level (Figure 19-2).

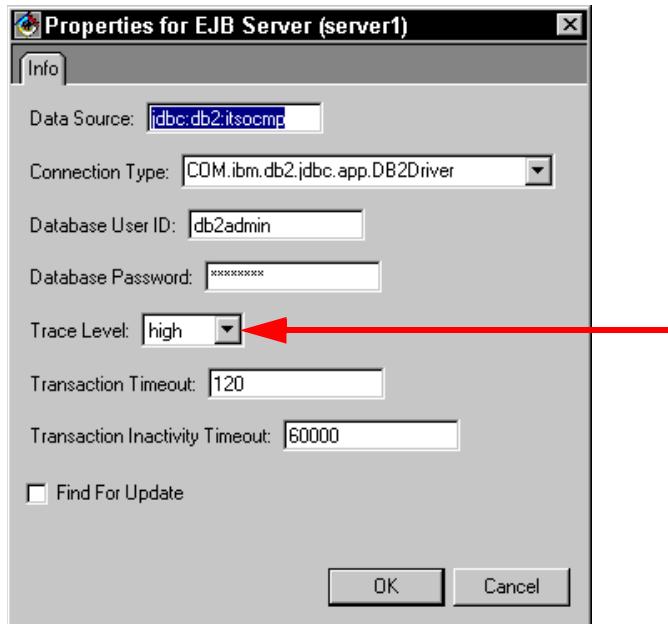


Figure 19-2 Trace option of EJB server

5. Click *OK* and start the server.
6. Run the Test client and recreate the problem.
7. Look at the trace in the console.
8. Search for the text *ROLLED* and you can see the exception you are getting at the client.
9. Page back from this location, until you come across the real exception.

Common reasons with EJB applications for getting exceptions are typical Java bugs, such as null pointers or SQL exceptions, indicating missing mandatory columns or locking problems.

**Important:** A `TransactionRollbackException` can also be generated by a database server when a database constraint is broken, for example, a trigger on balance (less than a given amount). These exceptions are among the most difficult exceptions to detect.

## Enhancing standard EJB exceptions

Standard exceptions can be extended, if necessary, to show the client that a specific error occurred. For instance, we can subclass `CreateException` to indicate more specifically that the parameters passed into `ejbCreate` are incorrect.

Subclassing standard EJB exceptions can be effective for other reasons, such as providing a better description for root errors.

## Limitations

The exception model uses the CORBA/RMI style of forcing our code to catch and handle exceptions. We can easily end up with big try/catch blocks for every remote method invocation.

It is very hard to make the code clear, simple and easy to understand when the few lines that actually do something are wrapped with many more lines of nested exception-handling.

A small but annoying feature of many Java exception hierarchies is the lack of subsystem-specific clustering. For example, consider the EJB and transaction exception hierarchies shown in Figure 19-3:

- ▶ If we consider the transaction based exceptions, they just extend from `RemoteException`. Therefore, when we try to catch only transaction exceptions, either we have to write catch block for each transaction exception or we have to catch `RemoteException` as a whole.
- ▶ A better approach is to subclass all transaction based exceptions from a common base class, say, for example, `TransactionException`. This gives more clarity.

The exceptions often inherit directly from `Exception` or `RemoteException` without having a common root for the subsystem's exceptions.

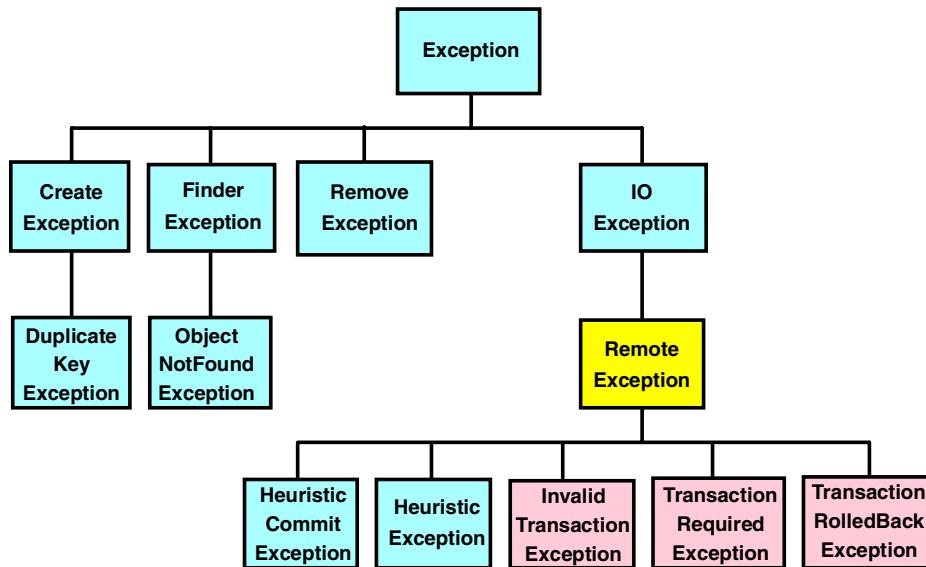


Figure 19-3 EJB and transaction exceptions without grouping

## EJB 2.0

Message-driven beans may not throw application exceptions while processing messages. This means that the only exceptions that may be thrown by a message-driven bean are runtime exceptions indicating a system-level error.

## Summary

Exceptions allow the separation of error handling code from the normal business logic code. There are two types of exception in EJB, application exceptions and system exceptions.

Application exceptions are checked exceptions. As application exceptions do not rollback the transaction, the client can continue to use the service of the bean, even after receiving an application exception.

When a system exception occurs, the transaction is rolled back, the instance is discarded and the runtime exception is thrown to the client.





## Security for enterprise beans

Distributed applications accessed through the Internet face the possibility of malicious attacks. This emphasizes security arrangements and the setting up of a security system. A security system gives us the power to determine who can and will access resources such as applications, servlets, EJBs, and Web pages. It also enables us to define the security policies to establish control of resources.

In this chapter, we discuss securing EJBs and the procedures involved in it. We will not go into actual implementation, but rather refer to other sources of documentation.

# Websphere security

In this section, we discuss the security features and architecture of the security sub-system found in WebSphere (Figure 20-1).

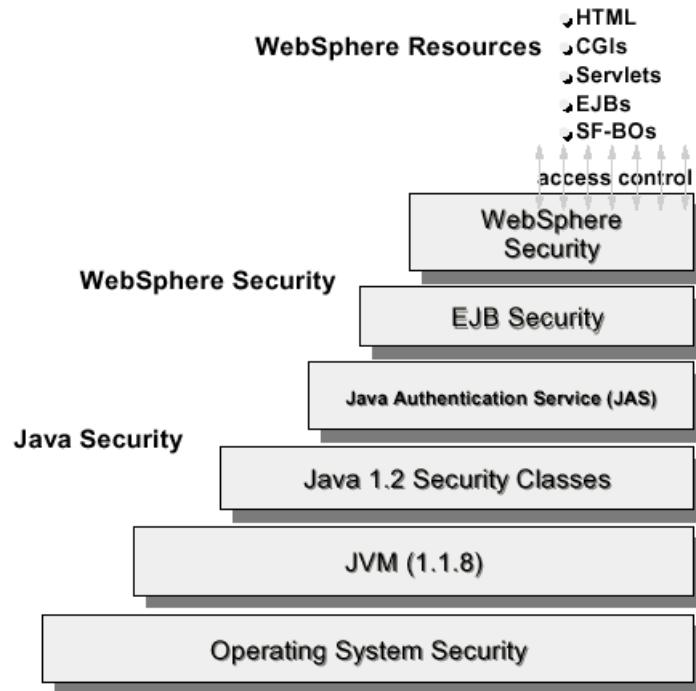


Figure 20-1 *WebSphere security*

## Security concepts

The WebSphere security system supports authentication, authorization and delegation. It is based on the Java 1.2 security model.

- ▶ Authentication is the process of verifying that users are who they say they are.
- ▶ Authorization is the process of determining whether a user is permitted to access a resource and what actions the user can take on the resource.
- ▶ Delegation is the process by which access to a resource is performed by any identity based on a defined delegation policy.

Now let us discuss these concepts in the following sections.

## **Principal**

It is a representation of an client or system identity. Therefore, authentication is basically done on a principal when it requests a protected resource from the server. A directory service or user registry provides the mechanism for validating the data presented by the principal.

## **WebSphere clients**

There are two types of clients in WebSphere. They are Web clients and fat clients. Web clients are authenticated using a browser. The client accesses a URL that has an access control list (ACL) associated with it. The user is prompted for user ID and password, which are used to authenticate the user. The fat client must be programmatically authenticated.

## **Access control list (ACL)**

Access control lists are used to authenticate users and manage access to network services. Each entry in an ACL contains a set of permission associated with a particular principal.

## **Realm**

A realm is a domain for a set of security features. It organizes security information and defines its range of operations. A realm has its own idea of principals, permissions, and ACLs. Particular security domains are reflected in Java as realm instances. A realm determines how a user is authenticated and retrieves access control lists for given names.

## **Authentication**

Authentication is the process by which a user proves his or her identity to a system. Authentication takes place when a principal, such as a user, tries to access an application. It is verified using identities such as tokens, passwords and certificates. It is performed in two steps:

- ▶ First, the server collects the credential data from the user.
- ▶ Second, it verifies the data against the information stored in a user registry or directory service.

Successful authentication is necessary, but not sufficient, for gaining access to protected resources. The authenticated user should be authorized to use the resource.

## Authorization

Authorization is the process by which the application server grants or denies permission to invoke the methods of an enterprise bean. Logical authorization is preceded by authentication. Note that for this feature, granularity is an important attribute. That is, highly trusted resources should be granted more access than those of more dubious origin.

## Non-repudiation

Non-repudiation is an important benefit that assured authentication provides. That is, the related entities can no longer deny knowledge of a transaction. The security system can prove they took part. As a result, cooperating parties can now execute electronic agreements because each side can authenticate and prove the other's participation

## Controlling information access

It is preferable to set up access control by declaratively defining method permissions in the enterprise bean's deployment descriptor. This enables container-managed security. Security holes is also less likely with container-managed security, because the security mechanism is implemented by the container.

Some times procedural access control is necessary for application specific situations. For example, an application might make authorization decisions based on the day of the week, or the internal state of an enterprise bean. Enterprise beans can ensure access control by using the `isCallerInRole` method, defined in the `EJBContext` class.

## Security role

A security role is basically a named grouping of information resource access permissions specified for a resource such as an application. Associating a user entity with a role admits the defined access permissions to that user as long as the user acts with the role. Each security role name is defined in a deployment descriptor.

For example, the following enterprise bean deployment descriptor fragment defines a role named `admin`, which is a reference to a superuser role. The `AccountBean` class uses the symbolic `admin` name to check permissions.

```

<enterprise-beans>
 ...
 <entity>
 <ejb-name>AccountBean</ejb-name>
 <ejb-class>AccountBean.class</ejb-class>
 ...
 <security-role-ref>
 <role-name>admin</role-name>
 <role-link>superuser</role-link>
 </security-role-ref>
 ...
 </entity>
</enterprise-beans>

```

## Methods

Enterprise beans provide these methods:

- ▶ The `getCallerPrincipal` method defined in the `EJBContext` interface returns the principal object. Using that object, we can identify the caller of the enterprise bean.
- ▶ To determine the role of the caller of the enterprise bean, we can use the `isCallerInRole` method:

```
boolean authorized = context.isCallerInRole("SuperUser")
```

Where, `context` is the variable of the `SessionContext` object.

**Attention:** `isCallerInRole` is not supported in WebSphere Version 3.5.

## Method-level security

Before enterprise beans, restricting access to an object or method by a particular user, was very difficult. However, enterprise beans now allow method-level security on any enterprise bean or method. Users and user groups can be created which can be granted or denied execution rights to any EJB or method.

In WebSphere, the user groups can be granted or denied access to Web resources such as servlets, JSPs, and HTML pages, and the user IDs can be passed from the Web resources to the EJBs by the underlying security framework.

A Java client that is handling secured resources will prompt for a user ID and password when a request is made to invoke a protected method on a bean. On entering the user ID and password, the method invocation request is passed to the application server. On successful authentication, the security collaborator will

consult the security application to make authorization decisions based on the principal making the method invocation. If the user is authorized to invoke a method, then the security collaborator will consult the delegation policies and set up the appropriate security context. If this invocation makes a downstream method call on another bean, that method call will be accessed under the user's identity. When the second method request is made on that bean, the security runtime will perform the security check based on the user's credential set on the security context.

## Security setting in VisualAge for Java

To set the security access for the bean in VisualAge for Java, follow these steps:

- ▶ In the EJB page, select the enterprise bean for which you want to set the properties, and *Properties*. The properties dialog appears (Figure 20-2).
- ▶ In the Run-As Mode drop-down box, select a run-as mode. This selection tells the container the security identity to associate with the execution of the enterprise bean method.
- ▶ Click *OK*.

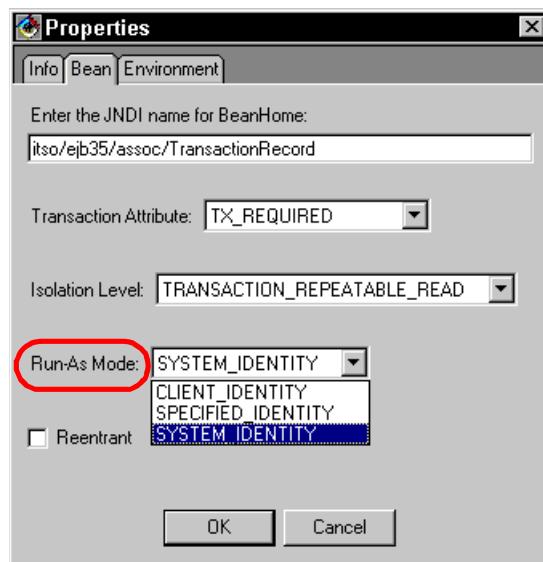


Figure 20-2 Setting the access identity for an enterprise bean

There are three possible options available for Run\_As Mode:

CLIENT\_IDENTITY      The security identity of the client is used when running the EJB.

SPECIFIED_IDENTITY	The security identity of a specified user account is used when running the EJB.
SYSTEM_IDENTITY	The identity of a privileged account is used when running the EJB. The container maps the abstract notion of a <i>privileged account</i> to a suitable privileged account on the underlying platform, such as the database administrator or the operating system administrator account.

## Security components

A WebSphere administrative server hosts the enterprise beans that make up the WebSphere Application Server system management facility including the security application. The security runtime consists of two core components:

- ▶ A security plug-in attached to a Web server. The plug-in helps make security decisions when users request Web resources such as HTML files and servlets from Web clients communicating over HTTP. It protects the URL name space and not the physical file name space. so two urls pointing to the same resource may have different access control policies.
- ▶ A security collaborator attached to every application server. The collaborator makes security decisions on method calls on resources hosted by the application server.

These two runtime components consult the security server, which controls security policies and performs authentication and authorization services.

## Security server

The security server, which is part of the security application, has essentially two purposes:

- ▶ To centralize control over the security policies such as permissions and delegations.
- ▶ To provide central security services such as authentication and authorization.

The security runtime components acquire security policy information from the security application. The security plug-in uses the security policies to determine which authentication and authorization services to invoke. The security collaborator uses these policies to make the authorization and delegation decisions.

## Implementing EJB security

In order to set up the security for an EJB client, the system administrator must perform the following steps:

1. Make sure that security has been enabled and configured in WebSphere.
2. Create an EJB container within the application server to hold the resources.
3. Deploy the beans into the container.
4. Create an enterprise application to be used in securing the enterprise beans.  
Place the beans in the enterprise application.
5. Configure application security for the enterprise application. This is an optional step when securing only EJBs.
6. Assign the application's methods to method groups. Run this task for all resources that make up the application.
7. Grant the client, the appropriate permissions. Use the assign permissions task to grant him or her access to the method groups.

The details on how these tasks are performed go beyond this redbook. Refer to the *WebSphere V3.5 Handbook*, SG24-6161, for detailed information.

## Summary

Security should always be a major part of the effort put into a real application on the Web. This is not only true in the Internet world, but also for intranet applications (disgruntled employees cause more damage than external hackers).

This book is about enterprise bean development and deployment. Although security can be applied at the enterprise bean level, an overall implementation must be employed for secure Web applications. This goes beyond the content of this book.



# EJB modeling with Rational Rose

In this chapter we describe how an entity model developed in Rational Rose can be converted into an EJB group in VisualAge for Java. We will only discuss the basic facility of importing a Rose model into the VisualAge for Java EJB development environment, but not go into details about the Rose modeling tool itself.

## Preparation

We assume that you have Rational Rose installed and know how to work with the product.

## VisualAge for Java

On the extra CD of VisualAge for Java you find the technical preview tool for importing Rose models into an EJB group. The extras\xmib directory contains two files: readme.txt and xmibridge.dat.

Follow the directions in the readme file to setup VisualAge for Java:

- ▶ Import the projects from the xmibridge.dat repository file into the VisualAge for Java repository.
- ▶ Add the *VisualAge Persistence XMI Toolkit* and *XMI Toolkit Imports* projects to the Workbench.

## Bank scenario

In this scenario we develop the banking model in Rational Rose and import the model into an EJB group of VisualAge for Java.

### Model in Rational Rose

Figure 21-1 shows the banking model as defined in Rational Rose:

- ▶ We defined the entities with attributes.
- ▶ We defined a few operations (methods) in the entities, such as deposit and withdraw in the BankAccount.
- ▶ We defined the Checking and Savings classes as subclasses of BankAccount. These associations are called *generalizations* in Rose.
- ▶ We defined all associations with cardinalities and appropriate role names.
- ▶ We defined the m:m association between Customer and BankAccount as two 1:m associations.
- ▶ The intermediate CustAcct class has no attributes.

We saved the model as ejbbank.mdl in sg2461244\sampcode\rose.

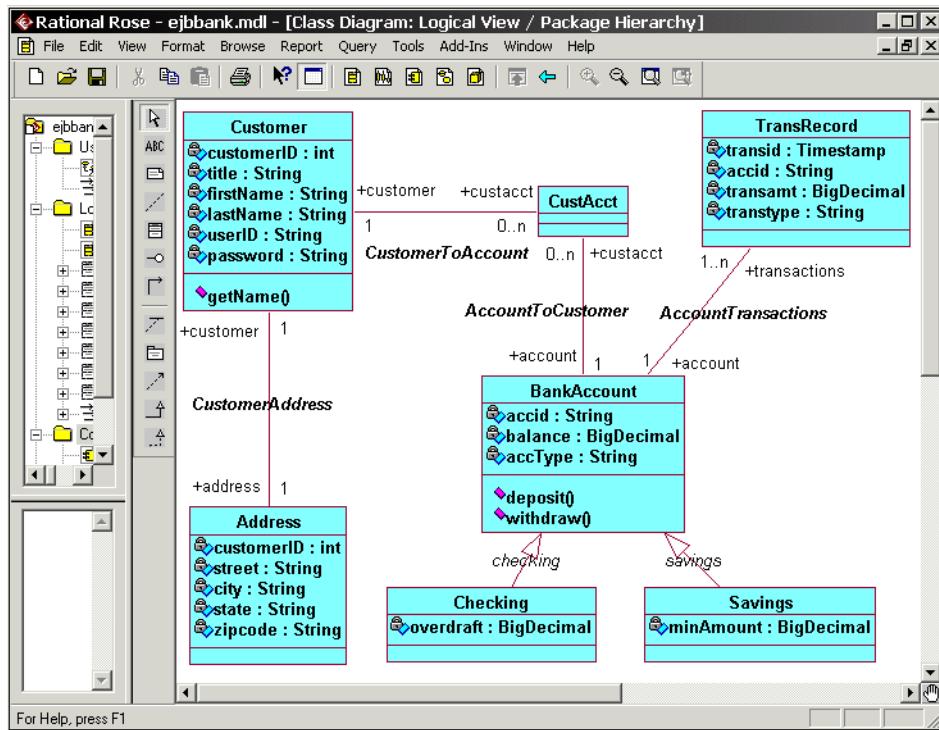


Figure 21-1 Rose model of the EJB BANK

## Importing the model into an EJB group

In VisualAge for Java define a new EJB group, for example, Rose\_Bank, then follow these steps:

- ▶ Select *EJB* -> *Add* -> *Import from Rose or XMI*
- ▶ In the import SmartGuide, enter the name of the model file, the EJB group, and the code package (Figure 21-2).
- ▶ Skip the next page (virtual paths) and click *Finish*.
- ▶ The EJBs are generated into the EJB group.

