



Chordiant.

Chordiant 5

FOUNDATION SERVER
APPLICATION COMPONENTS DEVELOPER'S GUIDE

Important Notice

Chordiant and the Chordiant logo are registered trademarks of Chordiant Software, Inc. with patents pending.

Copyright © 1997-2005 by Chordiant Software, Inc. All rights reserved.

This document is Chordiant Confidential Information intended for the exclusive use of Chordiant Software, Inc., its affiliates, and its customers and partners. No part of this publication may be reproduced, revised, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Chordiant Software, Inc. No licenses, express or implied, are granted with respect to any of the technology described in this document.

Trademark Acknowledgments

BEA and WebLogic are registered trademarks of BEA Systems, Inc. FioranoMQ is a registered trademark of Fiorano, Inc. IBM and WebSphere are registered trademarks of IBM Corporation. Microsoft and Windows are registered trademarks of Microsoft Corporation. Oracle is a registered trademark of Oracle Corporation. Sun, Sun ONE Directory Server, and J2EE are registered trademarks of Sun Microsystems, Inc.

Nokia is a registered trademark of Nokia Corporation. Nokia's product names are either trademarks or registered trademarks of Nokia.

The product name of Sony Ericsson product is a trademark or other protected right used by Sony Ericsson.

All other company or product names may be trademarks or registered trademarks of the respective companies with which they are associated.

Publishing Information

Chordiant 5 Foundation Server Application Components Developer's Guide
Release 5.7. Document version 1.0 February 2005

Please email suggestions and corrections to tech_com_ww@chordiant.com.

World Headquarters

Chordiant Software, Inc.
20400 Stevens Creek Blvd.
Cupertino, CA 95014
1-408-517-6100
Fax: 1-408-517-0270
www.chordiant.com

European Headquarters

Chordiant Software Int'l, Inc.
2 Goat Wharf, High Street
Brentford, Middlesex
TW8 0BA, UK
+44(0) 20 8380 0600
Fax: +44(0) 20 8380 0606

Customer Support

Americas:

us.support@chordiant.com
1-408-517-6200
Toll-free: 1-877-866-7243

EMEA:

+44 (0) 20 8580 0400
uk.support@chordiant.com

Contents

Preface	ix
Chapter 1	Introduction to Application Components 1
	Modeling Overview 2
	XMI, SMI, and CMI Files 2
	Generating Business Components 2
	Customization Introduction 4
	Customization Philosophy 4
	Customization Guidelines 5
	Rose Models..... 5
	Business Service Definition 5
	Business Object Definition..... 6
	Levels of Customization 6
Chapter 2	Setting Up Your System for Component Generation 9
	Configuring Rational Rose 9
	Configuring for XMI Export..... 9
	Configuring for Business Component Generation 10
	Modeling and Code Generation Concepts 12
	Issues in Rational Rose 12
	Data Type Errors in Rational Rose 12
	Missing Default Values in CMI..... 13
Chapter 3	Using the Business Component Generator 15
	Preparing for Code Generation 15
	Creating Your Model and Exporting to XMI 15
	Creating the Descriptor File 16
	Using the Business Component Generator 17
	Before You Begin..... 17
	Turning Off Automatic Code Generation 17
	Checking Classpath Variables 18
	Enabling Logging 18
	Allocating Memory and Resources 19
	Running the Business Component Generator 20
	Manually Starting a Build Process 30
	Advanced Topic: Creating Business Components Through Ant Scripts 32
	Before You Begin..... 32
	Running the Ant Script to Create Business Components 35

	Targets for Ant-Based Generation	37
	Deploying Your Application Components	38
	Creating a JAR File	38
Chapter 4	Creating or Modifying Business Services	41
	From Model to Service Components	42
	Business Service Framework Components Created	43
	Business Services Overview	44
	Modifying Service Framework Components	46
	Overloading in Services	51
	Performing Business Domain Number Generation	52
	Customization Concepts	55
Chapter 5	Tutorial: Creating and Extending a Business Service	57
	Part I: Creating a New Service	58
	Creating a Model and the Base Class	59
	Generating the Service Framework Components	62
	Verifying the Build	63
	Troubleshooting	63
	Adding Logic to the Service	63
	Registering the Service Through Configuration	66
	The Client Agent Tester	68
	Setting up the Runtime Environment	68
	Part II: Extending an Existing Service	76
	Introduction	76
	Extending the Service	76
	Generating the Service Framework Components	79
	Adding Business Logic to the Current Year Service	80
	Confirming the Client Agent Parent Class	81
	Updating the Configuration Settings	82
	Testing the Service	85
	Copying the Tester	85
	Exporting the JAR	85
Chapter 6	Creating Web Services	87
	Web Services Topology	88
	Chordiant-Provided Web Services	90
	Security and Web Services	91
	Configuring for Using Web Services	91
	Creating a Java Proxy Code Project	91
	Requirements for Creating Web Services	96
	Generating Chordiant Web Services	97
	Specifying the WSDL Distribution Method	97
	Handling Different Types of Services	98
	Model-Based Services	98

Services Without Models	98
Generating Web Service Components	99
Copying the JAR Files	99
Creating the WSDL Files	100
Creating the Deployment Descriptor and Proxy Files	103
Starting the Chordiant Server	106
Deploying Web Services	106
Undeploying Web Services	108
Editing the server-config.wsdd File and Restarting the Server	111
Modifying WSDL Files	111
Restarting the Server	112
Generating the Proxies Again and Testing Them	112
Listing Available Web Services Components	113
Deploying Web Services to a Production Environment	113
Before You Deploy.....	114
Deploying Individual Web Services in Production	114
Error Handling	114
Best Practices	115
Using Java Code.....	115
Calling External Web Services from Chordiant Applications.....	115
Regenerating and Redeploying WSDLs	115
Chapter 7 Invoking and Integrating Web Services	117
Static and Dynamic Invocation	117
Installing the Sample Applications for Web Services.....	118
Invoking Web Services Statically	120
Invoking Chordiant Web Services.....	120
Creating the Proxy Code for the PmfCustomer Web Service	121
Incorporating the Chordiant Proxy Code	124
Running the Sample Application	126
Using Non-Chordiant Web Services Within Chordiant.....	129
Creating a Java Proxy Code Project	129
Creating the Proxy Code for the Stock Quote Web Service	130
Incorporating the Proxy Code into a Chordiant Service	132
Running the Sample Application	142
Invoking Web Services Dynamically through SAAJ	144
Running the Sample Application	148
Chapter 8 Creating Task Descriptors	151
Task Descriptor Location	151
Creating and Updating Task Descriptors	152
Task Descriptor Types.....	152
Working with Services	152
Model-Based Services	153
Services Without Models	153

	Generating Task Descriptors	153
	Preparing to Generate	154
	Generating the Task Descriptors	158
Chapter 9	Creating or Modifying Business Objects	163
	Persistence Components Created	163
	Customizing Business Objects	165
	Important Notes	165
	Serialized Objects	165
	Web Services	165
	Criteria for Customizing Business Objects	165
	Steps for Customizing a Business Object	166
	Instantiating the Business Object.....	169
	Using Object Locking	169
	Transferring Attributes	172
	Data and Caching Considerations.....	173
	Specifying Persistence Metadata	173
Chapter 10	Extended Persistence Components	193
	Using Extended Persistence Components	193
	Generic Service Overview	194
	Sequence Flow Using Generic Service	195
	Generic Service Client Agent	196
	Architecture	196
	Components Generated Automatically	197
	Business Object Behavior.....	198
	Behavior Base Class	204
	Behavior Base Class Exceptions	206
	Helpers	206
	AccessStrategyInput and AccessStrategy.....	207
	Validation.....	209
	Business Object Behavior Factory	211
	Changing the Behavior Factory Configuration	213
	Loading Classes Dynamically	214
	Usage Model for Extended Persistence Components	215
Chapter 11	Specifying Extended Persistence	217
	Using Extended Persistence Components	217
	Extended Persistence Components Created	218
	Specifying Extended Persistence Information	220
	Access Strategy.....	221
	useBehavior	223
	Inheritance Tags	224
	Specifying Type to Aid Lookup Efficiency	225
	xrefTableName	228

	typeValue	229
	typeField	230
	CMI Example	231
	Associations	232
	Association Type	232
	Aggregations	234
	Multiple Associations	237
	Adding Documentation	241
Chapter 12	Tutorial: Extended Persistence Components	243
	The Tutorial Model	244
	Inheritance	245
	Creating a Model and the Base Class	245
	Adding a New Class, Subclassing from Base Class	246
	Creating a New Subclass	252
	Containment	255
	Associations	261
	Overriding Behavior.....	265
	Overriding Access Strategy	267
	Creating Testers	270
Chapter 13	Party Management Facility	273
	Using the Party Management Facility API	275
	Using the CommonObject Interface.....	277
	Using the Role Interface.....	279
	Using the Node Interface.....	281
	Using the Party Interface	284
	Using the PartyRole Interface	288
	Using the Manager Interface	292
	Using the PartyManager Interface	292
	Using the RoleManager Interface	293
	Using the PartyRoleManager Interface.....	294
	Using the RelationshipManager Interface.....	294
	Example Scenarios	295
	Creating a Customer.....	295
	Establishing Marriage Between Two People	295
	Converting a Prospect to a Customer.....	296
Chapter 14	Customizing the PartyRole Service	297
	Business Objects and Business Object Behavior	297
	Business Services	298
	Integrating Business Object Behavior	300
	Business Object Behavior Considerations	300
	Customizing Business Object Behavior	301
	Customization Example: PmfCustomerService	303

Chapter 15	Metadata	305
	Understanding the Metadata Format	306
	Business Metadata	307
	Exploring the Core Business Metadata.....	308
	Persistence Metadata	310
	SQL Persistence Tags	310
	MQ Persistence Tags	314
	Extended Persistence Tags	318
	Service Metadata	321
	Generating Code from CMI or SMI	325
Chapter 16	Chordiant 5 Application Components	327
	Task Descriptors for Business Process Designer	327
	Account Service	328
	Customizable Components	329
	Credit Card Account Service.....	331
	Delivery Service (Stub)	331
	EBC Interaction Service	331
	Customizable Components	333
	Guide Service	335
	Customizable Components	335
	Location Service	336
	Customizable Components	336
	Offering Service	337
	Customizable Components	337
	Order Fulfillment Service	338
	Customizable Components	338
	Order Generation Service	340
	Customizable Components	341
	Order Tracking Service	342
	Customizable Components	342
	Party Role Service	343
	Customizable Components	345
	PartyRoleClientAgent	347
	PmfCustomerClientAgent	349
	PmfDelegateService	350
	Product Service	351
	Customizable Components	352
	Backward Compatibility APIs	353
	Customer Service Client Agent.....	353
	EstablishmentClientAgent.....	353
Index	355

Preface

This manual describes how to create Chordiant 5 Foundation Server application components using Rational Rose and Chordiant's Business Component Generator.

Who Should Use This Guide

This manual is intended for Chordiant Application Developers who need to define and develop applications using the Chordiant 5 Foundation Server.

Manual Organization

This manual contains these chapters:

Chapter 1	The introduction describes what business components are and how they are generated.
Chapter 2	Describes how to set up Rational Rose for generating business components and includes important information to keep in mind as you model.
Chapter 3	Describes how to use the Business Component Generator, including important steps to take before running it.
Chapter 4	Describes how to create business services and associated service framework components by modeling them in Rational Rose. Automatic business domain number generation is also described.
Chapter 5	A tutorial which guides you through the steps of customizing and extending a service.
Chapter 6	Describes how to create web services from Chordiant services.
Chapter 7	Using tutorials and sample applications, describes how to incorporate web services into your Chordiant solution and how to consume Chordiant-based web services.
Chapter 8	Describes how to create Business Process Designer task descriptors based on business services.
Chapter 9	Describes how to create business objects and associated persistence application components.
Chapter 10	Provides background on extended persistence components, including the generic service, access strategies, and behaviors.
Chapter 11	Describes how to model extended persistence components in Rational Rose.

Chapter 12	A tutorial which guides you through the steps of creating extended persistence components.
Chapter 13	Describes the PartyManagementFacility provided by Chordiant.
Chapter 14	Describes how to customize the PartyManagementFacility using business object behaviors.
Chapter 15	Describes metadata - the data that describes the application components.
Chapter 16	Describes the application components provided by Chordiant.

Additional Documentation

These documents provide additional details about business components, Chordiant 5 Foundation Server, and the Chordiant Tools Platform:

- *Chordiant 5 Foundation Server Developer's Guide*
- *Chordiant 5 Tools Platform Getting Started Guide*

For definitions of Chordiant terms, refer to the *Chordiant 5 Terminology Guide*.

Typographical Conventions

This section explains how to interpret the font changes and notes that you see in this manual.

CONVENTION	EXAMPLE
System filenames and pathnames	Readme.txt is a text file that is stored on the application server in the /etc (for UNIX) or C:\ (for Windows NT) directory.
Document names and module names	See the “Security” section within the <i>Ongoing Tasks</i> document, or the online help from within the <i>Security</i> module.
Names of code elements and small pieces of code - or - Onscreen text and text typed on the keyboard	Use the <code>getInfo</code> method Type the password <code>cmyk</code> .
Screen element labels, including buttons and menus - or - Keys that you press on the keyboard	Click OK . Then from the File menu, select Save . To save the information on the page, press CTRL + SHIFT + s .
Variables that you must define based on your own settings	<code>{JAVA_HOME}</code> /com/chordiant/jxw

Gray boxes show code to be entered or viewed.

Note: A note shows important information that you should be sure to read. Many notes refer to other sections for more information.

Tip: A tip gives suggestions on how you can use the application faster or more efficiently.

Caution: A caution statement warns of steps you should take, or avoid, so you do not damage your equipment, data, or system reliability.



Introduction to Application Components

Chordiant 5 Foundation Server uses a Model-Driven Architecture (MDA), where many software building blocks are generated from models defined in a case tool, specifically Rational Rose. This model-driven architecture enables you to graphically define your business data and service models, then customize them by extending the model-generated classes. The MDA generates the components listed in [Table 1-1](#).

DATA MODELS	SERVICE MODELS
Java source	service skeletons
unit test code template	client agents
XML schema files	configuration files
database scripts	constants class

Table 1-1: Components Created by the Model-Driven Architecture

Using application components (including business services, client agents, and business objects) within your environment offers several advantages. Application components:

- perform well in a distributed computing environment
- enable enterprise business logic to execute in a J2EE application server
- enable enterprise business logic to be accessible from all client channels
- provide a flexible customization model that enables point- specific overrides without affecting the rest of the business processing

For background information on the JX Architecture, including services, client agents, and persistence, refer to the *Chordiant 5 Foundation Server Developer's Guide*.

MODELING OVERVIEW

The object model specifies information about business objects through metadata. The Chordiant 5 UML Extender for Rational Rose enables you to specify Chordiant-specific metadata for your model. After you export your model to XMI, Chordiant applies XSL style sheets in several steps to create a proprietary Chordiant Meta Information (CMI) XML file, which Chordiant can then transform into business components.

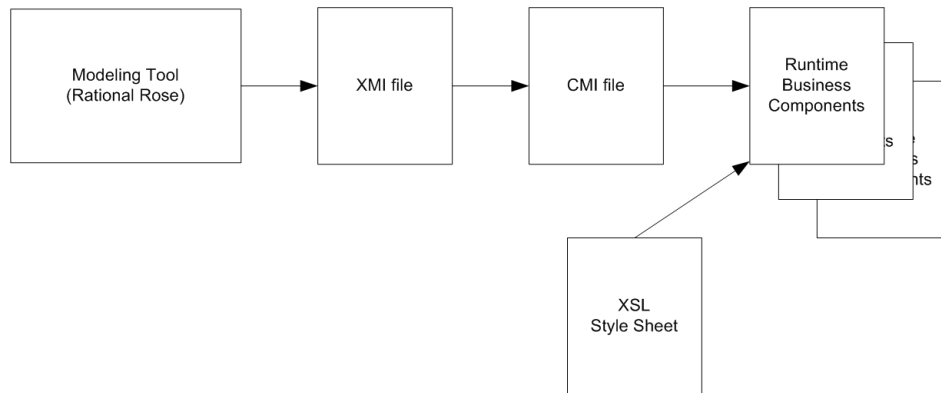


Figure 1-1: Flow of Business Component Generation

XMI, SMI, and CMI Files

Chordiant tools take the XMI exported from Rational Rose and turn it into Chordiant-based SMI (for service framework) and CMI (for business component and persistence) files. The transformations from the design tool through SMI and CMI are transparent to you, but enable the Business Component Generator to work more efficiently. The SMI and CMI files take only the necessary parts of the XMI file, so the Business Component Generator is not bogged down with extraneous information.

GENERATING BUSINESS COMPONENTS

The Business Component Generator creates a number of business components, listed later in this section. Creating and modifying these components is discussed in other sections of this guide.

Business components created with this tool use the JX Architecture and integrate well with the rest of the Chordiant 5 Foundation Server system.

Service Framework Components:

Refer to [“Business Service Framework Components Created” on page 43.](#)

- Service Skeleton
- Client Agent
- Configuration
- Constants

These components can be generated after you have created the service framework components.

- Web Services Components (Refer to [“Creating Web Services” on page 87.](#))
- Business Process Designer Task Descriptors (Refer to [“Creating Task Descriptors” on page 151.](#))

Persistence Components:

Refer to [“Persistence Components Created” on page 163.](#)

- Business Object
- Business Object Criteria
- Data Accessor (Standard Persistence API)
- Business Object Behavior Skeleton

Extended Persistence Components:

Refer to [“Extended Persistence Components Created” on page 218.](#)

- Business Object Behavior (Extended Persistence API)
- AccessStrategy and AccessStrategyInput
- Helper classes
- Validators

The PartyRole service has a different method of customization. It is discussed separately in [Chapter 14, “Customizing the PartyRole Service”.](#)

CUSTOMIZATION INTRODUCTION

You can customize application components within the Foundation Server environment, thereby enabling you to add or modify features in your application environment. Since application components consist of business services together with business objects and their associated behavior, Foundation Server enables you to customize:

- Business services
- Business objects and business object behavior, including attributes, methods, and persistence behavior

When performing customization, always place new code in packages and classes that are separate from Chordiant code. This helps avoid overwriting your customizations during upgrades to the Chordiant product.

CUSTOMIZATION PHILOSOPHY

You can customize Chordiant Application Components to add or modify the capabilities of your application environment.

Customization typically involves these tasks:

1. Locating an existing entity within the system which most closely resembles your required functionality.
2. Subclassing the original entity.
3. Adding, overriding, or overloading specific attributes and behavior within the derived entity.
4. Modifying the configuration files to notify the system of your changes.

Once you have completed the customizations, the system handles specific overrides and activates the customized entity, if available.

CUSTOMIZATION GUIDELINES

Before you begin the customization process, consider the criteria described in this section.

Rose Models

In the Chordiant platform, models should be purely service models or object models. Do not blend the two within one model.

Business Service Definition

In the Chordiant platform, a Business Service is a standard JX service that adheres to certain criteria:

- Subclassed from the `BusinessDataServiceBaseClass`.
- Are modeled in a UML modeling tool, namely Rational Rose.
- Contain channel-independent enterprise business logic.
- Are configured through Chordiant service configuration. Refer to the “Configuration” chapter of the *Chordiant 5 Foundation Server Developer’s Guide*.
- Are generated using the Chordiant Business Component Generator.
- Define methods that take Chordiant Business Objects as parameters and return only Chordiant Business Objects (as defined in the next section). They should *not* take interfaces as parameters.
- Uses only Chordiant Business Object object factories, and not constructors (`new()`’s) to create Business Objects.
- Use only the Chordiant Persistence Server component for persistent access. Refer to the “Persistence Server” chapter of the *Chordiant 5 Foundation Server Developer’s Guide*.

Business Object Definition

In the Chordiant platform, a Business Object is a “data only” class that contains only attributes, getters, and setters. Chordiant Business Objects do not contain business logic. Chordiant-provided Business Objects adhere to certain criteria:

- Subclass from the `CorporateBusinessClass`.
- Are modeled in a UML modeling tool, namely Rational Rose.
- Contain only data — no business logic.
- Have public getters and setters for all public attributes.
- May have persistent attributes, non-persistent attributes, or both.
- Are generated using the Chordiant Business Component Generator.
- Use only the Chordiant Persistence Server component for persistent access.

Levels of Customization

Chordiant Business Objects Level 1:

No-coding customization of an existing Chordiant Business Object

One way to customize existing Chordiant Business Objects is to use the no-coding customization model. In the no-coding customization model, all modifications are done in Rational Rose and no Java coding is required.

To use this customization model, you:

- **cannot** remove attributes from an existing Business Object.
- **cannot** change attribute data types of an existing Business Object.
- **can** change the attribute persistent mapping of an existing Business Object.
- **can** change the attribute size and precision of an existing Business Object. Note the size and precision can only be increased, but not decreased. For example, the size of a String attribute can be changed from 30 to 50 or any size larger than 30. Numeric attributes can also increase in precision as long as they adhere to the confines of the associated Java numeric data type.
- **can** subclass and specify an Object Factory override of an existing Business Object.
- **can** add attributes, in the subclass only, to an existing Business Object.

For details on specifying business objects, refer to [Chapter 9, “Creating or Modifying Business Objects”](#)

Chordiant Business Service Level 1: Customization of an Existing Chordiant Business Service

One way to customize an existing Chordiant Business Service is to add to, overload, and/or override its behavior. Behavior is business logic that operates on business data.

To use this customization model, you must observe these criteria:

- **cannot** remove behaviors from an existing Business Service.
- **can** subclass an existing Business Service.
- **can** add new behaviors, in the subclass only, to an existing Business Service.
- **can** provide behavior overloads to an existing Business Service and overrides to the Business Service subclass.

For step-by-step instructions on customizing business services, refer to [Chapter 4, “Creating or Modifying Business Services”](#)

Chordiant Business Service Level 2: Advanced Customization of an Existing Chordiant Business Service

If Level 1 Business Object customization and Business Service customization do not provide enough flexibility for a given problem, deeper level customizations of existing Chordiant Business Objects and Business Services can and do occur and are supported. When this deeper level of customization does occur, you are freed from many of the restrictions of the Level 1 customizations, but you also need to take over more existing components and must implement more custom Java code.

Even though Level 2 customization allows more flexibility, it is not an open-ended customization, as Chordiant wants to maintain an upgrade path for customers for future releases of the Chordiant-provided components. Level 2 customizations can vary from one Chordiant-provided component to another and must be carefully considered. Chordiant provides field representatives to analyze the specific requirements and help guide the customization to something that can be supported. Without some knowledge of which part of the Chordiant product is to be customized, and some knowledge of the target requirements, it is not possible to issue a detailed blanket recipe for such customizations.

Chordiant Business Service Level 3: Creating New Business Services

It is possible that if the Level 1 and Level 2 customizations are not flexible enough. In this case, an existing component customization might not be the answer and perhaps a new component should be built instead. It is Chordiant’s goal that this should not be standard practice, but it is a possible and a normal occurrence and should be considered.

Setting Up Your System for Component Generation

Before you can use the Business Component Generator to create business components, you must set up the Chordiant 5 UML Extender for Rational Rose. As you create your model, you must consider some Chordiant modeling concepts as well as some data issues with Rational Rose. Be sure to review [“Modeling and Code Generation Concepts” on page 12](#) and [“Issues in Rational Rose” on page 12](#).

Note: The Business Component Generator creates business components from your Rational Rose model. For instructions, refer to [Chapter 3, “Using the Business Component Generator”](#).

CONFIGURING RATIONAL ROSE

Rational Rose does not come equipped with XMI import and export support. You must add this functionality, which is available as a download from Rational’s corporate web site. Refer to the steps in the next section, [“Configuring for XMI Export”](#).

Configuring for XMI Export

To configure Rational Rose for Java file generation:

1. Install Rational Rose.
2. Obtain and run the `UnisysRoseXMLTools.exe` file.

Note: Be sure to use the version supported by Chordiant in this release. This information is located in the Supported Platforms documentation for this release.

3. Follow the prompts and instructions on the installer program.

You can choose the default settings in most cases. For component selection, select all three of these components: Rose MOF, Rose UML, and Rose XMI.

4. Complete the installation process.

5. Open Rational Rose and check the **File** menu for the new export and import options.

You should see these two options:

- **Export UML 1.1 to XMI**
- **Import UML 1.1 XMI File**

When you actually export your data, choose to export in the **ASCII** data type.

Configuring for Business Component Generation

You must run a Rational Rose script file before creating your model and generating Java files.

To complete the configuration of Rational Rose for business objects:

1. Using Rational Rose, from the **Tools** menu, select **Open Script**.
2. Select the Rational Rose JXC Support.ebs script using the **Open** dialog box.

You can find the Rational Rose JXC Support.ebs script file in this directory:

`{eclipse_root}/plugins/{data_model_plugin}`

3. From the **Debugger** menu, select **Go**.

Rational Rose executes the script.

4. Close the script once it has completed executing.

Rational Rose is now configured to accept input of Chordiant metadata, so you can generate business components from your model.

You can confirm a successful configuration by creating a new class and double-clicking on it. The resulting **Class Specification** window should look like [Figure 2-1](#).

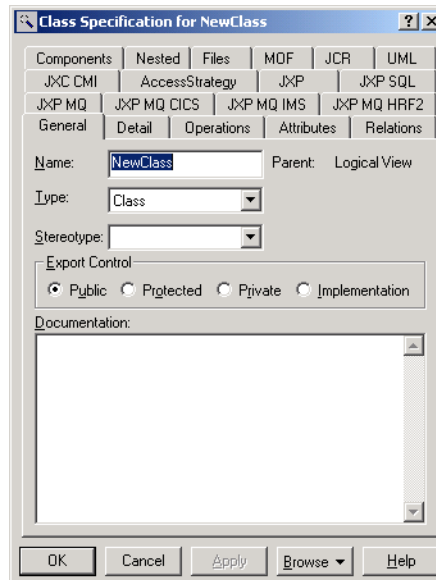


Figure 2-1: Class Specification Tabs in the Chordiant-Configured Rational Rose

Notes: Ensure that you are running the version of Rational Rose specified in the Technology Stack documents on the root level of the Chordiant installation CD. If you receive a **Petal Version Incorrect** error when you open a model, you are probably running the wrong version of Rational Rose.

You must run this script once for your installation of Rational Rose to see the Chordiant information for the Chordiant-provided models.

When using or creating other models, you must run this script against each model.

MODELING AND CODE GENERATION CONCEPTS

If you are using Rational Rose to generate application components, keep these points in mind:

- When creating your model, be sure to put your components in the appropriately named directory. If you do not use these directory names, your code will not generate properly.
 - services
 - clientagents
 - constants
 - accessstrategies
- Always try to make your customizations isolated from the Chordiant-provided components, in a separate package and, if possible, a separate diagram. You can choose to create a new model and copy only the relevant portions of the Chordiant model into your new model, to save time and to keep the model simple. Also see the next bullet point.
- If, in your project, both Chordiant and customized code are generated, use only your customized code. Only your code includes the changes you made.
- Chordiant 5 does not support many to many relationships. (For persistence models only.)
- All classes must be derived from the Chordiant base classes. Otherwise, you are not taking advantage of the JX architecture and hinder your ability to upgrade in the future. (For persistence models only.)
- Do not use a copyright symbol (©) in your model, including the Javadoc. The code generation will not work properly if it encounters a copyright symbol.

ISSUES IN RATIONAL ROSE

When you are using Rational Rose, you might encounter these issues.

Data Type Errors in Rational Rose

Occasionally, your Rational Rose model might have an issue with referenced classes, displaying something like:

Logical View::java::lang::String

This will cause a problem with the code generation, which works from your model's exported XML. When you try to create the code in the Business Component Generator, the compilation will fail with syntax errors, saying that a data type identifier is expected, but is not present. This leads back to a class with defined attributes, but missing a `javaType` tag value.

For example, you might encounter this chain of events:

Rose Model	→	CMI File	→	Java Source	→	Compiler Error
ClassA.attributeA type= 'Logical View::java::utils::String'		ClassA.attributeA <javaType></javaType>		Class A public class Class { private attributeA = null; }		'...<identifier> expected: ...private ^ attributeA = null;

From this chain of events, you can see where the problem may have occurred and go fix it in the model.

To fix this problem:

1. Open your model file within Rational Rose.
2. Find the locations of the logical view references, as described above.
3. Re-enter the information within the dialog boxes, as appropriate.
 - For standard Java types, re-enter the Java type. For example, if you see
Logical View::java::lang::String
re-enter `java.lang.String` or simply `String`.
 - For a referenced class, enter the fully-qualified class name.
4. Save your model and re-export as XMI. You might want to perform a search on the XMI file, as you did in [Step 2](#).

Missing Default Values in CMI

When you accept default values during the modeling process, those values may not appear in the CMI file, and thus will not appear in your generated code.

To avoid this problem, follow these guidelines:

- **When a value must be entered:** Enter the value in the appropriate location, even if this value is the default.
- **When a value is chosen from a drop-down list:** Select a value which is *not* the default. Then select the default value.

Using the Business Component Generator

The Business Component Generator creates business components from your Rational Rose models. Refer to other chapters in this guide for instructions and tutorials on modeling the business components to be created through the Business Component Generator.

PREPARING FOR CODE GENERATION

Before you use the Business Component Generator, you must perform some setup tasks. If you will be starting with a Chordiant-provided base model that you have not yet customized, you can skip ahead to [“Using the Business Component Generator” on page 17](#).

Creating Your Model and Exporting to XMI

Before you use the Business Component Generator, you can either create your model in Rational Rose, or create a Business Component Generator project and then create your model using the Chordiant Tools Platform. Refer to other chapters in this guide for instructions on creating your models.

Export your model to XMI using the plugin specified in [“Configuring Rational Rose” on page 9](#). This file should be called `{model}.xmi`, where `{model}` is the name of your Rational Rose MDL file. This XMI file contains metadata for your model. For details on Chordiant metadata, refer to [Chapter 15](#).

Note: Be sure that your model contains *either* service or business object information. If your model contains both types of information, the Business Component Generator cannot generate valid output.

Creating the Descriptor File

When you create your model, you must also create a descriptor file for the Business Component Generator. This XML file tells the Business Component Generator what types of components to generate. This file must be named `{model}Descriptor.xml`, where `{model}` is the name of your Rational Rose MDL file.

Note: You must specify the components you want to generate in the descriptor file *before* you create your project. If you have already created a project and realize that you need to create additional components, you must create a new project with the correct descriptor file. Changing the descriptor file after you have created your project has no affect.

Chordiant-provided models already have descriptor files associated with them.

Descriptor File for Persistence Files

When you set a value to `true`, you tell the Business Component Generator that a specific type of code can be created from your model. If your model does not support a specific type of code, set the value to `false`. For a data model, you must set the `modeltype` value to `data`.

[Code Sample 3-1](#) shows the `JXCTutorialDescriptor.xml` file for the `JXCTutorial.mdl`.

```
<project-descriptor>
  <project-name>Extended Persistence Tutorial</project-name>
  <title>Extended Persistence Tutorial Model</title>
  <modeltype>data</modeltype>
  <cmi>true</cmi>
  <persistence>true</persistence>
  <business-tier>true</business-tier>
  <services>false</services>
  <schema>true</schema>
  <database>true</database>
  <version>5.7</version>
  <unit-tests>true</unit-tests>
</project-descriptor>
```

Code 3-1: JXCTutorialDescriptor.xml Descriptor File

Descriptor File for Service Model

The descriptor file for service models is slightly different. The `name` and `title` tags are still required. Set the `modeltype` value to `service`. All other values should be `false`, as shown in [Code Sample 3-2](#).

```
<project-descriptor>
  <project-name>MyService</project-name>
  <title>Example Service Customization</title>
  <modeltype>service</modeltype>
  <cmi>false</cmi>
  <persistence>false</persistence>
  <business-tier>false</business-tier>
  <services>true</services>
  <schema>false</schema>
  <database>false</database>
  <version>5.7</version>
  <unit-tests>false</unit-tests>
</project-descriptor>
```

Code 3-2: Sample `serviceDescriptor.xml` Descriptor File

If you will be using the Ant script to generate service components, you must modify this file slightly. For full details, refer to “[Advanced Topic: Creating Business Components Through Ant Scripts](#)” on page 32.

USING THE BUSINESS COMPONENT GENERATOR

This section details how to use the Business Component Generator. There are two options for code generation — a wizard-based option and an Ant script-based option.

Before You Begin

Turning Off Automatic Code Generation

You might want to change the automatic generation setting to save yourself time. Otherwise, you must wait for code to regenerate every time you make a change to the code.

To turn off the automatic code generation:

1. Launch the Chordiant 5 Tools Platform.
2. From the **Window** menu, select **Preferences**.

3. (Optional) In the left panel, select **Workbench**. Clear the **Perform build automatically on resource modification** checkbox. Click **Apply**.

Note: This will save you time and you can build projects at another time. (Refer to [“Manually Starting a Build Process” on page 30](#).) If you leave this option selected, your system might automatically perform Java builds multiple times. Be aware that your computer might not respond for a period of time while the code is generating. You might want to turn this option on again when you have completed the project.

Checking Classpath Variables

Before you begin using the Business Component Generator, you must set the classpath variables so the tool can find the Chordiant JARs.

To check the classpath variables:

1. In the left panel of the **Preferences**, select **Java**, then select **Classpath Variables**.
2. Check that these variables exist and are pointing to the appropriate places:
 - **CHORDIANT_RUNTIME**: points to the Chordiant EAR project.
 - **FOUNDATION_LIB**: points to the directory which contains the Chordiant JAR files.
 - **J2EE_LIB**: points to the appropriate JAR file for your application server:
 - `j2ee.jar` for WebSphere
 - `weblogic.jar` for WebLogic

If these JARs are not already in the **J2EE_LIB** directory, add them there first.

 - **THIRD_PARTY_LIB**: points to the directory which contains the third party JAR files.

Enabling Logging

By default, Error messages are written to the metadata LOG file. If you choose, you can also add Info messages.

To enable logging:

1. In the left panel, select **Data Model Preferences**. Select the **Log enabled** checkbox. Click **Apply**.

Info log messages will appear with the Error log messages in the metadata LOG file.

Allocating Memory and Resources

Address the following topics to ensure that your generation process goes smoothly.

Free Disk Space

In general, your free disk space should be at least twice the amount of your machine's physical memory. Your application server might crash when the hard drive becomes full.

Large XMI Files

If your XMI file is larger than 1 MB, you might want to increase the memory for your development environment before you start it up with the Chordiant Tools Platform. Add the following parameter to your WSAD or Eclipse startup command to increase the memory of your development environment.

```
-vmargs -Xms512m -Xmx512m
```

JDK 1.4 Users

Due to an issue with JDK 1.4 (Weblogic/Eclipse DE uses JDK 1.4), you must take the additional steps described in this section.

Specifying the JVM

When using WebLogic with Eclipse or any DE using JDK 1.4, you must explicitly specify which JVM to use. If you do not specify the JVM through the command line argument, Eclipse uses the first JVM found on the operating system's path.

To specify which JVM to use:

Use a `-vm` command line argument, such as:

```
eclipse.exe -vm c:\bea\jdk141_05\jre\bin\javaw.exe -vmargs -Xmx512M
```

Using the Proper Version of Xalan

To avoid using a beta version of Xalan that is packaged with Sun's JDK, use the Endorsed Standards Override Mechanism, described at <http://java.sun.com/j2se/1.4/docs/guide/standards/>.

To make sure you are using the proper version of Xalan:

1. Locate the following JAR files in the `{WORKSPACE}\ChordiantEAR\lib-ext` directory:
 - `xalan_2.6.0.jar`
 - `xerces_2.6.2.jar`
 - `xerces_xml_apis_2.6.2.jar`
2. Copy the JAR Files to the `{JAVA_HOME}\jre\lib\endorsed` directory, where `{JAVA_HOME}` is the install location of the runtime software.
3. Click **New** to create a variable called `FOUNDATION_LIB` that points to the directory which contains the Chordiant JAR files. Click **OK**.

Running the Business Component Generator

The Business Component Generator runs within the Chordiant 5 Tools Platform.

Notes: You must be in the **Resource** perspective of the Chordiant Tools Platform to generate code.

By default, all error messages are written to the metadata .log file. To add Info logging messages, refer to [“Preparing for Code Generation”](#), [Step on page 18](#).

To use the Business Component Generator:

1. Within the Tools Platform, choose to make a new project.

From the **File** menu, select **New**.

Select **Chordiant** | **Projects** on the left and **JX Business Components** on the right.

Click **Next** to continue.

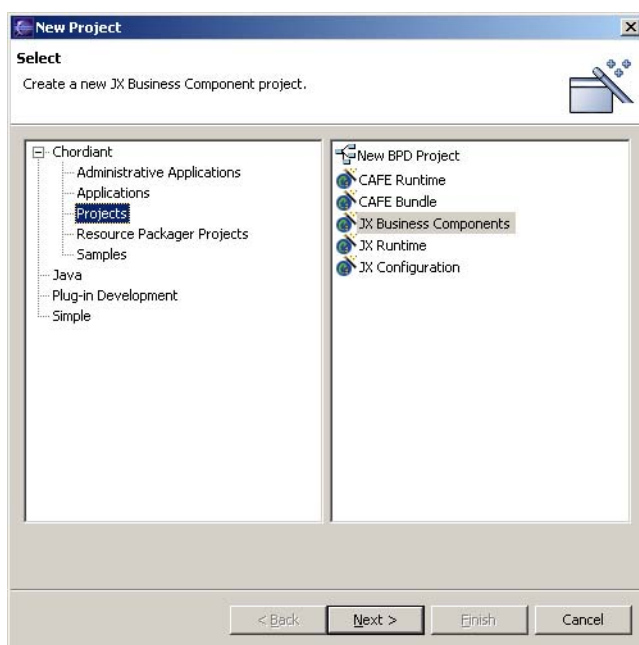


Figure 3-1: Creating a New Project for Service Generation

2. Specify the name of the new project to hold your generated code. Select the **Use Default** checkbox to use the default directory structure. Otherwise, clear the **Use Default** checkbox to create your own directory structure. Click **Next** to continue.

Notes: Do not use spaces in your project name. Spaces adversely affect the code generation.

You cannot change the name of a project once the project has been created. Refer to the note on [page 22](#) for implications.

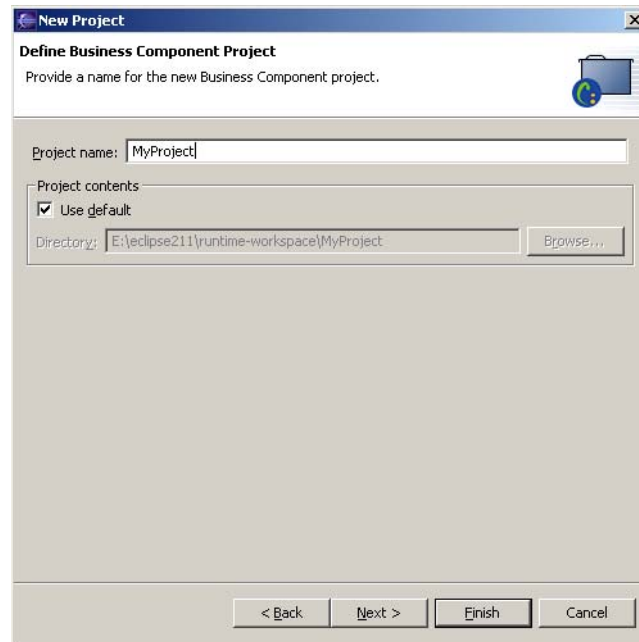


Figure 3-2: Specifying a Project

3. If your project refers to other projects, select those referenced projects now. Click **Next** to continue.

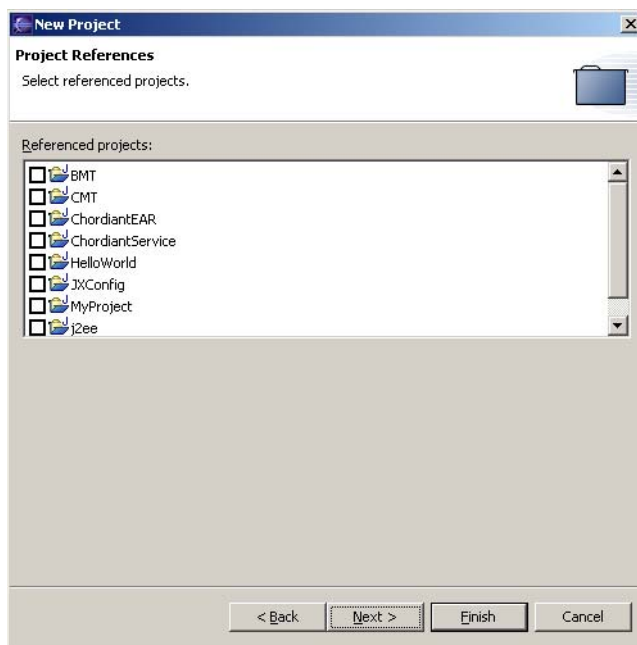


Figure 3-3: Specifying Referenced Project

4. Select the model to import. Select one of the following options, then click **Next** to continue.

Note: You cannot change the name of the model, its descriptor file, or its XMI file once they are a part of the Tools Platform project.

There are two approaches to this issue:

- 1). Use the original model name, but be sure to make individual project names meaningful so it is easy to distinguish your components.

2. Do not select a model from the data model plugin. Instead, copy the desired model from the plugin directory (`{eclipse_root}/plugins/{data_model_plugin}/rosemodels`) to another location where you can rename it, re-export the model as XMI, and change the name of the descriptor file. Then import the newly-named XMI file into the Business Component Generator, as described in [page 24](#).

- **A model from the data model plugin:** Select one of the models shipped with Chordiant, including the Chordiant Data Model and the tutorial models described in this guide.

Models included in the data model plugin are:

Chordiant BO Base Model: Includes the Chordiant Corporate Business Class from which you can create your own business objects. This process is described, in part, in [“Tutorial: Extended Persistence Components” on page 243](#).

Chordiant Data Model: Includes the Chordiant-provided data model with all of its business objects. You can use this model as a starting point for creating your own custom data model.