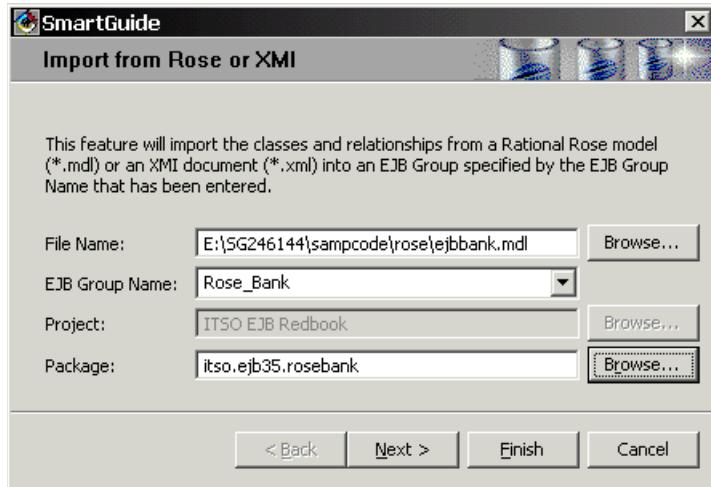


Figure 21-2 Importing a Rose model into an EJB group

## Tailoring the model in VisualAge for Java

Figure 21-3 shows the EJBs after importing the Rose model. The properties pane lists the four associations. All entity beans report errors.

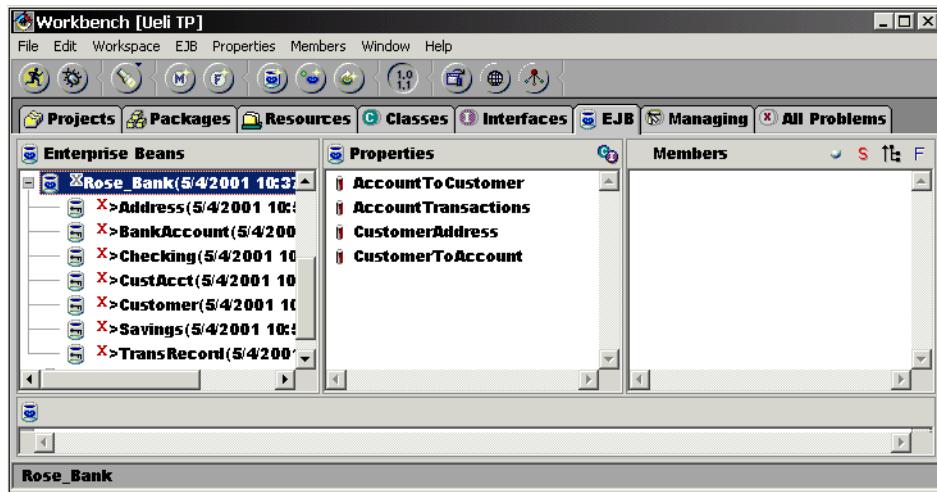


Figure 21-3 EJB group after import from Rose

## Fixing the errors

The errors are reported because no attribute has been designated as key.

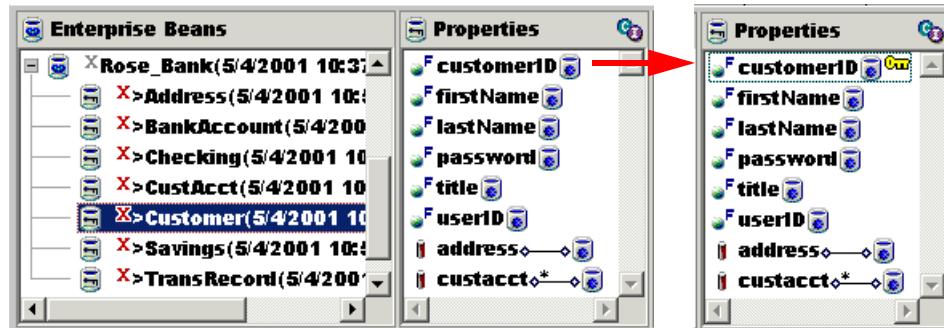


Figure 21-4 Setting the key attribute

Setting the keys for Customer, BankAccount, TransRecord, and CustAcct fixes all problems reported:

- ▶ Note that setting the key for BankAccount also fixes the Checking and Savings subclasses.
- ▶ The key of the CustAcct class is composed of the two associations roles.

At this point the Rose\_Bank group resembles the Bank\_Entities group we developed in Chapter 17. “Associations” on page 353.

The associations and the inheritance structure are mapped perfectly. The operations (deposit and withdraw methods) are not imported and must be added manually.

We did not develop this model any further for the redbook.

## Further tailoring

To make the Rose model operational, you would have to:

- ▶ Remove the `transtype` attribute from `BankAccount`; it will be used as discriminator column to distinguish between checking and savings accounts.
- ▶ Create schema and map:
  - For new development you can have the schema and map generated (*EJB -> Add -> Schema and Map from EJB Group*).
  - Import the existing tables into a schema and then map the entities to the schema.

## **Summary**

The Rational Rose modeling tool can be used effectively to create and document a new model and then import it into the VisualAge for Java EJB development environment.

The basic entity and association facilities are imported very well into an EJB group and can be converted into an operational EJB design with not too much further effort.



## Part 4

# EJB design guidelines and best practices

In this part we recapitulate enterprise bean concepts and give guidelines about when to use which EJB, how to design applications, and best practices for transactions, persistence, and performance.





# EJB design considerations

In this chapter we present guidelines that should help you with applications that use enterprise beans.

We start with guidelines for when to use EJBs and which EJBs to use. We then talk about the model-view-controller paradigm that fits very well for implementation of Web applications using servlets, JSPs, and EJBs.

Finally, we propose an architecture for applications using the enterprise beans technology.

## When EJB

If you are developing a new Web application, you have probably asked yourself where Enterprise JavaBeans (EJB) components fit into the whole architecture. Let us examine through some business requirements and technical merits to determine whether EJB is an appropriate technology for a given situation.

### Isolate business logic components

The business rules and data structures that make up the business logic are very sensitive data for any company. Therefore, it is a good idea to keep business logic behind a secure set of firewalls. Here, distributed object component architecture, such as EJB technology, enables you to isolate valuable corporate assets behind servers, while allowing presentation code to access the EJB servers for serving client requests.

### Independence from database implementation

EJB technology enables a clear separation of business logic from database access. Because the business logic is independent of the database schema applications can be deployed into organizations with different or changing database schemas. Only the mapping of container-managed entity beans may require tailoring when deploying to another organization.

### Multiple client access

Often, a single application will have multiple type of clients that have to access the same set of information. For instance, an application might have Web-based customers, and also Java-based client applications. However, this is not efficient, neither in programming time nor in utilization of the database, if multiple database locks could be held at one time.

The EJB components control the access to the back-end data and manage the current transactions and database locking internally. There are other solutions that may also work in this situation.

For instance, Java applications can access Java servlets through HTTP, while a browser can access a Java servlet through HTTP as well. The problem with this solution is that, if a servlet is used to present information in a browser, it contains some presentation logic that is unnecessary for conveying information to another program. Therefore, we end up with different implementations of servlets to handle both cases.

## Concurrent read and update

Managing access to shared data at the database level often results in complex schemas to deal with database locking and concurrency, or alternatively, loss of data integrity when these issues are not considered. As mentioned previously, the EJB components control the access to the back-end data and manage the current transactions and database locking internally.

## Access multiple data sources

Many applications require the ability to access multiple data sources. The key is that some applications require this access to be fully transactional, so that data integrity is maintained across the data sources. EJB is an ideal choice for this case, because it supports distributed transaction across multiple data sources.

## Method-level security

Until the advent of EJB technology, there was no way to restrict access to an object instance or method by a particular user. However, EJB technology allows method-level security on any EJB component.

# Which EJB

In this section, we discuss when session beans, entity beans, and JavaBeans are appropriate.

## Session bean

A session bean can be considered in the following cases in general:

- ▶ To perform a task for a client, for example, to perform a money transfer in a typical banking scenario.
- ▶ When the conversational state with the client does not have to be persistent.
- ▶ When the life-time of the bean instance can be terminated when the life-time of the client session ends. For example, the money transfer bean instance holding the account details, such as balance and account ID, can be garbage collected after the customer logs out of the application.
- ▶ For a relatively simple application.
- ▶ To encapsulate the business logic of the application.
- ▶ When the instance is not shared with many clients at a time.

## **Stateful session bean**

Consider using stateful session bean, when the following conditions are met:

- ▶ The state of the bean has to be initialized when it is created.
- ▶ The conversational state has to be held across multiple method invocations.
- ▶ The client is an interactive application, invoking different tasks related to each other.
- ▶ The bean represents the client.
- ▶ A transaction has to span across multiple methods of the session bean.

An example can be an online test, that is implemented by a stateful session bean, where the success rate of questions answered so far by a user is stored in the bean, which is used for deciding the components of successive questions.

## **Stateless session bean**

A stateless session bean can be considered in the following situations:

- ▶ To encapsulate a high-level of reusable tasks, that do not need a client state, for example, product list enquiry and feed back submission
- ▶ The bean does not need to hold information across method invocation
- ▶ To provide a facade to entity beans
- ▶ To encapsulate database query logic
- ▶ To transparently reuse instances to serve different clients

Examples of stateless session bean are income tax calculator, money transfer, and currency converter.

## **Entity**

Consider using entity beans over session beans in the following situations:

- ▶ To encapsulate the persistent data on the database
- ▶ To be shared across multiple clients at the same time
- ▶ To always reflect the current state of the data
- ▶ To represent major persistent business objects
- ▶ To identify each instance of the object with an unique identity

## CMP beans

Consider using CMP over BMP beans in the following situations:

- ▶ When there is a suitable JDBC driver and the right level of JDBC is supported by the container
- ▶ To simplify the development process
- ▶ When the application is large

## BMP beans

Consider using BMP over CMP beans in the following situations:

- ▶ When the developers need customized access to the data
- ▶ To have more flexibility in the development process
- ▶ When the EJB container's CMP features are insufficient to meet the requirements of the entity bean
- ▶ To synchronize the bean state with multiple datasources
- ▶ To handle sophisticated object-to-relational mapping not supported by the container

## JavaBean

Some enterprise class attributes have to be treated as a single value. For example, in the Customer bean class of the banking application the fields of the Address object are always used together. Using a remote interface for such entities is not advisable, because it requires a lot of unnecessary server communication. Therefore, all the data for the Address object can be retrieved once, sent to the client from the server in serialized form, and instantiated on the client. Subsequent accesses to the Address object are local, require no server communication, and use fewer resources.

Local access obviously has lower latency than access through a remote interface, so interface responsiveness improves and network traffic decreases. Such a client-side object can be created as a JavaBean, because it represents a concrete value from the server, not a reference to an object on the server.

Use JavaBeans when the business entity being modeled has:

- ▶ Only methods that get values from the object's internal state
- ▶ Life-cycle that is completely controlled by another object
- ▶ Relatively small size

# Design patterns

In this section we discuss some design patterns involving EJB. Refer to the IBM Patterns for e-business for overall system design guidelines:

<http://www.ibm.com/framework/patterns>

## Model-view-controller

Graphical, single-user applications are commonly organized around event-driven user interfaces, such as buttons and check boxes. The developer creates the user interface with a GUI building tool, then develops blocks of code that execute application responses for user actions. Some design methodologies emphasize starting with the user interface, which all too often solidifies into the final system design. The result is a program organized around user interface elements and user actions on those elements, with persistent data manipulation, application functionality, and display code completely intermixed.

But such an approach is not suited for distributed applications for several reasons:

- ▶ More sophisticated applications often require multiple data views and data manipulations. Implementing business logic in alignment with display code often results in redundant procedures in multiple places, which causes display inconsistencies. Any change to data display must be updated in multiple places, resulting in software flaws.
- ▶ Figuring out the application logic in a later time is very difficult, when data manipulation logic, formatting and display code, and user event handling are entangled.
- ▶ There will be redundant data accesses, because GUI elements individually query for similar information. Also, maintaining consistency between multiple viewers of a shared, dynamic data store is very difficult. The solution simply does not scale.
- ▶ Code is less reusable.

The model-view-controller (MVC) design pattern separates the application data GUI code and the accessing logic. It increases reusability by partially decoupling data presentation, data representation, and application operations. It also enables multiple simultaneous data views.

- ▶ The **model** component represents the application data, plus methods that operate on that data, with no user interface.
- ▶ The **view** component presents the data to the user.
- ▶ The **controller** component translates user actions into operations on the model. The model, in turn, updates the view to reflect changes to data.

In an application involving EJBs and realizing the MVC pattern, the model is implemented using a combination of JavaBeans and EJBs. This combination may access or encapsulate business logic by:

- ▶ Connecting to a local or remote database
- ▶ Wrapping business logic that may run on the application server
- ▶ Interacting with remote business logic through RMI-IIOP

The view is implemented using JavaServer Pages. The controller is implemented using servlets that contain the session control required to maintain a state during a user session (Figure 22-1).

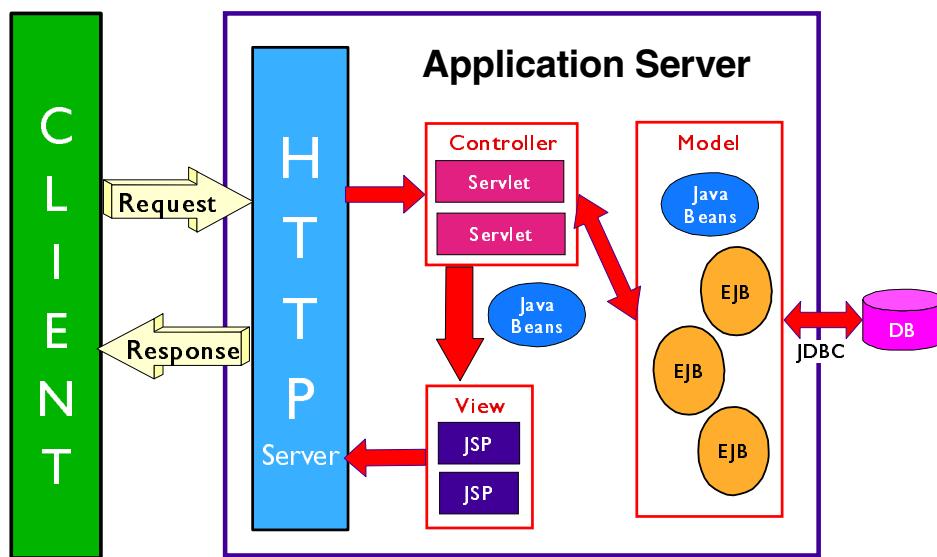


Figure 22-1 MVC pattern

Using MVC can have the following merits:

- ▶ Separates data modeling issues from data display and user interaction
- ▶ Different people skills are involved when developing the model or the view or the controller
- ▶ Allows the same data to be viewed in multiple ways
- ▶ Allows the same data to be viewed by multiple users
- ▶ Improves maintainability by encapsulating application functions behind well-known APIs
- ▶ Enhances reusability by decoupling application functionality from presentation

- ▶ Makes applications easier to distribute, since MVC boundaries are natural interface points
- ▶ Can be used to partition deployment and enable incremental updates
- ▶ Facilitates testability by forcing clear designation of responsibilities and functional consistency
- ▶ Enhances flexibility, because the data model, user interaction, and data display are “pluggable”

## Session entity facade

Entity beans provide object-oriented access to server-side business logic, and are commonly shared across multiple client applications. For example, both a customer relation management application and a general account management need access to customer information, although the specific details accessed could be different.

A common design approach is to create a single integrated data model. A component layer of enterprise beans will be implemented above the data model, that abstracts the persistent data from the database in component classes. Application development in this context involves developing a user interface which manipulates a composition of back-end component instances.

This approach can greatly improve application integration, since all applications share the same "view of the world", and the beans' remote interface clearly define what services are available from them. The drawbacks of this approach are with the complexity of large data and programming models:

- ▶ First, designers and developers have to understand an enormous, highly-normalized data model or set of component APIs, when writing an application that only uses one small corner of the model.
- ▶ Second, the entities in such a data model can often represent data abstractions that are not of any interest to a client application developer.
- ▶ Third, if applications are written directly in terms of the underlying database schema, any change in the schema can potentially affect multiple parts of applications. Writing applications in terms of database schema makes the schema difficult to modify.

For these reasons, it is often useful to provide an implementation of application functions in a single session bean class. This approach of designing is called a *session-entity facade*. Clients of the application use the session facade as a single access point for application functionality and data access. Meanwhile, it retains integration with the enterprise data model, because the session entity facade is also an enterprise bean. The facade should be a stateless session bean.

Figure 22-2 shows the facade pattern.

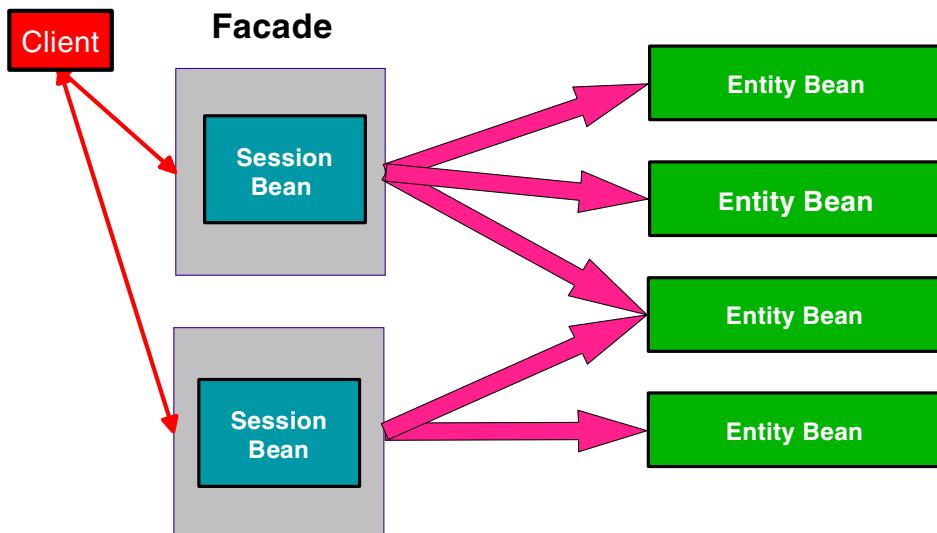


Figure 22-2 Facade pattern

The session entity facade pattern has the following merits:

- ▶ Simplifies client design.
- ▶ Supports thin client development. Because the application functionality is concentrated in the session entity facade, the client is left with little to do besides managing presentation and user interaction.
- ▶ Unifying the server interface within one class concentrates design discussions on what services are to be delivered, not how the services are implemented on either side of the facade.
- ▶ Instead of defining transaction control for all entity enterprise beans on the back end, application transaction control can be centralized at the facade.
- ▶ Decoupling client and server implementations promotes code reuse, because of decreased dependencies between modules.
- ▶ Reduces the number of select and update calls on the database. Each transaction calls in the beginning an ejbLoad and at the end an ejbStore. When a client directly interacts with the entity bean, then each method invocation (getter or setter) starts its own transaction and generates one select and update call on the database. However, when session beans are used for invoking getters or setters on behalf of the client, only one transaction is started, therefore one select and update call are made on the database, irrespective of the number of method calls.

## Batch session bean

This design pattern enables updating multiple server side object states within a single network communication. Therefore, it decreases network traffic by moving the update iteration to the server.

In specific cases a distributed application has to update multiple server objects simultaneously. Consider a client program accessing entity beans. It acquires a list of primary keys from an enterprise bean finder method, and then individually updates the enterprise beans through those beans remote interface.

Unfortunately, the server has to create and perform transaction management for each of these instances, resulting in unacceptable overhead.

The batch session bean pattern addresses this issue. It encapsulates the transaction to be performed on multiple server objects into a server-side stateless session bean.

Essentially, the batch session bean pattern improves performance by modifying multiple database rows represented by multiple entity beans with a single update statement, thereby avoiding the unnecessary overhead of entity beans construction and maintenance.

## Command framework

As an alternative to session beans WebSphere provides the command framework. A command encapsulates business logic tasks and provides a standard way to invoke business logic requests and data access form a client (for example, a servlet) to a server (for example, CICS, DB2, EJB server).

A command has these characteristics:

- ▶ A command object corresponds to a specific business logic task.
- ▶ A command hides the specific connector interface.
- ▶ A command has a simple uniform usage pattern.
- ▶ A command is implemented as a serialized object and can be shipped to another JVM for execution.

We did not experiment with command for this redbook. For more information on command refer to *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*, SG24-5754.

## Best design practices

In addition to discussions on choosing specific bean types, there are other design choices to be considered, when developing objects for the EJB tier. These choices include the type of business objects that could be enterprise beans and the role an enterprise bean can play in a collaborative environment.

Since enterprise beans are remote objects, they consume lot of network bandwidth. So, it is not suggested to model all business objects as EJBs. EJBs are suited for business objects that are accessed directly from the clients. Other objects can be modelled to encapsulate the data access logic and depend on EJBs.

Also, it is not suggested to enable direct client access for all EJBs. Therefore, some EJBs may act as the communication bridges between the client logic and the EJB modules.

## Data access strategies

Let us start this section with an example. Consider that you have a product catalog with thousands of items. Each item is categorized and a user can retrieve the items for a specified category using a Web application. We use finder methods to retrieve the products for a specified category, and assume that the query is returning thousand hits. The container has to transform the result set into EJBs, whose references are then placed in an enumeration and sent back to the EJB client. This significantly increase the response time of the container.

Instead, a better design would be to implement a session bean that uses JDBC to retrieve the result set data that the end user is querying. When the user is interacting with a single data element, such as by clicking the URL link of the product, the corresponding entity bean is instantiated and the reference to this object is communicated to the client. Using this approach, you avoid creating many objects and slowing down the server.

## Logic splitting

In this section we discuss modelling of business objects into the combination of different types of enterprise components based on the business logic.

### Data access

Data access logic can be modelled as separate data objects. The use of separate data objects with session beans results in the following merits:

- ▶ It keeps session beans simple and clear.
- ▶ Migration to container-managed persistence will be easier.

Let us discuss the various scenarios of database access, with an example as shown in Figure 22-3, where we consider three approaches.

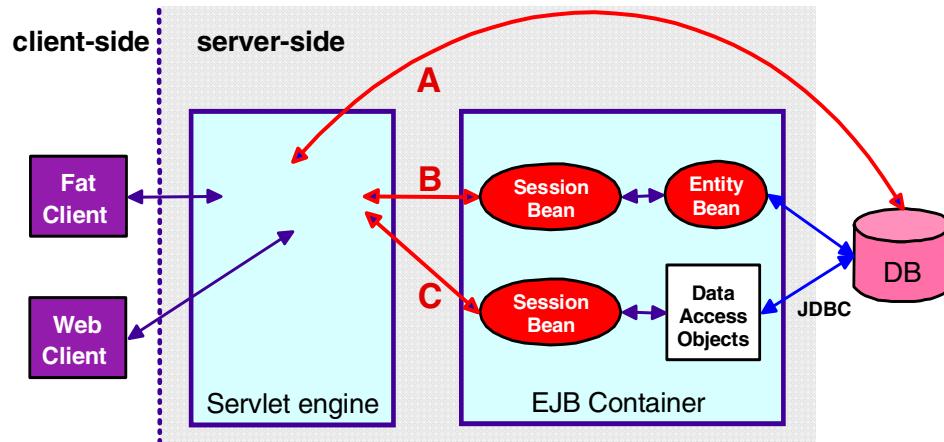


Figure 22-3 Data access in session beans

#### **Approach A**

In this approach the HTTP client interacts with the servlet, which directly accesses the database for all operations. But servlets, as discussed earlier in the MVC pattern, are better suited for bridging the user actions into business operations.

#### **Approach B**

Here the session beans model the business logic and use entity beans for all database interactions. This approach is suited in cases where the user wants to interact with a single database record, modelled as an entity bean. For example, when a user is changing an appointment, this approach is considered. The appointment is represented as an entity bean instance and the session bean models the business logic and interact with the entity bean for accessing and updating the appointment details. This approach is not suited in cases where the user has to retrieve a large amount of data records. For example, the user wants to view all the appointments. This approach would result in the container creating a large amount of entity bean objects, each representing an appointment, just for the purpose of a simple view.

#### **Approach C**

Here the session beans can use data objects to access the database. The session beans can directly interact with the database, but by using data access objects we separate business logic from data access logic. This approach is

suited in cases, where the user wants to view, for example, the product list of a company or all the appointments for the coming week.

## Business logic

Now let's discuss logic splitting in terms of business modelling and data operations. We consider the bank account bean and discuss how to model the object using different approaches.

In our example (Figure 22-4), we use the Account bean as an business object to serve banking operations related only to a single account. The operations that we wrap in the entity bean are deposit and withdraw. We also use the Account bean, as the data persistent module, to update the data in the database.

For business operations involving more than one object, we use an external component. In our example, we use the MoneyTransfer session bean, which interacts with and updates the two entity beans for account to account money transfer. This is an approach that utilizes entity beans both for data persistence as well as serving business methods that affect only the data of that instance. This prevents one account instance acting on another account data, by handing over the common logic to secure third party components. The clients can be either modelled as session bean facades to the business logic beans, or as simple Web clients.

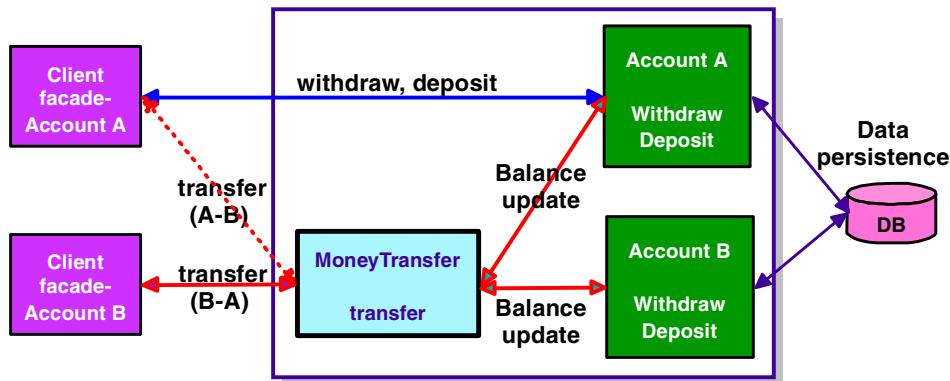


Figure 22-4 Modeling part of business logic with data persistent in entity beans

Another approach could be moving all the business logic operations to the session bean and use the entity beans only as the data persistent components. This approach is easy to develop and maintain. But such modelling does not ideally represent the business logic of the application and is also does not fully utilize the capabilities of the entity beans.

# The big picture: architecture

By now, you should be more than familiar with all the basic and some advanced concepts around enterprise JavaBeans. Before we continue, let's take a look at some of the most important topics we have discussed so far.

We saw the two different types of enterprise beans, session and entity. We discussed how session beans encapsulate business processes and contain conversational states with a particular client. We also saw how entity beans are used to embody permanent business data and provide methods to manipulate that data. When the enterprise beans are deployed in the container, we presented different ways a client can access them. We also explained the need for transactions and how the container and the beans can manage them. And finally, we covered some basics about the EJB security model and the ways it can be applied on either a bean or a method level.

The question that comes up now is how to provide an overall architecture that will comply with the EJB model and ensure that we are gaining the most out of this technology when designing an enterprise application.

Figure 22-5 shows an overall architecture that is recommended by the team that wrote this book.

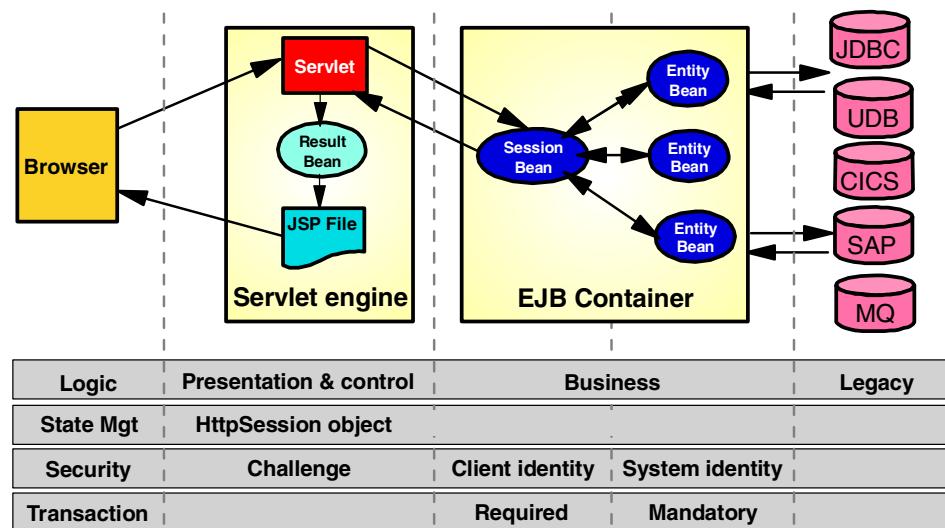


Figure 22-5 Overall architecture

Let's now discuss the design approach we used and see in more detail the four different layers we introduce in our architecture.

## About the model

The architectural model we use here is based on the model-view-control programming paradigm. All the interactions between the user and the application is done by the servlet. The business logic is defined in the EJB model and all the presentations to the user is delivered by JSPs.

Inside the EJB container, entity beans are used mainly for representing business data either in database or legacy systems. CMPs is the first choice. BMPs should be used, if the database or the JDBC level is not supported by the container, or the object has to map either to a complex data schema (joins of tables) or a legacy application (CICS).

Session beans add the main layer of business logic to our application and provide a facade to the entity beans. Stateless beans are the preferred ones. They are lightweight and have better performance. Stateful beans should be used seldom and only when the requirements restrict the use of stateless.

## Logic

As shown in Figure 22-5, the application logic is split into several pieces. This gives better overall control of the application and separates the development roles to distinct categories.

Control logic	The servlet is responsible for receiving a user request, parsing the parameters and passing them over to the beans.
Business logic	The processing of the request is done by the session and entity beans. They encapsulate all the business logic and the methods required. It is not always easy where to draw the line between them. Entity beans usually provide methods for manipulating the persistent data. Session beans represent most of the business logic by providing all the other required methods.
Presentation logic	The result of the request is assembled inside a JSP and the response is returned to the user.

## State management

All the information about the current user and the history of the interactions with the application should be preferably stored in the HttpSession object. The session object is designed for this purpose and it should contain all the session data of a user.

Storing all the session data in the HttpSession object allows us to use stateless session beans and achieve higher performance. The servlet manipulates the

session object and passes all the required session data to the session beans. Although stateful beans can store the state, they should be avoided due to poor performance and scalability. A typical example of session data is the shopping cart that contains all the products a customer wants to buy. It must be serializable and lightweight.

**Attention:** In the next version of WebSphere, multiple Web applications cannot share a single HttpSession object. Consider that when designing your application and servlets. If you have servlets that are defined within different Web applications, these servlets cannot use the same session data.

## Security

The user has usually public access to non-critical information. In order to access services and sensitive data, the user has to be authenticated. A personalized view can be made available, after a user has logged in.

It is the servlet that will trigger the security mechanism and challenge the user. After the user is authenticated, further control can be applied by the enterprise beans at the bean method level. As session beans act as facade to entity beans, most of the security control can be enforced by session beans.

Defining all the user roles and access control lists for the session beans, provides secured access to the entity beans and the persistent data they represent. Therefore, it is adequate if the client identity is used in the session beans and a system identity is generally used in the entity beans. This saves a great deal of administration work at deployment time and makes the application faster.

## Transactions

It is a general recommendation that entity beans should not initiate transactions. Instead they should require a transaction running, every time before they are invoked. If there is not transaction context, they should throw an exception. This is guaranteed by marking the entity beans as TX\_MANDATORY.

This approach ensures that entity beans cannot be accessed directly by clients, but only by session beans that have already started a transaction. This decreases dramatically the number of SQL statements to the database and prevents unauthorized access to the business data protecting the application from malicious attacks.

Session beans should delegate the transaction management to the container and not manage it on their own. You should avoid using TX\_BEAN\_MANAGED.



## Best practices for EJBs

In this chapter, we summarize all the tips we have posted here and there in the book. Having them altogether with pointers will help those of us who already know the EJB technology.

## Transaction tips

Here are good practices linked to transaction management:

- ▶ Make sure you use WebSphere data sources (and thus the WebSphere connection pooling) to ensure that the transaction manager is able to commit the updates in the right place.
- ▶ Avoid mixing isolation levels inside a single transaction. A connection is created at the beginning of a transaction with the isolation level of the method that started the transaction. If you try to change the isolation level, you will get an exception.
- ▶ Try to set entity method transactional attributes to TRANSACTION\_MANDATORY to ensure that no client can invoke the entity method directly (unless using a client-demarcated transaction). Have the session facade methods provide the transaction by setting its transactional attribute to TRANSACTION\_REQUIRED.
- ▶ Use the *Find For Update* setting to avoid deadlocks (see “Deadlocks” on page 239 for more information).

## Persistence tips

In this section, we outline tips in two different domains. One is a general set of tips and the other one is geared specifically at object associations with a specific section on m:m relationships.

### General tips

These tips concern all beans that needs persistence:

- ▶ When implementing BMP entity beans, ensure that a WebSphere datasource is used to connect to the database. It will provide connection pooling and also some level of abstraction towards the actual database. See the abstractions used in Chapter 8. “Bean-managed persistence entity beans” on page 139.
- ▶ In CMP entity beans with top-down approaches, do not use the generated schema as such. Tune it and avoid, for example, to use VARCHAR(30) when you can use CHAR, especially to store two or three characters, such as the title of a customer.
- ▶ For CMPs, avoid serializing data in the database, because it cannot be queried, read back by non-Java report writers and it can potentially use a lot of space. This is especially true for shared databases.
- ▶ Make sure that all getter methods of the entity beans are marked as read-only (see Figure 7-6 on page 113). Set this attribute also for methods that do not

modify the bean state (“Marking methods read-only” on page 119). It improves the bean performance, by instructing the container not to invoke the ejbStore method of the bean at the end of the method call. One of the major benefits of this is that no coding is required, it is mainly an administrative setting.

## Entity associations

Often you find a set of persistent objects with complex relationships 1:m and 1:1 relationships. When a user must navigate a tree or graph structure then often CMP EJBs are the best solution.

- ▶ CMP entities in WebSphere cannot handle arbitrary SQL like joins (other than on shared primary keys) or views. If you need to handle these, you must use another solution such as BMP or *session wrapped* persistence.
- ▶ CMP entities do not handle relational joins very well. It happens that some object structures are naturally mapped to joins, these are often needed for efficiency. In these specific cases, a BMP entity is the best solution.
- ▶ The ideal solution is to handle joins in one SQL call. VisualAge for Java Persistence Builder (PB) solves this with a feature called *preload* which adds each table into the join that initially happens. This feature will be added in WAS 4.0, but it could be simulated in BMP beans by storing read data in a cache shared by all entities.
- ▶ Here is a performance trick when writing bean-managed persistence beans. It is convenient to have a BMP to manage an object association, as in an Order/OrderLine relationship. It is more efficient to load only the order information in the ejbLoad method, and to defer loading the order lines until they are actually needed using the lazy initialization technique.
- ▶ In many domains there is a set of objects whose state is frequently read but rarely updated, such as geographical locations (states, counties) or codes and lookup tables. The key is that these objects are not transactional. They do not have to be EJBs; instead, make them dependent objects managed by a session bean, which could prove useful to cache the data.
- ▶ If you have a set of write-only objects, such as a system log, you never want to read the log entries, and therefore it is inefficient to use entity beans. A session bean and dependent object solution will fit a lot better.
- ▶ If you have to scroll through a large list, using entity beans to retrieve some fields is very inefficient. You can use a session bean and lightweight object solution to retrieve the data using JDBC. If you need the corresponding entity bean, the lightweight object should hold the key value to facilitate a retrieve using findByPrimaryKey.

## m:m relationships

You usually cannot map the result of a join to a single CMP entity bean as each CMP entity bean corresponds to a single row in a table. There is no way for tying multiple entity beans together. In this case, consider using BMP with dependent objects (JavaBeans). Persistence will then be managed by a BMP entity bean or by some other persistence solutions.

For example, you can encounter a m:m relationships when a bank account can have several owners and each owner can have several accounts (Figure 23-1).



Figure 23-1 Customer - BankAccount m:m relationship

You can implement this relationship with CMP entity beans, but the overall result may have poor performance by using a link entity bean, CustomerAccountLink, which is associated with Customer and BankAccount by 1:m associations.

To find the account for one customer, you will have to:

- ▶ First execute one SQL statement associated with the finder (`findCustomerAccountLinksForCustomer(Customer aCust)`) on the CustomerAccountLink bean, which returns back all of the appropriate CustomerAccountLink beans.
- ▶ Then you must traverse those beans and execute the `findByPrimaryKey` method on each object to retrieve the corresponding account, which is another SQL statement.

This is already complex, but what will happen if you want to find the customers with account whose balances are greater than a certain amount. That will get you a collection of accounts and for each of those, find the associated CustomerAccountLink beans and then for each of those, get the corresponding customer. Thus you execute three different types of queries.

A simpler solution would be to use a BMP for the CustomerAccountLink association and use dependent objects for Customer and BankAccount. A single SQL statement in the `ejbFindBy` method to retrieve the collection and for each element in the collection, an SQL statement in each `ejbLoad` method that does a join of three tables.

## Implementation tips

These practices bring on some useful insights when you are about to implement enterprise beans:

- ▶ Remote interfaces can inherit from other remote interfaces. This permits to extend component services and client calls can exploit polymorphism.
- ▶ Bean implementation classes can inherit from other bean implementation classes and thus promote code reuse (see Chapter 16. “Inheritance” on page 329).
- ▶ Cache the home locally to your bean or better use a helper class to retrieve all the home interfaces and cache them (Figure 23-2). The only problem that can occur with this approach is that you could have *stale homes* (invalid references due to redeployment). In this particular case, you need to be able to flush the cache.

For servlets it may be better to retrieve the homes in the `init` method and cache them in static variables.

```
package itso.ejb35.util;
import java.util.*;
import javax.naming.*;
import java.rmi.RemoteException;

public class HomeHelper {
 private static java.util.ResourceBundle resGlobal =
 ResourceBundle.getBundle("EJBhome");
 private static InitialContext initialContext = null;
 private static Hashtable cachedHomes = new Hashtable();

 public static Object getHomeInterface(String className, Class aClassName)
 throws RemoteException {
 try {
 Object cached = checkInCacheFor(className);
 if (cached != null)
 return cached;
 InitialContext initialContext = getInitialContext();
 Object homeObject = initialContext.lookup(className);
 Object remoteObject = javax.rmi.PortableRemoteObject.narrow(
 homeObject, aClassName);
 storeInCache(className, remoteObject);
 return remoteObject;
 } catch (NamingException e)
 { throw new RemoteException("Name not found in JNDI"); }
 }
}
```

Figure 23-2 HomeHelper class - part 1

```

public static InitialContext getInitialContext() throws RemoteException {
 try {
 if (initialContext == null){
 Properties prop = new Properties();
 prop.put(Context.PROVIDER_URL,
 resGlobal.getString("Provider_URL"));
 prop.put(Context.INITIAL_CONTEXT_FACTORY,
 resGlobal.getString("Context_Factory"));
 initialContext = new InitialContext(prop);
 }
 return initialContext;
 } catch (NamingException e) {
 throw new RemoteException("Error retrieving initial context");
 }
}

private static Object checkInCacheFor(String className) {
 if (cachedHomes.containsKey(className))
 return cachedHomes.get(className);
 else
 return null;
}

private static void storeInCache(String className, Object remoteObject) {
 cachedHomes.put(className, remoteObject);
}

public void flushCache() {
 cachedHomes = new Hashtable();
}
}

```

*Figure 23-3 HomeHelper class - part 2*

The initial context class name and provider URL are stored in a properties file (Figure 23-4).

```

Context_Factory = com.ibm.ejs.ns.jndi.CNInitialContextFactory
Provider_URL = iiop:///

```

*Figure 23-4 Properties file for home cache: EJBhome.properties*

Sample code to retrieve a home:

```

CustomerHome custhome = (CustomerHome)HomeHelper.getHomeInterface
 ("itso/ejb35/cmp/Customer", CustomerHome.class);

```

## Performance tips

In this section, we explore a number of tips to achieve good performance.

- ▶ A way to speed up your entities is to cache them. By default the WebSphere EJB container uses the *Option C* caching option. This means a bean will be reloaded for every transaction. You could change this to *Option A*, which means that a bean will keep its state across transactions, which results in fewer SQL SELECT statements and better performance. But this approach can only be used for standalone server accessing the application database exclusively. If you need scalability (using cloning), forget about this option because the database is shared by the clones.
- ▶ Caching the homes as explained in the previous section is an excellent way to reduce the number of remote calls.
- ▶ Design your system so that all logic resides in session and entity beans, and not in the client code. By doing this, you also limit the number of remote calls.
- ▶ Use lightweight objects to exchange data between the EJB container and the client. This operation avoids remote calls to read attributes and it also reduces the amount of data transferred.
- ▶ Do not use XML to exchange data between client and server unless required. XML will cause more data to be transferred on the network because of the inherent redundant format of XML. It also has a performance impact on the server because there is a need to create and parse the XML streams.