Chordiant Service Base Model: Includes the `BusinessDataServiceBaseClass` from which you can create your own services. This process is described in [“Tutorial: Creating and Extending a Business Service”, “Part I: Creating a New Service” on page 58](#).

Chordiant Service Model: Includes all Chordiant-provided business services, which are described in [“Chordiant 5 Application Components” on page 327](#).

Extended Persistence Tutorial Model: Includes creating a model which uses extended persistence components. Described in [“Tutorial: Extended Persistence Components” on page 243](#).

QueueItem Model: Includes the base model for customizing `QueueItems`. Refer to the *Chordiant 5 Foundation Server Customization Guide* for details on customizing `QueueItems`.

Service Creation Tutorial Model: Includes creating a service by subclassing the base class. Described in [“Tutorial: Creating and Extending a Business Service”, “Part I: Creating a New Service” on page 58](#).

Service Extension Tutorial Model: Includes extending the service you created in the Service Creation Tutorial. Described in [“Tutorial: Creating and Extending a Business Service”, “Part II: Extending an Existing Service” on page 76](#).

System Task Processor Service Model: Includes a base model for the system task processor. Refer to the *Chordiant 5 Foundation Server Business Process Server Developer’s Guide* for details on the System Task Processor Service.

- **An XMI file from a different location:** Specify the XMI file you want to import. You usually use this option if you are working with a model not provided by Chordiant or if you will be customizing a Chordiant model, but you want to change the model's name. Refer to the note on [page 22](#) for details.

Use the **Browse** button to locate this file or enter the file name directly. This file is usually *not* located within the `{eclipse_root}/plugins/{data_model_plugin}/rosemodels` directory, which is reserved for Chordiant-provided models.

Note: The XMI file option requires both an XMI file and a descriptor file. If you specify the name of the XMI file as `{model}.xml`, you must also have a descriptor file, named `{model}Descriptor.xml`, in the same directory.



Figure 3-4: Selecting the Data Model to Import

5. Specify which components you want to generate from your data model.

Notes: If you are generating service framework code, you will not see this screen. This screen is only used for data models. Models you specify as service models in the descriptor file will generate all of the service framework components.

If you choose, you can use Ant-based scripts to generate the business components. In this case, clear all of the checkboxes and click **Finish**. Then proceed to the instructions in [“Advanced Topic: Creating Business Components Through Ant Scripts”](#) on page 32.

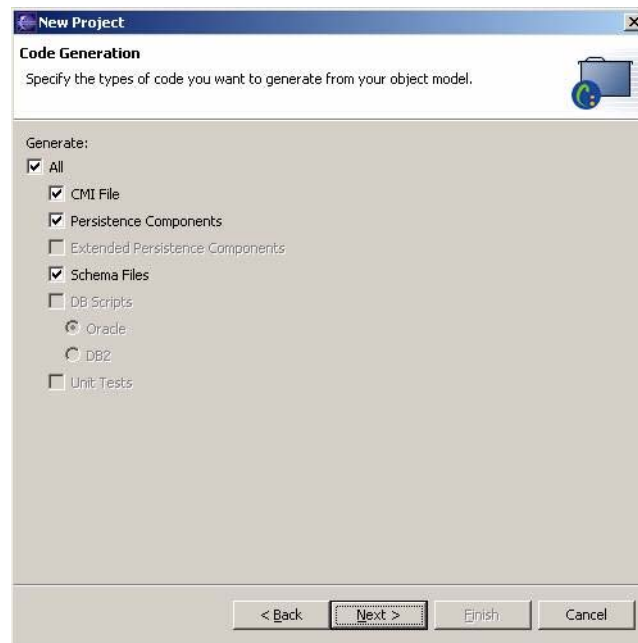


Figure 3-5: Specifying Components to Generate

Choose from the following options for your data model:

All: Creates all of the components that are available, based on the model descriptor file you created. For more information on the model descriptor file, refer to [“Creating the Descriptor File” on page 16](#).

CMI File: Creates the CMI file, a Chordiant-proprietary XML file from which you can create application components.

Persistence Components: Creates the business object, business object behavior (skeleton), business object criteria (BOC), and data accessor components.

Extended Persistence Components: Creates the business object, business object behavior, helper, access strategy, and `accessStrategyInput` components, as well as the `BehaviorFactoryConfiguration.xml` and `ValidatorFactoryConfiguration.xml` files.

Schema Files: Creates XSD files. (See Notes below.)

DB Scripts: Creates SQL files to populate your Oracle or DB2 database. The SQL script will need some modification to run in your database environment.

Unit Testers: Creates JUnit testers (skeleton) for testing your code.

Notes: Some options might not be available because those capabilities in the model are either not specified in the descriptor file, or are specified as **false**. If the author of the model did not build that capability into the model, the Business Component Generator cannot generate the corresponding code.

If you are using WebLogic with JDK 1.4, use the Ant script method to create XSD schema files. Refer to [“Advanced Topic: Creating Business Components Through Ant Scripts” on page 32](#) for details.

- Specify the Java settings for this project.

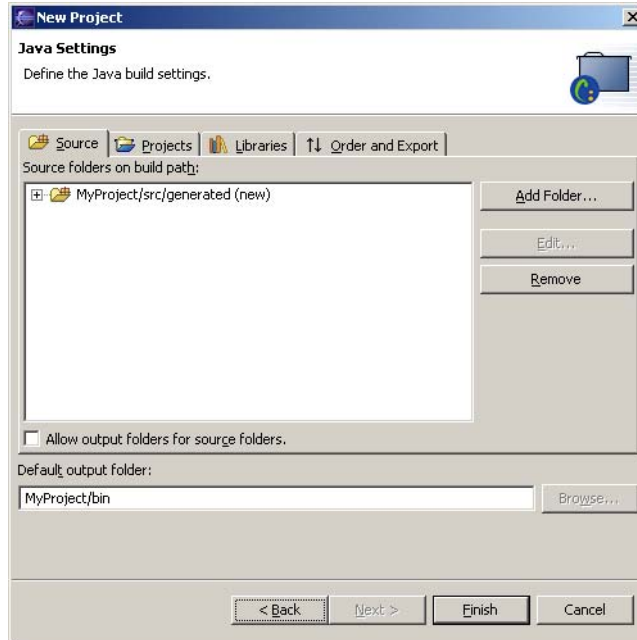


Figure 3-6: Specifying Java Settings

On the **Source** tab, select the default options to generate standard, unmodified code. By default, the Java project is set up with the `src/generated` folder input and `bin` output.

Note: You can add folders to the `src/generated` folder, but do not remove this folder. Code cannot be generated without this folder.

If you have made customizations, point to the directory containing your customized code.

7. Select the **Libraries** tab.

The libraries required are pre-configured for the development environment. Since you are customizing the model, you may require additional JARs. You can add or point to another JAR, but do not remove the pre-configured entries.

Note: Some customized code cannot be compiled in isolation from generated Java code. Both generated and customized code should be compiled together.

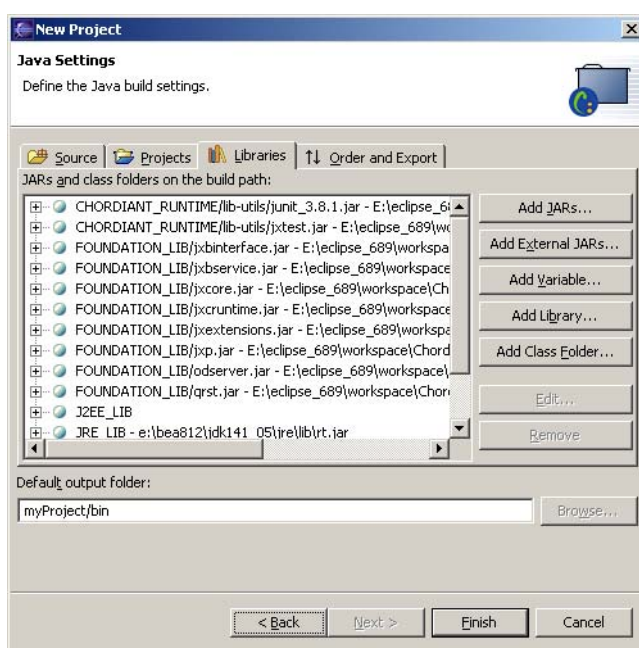


Figure 3-7: Specifying Library Files

8. Click **Finish**. Your components will generate automatically and be placed into the following directories:

Note: If you turned off the automatic build option in [Step 3](#), refer to “[Manually Starting a Build Process](#)” below.

- **bin:** Normal Java project output folder to contain `.class` files.
- **output:** Contains the following directories:
 - cmi:** Contains the generated CMI file.
 - config:** Contains the generated configuration files.
 - db:** Contains generated database scripts.
 - lib:** Contains JAR files. The Application Packaging Manager (APM) will include these files in the EAR and also update the EJB manifest classpath
 - smi:** Contains the generated SMI files for defining business services.
 - xmi:** Contains the scripts for converting CMI to XMI.
 - xsd:** Contains the generated schema files.

The output directory is used as the staging directory for these projects. If you want to create a JAR file for your code, create the JAR and place it in the `output/lib` directory. Refer to the *Chordiant 5 Tools Platform Getting Started Guide* for more information on staging.
- **rosemodels:** Contains copies of your `{model}.mdl`, `{model}.xml`, and `{model}Descriptor.xml` files, along with a copy of the `uml.dtd` file, required for validating the XMI file you created.
- **src/generated:** Specified in the `.classpath` file as the root of the generated Java source. Generated Java files are placed in this directory and its subdirectories corresponding to the Java packages.

Note: If you modify any of the generated components, be sure to move them out of these directories. Otherwise, if you run the Business Component Generator again, your modified components will be overwritten.

Manually Starting a Build Process

If you turned off the automatic build in [Step 3 on page 18](#), when you finish running a project, you will have mostly empty directories in your project. The only directories with contents will be:

- **rosemodels:** contains copies of your `{model}.mdl`, `{model}.xml`, and `{model}Descriptor.xml` files
- **output/cmi:** contains an XSL file. Do not modify this file.

At this point, you can either manually start a build process, described below, or you can generate components through the Ant script, described in [“Advanced Topic: Creating Business Components Through Ant Scripts” on page 32](#).

To manually start a build process if you have turned off the automatic rebuild:

1. Open the Chordiant 5 Tools Platform.
2. In the Navigator view, select the project you want to rebuild.
3. From the **Project** menu, select **Rebuild project**. This builds all of the code in your project.

Note: If you modify any of the generated components, be sure to move them out of these directories. Otherwise, if you run the Business Component Generator again, your modified components will be overwritten.

To specify which components to generate during a manual build:

1. In the **Resource** perspective, select the main folder of your project. From the **Project** menu, select **Properties**.
2. On the left side, select **Data Model Properties**. On the right side, select the checkboxes for the available components you want to generate. For example, if you only want to generate XSD scripts, select only that checkbox. When you have made your selections, click **OK**.

Note: If your project contains a service model, you will not see the **Data Model Properties** option available. When you rebuild the project, the service files will automatically be generated.

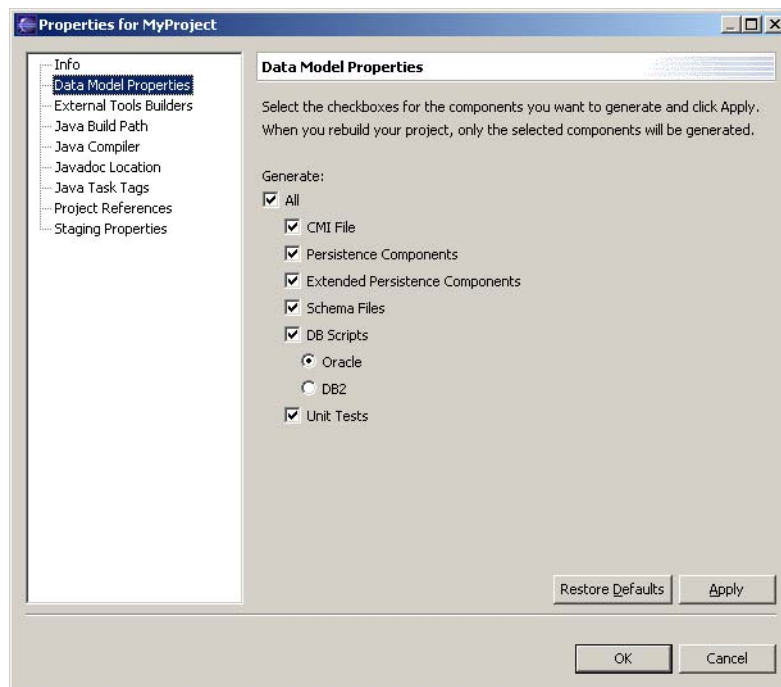


Figure 3-8: Selecting Components to Generate

3. Right-click on the main folder for your project and select **Rebuild project**. This builds the components you specified in [Step 2](#) of this process.

ADVANCED TOPIC: CREATING BUSINESS COMPONENTS THROUGH ANT SCRIPTS

As an alternative to the Business Component Generator, you can generate business components through Ant scripts. These Ant scripts perform the same functions as the Business Component Generator, but some may find them more convenient. Ant-based generation provides enhanced customization and debugging of the code generation process. It also provides a method of code generation outside the Chordiant Tools Platform.

Before You Begin

Complete these tasks before you begin using the Ant scripts for code generation:

1. Follow all of the instructions in [“Before You Begin” on page 17](#). These steps are important regardless of whether you are using the Business Component Generator or are running Ant scripts.
2. Make sure that the proper version of Ant and your Java runtime environment are installed. If you will be using the Ant scripts from the command line instead of through the Chordiant Tools Platform, ensure that you have registered Ant with the operating system environment. The version used by the operating system should match the version used by the Tools Platform. Refer to the Supported Platforms document that accompanies the Release Notes.
3. Make sure `{JAVA_HOME}` is defined properly in your operating system environment properties and that the correct version of Java is at the front of the `path` system property.

4. If you are using JDK 1.3.1, make sure that these JARs are loaded in Ant's classpath: `xalan_2.6.0.jar` and `xerces-xml-apis_2.6.2.jar`.
This step is not required for users of JDK 1.4.
 - **For command line:** Copy these JARs from the `/ChordiantEAR/lib-ext` directory and paste them into the `{ANT_HOME}/lib` directory.
 - **For the Chordiant Tools Platform:** Add these JARs to the Ant plug-in using the **Windows | Preferences** menu. As shown in [Figure 3-9](#), select the **Ant | Runtime** preferences in the left panel. Click **Add JARs** to reference the above JARs in the `ChordiantEAR/lib-ext` directory.

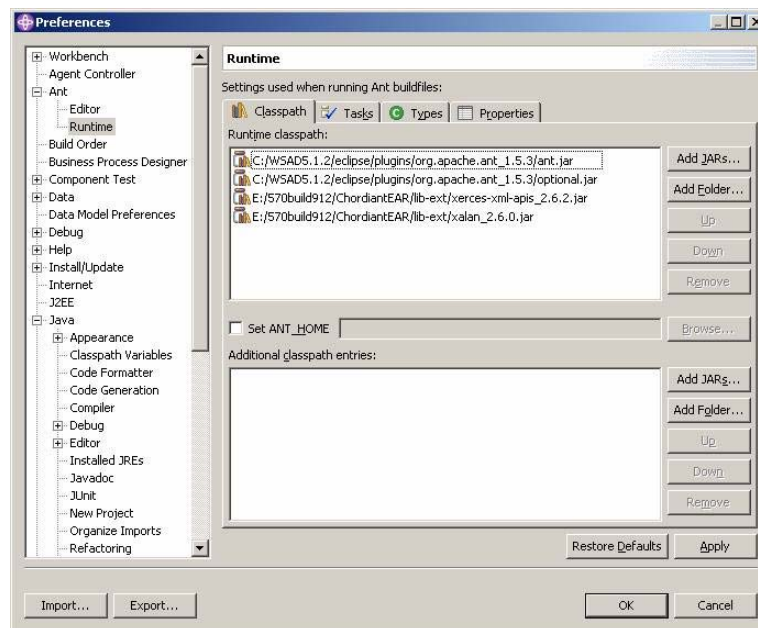


Figure 3-9: Adding JARs to the Ant Plug-in

5. Clear the **Perform Builds Automatically** checkbox, as described on [page 17](#).

6. Disable automatic generation for business components. This step is not needed for service models. For service models, skip to [Step 7](#).
 - a. Right-click the **JX Business Components** project and select **Properties**.
 - b. In the **Properties** window, select **Data Model Properties** on the left. Clear all of the components on the right side, ignoring the **All** checkbox. Click **OK**.

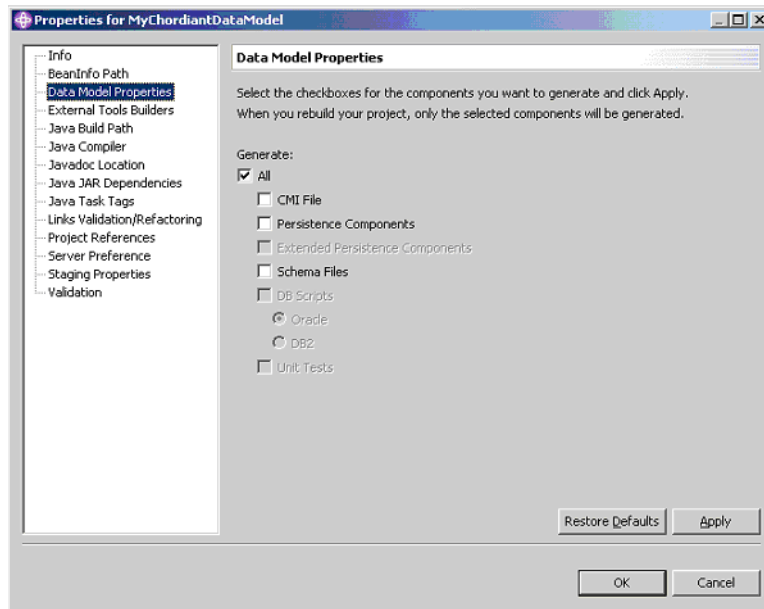


Figure 3-10: Disabling Automatic Generation of Business Components

7. For service models, you must alter the descriptor file to override the automatic code generation within the Business Component Generator. Change the value of the `modeltype` element to `antgeneratedservice`, as shown in [Code Sample 3-3](#).

If you have created your own model, you can modify that descriptor file. You can also make this modification to service model descriptors found in the Chordiant data model plug-in directory. This will disable automatic class generation for any project that imports these service models from the data model plug-in.

```
<?xml version="1.0" encoding="UTF-8"?>
<project-descriptor>
  <project-name>Service Creation Tutorial</project-name>
  <title>Service Creation Tutorial Model</title>
  <modeltype>antgeneratedservice</modeltype>
  <cmi>>false</cmi>
  <persistence>>false</persistence>
  <business-tier>>false</business-tier>
  <services>true</services>
  <database>>false</database>
  <schema>>false</schema>
  <unit-tests>>false</unit-tests>
</project-descriptor>
```

Code 3-3: Modifying the Descriptor File for Ant Script-Based Code Generation

Running the Ant Script to Create Business Components

The `ctpbuid.xml` Ant script offers class generation for each of the three data model types — persistence, extended persistence, and service.

When you create your **JX Business Components** project in the Chordiant Tools Platform, this script and its associated properties file are automatically copied into the root level of your project.

To begin the process of generating business components:

1. If desired, ensure that you have disabled your project from auto-generating classes within the Chordiant Tools Platform. This process is described in [“Before You Begin” on page 17](#).
2. In the `ctpbuid.xml` file, located in your **JX Business Component** project, customize the `_init` target to include any new JAR variables in its `jx.classpath` path element. Make sure that the path sets the variables in the order that you want class loading to occur. We recommend that the order of the JARs in the `jx.classpath` path element should emulate the order of the JARs in the project's `.classpath` file.
3. Modify the `ctpbuid.props` file, located in your **JX Business Component** project.

These two properties can also be expressed as `-D` parameters in the Ant script (`-Dmodel` and `-Dstylesheet`). Modifying the `ctpbuid.props` file saves you from specifying the `-D` parameters each time you run the Ant script.

- a. **model file property** — the name of the XMI file exported from the Rose model.

`model={modelname}.xml`

This XMI file is presumed to be in the `rosemodels` directory, so you do not need to specify the fully-qualified name of the XMI file.

- b. **stylesheet property** — should point to the correct location, usually the generator directory in the Chordiant data model plug-ins directory.

`stylesheet={eclipse_root}/plugins/{data_model_plugin}/generator`

If you do not provide the `stylesheet` property in the `ctpbuid.props` file, you must manually copy the generator directory into the project directory.

4. In the `ctpbuid.props` file, include any additional JARs required to compile the project's classes by creating additional variables in the `ctpbuid.props` file to reference the JARs.

You might also need to update `jar.j2ee` to point to `weblogic.jar` (for WebLogic) or to `j2ee.jar` (for WebSphere).

Once you have completed up through [Step 4](#), you can run the Ant script:

- [“Within the Chordiant Tools Platform”](#)
- [“From the Command Line”](#)

Within the Chordiant Tools Platform

To run the Ant script from the Chordiant Tools Platform:

1. Within your **JX Business Component** project, locate the `ctpbuid.xml` file.
2. Right-click the file and select **Run Ant**.
3. Select the checkbox for the target script you want to run. Targets are described in [“Targets for Ant-Based Generation” on page 37](#).
4. Click **Run**.

Your code is generated and placed in the directories described in [Step 8 on page 29](#).

From the Command Line

To run the Ant script from the command line:

1. Open a command line.
2. Make sure you are in the root directory of your **JX Business Component** project.
3. Type the command in this basic pattern:

```
ant -f ctpbuild.xml {targetname}
```

Additional examples are provided in [“Examples of Ant Script Commands”](#).

Your code is generated and placed in the directories described in [Step 8 on page 29](#).

Examples of Ant Script Commands

- Basic example. No parameters needed if the model property is set in the `ctpbuid.props` file.

```
ant -f ctpbuild.xml genAllService
```
- Using the `-Dmodel` property:

```
ant -f ctpbuild.xml -Dmodel=JXCTutorial.xml genPersistence
```
- Using the `-Dmodel` and `-Dstylesheet` properties:

```
ant -f ctpbuild.xml -Dmodel=JXCTutorial.xml -Dstylesheet=E:/Chordiant/generator  
genPersistence
```
- Retrieving a list of all available targets with their descriptions:

```
ant -f ctpbuild.xml -projecthelp
```


Targets for Ant-Based Generation

This section describes the targets available for the Ant scripts, whether run from the Chordiant Tools Platform or from the command line.

For each model type there are two choices:

- Generation of the classes files, with subsequent compilation and packaging the classes into a JAR file
- Generation of class files only

For extended persistence models, there is an additional target for generating JUnit-based tester classes.

Persistence Generation Targets

- **genAllPersistence** — Generate, compile, and JAR functions for the persistence data model
- **genPersistence** — Generate the classes for the persistence data model (no compiling or packaging as JARs)

Extended Persistence Generation Targets

- **genAllExtPersistence** — Generate, compile, and JAR functions for the extended persistence data model
- **genExtPersistence** — Generate the classes for the extended persistence data model (no compiling or packaging as JARs)
- **genAllExtPersistenceTesters** — Generate and compile all JUnit testers (for extended persistence data model only)

Service Generation Targets

- **genAllService** — Generate, compile, and JAR functions for service models.
- **genService** — Generate the classes for the service model (no compiling or packaging as JARs)

Additional Targets

These targets do not provide full data model class generation and some might not relate to all types of data models in all situations.

- **clean** — Performs a clean operation, removing all code and temporary files from generation.
- **cmi2xmi** — Converts Chordiant CMI file into Rational Rose-specific XMI for UML 1.1 import.
- **compile** — Compiles the application.
- **compile_debug** — Compiles the application with debug information
- **createDB2Schemas** — Creates DB2 schemas
- **createOracleSchemas** — Creates Oracle schemas
- **jar** — Builds the components and packages them into a JAR file.
- **jar_debug** — Builds the project with debug information and packages it into a JAR file.
- **usage** — Displays a usage summary.
- **xmi2cmi** — Converts XMI into CMI, for use in generating files from a persistence or extended persistence model
- **xmi2smi** — Converts XMI into SMI, for use in generating files from a service model

DEPLOYING YOUR APPLICATION COMPONENTS

Once you have created and customized your application components files, you must deploy them so they are available.

Creating a JAR File

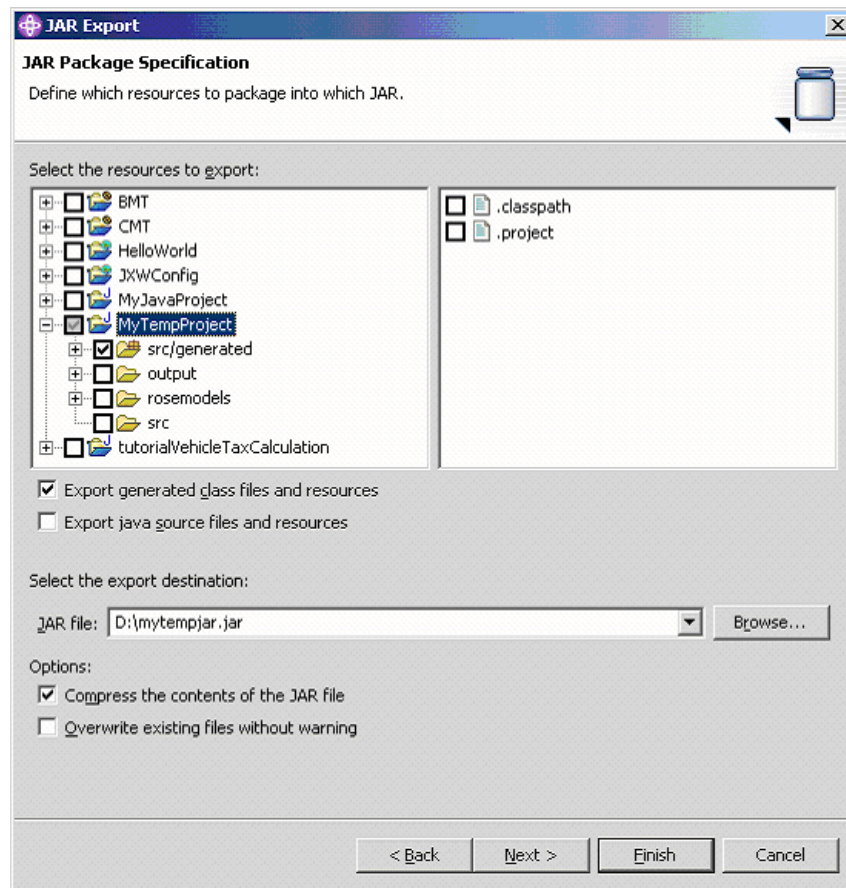
Generating a JAR file of your business component code is slightly different than for other projects.

Note: This is only required when you have used the Business Component Generator wizard interface. The Ant script has a target for generating JAR files.

To create a JAR file of your business component code:

1. In the Resources view, right-click on your project. Select **Export**, then select **JAR**.
2. Select and expand your project. Clear the checkboxes for all folders, except the ones containing your source code. This is usually `src/generated`, but can be a different location if you have moved your source.

Clear the **.classpath** and **.project** checkboxes on the right side.



3. Clear the **Export java source files and resources** checkbox.
4. Click **Finish** or continue the rest of the JAR export process as normal.

Once you have created your application components, you must deploy them. The Application Packaging Manager (APM) packages your files into an EAR file. Refer to the *Chordiant 5 Applications Deployment Guide* for details on using the APM.

Creating or Modifying Business Services

You can create new service framework components from your Rational Rose service model by using the Business Component Generator.

Note: The Business Component Generator creates business components from your Rational Rose model. For instructions, refer to [Chapter 3, “Using the Business Component Generator”](#).

Included with your Chordiant 5 Foundation Server installation is a Chordiant service base model for Rational Rose. You can copy it and use it as a starting template. The JXBServiceBase model is a Chordiant base business service model.

This model, which is included with this release, is located in `{eclipse_root}/plugins/{data_model_plugin}/rosemodels`.

FROM MODEL TO SERVICE COMPONENTS

Chordiant's model-driven architecture enables you to make an abstract service model and generate different types of physical components from it. [Figure 4-1](#) illustrates how you can create Java and WSDL (Web Service Descriptor Language) bindings from a single model to create:

- **Client Agents**, Java/J2EE binding. Client Agents are described in this chapter. Client agents interact with other service framework components, described in the next section, [“Business Service Framework Components Created”](#) on page 43.
- **Web Services**, WSDL binding. Web services are described in detail in [Chapter 6, “Creating Web Services”](#).
- **Business Process Designer Task Descriptors**, WSDL binding with additional Chordiant-specific information. Creating task descriptors is described in [Chapter 8, “Creating Task Descriptors”](#). Incorporating task descriptors into your workflow is described in the *Chordiant 5 Tools Platform Business Process Designer Developer's Guide* and in the *Chordiant 5 Foundation Server Business Process Server Developer's Guide*.

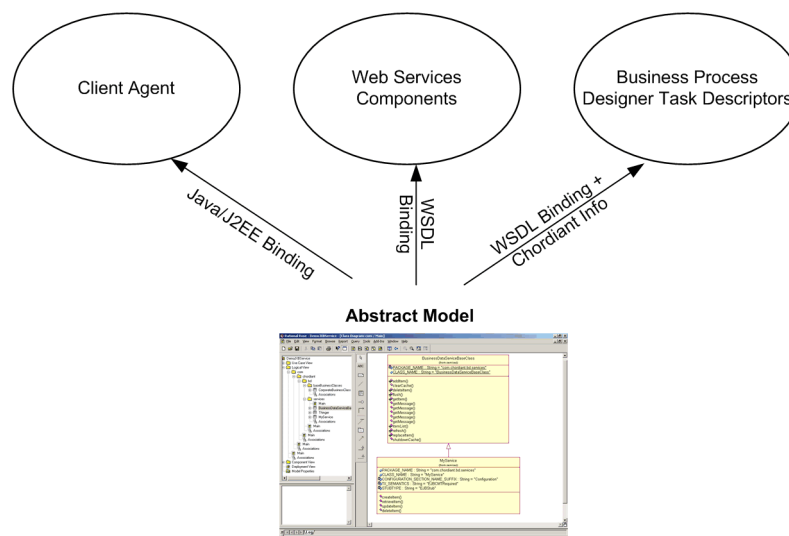


Figure 4-1: One Abstract Model Creates Different Service Components

BUSINESS SERVICE FRAMEWORK COMPONENTS CREATED

These components are associated with specific business services.

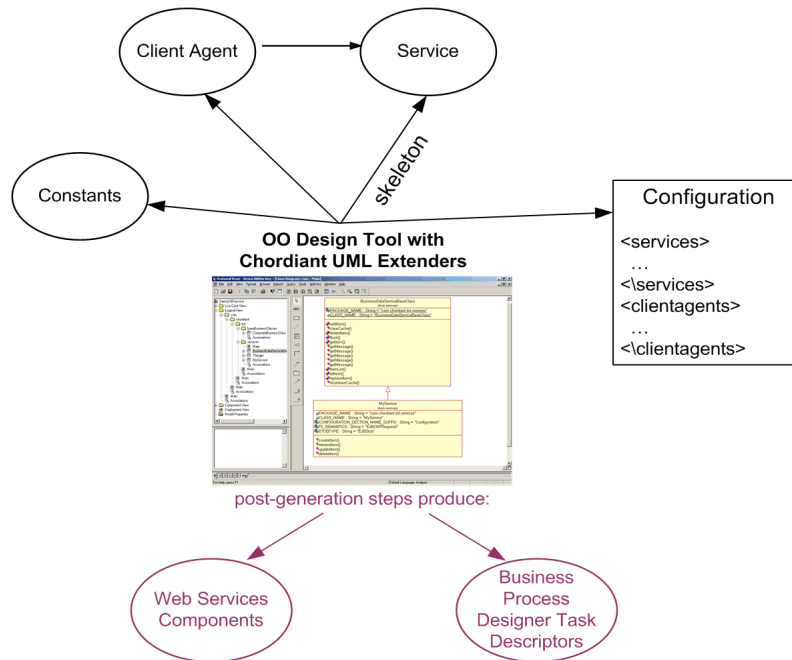


Figure 4-2: Business Service Framework Components Created Automatically

Note: With the exception of the business service skeleton, the Java files described in this section offer a full implementation of the component and are ready to use without modification. For full discussion of the components, refer to the *Chordiant 5 Foundation Server Developer's Guide*.

- **Client Agent** — `{yourservice}ClientAgent.java`
The client agent file is a full implementation of a client agent.
If you will be doing callbacks, you must add code to the generated client agent.
- **Constants** — `{yourservice}Constants.java`
The constants file contains a full complement of constants for the service name, all function names, and all parameter names.
- **Business Service Skeleton** — `{yourservice}.java`
The business service created is a code skeleton that is based on the model you created in your design tool. The methods and general infrastructure are present, including the implementation of the `processRequest` method and the ability to handle overloaded methods, but you must fill in the business logic of the methods using Java code.

- **Configuration File** — `serviceconfig.xml`

This configuration file registers the service with the system. For details on configuration files, refer to the *Chordiant 5 Foundation Server Developer's Guide*.

Note: By default, the generated service includes a Business Object Resource Manager, so your service can interact with a database. You must add the resource manager configuration parameters to the configuration file. Refer to the *Chordiant 5 Foundation Server Developer's Guide* for details on these configuration settings.

If your service does not interact with a database, override the `setup` method in your customized class so the Resource Manager is not initialized.

After generating the standard service framework files through the Business Component Generator, you can also create:

- **Web Services Files** — `{yourservice}.wsdl` and `{yourservice}.wsdd`

These files enable others to access your services through the web. For details on creating WSDL files and working with them, refer to [Chapter 6, "Creating Web Services"](#).

- **Business Process Designer Task Descriptor Files** — `{yourservice}.wsdl`

These files enable you to use your services as tasks in the Business Process Designer. For details, refer to [Chapter 8, "Creating Task Descriptors"](#) and to the *Chordiant 5 Tools Platform Business Process Designer Developer's Guide*.

BUSINESS SERVICES OVERVIEW

Tip: Chordiant services are described in-depth in the "Creating Foundation Server Components" chapter in the *Chordiant 5 Foundation Server Developer's Guide*. This section describes some highlights of business services.

Business services implement domain and application-specific logic within your enterprise. Services operate at a higher level than business objects, and generally function as a controller for coordinating calls to methods that implement business object behavior.

All Chordiant business services have a `processRequest` method, which serves as the common public entry point and dispatcher to the service.

Note: If you create all of your service framework components through the Chordiant Business Component Generator, you should never have to modify a service's `processRequest` method. Refer to ["Using the Business Component Generator" on page 15](#) for detailed instructions.

Most Chordiant services are located in the `com.chordiant.bd.services` package. The `PartyRole` service is located in the `com.chordiant.pmf.service` package and the `PmfCustomer` Service is located in the `com.chordiant.customer.service` package. You should store any newly-created services in different packages, so you can distinguish them easily.

The specific implementation of your business service methods will necessarily be domain-specific, however, here is a list of common operations typically performed in business service methods:

- Call another service method (service to service call)

You can call a peer service to have it perform required work. When performing a service to service call, the calling business service uses the `ClientAgentHelper` to request a client agent for the peer service. The calling service then uses a client agent to make service to service calls.

Refer to the “Implementing a Service to Service Call” section of the *Chordiant 5 Foundation Server Developer’s Guide* for more information about implementing service to service calls.

- Perform a callback to a client application

Callbacks enable services to request client applications to perform work. To enable a callback, a client application needs to register and receive a Network Presence Key, and then communicate the key to the service.

Once it has a Network Presence Key, a service can then use the `ClientAgentHelper` to create a client agent and then invoke methods on the client agent which will result in callbacks to the client endpoint.

You must provide a specific implementation in the client application (for the specified service name and function name) to actually process the callback. More information on how to implement callback processing for thick clients can be found in the “Implementing a Callback” section of the *Chordiant 5 Foundation Server Developer’s Guide*, and more information on how to implement callback processing for thin clients can be found in the “Network Presence” section of the *Chordiant 5 Foundation Server Developer’s Guide*.

The service uses the Network Presence Key to obtain a handle to the correct client agent using the `ClientAgentHelper`. For more information about implementing client application callbacks, refer to “Implementing a Callback” in the *Chordiant 5 Foundation Server Developer’s Guide*.

- Perform transactions

You can use transactions explicitly or implicitly to control data integrity (for example, two-phase commit) in underlying XA-compliant data stores. Chordiant provides for services that employ both Container Managed Transactions (CMT) or Bean Managed Transactions (BMT), as specified by J2EE.

For more information about transactions, refer to “Performing Transactions” in the *Chordiant 5 Foundation Server Developer’s Guide*. You should use transactions with every operation that modifies the database. However, you do not need to use transactions with read-only operations.

- Enterprise Business Logic

Ultimately, services embody the business logic of the system. This logic is manifested as Java code that you must implement.

For example, a common business logic activity is the generation of unique numbers. You can have the service automatically generate unique numbers, such as order numbers and customer numbers. Note that these are numbers are *not* IDs; they are business domain numbers. You can also choose to use methods other than the ones described here to generate numbers as long as the algorithm produces unique values. For more information on number generation, refer to [“Performing Business Domain Number Generation” on page 52](#).

MODIFYING SERVICE FRAMEWORK COMPONENTS

You are encouraged to use existing services as the basis for your customization. To ensure forward compatibility and to make sure that you are getting the most from the JX Architecture, we strongly urge you not to alter the existing code, but rather to subclass it and make your modifications and overrides within your new class.

You might want to extend a service:

- To update the capabilities of an existing service methods
- To add new interface methods to an existing service
- To invoke new business object behavior in customized BOB objects

For a discussion of the different BOB objects, refer to [Chapter 10, “Extended Persistence Components”](#) or [Chapter 14, “Customizing the PartyRole Service”](#)

To modify an existing component, use the Chordiant 5 UML Extender for Rational Rose and follow these general steps:

1. Open your model and locate an existing component that resembles your required functionality. You can find one listed in [Chapter 16, “Chordiant 5 Application Components”](#) or you can use one that you might have already created yourself. You can choose to copy the `JXBServiceBase.mdl` provided with this release and use it as a starting point. It contains a base service object.

This model is located in
`{eclipse_root}/plugins/{data_model_plugin}/rosemodels.`

Notes: You can open the model from anywhere or create a new project through the Business Component Generator, using this model. You can then customize the model from within your project.

You cannot change the name of the model, its descriptor file, or its XMI file once they are a part of the Tools Platform project. Refer to the note on [page 22](#) for details.

Always try to isolate your customizations from the Chordiant-provided components in a separate package, and if possible, a separate diagram.

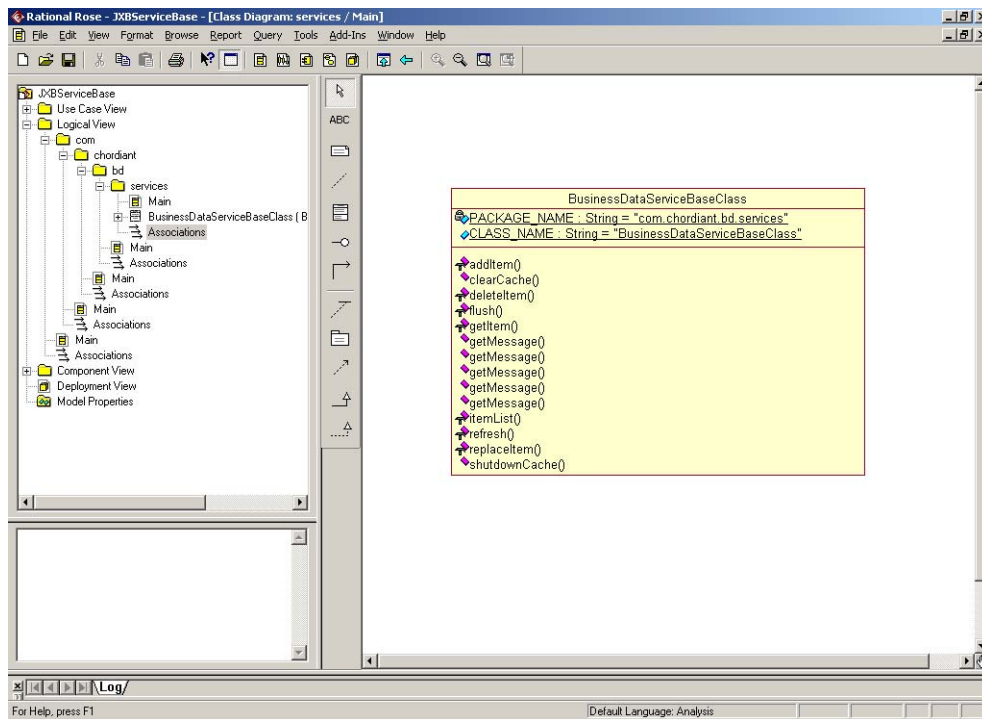


Figure 4-3: JXBSERVICE Model

2. Subclass the service.
3. Add or override specific attributes within the derived class using the standard Rose **Attribute** tab.

Notes: Refer to “Rose Models” and “Business Service Definition” within the “Customization Concepts” on [page 55](#) for other rules to follow when modeling business services.

Figure 4-4 illustrates a sample service customization.