## Part 5

# Appendices





# Setting up the environment

In this appendix we describe the installation procedure we used to run the redbook project. We installed:

- ▶ DB2 Enterprise Edition Version 7.1 plus Fixpack 1
- ▶ WebSphere Application Server Advanced Edition Version 3.5 plus Fixpack 3
- ▶ VisualAge for Java Enterprise Edition Version 3.5 plus Fixpack 2 or 3
- ▶ WebSphere Studio Advanced Edition Version 3.5 plus Fixpack 2

We also describe how to set up the EJB development environment in VisualAge for Java and how to setup WebSphere with an EJB execution environment and a Web application.

To test the examples we provide an EJBBANK database in DB2.

# Installation instructions for products

To install an environment similar to what we used to develop the example of this redbook, install the products as described here.

## Windows NT or Windows 2000

Install Windows NT 4.0 workstation (or server) with Fixpack 6a, or as an alternative, Windows 2000 Professional with Fixpack 1.

- ▶ For standalone operation you should install the MS Loopback Adapter.
- ▶ The system must have the Windows Scripting Host installed. This product may not be there on Windows NT; download it from

<http://msdn.microsoft.com/scripting>

## DB2 Version 7.1 Enterprise Edition

Installation procedure for DB2 Enterprise Edition:

- ▶ Select DB2 Enterprise Edition
- ▶ Select DB2 Application Development Client
- ▶ Select custom install with:
  - Communication protocols
  - Stored Procedure Builder
  - Sample Application
  - Documentation (remove Query Patroller, Warehouse, OLAP)
  - Administration/Configuration tools
  - Getting Started
- ▶ Installation directory, for example: **d:\SQLLIB**
- ▶ Create DB2 instance
  - User ID/password: db2admin/db2admin (use same value)
- ▶ Run *First Steps* after reboot and create the SAMPLE database
- ▶ **Change to JDBC 2.0 by running d:\SQLLIB\java12\usejdbc2.bat.** This is important for EJB development with data sources, otherwise you may get exceptions, such as `java.lang.AbstractMethodError`.
- ▶ Install **FixPack 1**
- ▶ Install the EJBBANK database. This database is described in “Bank database” on page 55 and the instructions for creating the database are in “Creating the database and tables” on page 62.

## WebSphere Application Server Advanced Version 3.5

Install the WebSphere Application Server if you want to run the examples in a real environment. Install Fixpack 3 afterwards.

Installation procedure:

- ▶ Language English
- ▶ Full install; this also installs HTTP Server and JDK 1.2.2
- ▶ User ID: use your Windows NT user ID (with administrative authority) or create a new user ID with full administrative authority
- ▶ Directory, for example: **d:\WebSphere\AppServer**, **d:\IBM HTTP Server**
- ▶ DB2 information should be filled in, use the db2admin user ID
  - Database was, jdbc:db2:was
  - Ignore warning that product exceeds level required
- ▶ Program folder: leave as is
- ▶ Reboot
  - WAS database is created
- ▶ In *Control Panel -> Services*: Start IBM WS AdminServer:
  - Wait.... the first start is slow because it initializes the WAS database
- ▶ Start the WebSphere Administrative Console: *Programs -> WebSphere -> Administrator's Console*:
  - Topology page: expand the node and start the Default Server
  - Open a browser and enter: <http://localhost/servlet/snoop> (it should work)
  - Close the Administrative Console
- ▶ In *Control Panel -> Services*: Stop IBM WS AdminServer

## VisualAge for Java Enterprise Version 3.5

Install VisualAge for Java, with the latest patches (**Patch 2** minimal, **Patch 3** suggested - VisualAge for Java 3.5.3).

Installation procedure:

- ▶ Click *Install Products*, select VisualAge for Java
- ▶ Select language
- ▶ Accept license
- ▶ Select custom install
  - Select Optional features pull-down, and select all subfeatures

- Optional: remove AS/400 and OS/390
- ▶ Installation directory, for example: **d:\IBMVJava**
- ▶ Let install go on... be patient at one point it seems to do nothing... but eventually it will continue
- ▶ Install the latest patch from the VisualAge Developer Domain:  
<http://www.ibm.com/vadd>

## WebSphere Studio Advanced Version 3.5

Install WebSphere Studio 3.5 plus Fixpack 2, or install Studio 3.5.2, which is available as a complete install. Studio is not required for any of the examples of this redbook.

Installation procedure:

- ▶ Accept license
- ▶ Installation directory, for example: **d:\WebSphere\Studio35**
- ▶ Select Studio, Applet Designer, Page Detailer (not Distributed Debugger)
- ▶ Program folder: leave IBM WebSphere
- ▶ Install **Fixpack 2** on Studio 3.5.

## Setting up the WebSphere execution environment

To implement the enterprise beans and execute the servlets and JSPs we setup an application server with an EJB container and a Web application.

### Configure the WebSphere administration server

First we start the administration server and create JDBC drivers and data sources, an application server with an EJB container and a servlet engine, and a Web application.

- ▶ Open the Windows Control Panel and double-click on the *Services* icon. (In Windows 2000, use *Programs -> Administrative Tools -> Services*.)
- ▶ Select the *IBM WS AdminServer* and click the *Start* button. Wait until the service is started.
- ▶ Launch the administrative console using *Programs -> IBM WebSphere -> Application Server V3.5 -> Administrator's console*.

- ▶ When the administration client appears, select the node you are running on, typically your system host name (CHUSA in our example), and expand it (Figure 23-5).

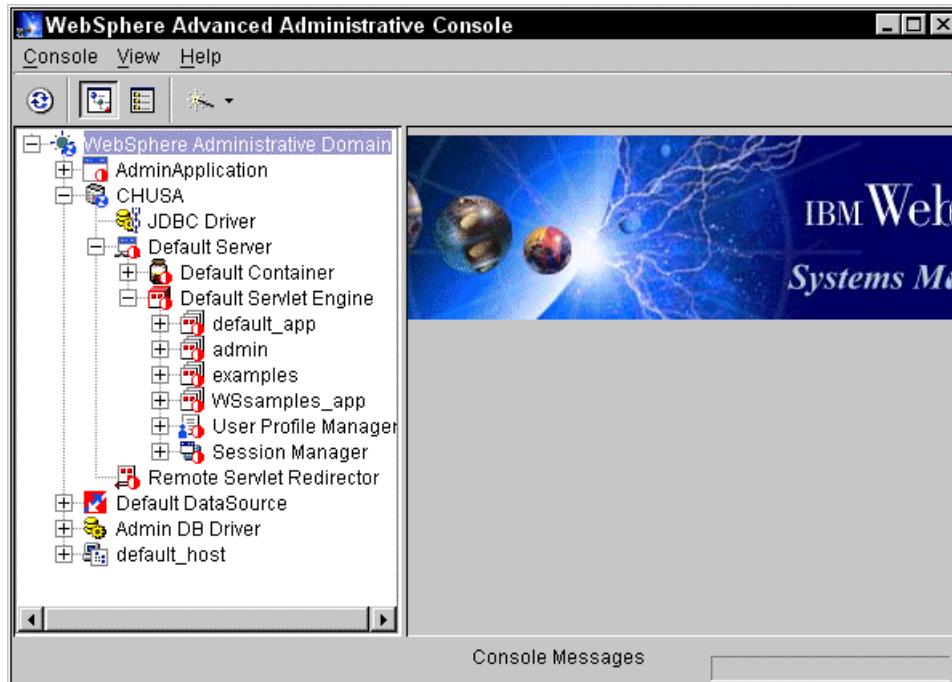


Figure 23-5 WebSphere administrative console

- ▶ A number of items are visible in the console display, such as the AdminApplication, the node, the Default DataSource and Admin DB Driver (they are used to access the WAS database), and the default\_host (it provides the TCP/IP host names and addresses).
- ▶ The node expands into application servers (you always have a Default Server) and a remote servlet redirector (for workload balancing of servlets and JSPs), and under the application server you find an EJB container, and a servlet engine.

## Creating JDBC drivers

JDBC drivers are required to access relational databases from WebSphere. To create a JDBC driver:

- ▶ Select the *WebSphere Administrative Domain* at the root of the tree, and *Create -> JDBC driver* from the context menu. Name it DB2AppDriver, with

COM.ibm.db2.jdbc.app.DB2Driver as the class, jdbc:db2 as the URL prefix, and not JTA enabled (Figure 23-6).

- Repeat this for a second JDBC driver (DB2AppDriverJTA) that is JTA enabled.

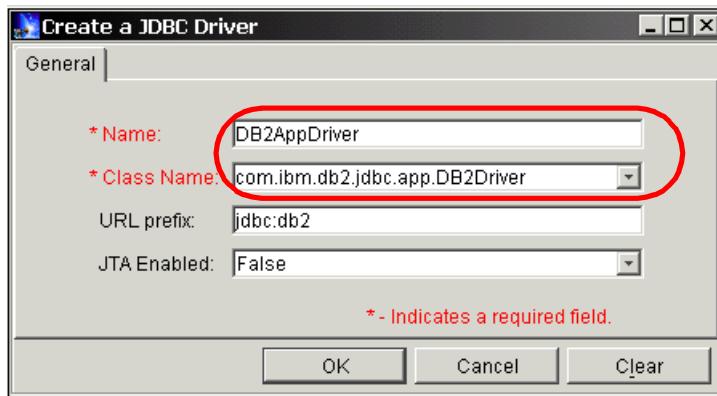


Figure 23-6 Creating a JDBC driver

JDBC drivers must be installed in WebSphere, that is the JAR/ZIP file with the actual driver class must be added:

- Select the DB2AppDriver and *Install*, Select the node (CHUSA) and click on *Browse* to locate the JAR or ZIP file that contains the driver class. For DB2, the zip file is d:\SQLLIB\java\db2java.zip (Figure 23-7).  
Make sure that JDBC 2.0 has been enabled for DB2. See “DB2 Version 7.1 Enterprise Edition” on page 454.
- Repeat the installation for the second JDBC driver (DB2AppDriverJTA). This is required even if the zip file is the same.

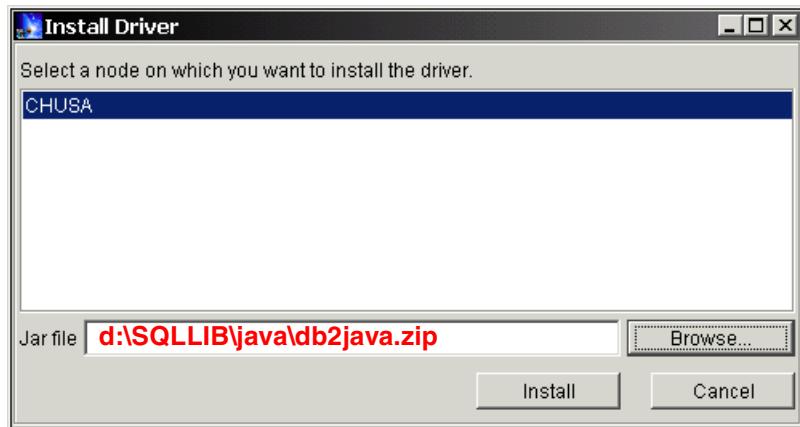


Figure 23-7 Installing a JDBC driver

### Creating data sources

Data sources are used for connection pooling when accessing relational databases from servlets and EJBs. A data source uses a JDBC driver to access the database. To create a data source:

- ▶ Select the WebSphere Administrative Domain at the root of the tree, and select from the context menu *Create* -> *DataSource*. Name the data source EJBBANK, for the EJBBANK database, using the DB2AppDriver (Figure 23-8).
- ▶ Repeat this and create an EJBBANKJTA data source that uses the JDBC driver DB2AppDriverJTA.

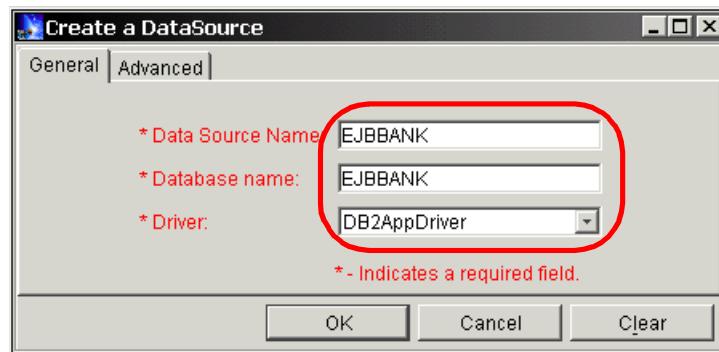


Figure 23-8 Creating a data source

## Creating an application server with a Web application

WebSphere provides multiple ways to create objects. An application server with an EJB container and a servlet engine can be created by performing each task individually, or you can use a wizard that guides you through all the steps at once.

### Create application server wizard

To start the wizard, select *Console -> Tasks -> Create Application Server*. The wizard consists of many pages, we will only show a few important pages in this description.

- ▶ Types of Resources page: Select both resource types, enterprise beans and Web application, so that the get a container and a servlet engine created as well. We will add the enterprise beans later. Click Next.
- ▶ Application Server Properties page: Enter the name of the new application server, for example, ITSO\_EJB35 (Figure 23-9).

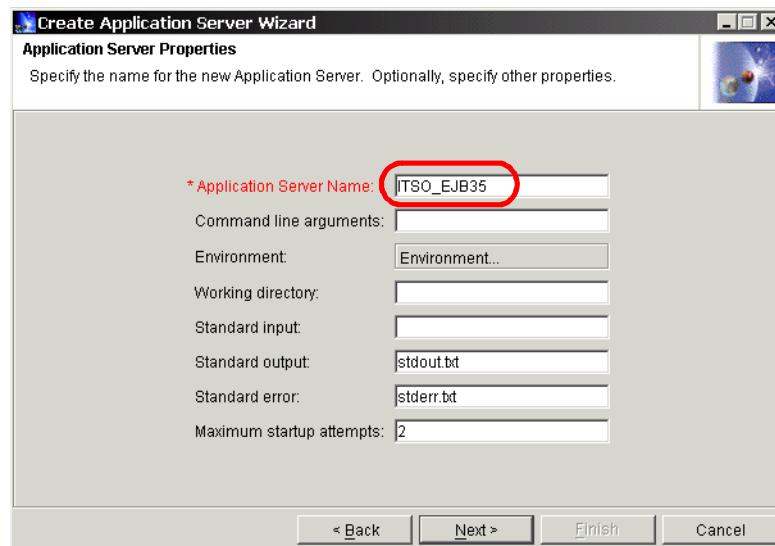


Figure 23-9 Create an application server: name

- ▶ Application Server Start Option page: select *Do not start the server automatically after creating it*.
- ▶ Node Selection page: Select the node of your machine.
- ▶ Add Enterprise Beans page: Leave this page empty for now, we will add the beans later.

- ▶ EJBContainer Properties page: Accept the generated name for the container, for example, ITSO\_EJB35Container. Leave the defaults on Advanced and DataSources tabs.
- ▶ Specify Virtual Host page: Select the default\_host.
- ▶ Servlet Engine Properties page: Accept the generated name for the servlet engine, for example, ITSO\_EJB35ServletEngine (Figure 23-9).

**Attention:** With Fixpack 3 (WebSphere 3.5.3) you have to decide if you want to support Servlet 2.2/JSP1.1 APIs, or keep compatibility with WebSphere 3.5.

Leave the defaults on the Advanced tab.

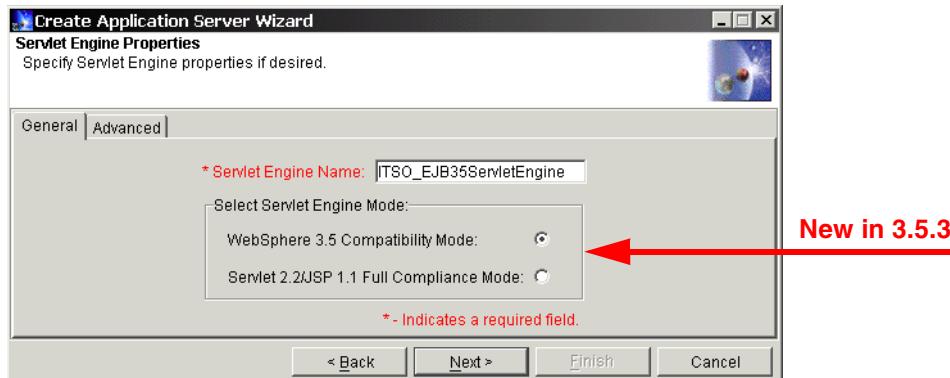


Figure 23-10 Create an application server: servlet engine

The wizard continues for the creation of a Web application. You can also create new Web applications through *Console -> Tasks -> Create a Web application*. After selecting the application server and servlet engine, you will get to the same panels as when continuing with the wizard.

## Creating a Web application

Continuing with the Create Application Server Wizard:

- ▶ Web Application Properties page: Enter the name of the Web application, for example, `itsoejb`. Another important field is the Web application path. The default name would be `/webapp/itsoejb`, but we changed this to `/itsoejb` (Figure 23-11).

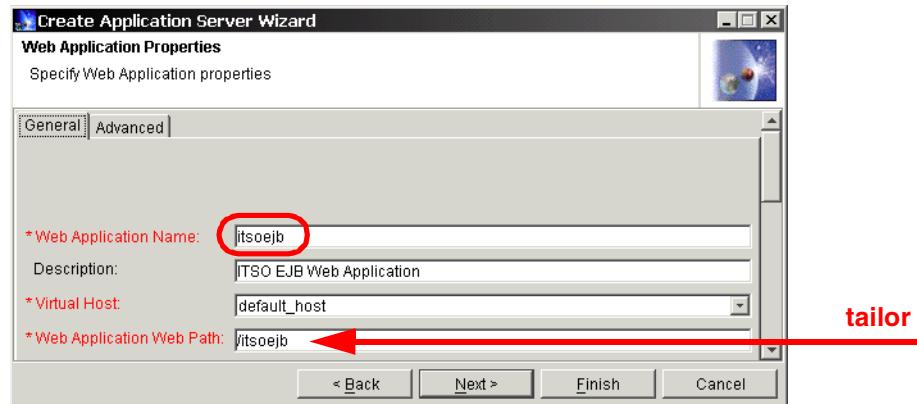


Figure 23-11 Creating a Web application

- ▶ On the Advanced tab, you specify the locations for Web content (HTML and JSP) and servlets. The defaults should be fine (Figure 23-12).

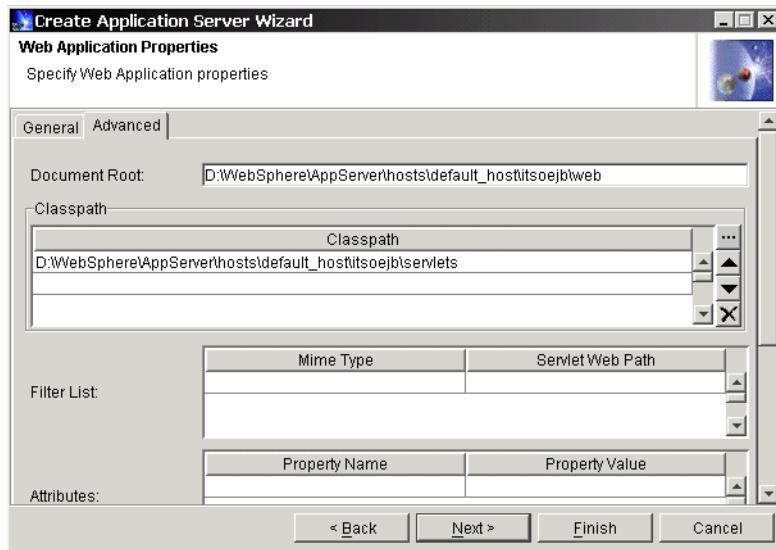


Figure 23-12 Creating a Web application: directories

- Specify System Servlets page: Select *Enable File Servlet* (WebSphere will serve HTML from the Web application directory), *Serve Servlet By Classname* (servlets can be invoked by full class name), and the JSP level (Figure 23-13).

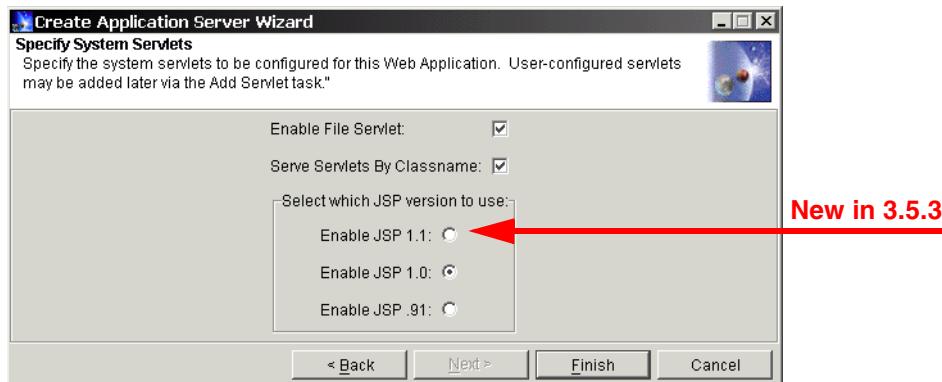


Figure 23-13 Web application special servlets and JSP compiler

- Click *Finish* and the application server with the EJB container and the servlet engine, as well as the Web application are added to the node.

The new application server appears in the Administrator's Console as shown in Figure 23-14.

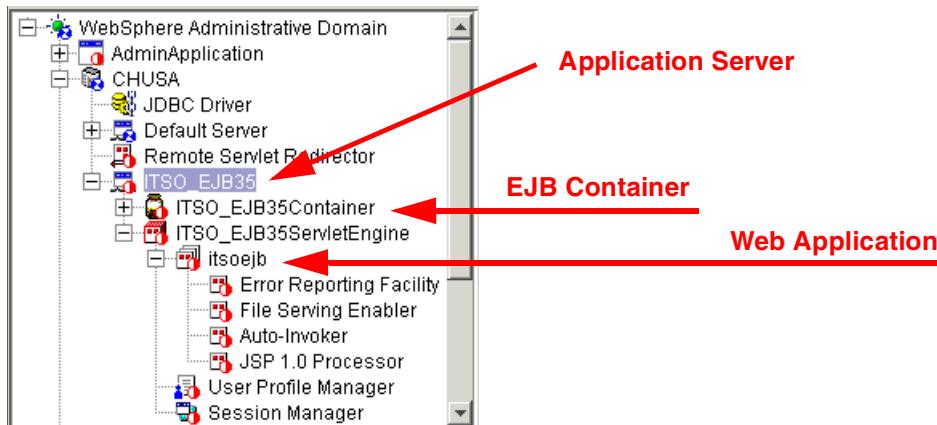


Figure 23-14 New application server

## Create Web application directories

The directories specified for the Web application must be created manually:

```
d:\WebSphere\AppServer\hosts\default_host\itsoejb\web
d:\WebSphere\AppServer\hosts\default_host\itsoejb\servlets
```

To actually run servlets with EJBs, we will add the client JAR file generated by VisualAge for Java to the Web application servlets subdirectory (see “Adding the client JAR file to the class path” on page 290).

## Adding a servlet to the Web application

Servlets can be invoked by their full class name, for example,  
`itso.ejb35.cmp.servlet.CustomerFind` (if the option *Serve Servlet By Classname* was specified for the Web application).

However, in a real execution environment all servlets should be specified to WebSphere with a short alias name. To add a servlet, follow these steps:

- ▶ Select the Web application and *Create -> Servlet* (Figure 23-15). Enter the name, the full class name, and the Web path, that is the short name used to invoke the servlet (click on *Add* to enter the short name `customerFinder`).

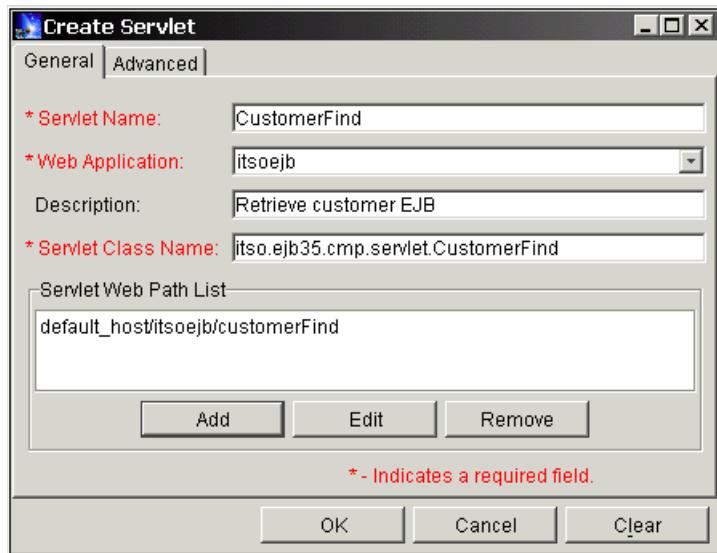


Figure 23-15 Creating a servlet

- ▶ You can add initial parameters on the Advanced tab. Such parameters should be read by the servlet in the init method using coding such as:

```
String parmValue = getServletConfig().getInitParameter("parmname");
```
- ▶ Click *OK* to add the servlet to the Web application.

The CustomerFind servlet can now be invoked in two ways from a browser:

```
http://hostname/itsoejb/servlet/itso.ejb35.cmp.servlet.CustomerFind
http://hostname/itsoejb/customerFind
```

## Start and stop of the application server

You can start and stop the application server (ITS0\_EJB35) from the administrative console using toolbar icons or context menus.

If you shut-down the administration server from the *Services* window while the application server is running, the next time you start the administration server the application server is restarted as well.

For now, stop the application server, the administrative console, and the administration server. We will use it only after we have developed a few enterprise beans and sample applications.

# Setting up VisualAge for Java

Here are the instructions to prepare VisualAge for Java for development of enterprise beans.

## JDBC driver

Start VisualAge for Java and open *Windows -> Options*. Select the Resources page and add the DB2 JDBC driver d:\SQLLIB\java\db2java.zip file to the Workspace class path using *Edit -> Add Jar/Zip*.

Make sure that JDBC 2.0 is enabled for DB2 (see “DB2 Version 7.1 Enterprise Edition” on page 454). If you enable JDBC 2.0 later, you have to stop and restart VisualAge for Java.

## EJB development environment

For EJB development we have to add that feature to the Workbench:

- ▶ In the Workbench projects page, click *File -> Quick Start* or press *F2*. The Quick Start dialog appears.
- ▶ In the left pane of the Quick Start dialog, select *Features*, then in the right pane, select *Add Feature* and click *OK*.
- ▶ The Selection Required dialog appears.
- ▶ Select **IBM EJB Development Environment** and click *OK*. The IBM EJB Development Environment feature is loaded, as well as other required features, such as, IBM WebSphere Test Environment, IBM WebSphere Java Libraries, IBM Enterprise Extension Libraries, IBM XML Parser for Java, Shared Persistence Tools, and Sun Servlet Java API.

## Redbook project

We ship all the examples of this redbook on the redbooks Web site (see “Using the Web material” on page 492). Our project is named **ITSO EJB Redbook**. We suggest that you define this project so that you can load our example packages into the Workbench if you have problems with the directions, or if you want to work with our examples.

Select *Types -> Add Project* and enter ITSO EJB Redbook as the project name. Later you can load selected packages into the project.

## **Product support**

The IBM Internet Web site has information about all the products and the latest Fixpacks:

DB2	<a href="http://www.ibm.com/software/data/">http://www.ibm.com/software/data/</a>
WebSphere	<a href="http://www.ibm.com/software/webservers/appserv/">http://www.ibm.com/software/webservers/appserv/</a>
- Developer Domain	<a href="http://www7b.boulder.ibm.com/wsdd/">http://www7b.boulder.ibm.com/wsdd/</a>
VisuaAge for Java	<a href="http://www.ibm.com/software/ad/vajava/">http://www.ibm.com/software/ad/vajava/</a>
- Developer Domain	<a href="http://www.ibm.com/software/vad/">http://www.ibm.com/software/vad/</a>





B

# Early information: deployment to WebSphere Version 4

In this appendix we describe how to deploy EJBs from VisualAge for Java Version 4 to WebSphere Application Server Version 4.

We used the beta code of WebSphere Application Server Version 4 and an early development driver of VisualAge for Java Version 4 to document how these products may work together in the near future.

**Attention:** This example is not fully EJB 1.1 compliant, but it illustrates that EJB 1.0 compliant enterprise beans can be deployed from VisualAge for Java into WebSphere Version 4.

# VisualAge for Java Version 4

VisualAge for Java Version 4 is not much different from Version 3.5.3. It does include a new utility to deploy EJBs in EJB 1.1 specification format to WebSphere Application Server Version 4.

In this chapter we concentrate on showing the new function and do not explain in detail how the sample enterprise bean is created.

## Account example

To illustrate the new function we developed a new entity group, CMP11\_Entity, with one container-managed entity bean, Account.

### Account entity bean

The Account EJB contains three container-managed fields:

- ▶ accID—account ID, a string, the key field
- ▶ balance—account balance, java.math.BigDecimal
- ▶ accType—account type, a string

The account bean maps to the existing ITSO.ACCTOUNT table, with the columns accid—CHAR(8); balance—DECIMAL(8,2); and acctype—VARCHAR(8).

We used three packages:

- ▶ itso.ejb35.cmp11 for the EJB code
- ▶ itso.ejb35.cmp11.client for a stand-alone client and a servlet
- ▶ itso.ejb35.cmp11.schema for the schema and mapping information

It should be straight forward to create the account entity bean with the schema and mapping to the existing table.

After creating the entity bean, generate the deployed code.

### Stand-alone client

We created one stand-alone client program that retrieves the account home, creates a new account, retrieves an existing account, modifies the new account, and deletes the new account.

Figure B-1 shows the AccountClient program with the main method that does all the processing.

```

package itso.ejb35.cmp11.client;
import itso.ejb35.cmp11.*;
import java.math.BigDecimal;
public class AccountClient {
public static void main(java.lang.String[] args) {
 try {
 javax.naming.InitialContext initialContext =
 new javax.naming.InitialContext();
 Object objHome = initialContext.lookup("itso/ejb35/cmp11/Account");
 AccountHome accountHome = (AccountHome)
 javax.rmi.PortableRemoteObject.narrow(objHome,AccountHome.class);

 Account acc1 = accountHome.create
 ("111-1111", new BigDecimal("111.11"),"Checking");
 System.out.println("Account 111-1111 created: balance="
 +acc1.getBalance()+" Type="+acc1.getAccType());
 Account acc2 = accountHome.findByPrimaryKey(new AccountKey("101-1001"));
 System.out.println("Account 101-1001: balance="
 +acc2.getBalance()+" Type="+acc2.getAccType());
 acc1.setBalance(new BigDecimal("222.22"));
 System.out.println("Account 111-1111 changed: balance="
 +acc1.getBalance()+" Type="+acc1.getAccType());
 acc1.remove();
 System.out.println("Account 111-1111 removed");
 } catch(Exception ex) {
 ex.printStackTrace();
 }
}
}

```

*Figure B-1 EJB 1.1 example: client program*

## Servlet

We also created a servlet/JSP example that gets an account ID from an HTML file, retrieves the account home and the given account, and calls a JSP to display the account details.

Figure B-2 shows the AccountDetails servlet with the performTask method that does all the processing.

```

package itso.ejb35.cmp11.client;
import itso.ejb35.cmp11.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.math.BigDecimal;
public class AccountDetails extends HttpServlet {
 public void doGet(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException {
 performTask(request, response);
 }
 public void doPost(.....) {.....}

 public void performTask(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException {
 // Read the input parameter from the HTML Form
 String id = request.getParameter("accountID");
 // Set the results page URL
 String url = "/ejb/cmp11/AccountDetails.jsp";
 try {
 // Get the initial naming context
 javax.naming.InitialContext initialContext =
 new javax.naming.InitialContext();
 // Obtain the EJBHome for Account
 Object objHome = initialContext.lookup("itso/ejb35/cmp11/Account");
 AccountHome accountHome = (AccountHome)
 javax.rmi.PortableRemoteObject.narrow(objHome, AccountHome.class);

 // Find the account
 Account account = accountHome.findByPrimaryKey(new AccountKey(id));

 // Forward to the results JSP
 request.setAttribute("account", account);
 getServletConfig().getServletContext().getRequestDispatcher(url).
 forward(request, response);
 }
 catch (Exception e)
 {
 System.out.println("Exception thrown for account with id:" + id);
 e.printStackTrace();
 }
 }
}

```

*Figure B-2 EJB 1.1 example: servlet*

Figure B-3 shows the HTML file and the result JSP. Notice that we call the servlet using a short alias name within a Web application.

```
HTML file
<HTML>
 <HEAD><TITLE>Account Details</TITLE></HEAD>
 <BODY>
 <H2>Account Details (WebSphere)</H2>
 <FORM METHOD="post" ACTION="/webapp/itsoejb/accountDetails">
 <P>Please complete the form</P>
 <P>Account ID:
 <INPUT TYPE="text" NAME="accountID" ID="accountID" SIZE="8"
 MAXLENGTH="8">
 <P> <INPUT TYPE="submit" NAME="Submit" ID="Submit" VALUE="Submit">
 <INPUT TYPE="reset" NAME="Reset" ID="Reset" VALUE="Reset">
 </FORM>
 </BODY>
</HTML>

=====
JSP file
<HTML>
 <HEAD><TITLE>Account Details Output</TITLE></HEAD>
 <BODY>
 <H2>Account Details Output</H2>
 <jsp:useBean id="account" type="itso.ejb35.getCmp11.Account"
 scope="request"/>
 <H3>Account:
 <%= ((itso.ejb35.getCmp11.AccountKey)account.getPrimaryKey()).accID %></H3>
 <TABLE border="0" cellpadding="4">
 <TR>
 <TD>Balance:</TD> <TD><%= account.getBalance() %></TD>
 </TR><TR>
 <TD>Type:</TD> <TD><%= account.getAccType() %></TD>
 </TR>
 </TABLE>
 </BODY>
</HTML>
```

Figure B-3 EJB 1.1 example: HTML and JSP

## Generating the deployment JAR files

From VisualAge for Java we generate the EJB 1.1 JAR file and the client JAR file.

### EJB 1.1 JAR file

Generating an EJB 1.1 JAR file is a new function of VisualAge for Java Version 4. Select the EJB group in the Workbench and *Export -> EJB 1.1 JAR*.

The export dialog opens (Figure B-4).

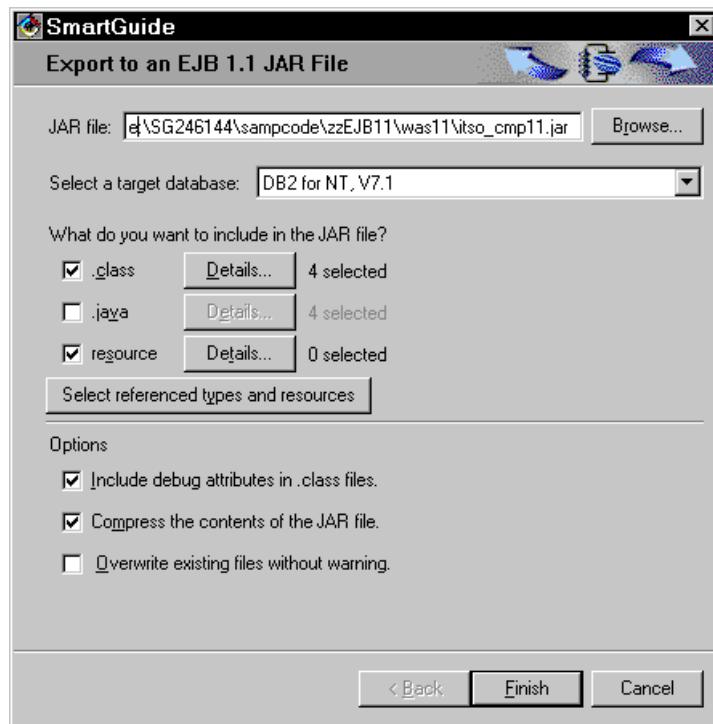


Figure B-4 Export to an EJB 1.1 JAR file

Enter a target directory and JAR file name (itso\_cmp11.jar) and select the target database (DB2 on different platforms, Oracle, Informix, Sybase, SQL Server). The account EJB does not use any utility classes or resource files so there is no requirement to click on *Select referenced types and resources*.

### Client JAR file

Select the EJB group in the Workbench and *Export -> Client JAR*. For our example we name the client JAR file itso\_client11.jar.

## EJB 1.1 JAR file

An EJB 1.1 JAR file is not the same as a deployed JAR file, which we generated in VisualAge for Java for deployment to WebSphere Version 3.5.

The EJB 1.1 JAR file does not contain deployed code. It contains the bean and interface classes, and XMI files with the schema and mapping information. From this meta information, WebSphere Version 4 generates the deployed code.

## WebSphere Application Server Version 4

WebSphere Advanced Edition Version 4 comes as full version (similar to Version 3.5) and as single server version, which is very practical for an individual developer for testing.

The single server version does not require a database for the configuration information. It also does not provide the Administrative Console; configuration is performed using a Web client or command line tools.

## Setup

To test the entity bean, servlet, and client application, we setup WebSphere with a JDBC driver (DB2AppDriver) and a data source (EJBBANK) to mirror the deployment we performed for Version 3.5.

## Application Assembly Tool

WebSphere Version 4 provides a J2EE compliant tool for assembling components into enterprise applications.

An enterprise application can contains these components:

- ▶ **EJB module**—a set of enterprise beans, stored in a deployed JAR file
- ▶ **Web module**—a set of servlets, JSPs, and HTML files (similar to a Web application in Version 3.5), stored in a WAR file (Web archive)
- ▶ **Client module**—a client program that uses J2EE functions, such as EJBs, stored in a JAR file

An enterprise application is stored in an EAR file (enterprise archive)

The Application Assembly Tool is started from the Administrative Console or using the assembly.bat file in the BIN directory of WebSphere.

A welcome panel is displayed when the tool is started (Figure B-5). A function can be selected from the Welcome panel or from the *File* menu.

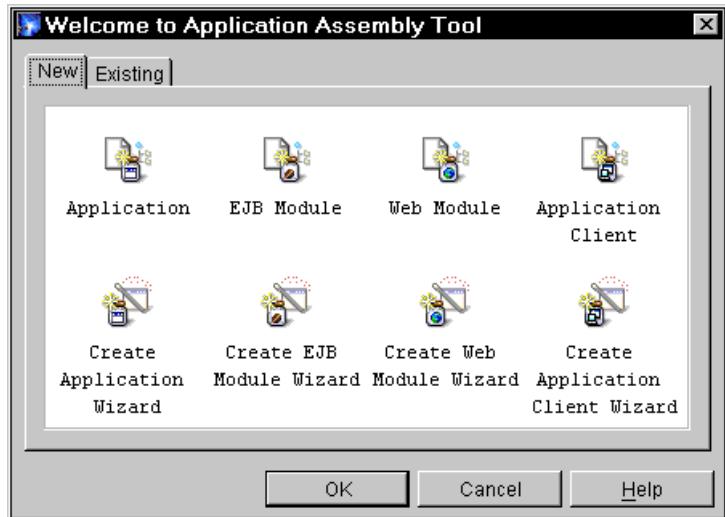


Figure B-5 Application Assembly Tool: Welcome

There are two ways to create components:

- ▶ Component dialogs, where you can specify the parts in any order using property panels and selection dialogs (top row)
- ▶ Wizards, where you are guided through a set of panels and you end up with the component dialog, where you can modify anything you specified (bottom row)

To deploy the sample EJB (Account entity bean), the Web application (Servlet, HTML, and JSP), and the stand-alone client, we will build individual modules and then assemble the modules into two EAR files, one for the EJB and Web application and one for the stand-alone client.

Note that we could package all modules into one EAR file, or each module into its own EAR file.

For illustration purposes we will build the EJB module using a Wizard, and the other two modules using the component dialogs. We will only enter required information and skip many of the possible specifications.

## Create an enterprise application

Select *File -> New -> Application* in the Application Assembly Tool. Enter the display name as ITSO Account Application One and click *Apply* (Figure B-6).

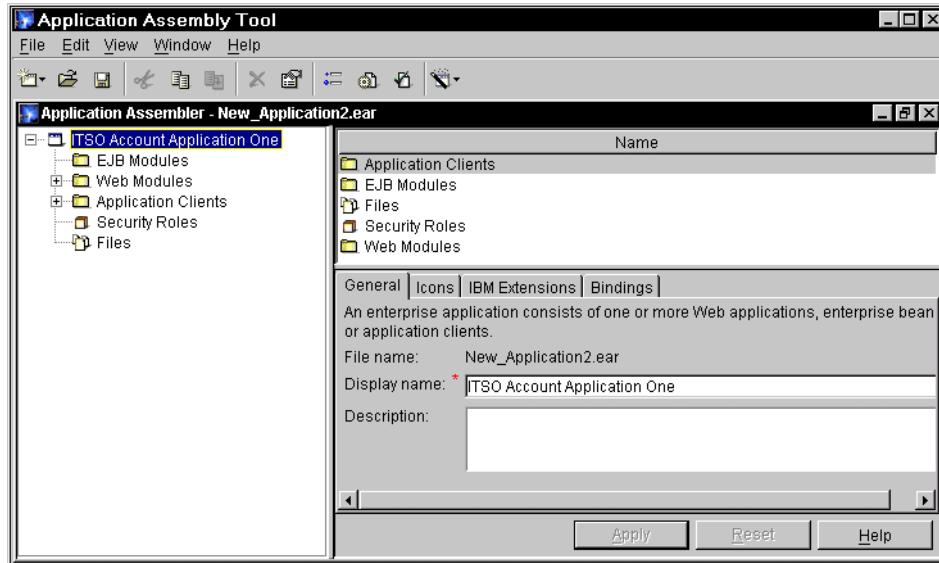


Figure B-6 Creating an enterprise application

## Create and add an EJB module

You can create new EJB modules, where WebSphere creates the deployment information, or you can import an EJB 1.1 JAR from VisualAge for Java with all the deployment information (schema and mapping).