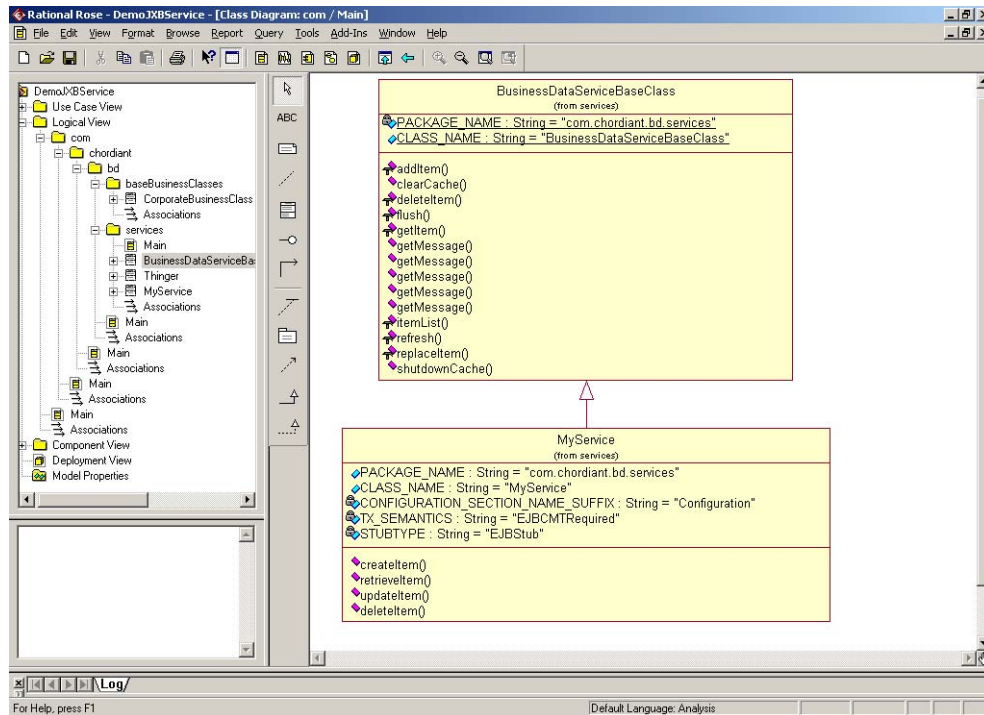


Figure 4-4: MyService Extending from Business Data Service Base Class

You must have these attributes in your service, as shown in Figure 4-4:

- **PACKAGE_NAME** (public, static): Specify the class package name for your service.
- **CLASS_NAME** (public, static): Specify the class name for your service.
- **CONFIGURATION_SECTION_NAME_SUFFIX** (private): Specify the service configuration suffix.
- **TX_SEMANTICS** (private): Specify your service transaction semantics. The value is either EJBMT or EJBCTRequired
- **STUBTYPE** (private): Specify your service stub type, use EJBStub for a client agent connecting to an EJB.

Refer to [page 322](#) of the “Metadata” chapter for details on the last two attributes dealing with transactions. Additional information on transactions is available in the *Chordiant 5 Foundation Server Developer’s Guide*.

You must also include these two attributes on each service, for security purposes:

- **username**: the name of the user.
- **authenticationToken**: acquired from the Security service.

4. Add or override specific behavior within the derived class using the standard Rose **Operation** tab.

Notes: DO NOT remove methods from existing services.

You cannot create new methods whose names are substrings of standard methods that already exist in the service base class. These standard methods are: **processRequest**, **setup**, **status**, **reinitialize**, **shutdown**, and **initResourceManager**. For example, you cannot create a new method called **process**, because a method with the name **processRequest** already exists. Similarly, you cannot create a new method called **set** or **reinit** because these are also substrings of these standard base class methods.

Only public APIs are generated into client agent APIs and the Constant definitions class, as well as automatically included in the generated **processRequest** implementation. Non-public APIs will still be in the generated service class, but will not be present in the generated client agent.

5. Export the model to XMI. In Rational Rose, from the **File** menu, select **Export UML 1.1 to XMI**. When prompted, select to export your data as **ASCII**.
6. Use the Business Component Generator to create the business service skeleton, client agent, configuration file, and constants class.

Note: You must also create a specialized descriptor file before using the Business Component Generator.

For instructions, refer to [Chapter 3, "Using the Business Component Generator"](#), including ["Creating the Descriptor File"](#) on page 16.

7. In your new Chordiant Tools Platform project, create a new directory where you will create your customized code. You might want to call it `custom`, or something similar to distinguish it from the generated code in the `src` directory.
8. In the `custom` directory, create a new class that subclasses from the services skeleton generated in the `src/generated` directory. You will make your modifications to this class instead of to the generated class.

Make sure that your customized class provides an implementation for each API defined in the generated skeleton and client agent class.

Note: Remember that when you rebuild the project, the code in the `src/generated` directory will be overwritten. By subclassing from the generated service file, you will get any updates from regenerating the service in your extended service, but you will not lose any of your customized code.

There is no need to subclass the client agent or constants class code. The configuration file requires a small change, as described in [Step 12 on page 50](#).

9. Add or modify code in the business service.

Note that the generated code will define empty methods for the service. If you want to modify an existing method from the parent class, copy that method into the new code and make your additions or modifications there. You can also add brand new methods.

10. You might choose to call another service to do some of the work. Add your service to service calls within this service.
11. Most of your logic will likely reside within the business service. If you choose, you can move some logic outside of the service into a business object behavior. If you are not using business object behaviors, proceed to [Step 14](#).

If you are using business object behavior (as described in [Chapter 14, “Customizing the PartyRole Service”](#)), call methods on BOB objects to invoke behavior on business object data.

You must retrieve the BOB object using the Object Factory before invoking methods on BOB objects. You can use the `getBehaviorForName` method on the Resource Manager to retrieve a BOB object from the Object Factory.

The Object Factory is aware of metadata for business object behavior, and always returns the customized object, if available.

You should always use the Object Factory to return instances of a business object, data accessor, business object criteria object, or business object behavior object. Do not create new instances of these objects using the `new` operator.

12. Update the configuration file with the classname for your extended service. Currently, that file points to the generated service skeleton, not to your extended service that you created in [Step 8 on page 49](#).
13. Check that the configuration file has the correct fully-qualified classname for the `clientagents` entry. If not, modify it.

14. Move the configuration file from the `{WORKSPACE}/{project}/output/config` directory (where it was generated) into the `{CHORDIANT_ROOT}/config/Chordiant/components` directory, so the parameters of the service are visible to the application server.

`{CHORDIANT_ROOT}` corresponds to the `chordiant.configuration.configurationRootDirectory` parameter in your application server.

Since there might already be a configuration file in that directory named `serviceconfig.xml`, you might want to rename this XML file. By convention, give the XML file the same name as your service.

If you create more than one service using the Business Component Generator, each service will be listed in this configuration file, while this XML file is within the `{WORKSPACE}/{project}/output/config` directory. Once you move the file out of this directory, a new `serviceconfig.xml` file will be created the next time you create service files through the Business Component Generator.

Notes: By default, the generated service includes a Business Object Resource Manager, so your service can interact with a database. You must add the resource manager configuration parameters to the configuration file. Refer to the *Chordiant 5 Foundation Server Developer's Guide* for details on these configuration settings.

If your service does not interact with a database, override the `setup` method in your customized class so the Resource Manager is not initialized.

The configuration file located within your project will be overwritten if you rebuild your project. So be sure to make any changes, or move any changed files, outside of that directory.

Overloading in Services

When you create overloaded methods in services, it is important to keep track of the different methods. Chordiant uses a standardized naming convention to keep track of the differences among the methods. When the Business Component Generator creates the service skeleton, the client agent, and the constants class, the input parameter becomes part of the constant for the method name, making it easy to distinguish among the various method forms.

For example, if you have two forms of `myApi`, you will have two entries in the constants class, as shown in [Code Sample 4-1](#). One form takes one `String` input parameter, the other takes two `String` inputs.

```
...FN_myAPI_String = "myAPI_String";
...FN_myAPI_String_String = "myAPI_String_String";
```

Code 4-1: Code for Overloaded Services

This standardized naming convention enables you to know the names of the methods and their inputs so you can call them appropriately.

Additionally, if you want to manage security on a per service and per method basis, you will want to add the service names and method names in the Resource table. Refer to the “Security” chapter of the *Chordiant 5 Foundation Server Developer’s Guide* for details.

PERFORMING BUSINESS DOMAIN NUMBER GENERATION

You can have the service automatically generate numbers, such as order numbers and customer numbers. Note that these are numbers are *not* IDs; they are business domain numbers. You can also choose to use methods other than the ones described here to generate numbers as long as the algorithm produces unique values.

Note also that the procedure described here is designed to work against Oracle and DB2 databases. To use the procedure against other databases, override and modify the `com.chordiant.bd.sql.SQLNumberGeneration` class.

Before You Begin

Observe that there are several number generators already specified in the context of the `jxb.xml` configuration file. See if one of them meets your needs before you create your own. These number generators include:

- `AccountNumberGenerator`
- `CaseNumberGenerator`
- `CustomerNumberGenerator`
- `OrderNumberGenerator`
- `RmaNumberGenerator`—for Return Merchandise Authorization (RMA) number generation.

The `NumberGeneratorHelper` will create an entry in the number generator hash table with the `key` and `classname` tag values from the appropriate section of the `jxb.xml` configuration file.

The value of `classname` should be a fully-qualified classname for the number generator class.

[Code Sample 4-2](#) shows a section of the `jxb.xml` configuration file. Refer to the file directly for code annotations.

```
<Section>BusinessObjectNumberGeneration
  <Tag>numbergenerator
    <Value>AccountNumberGenerator</Value>
  </Tag>
  <Tag>numbergenerator
    <Value>CaseNumberGenerator</Value>
  </Tag>
  <Tag>numbergenerator
    <Value>CustomerNumberGenerator</Value>
  </Tag>
  <Tag>numbergenerator
    <Value>OrderNumberGenerator</Value>
  </Tag>
```

Code 4-2: Section of the JXB.xml Configuration File


```
<Tag>numbergenerator
  <Value>RmaNumberGenerator</Value>
</Tag>
</Section>

<Section>AccountNumberGenerator
  <Tag>key
    <Value>AccountNumber</Value>
  </Tag>
  <Tag>classname
    <Value>com.chordiant.bd.numberGeneration.AccountNumberGenerator</Value>
  </Tag>
</Section>

<Section>CaseNumberGenerator
  <Tag>key
    <Value>CaseNumber</Value>
  </Tag>
  <Tag>classname
    <Value>com.chordiant.bd.numberGeneration.CaseNumberGenerator</Value>
  </Tag>
</Section>

<Section>CustomerNumberGenerator
  <Tag>key
    <Value>CustomerNumber</Value>
  </Tag>
  <Tag>classname
    <Value>com.chordiant.bd.numberGeneration.CustomerNumberGenerator</Value>
  </Tag>
</Section>

<Section>OrderNumberGenerator
  <Tag>key
    <Value>OrderNumber</Value>
  </Tag>
  <Tag>classname
    <Value>com.chordiant.bd.numberGeneration.OrderNumberGenerator</Value>
  </Tag>
</Section>

<Section>RmaNumberGenerator
  <Tag>key
    <Value>RmaNumber</Value>
  </Tag>
  <Tag>classname
    <Value>com.chordiant.bd.numberGeneration.RmaNumberGenerator</Value>
  </Tag>
</Section>
```

Code 4-2: Section of the JXB.xml Configuration File (Continued)

To generate unique business domain numbers:

1. Specify the class responsible for the number generator in the **BusinessObjectNumberGeneration** section of the configuration file.

You can specify the number generator using the tags shown in [Code Sample 4-3](#).

```
<Tag>numbergenerator
  <Value>{MyNumberGeneratorName}</Value>
</Tag>

<Section>MyNumberGeneratorClassname
  <!-- Specify the package and class name for the generator class -->
  <Tag>classname
    <Value>com.chordiant.bd.numbergeneration.
      {MyNumberGeneratorClassname}</Value>
  </Tag>
  <!-- This key will be used to retrieve the my number generator -->
  <Tag>key
    <Value>{MyNumberGeneratorName}</Value>
  </Tag>
</Section>
```

Code 4-3: Specifying the Number Generator

2. Extend `com.chordiant.bd.numberGeneration.NumberGenerator`, which implements the `ItemNumberGenerator` interface.

[Code Sample 4-4](#) shows the signatures of the three methods included in the `ItemNumberGenerator` interface.

```
public String getItemNumber()
public String keyNameForObject()
public void setup( Object obj )
```

Code 4-4: ItemNumberGenerator Interface Methods

3. Instantiate an instance of the number generator within your business service class, and use it to generate a number.

Use the `NumberGeneratorHelper` class, which is a factory, to dynamically load and store number generator objects.

This class has a set of static methods including `setup`, `newInstance`, and `getInstance`, which you can use to create and initialize your number generator.

4. At the class level (that is, not within any function) declare your class as type `ItemNumberGenerator`, as illustrated in [Code Sample 4-5](#).

```
public class MyBusinessService {
  ...
  ItemNumberGenerator myNumberGenerator = null;
  ...
}
```

Code 4-5: Declaring Class as Type ItemNumberGenerator

5. The `NumberGeneratorHelper` is initialized the first time you call it, as illustrated in [Code Sample 4-6](#).

```
protected ServiceControlResponse setup(ServiceControlRequest theRequest)
    throws Throwable {
    BusinessObjectResourceInterface myResourceManager = initResourceManager();
    accountNumberGenerator = NumberGeneratorHelper.getInstance(
        AccountNumberGenerator.ACCOUNT_KEY_NAME, myResourceManager);
}
```

Code 4-6: Initializing the `NumberGeneratorHelper`

6. At this point, you can use your defined number generator to generate an item number for `MyBusinessService`, as illustrated in [Code Sample 4-7](#).

```
String myItemNumber = myNumberGeneratorClass.getItemNumber();
```

Code 4-7: Generating an Item Number for `MyBusinessService`

CUSTOMIZATION CONCEPTS

When performing your customizations, please consider:

- Do not add business logic to the generated client agent. Business logic belongs in the business service.
- If you choose to use the extended persistence functionality, including the generic service, generic service client agent, business object behavior, and associated classes, use them in conjunction with a Chordiant business service (such as those described in this chapter). Refer to [“Usage Model for Extended Persistence Components” on page 215](#) for more information.

Tutorial: Creating and Extending a Business Service

This tutorial is designed to walk you through a model in which you subclass Chordiant's `BusinessDataServiceBaseClass` to create your own service and then extend the service you created in the first part, as an existing service.

This chapter is broken into two parts:

- [“Part I: Creating a New Service”](#)
- [“Part II: Extending an Existing Service”](#)

The model created in this tutorial, `VehicleTaxCalculation.mdl`, is available in final form in `{eclipse_root}/plugins/{data_model_plugin}/rosemodels/`

You can also find it through the Business Component Generator.

You can refer to it while you step through the tutorial.

This tutorial assumes that you have a working knowledge of object modeling using Rational Rose.

PART I: CREATING A NEW SERVICE

Before you begin, you might want to open up the finished tutorial model.

Figure 5-1 illustrates the final model for Part I.

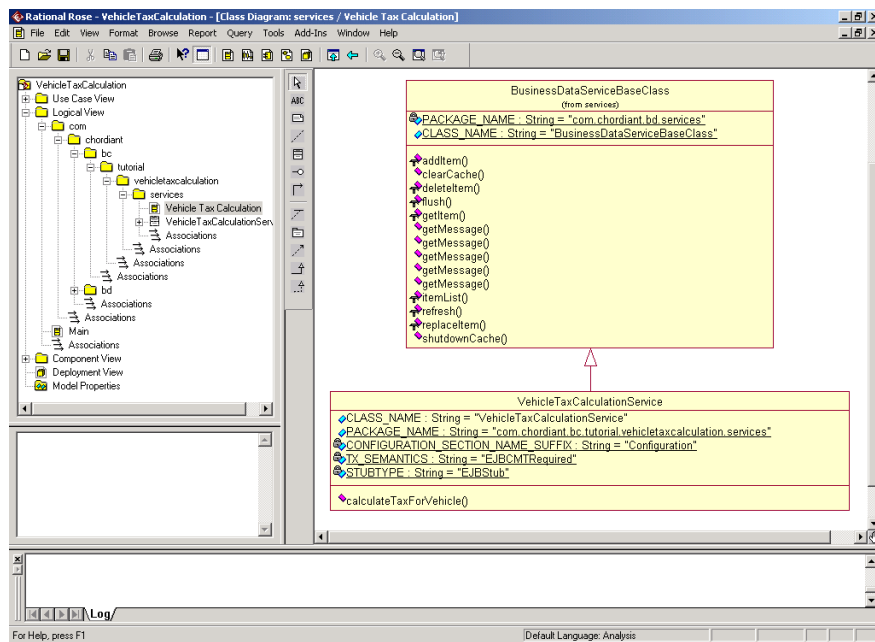


Figure 5-1: VehicleTaxCalculation Tutorial Model

This model shows how to create your own service, based on the Chordiant business service base class.

Notes: Early steps in the tutorial affect later steps. You will likely want to follow these procedures from start to finish.

It is a good idea to save your model often as you progress through the tutorial. This can help you find any mistakes you may have made more easily than if you wait until the end.

Creating a Model and the Base Class

To create a model:

1. In Rational Rose, open Chordiant's base service model, JXBServiceBase.mdl.

This model is located in

`{eclipse_root}/plugins/{data_model_plugin}/rosemodels`

Notes: Copy this file to another location and open it there, or create a new project through the Business Component Generator, using this model. Then you can customize the model from within your project. Refer to [“Using the Business Component Generator” on page 15](#) for details on creating a new project.

You cannot change the name of the model, its descriptor file, or its XMI file once they are a part of the Tools Platform project. Refer to the note on [page 22](#) for details.

Do not make your customizations within the base model. Always use a copy.

2. If you are modifying the model outside of the Chordiant Tools Platform, save this model as `VehicleTaxCalculation.mdl`.
3. In the Logical View, create a new series of nested packages for this tutorial, to result in the final package: `com.chordiant.bc.tutorial.vehicletaxcalculation.services`.

Note: You will likely see a warning message about having “services” in multiple locations. This condition is not a problem, so you can ignore the message.

4. Create a new class diagram in this package called **Vehicle Tax Calculation**.
5. Open the **Vehicle Tax Calculation** class diagram.
6. Drag the **BusinessDataServiceBaseClass** into the **Vehicle Tax Calculation** class diagram.
7. Create a new class inside the new **taxcalculation** service package called **VehicleTaxCalculationService**.
8. Drag the **VehicleTaxCalculationService** class into the class diagram work space.
9. Draw a generalization arrow from the **VehicleTaxCalculationService** class to the **BusinessDataServiceBaseClass**.
10. Save your model.

Create the Class Constants

11. Open the **Class Specification** window and select the **Attributes** tab.
12. Add the new *static* class attributes listed below.

To specify an attribute as *static*, go to the **Details** tab and select the **Static** checkbox. Be sure to click **OK** after each addition.

- PACKAGE_NAME: public String =
"com.chordiant.bc.tutorial.vehicletaxcalculation.services"
- CONFIGURATION_SECTION_NAME_SUFFIX: private String = "Configuration"
- TX_SEMANTICS: private String = "EJBCMTRequired"
- STUBTYPE: private String = "EJBStub"
- CLASS_NAME: public String = "VehicleTaxCalculationService"

Note: This is an additional CLASS_NAME attribute. Do not change the inherited CLASS_NAME from BusinessDataServiceBaseClass.

13. Save your model.

Create an Operation

14. Open the specifications for the VehicleTaxCalculationService class.
15. Select the **Operations** tab. Right-click in the open space and select **Insert new operation**. Name this new operation calculateTaxForVehicle.

16. Double-click on the `calculateTaxForVehicle` operation to open the **Operation Specification** window.

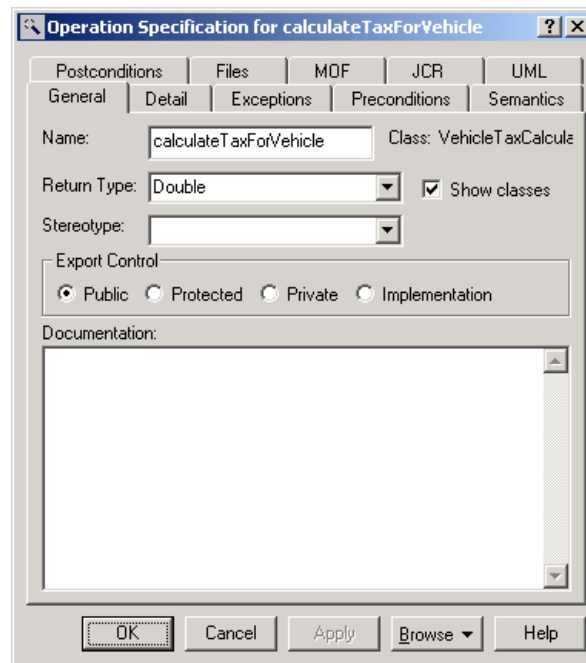


Figure 5-2: Operation Specification Window for `calculateTaxForVehicle`

17. In the **Operation Specification** window, select the **General** tab. Set the return type to Double.

Create the Arguments

18. Select the **Detail** tab. Right-click in the text area and select **Insert**. Create the following arguments, in order:

NAME	TYPE
username	String
authentication	String
registrationNumber	String

Table 5-1: Creating Three Arguments

19. Click **OK** on the Operation and **Class Specification** windows.
 20. Save your model again.

21. Export your model to XMI, in ASCII format.

Tip: To export XMI, the class diagram must be open.

22. If you have been working in the Chordiant Tools Platform, you might need to refresh your project to see the new XMI file. Right-click the project name and select **Refresh**.

Generating the Service Framework Components

You are now ready to use the Business Component Generator to create your service framework components.

To create the service framework components:

1. If you have been working with your model outside of the Chordiant Tools Platform, you must create a descriptor file for your service. It should be called `VehicleTaxCalculationDescriptor.xml` and should look like [Code Sample 5-1](#).

```
<project-descriptor>
<project-name>tutorialVehicleTaxCalculation</project-name>
<title>ServiceCreationTutorial</title>
<modeltype>service</modeltype>
<cmi>false</cmi>
<persistence>false</persistence>
<business-tier>false</business-tier>
<services>true</services>
<schema>false</schema>
<database>false</database>
<unit-tests>false</unit-tests>
</project-descriptor>
```

Code 5-1: VehicleTaxCalculationDescriptor.xml File

2. You must import the model, the XMI file, and the descriptor file into the Chordiant Tools Platform to generate the service components. Refer to [“Using the Business Component Generator” on page 15](#) for detailed steps.

Note: If you have been customizing the base model you imported into the Chordiant Tools Platform, just rebuild your project. From the **Project** menu, select **Rebuild project**.

Verifying the Build

To verify the build:

1. Within the development environment, refresh your VehicleTaxCalculation project. You should see a generated source directory containing your new service, client agents, and constants classes.
2. Check for any compilation errors and fix them.

Troubleshooting

If your project does not compile, you might need to perform these steps:

1. Include the `jxextensions.jar` in your project.
 - a. Right-click the project directory and select **Properties**.
 - b. On the left side, choose **Java Build Path**. On the right side, select the **Libraries** Tab. Click **Add JARs**.
 - c. Select `FOUNDATION_LIB`.
 - d. Click the **Extend** button. Select the `jxextensions.jar`.
 - e. Click **OK** three times to close the open dialog boxes.
2. Perform [Step 1](#) again, for the Java Variable: `{WAS_RUNTIME}\lib\j2ee.jar`. For this step, click the **Add External JAR** button, instead of the **Add JARs** button.
3. Recompile the `tutorialVehicleTaxCalculation` project.

Adding Logic to the Service

At this point the service is a skeleton, with all the plumbing for the new function. However, it does not have any logic for the actual business process. For now, we will create a static process that calculates a static tax rate of \$50 per vehicle.

Remember that when you rebuild the project, the code in the `src/generated` directory will be overwritten. So any customizations you make to this service will be lost. So you can extend the generated service in a separate directory to make sure your changes are tied to your model, but are not overwritten on code regeneration.

To add logic to the service:

1. In the Chordiant Tools Platform, open the **Java** perspective.
2. In your new project, create a new directory where you will create your customized code. You might want to call it `custom`, to distinguish it from the generated code in the `src` directory.

To create the new directory:

- a. Right-click the project name and select **Properties**.
 - b. On the left side, select **Java Build Path**. On the **Source** tab, add a folder called `custom` within the `src` directory.
 - c. Refresh the project to see this new `custom` directory.
3. In the `custom` directory, create a new class that subclasses from the services skeleton generated in the `src/generated` directory. Call this class `customVehicleTaxCalculationService`. You will make your modifications to this class instead of to the generated class.

To create the new class:

- a. Right-click the project name and select **New Class**.
- b. Specify the directory name as `{project name}/src/custom`.
- c. Specify the package name as `com.chordiant.bc.tutorial.vehicletaxcalculation.services.custom`
- d. Specify the class name as `customVehicleTaxCalculationService`.
- e. Mark the class as public.

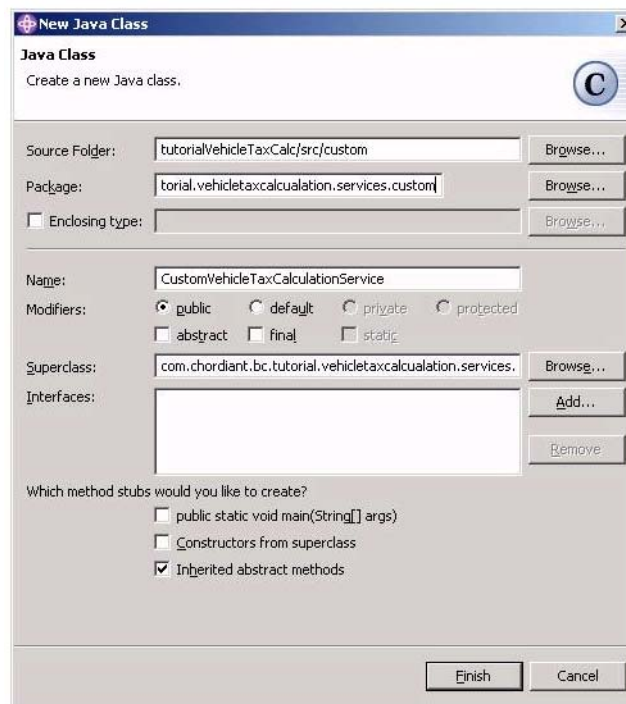


Figure 5-3: Creating a New Class

- f. Specify the superclass as the generated service class in the `src` directory, `vehicleTaxCalculationService`.
4. Copy the following bits of code from the generated service class to your new custom class:
 - `PACKAGE_NAME` and `CLASS_NAME` constants
 - `setup` method
 - `calculateTaxForVehicle` method
 5. Update the copied constants with your new information, reflecting the new `PACKAGE_NAME` (`com.chordiant.bc.tutorial.vehicletaxcalculation.services.custom`) and `CLASS_NAME` (`customVehicleTaxCalculationService`).

6. In the `setup` method, comment out the `resourcemanager` line. You do not need the resource manager, since this service does not touch a back-end database.

```
//resourcemanager = initResourceManager()
```

7. In the imports section of your class, right-click and select **Source**, then **Organize imports**. This adds the imports you need into your new class. If there are any conflicts, a dialog box will appear, where you can choose the appropriate import statement.

Tip: You can choose to import Chordiant's `LogHelper` at this point, to help you in your testing. It is very helpful in Part II of this tutorial, beginning on [page 76](#). The `LogHelper` is in the package `com.chordiant.core.log`.

8. Enter the logic for the `calculateTaxForVehicle` method, as shown in [Code Sample 5-2](#).

```
public Double calculateTaxForVehicle(
    String username,String authentication,String registrationNumber)
    throws Exception
{
    final String METHOD_NAME = "calculateTaxForVehicle";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    Double retVal = null;

    LogHelper.info(PACKAGE_NAME, CLASS_NAME, METHOD_NAME, "calculating tax for vehicle #: " +
        registrationNumber);

    retVal = new Double("50.00");

    LogHelper.info(PACKAGE_NAME, CLASS_NAME, METHOD_NAME,"tax calculated is: " +
        retVal.toString());

    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    return retVal;
}
```

Code 5-2: Complete calculateTaxForVehicle Method

9. Save this customized service class.

Registering the Service Through Configuration

To register the service:

1. Copy the configuration file in your project's `output/config` directory to your ChordiantEAR project's `/config/Chordiant/components` directory.
2. Rename the file after the service: `customVehicleTaxCalculationService.xml`.

3. The configuration file requires some modification after generation. For the `clientagent` entries, fix the fully-qualified class reference from:
`com.chordiant.bc.tutorial.vehicletaxcalculation.
services.client.VehicleTaxCalculationClientAgent`
to:
`com.chordiant.bc.tutorial.vehicletaxcalculation.
clientAgents.VehicleTaxCalculationClientAgent`
4. Change the name and classpath value for the service to point to your customized service that you just created in [“Adding Logic to the Service” on page 63](#).

Note that you do not need to modify the client agent section in this step. The generated client agent points to the generated service. In this step, you are pointing the generated service to your customized class. This one change takes care of the issue.

[Code Sample 5-3](#) shows the updated configuration file.

```
<Section>services
  <Tag>VehicleTaxCalculationService.name
    <Value>VehicleTaxCalculationService</Value>
  </Tag>
</Section>

<Section>VehicleTaxCalculationService
  <Tag>classname
    <Value>com.chordiant.bc.tutorial.vehicletaxcalculation.services.custom.
      customVehicleTaxCalculationService</Value>
  </Tag>
  <Tag>ConnectionName
    <Value>EJBCTRequired</Value>
  </Tag>
</Section>

<Section>clientagents
  <Tag>VehicleTaxCalculationClientAgent.agent
    <Value>VehicleTaxCalculationClientAgent</Value>
  </Tag>
</Section>

<Section>VehicleTaxCalculationClientAgent
  <Tag>classname
    <Value>com.chordiant.bc.tutorial.vehicletaxcalculation.clientAgents.VehicleTaxCalculationClientAgent</Value>
  </Tag>
  <Tag>stubtype
    <Value>EJBStub</Value>
  </Tag>
</Section>
```

Code 5-3: Updated customVehicleTaxCalculationService.xml Configuration File

5. Save your files and rebuild the project. Make sure that you do not have any errors.

The Client Agent Tester

1. Using Windows Explorer, copy the tester from the `{eclipse_root}/plugins/{data_model_plugin}/ServiceExtensionTutorial_source/src/com/chordiant/bc/tutorial/vehicletaxcalculation/unittestester` directory to your custom project directory.
2. Update the package definition in the tester to reflect that it is located in your custom directory: `com.chordiant.bc.tutorial.vehicletaxcalculation.services.custom`.
3. Refresh your project to see the new tester.

Note: The default security setting allows all users to call all APIs on all services. Therefore, without doing anything, your new service is accessible to all users, so you should be able to test it easily. To add access control to your services, follow the steps defined in “Adding a Service as a Resource” in the “Security” chapter of the *Chordiant 5 Foundation Server Developer’s Guide*.

Setting up the Runtime Environment

Exporting the JAR

1. Export the project as a JAR and save it in the `ChordiantEAR\lib-cust` project. This will set up the JAR in the server-side test environment.
 - a. To export the JAR, highlight the project directory in the development environment.
 - b. Right-click and select **Export**.
 - c. Choose the JAR option and click **Next**.
 - d. Deselect all options except `src/generated` and `custom`, or wherever your source files are located if you moved them out of this directory. Deselect **.classpath** and **.project** on the right side as well.
 - e. Clear the **Export java source files and resources** checkbox.
 - f. Save the JAR file as `tutorialvehicletaxcalculation.jar` within the server library directory: `\ChordiantEAR\lib-cust`. Click **Finish**.
2. Copy the JAR file to the `output\lib` directory of your project so it will be included in an EAR file that you can create with the Application Packaging Manager (APM).
3. Open the **J2EE** perspective in the development environment.

- Expand the EJB Modules directory. Right-click on the CMT bean and choose the **open with ... Jar dependency editor** option to open the **JAR Dependency** window.

The **JAR Dependency Editor** is a graphical interface for editing the `manifest` file, as seen through the **J2EE** perspective. If you select the **Source** tab at the bottom of the window, you will see the manifest file.

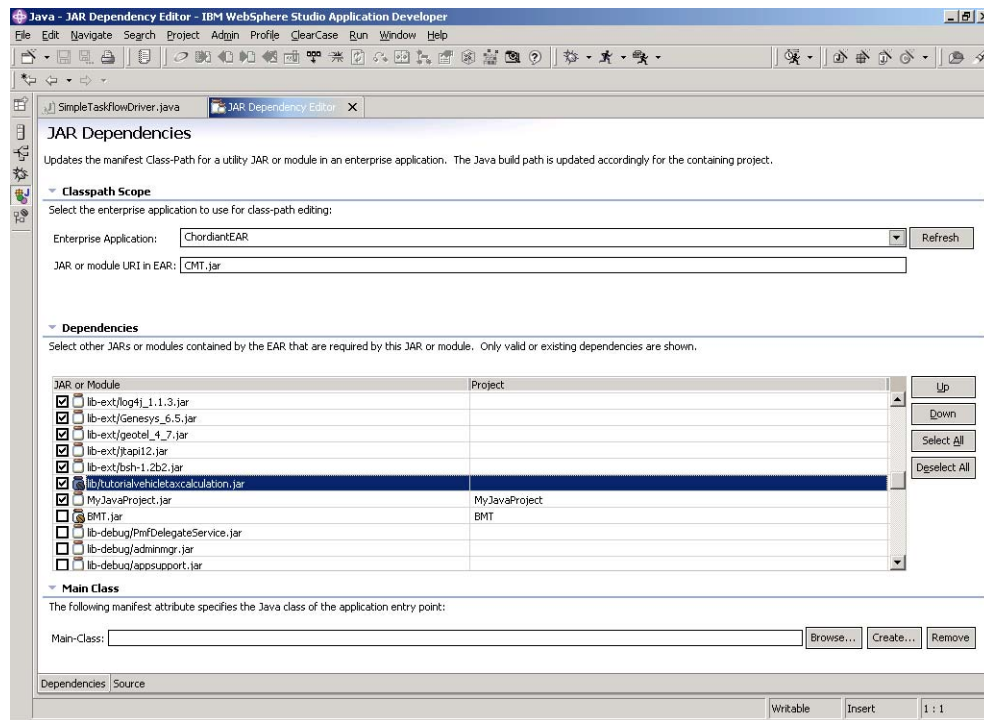


Figure 5-4: JAR Dependency Window

Note: WebLogic users must add the JAR to the WebLogic launch configuration. From the **Run** menu in the **Java** perspective, select **Run**. Select the **classpath** tab. Add the JAR to the `servers-weblogic` configuration. Click **Close**. This procedure is somewhat similar to that described in [“Setting Up and Running the Tester” on page 85](#), but be sure to modify the server configuration file, usually called `servers-weblogic`.

5. Scroll down to the bottom of the Dependency list to find the project export JAR you created. Check the box to the left of this entry to add the JAR to the CMT's JAR dependency list.

Note: If you do not see your JAR file in this list, close the **JAR Dependency** window and refresh the CMT bean in the **J2EE** perspective. Then open the **JAR Dependency** window again.

6. **(Optional)** Highlight the JAR's row and click the **Up** button until the JAR moves up to the top of the list. This step is not required. You might find it helpful to move JARs for your project up to the top of the list so you can find them more easily.
7. Save and close the **Dependency Editor** window. Close the **J2EE** perspective.

Setting Up and Running the Tester

You must set up your system properly before you can run your tester.

To set up your system for running the tester:

1. Open the **Server** perspective in the Chordiant Tools Platform.
2. Start the server. It will take a few minutes to get started.
3. In the meantime, open the **Java** perspective in your development environment.
4. From the **Run** menu, select **Run**.
5. On the left side, select **Java Application**. Select **New**.

6. On the **Main** tab, enter or browse to find the name of the project and the class name of the tester.

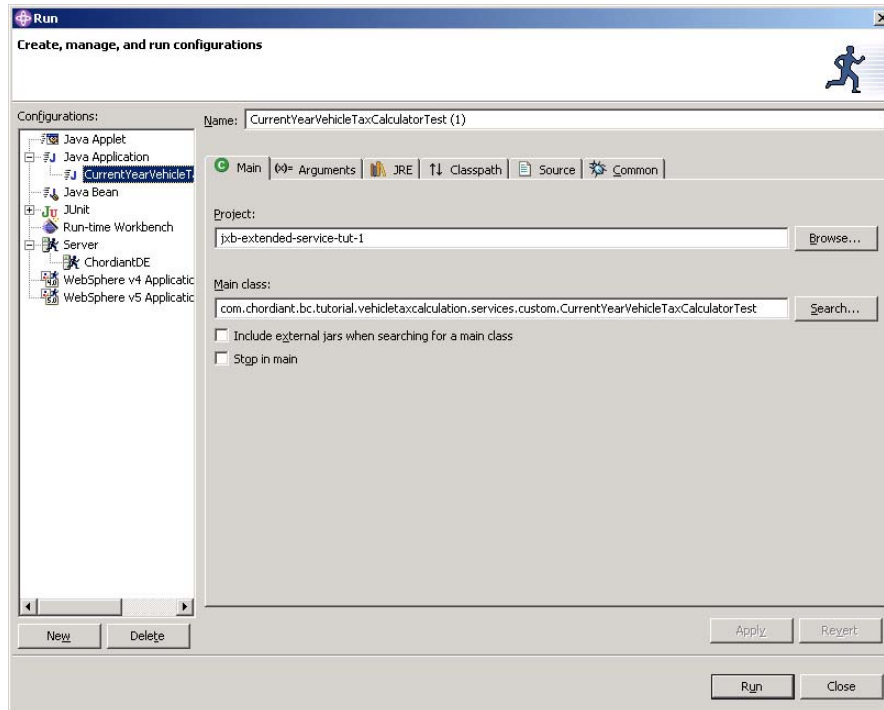


Figure 5-5: Setting Up the Tester: Main Tab

7. On the **Arguments** tab, enter the arguments shown in [Code Sample 5-4](#).

For WebSphere Studio Application Developer (WSAD) with WAS 5.0, refer to [Code Sample 5-4](#).

```
-Dpassword=mmalone
-Dusername=mmalone
-Djava.naming.provider.url=iiop://localhost:2809
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
-Dcom.chordiant.config.ejb.jndi.name=com_chordiant_service_ejb_EJBGatewayServiceBMT
```

Code 5-4: -D Arguments for the Tester in WSAD

Note: You must add the password and username parameters. The others should be present from when you set up your Chordiant projects.

For WebLogic, you can enter the arguments as shown in [Code Sample 5-5](#), or within the Server Properties, as described in the *Chordiant 5 Tools Platform Getting Started Guide*.

```
-Dpassword=mmalone
-Dusername=mmalone
-Djava.naming.provider.url=t3://localhost:7001
-Djava.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
-Dcom.chordiant.config.ejb.jndi.name=com_chordiant_service_ejb_EJBGatewayServiceBMT
```

Code 5-5: -D Arguments for the Tester in WebLogic

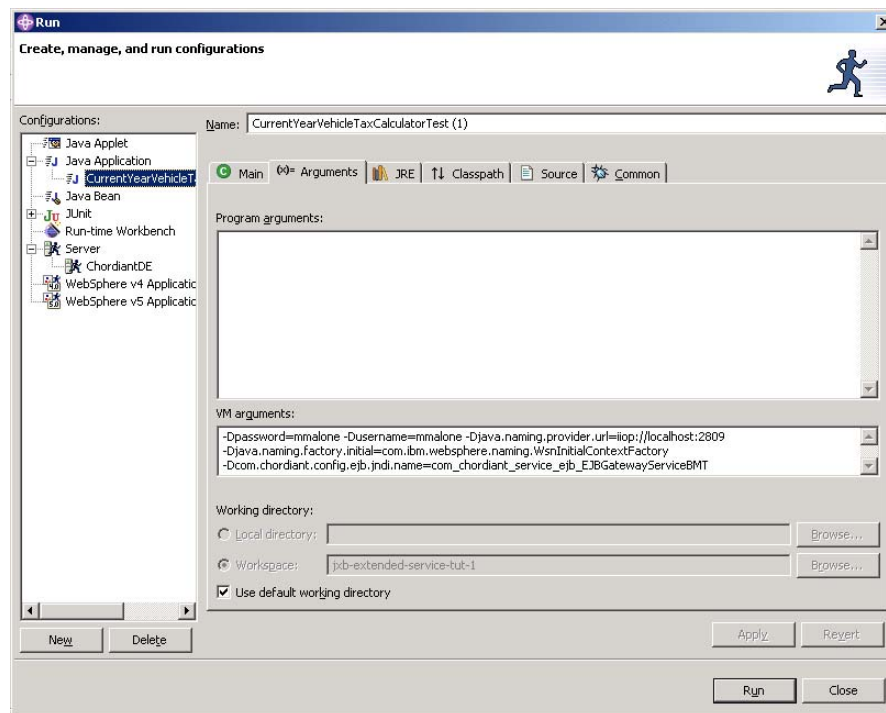


Figure 5-6: Setting Up the Tester: Arguments Tab

8. On the **JRE** tab, select the JRE that matches that of your server. In this example, we are selecting the WebSphere 5 JRE.

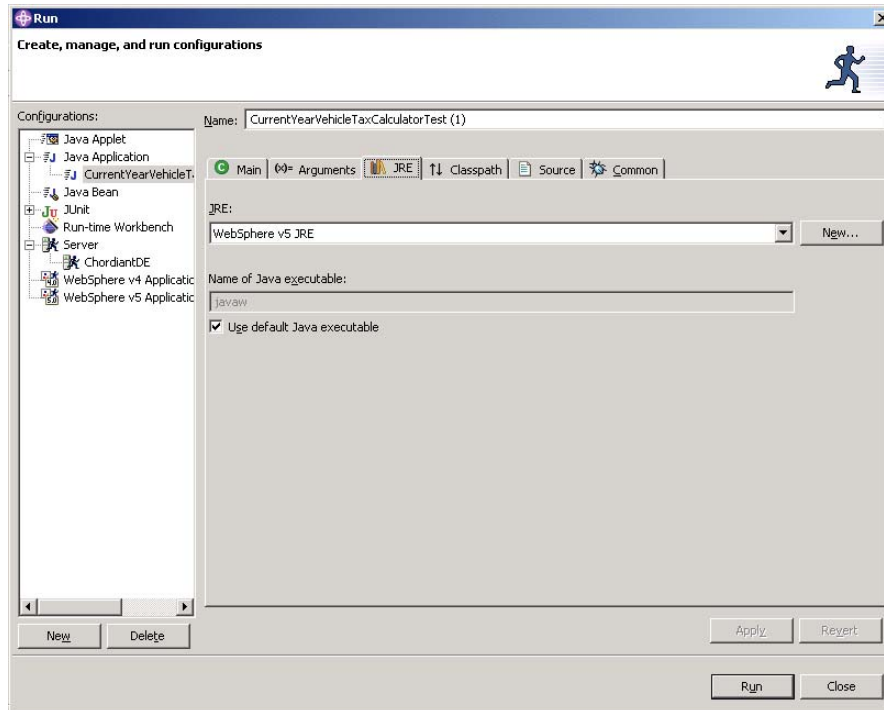


Figure 5-7: Setting Up the Tester: JRE Tab

9. On the **Classpath** tab, make sure that you have all of the appropriate JARs included.
 - a. Be sure to include the **CMT** project.
 - Click **Add projects**.
 - Select **CMT** from the list of available projects, and click **OK**. This action automatically adds the CMT project and its associated JARs.

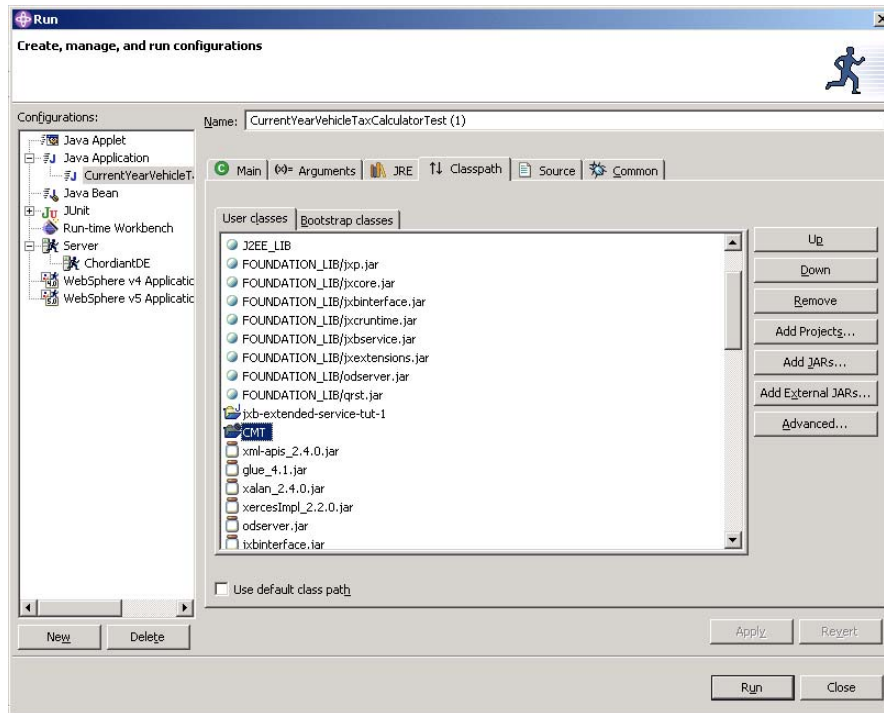


Figure 5-8: Setting Up the Tester: Classpath Tab

- b. Add the appropriate JARs for your application server. Click **Add External Jars** and navigate to the appropriate directory, based on your application server. Select the JARs and click **OK**.

Tip: To select multiple JARs in the selection window, hold the **CTRL** key while making your selections.

WebSphere 5

`{WebSphere_Install_Location}/runtimes/basev5/lib/` directory:

- j2ee.jar
- namingclient.jar
- ecutils.jar

WebLogic

`{WebSphere_Install_Location}/server/lib/` directory:

- `weblogic.jar`

c. For WebSphere 5 only, add the `properties` folder.

- Click **Advanced**. Click **Add External Folder**, then click **OK**.

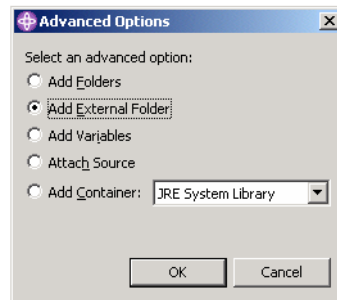


Figure 5-9: Adding an External Folder

- Navigate to the `{WebSphere_Install_Location}/runtimes/basev5` directory and select the `properties` directory. Click **OK** to add this `properties` directory to the classpath.

Figure 5-10 shows the end of the classpath list with the WebSphere JARs and `properties` directory added.

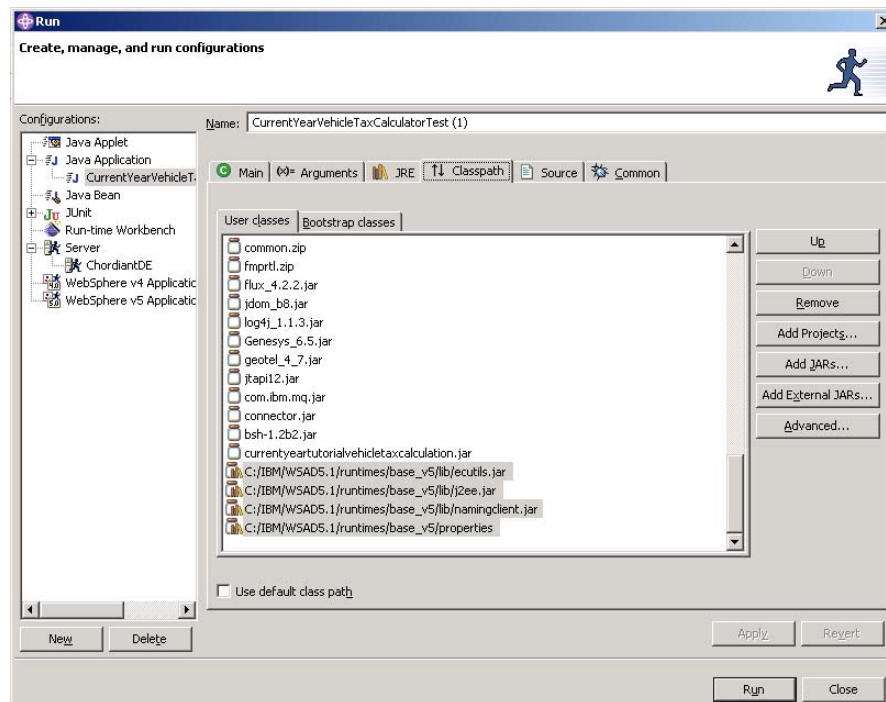



Figure 5-10: Classpath with WebSphere 5 JARs and Folder Added

10. When you have added all of your settings, make sure the server is started, and click **Run**. Your results will appear in the server and client logs in the console.

In the **Server** perspective, select the **Console** tab. Use the “Display Output of Selected Processes” icon  to toggle between the logs.

Note: This is the end of the first part of the services tutorial. Please feel free to explore the tutorial more.

If you choose to make your own customizations to it, as with all customizations, we suggest that you make your changes within a separate directory so you will not lose the functionality of the original and so your components will not get overwritten when you run the Business Component Generator.

PART II: EXTENDING AN EXISTING SERVICE

Introduction

In this next portion of the tutorial, you will extend the service you just created in [“Part I: Creating a New Service” on page 58](#). This is a simplistic example, meant to simulate extending an out-of-the-box Chordiant service and modifying a service API. Ordinarily, you would not customize a service that you have already customized; you would update that customized service in your model, regenerate the code, and update your implementation accordingly.

Extending the Service

If you want to skip Part I, you can use the finished `VehicleTaxCalculation.mdl` file provided in `{eclipse_root}/plugins/{data_model_plugin}/rosemodels/` as your starting point.

The finished model from Part II, `CurrentYearVehicleTaxCalculation.mdl`, is also available in the `{eclipse_root}/plugins/{data_model_plugin}/rosemodels/` directory, if you want to look at it before creating it yourself.

Note: Performing the steps in Part II of the tutorial overwrites some of the steps you performed in Part I. As a result, you will not be able to run the tester from Part I once you complete the steps in Part II.

Figure 5-1 illustrates the final model for Part II.

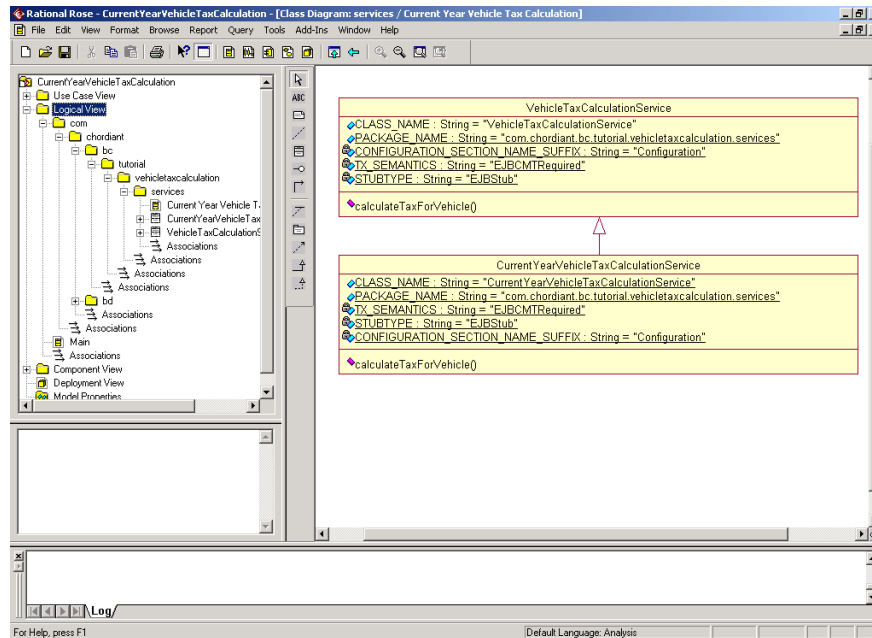


Figure 5-11: CurrentYearVehicleTaxCalculationService Model

To extend your VehicleTaxCalculation service:

1. Open a copy of the Rational Rose model for the vehicle tax calculator service.

You can either continue using the model from the first part of the tutorial or open the VehicleTaxCalculation.mdl file provided in `{eclipse_root}/plugins/{data_model_plugin}/rosemodels/`.

Note: If you choose to continue using the model from Part I of the tutorial, these changes will overwrite your work from that first tutorial. You can choose to create a new project, importing the model you created from Part I or importing the VehicleTaxCalculation mentioned above.

Subclass the Existing Service

2. Inside the `com.chordiant.bc.tutorial.vehicletaxcalculation.services` package, create a new class called `CurrentYearVehicleTaxCalculationService`. Drag it into the Vehicle Tax Calculation class model you made earlier.
3. Create a generalization line from the `CurrentYearVehicleTaxCalculationService` class to the `VehicleTaxCalculationService`.

4. Create the following new static attributes for this class, as you did in the original model. Refer to [“Create the Class Constants” on page 60](#) for details. Note that this new CLASS_NAME attribute will be the third CLASS_NAME attribute for this class. Do not overwrite the existing CLASS_NAME attributes.
 - CLASS_NAME: public String = “CurrentYearVehicleTaxCalculationService”
 - PACKAGE_NAME: public String = “com.chordiant.bc.tutorial.vehicletaxcalculation.services”
 - CONFIGURATION_SECTION_NAME_SUFFIX: private String = “Configuration”
 - TX_SEMANTICS: private String = “EJBCMTRequired”
 - STUBTYPE: private String = “EJBStub”
5. Save your model.

Create a New Operation

6. Create the following new public operation and its associated arguments. Refer to [“Create an Operation” on page 60](#) for details.

calculateTaxForVehicle(String username, String authorization, String registrationNumber,
String registrationType)

return type = Double

At this point, your model will resemble the model shown in [Figure 5-12](#).

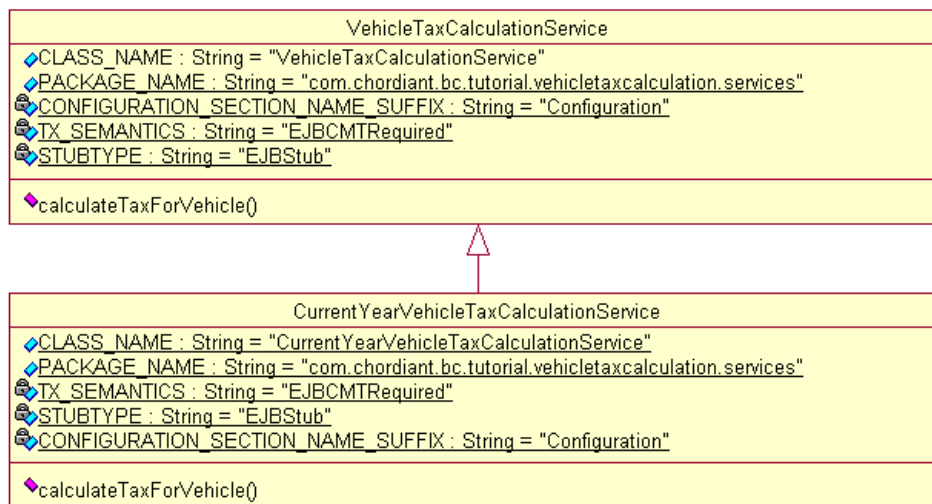


Figure 5-12: CurrentYearVehicleTaxCalculationService Model

7. Save the model and export it as an XMI file in ASCII format.

Generating the Service Framework Components

You are now ready to use the Business Component Generator to create your service framework components.

Copy the descriptor file from the original service and name it `CurrentYearVehicleTaxCalculationDescriptor.xml`. Most of the entries can remain the same, but change the project name to make it more specific to this new service.

```
<project-descriptor>
<project-name>Service Extension Tutorial</project-name>
<title>Service Extension Tutorial Model</title>
<modeltype>service</modeltype>
<cmi>false</cmi>
<persistence>false</persistence>
<business-tier>false</business-tier>
<services>true</services>
<schema>false</schema>
<database>false</database>
<unit-tests>false</unit-tests>
</project-descriptor>
```

Code 5-6: CurrentYearVehicleTaxCalculationDescriptor.xml Descriptor File

Note: The tutorial is written assuming that you will create your code in the same project you created in Part I. You can also choose to create a brand new project for this new code.

Refer to [“Using the Business Component Generator” on page 15](#) for steps on creating your components.

Adding Business Logic to the Current Year Service

This year, the state government has adjusted the tax for vehicles. Instead of a flat rate of \$50 for all vehicles, the state has implemented a rate of \$100 for commercial vehicles and a rate of \$50 for non-commercial vehicles. We must modify our logic to handle this new requirement.

New Constants

To distinguish between these two types of vehicles, you will add constants to the configuration file. Refer to [“Updating the Configuration Settings” on page 82](#), where you can make this change while you are making other configuration file changes.

By adding these constants to the configuration file, they can be added and changed without restarting the server. You can easily update the configuration by refreshing the ConfigurationHelper in the Administrative Console. For details on using the Administrative Console, refer to the *Chordiant 5 Foundation Server Developer’s Guide*.

New Logic

To add the new logic for the service to handle the new constants:

1. In your Chordiant Tools Platform project’s `custom` directory, create a new `customCurrentYearVehicleTaxCalculationService` class that subclasses from the new `CurrentYearVehicleTaxCalculationService` skeleton generated in the `src/generated` directory. You will make your modifications to this class instead of to the generated class.

You performed these steps in Part I of the tutorial. For details, refer to [Step 1 - Step 3 on page 64](#).
2. Copy the relevant constants and methods from the generated class and organize your imports, as you did in [Step 4 on page 65](#) through [Step 7 on page 66](#). Importing the LogHelper here is helpful for the current year tax calculation testing.
3. Open the `customCurrentYearVehicleTaxCalculationService` in the `custom` directory and look at the `processRequest` method. Notice that the `processRequest` method is aware of your new method (with four parameters).
4. Enter the logic for the new service, as illustrated in [Code Sample 5-7](#).

```
public Double calculateTaxForVehicle(
    String username, String authorization, String registrationNumber, String registrationType) throws Exception
{
    final String METHOD_NAME = "calculateTaxForVehicle";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    Double retVal = null;

    LogHelper.info(PACKAGE_NAME, CLASS_NAME, METHOD_NAME, "calculating tax for vehicle #: " +
        registrationNumber);
    final String registrationTypeCommercial = ConfigurationHelper.getConfigurationValue(
        "VehicleRegistrationConstants", "RegistrationTypeCommercial");
    final String registrationTypeNonCommercial = ConfigurationHelper.getConfigurationValue(
        "VehicleRegistrationConstants", "RegistrationTypeNonCommercial");
```

Code 5-7: Logic for calculateTaxForVehicle Service

```

    if (registrationType.equals(registrationTypeCommercial))
    {
        retVal = new Double("100.00");
    }
    else if (registrationType.equals(registrationTypeNonCommercial))
    {
        retVal = new Double("50.00");
    }
    else {
        Exception e = new Exception("Invalid registration type");
        throw e;
    }
    LogHelper.info(PACKAGE_NAME, CLASS_NAME, METHOD_NAME,"tax calculated is: " + retVal.toString());

    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retVal;
}

```

Code 5-7: Logic for calculateTaxForVehicle Service (Continued)

Confirming the Client Agent Parent Class

To confirm that the parent class for the client agent is correct:

1. Open the generated client agent.
2. Confirm that the parent class is the `VehicleTaxCalculationClientAgent`.

```

public class CurrentYearVehicleTaxCalculationClientAgent extends
VehicleTaxCalculationClientAgent

```

Updating the Configuration Settings

Adding the New Constants

To implement this logic, provide new constants to the service using the Chordiant configuration system:

1. In the ChordiantEAR/config/Chordiant/components directory, rename the customVehicleTaxCalculationService.xml from Part I to reflect the new service name. Call it customCurrentYearVehicleTaxCalculationService.xml.
2. Open the customCurrentYearVehicleTaxCalculationService.xml file and add the VehicleRegistrationConstants section, as shown in [Code Sample 5-8](#).

```
<Section>VehicleRegistrationConstants
  <Tag>RegistrationTypeCommercial
    <Value>"Commercial"</Value>
  </Tag>
  <Tag>RegistrationTypeNonCommercial
    <Value>"NonCommercial"</Value>
  </Tag>
</Section>
```

Code 5-8: New VehicleRegistrationConstants Section of the Configuration File

By adding these constants to the configuration file, they can be added and changed without restarting the server. You can easily update the configuration by refreshing the ConfigurationHelper in the Administrative Console. For details on using the Administrative Console, refer to the *Chordiant 5 Foundation Server Developer's Guide*.

Updating References

The generated configuration file needs some modification to make sure that your new service will be used.

Note: If you choose, you can copy the configuration file provided with this tutorial, rather than going through this procedure for your generated configuration file. Even if you do use the provided XML file, you might want to see how the steps below produced the provided XML file. The ServiceExtensionTutorialConfig.xml configuration file is located in the `{eclipse_root}/plugins/{data_model_plugin}/ServiceExtensionTutorial_source/config` directory. If you want to use it for this tutorial, change the name to customVehicleTaxCalculationService.xml.

To modify the file:

1. Open the Chordiant configuration file for the service:
`\ChordiantEAR\config\chordiant\components\customCurrentYearVehicleTaxCalculationService.xml`.
2. In the `clientagents` section, make sure that the tags all have the same name, which should be the same as the configuration file name: `VehicleTaxCalculationClientAgent.agent`. As shown in [Code Sample 5-9](#), there should be two entries for this, one with a value of `VehicleTaxCalculationClientAgent`, and one with a value of `CurrentYearVehicleTaxCalculationClientAgent`.

```
<Section>clientagents
  <Tag>VehicleTaxCalculationClientAgent.agent
    <Value>VehicleTaxCalculationClientAgent</Value>
  </Tag>
  <Tag>VehicleTaxCalculationClientAgent.agent
    <Value>CurrentYearVehicleTaxCalculationClientAgent</Value>
  </Tag>
</Section>
```

Code 5-9: ClientAgents Section of customCurrentYearVehicleTaxCalculationService.xml

3. In the `VehicleTaxCalculationClientAgent` section, change the `classname` tag to reflect the new Client Agent: `com.chordiant.bc.tutorial.vehicletaxcalculation.clientAgents.CurrentYearVehicleTaxCalculationClientAgent`.

This will force all existing code that accesses the existing client agent to receive an instantiated customized client agent instead. Remember that this code will still need to cast the client agent it receives to a `CurrentYearVehicleTaxCalculationClientAgent` and to access the new API.

```
<Section>VehicleTaxCalculationClientAgent
  <Tag>classname
    <Value>com.chordiant.bc.tutorial.vehicletaxcalculation.clientAgents.
      CurrentYearVehicleTaxCalculationClientAgent</Value>
  </Tag>
  <Tag>subtype
    <Value>EJBStub</Value>
  </Tag>
</Section>
```

Code 5-10: Updated classname Tag in Configuration File

4. Add a `CurrentYearVehicleTaxCalculationClientAgent` section that provides a fully-qualified classname value and an EJBStub subtype. This classname and EJBStub subtype are identical to the values in the `VehicleTaxCalculationClientAgent` section discussed in [Step 3](#).

```
<Section>CurrentYearVehicleTaxCalculationClientAgent
  <Tag>classname
    <Value>com.chordiant.bc.tutorial.vehicletaxcalculation.clientAgents.
      CurrentYearVehicleTaxCalculationClientAgent</Value>
  </Tag>
  <Tag>subtype
    <Value>EJBStub</Value>
  </Tag>
</Section>
```

Code 5-11: New CurrentYearVehicleTaxCalculationClientAgent Section of Configuration File

5. Since you just implemented the new `CurrentYearVehicleTaxCalculationClientAgent`, and unimplemented the old `VehicleTaxCalculationClientAgent`, you must now replace the `VehicleTaxCalculationService` configurations with values for the new `customCurrentYearVehicleTaxCalculationService`.

In the `services` section, make sure that the tag name is the same as the configuration file: `VehicleTaxCalculationService.name`. Change the value to `customCurrentYearVehicleTaxCalculationService` to make sure that the new service is used.

```
<Section>services
  <Tag>VehicleTaxCalculationService.name
    <Value>customCurrentYearVehicleTaxCalculationService</Value>
  </Tag>
</Section>
```

6. Find the `VehicleTaxCalculationService` section. Change the section name and the classname tag to reflect the new service, `customCurrentYearVehicleTaxCalculation`, as shown in [Code Sample 5-12](#). Replace all of the section names that referred to the old service with the `{packagename}.{classname}` for the new extended `CurrentYearVehicleTaxCalculation` service, which you created in the `custom` directory.

```
<Section>customCurrentYearVehicleTaxCalculationService
  <Tag>classname
    <Value>com.chordiant.bc.tutorial.vehicletaxcalculation.services.custom.
      customCurrentYearVehicleTaxCalculationService</Value>
  </Tag>
  <Tag>ConnectionName
    <Value>EJBCMTRrequired</Value>
  </Tag>
</Section>
```

Code 5-12: Updating the Configuration File

Note: By default, the generated service includes a Business Object Resource Manager, so your service can interact with a database. You must add the resource manager configuration parameters to the configuration file. Refer to the *Chordiant 5 Foundation Server Developer's Guide* for details on these configuration settings.

If your service does not interact with a database, override the `setup()` method in your customized class so the Resource Manager is not initialized.

7. At this point, you might also want to update your logging configuration settings in the `loghelper.xml` file, to help you when you are running the tester. You will probably want to turn on `debug` and `info` messages for this class. Refer to the “LogHelper” section of the *Chordiant 5 Foundation Server Developer's Guide*.

Testing the Service

Note: Remember that the client agent tests need to ensure that the client agent received is cast as a `CurrentYearVehicleTaxCalculationClientAgent`. The calculation tests need to ensure that a correct calculation is achieved.

Copying the Tester

1. Copy the `currentYearVehicleTaxCalculatorTest` tester from the `{eclipse_root}/plugins/{data_model_plugin}/ServiceExtensionTutorial_source/src/com/chordiant/bc/tutorial/vehicletaxcalculation/unittester` directory to your project directory.
2. Update the package definition in the tester to reflect that it is located in your custom directory: `com.chordiant.bc.tutorial.vehicletaxcalculation.services.custom`.

The tester includes these five test cases:

- Obtain a `CurrentYearVehicleTaxCalculationClientAgent` by providing a `VehicleTaxCalculationClientAgent` class name.
- Obtain a `CurrentYearVehicleTaxCalculationClientAgent` by providing a `CurrentYearVehicleTaxCalculationClientAgent` class name.
- Calculate the tax for a commercial registration.
- Calculate the tax for a non-commercial registration.
- Attempt to calculate the tax for an invalid registration type.

Exporting the JAR

Export the JAR file, following the steps from [“Exporting the JAR” on page 68](#).

Note: Be sure to restart the server after you export this new JAR file.

Setting Up and Running the Tester

You can leverage your tester setup from Part I for this new tester. You do not need to create a new tester from scratch.

To set up your tester for Part II:

1. Open the **Server** perspective and start the server.
2. From the **Run** menu, select **Run**.
3. Right-click the Java Application launch you created in Part I and select **Duplicate**.

4. Give this file a relevant name, like `CurrentYearTest`.
5. Change the mainclass to the new tester name, `currentYearVehicleTaxCalculatorTest`.
6. Click **Apply**.
7. Run this new launch file to test. Check the logging to see the messages. Remember that some errors are expected, based on the test cases. You will want to see these errors to make sure that your service is operating processing.

Note: This is the end of the second part of the services tutorial. Please feel free to explore the tutorial more.

If you choose to make your own customizations to it, as with all customizations, we suggest that you extend classes in a separate directory and alter those, or make your changes within a separate directory so you will not lose the functionality of the original and so your components will not get overwritten when you run the Business Component Generator.

Creating Web Services

Web services provide a platform-independent mechanism for Chordiant and non-Chordiant systems to work together.

- **Chordiant Web Services** — You can publish Chordiant web services so remote or non-Chordiant systems within your enterprise can take advantage of Chordiant functionality, including functionality that you have customized for your solution.
- **Non-Chordiant Web Services** — By making calls to web services published by other sources, your Chordiant system can encapsulate non-Chordiant functionality seamlessly within your Chordiant solution.

Chordiant's web services components include *WSDLs* and *WSDDs*.

- *WSDLs* are Web Service Description Language files. These platform-independent files specify the service interface in XML format. Any business object information associated with the service is automatically encapsulated within the *WSDL*.
- *WSDDs* are Web Service Deployment Descriptor files and are used in the *WSDL* deployment process. They include details on the business objects required by the web services and are used by the web services infrastructure for configuration and deployment.

Chordiant provides undeployed *WSDLs* and *WSDDs* for most Chordiant services. You can choose to deploy the *WSDLs* you want to use.

WEB SERVICES TOPOLOGY

At design time, Chordiant web services are created from Chordiant client agents. [Figure 6-1](#) illustrates how web services are customized at the same time as Chordiant services, through the Rational Rose model. After creating the model, you can use the Business Component Generator to create Chordiant services and client agents. The client agents act as proxies to the business services.

The client agents are inputs into the web service infrastructure code when generating web services files. The web services also act as proxies to Chordiant services, like the client agents do. These files are used during runtime, as shown in [Figure 6-2](#).

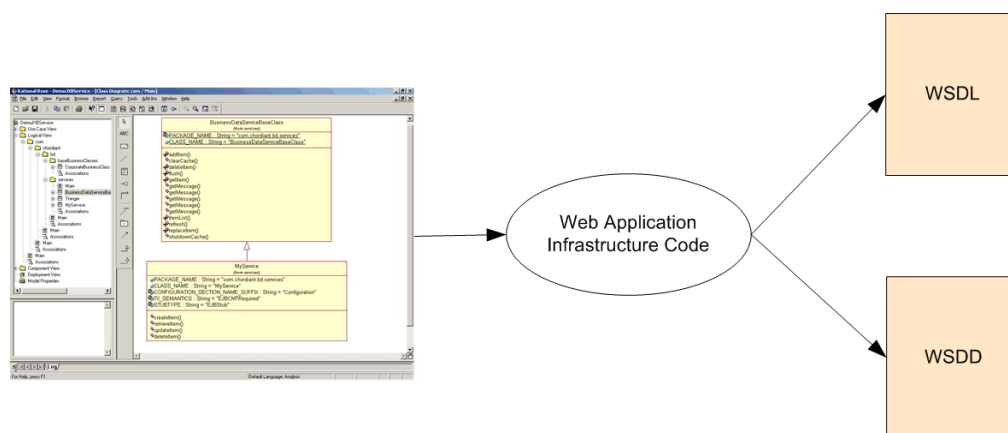


Figure 6-1: Web Services Topology: Design Time

Figure 6-2 illustrates that at runtime the web service infrastructure runs as a web application servlet within the application server, along with the JX EJB. You configure the web services exposed by this servlet using the WSDD file that you created during design time.

External applications using code based on the WSDL file contact the web service infrastructure through SOAP over HTTP. The web service infrastructure, in turn, contacts the Chordiant client agent and, ultimately, the JX service.

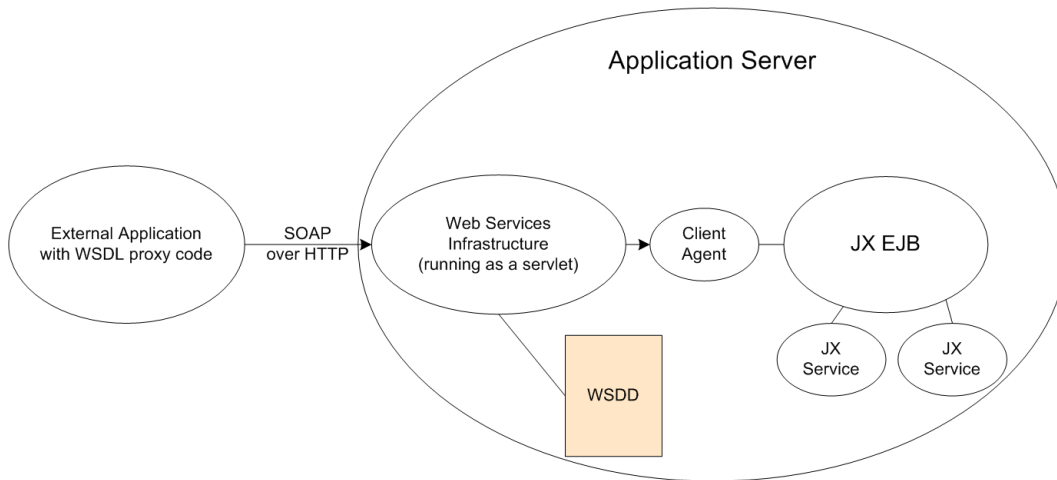


Figure 6-2: Web Services Topology: Runtime

CHORDIANT-PROVIDED WEB SERVICES

Most Chordiant services are provided with WSDLs and WSDDs. You can customize these Chordiant services and then generate your own customized WSDLs and WSDDs. For details on the APIs for these Chordiant services, refer to [Chapter 16, “Chordiant 5 Application Components”](#) and to the Javadoc. These two resources will help you to select appropriate client agents and specify the input parameters and return values for each service.

Only web services for Chordiant services that are on this list are supported at this time. Note that Chordiant supports RPC-style web services.

- AccountClientAgentRpcEnc
- CreditCardAccountClientAgentRpcEnc
- CtiBusinessServicesClientAgentRpcEnc
- CustomerClientAgentRpcEnc
- EbcInteractionClientAgentRpcEnc
- EstablishmentClientAgentRpcEnc
- GuideClientAgentRpcEnc
- HelloWorldClientAgentRpcEnc
- InventoryClientAgentRpcEnc
- LocationClientAgentRpcEnc
- LockClientAgentRpcEnc
- NumberGenerationClientAgentRpcEnc
- OfferingClientAgentRpcEnc
- OrderFulfillmentClientAgentRpcEnc
- OrderGenerationClientAgentRpcEnc
- OrderTrackingClientAgentRpcEnc
- PartyRoleClientAgentRpcEnc
- PeerMessageClientAgentRpcEnc
- PersistenceCacheManagerClientAgentRpcEnc
- PmfCustomerClientAgentRpcEnc
- PmfDelegateClientAgentRpcEnc
- ProductClientAgentRpcEnc
- QueueClientAgentRpcEnc
- SecurityMgrBeanClientAgentRpcEnc
- SeedClientAgentRpcEnc
- SessionClientAgentRpcEnc
- TimerClientAgentRpcEnc
- XMLClientAgentRpcEnc
- XMLStorageClientAgentRpcEnc

Chordiant-provided WSDLs are not deployed by default, but you can choose to deploy any Chordiant WSDLs for your solution.

Note that methods inherited from the service base class, such as `getMessage` and `refresh`, are not available for use with web services. For details on specific methods within the services listed above, refer to the Release Notes and to Chordiant Technical Support.

SECURITY AND WEB SERVICES

Chordiant web services are secure because the authentication token is embedded in each request to every Chordiant service, rather than just being part of the request container. Each Chordiant service call requires `username` and `authenticationToken` parameters. Without these parameters, you cannot access the service. Therefore, before you can call any Chordiant web service, you must first call the Chordiant Security Manager web service's `authenticate` method to receive an authentication token. Once you acquire the token, you can use it in your subsequent calls to any Chordiant web service. For more information on security, refer to the "Security" chapter in the *Chordiant 5 Foundation Server Developer's Guide*.

If you are using Secure Sockets Layer (SSL) for web security, web services will still be fully accessible and functional.

CONFIGURING FOR USING WEB SERVICES

To use web services with Chordiant, you must have installed the web services project in the Chordiant Tools Platform by following the directions and verification steps in the Resource Project section of the *Chordiant 5 Tools Platform Getting Started Guide*.

Creating a Java Proxy Code Project

When working with web services, you create proxy code. Even if you are not invoking a web service, this proxy code is generated when you create the required WSDD files. This proxy code is generated in a proxy project. You will use this project and proxy code throughout the examples and tutorials in these web services chapters.

Before you can create the proxy code used to invoke web services, you must create a new Java project and specify its Java build path to hold this code. Without the proper files in the build path, the proxy code will have errors.

To create a new Java project to hold your proxy code:

1. On the Chordiant Tools Platform's **File** menu, select **New Project**.
2. Create a new **Java** Project. On the left side, select **Java**. On the right side, select **Java Project**. Click **Next** to continue.

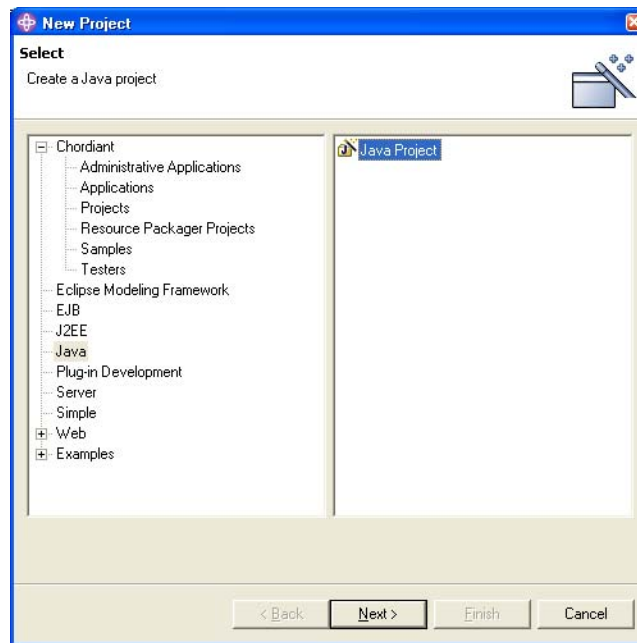


Figure 6-3: Creating a New Java Project

3. Name your new project. In the examples in this chapter, we use the name **WebServicesProxyProject**. Click **Finish** to create the project.

Note: The various web services scripts use the name **WebServicesProxyProject** for this project. If you choose to use a different name, you must update the scripts with the name you chose.

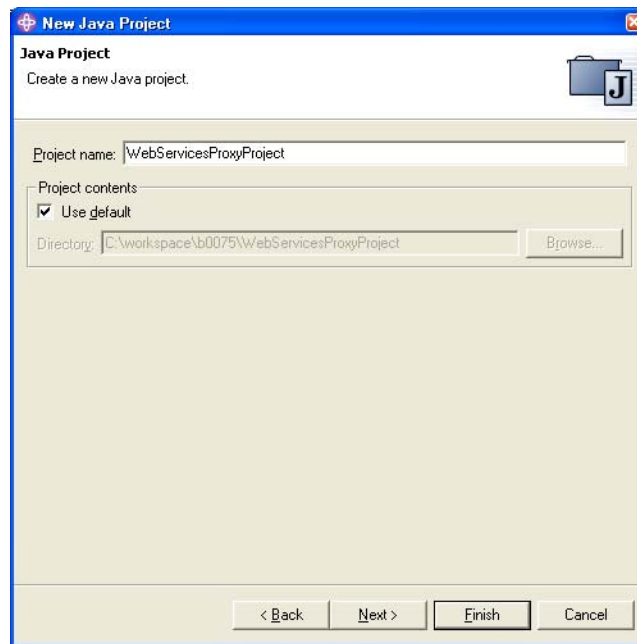


Figure 6-4: Naming the Java Project

4. The Java proxy code requires additional files in the build path. In the Tools Platform, right-click on this new project and select **Properties**.

5. On the left side, select **Java Build Path**. On the right side, click **Add JARs**.

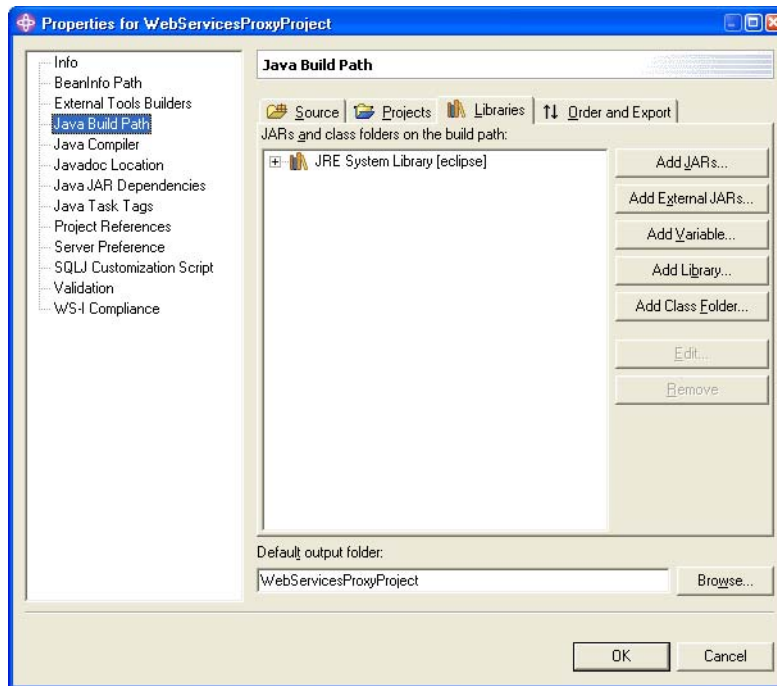


Figure 6-5: Adding JARs to the Build Path

6. In the **JAR Selection** Window, select the following files while holding the **CTRL** key:

a. Navigate to the appropriate directory and select the JAR files listed below.

— **WebSphere:** WebServices\WebContent\WEB-INF\lib

— **WebLogic:** WebServices\WEB-INF\lib

JAR Files:

— axis.jar

— axis-ant.jar

— commons-discovery.jar

— commons-logging.jar

— jaxrpc.jar

— log4j-1.2.8.jar

— saaj.jar

— wsdl4j.jar

b. Navigate to the ChordiantEAR\lib-ext directory and select these JAR files:

— jdom_b8.jar

— xalan_2.6.0.jar

— xerces_2.6.2.jar

— xerces-xml-apis_2.6.2.jar

When you have finished selecting, click **OK**. [Figure 6-6](#) shows the resulting properties for the project with all of the appropriate files in the Java build path.

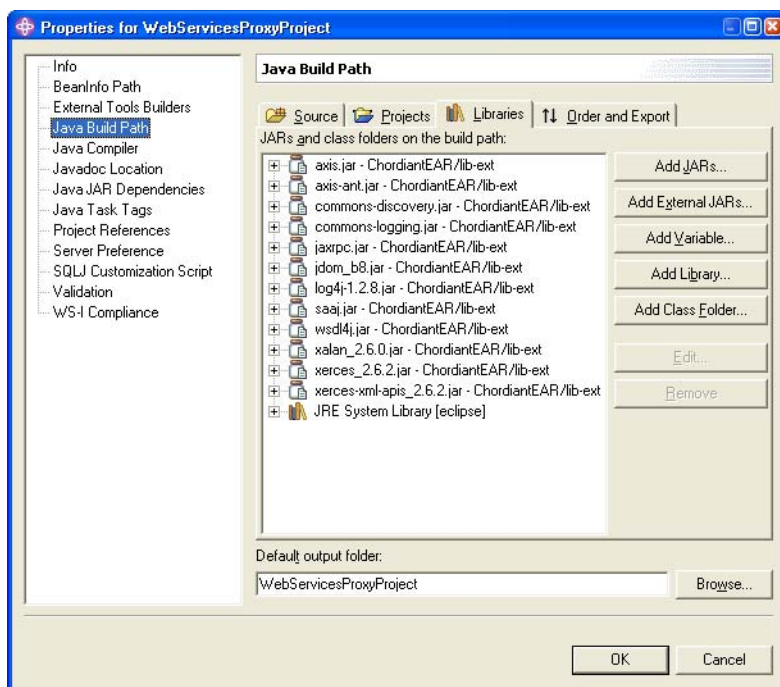


Figure 6-6: Final Java Build Path for the Java Proxy Project

Requirements for Creating Web Services

Web services are based on client agents. Since client agents are dependent on business objects, so are web services.

Business objects used in the services should be Java Beans, which means they should have:

- a default no-argument constructor
- getters and setters for all the attributes defined in the classes

These requirements are met when you model your objects in Rational Rose and use the Chordiant Business Component Generator to create your code. All business objects for the web services provided by Chordiant meet these requirements.