Select *EJB Modules -> Import*. In the Open dialog navigate to the EJB 1.1 JAR file generated by VisualAge for Java (itso\_cmp11.jar) and click *Open*. You are prompted for the file name of the resulting EJB module, with the default being the name of the EJB 1.1 JAR file (Figure B-7).



Figure B-7 Specify the JAR file name of the EJB module

Click *OK*. The EJB module is named **CMP11\_Entity** (the EJB group name) and the Account bean is added as an entity bean. Select the **CMP11\_Entity** module and on the Bindings page enter **jdbc/EJBBANK** as the JNDI name for the data source. Click *Apply* (Figure B-8).

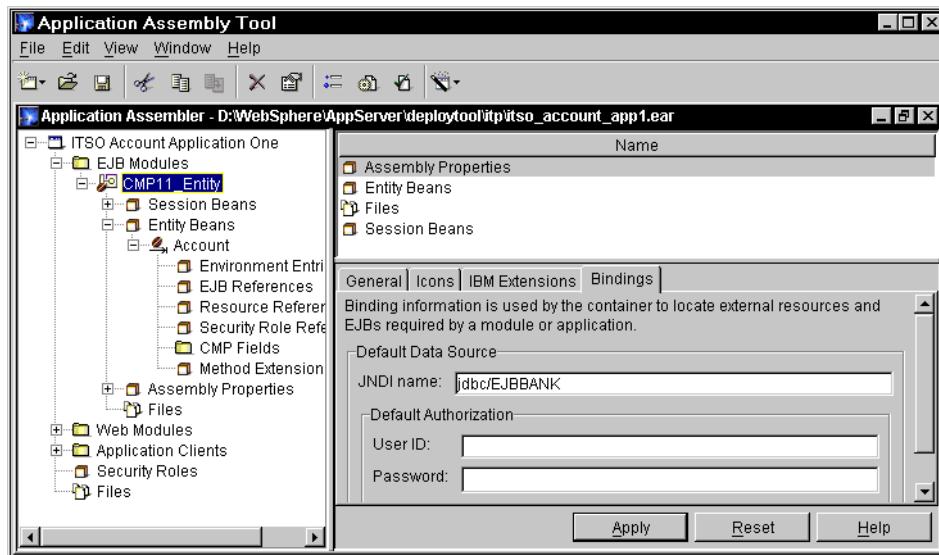


Figure B-8 Creating an EJB module

Select *File -> Save As* and enter **itso\_account\_app1.ear** as the file name for the enterprise application. Use this directory for all files of the assembly tool:

```
d:\WebSphere\AppServer\deploytool\itp
```

## Create and add a Web module

We can add a Web module directly to the enterprise application, or we can use a Wizard to create and add a Web module (this did not work in the beta code).

Select *Web Modules -> New* to start the dialog:

- ▶ In the New Web Module dialog (Figure B-9) enter the file name (**itso\_account\_servlet.war**), the context root (**/webapp/itsoejb**), the class path (**itso\_cmp11.jar**), and the display name of the module (**ITSO Account Servlet**).
- ▶ Click *Apply*.
- ▶ On the Bindings page enter **default\_host** as the virtual host name. Click *OK*.

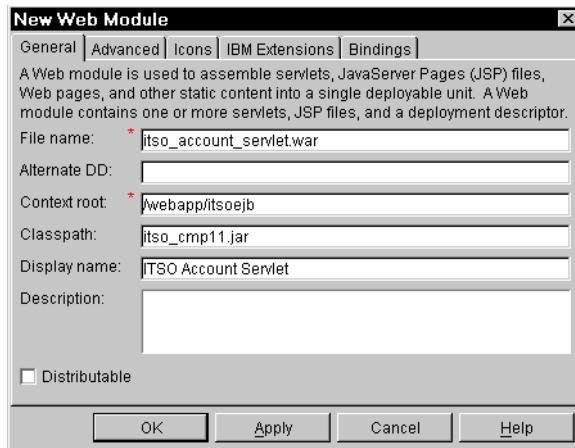


Figure B-9 Creating a Web module

The new Web module is added to the enterprise application. Now we add the HTML, JSP, and servlet to the Web module:

- ▶ Expand the module, expand *Files*, select *Resource Files* -> *Add Files* (Figure B-10).

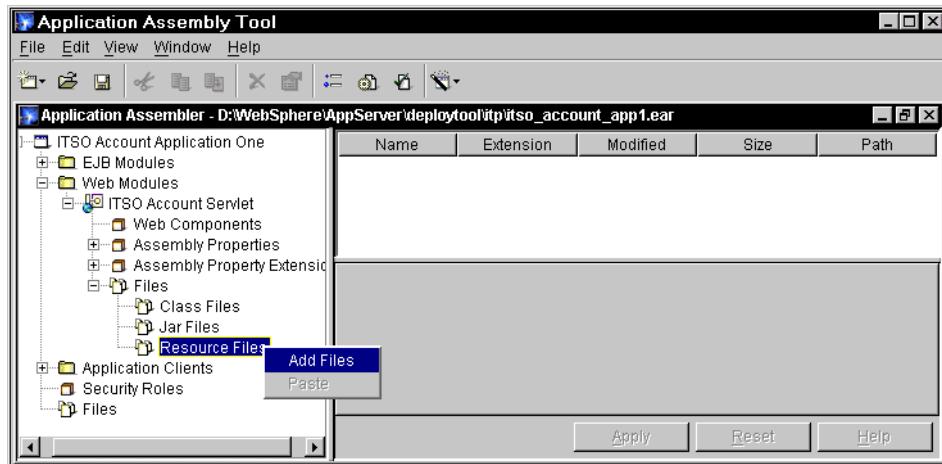


Figure B-10 Adding files to a Web module

- ▶ The Adding Files dialog opens. Click *Browse* and navigate to the directory where the HTML and JSP files are located. Click *Select*. The directory is shown in the Adding Files dialog (Figure B-11). Expand subdirectories and select the HTML and JSP files and click *Add*.

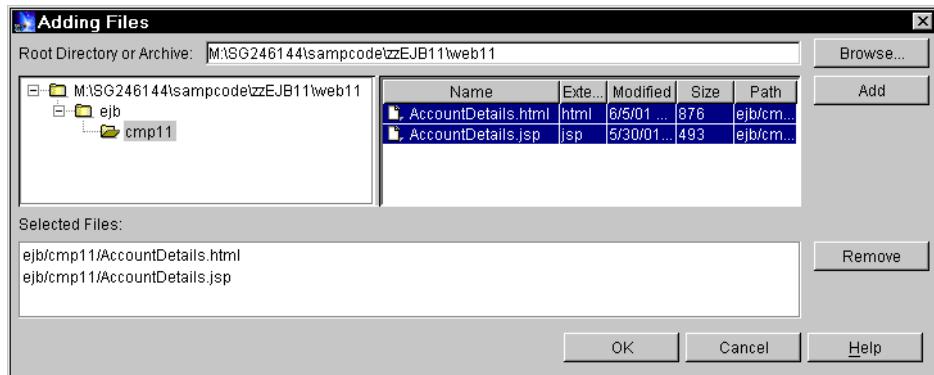


Figure B-11 Adding HTML and JSP files to a Web module

**Important:** We call the HTML file as **/ejb/cmp11/AccountDetails.html** and the JSP as **/ejb/cmp11/AccountDetails.jsp** (see Figure B-2 on page 472). Therefore, only browse to the directory that contains the ejb subdirectory.

- ▶ Select *Files* -> *Class Files* -> *Add Files*. In the Adding Files dialog click *Browse* and navigate to the directory where the servlet file is located. Click *Select*. The directory is shown in the Adding Files dialog (Figure B-12). Expand subdirectories and select the servlet file and click *Add*. Notice that the servlet is called as `itso.ejb35.cmp11.client.AccountDetails` and therefore the containing subdirectory is selected.

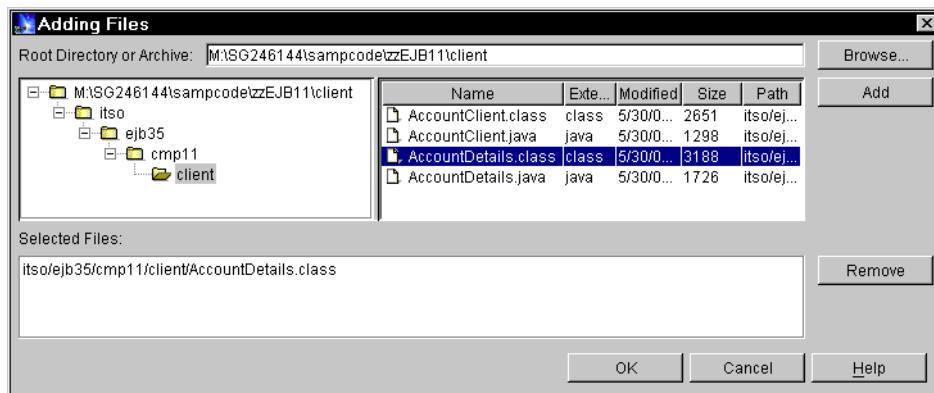


Figure B-12 Adding servlet class files to a Web module

Select *Web Components* -> *New* to define the servlet and optionally the JSP:

- ▶ In the New Web Component panel (Figure B-13) enter a component name (AccountDetailsServlet), a display name (Account Details Servlet), and select the servlet class name using *Browse*. Click *OK*.

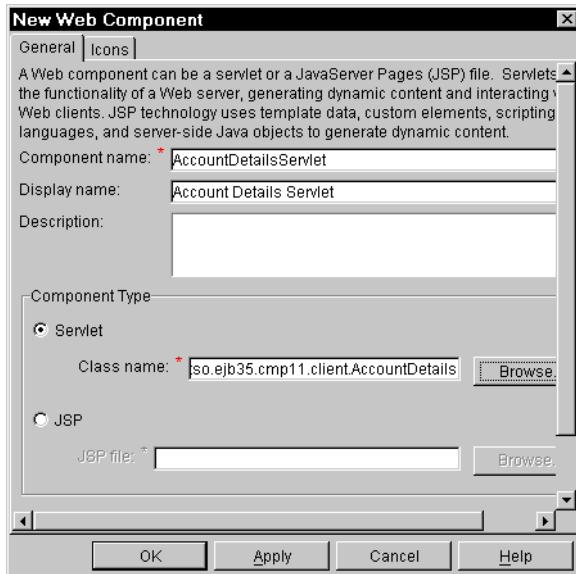


Figure B-13 Defining Web components in a Web module

- ▶ Perform the same step for the Account Details JSP, but this time select the AccountDetails.jsp file. Note that adding JSPs is optional.

Expand *Assembly Properties*, select *Servlet Mapping* -> *New*:

- ▶ On the New Servlet Mapping panel enter accountDetails as the URL pattern and AccountDetailsServlet as the servlet (Figure B-14). Click *OK*.



Figure B-14 Defining a servlet alias name

This completes the definition of the Web module (Figure B-15). The other specifications are not required.

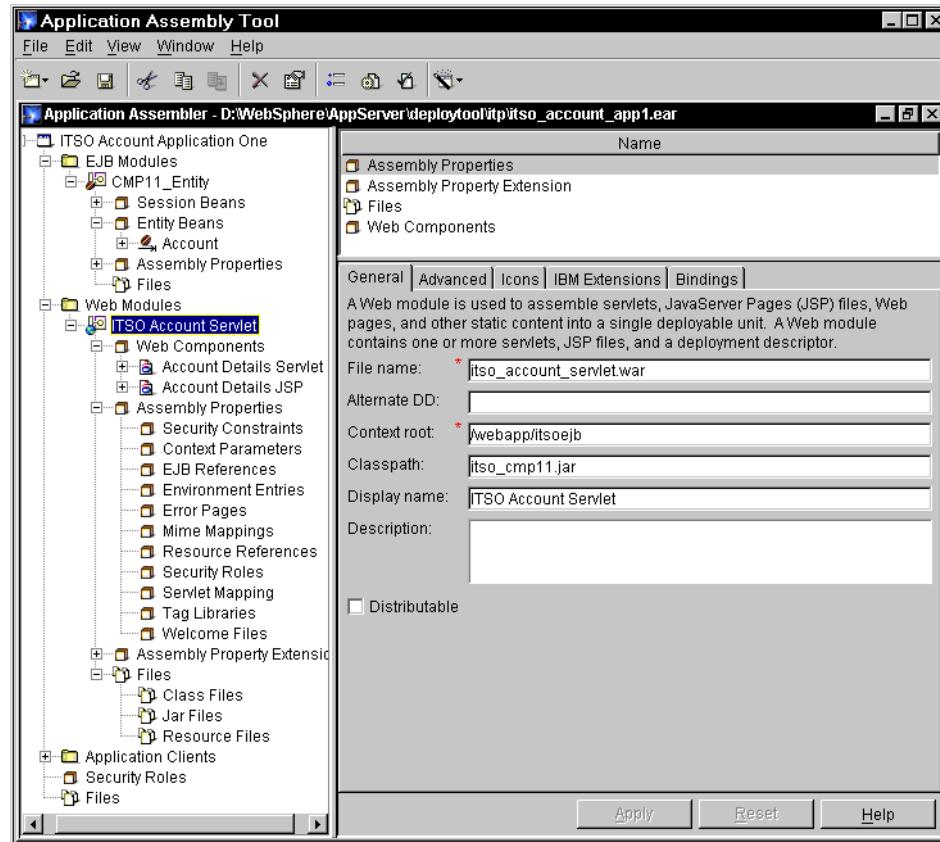


Figure B-15 Completed Web module

Save the EAR file (*File -> Save*).

## Install enterprise application

To run the EJBs in a container and the Web application (servlet and JSP) in a servlet engine the enterprise application must be installed.

This can be done in the Administrative Console (*Console -> Wizards -> Install Enterprise Application*) for the Advanced Edition or using a command (SEAppInstall) for the Single Server Edition:

```
SEAppInstall -install itso_account_app1.ear
```

## Application installation wizard

The wizards presents a series of panels for installation of an enterprise application:

- ▶ In the Specifying the Application or Module panel, click on *Browse* to locate the EAR file (Figure B-16).

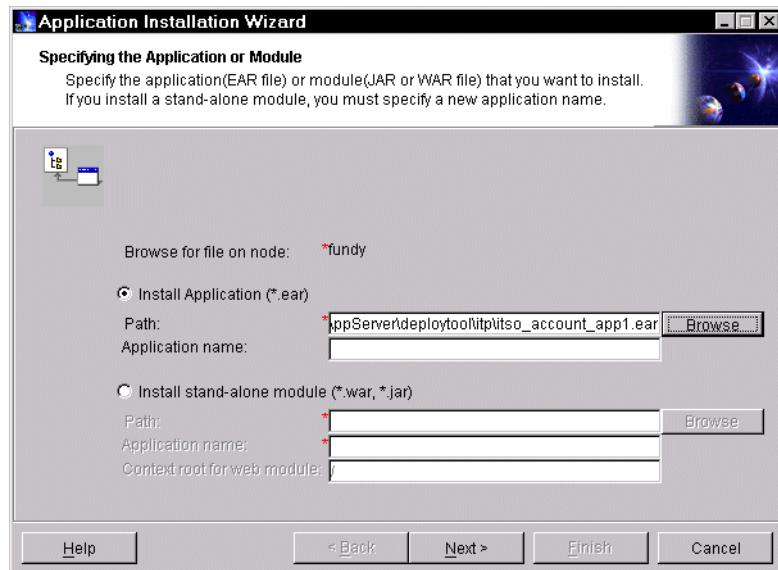


Figure B-16 Installing an enterprise application

- ▶ Skip the Specify Role Mappings for Bean Methods panel.
- ▶ Skip the Mapping Users to Roles panel.
- ▶ Skip the Mapping EJB RunAs Roles to Users panel.
- ▶ Skip the Mapping Resource References to Resources panel.
- ▶ In the Bind EJB to JNDI Name panel you can see the Account EJB being mapped to the `itso/ejb35/cmp11/Account` JNDI name. The JNDI name could be tailored.
- ▶ Skip the Mapping EJB References to EJB panel.
- ▶ In the Specifying the Default Datasource for EJBModule panel (Figure B-17), click on *Select a Datasource* and select the `EJBBANK` data source for the `CMP11_Entity` module (it should be already filled with `EJBBANK` because we defined it for the EJB module).

EJB Module	JNDI Name	Select a Datasource
CMP11_Entity	jdbc/EJBBANK	

Figure B-17 Selecting the data source for an EJB module

- ▶ Skip the Specifying Datasource for individual CMP Beans panel (we could overwrite the data source here).
- ▶ In the Selecting the Virtual Hosts for WAR modules panel, select the default\_host virtual host (it should be already filled because we defined it for the Web module).
- ▶ In the Selecting the Application Server panel (Figure B-18), select the application server for the CMP11\_Entity EJB module and for the ITSO Account Servlet Web module. We defined an application server named ITSO\_EJB40.

Module	Application Server	Select Server
CMP11_Entity	ITSO_EJB40	
ITSO Account Servlet	ITSO_EJB40	

Figure B-18 Selecting the application server

- ▶ In the Completing the Application Installation Wizard click *Finish*. Click OK in the confirmation dialog.

Installation of an enterprise application takes a while, be patient. The application shows up in the Administrative Console (Figure B-19).

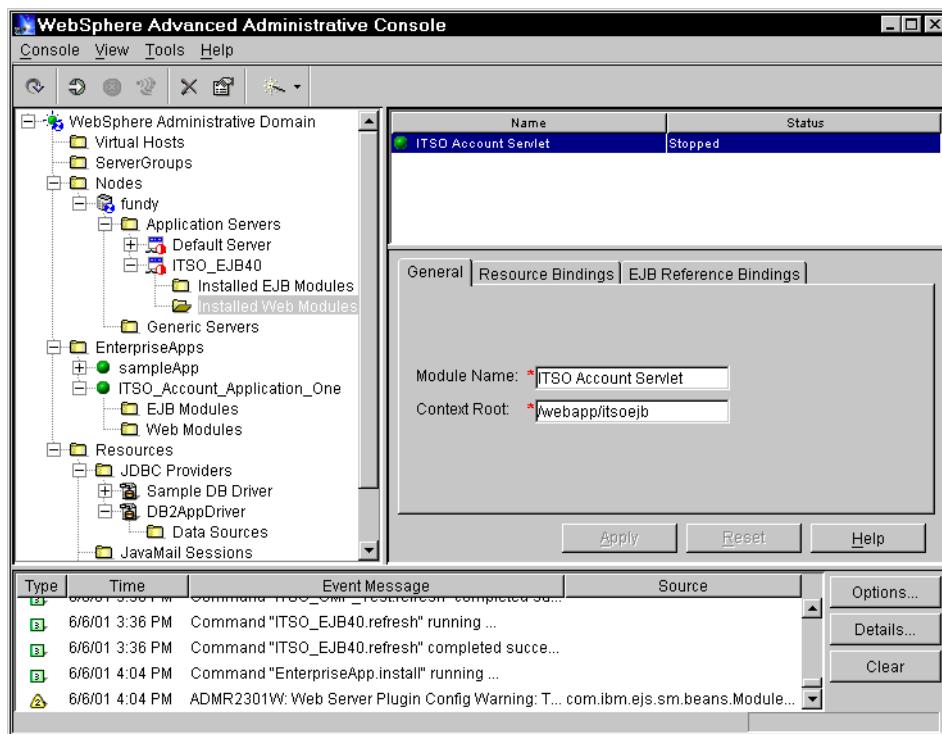


Figure B-19 Administrative Console with installed enterprise application

Installed applications appear in the directory `installedApps` directory (Figure B-20):

`d:\WebSphere\AppServer\installedApps`

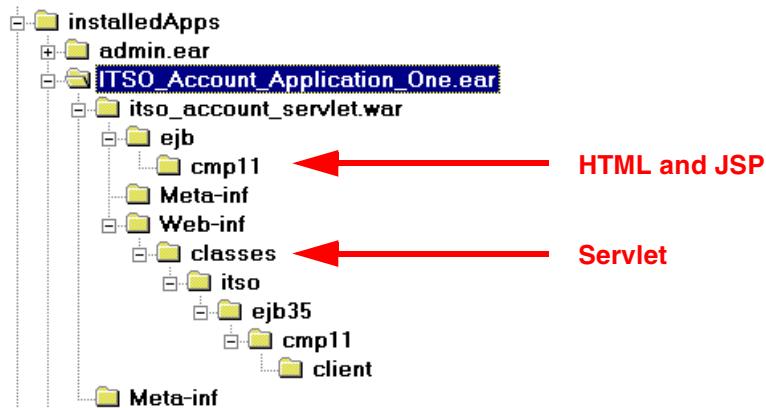


Figure B-20 Directory with installed enterprise applications

## Create client application

Select *File -> New -> Application* in the Application Assembly Tool. Enter the display name as **ITSO Account Application Two** and click *Apply*.