GENERATING CHORDIANT WEB SERVICES

Chordiant provides Ant scripts for generating web services components. Processes in this chapter describe running the Ant scripts through the development environment. If you choose, you can also run them through standard Ant commands on the command line:

```
ant -f {script name} {target list}
```

Specifying the WSDL Distribution Method

As part of the WSDL generation script, you must specify how you want to distribute the WSDLs. When you publish your WSDLs, you must tell people who will be using them how they can access the WSDLs. There are two possibilities:

- **Statically** — Generate the WSDL and expose it through any shared file system. This can include a shared hard drive or even a URL. Be aware that using a URL here is different from the dynamic distribution described in the next section. From this access point, a requester can access the WSDL and build Java proxies to interact with the service. However, since the WSDL is already generated, any change in the class files would require a new WSDL to be generated. Users who had accessed this static WSDL can easily get out of sync with the actual service code if the services are changed.
- **Dynamically** — Through an established URL, the WSDL is generated dynamically if the web service is deployed and the server is running. The WSDL is passed onto the requester's system. By requesting a dynamically-distributed WSDL, the user will receive the most up-to-date WSDL available. It is still possible to get out of sync with the actual service code. For example, if the actual service code is updated after the requester has obtained and used the WSDL, that user's code will be out of date and may not work appropriately.

The format for a dynamically distributed WSDL is:

```
http://{hostname}/WebServices/services/{ServiceName}?wsdl
```

where *{hostname}* is the name of the host where the service is deployed. If this is local, then the value will be `localhost`. The address will always end with `?wsdl`.

When you create the WSDLs through the `ws-java2wsdl.xml` Ant script, you specify the location of the WSDL. See [Code Sample 6-1 on page 100](#) for an example.

By default, all WSDLs provided by Chordiant have the address based on `localhost`. If you are not running from the `localhost`, change the value of the address element in the WSDL file itself or change the address in the `ws-java2wsdl.xml` Ant script and regenerate the WSDLs. In either case, if you have already generated proxies for the WSDLs that specify `localhost`, regenerate the proxies after you have updated or regenerated the WSDL files.

Handling Different Types of Services

You can create web services components for both services created from a model and the Business Component Generator and services created by hand.

Model-Based Services

To adhere to Chordiant's customization model, create your business services starting with a model, then use the Business Component Generator to create your code. Follow the instructions in:

- [Chapter 3, "Using the Business Component Generator"](#)
- [Chapter 4, "Creating or Modifying Business Services"](#)
- [Chapter 5, "Tutorial: Creating and Extending a Business Service"](#)

Make sure that you have created the customizations you require within your model, then follow the procedures to run the Business Component Generator. Also make sure that your `{Model}.descriptor.xml` file specifies that you are generating service components.

Note: Be sure to use a new name for your customized service. If you create customizations with the same name as Chordiant-provided services, your customizations might be overwritten if you install a new version of Chordiant. This includes web services files as well as basic Chordiant services.

Remember that the service file is generated as a skeleton and you must add the actual functionality to the methods you have specified. You might also need to update the generated configuration file, as described on [page 44](#).

You create the web services components after you have created the other business components, because web services are based on the client agent code. Follow the steps described in ["Generating Web Service Components"](#).

Services Without Models

Some services, like system services, do not have associated models. You can still generate web services components for them if you want others to have remote access to their functionality.

Follow the steps described in ["Generating Web Service Components"](#).

Generating Web Service Components

Chordiant uses Apache's Axis for web service infrastructure. This infrastructure requires several basic steps for creating and deploying web services. These steps are detailed throughout this chapter.

Basic steps for creating and deploying Chordiant web services:

1. Copy your source JAR files to the WEB-INF\lib directory. (Refer to [page 99](#).)
2. Edit and run the ws-java2wsdl.xml script to create the WSDL files. (Refer to [page 100](#).)
3. Edit and run the ws-wsdl2java.xml script to create the deployment descriptors (WSDD) and the proxy code. (Refer to [page 103](#).)
4. Start the server. If present, be sure to delete the server-config.wsdd file before restarting so you are starting with a clean system. (Refer to [page 106](#).)
5. Edit and run the ws-deploy.xml script for web services you want to deploy. This copies the deployment descriptor information to the server-config.wsdd file. Ensure that you are able to see the WSDL for the newly deployed service. (Refer to [page 106](#).)
6. Edit the ws-undeploy.xml script for any web services you want to undeploy. (Refer to [page 108](#).)
7. Edit the server-config.wsdd file and restart the server. (Refer to [page 111](#).)
8. Generate the proxies so you can test them. (Refer to [page 112](#).)
9. Write your application to test or use the proxies. Invoking web services is described in [Chapter 7, "Invoking and Integrating Web Services"](#).

Copying the JAR Files

Before you deploy the web services, you must make the underlying code available in the **WebServices** project.

To copy the JAR files:

1. Locate the service JAR files and business object JAR files from your project — either a **JX Business Components** project (for modeled services) or a **Java** project (for services without a model).
2. Copy them to the appropriate directory.
 - **WebSphere:** {WORKSPACE}\WebServices\WebContent\WEB-INF\lib
 - **WebLogic:** {WORKSPACE}\WebServices\WEB-INF\lib

Note: In general, any JARs that contain business services and their associated business objects that you would put in the ChordiantEAR\lib directory should be copied to the WEB-INF\lib directory.

Creating the WSDL Files

To generate the WSDL files from your Java code:

1. Open the **ChordiantUtils** project and navigate to the `WebServices` directory.
2. Open the `ws-java2wsdl.xml` file in the text editor. This Ant-based script generates WSDL documents from Java code.
3. Update the script with a new section for each new target. You can copy an existing section and modify it for the new or modified web service.

[Code Sample 6-1](#) shows a the setup parameters at the start of the script and then a sample target section for Chordiant's `PmfCustomer` client agent. The portions you must change are described in the column on the right. The `RpcEnc` suffix denotes that this WSDL is the RPC style.

Notes: The `ws-java2wsdl.xml` file is where you specify the location where you will publish the WSDL. This can be either a file path or a URL. Be sure that this location has proper access rights so people can access your WSDLs. For more information on distribution methods, refer to [“Specifying the WSDL Distribution Method” on page 97](#).

You must also edit the script with standard information about your development environment, like the base directory and location of your application server.

Weblogic property definitions

```
<!-- The project that represents the runtime workspace -->
<property name="workspace.runtime"
  value="../../../ChordiantEAR"/>
<property name="axis.lib" value="../WEB-INF/lib"/>

<property name="wsdl" value="../WSDL"/>

<property name="wsdd" value="../WSDD"/>

<property name="proxy"
  value="../../../WebServicesProxyProject"/>

...

<!-- The project that represents the runtime workspace -->
<property name="workspace.runtime"
  value="../../../ChordiantEAR"/>
<property name="axis.lib" value="../WEB-INF/lib"/>
<property name="wsdl" value="../WSDL"/>
<property name="wsdd" value="../WSDD"/>
```

Define properties for WebLogic. Properties for WebSphere are identical and are found later in this file.

Location of the ChordiantEAR project.

Location of the axis.lib directory. The web services infrastructure uses this directory for storing the web services components.

Location of the WSDL directory, that stores the WSDL files.

Location of the WSDD directory, that stores the WSDD files.

Location of the proxy project you created in [“Creating a Java Proxy Code Project” on page 91](#).

Code 6-1: Parameter Definitions and Sample Target Entry in ws-java2wsdl Script

<pre> <property name="proxy" value="../../../WebServicesProxyProject"/> <!-- ===== PmfCustomerClientAgentRpcEnc ===== --> <target name="PmfCustomerClientAgentRpcEnc" depends="_init" description="Java to WSDL for PmfCustomerClientAgentRpcEnc"> <java fork="true" classname="org.apache.axis.wsdl.Java2WSDL" classpathref="classpath.tools" jvm="\${tools.jvm}" > <arg value="-o"/> <arg value="\${wsdl}/PmfCustomerClientAgentRpcEnc.wsdl"/> <arg value="-l"/> <arg value= "http://localhost/WebServices/services/ PmfCustomerClientAgentRpcEnc"/> <arg value="-P"/> <arg value="PmfCustomerClientAgent"/> <arg value="-b"/> <arg value="PmfCustomerClientAgent"/> <arg value="-S"/> <arg value="PmfCustomerClientAgentService"/> <arg value="-s"/> <arg value="PmfCustomerClientAgentRpcEnc"/> <arg value="-e"/> <arg value= "com.chordiant.pmf.businessClasses.Person, com.chordiant.customer.businessClasses.Customer"/> <arg value= "com.chordiant.customer.client.PmfCustomerClientAgent"/> </java> </target> </pre>	<p>end of header information</p> <p>Add a descriptive heading comment</p> <p>Specify a unique target name and description for creating a WSDL from Java code.</p> <p>Calls the web services infrastructure's java2wsdl tool, with the required classpath. Do not change this line.</p> <p>Specify the name and location for new WSDL file.</p> <p>Specify the URL where you want people to access the WSDL.</p> <p>Specify the port type name for the client agent. This is the client agent name.</p> <p>Specify the binding name for the WSDL. This is the client agent name.</p> <p>Specify the service element name used for the WSDL. This is the client agent name with a "Service" suffix.</p> <p>Specify the service port name. This is the client agent name with "RpcEnc" as a suffix, indicating that this is an Rpc-type web service.</p> <p>Some classes use objects in an indirect manner. For example, when a method parameter specifies an interface, you can pass any object that implements this interface as an argument. In this case, the actual object that is passed as an argument is not defined in the WSDL file. Such undefined objects cannot be serialized and deserialized. Therefore, it is necessary to identify and specify all such objects using the -e option.</p> <p>Specify a space- or comma-separated list of fully-qualified class names for business objects required for the interface.</p>
--	--

Code 6-1: Parameter Definitions and Sample Target Entry in ws-java2wsdl Script (Continued)

4. Save and close the ws-java2wsdl.xml file.
5. Right-click this newly-modified ws-java2wsdl.xml file and select **Run Ant**.

- In the **Modify Attributes and Launch** window, select the **Targets** tab if it is not already selected. Select the checkboxes next to the client agents for which you want to create web services components. Only client agents that you specified in [Step 3 on page 100](#) will be available as targets.

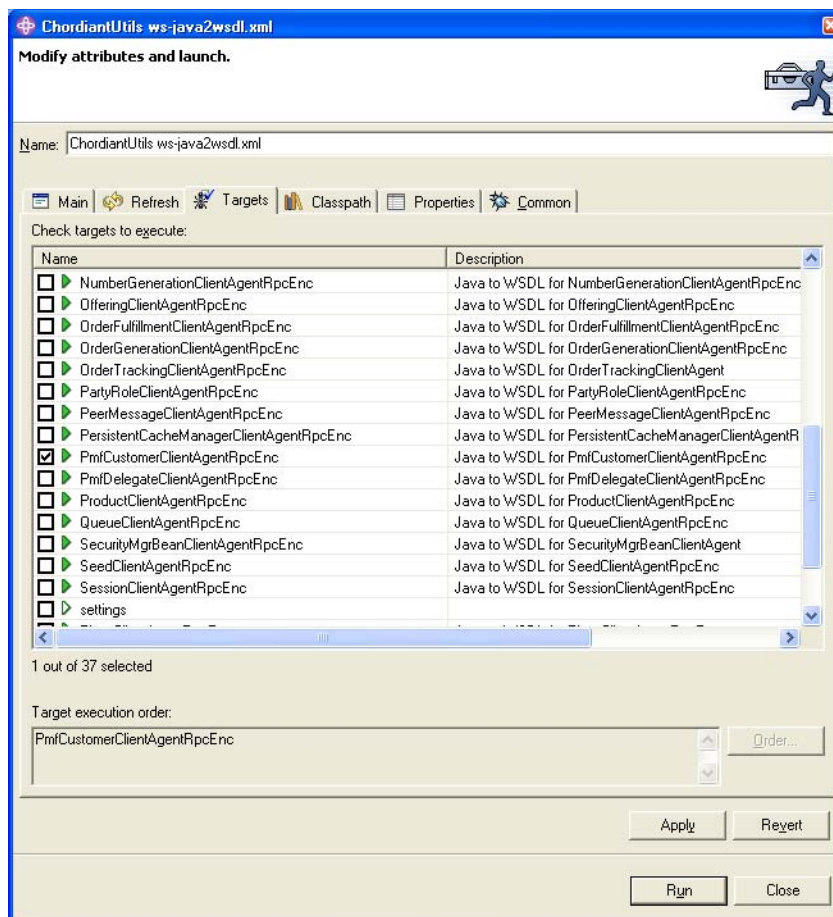


Figure 6-7: Specifying Client Agents in the ws-java2wsdl.xml Script

- Click **Apply** to save the changes and click **Run** to close the dialog and run the script.

You might see one or more black command windows appear during the process. This is normal. You can watch the **Server Console** if you want to make sure that the script ran successfully.

The WSDLs are generated and are placed into **ChordiantUtils** directory in the **WebServices/WSDL** directory.

- To see the new WSDLs in the **ChordiantUtils** project, you must refresh the WSDL directory.

Creating the Deployment Descriptor and Proxy Files

Before you can deploy the web services files, you must create the WSDD files.

Note: To start with a clean version, delete any existing `server-config.wsdd` files before performing these steps.

To create the deployment descriptor and proxy files:

1. Locate the WSDL you want to use. In this case, you will be using the **PmfCustomer** WSDL you just created on your local machine. Note its location. You will point to the location of this WSDL when you update the script in [Step 3](#).
2. In the **ChordiantUtils** project, navigate to the `WebServices` directory and open the `ws-wsdl2java.xml` file. This Ant-based script generates Java proxies from WSDL files.
3. Update the script with changes to the properties definitions at the top and with a new section for each new WSDL target. [Code Sample 6-2](#) shows a sample target section for Chordiant's **PmfCustomer** client agent. The portions you must change are described in the column on the right. The `arg` values are detailed in the *Axis Reference Guide*, located at <http://ws.apache.org/axis/java/reference.html>.

Notes: This script includes definitions at the top of the file. These are the same definitions described in [Code Sample 6-1 on page 100](#).

You must also edit the script with standard information about your development environment, like the base directory and location of your application server.

<pre><!-- ===== PmfCustomerClientAgentRpcEnc ===== --> <target name="PmfCustomerClientAgentRpcEnc" depends="_init" description="WSDL to Java for PmfCustomerClientAgentRpcEnc"> <java fork="true" classname="org.apache.axis.wsdl.WSDL2Java" classpathref="classpath.tools" jvm="\${tools.jvm}" > <arg value="-o"/> <arg value="\${proxy}"/> <arg value="\${wsdl}/PmfCustomerClientAgentRpcEnc.wsdl"/> </pre>	<p>Add a descriptive heading comment</p> <p>Specify a unique target name and description for creating Java code from the WSDL.</p> <p>Calls the web services infrastructure. Do not change this line.</p> <p>Specify location and name for new proxy files. This is in the proxy project you created in the Tools Platform.</p> <p>Specify the location of the WSDL file you acquired. This example uses a static WSDL, but you can also use a URL to obtain a dynamically-generated WSDL file. See “Specifying the WSDL Distribution Method” on page 97 for additional details.</p>
---	--

Code 6-2: Sample `ws-wsdl2java.xml` File

```
<arg value="-d"/>
<arg value="Session"/>

<arg value="-s"/>
<arg value="-S"/>
<arg value="true"/>
</java>

<copy file=
"${proxy}/com/chordiant/customer/client/deploy.wsdd"
tofile="${wsdd}/deploy_PmfCustomerClientAgentRpcEnc.wsdd"
overwrite="true"/>
<copy file=
"${proxy}/com/chordiant/customer/client/undeploy.wsdd"
tofile="${wsdd}/undeploy_PmfCustomerClientAgentRpcEnc.wsdd"
overwrite="true"/>

<delete file=
"${proxy}/com/chordiant/customer/client/deploy.wsdd"/>
<delete file=
"${proxy}/com/chordiant/customer/client/undeploy.wsdd"/>
</target>
```

Adds this session to the deploy scope. Use the format shown here.

Emits server-side bindings for web service
The two arg values of -S and true give direction to generate a skeleton in the deploy.wsdd file.

Copies the deploy and undeploy WSDD files from your proxy project into the WSDD directory. These files have generic names and will be overwritten in this location.
Provide the WSDD files with unique names so they are easily identified.

Deletes the deploy and undeploy WSDD files from your proxy project. They are no longer needed here.

Code 6-2: Sample ws-wsdl2java.xml File

4. Save and close the ws-wsdl2java.xml file.
5. Right-click this newly-modified ws-wsdl2java.xml and select **Run Ant**.

6. In the **Modify Attributes and Launch** window, select the **Targets** tab if it is not already selected. Select the checkboxes next to the client agents for which you want to create Java proxies. Only WSDLs that you specified in [Step 3](#) will be available as targets.

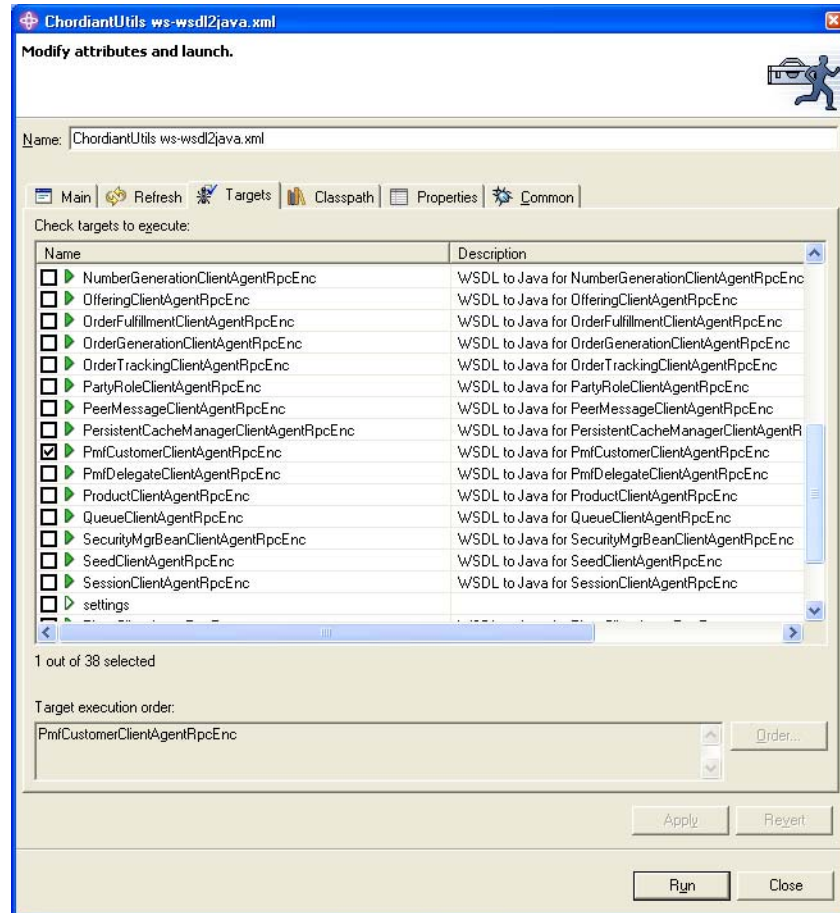


Figure 6-8: Specifying WSDLs in the ws-wsdl2java.xml Script

7. Click **Apply** to save the changes and click **Run** to close the dialog and run the script.

You might see one or more black command windows appear during the process. This is normal. You can watch the **Server Console**, if you want to make sure that the script ran successfully.

8. Refresh the **ChordiantUtils** project to see the newly-generated WSDD files. Refresh the proxy project to see the new Java proxy files. The WSDD files are placed in the WSDD directory, next to the WSDL directory. The Java proxies are placed into the directory you specified in [“Creating a Java Proxy Code Project” on page 91](#).

The proxy code in the directory includes one or more packages for the code that mirror the packages for the original code. In addition to any service-specific files, the following SOAP files are generated for each WSDL:

- `{ServiceName}ServiceLocator.java`
- `{ServiceName}SoapBindingImpl.java`
- `{ServiceName}Skeleton.java`
- `{ServiceName}SoapBindingStub.java`

For Chordiant web services, a client agent and service file are also created.

Starting the Chordiant Server

You must start the Chordiant server to deploy the web services you have just created.

To start with a clean system without any web services deployed, be sure to delete the `server-config.wsdd` configuration file before starting the server. The `server-config.wsdd` file is located in this directory:

- **WebSphere** — `{WORKSPACE}\WebServices\WebContent\WEB-INF`
- **WebLogic** — `{WORKSPACE}\WebServices\WEB-INF`

Deploying Web Services

Deploying web services updates the `server-config.wsdd` file, which controls which web services are available through the server. If the `server-config.wsdd` file does not already exist, running the deployment script creates this file.

To deploy a web service:

1. Start the server.
2. Open the **ChordiantUtils** project. Navigate to the `WebServices` directory.
3. Open the `ws-deploy.xml` file in the text editor. This is the Ant-based script that deploys WSDLs.
4. Update the script with a new section for each WSDD deploy file. [Code Sample 6-3](#) shows a sample target section for `PmfCustomerClientAgentRpcEnc`. The portions you must change are described in the column on the right.

Notes: The deploy script includes definitions at the top of the file. These are the same definitions described in [Code Sample 6-1 on page 100](#).

You must also edit the script with standard information about your development environment, like the base directory and location of your application server.

<pre><!-- ===== PmfCustomerClientAgentRpcEnc ===== --></pre>	Add a descriptive heading comment.
<pre><target name="PmfCustomerClientAgentRpcEnc" depends="_init" description="Deploy PmfCustomerClientAgentRpcEnc"></pre>	Specify a unique target name and description for deploying.
<pre><java fork="true" classname="org.apache.axis.client.AdminClient" classpathref="classpath.tools" jvm="\${tools.jvm}" ></pre>	Calls the web services infrastructure. Do not change this line.
<pre><arg value="-lhttp://localhost/WebServices/ services/AdminService"/></pre>	Specify the location where the AdminService is deployed. The AdminService is a deployed web service deploys other web services.
<pre><arg value="\${wsdd}/deploy_PmfCustomerClientAgentRpcEnc.wsdd"/> </java> </target></pre>	Specify the location of the deploy WSDD file for this web service. The WSDD value is described at the top of the file, as described in Code Sample 6-1 on page 100 .

Code 6-3: Sample Target Entry in ws-deploy Script

5. Save and close the ws-deploy.xml file.
6. Right-click this newly-modified ws-deploy.xml and select **Run Ant**.

7. In the **Modify Attributes and Launch** window, select the **Targets** tab if it is not already selected. Select the checkboxes next to the web services you want to deploy. Only web services that you specified in [Step 4 on page 106](#) will be available as targets.

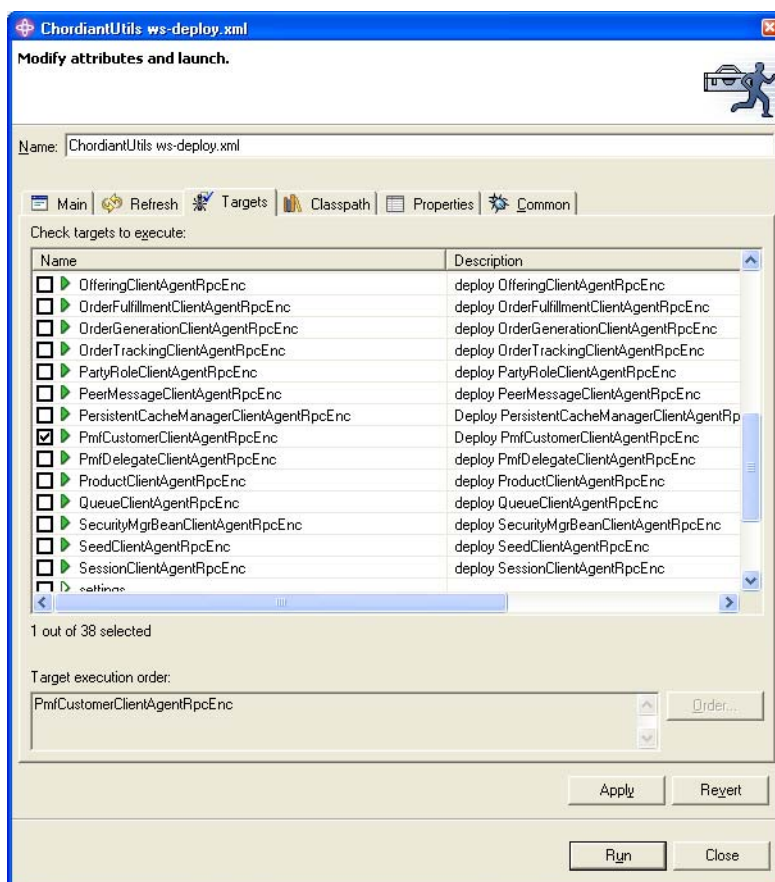


Figure 6-9: Specifying WSDLs to Deploy in the ws-deploy.xml Script

8. Click **Apply** to save the changes and click **Run** to close the dialog and run the script.
9. Watch the **Service Monitor** to see that the deployment was successful.

Undeploying Web Services

When you no longer want a web service to be accessible, you can undeploy it. This process is very similar to the deployment process and is performed only after a web service is currently deployed.

To undeploy a web service:

1. Start the server.
2. In the Chordiant Tools Platform, open the **ChordiantUtils** project. Navigate to the WebServices directory.

3. Open the `ws-undeploy.xml` file in the text editor. This Ant script undeploys WSDLs, that is, removing WSDLs that are currently deployed, making them unavailable to potential users.
4. Update the script with a new section for each WSDD undeploy file. [Code Sample 6-4](#) shows a sample target section for `PmfCustomerClientAgentRpcEnc`. The portions you must change are described in the column on the right.

Notes: The undeploy script includes definitions at the top of the file. These are the same definitions described in [Code Sample 6-1 on page 100](#).

You must also edit the script with standard information about your development environment, like the base directory and location of your application server.

<pre> <!-- ===== PmfCustomerClientAgentRpcEnc ===== --> <target name="PmfCustomerClientAgentRpcEnc" depends="_init" description="Deploy PmfCustomerClientAgentRpcEnc"> <java fork="true" classname="org.apache.axis. client.AdminClient" classpathref="classpath.tools" jvm="\${tools.jvm}" > <arg value="-lhttp://localhost/WebServices/ services/AdminService"/> <arg value="{wsdl}/ undeploy_PmfCustomerClientAgentRpcEnc.wsdd"/> </java> </target> </pre>	<p>Add a descriptive heading comment</p> <p>Specify a unique target name and description for the service to be undeployed.</p> <p>This line calls the web services infrastructure. Do not change this line.</p> <p>Specify the location where the AdminService is deployed. The AdminService is a deployed web service used to deploy other web services.</p> <p>Specify the location of the undeploy WSDD file for this web service.</p>
--	---

Code 6-4: Sample Target Entry in ws-undeploy Script

5. Save and close the `ws-undeploy.xml` file.
6. Right-click this newly-modified `ws-undeploy.xml` and select **Run Ant**.

7. In the **Modify Attributes and Launch** window, select the **Targets** tab if it is not already selected. Select the checkboxes next to the web services you want to undeploy. Only web services that you specified in [Step 4 on page 109](#) will be available as targets.

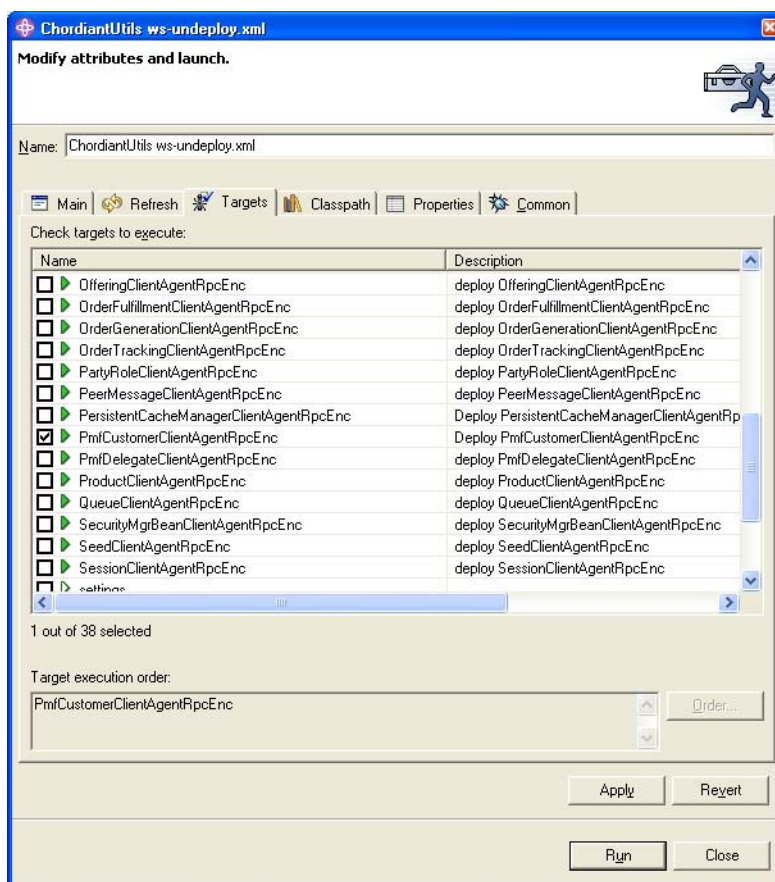


Figure 6-10: Specifying WSDLs to Deploy in the ws-undeploy.xml Script

8. Click **Apply** to save the changes. Click **Run** to close the dialog and run the script.
9. Watch the **Service Monitor** to verify that the undeployment was successful.

Editing the server-config.wsdd File and Restarting the Server

You must edit the `server-config.wsdd` file to ensure that the web services and proxies work correctly.

Tips: If you choose, you can copy information from your modified WSD file and paste it into the `server-config.wsdd` file.

If the server is running when you modify this file, you must restart the server so it can use the new version of the `server-config.wsdd` file.

If you are working with just one or two web services, you can avoid restarting the server by making these changes in the `deploy.wsdd` file instead of the `server-config.wsdd`.

To edit the `server-config.wsdd` file:

1. Locate the `server-config.wsdd` file in the **WebServices** project:
 - **WebSphere:** within the `WebContent/WEB-INF` directory
 - **WebLogic:** within the `WEB-INF` directory

Open it in a text editor. This can be the text editor available through the Chordiant Tools Platform.

2. Perform a global search and replace, replacing the string “skeleton” with nothing, effectively removing that string throughout the document.

Note: [Step 2](#) is mandatory. If you do not perform this step, web services publishing will not work.

3. Save and close the `server-config.wsdd` file.
4. Restart the server.

Modifying WSDL Files

With standard usage, you do not need to modify WSDL files. If you do need to modify a WSDL file, you can use any text editor, including the text editor available through the Chordiant Tools Platform.

Restarting the Server

Once you have edited the `server-config.wsdd` file, restart the server so it will pick up the your changes.

Refresh the workspace to see the new and refreshed files:

- **ChordiantUtils** project — WSDL and WSDD files
- **WebServicesProxyProject** — proxy files
- **WebServices** project — `server-config.wsdd` file

Generating the Proxies Again and Testing Them

To see how the deployed web services are working, regenerate the proxies to be sure they are based on the deployed services, following the directions in [“Creating the Deployment Descriptor and Proxy Files” on page 103](#). Then use the `ws-wsd12java-dynamic.xml` Ant script to access the web services from the running server.

Note: These steps are only required when you generate proxies from a dynamic environment. In other cases, you can disregard this section.

To test deployed web services on a running server:

1. Open both the `ws-wsd12java.xml` and the `ws-wsd12java-dynamic.xml` files.
2. In the `ws-wsd12java.xml` file, locate the section you created for your `PmfCustomer` web service. Copy it and paste it into the `ws-wsd12java-dynamic.xml` file.
3. Change the argument for the WSDL location from the file path location to the location on the running server.
4. Save the script and run it as you ran the `ws-wsd12java.xml` script, described in [Step 6 on page 105](#).

You are now ready to create a tester application to make sure that your web services files are operating as intended. Examples of tester applications are described in [Chapter 7, “Invoking and Integrating Web Services”](#).

LISTING AVAILABLE WEB SERVICES COMPONENTS

You might want to see a list of available web services to make sure that you have deployed all of the required WSDLs or perhaps to provide a list of web services to prospective users.

To view a list of deployed WSDLs while a Chordiant system is operating:

1. Open a browser.
2. Enter this URL:

`http://{HOSTNAME}/WebServices/servlet/AxisServlet`

You will see a list of the deployed services within the browser.

To view a list of deployed WSDLs, while a Chordiant system is *not* operating:

1. If it is not already open, launch the Chordiant Tools Platform.
2. Open the `server-config.wsdd` file, located in this directory:
 - **WebSphere:** `{WORKSPACE}\WebServices\WebContent\WEB-INF`
 - **WebLogic:** `{WORKSPACE}\WebServices\WEB-INF`

You will see a list of the deployed services within the text editor. This file does not display the services in a clean list like you would see using the URL. For documentation of the elements and attributes in the `server-config.wsdd` file, refer to the Axis documentation at <http://ws.apache.org/axis/java/reference.html>.

DEPLOYING WEB SERVICES TO A PRODUCTION ENVIRONMENT

Deploying web services to a production environment is done the same way as deploying other Chordiant projects. Refer to the *Chordiant 5 Applications Deployment Guide* for detailed steps on the deployment process.

In general, deploying web services to a production environment is a two-step process:

1. Generate a WAR file containing web services and Application Packaging Manager components, using the `build.xml` file in the **WebServices** project. In WebSphere, it is contained within the `WebContent` directory.
2. Generate an EAR file that includes the web services WAR file from [Step 1](#) and configures all WSDL URL addresses.

Before You Deploy

Remember that your WSDL and proxy files were originally created using `localhost` for the host name. You must update the files to use the actual host name instead of `localhost`. Perform one of the following:

- Go back and regenerate your WSDL and proxy files, running both `ws-java2wsdl.xml` (refer to [page 100](#)) and `ws-wsdl2java.xml` (refer to [page 103](#)). In the scripts, substitute the actual host name on your production environment for `localhost`. When you deploy the **WebServices** project as a WAR file, these new WSDL and proxy files will be included.
- OR
- Deploy the web services on your production environment and access them from the running server, using `ws-wsdl2java-dynamic.xml` (refer to [page 112](#) and to [page 121](#)). In this case, open the `ws-wsdl2java-dynamic.xml` file and globally replace `localhost` with the actual host name on your production environment.

Deploying Individual Web Services in Production

Ideally, you have performed all of your testing in the development environment and have deployed the required web services. In this case, the `server-config.wsdd` file that is deployed in the **WebServices** WAR file already contains all of the deployment information, so the appropriate web services will be deployed on the production environment.

If you need to deploy one or two additional web services directly in the production environment, edit the `deploy.wsdd` file to remove the string “skeleton”. Save the file. By editing the `deploy.wsdd` file instead of the `server-config.wsdd` file, there is no need to restart the server.

ERROR HANDLING

Error handling in Chordiant web services is similar to error handling in the rest of the Chordiant 5 Foundation Server. For details on this subject, refer to the “Error Handling” section of the *Chordiant 5 Foundation Server Developer’s Guide*.

BEST PRACTICES

Chordiant recommends these best practices for using Chordiant web services.

Using Java Code

Chordiant web services are intended for use when you don't have access to Chordiant code and by applications using languages other than Java. If you are using Java code to access Chordiant services and you have access to the Chordiant code, use standard Chordiant client agents, rather than Chordiant web services. It is a more direct path and provides you with full Chordiant functionality, including built-in transaction handling and security.

Use arrays in your Java code instead of sets or hash tables. Arrays are accepted in other languages, so they can be easily interpreted by other systems accessing Chordiant web services.

Calling External Web Services from Chordiant Applications

Make all calls to external web services from within a Chordiant service. Do not make these calls from a custom object or within the EJB or servlet/JSP layers. Wrapping the call within a Chordiant service provides you with the benefits of the Chordiant architecture and ensures that the call is handled appropriately.

Regenerating and Redeploying WSDLs

Regenerate and redeploy WSDLs whenever you:

- add a new attribute on a business object
- add or modify a method signature

Whenever you make these changes, you must manually update the web services components to provide users with the latest code.

Invoking and Integrating Web Services

[Chapter 6](#) describes how to create and deploy web services. This chapter explains how you can use these web services in Chordiant and non-Chordiant systems. This chapter includes tutorials and sample applications that demonstrate how to invoke a web service and integrate it into your application code.

For details on creating and deploying web services, refer to [Chapter 6, “Creating Web Services”](#).

This book is not designed to provide a lot of general background information on web services or invoking web services. There is plentiful background information on this subject on the internet.

STATIC AND DYNAMIC INVOCATION

There are two ways to invoke web services — static invocation and dynamic invocation.

- *Static* invocation involves using Java proxies that are created from a WSDL you obtain. The service itself can be either static or dynamic. The static here refers to the proxies, because once you have created these proxies, they are static. If the WSDL changes, you must recreate your proxies. Static invocation is best suited for business-type services, which rarely change and do not require the overhead of a dynamic interface. For more information on static invocation, refer to the next section, [“Invoking Web Services Statically” on page 120](#).
- *Dynamic* invocation involves creating code in your service while you are contacting the WSDL. Dynamic invocation is best suited for engine-type services, such as a workflow engine, that should not tied down to static code.

SAAJ (SOAP with Attachments API for Java) is a type of dynamic invocation that enables you to create, send, and read XML messages over the internet using Java. For details on using SAAJ, refer to [“Invoking Web Services Dynamically through SAAJ” on page 144](#).

Tips: The sample applications described in the tutorials in this chapter are provided for your reference. Instructions for installing the sample applications code are provided in the next section, [“Installing the Sample Applications for Web Services” on page 118](#).

The code samples in this chapter are broken up into steps with commentary. The complete, uninterrupted code is part of the sample applications you install, as described in the next section.

You can run the provided sample code to see the outcome or you can follow along with the tutorials and run your own code that you create.

Installing the Sample Applications for Web Services

To use the sample applications in this chapter, you must install the sample application code.

To install the sample application code:

1. On the Chordiant Tools Platform's **File** menu, select **New Project**.
2. Create a new **Samples** project. On the left side, select **Samples**. On the right side, select **Web Services Chordiant Samples**. Click **Next** to continue.

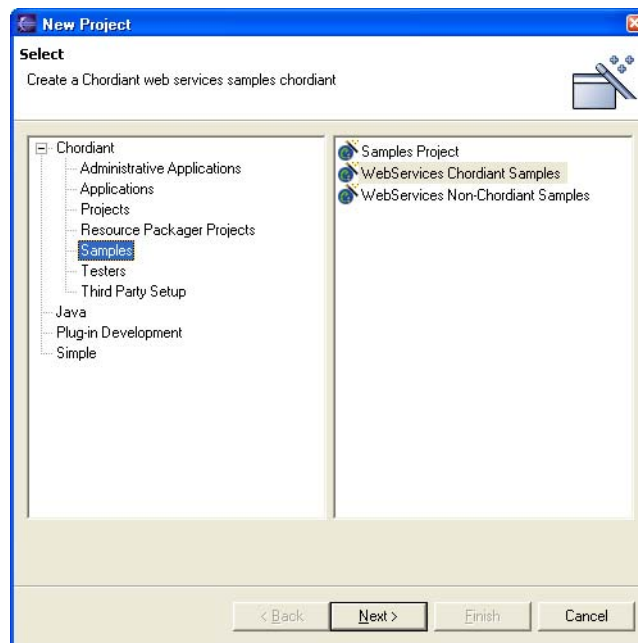


Figure 7-1: Creating a New Web Services Samples Project

3. Name your new project. In the examples in this chapter, we use the name `WebServicesSamples_Chordiant`. Click **Finish** to create the project.

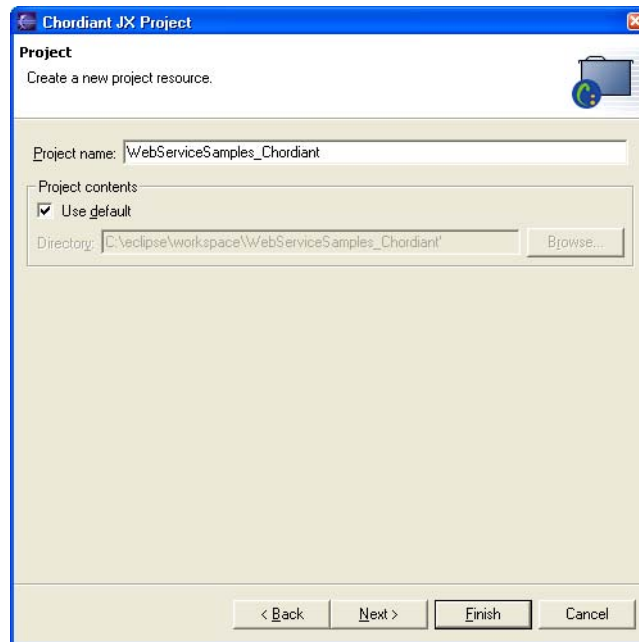


Figure 7-2: Naming the Sample Code Project

All of the code required for the Chordiant sample application is loaded in the project you specify.

4. On the left side, select **Java Build Path**. On the right side, click **Add JARs**. Add the same JAR files you added in [Step 6 on page 95](#).
5. Repeat [Step 1](#) through [Step 4](#) to install the sample application that uses a non-Chordiant WSDL. Select **Web Services Non-Chordiant Samples** and name your project `WebServicesSamples_Non-Chordiant`.

Note: For the non-Chordiant sample project, you must explicitly avoid having **WebServicesProxyProject** in the build path, but include `ChordiantEar/lib` folder in the build classpath.

You should now be able to see both of the samples projects in your workspace, as illustrated in [Figure 7-3](#).

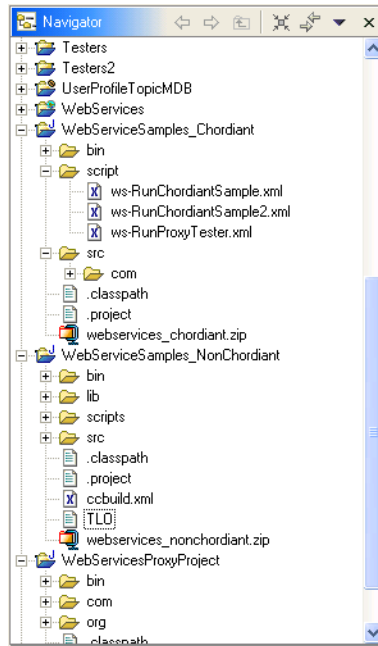


Figure 7-3: Workspace Showing Sample Applications Projects, WebServices Project, and the WebServicesProxyProject

To avoid potential compilation errors in the **WebServicesSample_Chordiant** project, you should have already created proxies for **SecurityMgrBeanClientAgent** and **PmfCustomerClientAgent** in [Chapter 6](#). If you have not already done so, please create these required proxies for these client agents by following the steps in [“Creating the WSDL Files” on page 100](#) and [“Creating the Deployment Descriptor and Proxy Files” on page 103](#).

INVOKING WEB SERVICES STATICALLY

To use a web service, you invoke it from either a running server or from a file system, then create Java proxy code from it. You can then use this proxy code within a Chordiant service to perform work.

Invoking Chordiant Web Services

This tutorial explains how to use a static WSDL to use Chordiant’s **PmfCustomer** service to locate a customer, based on their customer number. The application must first make a call to the security client agent, **SecurityMgrBeanClientAgent**, to obtain an authentication token before contacting the Chordiant service.

Using non-Chordiant web services is a similar process. For details, refer to [“Using Non-Chordiant Web Services Within Chordiant” on page 129](#).

Invoking a Chordiant web service includes the following general steps:

1. Create the proxy project. You should have already created this project, as described in [“Creating a Java Proxy Code Project” on page 91.](#)
2. Create the proxy code from the WSDL file. Described in [“Creating the Proxy Code for the PmfCustomer Web Service” on page 121.](#)
3. Integrate the proxy code into your application code. Described in [“Incorporating the Chordiant Proxy Code” on page 124.](#)

Creating the Proxy Code for the PmfCustomer Web Service

This example uses a Chordiant web service, but this same process works for non-Chordiant web services.

Note: If you are regenerating proxies that you have already created, for example in testing scenarios or because you have received a new WSDL, it is good practice to clear out all of the existing proxy code from your **WebServicesProxyProject** and regenerate it all again.

To access a web service:

1. Locate the WSDL you want to use. This can be from a file location or from a URL. In this tutorial, you will be using Chordiant’s PmfCustomer web service.
2. In the Chordiant Tools Platform, open the **ChordiantUtils** project and locate the appropriate `{wsdl2java}` script within the `WebServices` directory.
 - If you have the WSDL locally, edit the `ws-wsdl2java.xml` file. This script is used most often during active development.
 - If you are accessing the WSDL from a running server, edit the `ws-wsdl2java-dynamic.xml` file. This script is used when accessing a dynamic web service located on a running server and when testing what the end user will receive when accessing a WSDL.

These scripts are identical. They serve as a way to separate accessing static and dynamic WSDLs and save you from retyping the location argument in the `ws-wsdl2java.xml` file.

3. Update the appropriate script with a new section for each new WSDL target. [Code Sample 7-1](#) shows a sample target section for Chordiant's PmfCustomer client agent. This is an example for a Chordiant system. Using a WSDL from a non-Chordiant site would be similar. Refer to ["Using Non-Chordiant Web Services Within Chordiant" on page 129](#) for details.

The portions you must change are described in the column on the right. The arg values are detailed in the *Axis Reference Guide*, located at <http://ws.apache.org/axis/java/reference.html>.

<pre> <!-- ===== PmfCustomerClientAgentRpcEnc ===== --> <target name="PmfCustomerClientAgentRpcEnc" depends="_init" description="WSDL to Java for PmfCustomerClientAgentRpcEnc"> <java fork="true" classname="org.apache.axis.wsdl.WSDL2Java" classpathref="classpath.tools" jvm="\${tools.jvm}" > <arg value="-o"/> <arg value="\${proxy}"/> <arg value="\${wsdl}/PmfCustomerClientAgentRpcEnc.wsdl"/> <arg value="-d"/> <arg value="Session"/> <arg value="-s"/> <arg value="-S"/> <arg value="true"/> </java> <copy file="\${proxy}/com/chordiant/customer/client/ deploy.wsdd" tofile="\${wsdd}/deploy_PmfCustomerClientAgentRpcEnc.wsdd" overwrite="true"/> <copy file="\${proxy}/com/chordiant/customer/client/ undeploy.wsdd" tofile="\${wsdd}/undeploy_PmfCustomerClientAgentRpcEnc.wsdd" overwrite="true"/> <delete file= "\${proxy}/com/chordiant/customer/client/deploy.wsdd"/> <delete file= "\${proxy}/com/chordiant/customer/client/undeploy.wsdd"/> </target> </pre>	<p>Add a descriptive heading comment</p> <p>Specify a unique target name and description for creating Java code from the WSDL.</p> <p>Calls the web services infrastructure. Do not change this line. Specify location and name for new proxy files. This is in the proxy project you created in the Tools Platform. The definition for the proxy is in the beginning of this script file, as described in Code Sample 6-1 on page 100.</p> <p>Specify the location of the WSDL file you acquired. Here, a static WSDL is used. You can also use a URL to obtain a dynamically-generated WSDL file. See "Specifying the WSDL Distribution Method" on page 97 for additional details. The wsdl value is defined at the beginning of this script file.</p> <p>Specify the deploy scope. For all services, this will be "Session".</p> <p>Emit server-side bindings for web service. The two arg values of -S and true give direction to generate a skeleton in the deploy.wsdd file.</p> <p>Copy the WSDD files - both deploy and undeploy - from your proxy project into the WSDD directory. They have generic names and will be overwritten in this location. Provide the WSDD files with unique names so they are easily identified.</p> <p>Delete the WSDD files - both deploy and undeploy - from your proxy project. They are no longer needed here.</p>
---	--

Code 7-1: Sample ws-wsdl2java.xml File

4. Save and close the ws-wsdl2java.xml file you just edited.
5. Right-click this newly-modified XML file and select **Run Ant**.

6. In the **Modify Attributes and Launch** window, select the **Targets** tab if it is not already selected. Select the checkboxes next to the client agents for which you want to create Java proxies. Only the WSDLs that you specified in [Step 3 on page 122](#) will be available as targets.

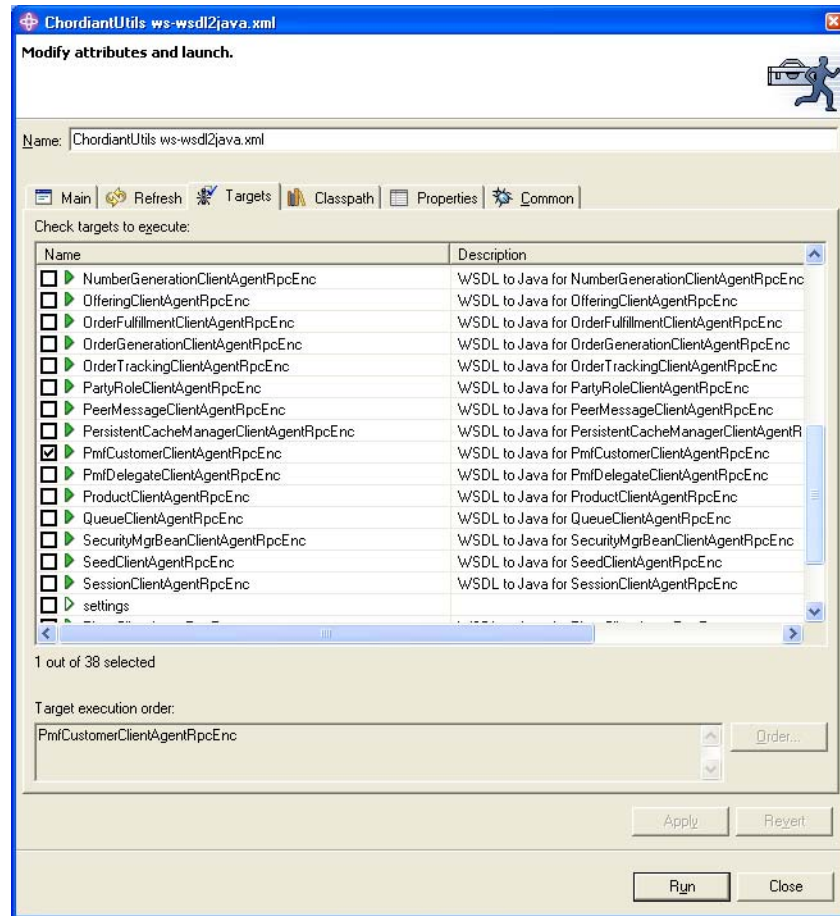


Figure 7-4: Specifying WSDLs in the ws-wsdl2java.xml Script

7. Click **Apply** to save the changes and click **Run** to close the dialog and run the script.

You might see one or more black command windows appear during the process. This is normal. You can watch the **Server Console** to verify that the script ran successfully.

The Java proxies are generated and are placed into the directory you specified in [“Creating a Java Proxy Code Project” on page 91](#). The WSDLs are placed in a WSDL directory alongside the specified WSDL directory. Refresh the Java proxy project to make sure you can see the new proxies. Refresh the **ChordiantUtils** project to make sure you see all of the new WSDL files.

The proxy code in the directory includes one or more packages for the code that mirror the packages for the original code. In addition to any service-specific files, the following SOAP files are generated for each WSDL:

- `{ServiceName}ServiceLocator.java`
- `{ServiceName}SoapBindingImpl.java`
- `{ServiceName}Skeleton.java`
- `{ServiceName}SoapBindingStub.java`

For Chordiant web services, a client agent and service file are also created.

Incorporating the Chordiant Proxy Code

You now have a Java proxy for the web service.

- If you are incorporating Chordiant code into a non-Chordiant system, use the best practices for your application when you add the proxy code to it.
- If you are working within a Chordiant system, incorporate non-Chordiant proxy code into your Chordiant service code to perform work. Wrapping the code within a service is important to make sure the call is handled appropriately. For more details, refer to the [“Best Practices”](#) section about [“Calling External Web Services from Chordiant Applications” on page 115](#).

When using object-oriented coding methods of static invocation, you hardly notice that you are using web services.

To incorporate the proxy code into a basic Java application:

1. Include the standard package declaration. Add import statements for all of the packages that you obtained when you obtained the proxy code. Refer to [Step 7](#) in the previous section.

```
package com.chordiant.axis.webservices.testner;  
  
import com.chordiant.customer.businessClasses.Customer;  
import com.chordiant.userprofile.ejb.security.client.SecurityMgrBeanClientAgent;  
import com.chordiant.userprofile.ejb.security.client.SecurityMgrBeanClientAgentServiceLocator;  
import com.chordiant.customer.client.PmfCustomerClientAgentServiceLocator;  
import com.chordiant.customer.client.PmfCustomerClientAgent;  
import com.chordiant.pmf.businessClasses.CommonObject;
```


2. Declare the class and start the main section. Create null placeholders for the customer object and authentication token, which will hold the returns from the `authenticate` and `getCustomer` method calls. Assign the standard Chordiant username and password values so the test authentication will succeed.

```
public class PmfCustomerWithProxyTester {

    public static void main(String[] args) throws Exception
    {
        Customer aCustomer = null;
        String authentication = null;
        String userID = "hmonroe";
        String password = "hmonroe";
        System.out.println("This is a proxy based tester");
    }
}
```

3. In a `try` block, instantiate both the `SecurityMgrBeanClientAgentServiceLocator` and the `PmfCustomerClientAgentServiceLocator` classes. These classes were created when you generated Java proxies from the WSDL file.

```
try
{
    SecurityMgrBeanClientAgentServiceLocator securityLocator =
        new SecurityMgrBeanClientAgentServiceLocator();
    System.out.println("Step 1: Created Security Locator.");
    PmfCustomerClientAgentServiceLocator customerLocator = new PmfCustomerClientAgentServiceLocator();
    System.out.println("Step 2: Created Customer Locator.");
}
```

4. Get client side proxies to the `SecurityMgrBeanClientAgent` and to the `PmfCustomerClientAgent`, based on the locator files you just created. The proxies will call the actual Chordiant client agents.

```
SecurityMgrBeanClientAgent securityCA = securityLocator.getSecurityMgrBeanClientAgentRpcEnc();
if ( securityCA == null )
{
    System.out.println("Step 3: Failed to Create SecurityMgrBeanClientAgent.");
    System.out.println("Exiting Tester.");
    System.exit(0);
}
else
{
    System.out.println("Step 3: Created SecurityMgrBeanClientAgent.");
}

PmfCustomerClientAgent customerCA = customerLocator.getPmfCustomerClientAgentRpcEnc();
if ( customerCA == null )
{
    System.out.println("Step 4: Failed to Create PmfCustomerClientAgent.");
    System.out.println("Exiting Tester.");
    System.exit(0);
}
else
{
    System.out.println("Step 4: Created PmfCustomerClientAgent.");
}
```

5. Pass the `userID` and password that you defined in [Step 2 on page 125](#) to the `SecurityMgrBeanClientAgent` to obtain an authentication token.

```
authentication = securityCA.authenticate(userID, password);
if ( authentication == null )
{
    System.out.println("Step 5: Failed to obtain an authentication token.");
    System.out.println("Exiting Tester.");
    System.exit(0);
}
else
{
    System.out.println("Step 5: Authentication token obtained...  " );
}
```

6. Create a new instance of a customer. Provide it with the customer number and obtain the customer object for that number from the web service. Include error and exception handling code.

```
aCustomer.setCustomerNumber("536477");
System.out.println("Step 6: The PmfCustomer Challenge data for customer 536477 before the
    service is invoked: " + aCustomer.getChallengeData());
aCustomer = (Customer)customerCA.getCommonObject(userID, authentication, aCustomer);
System.out.println("Step 7: PmfCustomer web service contact successful. Obtained a customer.");
if ( aCustomer != null)
{
    System.out.println("Step 8: The PmfCustomer Challenge data for customer 536477 after the
        service is invoked:  " + aCustomer.getChallengeData());
}
else
{
    System.out.println("Step 8: The customer object returned was Null. ");
}
}
catch (Exception e)
{
    System.out.println("Exception happened while running tester.");
    e.printStackTrace();
}
}
```

Running the Sample Application

You can run your sample code to see the results. This section describes how to run the sample code provided with this release.

To run the sample application provided by Chordiant:

1. In the **WebServicesSamples_Chordiant** project you created in [“Requirements for Creating Web Services” on page 96](#), locate the `ws-runChordiantSample.xml` script. Open the script and ensure that configuration parameters are correct.
2. Create a JAR file for the package containing the Java class, `com.chordiant.axis.webservices.testers`. Name it `ws-chordiant.jar` and place it in the `ChordiantEAR/lib-cust` directory.

3. Create a JAR file for all the **WebServicesProxyProject**. Name it `ws-proxy.jar` and place it in the `ChordiantEAR\lib-cust` directory.

Verify that you are not referring to the JAR files in the `ChordiantEAR\lib` directory. This folder contains the Chordiant JARs and you will have mismatches between proxy classes and the actual Chordiant classes if you refer to `\lib` directory in the classpath.

4. Start the Chordiant server.
5. Be sure that you have deployed the `PmfCustomer` web service, as described in [“Deploying Web Services”](#) on page 106.
6. In the **WebServicesSamples_Chordiant** project, locate the `ws-runChordiantSample.xml` script. Right-click and select **Run Ant**.

7. In the **Modify Attributes and Launch** window, select the appropriate target, RunProxyTester. Click **Run**.

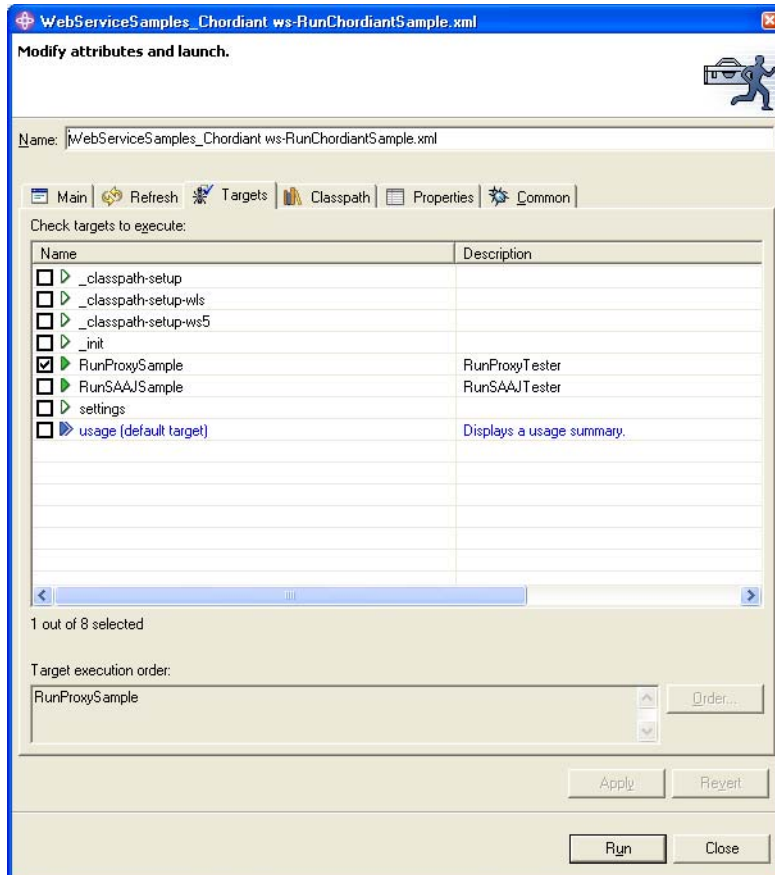


Figure 7-5: Running the Sample Application

The console displays the results of your query to the PmfCustomer web service.

```
RunProxyTester:
[java] This is a proxy based tester
[java] Step 1: Created Security Locator.
[java] Step 2: Created Customer Locator.
[java] Step 3: Created SecurityMgrBeanClientAgent.
[java] Step 4: Created PmfCustomerClientAgent.
[java] Step 5: Authentication token obtained...
[java] Step 6: The PmfCustomer Challenge data for customer 536477 before the service is invoked: null
[java] Step 7: PmfCustomer web service contact successful. Obtained a customer.
[java] Step 8: The PmfCustomer Challenge data for customer 536477 after the service is invoked: Fleming
```

Code 7-2: Results of Running the Proxy-Based Chordiant Sample

Using Non-Chordiant Web Services Within Chordiant

The process of using non-Chordiant web services within your Chordiant system is similar to the process just described in [“Invoking Chordiant Web Services” on page 120](#). However, when you are using non-Chordiant code, you must wrap the call to the non-Chordiant code into a Chordiant service, model this Chordiant service in Rational Rose, and then use the Business Component Generator to create the required files.

In this tutorial, you will use a delayed stock quote web service from a non-Chordiant source and wrap it inside a Chordiant business service to create a sample application that retrieves the 20-minute delayed stock quote for the Dow Jones Industrial Average (symbol = ^DJI), a standard stock index in the United States.

Note: The third-party delayed stock quote web service in this example was available at the time of writing. If it is not available when you go through this exercise, you may use a different stock quote web service. You will go through the same steps, with some minor modifications to the code to suit the specifics of the web service you use.

Creating a Java Proxy Code Project

If you have not already done so, create a project to hold your proxy code. Follow the instructions in [“Creating a Java Proxy Code Project” on page 91](#).

Creating the Proxy Code for the Stock Quote Web Service

These instructions are similar to [“Creating the Proxy Code for the PmfCustomer Web Service” on page 121](#).

To create proxy code for the non-Chordiant stock quote web service:

1. In the Chordiant Tools Platform, open the **ChordiantUtils** project, navigate to the WebServices directory and open the `ws-wsd12java-dynamic.xml` file. This is the Ant-based script used for generating Java proxies from dynamic WSDL files running on a server.
2. Update the script with a new section for the new delayed stock quote WSDL target:
`http://ws.cdyne.com/delayedstockquote/delayedstockquote.asmx?wsdl`
as shown in [Code Sample 7-3](#). If you want to view the WSDL itself, navigate to this URL.

<pre><!-- ===== A StockQuote Service ===== --> <target name="AStockQuoateServiceRpcEnc" depends="_init" description="WSDL to Java for AStockQuoateServiceRpcEnc"> <java fork="true" classname="org.apache.axis.wsd1.WSDL2Java" classpathref="classpath.tools" jvm="\${tools.jvm}" > <arg value="-o"/> <arg value="../WebServicesSamples_NonChordiant/src"/> <arg value="http://ws.cdyne.com/delayedstockquote/ delayedstockquote.asmx?wsdl"/> <arg value="-d"/> <arg value="Session"/> <arg value="-s"/> <arg value="-S"/> <arg value="true"/> </java> </target></pre>	<p>Add a descriptive name for the service.</p> <p>Add a descriptive title for the selection.</p> <p>Calls the web services infrastructure. Do not change this line.</p> <p>Specify the output directory for the proxy code.</p> <p>Specify the location of the WSDL running on a server.</p> <p>Specify the deploy scope. For all services, this will be “Session”.</p> <p>Emit server-side bindings for web service.</p> <p>The two arg values of -S and true give direction to generate a skeleton in the deploy.wsdd file.</p>
--	---

Code 7-3: Sample Addition to the ws-wsd12java-dynamic Script

3. Save and close the `ws-wsd12java-dynamic.xml` file.
4. Right-click this newly-modified `ws-wsd12java-dynamic.xml` and select **Run Ant**.

5. In the **Modify Attributes and Launch** window, select the **Targets** tab if it is not already selected. Select the checkbox next to the stock quote service so you can create proxies for it.

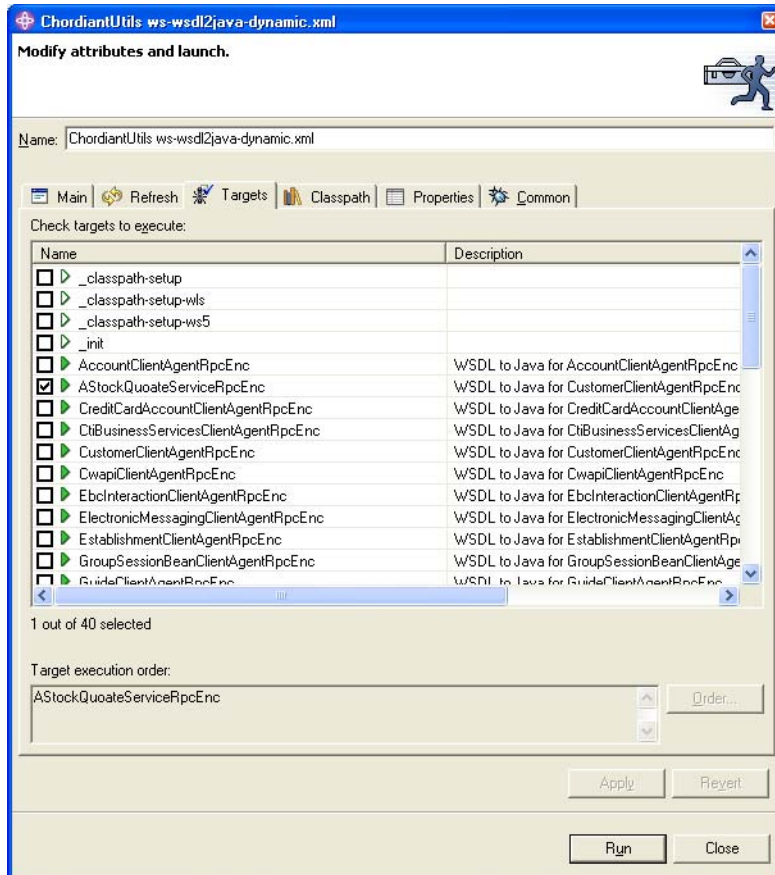


Figure 7-6: Specifying WSDLs in the ws-wsdl2java.xml Script

6. Click **Apply** to save the changes and click **Run** to close the dialog and run the script.
You might see one or more black command windows appear during the process. This is normal. You can watch the **Server Console** to verify that the script ran successfully.
7. The Java proxies are generated and are placed into the directory you specified in [“Creating a Java Proxy Code Project” on page 91](#).

The proxy code in the directory includes one or more packages for the code that mirror the packages for the original code.

Incorporating the Proxy Code into a Chordiant Service

Now that you have the proxy code, incorporate it into a Chordiant service. There are several parts to this process:

- [“Creating the Service” on page 132](#)
- [“Creating the Client Agent” on page 136](#)
- [“Adding to Configuration” on page 138](#)
- [“Testing the Service” on page 139](#)

Creating the Service

Create your service as a Rose model, using the Business Component Generator, then add the required logic to the generated service skeleton. This section describes the relevant points of the service. As you go through these steps, confirm the generated parts of the service and add any required logic as indicated.

1. Declare the package name for the class.

```
package com.chordiant.outbound.webservices.testers;
```

2. Add the import statements required for the class. You must import some of the proxy code from the stock quote web service. Since you are using Chordiant services, you must import some Chordiant classes.

```
import java.math.BigDecimal;
import com.cdyne.ws.DelayedStockQuoteLocator;
import com.cdyne.ws.DelayedStockQuoteSoap;
import com.chordiant.core.log.LogHelper;
import com.chordiant.service.BaseServiceControlResponse;
import com.chordiant.service.PayloadData;
import com.chordiant.service.Service;
import com.chordiant.service.ServiceBaseClass;
import com.chordiant.service.ServiceControlRequest;
import com.chordiant.service.ServiceControlResponse;
```

3. Declare the class, which should extend the service base class.

```
public class OutboundWebService extends ServiceBaseClass implements Service {
```

4. Define the constants for the class, including the method and parameter names used for the stock quote.

```
public final static String CLASS_NAME = "OutboundWebService";
public final static String PACKAGE_NAME = "com.chordiant.outbound.webservices.testers";
public final static String FUNCTION_GET_QUICK_STOCK_QUOTE = "getQuickStockQuote";
public final static String STOCK_SYMBOL = "symbol";
public final static String STOCK_PRICE = "stock_price";
```


5. Include the `processRequest` method. Most of this method is standard for Chordiant services. If you modeled your service, it should be generated for you automatically by the Business Component Generator. Pay special attention to the section dealing with the stock quote, shown in bold.

```
public Object processRequest(String username, String authentication, String serviceName,
    String functionName, Object payload)
    throws Throwable
{
    final String METHOD_NAME = "processRequest";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    Object retVal = null;
    PayloadData requestPayload = null;
    PayloadData responsePayload = null;
    String stockName = null;

    if ((functionName != null) && (functionName.length() > 0))
    {
        try
        {
            // Do a simple "if statement" dispatcher
            if (functionName.compareToIgnoreCase(FUNCTION_GET_QUICK_STOCK_QUOTE) == 0)
            {
                // Cast the payload as needed.
                requestPayload = (PayloadData) payload;
                // Pull any required parameters out of the payload
                // as needed for the typed local function.
                if (requestPayload == null)
                {
                    LogHelper.debug(PACKAGE_NAME, CLASS_NAME, METHOD_NAME, "No request payload found");
                } else
                {
                    LogHelper.debug(PACKAGE_NAME, CLASS_NAME, METHOD_NAME,
                        "Payload found for outbound web service execution, about to unpack");
                    stockName = (String) requestPayload.getDataWithName(STOCK_SYMBOL);
                    if (stockName == null)
                    {
                        LogHelper.debug(PACKAGE_NAME, CLASS_NAME, METHOD_NAME,
                            "The application did not supply a Stock symbol");
                    }
                    // Call the specific local function
                    String results = getQuickStockQuote(stockName);
                    if (results == null)
                    {
                        LogHelper.debug(PACKAGE_NAME, CLASS_NAME, METHOD_NAME, "No results were returned");
                        retVal = null;
                    } else
                    {
                        LogHelper.debug(PACKAGE_NAME, CLASS_NAME, METHOD_NAME,
                            "Objects being prepared to be returned:" + results);
                        responsePayload = new PayloadData();
                        // Fill the return payload with the appropriate parameters.
                        responsePayload.putDataWithName(STOCK_SYMBOL, stockName);
                        responsePayload.putDataWithName(STOCK_PRICE, results);
                        // Assign the response payload to the return value of this method.
                        retVal = responsePayload;
                    }
                }
            }
        }
    }
}
```

```

        } else
        {
            LogHelper.error(PACKAGE_NAME, CLASS_NAME, METHOD_NAME,
                "Unknown function name [" + functionName + "]");
        }
    } catch (Throwable e)
    {
        LogHelper.error(PACKAGE_NAME, CLASS_NAME, METHOD_NAME, "Exception occured", e);
        throw e;
    }
} else
{
    LogHelper.error(PACKAGE_NAME, CLASS_NAME, METHOD_NAME, "Null or zero length function name");
}
LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
return retVal;
}

```

6. Declare the **getQuickStockQuote** method, which takes one parameter, **stockSymbol**. If you created your service through a model, the framework for this method already exists. You must fill in the logic.

```

public String getQuickStockQuote(String stockSymbol)
    throws Throwable
{
    final String METHOD_NAME = "getQuickStockQuote";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    String retVal = null;
    LogHelper.debug(PACKAGE_NAME, CLASS_NAME, METHOD_NAME,
        "Executing a get quick quote for stock symbol " + stockSymbol);
}

```

7. In the try block, instantiate the **Locator**, just as you did in the Chordiant example, described in [Step 3 on page 125](#).

```

try
{
    DelayedStockQuoteLocator stockQuoteService = new DelayedStockQuoteLocator();
    System.out.println("Step 1: Created StockQuote Locator.");
}

```

8. Call the **getDelayedStockQuote** method from the web service. Cast the response as a **DelayedStockQuoteSoap**. In this case, **DelayedStockQuoteSoap** is acting as a client agent.

```

DelayedStockQuoteSoap stockQuoteWebService =
    (DelayedStockQuoteSoap) stockQuoteService.getDelayedStockQuoteSoap();
System.out.println("Step 2: Created the service object.");

```

9. Define the `stockPrice` as a `BigDecimal`. Receive and return the `stockPrice` for the `stockSymbol` you passed. Handle any exceptions that might occur.

```

        BigDecimal stockPrice = stockQuoteWebService.getQuickQuote(stockSymbol, "");
        System.out.println("Step 3: StockPrice obtained from web service is " + stockPrice );
        retVal = stockPrice.toString();
    }
    catch (Exception e)
    {
        System.out.println("Exception happened while running Outbound Web Service:");
        e.printStackTrace();
    }
    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retVal;
}

```

10. Include the standard service control methods that enable the service to be controlled through the Administration Utility. For details on service control methods, refer to the “Administration” chapter in the *Chordiant 5 Foundation Server Developer’s Guide*.

```

protected ServiceControlResponse reinitialize(ServiceControlRequest theRequest) throws Throwable
{
    final String METHOD_NAME = "reinitialize";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    ServiceControlResponse retVal = null;
    retVal = new BaseServiceControlResponse();
    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retVal;
}

protected ServiceControlResponse setup(ServiceControlRequest theRequest) throws Throwable
{
    final String METHOD_NAME = "setup";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    ServiceControlResponse retVal = null;
    retVal = new BaseServiceControlResponse();
    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retVal;
}

protected ServiceControlResponse shutdown(ServiceControlRequest theRequest) throws Throwable
{
    final String METHOD_NAME = "shutdown";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    ServiceControlResponse retVal = null;
    // Do some shutdown stuff here
    retVal = new BaseServiceControlResponse();
    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retVal;
}

```

```
protected ServiceControlResponse status(ServiceControlRequest theRequest) throws Throwable
{
    final String METHOD_NAME = "status";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    ServiceControlResponse retVal = null;
    // Do some status stuff here
    retVal = new BaseServiceControlResponse();
    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retVal;
}
}
```

Creating the Client Agent

You will use a client agent to contact your new service. Create a client agent for your service, or verify the one that was created automatically for you if you used the Business Component Generator. For details on creating client agents, refer to the *Chordiant 5 Foundation Server Developer's Guide*.

1. Declare the package and include relevant import statements for a Chordiant client agent.

```
package com.chordiant.outbound.webservices.testers;

import com.chordiant.core.configuration.ConfigurationHelper;
import com.chordiant.core.log.LogHelper;
import com.chordiant.service.PayloadData;
import com.chordiant.service.clientagent.ClientAgent;
import com.chordiant.service.clientagent.ClientAgentBaseClass;
```

2. Declare the class by extending from the Chordiant ClientAgentBaseClass.

```
public class OutboundServiceClientAgent extends ClientAgentBaseClass implements ClientAgent
{
```

3. Declare the constants for the client agent.

```
public final static String CLASS_NAME = "OutboundServiceClientAgent";
public final static String PACKAGE_NAME = "com.chordiant.outbound.webservices.testers";
public final static java.lang.String SERVICE_EXT = "_service";
```

4. Define the entry for the getQuickStockQuote method, including the relevant parameters.

```
public String getQuickStockQuote(String username, String authentication, String company)
{
    final String METHOD_NAME = "getQuickStockQuote";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    PayloadData requestPayload = null;
    PayloadData responsePayload = null;
    String retVal = null;
```

5. Retrieve the name of the service you are accessing from configuration. Refer to [“Adding to Configuration” on page 138](#) for details on the configuration file.

```
try
{
    String serviceName = ConfigurationHelper.getConfigurationValue("services", "OutboundWebService.name");
    LogHelper.debug(PACKAGE_NAME, CLASS_NAME, METHOD_NAME,
        "The name of the OutBound Web Sample service is " + serviceName);
}
```

6. Pass the payload, which includes the stock symbol information.

```
requestPayload = new PayloadData();

if (company != null)
{
    requestPayload.putDataWithName(OutboundWebService.STOCK_SYMBOL, company);
}

responsePayload = (PayloadData) processRequest(username, authentication, serviceName,
    OutboundWebService.FUNCTION_GET_QUICK_STOCK_QUOTE, requestPayload);
```

7. Receive the return value from the service and handle any exceptions and logging.

```
// Pull out any needed return values from the payload as appropriate for the return value
// of this method.
if (responsePayload == null)
{
    LogHelper.debug(PACKAGE_NAME, CLASS_NAME, METHOD_NAME, "No response payload returned");
}
else
{
    {
        retVal = (String) (responsePayload.getDataWithName(OutboundWebService.STOCK_PRICE));
    }
}
catch (Throwable e)
{
    LogHelper.error(PACKAGE_NAME, CLASS_NAME, METHOD_NAME, "Exception occurred", e);
}
LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
return retVal;
}
}
```

Adding to Configuration

As with any Chordiant service, you must inform the system about this service through the configuration file. Create a configuration file for your service, or verify the one that was created automatically for you if you used the Business Component Generator.

[Code Sample 7-4](#) shows the `OutboundWebServiceSample.xml`, the configuration file for our service that contains the call to the non-Chordiant web service. The full, annotated version of this code is found in the sample code you installed in [“Installing the Sample Applications for Web Services” on page 118](#). For details on adding services through configuration files, refer to the *Chordiant 5 Foundation Server Developer’s Guide*.

```
<Root>
  <Section> services
    <Tag>OutboundWebService.name
      <Value>OutboundWebService</Value>
    </Tag>
  </Section>

  <Section>OutboundWebService
    <Tag> classname
      <Value> com.chordiant.outbound.webservices.testers.OutboundWebService</Value>
    </Tag>
    <Tag>ConnectionName
      <Value>EJBCMTRequired</Value>
    </Tag>
  </Section>

  <Section> clientagents
    <Tag>OutboundWebService.agent
      <Value>OutboundServiceClientAgent</Value>
    </Tag>
  </Section>

  <Section>OutboundServiceClientAgent
    <Tag>classname
      <Value>com.chordiant.outbound.webservices.testers.OutboundServiceClientAgent</Value>
    </Tag>
    <Tag>stubtype
      <Value>EJBStub</Value>
    </Tag>
  </Section>
</Root>
```

Code 7-4: OutboundWebServiceSample.xml Configuration File

Testing the Service

Write a tester application to try out your new service. In this tester, we are checking the stock quote for the Dow Jones Industrial Average (ticker symbol: ^DJI).

1. Declare the package names and include the relevant import statements. Since this is a thick client tester, you must import the appropriate Chordiant classes.

```
package com.chordiant.outbound.webservices.testers;

import com.chordiant.core.FatClientStaticHelper;
import com.chordiant.service.clientagent.ClientAgentHelper;
import com.chordiant.userprofile.ejb.security.client.SecurityMgrBeanClientAgent;
```

2. Declare the name of this tester class.

```
public class ChordiantOutboundWebServiceTester {
```

3. Set up the main method and create placeholders for the required parameters. Notice we have selected the stock quote ticker symbol here.

```
    public static void main(String[] args) throws Exception
    {
        String stockSymbol = "^DJI";
        String stockPrice = null;
        String authentication = null;
        String userName = "hmonroe";
        String passWord = "hmonroe";
        boolean notDone = true;
        System.out.println("Starting the application. ");
```

4. In a try block, get a **SecurityMgrBean** client agent and then call it to get an authentication token.

```
try
{
    FatClientStaticHelper.serviceControl(FatClientStaticHelper.SERVICE_CONTROL_COMMAND_SETUP);
    SecurityMgrBeanClientAgent smbca =
        (SecurityMgrBeanClientAgent)ClientAgentHelper.getClientAgent("SecurityMgrBeanClientAgent");

    if ( smbca != null )
    {
        System.out.println("Obtained Security Manager Bean client agent using Client agent Helper.");
    }
    else
    {
        System.out.println("Failed to Obtain Security Manager Bean client agent using
            Client Agent Helper.");
        System.out.println("Exiting Tester.");
        System.exit(0);
    }
}
```



```
authentication = smbca.authenticate(userName, passWord);
if ( authentication != null )
{
    System.out.println("Obtained authentication token.");
}

else
{
    System.out.println("Failed to obtain authentication token.");
    System.out.println("Exiting Tester.");
    System.exit(0);
}
```

5. Get a client agent for the new service with the stock quote web service call, named the Outbound Service.

```
OutboundServiceClientAgent outboundCA = (OutboundServiceClientAgent)ClientAgentHelper.
getClientAgent("OutboundServiceClientAgent");

if ( outboundCA != null )
{
    System.out.println("Obtained Outbound Service Client Agent.");
}

else
{
    System.out.println("Failed to obtain Outbound Service Client Agent.");
    System.out.println("Exiting Tester.");
    System.exit(0);
}
```

6. Call the Chordiant Outbound Service you created, by using its client agent. Include all of the input parameters — both those required by Chordiant and those required by the web service. Return the stock price as a printout.

```
stockPrice = outboundCA.getQuickStockQuote(userName, authentication, stockSymbol);
if ( stockPrice != null )
{
    System.out.println("The stock price for " + stockSymbol + " is " + stockPrice + "\n");
}

else
{
    System.out.println("Failed to Obtain stock price.");
    System.out.println("Exiting Tester.");
    System.exit(0);
}
```

7. Perform standard shutdown and exception-handling tasks.

```
FatClientStaticHelper.serviceControl(FatClientStaticHelper.SERVICE_CONTROL_COMMAND_SHUTDOWN);
}
catch (Exception e)
{
    System.out.println("Exception happened while running tester:");
    e.printStackTrace();
}
System.out.println("Exiting the application");
}
}
```

Running the Sample Application

You can run your sample code to see the results. This section describes how to run the sample code provided with this release.

To run the sample application provided by Chordiant:

1. Start the Chordiant server in your development environment.
2. Copy the `ws-nonchordiant-src.jar` from the **WebServicesSamples_Non-Chordiant** project you created in [“Requirements for Creating Web Services” on page 96](#) to the `ChordiantEAR\lib` and the `JXRuntime\lib` directories.
3. Add this `ws-nonchordiant-src.jar` file to the classpath, as appropriate for your application server.
4. In the **WebServicesSamples_Non-Chordiant** project, open the `scripts` directory. Open the appropriate `ws-run-nonchordiant-{application_server}.properties` file in a text editor.
5. Configure the `chordiant.home` and `appserver.home` properties as appropriate for your installation. For WebLogic only, configure the `appserver.jvm` property.
Review the other properties and make sure they are correct for your application server.
6. Right-click the `ws-run-nonchordiant-{application_server}.xml` file and select **Run Ant**.

7. In the **Modify Attributes and Launch** window, select the **WebServiceNonChordiant** target. Click **Run**.

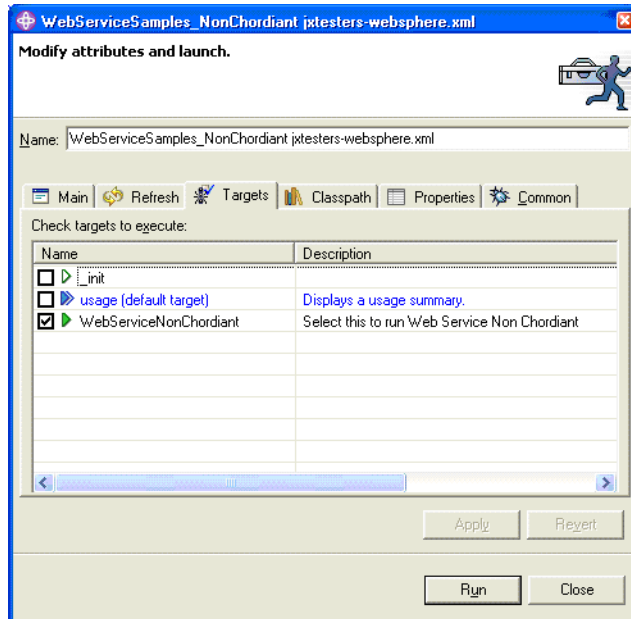


Figure 7-7: Running the Sample Non-Chordiant Web Services Sample Application (WebSphere Environment)

The console displays the results of your query to the stock quote web service.

```
Buildfile: Y:\foundationp\JX_Toolkit\ChordiantWebServices\WebServicesSamples_NonChordiant\scripts\
ws-run-nonchordiant-websphere.xml

_init:

WebServiceNonChordiant:
[java] Starting the application.
[java] [STARTUP] GLUE Professional 4.1.2 (c) 2001-2003 The Mind Electric
[java] Obtained Security Manager Bean client agent using Client Agent Helper.
[java] Obtained authentication token.
[java] Obtained Outbound Service Client Agent.
[java] The stock price for ^DJI is 10400.07
[java]
[java] Exiting the application
BUILD SUCCESSFUL
Total time: 4 seconds
```

INVOKING WEB SERVICES DYNAMICALLY THROUGH SAAJ

As an alternative to creating Java proxy code, as described in “[Invoking Web Services Statically](#)”, you can use a different technique to build code dynamically using SAAJ (SOAP with Attachments API for Java). SAAJ invokes the WSDL on a URL and dynamically builds XML-based code within your Chordiant service. As the name suggests, the functionality is achieved through sending SOAP messages through the internet. You will notice that much of the code is involved with building, sending, and receiving these messages. You will also notice that you define the methods you will call and then you use an `invoke` method to call the other methods.