Select *Application Clients -> New* (Figure B-21):

- ▶ In the New Application Client dialog enter **itso\_account\_client.jar** as file name, **ITSO Account Client** as display name, and **itso\_cmp11.jar** as class path (we require access to the Account EJB).
- ▶ Click *Browse* to locate the client application class (locate the directory where the package name starts). Click *OK*.

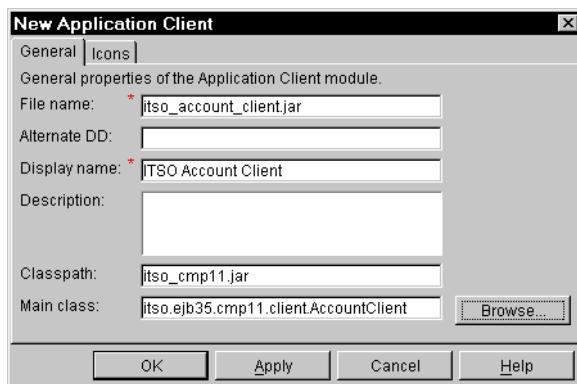


Figure B-21 Creating a client application

We have to add the JAR file with the Account EJB to the EAR file. This must be the deployed JAR file with all the stub classes to communicate with the EJB.

Select *Files -> Add Files* (Figure B-22) and locate the **itso\_cmp11.jar** file in:

d:\WebSphere\AppServer\installedApps\ITSO\_Account\_Application\_One.ear

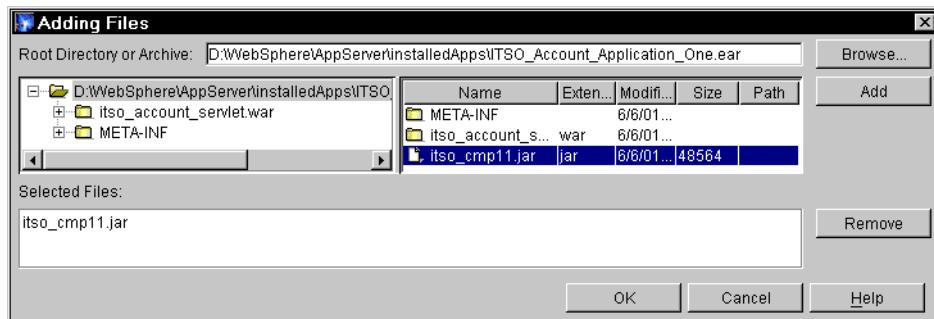


Figure B-22 Adding the EJB Jar file to the client application

Save the client application as `itso_account_app2.ear` (Figure B-23).

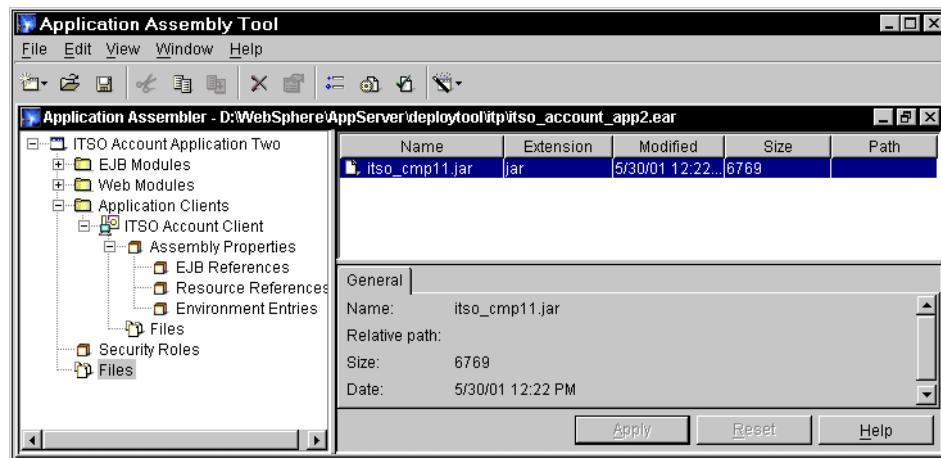


Figure B-23 Completed client application

## Running the account sample applications

To run the applications start the application server in the Administrative Console.

### Running the servlet

Open a browser for the HTML file to invoke the servlet (Figure B-24):

`http://localhost:9080/webapp/itsoejb/ejb/cmp11/AccountDetails.html`

<b>Account Details (WebSphere)</b>  Please complete the form  Account ID: <input type="text" value="101-1001"/>  <input type="button" value="Submit"/> <input type="button" value="Reset"/>	<b>Account Details Output</b>  <b>Account:</b> 101-1001  <b>Balance:</b> 300.00 <b>Type:</b> CHECKING
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------

Figure B-24 Running the servlet

Note that WebSphere Version 4 provides an HTPP server on port 9080 for testing without having a real HTTP server started.

## Running the client application

The client application runs in a client container provided by WebSphere. To start the application use the launchclient.bat file:

```
launchclient itso_account_app2.ear
```

The output of a sample run is shown in Figure B-25.

```
D:\WebSphere\AppServer\deploytool\itp>launchclient itso_account_app2.ear
IBM WebSphere Application Server, Release 4.0
J2EE Application Client Tool, Version 1.0
Copyright IBM Corp., 1997-2001

WSCL0012I: Processing command line arguments.
WSCL0013I: Initializing the J2EE Application Client Environment.
WSCL0035I: Initialization of the J2EE Application Client Environment has
completed.
WSCL0014I: Invoking the Application Client class
 itso.ejb35.cmp11.client.AccountClient
Account 111-1111 created: balance=111.11 Type=Checking
Account 101-1001: balance=300.00 Type=CHECKING
Account 111-1111 changed: balance=222.22 Type=Checking
Account 111-1111 removed
```

Figure B-25 Running the client in a WebSphere container

## Running the client application without a WebSphere container

It is possible to run the stand-alone client using the JDK. But, you have to get the class path exactly right:

```
rem client JAR file generated by VisualAge for Java
set classpath=itso_client11.jar;%CLASSPATH%
set classpath=d:\WebSphere\AppServer\lib\ivjejb35.jar;%CLASSPATH%
set classpath=d:\WebSphere\AppServer\lib\ujc.jar;%CLASSPATH%
set classpath=d:\WebSphere\AppServer\lib\ns.jar;%CLASSPATH%
set classpath=d:\WebSphere\AppServer\lib\jts.jar;%CLASSPATH%
set classpath=d:\WebSphere\AppServer\lib\csicpi.jar;%CLASSPATH%

java
-Djava.naming.factory.initial="com.ibm.websphere.naming.WsnInitialContextFactory"
 itso.ejb35.cmp11.client.AccountClient

java itso.ejb35.cmp11.client.AccountClient
```

Both invocations of the client class file work.

## Problems

The sample applications did not run at first. The deployment in WebSphere omitted the table prefix (ITSO.ACOUNT) and accessed the table as ACCOUNT without the prefix. To solve this problem we defined a table alias:

```
db2 create alias account for itso.account
```

This is most probably an error in the beta code of WebSphere, because the deployment descriptor file, schema.dbxmi, in the itso\_cmp11.jar file does include the ITSO prefix.

## Deployment of a regular EJB JAR file

VisualAge for Java can also create a non-deployed JAR file using *Export -> EJB JAR* for the EJB group.

Such a JAR file can be used in WebSphere Version 4 and all the deployed code, schema, and mapping are generated. Deployment of an EJB JAR file can be performed using the Application Assembly Tool, or by using the EJBDeploy command line tool. The Application Assembly Tool uses EJBDeploy under the covers.

### EJBDeploy command line tool

The EJBDeploy command line syntax is:

```
ejbdeploy inputJar workingDirectory outputJar [options]
```

The options are explained when typing ejbdeploy without parameters.

To illustrate the EJBDeploy tool we generated a regular EJB JAR file (itso\_cmp11\_n.jar) and deployed it with this command:

```
ejbdeploy itso_cmp11_n.jar temp itso_cmp11_n_deployed.jar
```

```
Starting workbench.
Registering package: mapping.xmi->com.ibm.etools.emf.mapping.impl.
 MappingPackageImpl
Creating the project.
Creating Top-Down Map...
Generating deployment code
[*Error] and [*Warning] messages
Building: EJBDeploymentSolution.
Invoking RMIC.
Building: EJBDeploymentSolution.
Shutting down workbench.
EJBDeploy complete.
1 Errors, 4 Warnings, 0 Informational Messages
```

The tool reports errors and warnings in regard to violation of the EJB 1.1 specifications.

### ***Errors***

- ▶ AccountBean must not declare the `finalize` method (this must be generated by VisualAge for Java; it is not in the Java source)
- ▶ AccountBean `ejbCreate` should return the primary key type instead of `void` (to eliminate this problem change the method signature and return `null`—the returned value is discarded by the container)

### ***Warnings***

- ▶ AccountBean `ejbCreate` and `ejbPostCreate` should throw `EJBException` instead of `RemoteException` (this is allowed in EJB 1.1, but deprecated)
- ▶ There must be an `ejbPostCreate` method for every `ejbCreate` method (add one if it is missing)

## **A word of caution**

**Attention:** This example was created with early driver code of VisualAge for Java Version 4 and with the beta code of WebSphere Version 4. Screen shots as well as functionality of the tools may be different in the final products.



C

# Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

## Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG246144/>

Alternatively, you can go to the IBM Redbooks Web site at:

[ibm.com/redbooks](http://ibm.com/redbooks)

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG24-6144.

# Using the Web material

The additional Web material that accompanies this redbook includes the following:

- ▶ `readme.txt`—short instructions
- ▶ `sg246144code.zip`—all sample code in ZIP format

## System requirements for downloading the Web material

The following system configuration is recommended for installing VisualAge for Java and WebSphere Application Server to work with the additional Web material:

**Hard disk space:** 3 GB  
**Operating System:** Windows NT, 2000, or 98  
**Processor:** 400 MHz or better  
**Memory:** 385 MB, recommended 512 MB

## How to use the Web material

Create a subdirectory (folder) on your workstation and download the `sg246144code.zip` file into this folder. Unzip the file onto a hard drive to create this directory structure and files:

```
sg246144
 sampcode
 ejbbank
 ejbclient
 itso\ejb35\xxxx
 repository
 rose
 was
 web
 zsource
 zzejb11
```

- master directory  
- DDL and SQL to load the EJBBANK database  
- exported client code and client JAR file  
- sample client applications  
- exported VisualAge for Java repository file  
- model from Rational Rose  
- WebSphere (ejbjar, xmlconfig, servlets)  
- HTML and JSP files  
- exported Java source files  
- deployment example to WebSphere Version 4

The **`ejbbank`** subdirectory contains the code to define and load the EJBBANK database in DB2.

The **`ejbclient`** subdirectory contains the client JAR file and the sample application code to run the sample applications against WebSphere.

The **`repository`** subdirectory contains an exported VisualAge for Java repository file. A subdirectory (`itsoejb35.dat.pr`) contains the project resource files. See “VisualAge for Java project” on page 493 for detailed content.

The **`rose`** subdirectory contains the bank model created in Rational Rose.

The **was** subdirectory contains three subdirectories: **ejbjar** with the exported JAR files from VisualAge for Java (can be used to install the EJBs into a WebSphere application server and EJB containers); **servlets** with exported class files of the sample servlets; **xmlconfig** with sample XML files to define an EJB container and install the bank EJBs into the container.

The **web** subdirectory contains the HTML and JSP files to run the servlets.

The **zsource** subdirectory contains all the Java source files for browsing.

The **zzejb11** subdirectory contains three subdirectories: **client** with a sample application and a sample servlet; **web11** with HTML and JSP files for the sample servlet; **was11** with exported and deployed files; **repository** with this code exported from VisualAge for Java.

## VisualAge for Java project

If you want to load all the sample solutions into VisualAge for Java import the **itsoejb35.dat** file into the repository. Then load the **ITSO EJB Redbook** project into the workspace. You will see a number of errors because some samples use features that you have not loaded into your workspace.

Here is a short overview of the packages included in the project:

itso.ejb35.util	Utility classes (CustomerAddress, HomeHelper, InsufficientFundException).
CMP_EntityEJBReserved	<b>CMP_Entity</b> group (reserved package)
itso.ejb35.cmp	CMP types
itso.ejb35.cmp.schema	CMP schema and mapping
itso.ejb35.cmp.client	CMP client applications
itso.ejb35.cmp.servlet	CMP servlet application
itso.ejb35.gui	CMP GUI application
BMP_EntityEJBReserved	<b>BMP_Entity</b> group (reserved package)
itso.ejb35.bmp	BMP types
itso.ejb35.bmp.schema	BMP schema and mapping
itso.ejb35.bmp.client	BMP client application
Stateless_SessionEJBReserved	<b>Stateless_Session</b> group (reserved package)
itso.ejb35.session	Session types
Bank_EntitiesEJBReserved	<b>Bank_Entities</b> group (reserved package)
itso.ejb35.bank	Bank application types
itso.ejb35.bank.schema	Bank schema and mapping
itso.ejb35.bank.client	Bank client applications
itso.ejb35.bank.servlet	Bank servlet application

EAB_SessionEJBRReserved itso.ejb35.eabsession	<b>EAB_Session</b> group (reserved package) EAB session types
Java_InheritEJBRReserved itso.ejb35.javainherit	<b>Java_Inherit</b> group (reserved package) Java inheritance types
Bank_ReverseEJBRReserved itso.ejb35.reverse itso.ejb35.reverse.schema	<b>Bank_Reverse</b> group (reserved package) Bank reverse engineering types Reverse engineering schema and mapping.
RoseBankEJBRReserved itso.ejb35.rosebank	<b>Rose_Bank</b> group (reserved package) Bank imported from Rational Rose
Default package	Sample stored procedure
CMP11_EntityEJBRReserved itso.ejb35.cmp11 itso.ejb35.cmp11.schema itso.ejb35.cmp.client	<b>CMP11_Entity</b> group (reserved package) CMP type (deployment to WebSphere V4) CMP schema and mapping CMP client application and servlet

## Running the examples

To run the examples:

- ▶ Install and load the EJBBANK database in DB2 (see “Creating the database and tables” on page 62).
- ▶ Prepare VisualAge for Java as described in “Setting up VisualAge for Java” on page 466.
- ▶ Prepare WebSphere Application Server as described in “Setting up the WebSphere execution environment” on page 456.

## Run in VisualAge for Java

Import the repository file (*itsoejb35.dat*) into VisualAge for Java, and load the ITSO EJB Redbook project into the Workbench.

You can run the sample applications and servlets using the WebSphere Test Environment:

- ▶ Start the persistent name server.
- ▶ Create an EJB server configuration with the **CMP\_Entity**, **Bank\_Entities**, and **Stateless\_Session** EJB groups and start the EJB server.

## *Applications*

Be sure to calculate the class path for the applications, and add the *IBM WebSphere Test Environment* project manually. Then execute the application and watch the result in the Console window.

## **Servlets**

Copy the HTML and JSP files to the appropriate directories for test and execution by copying the content of the **web** subdirectory to:

```
<VAJ-HOME>\ide\project_resources\IBM WebSphere Test Environment\
hosts\default_host\default_app\web
```

Start the servlet engine after setting the class path (Figure 6-3 on page 84), and point a browser to:

```
http://localhost:8080/ejb/ITS0ejb.html
```

## **Run in WebSphere**

You can run the sample applications and servlets against the WebSphere Application Server after defining the application server and installing the EJBs:

- ▶ Install the EJBs in WebSphere (the JAR files are provided in the **ejbjar** subdirectory). See instructions in “Installation of the deployed code in WebSphere” on page 283 and “Deployment to WebSphere” on page 372.
- ▶ You can use the XMLConfig tool to define the application server, EJB container, and servlet engine (`itsoejb35server.xml` and `itsoejb35bank.xml`).

## **Applications**

Run the applications from the **ejbclient** subdirectory. Set the class path (`setCP.bat` file, must tailor), then execute using commands such as:

```
java itso.ejb35.cmp.client.Xxxxxxx
java itso.ejb35.bmp.client.Xxxxxxx
java itso.ejb35.gui.CustomerApplet
java itso.ejb35.bank.client.Xxxxxxx
```

## **Servlets**

Copy the HTML and JSP files to the appropriate directories for test and execution by copying the content of the **web** subdirectory to:

```
<WAS-HOME>\hosts\default_host\itsoejb\web
```

Copy the `itso_client.jar` file from the **ejbclient** subdirectory and the servlet class files from the **servlets** subdirectory to the Web application `servlets` directory:

```
<WAS-HOME>\hosts\default_host\itsoejb\servlets
```

Point a browser to:

```
http://localhost/itsoejb/ejb/ITS0ejb.html
```



# Related publications

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

## IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 499.

- ▶ *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*, SG24-5754
- ▶ *Enterprise JavaBeans Development Using VisualAge for Java*, SG24-5429
- ▶ *Programming with VisualAge for Java Version 3.5*, SG24-5264
- ▶ *WebSphere V3.5 Handbook*, SG24-6161
- ▶ *Version 3.5 Self Study Guide: VisualAge for Java and WebSphere Studio*, SG24-6136
- ▶ *How about Version 3.5? VisualAge for Java and WebSphere Studio Provide Great New Function*, SG24-6131
- ▶ *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755
- ▶ *Revealed! Architecting Web Access to CICS*, SG24-5466
- ▶ *IMS Version 7 and Java Application Programming*, SG24-6123
- ▶ *Migrating WebLogic Applications to WebSphere Advanced Edition*, SG24-5956
- ▶ *WebSphere Personalization Solutions Guide*, SG24-6214
- ▶ *User-to-Business Patterns Using WebSphere Advanced and MQSI: Patterns for e-business Series*, SG24-6160
- ▶ *WebSphere Scalability: WLM and Clustering Using WebSphere Application Server Advanced Edition*, SG24-6153
- ▶ *Patterns for e-business: User-to-Business Patterns for Topology 1 and 2 using WebSphere Advanced Edition*, SG24-5864
- ▶ *The XML Files: Using XML for Business-to-Business and Business-to-Consumer Applications*, SG24-6104
- ▶ *The XML Files: Using XML and XSL with IBM WebSphere 3.0*, SG24-5479

- ▶ *CCF Connectors and Database Connections Using WebSphere Advanced Edition Connecting Enterprise Information Systems to the Web*, SG24-5514
- ▶ *WebSphere V3 Performance Tuning Guide*, SG24-5657
- ▶ *WebSphere Application Servers: Standard and Advanced Editions*, SG24-5460
- ▶ *VisualAge for Java Version 3: Persistence Builder with GUIs, Servlets, and Java Server Pages*, SG24-5426
- ▶ *IBM WebSphere and VisualAge for Java Database Integration with DB2, Oracle, and SQL Server*, SG24-5471
- ▶ *Developing an e-business Application for the IBM WebSphere Application Server*, SG24-5423
- ▶ *The Front of IBM WebSphere Building e-business User Interfaces*, SG24-5488
- ▶ *VisualAge for Java Enterprise Version 2: Data Access Beans - Servlets - CICS Connector*, SG24-5265
- ▶ *Programming with VisualAge for Java Version 2*, SG24-5264-00, published by Prentice Hall, ISBN 0-13-021298-9, 1999
- ▶ *VisualAge for Java Enterprise Version 2 Team Support*, SG24-5245
- ▶ *Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications*, SG24-5276
- ▶ *Managing Your Java Software with IBM SecureWay On-Demand Server Release 2.0*, SG24-5846
- ▶ *Creating Java Applications Using NetRexx*, SG24-2216

## Other resources

These publications are also relevant as further information sources:

- ▶ *Enterprise JavaBeans*, Richard Monson-Haefel, published by O'Reilly, ISBN 1-56592-869-5.
- ▶ *Java Servlet Programming*, Jason Hunter with William Crawford, published by O'Reilly, ISBN 1-56592-391-X.
- ▶ *Developing JavaBeans with VisualAge for Java Version 2*, SC34-4735
- ▶ *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, published by Addison-Wesley Professional Computing Series, ISBN 0-201-633612, 1995

## Referenced Web sites

These Web sites are also relevant as further information sources:

- ▶ <http://www.ibm.com/software/data/>  
Database and Data Management
- ▶ <http://www.ibm.com/framework/patterns>  
IBM Patterns for e-business
- ▶ <http://www.ibm.com/software/webservers/appserv/>  
WebSphere Application Server
- ▶ <http://www7b.boulder.ibm.com/wsdd/>  
WebSphere Developer Domain
- ▶ <http://www.ibm.com/software/ad/vajava/>  
VisualAge for Java
- ▶ <http://www.ibm.com/software/vad/>  
VisualAge Developer Domain
- ▶ <http://www.flashline.com>  
Flashline Home Page
- ▶ <http://msdn.microsoft.com/scripting>  
Microsoft Windows Script Technologies

## How to get IBM Redbooks

Search for additional Redbooks or Redpieces, view, download, or order hardcopy from the Redbooks Web site:

[ibm.com/redbooks](http://ibm.com/redbooks)

Also download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become Redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

## IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.



# Special notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The 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. 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.