To invoke a web service dynamically using SAAJ:

Setting Up the Class

1. Include the standard package declaration. Add import statements that enable the web service infrastructure.

```
package com.chordiant.axis.webservices.testster;

import java.net.URL;
import java.util.Iterator;

import javax.xml.soap.MessageFactory;
import javax.xml.soap.Name;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPBodyElement;
import javax.xml.soap.SOAPConnection;
import javax.xml.soap.SOAPConnectionFactory;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPFactory;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPMessage;

import com.chordiant.customer.businessClasses.Customer;
```

2. Declare the class.

```
public class SAAJTester {
```

Defining the Authenticate Method

3. Define the authenticate method called `soapAuthenticate`.

```
    public static String soapAuthenticate(String userID, String password)
    {
```

Setting Up for Future Objects

4. Define a `bodyElement` object to hold the information contained in the SOAP message.

```
        SOAPElement bodyElement;
```

5. Create a null object for the authentication token, which you will receive from the security client agent.

```
String authent = null;
```

Creating the Connections and Messages

In the try block, add the code that will create the SOAP messages dynamically.

6. A `SoapConnection` object is a point-to-point connection, connecting you to the URL you specify. Get a new instance of the `SoapConnectionFactory`, which you need to create a `SoapConnection`. Then create a `SoapConnection` using the factory you just instantiated.

Get a new instance of the `SoapFactory`, which you will use to create other parts of the SOAP message.

```
try
{
    SOAPConnectionFactory soapConnectionFactory = SOAPConnectionFactory.newInstance();
    SOAPConnection connection = soapConnectionFactory.createConnection();
    SOAPFactory soapFactory = SOAPFactory.newInstance();
```

7. Get an instance of the `MessageFactory` and then use it to create a SOAP message.

```
MessageFactory factory = MessageFactory.newInstance();
SOAPMessage message = factory.createMessage();
```

8. Use the `getSOAP{MessagePart}` methods to get the empty header and body for this new message you just created.

```
SOAPHeader header = message.getSOAPPart().getEnvelope().getHeader();
SOAPBody body = message.getSOAPPart().getEnvelope().getBody();
```

9. This example does not use the message header, so you can delete it. Since it uses the `Node` interface, you must use the `detachNode` method to detach it.

```
header.detachNode();
```

Note: Although this example does not use the message header, you might want to use it in your own implementations. We show creating and removing the header in these steps so you are familiar with this procedure.

Creating Name Objects for the SOAP Message

10. Create a **Name** object that will be a part of the message **bodyElement** you defined in [Step 4 on page 144](#). The **Name** object must be fully-qualified, including a local name, a prefix for the namespace being used, and a URI for the namespace. Fully qualifying the object is important for distinguishing objects with the same local name.

Here, the name is for the **authenticate** method. The URI specified is the reverse of the package name containing the security client agent.

Once you have created the **Name** object, add it to the **bodyElement**.

```
Name bodyName = soapFactory.createName("authenticate", "m",  
    "http://client.security.ejb.userprofile.chordiant.com");  
bodyElement = body.addBodyElement(bodyName);
```

11. Create additional **Name** objects, one for the username and one for the password.

```
Name arg1 = soapFactory.createName("username");  
SOAPElement symbol1 = bodyElement.addChildElement(arg1);  
symbol1.addTextNode(userID);  
  
Name arg2 = soapFactory.createName("password");  
SOAPElement symbol2 = bodyElement.addChildElement(arg2);  
symbol2.addTextNode(password);
```

Setting the Endpoint and Make the Connection

12. Specify the endpoint object to contain the URL of your target web service.

```
URL endpoint = new URL("http://localhost/WebServices/services/SecurityMgrBeanClientAgentRpcEnc");
```

13. The **connection.call** method takes the message contents and the endpoint you just specified. After you complete your call, close the connection to conserve system resources.

```
SOAPMessage response = connection.call(message, endpoint);  
connection.close();
```

Getting the Contents of the Message

14. Look into the `soapBody` element to get the child elements of the `Names` you created. Use the `java.util.Iterator` object and the `getChildElements` method to walk through all of the child elements contained within a `Name` object. The method `Iterator.next` returns a Java object, so you must cast the object it returns to a `SOAPBodyElement` object before assigning it to the variable `bodyElement`.

In Chordiant services, there is only one element with the name `bodyName`, so use an `if` statement. If you are using a non-Chordiant web service that has more than one element, use a `while` loop with the method `Iterator.hasNext` to make sure that you get all of the elements.

```
SOAPBody soapBody = response.getSOAPPart().getEnvelope().getBody();
Iterator iterator = soapBody.getChildElements();
// for this call we expect just one child element
if (iterator.hasNext()) {
    bodyElement = (SOAPBodyElement)iterator.next();
    // for this call we expect just one inner body element
    Iterator inner = bodyElement.getChildElements();
    if (inner.hasNext())
    {
        SOAPBodyElement innerbodyElement = (SOAPBodyElement)inner.next();
        authent = innerbodyElement.getValue();
    }
    else
    {
        System.out.println("Authentication token has not been returned");
    }
}
else
{
    System.out.println("Returned SOAP Message does not have Child Elements");
}
}
```

15. Catch any exceptions and receive the return value for the authentication token. You will need the authentication token in the next call to the `PmfCustomer` service.

```
catch (Exception ex)
{
    ex.printStackTrace();
}
return authent;
}
```

Defining the Invoke Method

16. Define an `invoke` method to run the other methods in the class. In this example, the parameters are populated with appropriate user name, password, and customer number information to ensure success of the web service.

```
public static void invoke()
{
    Customer aCustomer = null;
    String authent;
    String userID = "hmonroe";
    String password = "hmonroe";
    System.out.println("This is a SAAJ tester");
    authent = soapAuthenticate(userID, password);
    System.out.println("The authentication token is " + authent);
}
```

Calling the Invoke Method

17. In the main method of your class, call the `invoke` method to receive your results.

```
public static void main(String[] args) throws Exception
{
    invoke();
}
```

Running the Sample Application

You can run your sample code to see the results. This section describes how to run the sample code provided with this release.

To run the sample application provided by Chordiant:

1. Start the Chordiant server.
2. In the **WebServicesSamples_Chordiant** project you created in [“Requirements for Creating Web Services” on page 96](#), locate the `ws-RunChordiantSample.xml` script.
3. Right-click and select **Run Ant**.

4. In the **Modify Attributes and Launch** window, select the appropriate target, RunSAAJSample. Click **Run**.

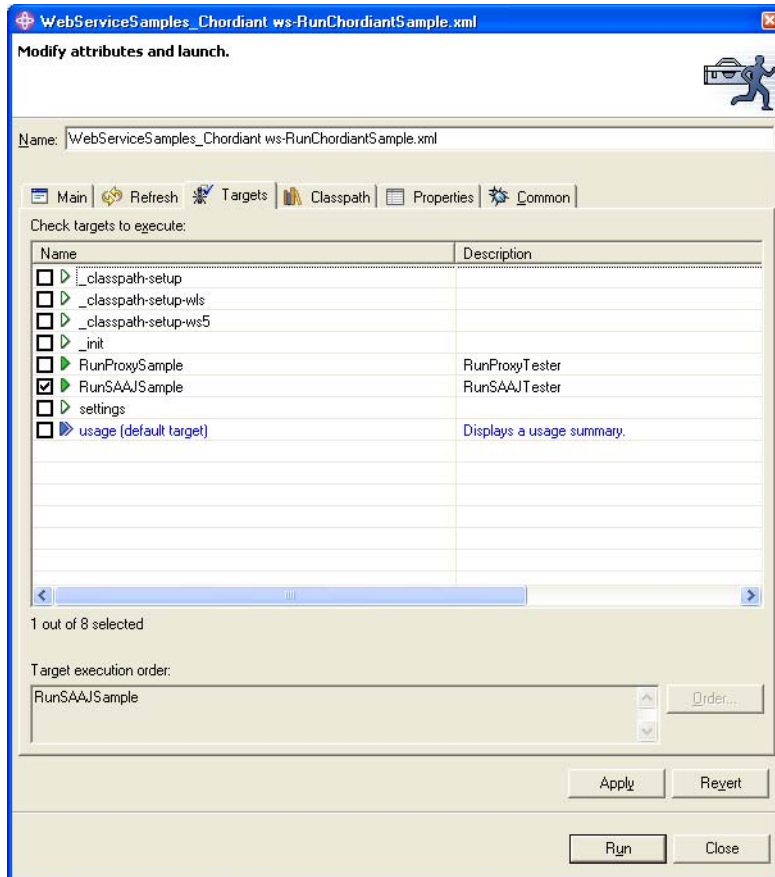


Figure 7-8: Running the Sample SAAJ Application

The console displays the results of your query to the SecurityMgrBean web service, showing the authentication token you received.

```
This is a SAAJ tester.
The authentication token is 3645)4>{3645)4>{m{k
```

Code 7-5: Results of Running the SAAJ-Based Chordiant Sample

Creating Task Descriptors

The Chordiant 5 Business Process Designer uses task descriptors to describe the inputs and outputs of business processes. These task descriptors are based on Chordiant business services and CAFE tasks. They are placed on a node of a process flow and perform the work of the service when that node is reached.

Chordiant provides many task descriptors which are ready to use with your tasks and process flows. A list of provided task descriptors based on Chordiant business services is provided in [Chapter 16, “Chordiant 5 Application Components”](#). A list of CAFE task and legacy task descriptors is provided in the *Chordiant 5 Tools Platform Business Process Designer Developer’s Guide*.

You can also create your own task descriptors, based on your own customizations. This chapter details how to create your own task descriptors based on business services.

Business Process Designer task descriptors are based on the Web Services Description Language (WSDL). They are not used as web services. Chordiant is leveraging this standardized language for a different use — to describe functions, inputs, and outputs that can be included in a business process. Do not confuse these WSDL files with the WSDL files used for Chordiant Web Services, as described in [Chapter 6](#) and [Chapter 7](#).

Task descriptors are subject to the same requirements as web services, in that all input parameters must be business objects with default constructors. Refer to [“Requirements for Creating Web Services”](#) on page 96.

For more information, refer to the following documentation:

- *Chordiant 5 Tools Platform Business Process Designer Developer’s Guide* — Explains how to use task descriptors within the Business Process Designer and a list of all of the task descriptors provided by Chordiant.
- *Chordiant 5 Foundation Server Business Process Server Developer’s Guide* — Provides details on the structure and functionality of task descriptors.

TASK DESCRIPTOR LOCATION

In an installed Chordiant system, task descriptors provided by Chordiant are located in the `{WORKSPACE}/ChordiantUtils/TaskDescriptors` directory. This directory contains two sub-directories that hold the task descriptor WSDL files:

- `BusinessServices` — Holds the task descriptors based on Chordiant services or basic Java services, which are not necessarily invoked by a client agent proxy.
- `CAFE` — Holds task descriptors that are stateful or based on CAFE legacy tasks.

You can use these provided task descriptors and you can also include your own, whether they are modifications to existing Chordiant services or based on services that you create on your own.

By default, the Business Process Designer uses the `BusinessServices` and `CAFE` directories as the primary source of task descriptors. If you do not use these directories, you must update the Business Process Designer properties with the alternate locations of your task descriptors.

The Ant script for creating your own task descriptors is located at the root of the `TaskDescriptors` directory and is described in [“Generating Task Descriptors” on page 153](#).

CREATING AND UPDATING TASK DESCRIPTORS

When you create or customize a business service or any Java class, you can create a task descriptor for it.

Task Descriptor Types

There are three different types of task descriptors, based on the type of code they represent in the Business Process Designer.

- **Client** — Represents standard Chordiant business services that are accessed by client agents.
- **Java** — Represents any Java class that is not a Chordiant client-agent-based business service.
- **Stateful** — Represents tasks derived from the `StatefulTaskAdapter`, including stateful `CAFE` tasks that derive from the `StatefulTaskBaseClass`. You will likely not create this type of task descriptor.

For details on the types of task descriptors, refer to the *Chordiant 5 Foundation Server Business Process Server Developer's Guide*.

Working with Services

You can create task descriptors for services which you create from a Rational Rose model and the Business Component Generator or for services that you create by hand.

Note: If you change an existing Chordiant business service or create your own services, you must create or update the task descriptors.

Model-Based Services

To adhere to Chordiant’s customization model, create your business services starting with a model, then use the Business Component Generator to create your code. Follow the instructions in:

- [Chapter 3, “Using the Business Component Generator”](#)
- [Chapter 4, “Creating or Modifying Business Services”](#)
- [Chapter 5, “Tutorial: Creating and Extending a Business Service”](#)

Make sure that you have created the customizations you require within your model, then follow the procedures to run the Business Component Generator. Also make sure that your `{Model}descriptor.xml` file specifies that you are generating service components.

Note: Be sure to use a new name for your customized service. If you create customizations with the same name as Chordiant services, they might be overwritten if you install a new version of Chordiant. This includes task descriptor files as well as basic Chordiant services.

Since the service file is generated as a skeleton, you must add the actual functionality to the methods you have specified. You might also need to update the generated configuration file, as described on [page 44](#).

You create the web services components after you have created the other business components, because web services are based on the client agent code. Follow the steps described in [“Generating Task Descriptors”](#) on [page 153](#).

Services Without Models

Some services, like system services, do not have models associated with them. You can still generate task descriptors for them. Follow the instructions in the next section, [“Generating Task Descriptors”](#).

Generating Task Descriptors

The Java class that generates Chordiant Task Descriptors is `com.chordiant.tools.GenerateTaskDescriptor`. Generating task descriptor is a two-step process:

- First, the tool uses the web service infrastructure’s **java2wsdl** tool to create a WSDL file with a SOAP binding. As described on [page 151](#), Chordiant uses WSDL as the description language for Chordiant Task Descriptor.
- Then, the tool uses the **GenerateTaskDescriptor** tool to add Chordiant-specific binding and tags to the WSDL file generated in the first step.

This two-step process is automated in the provided Ant script, `generate-tasks.xml`, located in the `{WORKSPACE}/ChordiantUtils/TaskDescriptors` directory. This Ant script includes targets for generating each of the types of task descriptor described in “Task Descriptor Types” on page 152. You will likely only use the Client and Java types. This script creates the initial set of Chordiant-provided task descriptors. You can also add your own customized services as targets in the script to generate task descriptors for them.

Preparing to Generate

To create task descriptors from your services:

1. In the `{WORKSPACE}/ChordiantUtils/TaskDescriptors` directory, locate the `generate-tasks.xml` Ant script. Open it with a text editor.
2. Update the properties at the end of the script, in the `_init` target, if needed. These properties are:
 - **jar.home**: The location of the JAR files containing your binary code.
Default value = `../../ChordiantEAR/lib-debug`
 - **src.home**: The location of the JAR files containing your source code.
Default value = `../../ChordiantEAR/lib-src`

You must put your JAR files in these directories or change these properties to match the location of your JAR files.

You can also use the `-D` option to define the value of these two properties on the command line when you invoke the Ant script, as shown in [Code Sample 8-1](#).

```
ant -f generate-tasks.xml -Djar.home={filepath} -Dsrc.home={filepath}
```

Code 8-1: Defining Parameters Through the Command Line

3. If you have not already done so, copy your binary and source JAR files to the directories specified in the `_init` target, as described in [Step 2](#).

When your source code is available in the appropriate directory, the `GenerateTaskDescriptor` tool incorporates your class and API-based Javadoc into the task descriptor so it is available in the Business Process Designer.

4. Review the action items for the Ant target `_tdgen_client` in the upper portion of the `generate-tasks.xml` Ant script, as shown in [Code Sample 8-2](#) and [Code Sample 8-3](#). You do not have to change anything in this section, but it is helpful to understand what the script and tool are doing.

In the first part of the script, an intermediate WSDL file is created from your Java code. Three parameters are included in this portion. You can see their values in [Code Sample 8-2](#).

- **-o:** The name and location of the intermediate WSDL file.
- **-w:** The type of output from the tool. This value is always `interface`.
- **-x:** lists methods that you don't want to include in task descriptors. This value is taken from a list specified in the `_init` target.

Note that this list of specified excluded APIs is for targets excluded from the out-of-the-box task descriptors. If you change this excluded API list, when you next generate task descriptors, all of the Chordiant task descriptors will be regenerated again, with the altered list. The script is set up to create all of the task descriptors each time it is run. There are several ways to work around this solution, including having separate targets for your own task descriptors using a different set of excluded APIs, as described in the Tip on [page 157](#).

```
<target name="_tdgen_client" depends="_init" description="Generating task descriptor for ${td.name}">
  <java fork="true" classname="org.apache.axis.wsdl.Java2WSDL" classpathref="classpath.tdgen" >
    <arg value="-o"/>
    <arg value="tmp/${td.name}.wsdl"/>
    <arg value="-w"/>
    <arg value="interface"/>
    <arg value="-x"/>
    <arg value="{excluded_api}"/>
    <arg value="{td.class}"/>
  </java>
```

Code 8-2: `_tdgen_client` Target of the `generate-tasks.xml` Ant Script: Part 1

In the second part of the script, shown in [Code Sample 8-3](#), the task descriptor generation tool adds Chordiant-specific binding and tags to the intermediate WSDL file, enabling it to be used as a Business Process Designer task descriptor. Two parameters are set:

- **-type**: In this `_tdgen_client` target, the value is always `client`. This corresponds to tasks based on Chordiant services that use client agents, a form of “stateless task”.

If you want to create a task descriptor from a basic Java class that doesn’t use client agents, call the `_tdgen_java` target, which specifies the type as `java`. These types of task descriptors are also called “stateless”.

If you want to create a task descriptor for a stateful task, use the `_tdgen_stateful` target. As mentioned earlier, you will likely not use this target for creating CAFE task descriptors. This target is mainly used by Chordiant to create the initial set of stateful task descriptors.

- **-wsdl**: This is the name of the intermediate WSDL file created in the first part of the script. This parameter ensures that the two parts of the script work together.

```
<java fork="true" classname="com.chordiant.tools.GenerateTaskDescriptor" classpathref="classpath.tdgen"
  output="BusinessServices/${td.name}.wsdl">
  <arg value="-type"/>
  <arg value="client"/>

  <arg value="-wsdl"/>
  <arg value="tmp/${td.name}.wsdl"/>
</java>
</target>
```

Code 8-3: `_tdgen_client` Target of the `generate-tasks.xml` Ant Script: Part 2

5. The `tdgen_client` target includes `antcall` actions for all of the services available to be used as task descriptors. When you run the `tdgen_client` target, the generation tool creates task descriptors for each of the `antcall` actions it contains.

Update the script with your service information.

- If you have modified an existing service, locate the section for that service and update it, if necessary.
- If you have created a new service, create a new target by copying and pasting an existing `antcall` actions and filling in the appropriate information.
- If you are working with an ordinary Java class instead of a Chordiant business service class, add your `antcall` action to the `_tdgen_java` target instead of the `_tdgen_client` target.

You must specify or verify the two parameters:

- **td.class:** The fully-qualified class name of the service
- **td.name:** The desired name of the task descriptor. This is usually the same as the class name, without the package name. This name will identify your task descriptor within the Business Process Designer.

```
<target name="tdgen_client" depends="_init" description="Generating client agent task descriptors">
  <antcall target="_tdgen_client">
    <param name="td.class" value="com.chordiant.bd.clientAgents.AccountClientAgent"/>
    <param name="td.name" value="AccountClientAgent"/>
  </antcall>
  <antcall target="_tdgen_client">
    <param name="td.class" value="com.chordiant.bd.clientAgents.CustomerClientAgent"/>
    <param name="td.name" value="CustomerClientAgent"/>
  </antcall>
  <antcall target="_tdgen_client">
    <param name="td.class" value="com.chordiant.bd.clientAgents.DeliveryClientAgent"/>
    <param name="td.name" value="DeliveryClientAgent"/>
  </antcall>
</target>
```

Code 8-4: Sample Targets for Client-Agent-Based Services

Tip: To create your own task descriptors without regenerating the Chordiant-provided task descriptors, copy the `_tdgen_java` target, modify it slightly, and add your `antcall` actions under that target.

6. Save and close the script.

Generating the Task Descriptors

This procedure requires JDK 1.4, so the steps are different for WebLogic and WSAD.

For WebLogic

To generate the task descriptors you specified in [“Preparing to Generate” on page 154](#):

1. Right-click the `generate-tasks.xml` Ant script and select **Run Ant**.
2. In the **Modify Attributes and Launch** window, select the `tdgen_client`, `tdgen_java`, or your own custom target to create all of the task descriptors. Click **Run**.

Tip: If you prefer to run this script from the command line, at the command line, type:

```
ant -f generate-tasks.xml {target_name}
```

3. Continue with [“Task Descriptors Created” on page 161](#) for details on the output of this script.

For WSAD

For WSAD, you must specify that you are using JDK 1.4.

To set up for generation:

1. Be sure that you have Ant installed.
2. Set `JAVA_HOME` to the JDK 1.4 runtime. Select **Start | Control Panel | System | Advanced | Environment Variables**. Define the `JAVA_HOME` environment variable to point to a JDK 1.4 runtime.

To generate the task descriptors you specified in [“Preparing to Generate” on page 154](#):

1. Be sure that you have completed the steps to specify JDK 1.4, described in the previous section.
2. Once you have defined `JAVA_HOME`, start WSAD.
3. In WSAD, select **Run | External Tools | External Tools**.

4. In the **External Tools** dialog, select **Program**, then click **New**.

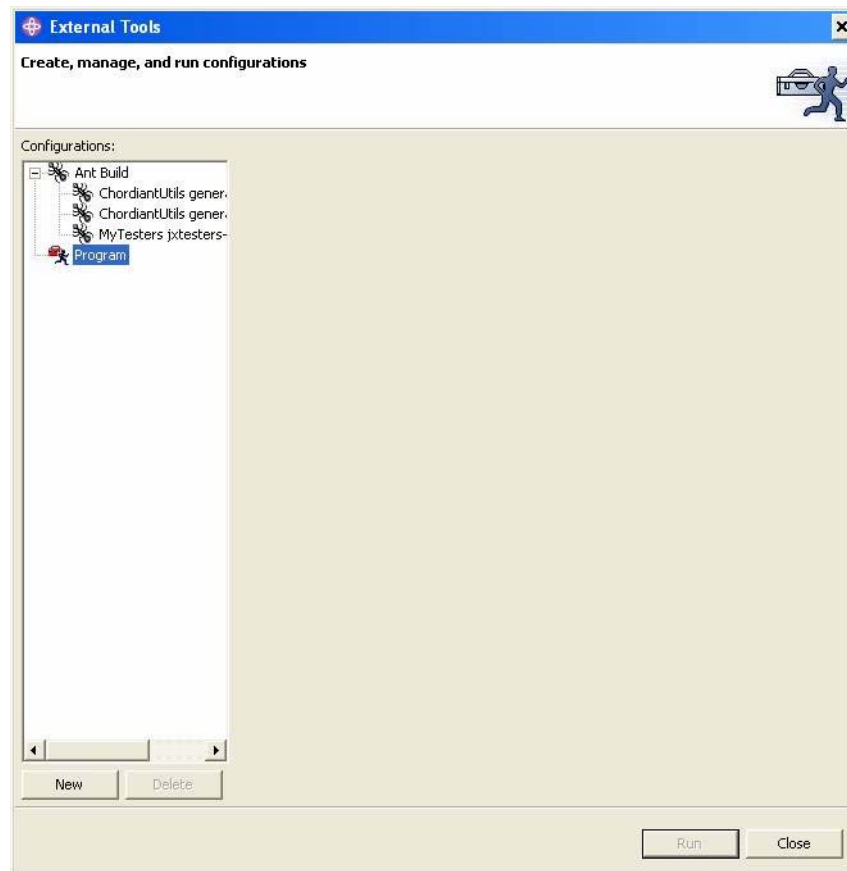


Figure 8-1: External Tools Dialog

5. Complete the information in the **Main** tab of the **External Tools** dialog, as described in Table 8-1.

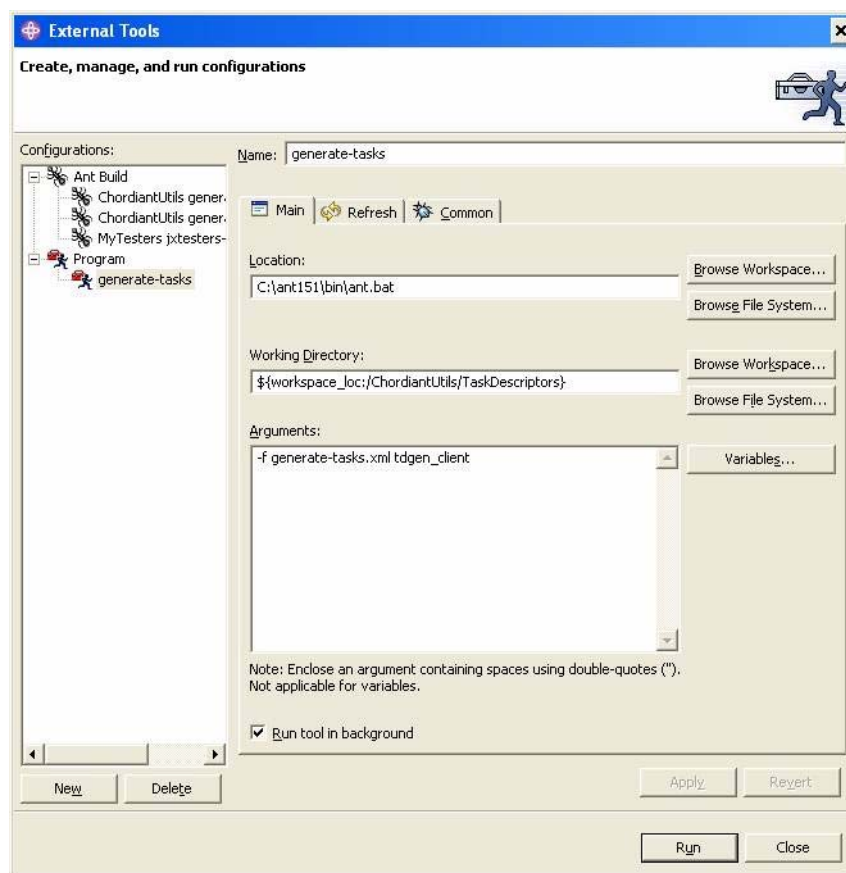


Figure 8-2: Completed External Tools Dialog

FIELD	DESCRIPTION
Name	Give the configuration a meaningful name, like generate-tasks , so you can find it easily when you want to run it at a later date.
Location	Enter the path to your ant.bat file. You can also browse through your file system or Workspace and navigate to the ant.bat file.
Working Directory	Enter the path to the ChordiantUtils/TaskDescriptors directory in your workspace, which contains the generate-tasks.xml Ant script and output directories. You can also browse through your file system or Workspace and navigate to this location.

Table 8-1: Information Required in External Tools Dialog

FIELD	DESCRIPTION
Arguments	Enter the command for running the <code>generate-tasks.xml</code> script from a command line: ant -f generate-tasks.xml {target_name} Remember that you can use <code>tdgen_client</code> , <code>tdgen_java</code> , or your own custom target.
Run tool in background	Select this checkbox so you can run the tool while you perform other work.

Table 8-1: Information Required in External Tools Dialog (Continued)

- Click **Apply** to save this configuration. To run the tool now, click **Run**.

A command window opens while the task descriptors are generated.

Continue with “[Task Descriptors Created](#)” for details on the output of this script.

Once you have set up this configuration, it is easy to generate task descriptors in the future.

To generate task descriptors using this same configuration in WSAD in the future:

- In WSAD, select **Run | External Tools | External Tools**.
- In the left side of the External Tools dialog, you will see the configuration you created, `generate-tasks` or whatever you named it, in the list of Ant configurations.
- Select the name of your configuration and click **Run**.

Task Descriptors Created

For Chordiant and non-Chordiant Java services, the task descriptors are placed in the `BusinessService` directory. Task descriptors end with the `.wsdl` extension. As described on [page 151](#), although they look like web services components, they are not. Watch the console to make sure that your task descriptors generated properly.

Note: It is important to keep the task descriptors in these directories, because task descriptors outside these directories are not available to workflows you create in the Business Process Designer.

In general, you will not need to modify the task descriptor files. You might need to edit the parameter and method documentation. Refer to the *Chordiant 5 Foundation Server Business Process Server Developer's Guide* for details.

While additional targets exist for other types of task descriptors, these are used by Chordiant to create the initial task descriptors. You do not need to use these other targets.

For further details on task descriptors, refer to:

- *Chordiant 5 Tools Platform Business Process Designer Developer's Guide* — Describes how to use the task descriptors.
- *Chordiant 5 Foundation Server Business Process Server Developer's Guide* — Describes the different types of task descriptors and details their structure.

Creating or Modifying Business Objects

You can specify persistence information for your services within your Rational Rose model.

For details on the Chordiant Persistence Server, refer to the “Chordiant Persistence Server” chapter of the *Chordiant 5 Foundation Server Developer's Guide*.

PERSISTENCE COMPONENTS CREATED

You can use the Chordiant 5 UML Extender for Rational Rose along with the Business Component Generator to create persistence components to go along with your service.

These classes are associated with persistence.

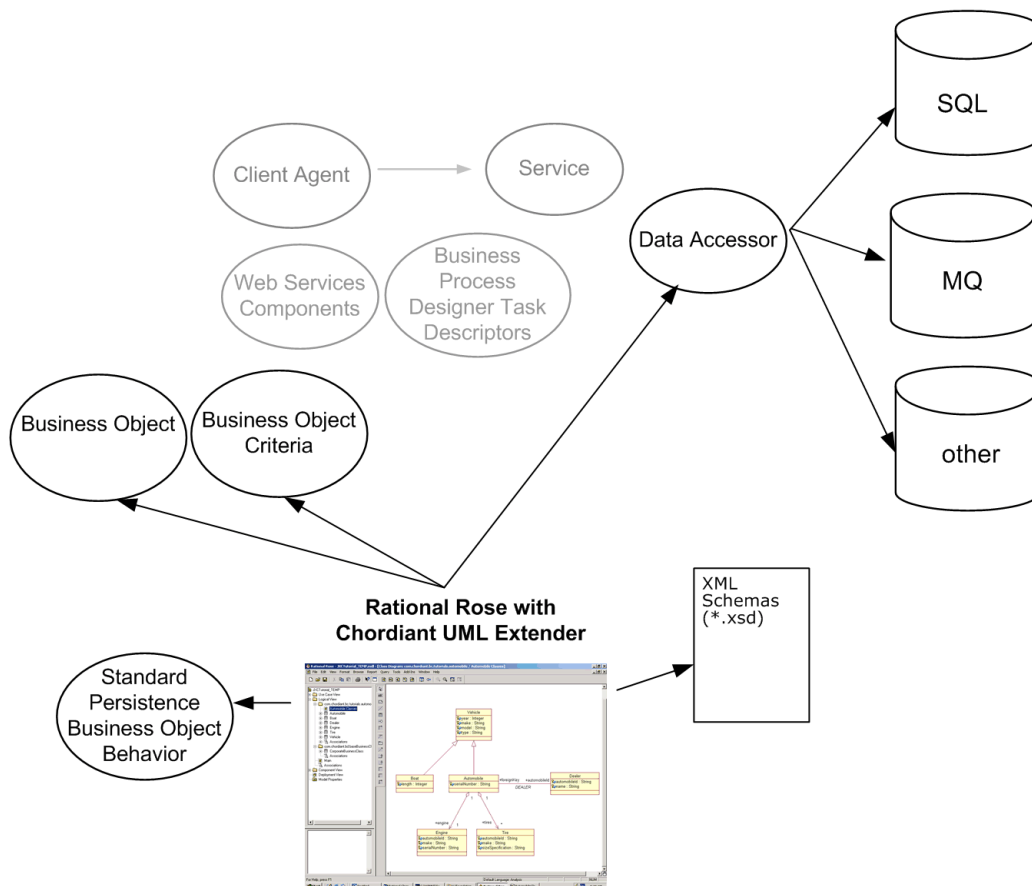


Figure 9-1: Persistence Components Created Automatically

For details on these persistence components, refer to the “Persistence Server” chapter of the *Chordiant 5 Foundation Server Developer’s Guide*.

- **Business Objects** — `{yourobject}.java`

The business objects (BO) class contains the business objects for the Persistence Server interface. The business object defines basic getter and setter methods for the associated data. Note that business objects may or may not include persistent attributes.

- **Business Object Criteria** — `{yourobject}Criteria.java`

The business object criteria (BOC) class contains methods enabling developers to identify groups of business data, such as equivalences, ranges, and sets for database queries. The BOC mirrors the BO, except each field has an additional matching criteria field that is used to specify criteria for the query.

For example, if a business object has a field for `CreateDate`, then the BOC will have both the `CreateDate` field and a `CreateDateCriteria` field. You can use the criteria field to perform complex database queries, such as locating all objects created before a certain date.

- **Data Accessor** — `{yourobject}DataAccess_{database}.java`

The data accessor class offers methods for interacting with the specific data store, such as a Relational Database Management System (RDBMS) using SQL, MQ, and others. The interface supports operations for a single object (point), or multiple objects (set, ray, or segment). The generated interface also supports either optimistic and pessimistic locking.

The parent class, `{yourobject}DataAccess.java`, has minimal functionality. Use the data accessor with the database name specified.

- **Business Object Behavior** — `{yourobject}Behavior.java`

Notes: This class is used only in the PartyRole service. For more information, refer to [Chapter 14, “Customizing the PartyRole Service”](#).

The business object behavior (BOB) object is generated as a skeleton and contains behavior specific to a single BO. Most methods on the BOB accept a business object as an input parameter.

- **XML Schemas (XSDs)** — `{yourobject}.xsd`

The schema defines, in XML form, attributes of the objects in the model you created.

CUSTOMIZING BUSINESS OBJECTS

You can customize business objects to add or modify attributes within the object. You might want to do this when you need to enable the processing and storing of new data.

Important Notes

Consider these notes when creating and customizing business services.

Serialized Objects

If a business object has been passed to the persistent cache manager or if it has been included in a workflow context, it will be serialized and stored as a SOAP-encoded message in either the Chordiant database or in JMS.

If your business object has been or might be serialized, you should not remove or modify the data name, data type, or setter and getter names of an existing attribute. If you do, the system will not be able to deserialize the object. You will have to go through a data migration/conversion process which is not provided by Chordiant. To work around this limitation, you can add new attributes to suit your needs. Be aware that if you add new attributes and deploy the object, the same rules apply — if you change this new attribute on a serialized object, the system will not be able to deserialize it.

Web Services

If you will be creating web services, be sure that all of the business objects associated with the web service you will create must have default constructors. Business objects provided by Chordiant meet this requirement. Be sure that if you are creating your own business objects that you include default constructors for them, along with any necessary getter and setter methods.

Criteria for Customizing Business Objects

Be sure to refer to [“Rose Models”](#) and [“Business Object Definition”](#) sections within the [“Customization Guidelines”](#) on page 5 for other rules to follow when modeling business objects.

Steps for Customizing a Business Object

Before you begin the customization process, familiarize yourself with the business object definition and customization guidelines, starting on [page 6](#).

To add an attribute to a business object:

1. Open the appropriate object model.

Notes: You can open the model from anywhere or create a new project through the Business Component Generator, using this model. Then you can customize the model from within your project.

If you choose a base model to create a project in the Business Component Generator, be aware that the model name will always contain the word “base”. You cannot change the name of the model, its descriptor file, or its XMI file once they are a part of the Tools Platform project. Refer to the note on [page 22](#) for details.

2. Locate the business object which most closely matches your needs.
3. Subclass the business object to which you want to add the attribute.

You can use Rational Rose to update the model for the business object. We recommend that you derive a new object instead of modifying an existing object directly.

4. Add the attribute to the derived business object using the modeling tool.

Figure 9-2 illustrates adding a new attribute to a derived object.

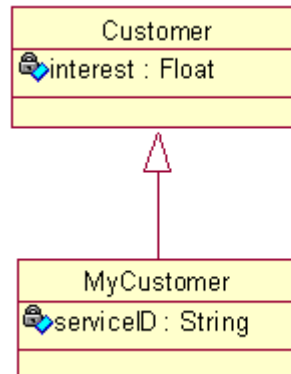


Figure 9-2: Adding an Attribute to a Derived Business Object

Note: Do not specify an attribute called “value” on a Business Object. The setter for that attribute will not work properly because there is a clash between the parameter and attribute names. This is a known problem with the code generator.

5. Specify the existing Chordiant class that your customized class will override. This will appear as an **override** metadata tag in the CMI file.

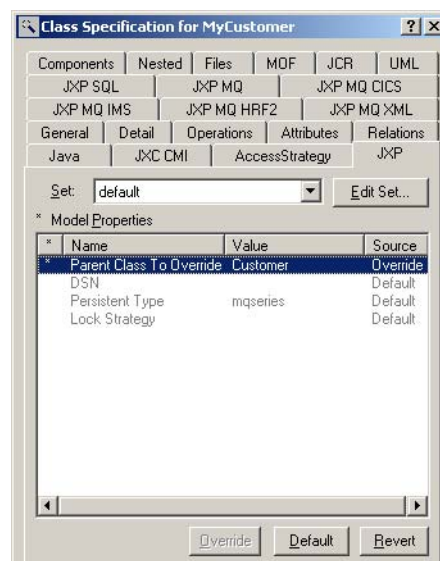


Figure 9-3: Specifying the Class to Override

When the Object Factory finds an override tag, it will load your custom class instead of the original class that you are overriding.

6. Modify the database table to include the new attribute. This is only required if the attribute will be persisted.

You must keep your attributes and tables in sync. If you modify or add an attribute, make sure that you make the appropriate change in the database table. If you are pointing to a different table, make sure that this other table exists in your schema. It is easiest to make all of your changes in the model first, then work on the database.

7. Use the Business Component Generator to automatically generate these components:
 - Business Object Class
 - Business Object Criteria Class
 - Data Accessor Class
 - Business Object Behavior Class (skeleton)
Only used with the `PartyRole` Service or `PmfCustomerService`, as described in [Chapter 14](#).

Notes: This is not the same as the Business Object Behavior created for extended behavior, which is described in [“Business Object Behavior” on page 198](#).

You must also create a descriptor file before using the Business Component Generator.

For instructions, refer to [Chapter 3, “Using the Business Component Generator”](#), including [“Creating the Descriptor File” on page 16](#).

Note that the generated code will define empty methods for all business object behavior. If you want to modify an existing method from the parent class, copy that method into the new code and make your additions or modifications there. You can also add brand new methods

8. Since you will be making modifications, move the generated code from the generated directory into a separate directory so it is not overwritten if you run the Business Component Generator again.
9. Modify the business service to use the new attribute, as required. Refer to [“Modifying Service Framework Components” on page 46](#) for more information.
10. If you are customizing the `PartyRole` Service, modify the business object behavior, as required. Refer to [“Customizing Business Object Behavior” on page 301](#) for more information.

Instantiating the Business Object

When you want to get an instance of a business object you create, use the object factory from the Resource Manager. You might need to update the configuration of the resource manager, `BusinessObjectFactoryService.xml`, to ensure that the resource manager can find the path to the business object's associated CMI file.

You can add the path to the new CMI file to the `resourcetype` value of the `CMI_PATH` section of the resource manager configuration file. Refer to the “Resource Manager Configuration” section of the *Chordiant 5 Foundation Server Developer's Guide* for more information.

Using Object Locking

You can add locking support for your business objects to limit write access to the object. For performance reasons, optimistic locking is recommended for business objects, rather than pessimistic locking.

You should always perform object locking for a service within a transaction block in case the transaction fails. In an optimistic model, locks are placed in a database and can therefore be rolled back. For pessimistic locking, the service that obtains a lock is responsible for cleaning up the lock.

Refer to the “Persistence Server” chapter of the *Chordiant 5 Foundation Server Developer's Guide* for more information on optimistic and pessimistic locking, including programming pessimistic locking through the lock service and combining the two strategies within one model.

To add optimistic locking support for a business object:

1. Define the object locking strategy in your object model.

In the class specifications, select the **JXP** tab and select the desired locking strategy. Refer to [Step 2 on page 175](#) for more information on this tab.

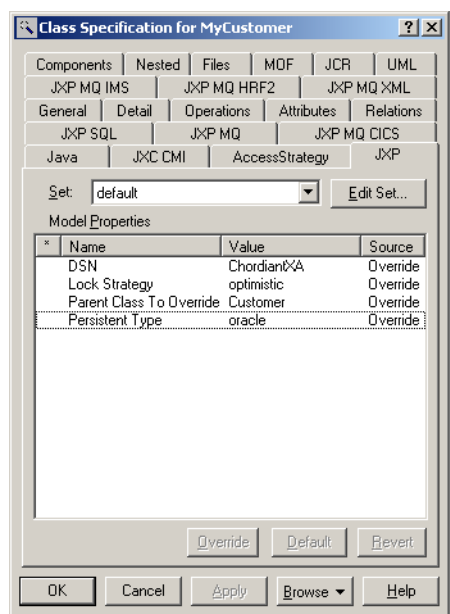


Figure 9-4: Optimistic LockStrategy Specified for a Class

This action will add the following tag to the appropriate class definition in the CMI file:

```
<LockStrategy>optimistic</LockStrategy>
```

2. Specify the attribute which will serve as the lock field.

In Rational Rose, specify which attribute is the lock field through the Attributes Specification.