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere.,The Power To Manage., Anything. Anywhere.,TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

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

# Abbreviations and acronyms

<b>ACL</b>	access control list	<b>IBM</b>	International Business Machines Corporation
<b>API</b>	application programming interface	<b>IDE</b>	integrated development environment
<b>AWT</b>	abstract windowing toolkit	<b>IDL</b>	Interface Definition Language
<b>BLOB</b>	binary large object	<b>IIOP</b>	Internet Inter-ORB Protocol
<b>BMP</b>	bean-managed persistence	<b>IMS</b>	Information Management System
<b>BMS</b>	basic mapping service		
<b>CCF</b>	Common Connector Framework	<b>ITSO</b>	International Technical Support Organization
<b>CICS</b>	Customer Information Control System	<b>J2EE</b>	Java 2 Enterprise Edition
<b>CLOB</b>	character large object	<b>J2SE</b>	Java 2 Standard Edition
<b>CMP</b>	container-managed persistence	<b>JAF</b>	Java Activation Framework
<b>CORBA</b>	Component Object Request Broker Architecture	<b>JAR</b>	Java archive
<b>DBMS</b>	database management system	<b>JDBC</b>	Java Database Connectivity
<b>DCOM</b>	Distributed Component Object Model	<b>JDK</b>	Java Developer's Kit
<b>DDL</b>	data definition language	<b>JFC</b>	Java Foundation Classes
<b>DLL</b>	dynamic link library	<b>JMS</b>	Java Messaging Service
<b>DTD</b>	document type description	<b>JNDI</b>	Java Naming and Directory Interface
<b>EAB</b>	Enterprise Access Builder	<b>JSDK</b>	Java Servlet Development Kit
<b>EIS</b>	Enterprise Information System	<b>JSP</b>	JavaServer Page
<b>EJB</b>	Enterprise JavaBeans	<b>JTA</b>	Java Transaction API
<b>EJS</b>	Enterprise Java Server	<b>JTS</b>	Java Transaction Service
<b>FTP</b>	File Transfer Protocol	<b>JVM</b>	Java Virtual Machine
<b>GUI</b>	graphical user interface	<b>LDAP</b>	Lightweight Directory Access Protocol
<b>HOD</b>	host-on-demand	<b>MFS</b>	message format services
<b>HTML</b>	Hypertext Markup Language	<b>MVC</b>	model-view-controller
<b>HTTP</b>	Hypertext Transfer Protocol	<b>OMG</b>	Object Management Group
		<b>OS</b>	operating system
		<b>OTS</b>	object transaction service

<b>PB</b>	Persistence Builder
<b>PAO</b>	procedural adapter object
<b>RAD</b>	rapid application development
<b>RDBMS</b>	relational database management system
<b>RMI</b>	Remote Method Invocation
<b>SCCI</b>	source control control interface
<b>SCM</b>	software configuration management
<b>SCMS</b>	source code management systems
<b>SDK</b>	Software Development Kit
<b>SPB</b>	Stored Procedure Builder
<b>SQL</b>	structured query language
<b>SSL</b>	secure socket layer
<b>TCP/IP</b>	Transmission Control Protocol/Internet Protocol
<b>UCM</b>	Unified Change Management
<b>UDB</b>	Universal Database
<b>UML</b>	Unified Modeling Language
<b>UOW</b>	unit of work
<b>URL</b>	uniform resource locator
<b>VCE</b>	visual composition editor
<b>VXML</b>	voice extensible markup language
<b>WAS</b>	WebSphere Application Server
<b>WML</b>	Wireless Markup Language
<b>WTE</b>	WebSphere Test Environment
<b>WWW</b>	World Wide Web
<b>XMI</b>	XML metadata interchange
<b>XML</b>	eXtensible Markup Language

# Index

## A

AbstractAccessBean 207  
AbstractEntityAccessBean 208  
AbstractSessionAccessBean 208  
access bean 205  
    association 369  
    associations 386  
    client programs 214  
    inheritance 384  
    programming 254  
    session bean 270  
    types 208  
VisualAge for Java 212  
access control list 36, 413  
account  
    table 57, 58  
activation 28, 160  
administration  
    database 40  
Administrative Console 283, 455  
afterInteraction 189  
applet  
    EJB example 258  
Application Assembly Tool 475  
application server wizard 460  
architecture 440  
associations 203, 353  
    access bean 386  
    deployment 372  
    editor 356  
    generated code 357  
    guidelines 445  
    mandatory 355, 371  
    many-to-many 361  
    programming 384  
    servlet and JSP 389  
    VisualAge for Java 354  
attributes 217  
authentication 413  
authorization 414

## B

bank

database 55  
DDL 59  
example 53  
inheritance 330  
model 54, 132  
sample data 61  
transactions 392  
bank model  
    associations 354  
BankAccount 132, 317  
batch session bean 436  
bean life-cycle 35  
BeanFinderHelper 116, 194  
BeanFinderObject 194, 197  
bean-managed persistence  
    see BMP  
bean-managed transaction 233  
beforeInteraction 189  
BigDecimal  
    constructor 101  
    EJB test client 101  
    inspect 101  
BLOB 123, 303  
BMP 26  
    design 431  
    field 143  
    generated classes 143  
    methods 146  
    overview 139  
    VisualAge for Java 140  
    why 140  
bottom-up development 122, 309  
breakpoint 97, 293  
business  
    logic 439  
    methods 170

## C

C++ 21  
caching 34  
callback methods 144  
checking table 57  
CICS 6, 27

class path  
    client JAR 290  
    servlet engine 84  
cleanup 145  
client  
    application 486  
    applications 288  
        WebSphere 397  
    JAR 49, 278, 288, 474  
    module 475  
    programming 243, 381  
    types 8  
CLIENT\_IDENTITY 416  
client-managed transaction 234  
CMP 26, 47  
    architecture 300  
    creation in VisualAge for Java 110  
    design 431  
    development 108  
    field 106, 113  
    mapping 299  
    overview 105  
COBOL 6  
command 185  
commitCopyHelper 210  
Common Connector Framework 185  
complex attribute 325  
Component Broker 278  
Component Object Request Broker Architecture  
    see CORBA  
composer 311, 325  
connection 124  
ConnectionSpec 185  
container-managed persistence  
    see CMP  
container-managed transaction 232  
context factory 246  
conversation 158  
converter 303, 311, 314  
copy helper 209, 254  
    servlet 257  
CORBA 8, 20, 221  
CosNaming 9, 41  
CreateException 403  
custom finder methods 193  
Customer 110, 315  
customer  
    table 56  
CustomerAddress 114, 312

**D**  
data access beans 182  
data locking 228  
data source 41, 459  
    configuration 89  
    EJB deployment 283  
    WebSphere Test Environment 89  
database  
    connection 124  
    map 121  
    schema 121  
DB2 6  
    installation 454  
DCOM 20  
DDL 47, 59, 301  
deadlock 239  
deployed JAR 49, 278, 372  
deployment 29, 126  
    associations 372  
    code generation 127  
    descriptor 126, 147, 282  
    session bean 177  
    Version 4 469  
    VisualAge for Java 278  
deposit 135  
design 427  
    patterns 432  
    practices 437  
dirty read 229  
discriminator 331  
distinct table mapping 332  
distributed  
    objects 5  
    transaction 236  
Distributed Debugger 291, 294  
Domino 45  
DuplicateKeyException 403

**E**  
EAB 185  
    command 185  
    session bean tool 186  
    test client 186  
    tools 186  
EJB 17  
    applet example 259  
    best practices 443  
    client applications 243

concepts 19  
container 7, 12, 23, 24, 41, 283  
    responsibility 33  
data source 283  
debugging 97  
deployment 29, 126, 277  
design 428  
development environment 49, 65  
exceptions 399, 408  
generated types 116  
group 67, 109  
home 247  
IBM products 39  
in J2EE 9  
inheritance 329  
instance life cycle 106  
JAR 29, 49, 278, 474  
life-cycle 35  
model 19, 300  
modeling 419  
module 475, 477  
page 46, 66  
persistence 106  
programming 245  
properties 126, 285  
reference 247  
security 411, 418  
server  
    configuration 91, 128  
    properties 92, 129  
    start, stop 93  
    tracing 407  
servers 7  
specification 12, 402  
    1.0 12  
    1.1 13  
    2.0 14  
technology 428  
test client 80, 94, 130, 155  
    associations 371  
    BigDecimal 101  
    copy and paste 102  
    inheritance 346  
    WebSphere 287  
    why 5  
ejbActivate 117, 145, 168  
ejbCreate 14, 116, 117, 144, 150, 167  
    tailoring 120  
EJBDeploy 43  
EJBDeploy tool 489  
EJBException 404  
ejbFindByPrimaryKey 144, 152  
ejbLoad 116, 117, 144, 148  
EJBObject 22  
ejbPassivate 117, 145, 169  
ejbRemove 118, 145, 152, 169  
ejbStore 116, 117, 144, 149  
EJS 24  
EJSJDBCVerifier 196  
Encina 6  
Enterprise Access Builder  
    see EAB  
enterprise application 475, 477  
    installation 482  
enterprise connectivity 45  
Enterprise Java Server  
    see EJS  
Enterprise JavaBeans  
    see EJB  
entity bean 26  
    design 430  
enumeration 249  
environment  
    setup 453  
exceptions 399  
executeInteraction 189  
externalization 253

**F**

facade pattern 435  
find for update 239, 444  
findByPrimaryKey 37, 116, 118, 131  
findByPrimaryKeySqlString 195, 198  
finder  
    inheritance 346  
    method 249  
    associations 388  
FinderException 14, 247, 403  
foreign key relationship 304, 363  
forward engineering 307

**G**

getCallerIdentity 14  
getCallerPrincipal 14, 415  
getConnection 145  
getDatasource 145  
getEntityContext 146

getGenericFindInsertPoints 199  
getGenericFindSqlString 199  
getMergedPreparedStatement 197, 198  
getMergedWhereCount 197, 198  
getPreparedStatement 199  
getSessionContext 169  
greedy mode 193, 202

## H

Hashtable 263  
home  
  caching 447  
  create 247  
  find 247  
  lookup 246  
  remove 247  
home interface 36, 37, 116  
  customization 120  
  EJB test client 96  
  session bean 172  
HTTP 17  
HttpServletRequest 245  
HttpSession 245

## I

IDL 21  
IIOP 9, 20  
implementation  
  guidelines 447  
inheritance 329  
  access bean 384  
  deployment 372  
  finder 346  
  programming 382  
  schema and map 338  
  VisualAge for Java 336  
initial context 106, 246  
installation  
  DB2 454  
  products 454  
  VisualAge for Java 455  
  WebSphere Application Server 455  
  WebSphere Studio 456  
  Windows NT/2000 454  
instance pooling 35  
InstantDB 87  
Integrated Development Environment 44  
InteractionSpec 185

Interface Definition Language  
  see IDL  
Internet Inter-ORB protocol  
  see IIOP  
isCallerInRole 14  
isolation levels 126, 217, 227, 230

## J

J2EE 3, 9  
  IBM products 16  
  technologies 11  
  transaction 220  
J2SE 9  
JAF 11, 17  
JAR  
  EJB deployment 278, 474  
  installation 283  
Java  
  inheritance 330, 348  
Java 2 Enterprise Edition  
  see J2EE  
Java Activation Framework  
  see JAF  
Java Database Connectivity  
  see JDBC  
Java IDL 10  
Java Mail 11, 17  
Java Messaging Service  
  see JMS  
Java Naming and Directory Interface  
  see JNDI  
Java Transaction API  
  see JTA  
Java Transaction Service  
  see JTS  
JavaBean  
  wrapper 208  
JavaBeans 23, 431  
JavaServer Pages  
  see JSP  
JDBC 9, 16  
  coding 180  
  driver 124, 431, 457, 466  
  isolation level 232  
JetAce 42, 281  
JMS 10, 14, 16  
JNDI 9, 16, 25, 35, 38, 41, 82, 209  
  lookup 95

JSP 5, 10, 17  
  associations 389  
  compiler 48  
  example 252  
  execution monitor 80, 88  
  source debugging 85  
JTA 10, 17, 23, 220, 221  
  database driver 90  
JTS 10, 17, 220

## K

key class 37

L  
lazy mode 193, 203  
LDAP 9  
locking 228  
logic splitting 437, 441

M  
management methods 168  
mandatory association 355  
map 121  
  associations 359  
  browser 123, 306  
  inheritance 339, 343  
meet-in-the-middle development 122, 310  
message-driven beans 14  
method  
  control descriptor 127  
  read-only 119  
method-level  
  attributes 225  
  security 415, 429  
methods  
  finder 193  
middle-tier  
  architecture 8  
middleware 4  
model-view-controller 10, 244, 432  
MQSeries 6

N  
non-deployed JAR 280, 286  
nonrepeatable read 229  
non-repudiation 414  
NoSuchEntityException 405

numberOfRows 211

O  
object  
  caching and pooling 7  
  distribution 5  
  persistence 5  
  request brokers 4  
Object Level Trace 291  
Object Management Group 9, 20  
ObjectNotFoundException 403  
object-oriented  
  database 35  
object-to-relational persistence 35  
Oracle 41  
OTS 221

P  
passivation 28, 160, 175  
performance  
  guidelines 449  
persistence 34  
  guidelines 444  
Persistence Builder 45, 445  
persistent name server 80, 86  
  configuration 87  
  start, stop 88  
phantom read 229  
platform independence 6  
property map 306

Q  
queries 180  
query string 195

R  
Rational Rose 419  
  model 420  
read-only method 119  
realm 413  
record  
  Java 185  
Redbooks Web Site 499  
  Contact us xxi  
refreshCopyHelper 210  
remote interface 36  
  add methods 119

EJB test client 96  
session bean 171  
Remote Method Invocation  
  see RMI  
RemoteException 247, 402, 405  
RemoveException 403  
resource manager 220, 237  
reverse engineering 309, 376  
RMI 20  
  overview 21  
RMI-IIOP 8, 9, 17, 22, 221  
rollback 239  
root/leaf table mapping 331, 341  
rowset 210  
  programming 256  
RuntimeException 400, 402, 404

## S

sample code 492  
SAP 45  
savings table 57  
schema 121  
  associations 358  
  browser 122, 302  
  import 376  
  inheritance 339, 341  
secondary tables 321  
security 411, 442  
  applet 261  
  method level 415  
  role 414  
  server 417  
  VisualAge for Java 416  
  WebSphere 291, 412  
Select bean 180  
serializable 304  
servlet 5, 17  
  associations 389  
  copy helper 257  
  EJB example 250  
  engine 41, 252  
    class path 84  
    settings 85  
    start, stop 86  
  session bean 395  
  session facade 264  
  testing 252, 289  
servlet engine 80

session bean 27, 157  
  access bean 270  
  batch 436  
  comparison 29  
  data access beans 182  
  database query 180  
  deployment 286  
  design 429  
  EAB 186  
  field 175  
  implementation 164  
  JDBC coding 181  
  lifetime 158  
  overview 158  
  servlet 395  
  stored procedure 183  
  test 178  
  VisualAge for Java 174  
session facade 261, 434  
SessionSynchronization 14  
setEntityContext 145  
setTxRecHome 135  
single-table mapping 331  
Smalltalk 21  
SPECIFIED\_IDENTITY 417  
SQL Assist 182  
SQL Server 41  
SQLJ 27, 45  
state management 35, 441  
stateful session bean 28, 159, 430  
stateless session bean 28, 162, 430  
  life cycle 160, 163  
stored procedure 180, 183  
Stored Procedure Builder 45, 183  
Sybase 41  
SYSTEM\_IDENTITY 417

## T

table map 306  
three-tier architecture 245  
throwable 400  
top-down development 121, 307  
tracing 292  
transaction 217, 442  
  attributes 126, 222  
  bean-managed 233  
  client-managed 234  
  container-managed 34, 232

context 237  
deadlock 239  
demarcation 232  
distributed 236  
guidelines 240, 444  
management 6, 217  
monitors 4  
overview 218  
table 58  
**TRANSACTION\_MANDATORY** 444  
**TRANSACTION\_READ\_COMMITTED** 231  
**TRANSACTION\_READ\_UNCOMMITTED** 230  
**TRANSACTION\_REPEATABLE\_READ** 231  
**TRANSACTION\_REQUIRED** 444  
**TRANSACTION\_SERIALIZABLE** 231  
TransactionException 408  
TransactionRecord 140  
TransactionRequiredException 222, 406  
TransactionRolledBackException 406  
TransactionRolledbackException 405  
TransRecord 133, 317  
two-phase commit 236  
**TX\_BEAN\_MANAGED** 223, 442  
**TX\_MANDATORY** 222, 406, 442  
**TX\_NOT\_SUPPORTED** 223  
**TX\_REQUIRED** 223  
**TX\_REQUIRES\_NEW** 223  
**TX\_SUPPORTS** 223

**U**  
unsetEntityContext 145  
user profile management 42  
UserTransaction 14, 233, 406

**V**  
VapBinaryStreamToSerializableConverter 303  
VapConverter 303, 314  
VapEJSJDBCFinderObject 198  
VapTrimStringConverter 303  
VisualAge for Java 39  
    installation 455  
    overview 44  
    project 493  
    setup 466  
    Version 4 470  
    XMI Bridge 420

**W**  
Web  
    application 42, 82, 460  
        directories 464  
    containers 12  
    module 475, 478  
WebSphere  
    application assembly 475  
    Application Server 39  
        installation 455  
    architecture 40  
    configuration 456  
    EJB deployment 277  
    Enterprise 278  
    installation 455  
    security 412  
    single server version 475  
    Studio  
        installation 456  
    Version 4 469, 475  
WebSphere Test Environment 46, 48, 79  
    Control Center 81  
where clause 194  
withdraw 135  
workload management 7, 43

**X**  
X/Open XA 220  
XA protocol 236  
XMI Toolkit 420  
XML 17  
    configuration tool 83, 290, 373

**Z**  
zero argument constructor 212



**IBM**



**Redbooks**

# EJB Development with VisualAge for Java for WebSphere Application Server

(1.0" spine)  
0.875" <-> 1.498"  
460 <-> 788 pages







# EJB Development with VisualAge for Java for WebSphere Application Server

## Jumpstart EJB development with VisualAge for Java

## Deploy EJBs to WebSphere Application Server

## EJB mapping techniques for relational databases

This IBM Redbook provides detailed information on how to effectively use VisualAge for Java Enterprise for the development of applications based on the Enterprise JavaBeans architecture, and deployment of such applications to a WebSphere Application Server. This redbook is a companion book to the redbooks on IBM Patterns for e-business.

In Part 1 we introduce Enterprise JavaBeans as a part of Java 2 Enterprise Edition and cover the basic concepts and the architecture. We also introduce the IBM products that support this architecture.

In Part 2 we describe the IBM development and test environment for enterprise beans, and we create rather simple container-managed and bean-managed entity beans, session beans, finder methods, access beans, and client programs. We explain the basic transaction management facilities and describe how to deploy enterprise beans to a WebSphere Application Server.

In Part 3 we explore advanced facilities for mapping entity beans to database tables, including the extended IBM support for inheritance and associations. We describe how these advanced facilities are used in client programs and touch on exception handling, security, and modeling with Rational Rose.

In Part 4 we provide design guidelines and best practices when developing applications based on enterprise beans.

Throughout the book we provide examples based on a simple banking application with an underlying relational database.

## INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

## BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:**  
[ibm.com/redbooks](http://ibm.com/redbooks)