In the Attribute Specification, select the **JXP** tab and select whether this attribute will act as a lock field. Refer to [Step 7 on page 187](#) for more information on this tab.

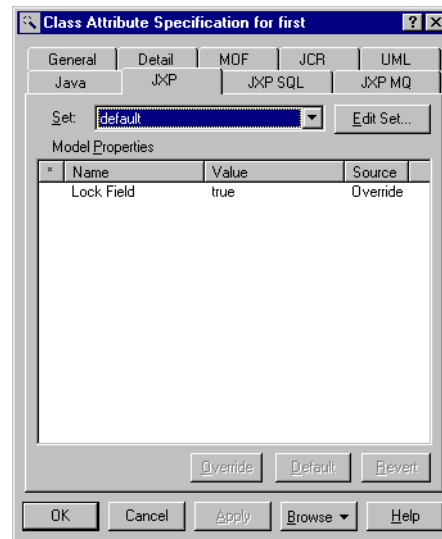


Figure 9-5: Specifying Lock Field for an Attribute

Note that the actual column/attribute field is not important. What matters is that:

- it is only used for the locking value, and
- it is an optimistic locking model

There should be only one lock field for a given class.

This action specifies that the attribute can be used to obtain a persistence lock on the corresponding data for the object. Notice the CMI attribute definition, and the additional **LockField** tag, of the object you need to lock.

```
<attribute>
  <name>LockFlag</name>
  <multiplicity>1..1</multiplicity>
  <rdbPhysicalName>locktokentext</rdbPhysicalName>
  <rdbLogicalType>VARCHAR</rdbLogicalType>
  <rdbSize>256</rdbSize>
  <rdbDigits>0</rdbDigits>
  <rdbPrimaryKey>false</rdbPrimaryKey>
  <rdbNotNull>false</rdbNotNull>
  <LockField>true</LockField>
  <javaType>java.lang.String</javaType>
</attribute>
```

Code 9-1: Specifying the Lock Flag

3. If you are *not* subclassing from an existing Chordiant business object, you must add a **lockflag** column to the corresponding database table. This column already exists for Chordiant-provided business objects.

As with any attribute in a business object, when you add the attribute in the business object, be sure to include the column for that attribute in your database table. Refer to [Step 6 on page 168](#) for more information.

When working with a graph of objects and tables, you only need to lock the head/root object. Then, when you need to update tables related to the parent object/table, you can use the `updatePointOptimistic` method. As long as you are retrieving the entire graph as a single entity, you can make changes to any data in the graph and still rely on the locked head object and the `updatePointOptimistic` method to update the entire graph. It is important to remember that the entire graph must be treated as a single entity throughout the process.

Otherwise, for all other updates inside the transaction, use the `updatePoint`, `updateSegment`, `updateSet`, or `updateRay` methods.

Refer to the “Persistence Server” chapter of the *Chordiant 5 Foundation Server Developer’s Guide* for more information on locking. For more information on tags within the CMI file for locking objects in the database, refer to [Chapter 15, “Metadata”](#).

Transferring Attributes

It is often necessary to transfer attribute values from one business object to another. You can use the `TransferAttributesHelper` class, found in `com.chordiant.jx_extensions`, to assist in this process.

The `TransferAttributesHelper` class provides methods to automate the process of copying public member attributes from one object to another (possibly different) object, providing that the attributes have identical names and datatypes in each class.

This can be helpful in the context of business objects, as it is often necessary to copy fields from one BO or BOC to another. The following methods are defined in the `TransferAttributesHelper` class:

- **`objectIsEmpty(Object obj)`**
Returns `true` when all properties on the object are null.
- **`populateCriteria(BusinessObjectCriteria boc, String criteria)`**
Sets each populated attribute field on the BOC to have the specified criteria.
- **`transferLikeAttributes(Object source, Object target)`**
Transfers all identically-named attribute values (of the same type) from the source object to the target object.

- **transferLikeAttributesAndSetCriteria(Object sourceBO, BusinessObjectCriteria targetBOC, String criteria)**

Transfers all identically-named attributes from the source BO to the target BOC, and sets the criteria field for each transferred attribute to the specified criteria.

Data and Caching Considerations

Consider these points for your persistence implementation:

- Be aware that whenever you modify data in the persistence layer, you must also update the cache.
- Chordiant 5 Foundation Server uses a distributed architecture. Be sure to synchronize your data across JVMs.

SPECIFYING PERSISTENCE METADATA

Chordiant 5 Foundation Server enables you to use Rational Rose to create the conceptual model, as well as generate the Java classes for the Data Accessor.

This section describes how to use Rational Rose to specify the persistence metadata information required for use with Persistence Server. For in-depth discussions of persistence, refer to the “Persistence Server” chapter of the *Chordiant 5 Foundation Server Developer’s Guide*.

For more information on persistence metadata, refer to [Chapter 15, “Metadata”](#).

Notes: Before specifying your persistence metadata, you must configure Rational Rose to accept the properties and generate the required Java files, as described in [“Configuring Rational Rose” on page 9](#).

This section describes how to use Rational Rose to specify the persistence metadata. For more information about generating the Java classes, refer to [“Using the Business Component Generator” on page 15](#).

Included with your Chordiant 5 Foundation Server installation is a Chordiant base model for Rational Rose, called `JXBB0Base.mdl`. You can copy it and use it as a starting template. The model contains a Chordiant base business object.

This model is located in `{eclipse_root}/plugins/{data_model_plugin}/rosemodels`.

To specify the persistence metadata using Rational Rose:

1. Start the Rational Rose application. Create your object model or open an existing object model.

Figure 9-6 illustrates the Chordiant-provided ProductCatalogTable.

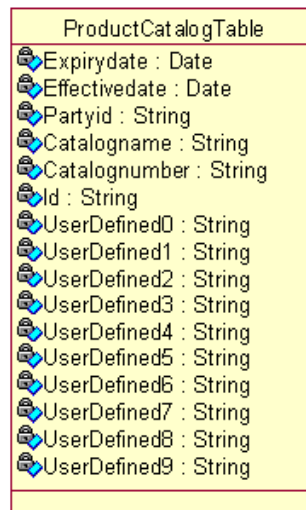


Figure 9-6: Working with the ProductCatalogTable

Note: This table is being used as an example. You should not modify Chordiant-provided objects. You can, however, subclass them and then make modifications.

2. Double-click the class to open the **Class Specification** window. Select the **JXP** tab.

Rational Rose displays a dialog containing the JXP properties for the class.

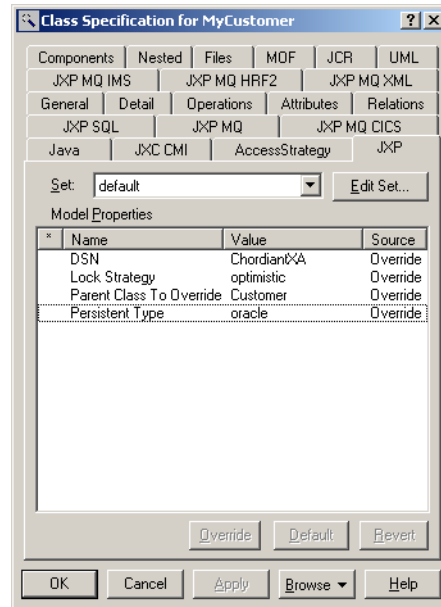


Figure 9-7: Rational Rose JXP Properties

Table 9-1 describes the JXP class-level properties that you can specify. All metadata is discussed in [“Metadata” on page 305](#).

Note: Do not change the data name, data type, or getters and setters of an existing or deployed attribute. Refer to [“Important Notes” on page 165](#).

PROPERTY	NOTES
DSN	The data source name.
Persistent Type	<p>The data store type. This entry instructs the system of the specific stylesheet to use when generating the Persistence Server Data Accessor Java classes. The valid options are mqseries, mqseriesxml, oracle8, and db2udb.</p> <p>Note: If you select mqseriesxml, be sure to complete the information on the JXP MQ XML tab, described in Step 5 on page 185.</p> <p>Note: If you will be using a DB2UDB database with the Chordiant-provided business services, be sure to use the DB2UDB-specific CMI file (chordiantcmi_db2udb.xml) to generate the persistence components. If you are using Oracle, use the chordiantcmi.xml file.</p>
Lock Strategy	<p>The locking mechanism for the class. The valid options are:</p> <ul style="list-style-type: none"> • optimistic • pessimistic • none (default)

Table 9-1: JXP Class-Level Properties

Note: Rational Rose displays all tabs for JXP SQL and JXP MQ at all times, independent of your setting for the Persistent Type. Complete only the tabs that apply for your selected Persistent Type, as appropriate.

3. Select the **JXP SQL** or **JXP MQ** tab to specify the class-level properties, as appropriate.

Figure 9-8 illustrates the **JXP SQL** tab containing the class-level properties for the selected object.

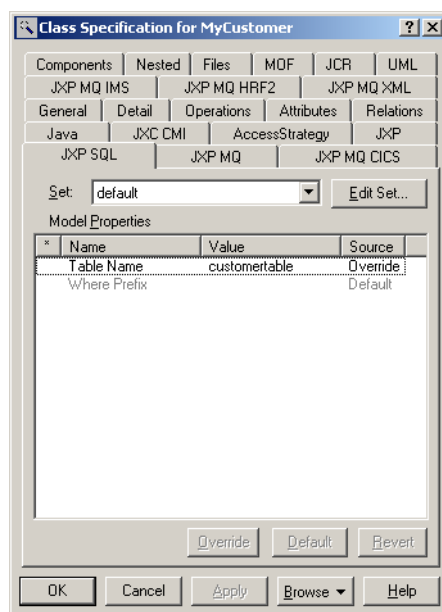


Figure 9-8: JXP SQL Class-Level Properties

Table 9-2 describes the JXP SQL class-level properties that you can specify.

PROPERTY	NOTES
Table Name	The name of the table in the relational database system.
Where Prefix	A conditional clause, using the table.column notation, to specify the conditional relating identifiers for a join operation. For more information on joins, refer to the "Persistence Server" chapter of the <i>Chordiant 5 Foundation Server Developer's Guide</i> .

Table 9-2: JXP SQL Class-Level Properties

Figure 9-9 illustrates the **JXP MQ** tab containing the class-level properties for the selected object.

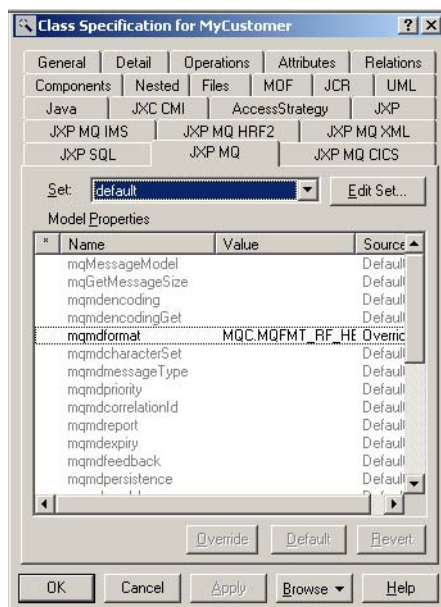


Figure 9-9: JXP MQ Class-Level Properties

Table 9-3 outlines the class-level properties you can specify for JXP MQ.

PROPERTY	NOTES
mqMessageModel	(Default) RequestReply Specifies the message model for the application. You can set this property to one of the following values: <ul style="list-style-type: none"> RequestReply—Send a request and wait for a response FireForget—Send a request without expecting a response
mqmdencoding	(Default) MQC.MQENC_NATIVE
mqGetMessageSize	(Optional) The size of the response message. You need to specify this numeric field only if you expect response messages greater in size than 4096 bytes. The range of valid values for this field is any numeral greater than 4096. Note that you must set this field if you receive an MQ 2079 error message.
mqmdencodingGet	(Optional) Specifies the encoding used for processing response messages. You can use this property to override the mqmdencoding property to specify different processing the response messages.
mqmdformat	(Default) MQC.MQFMT_STRING
mqmdcharacterSet	(Default) MQC.MQCCSI_DEFAULT

Table 9-3: JXP MQ Class-Level Properties

PROPERTY	NOTES
mqmdmessageType	(Default) MQC.MQMT_REQUEST
mqmdpriority	(Default) 8
mqmdcorrelationId	(Default) MQC.MQCI_NEW_SESSION. Set this property if you select a value other than the default for the mqmdformat property.

Table 9-3: JXP MQ Class-Level Properties (Continued)

Note: Many of the JXP MQ properties are set by WebSphere MQ during execution. You can override these properties using the **JXP MQ** tab. For more information about the specific properties, refer to the WebSphere MQ documentation.

- If you specified a value for mqmdformat other than MQC.MQFMT_STRING when specifying MQ persistence information, complete the additional JXP MQ format-specific tab.

The specific tab you must complete depends on your selection of mqmdformat, and is summarized here:

- For MQC.MQFMT_CICS, complete the **JXP MQ CICS** tab, illustrated in [Figure 9-10 on page 180](#).
- For MQC.MQFMT_IMS, complete the **JXP MQ IMS** tab, illustrated in [Figure 9-11 on page 182](#).
- For MQC.MQFMT_RF_HEADER_2, complete the **JXP MQ HRF2** tab, illustrated in [Figure 9-12 on page 183](#).

Figure 9-10 illustrates the **JXP MQ CICS** tab.

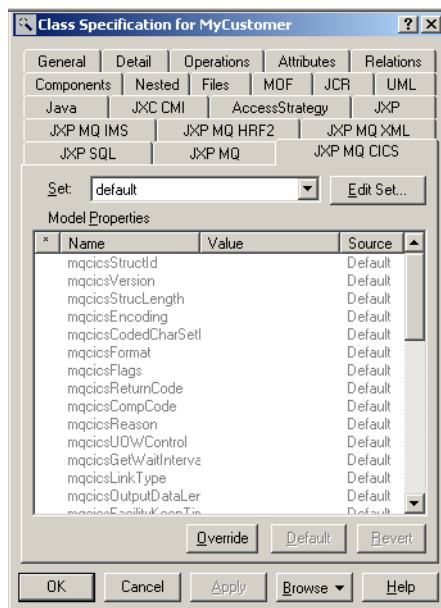


Figure 9-10: JXP MQ CICS Properties

Table 9-4 outlines the properties you can specify for JXP MQ CICS.

PROPERTY	NOTES
mqmdcharacterSet	(Default) MQC.MQCCSI_DEFAULT
mqcicsStructId	(Default) CIH
mqcicsVersion	(Default) 2
mqcicsStrucLength	(Default) 180
mqcicsEncoding	(Default) 0
mqcicsCodedCharSetId	(Default) 0
mqcicsFormat	(Default) MQSTR
mqcicsFlags	(Default) 0
mqcicsReturnCode	(Default) 0
mqcicsCompCode	(Default) 0
mqcicsReason	(Default) 0
mqcicsUOWControl	(Default) 273
mqcicsGetWaitInterval	(Default) -2

Table 9-4: JXP MQ CICS Properties

PROPERTY	NOTES
mqcicsLinkType	(Default) 1
mqcicsOutputDataLength	(Default) -1
mqcicsFacilityKeepTime	(Default) 0
mqcicsADSDescriptor	(Default) 0
mqcicsConversationalTask	(Default) 0
mqcicsTaskEndStatus	(Default) 0
mqcicsFacility	(Default) 00000000
mqcicsFunction	(Default) Set to blanks.
mqcicsAbendCode	(Default) Set to blanks.
mqcicsAuthenticator	(Default) Set to blanks.
mqcicsReserved1	(Default) Set to blanks.
mqcicsReplyToFormat	(Default) Set to blanks.
mqcicsRemoteSysId	(Default) Set to blanks.
mqcicsRemoteTransId	(Default) Set to blanks.
mqcicsTransactionId	You must specify your CICS transaction for this field.
mqcicsFacilityLike	(Default) Set to blanks.
mqcicsAttentionId	(Default) Set to blanks.
mqcicsStartCode	(Default) Set to blanks.
mqcicsCancelCode	(Default) Set to blanks.
mqcicsNextTransactionId	(Default) Set to blanks.
mqcicsReserved2	(Default) Set to blanks.
mqcicsReserved3	(Default) Set to blanks.
mqcicsUOWCursorPosition	(Default) 0
mqcicsErrorOffset	(Default) 0
mqcicsInputItem	(Default) 0
mqcicsReserved4	(Default) 0

Table 9-4: JXP MQ CICS Properties (Continued)

Figure 9-11 illustrates the **JXP MQ IMS** tab.

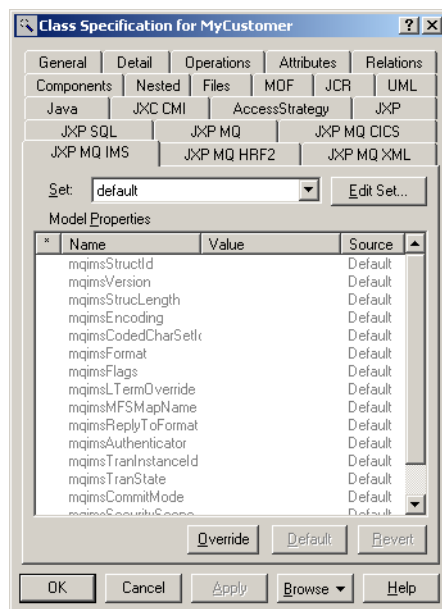


Figure 9-11: JXP MQ IMS Properties

Table 9-5 outlines the properties you can specify for JXP MQ IMS.

PROPERTY	NOTES
mqimsStructId	(Default) IIH
mqimsVersion	(Default) 1
mqimsStrucLength	(Default) 84
mqimsEncoding	(Default) 0
mqimsCodedCharSetId	(Default) 0
mqimsFormat	(Default) MQSTR
mqimsFlags	(Default) 0
mqimsLTermOverride	(Default) Set to blanks.
mqimsMFSMapName	(Default) Set to blanks.
mqimsReplyToFormat	(Default) Set to blanks.
msimsAuthenticator	(Default) Set to blanks.
mqimsTranInstanceId	(Default) 0000000000000000
mqimsTranState	(Default) Set to blank.

Table 9-5: JXP MQ IMS Properties

PROPERTY	NOTES
mqimsCommitMode	(Default) 0
mqimsSecurityScope	(Default) C
mqimsReversed	(Default) Set to blank.

Table 9-5: JXP MQ IMS Properties (Continued)

Figure 9-12 illustrates the **JXP MQ HRF2** tab.

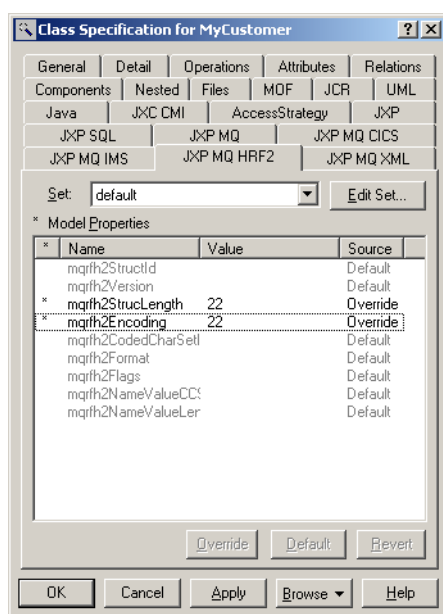


Figure 9-12: JXP MQ HRF2 Properties

Table 9-6 outlines the properties you can specify for JXP MQ HRF2.

PROPERTY	NOTES
mqrh2StructId	(Default) RFH
mqrh2Version	(Default) 2
mqrh2StructLength	(Default) 36 You must modify this property when using MQ Series Integrator (MQSI).
mqrh2Encoding	(Default) 0
mqrh2CodedCharSetId	(Default) 0

Table 9-6: JXP MQ HRF2 Properties

PROPERTY	NOTES
mqrh2Format	(Default) MQSTR
mqrh2Flags	(Default) 0
mqrh2NameValueCCSID	(Default) 1208
mqrh2NameValueLength	(Default) 0 Used to create the MQRFH2 header message to communication with MQSI. When using MQSI, set this value to the total length of the message following the binary header. MQSI requires that you pad this portion of the message to be a multiple of four bytes.

Table 9-6: JXP MQ HRF2 Properties (Continued)

5. If you selected the mqseriesxml persistent type (on [page 176](#)), you must complete the information in the **JXP MQ XML** tab. For other persistent types, continue with [Step 6 on page 186](#).

[Figure 9-13](#) illustrates the **JXP MQ XML** tab.

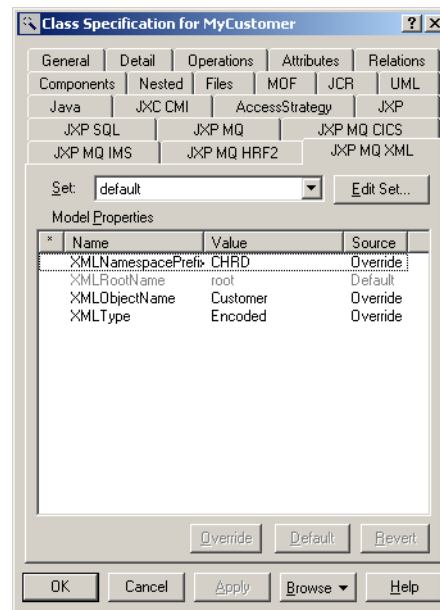


Figure 9-13: JXP MQ XML Properties

[Table 9-7](#) outlines the class-level properties that you can specify for JXP MQ XML.

PROPERTY	NOTES
XMLNamespacePrefix	(Optional) Specifies the prefix to use to create a fully qualified name space within the XML document.
XMLRootName	(Default) root The high level node of the XML document.
XMLObjectName	Specifies the name of a particular object within the XML file. XML files can contain multiple objects.
XMLType	Specifies the type of the XML document generated. This document type must match the type expected by the receiving application. Possible values: <ul style="list-style-type: none"> Literal Encoded

Table 9-7: JXP MQ XML Class-Level Properties

6. Select the **Attributes** tab in the **Class Specification** dialog box.

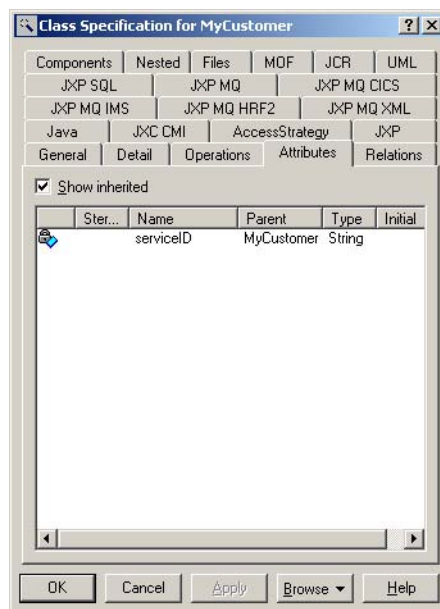


Figure 9-14: Rational Rose Attributes Properties

7. Right-click an attribute in the list and choose **Specification** from the pop-up menu. Select the **JXP** tab.

Rational Rose displays a dialog box containing the JXP properties for the attribute, as illustrated in [Figure 9-15](#).

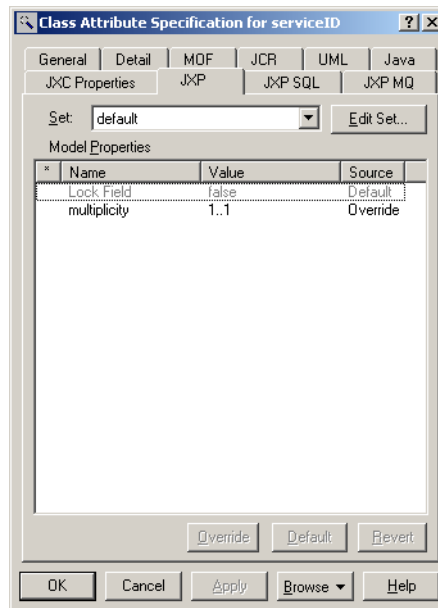


Figure 9-15: Rational Rose JXP Attribute-Level Properties

[Table 9-8](#) describes the JXP attribute-level properties that you can specify.

PROPERTY	NOTES
Lock Field	<p>The lock field is used to specify that the attribute will serve as the lock for an optimistic locking strategy. Possible values are:</p> <ul style="list-style-type: none"> • True • False (default)
Multiplicity	<p>The relationship between the attribute and the associated object, encoded as 1..n. For example, in the case of the relationship between a customer object and associated aliases (other names used by a customer), you might specify 1..3 to indicate that a customer can have as many as three potential aliases.</p>

Table 9-8: JXP Attribute-Level Properties

8. Select the **JXP SQL** or **JXP MQ** tab to specify the attribute-level properties, as appropriate. **MQ** tab information is shown starting on [page 189](#).

JXP SQL Tab

Figure 9-16 illustrates the **JXP SQL** tab containing the attribute-level properties for the selected object.

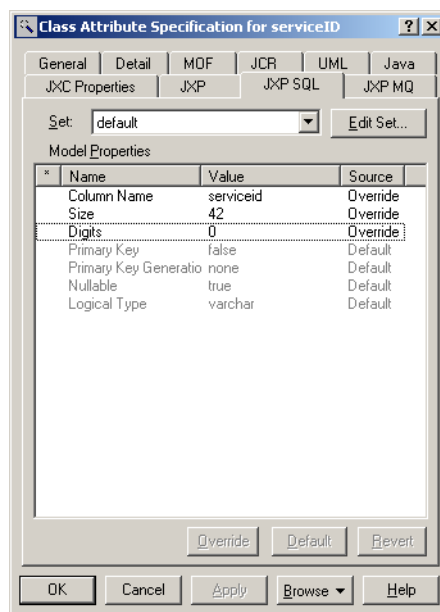


Figure 9-16: JXP SQL Attribute-Level Properties

Table 9-9 outlines the attribute-level properties you can specify for JXP SQL.

PROPERTY	NOTES
Column Name	The name of the column in the table. Note: When using the modeling tool, if the attribute is non-persistent, you must leave the Column Name field blank.
Size	The physical size of the column.
Digits	The number of digits following the decimal point, as defined in the database.
Primary Key	Specifies whether the column serves as a primary key for the table. Possible values: <ul style="list-style-type: none"> true false

Table 9-9: JXP SQL Attribute-Level Properties

PROPERTY	NOTES
Primary Key Generation Type	Specifies whether the system should autogenerate the primary key. Possible values: <ul style="list-style-type: none"> • auto—Autogenerate the primary key for the table using the Chordiant 5 Foundation Server GUID. (Refer to the “Persistence Server” chapter of the <i>Chordiant 5 Foundation Server Developer’s Guide</i> for information on GUIDs) • none—Do not autogenerate the primary key
Nullable	(Default) true Specifies whether the column can assume a null value or not. Possible values: <ul style="list-style-type: none"> • true—Specifies that the column value can include the null value. • false—Specifies that the column value cannot have a null value.
Logical Type	The data type of the column, as defined in the database.

Table 9-9: JXP SQL Attribute-Level Properties (Continued)

JXP MQ Tab

If you are using JXP SQL, continue with [Step 9 on page 191](#).

[Figure 9-17](#) illustrates the **JXP MQ** tab containing the attribute-level properties for the selected object.

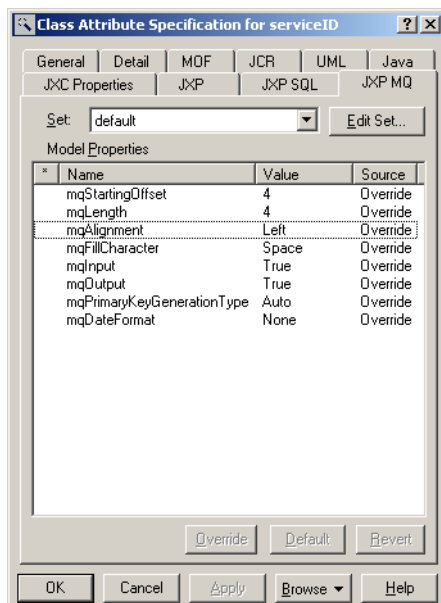


Figure 9-17: JXP MQ Attribute-Level Properties

These same properties can also be set for an aggregation, as shown in [Figure 9-18](#).

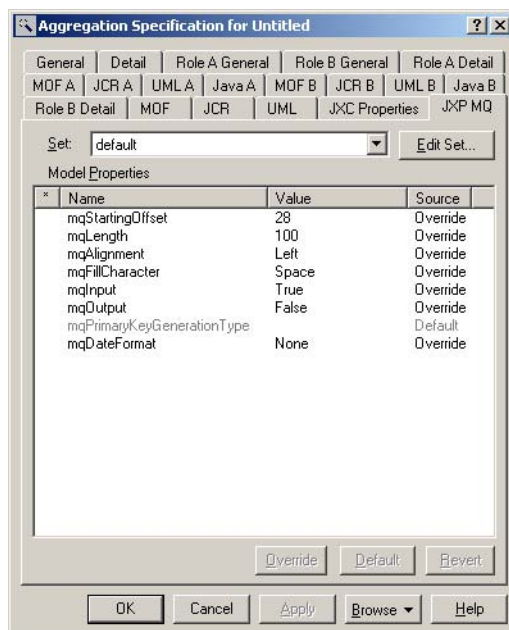


Figure 9-18: JXP MQ Aggregation Specifications

[Table 9-10](#) outlines the attribute-level properties that you can specify for JXP MQ. These same properties can be used for specifying aggregation information.

PROPERTY	NOTES
mqStartingOffset	The numeric offset of the data in the message relative to zero.
mqLength	The numeric length of the data in the message.
mqAlignment	The alignment of the data in the message. Possible values: <ul style="list-style-type: none"> • Left • Right • None
mqFillCharacter	The fill character to add or remove from the aligned data message. Possible values: <ul style="list-style-type: none"> • Space • Zero • None

Table 9-10: JXP MQ Attribute-Level and Aggregation Properties

PROPERTY	NOTES
mqInput	Instructs the system to parse response messages using this attribute. Possible values: <ul style="list-style-type: none"> • true • false
mqOutput	Instructs the system to build request messages using this attribute. Possible values: <ul style="list-style-type: none"> • true • false
mqPrimaryKeyGenerationType	Specifies a pessimistic Lock Strategy for this object. Possible values: <ul style="list-style-type: none"> • Auto—Automatically generate a unique nine-digit key for this attribute of type java.lang.String or java.lang.Integer. You must specify one attribute as Auto. • None—Does not automatically generate a primary key for this attribute.
mqDateFormat	Specifies the date format of the data in the message, using the Java date format. Possible values: <ul style="list-style-type: none"> • yyyyMMdd • MM/dd/yyyy • None

Table 9-10: JXP MQ Attribute-Level and Aggregation Properties (Continued)

9. Click **OK** to close the **Class Attribute Specification** dialog box.
10. Export your code to an XMI file.
11. Run the Business Component Generator, according to the instructions in [Chapter 3, “Using the Business Component Generator”](#).

Extended Persistence Components

Several application components exist to help perform extended persistence tasks, layered on top of the persistence already described in [Chapter 9](#). These components, including the Generic Service, Generic Service Client Agent, business object behaviors, helpers, and the behavior factory work together with standard services to create a customized solution with extended persistence features, such as retrieving a graph of information and other additional create, read, update, and delete (CRUD-type) operations. The Business Component Generator creates all of these components, except the Generic Service, the Generic Service Client Agent, and the behavior factory.

Notes: We recommend that you use these application components only in conjunction with standard business services and client agents. Refer to [“Using Extended Persistence Components” on page 193](#) and [“Usage Model for Extended Persistence Components” on page 215](#) for details.

Any locking strategies that you specify in the standard persistence will be carried over to the related extended persistence components.

USING EXTENDED PERSISTENCE COMPONENTS

The extended persistence components are an add-on to the standard Chordiant persistence components. They offer CRUD-based operations for working with graphs of objects and enable you to perform additional data manipulation. They are intended to be used in place of the standard persistence components when you need additional functionality. They are not intended to interface directly with your user-facing application.

The extended persistence components should be used only in conjunction with standard Chordiant services and client agents, since these afford a wider scope of control and can better manage your business processes.

For more details on using the extended persistence components, refer to [“Usage Model for Extended Persistence Components” on page 215](#).

GENERIC SERVICE OVERVIEW

The Generic Service was designed to save time in implementation. It enables you to perform basic functions, including extended persistence functionality, without having to create new, separate services for each function.

Note: The Generic Service is exclusively used by the generated components described in this chapter.

The Generic Service is similar to other Chordiant services, but it contains very little logic. A lot of the logic is contained in AccessStrategies and AccessStrategyInputs. The AccessStrategyInput comes from the client, through the Generic Service Client Agent, to the Generic Service. The Generic Service then dispatches the work to one or more AccessStrategies, which work through the Persistence Server to make database calls. Refer to [Figure 10-1 on page 195](#) for the sequence flow. Since the Generic Service is so generic, it can perform basic database operations for many different kinds of business objects.

Although the data access functionality still exists in the business tier, the Generic Service exposes some of this functionality directly to the application layer. This allows for increased flexibility. There are some issues to keep in mind when using the Generic Service. Refer to [“Usage Model for Extended Persistence Components” on page 215](#) for more information.

Sequence Flow Using Generic Service

Figure 10-1 illustrates the flow from the application layer to the persistence layer. Individual components are described in specific sections later in this chapter.

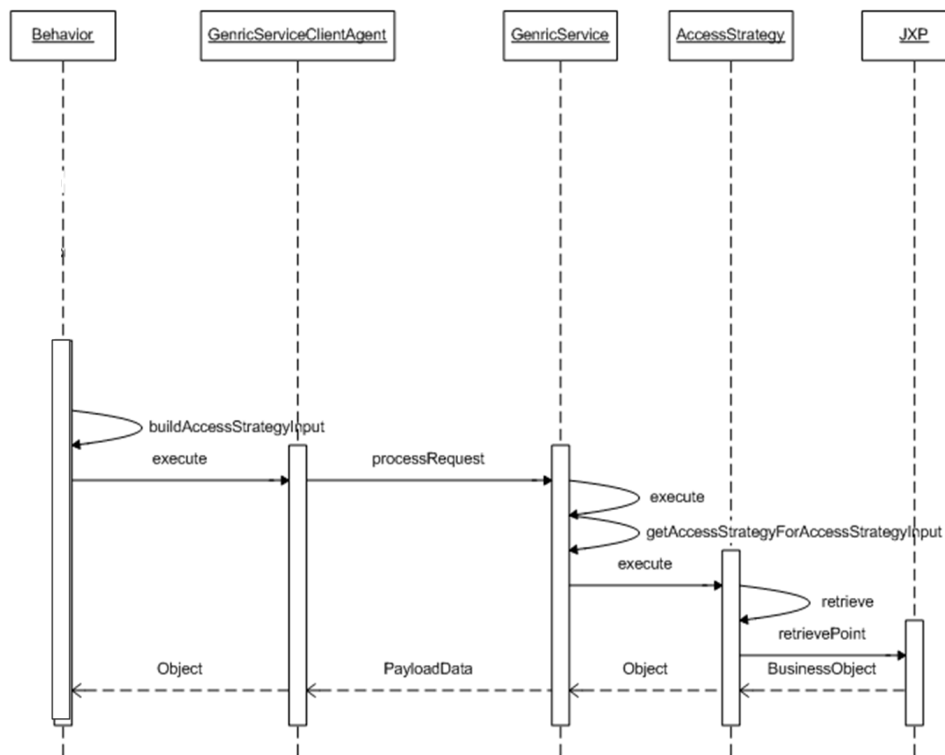


Figure 10-1: Sequence Flow Using Generic Service and Associated Components

These steps describe the sequence flow:

1. The Behavior builds an AccessStrategyInput to send to the Generic Service Client Agent.
2. The Behavior makes an execute call to the Generic Service Client Agent, including the AccessStrategyInput.
3. The Generic Service Client Agent calls the Generic Service using **processRequest**, including the AccessStrategyInput in its payload.
4. Given the AccessStrategyInput, the Generic Service gets a corresponding AccessStrategy.
5. The Generic Service calls the **execute** function of the appropriate AccessStrategy.
6. The AccessStrategy makes a persistence call (**retrievePoint**) to the Chordiant Persistence Server.
7. The Persistence Server returns a Business Object to the AccessStrategy.
8. The AccessStrategy passes the Object to the Generic Service.

9. The Generic Service passes the Object, packed as PayloadData, to the Generic Service Client Agent.
10. The Generic Service Client Agent unpacks the Object and passes it to the Behavior.

Generic Service Client Agent

The Generic Service Client Agent is basically the same as any other Chordiant 5 Foundation Server client agent. It uses the `processRequest` method to call a service—in this case the Generic Service. The Generic Service Client Agent passes an `AccessStrategyInput` within the `payloadData` specified in the `processRequest` method. Refer to the next section for more information about `AccessStrategyInput`.

For recursive processes, the Generic Service can call the Generic Service Client Agent.

Architecture

The Generic Service and its associated components complement the existing JX Architecture, as illustrated in [Figure 10-2](#).

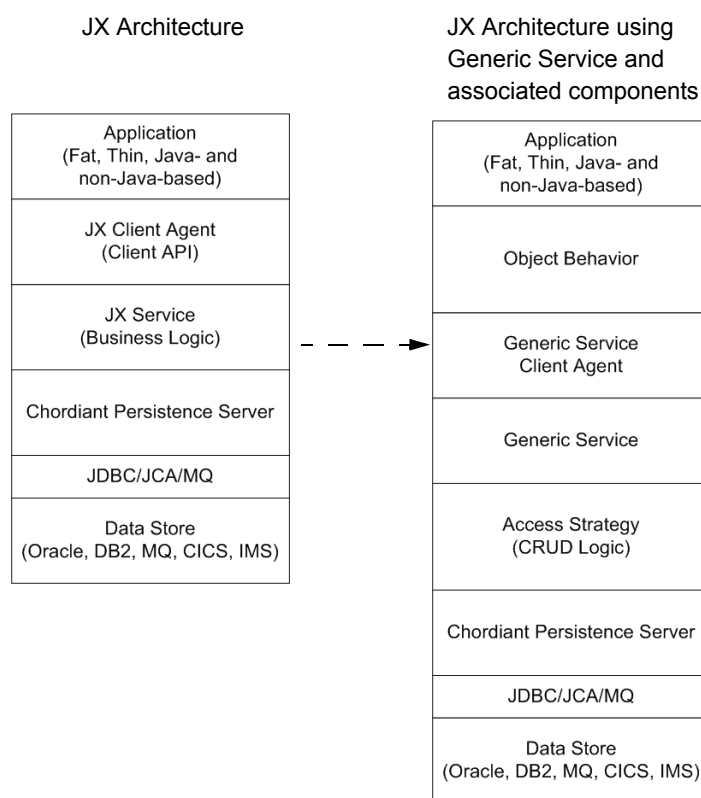


Figure 10-2: Comparison of Architecture
Dotted line shows optional call from JX Service to GS Client Agent for Extended Persistence.

Here, you can see that:

- The Object Behavior provides some functionality before the client ever contacts a service.
- The AccessStrategy contains create, read, update, and delete (CRUD) logic, building on the functionality in the Chordiant Persistence Server. This is why the customization model for the Generic Service includes the JX service. Refer to [“Usage Model for Extended Persistence Components” on page 215](#) for more information.

COMPONENTS GENERATED AUTOMATICALLY

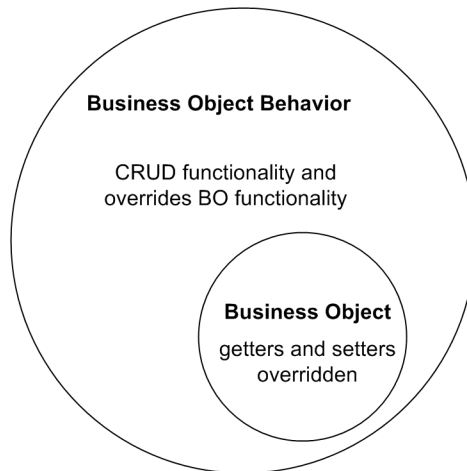
The following components are generated automatically by the Business Component Generator:

- [“Business Object Behavior”](#)
- [“Helpers”](#) (Refer to [page 206](#))
- [“AccessStrategyInput and AccessStrategy”](#) (Refer to [page 207](#))
- [“Validation”](#) (Refer to [page 209](#))

Note: The Generic Service and the Generic Service Client Agent are used exclusively by components generated from the Business Component Generator. We recommend that you refrain from using them directly.

Business Object Behavior

Business Object Behaviors are the extended persistence API and provide several convenience methods and use the Generic Service to fulfill their requests. This saves you from having to develop individual services for these basic tasks and also exposes CRUD functionality to the application layer. You can also use standard business objects with client agents, without using the behaviors.



When you instantiate a business object (BO), you use a factory (the BehaviorFactory) to create an instance of a business object behavior. (For more information, refer to [“Business Object Behavior Factory” on page 211](#).) The instantiated BO is stored inside the behavior. The BO only contains getter and setter methods, but this functionality is taken over by the behavior that is wrapped around it. In addition to the getter and setter methods, the generated behavior includes the following standard methods that it can perform. (Examples are based on the Tutorial Model shown in [Figure 10-3 on page 199](#). Most code examples are based on the `AutomobileBehaviorTest.java` tester, included with this release.)

- **create** — Creates an object in the database and automatically fills in a primary key (usually id), lock token, and type field (if one exists). The method will return `True` if the create was successful.

```
public boolean create() throws Exception
```

For example, you can create a new automobile object. [Code Sample 10-1](#) shows an example of using the `create` method within a tester. The constants are defined earlier in the tester.

```
AutomobileBehavior automobilebehavior =
    (AutomobileBehavior)BehaviorFactory.getBehaviorByObject(
        sUserName, sAuthToken, new Automobile() );
automobilebehavior.setYear( AUTO_YEAR );
automobilebehavior.setMake( AUTO_MAKE );
automobilebehavior.setModel( AUTO_MODEL );
automobilebehavior.setSerialNumber( AUTO_SERIAL_NUMBER );
automobilebehavior.create();
```

Code 10-1: Creating a New Automobile

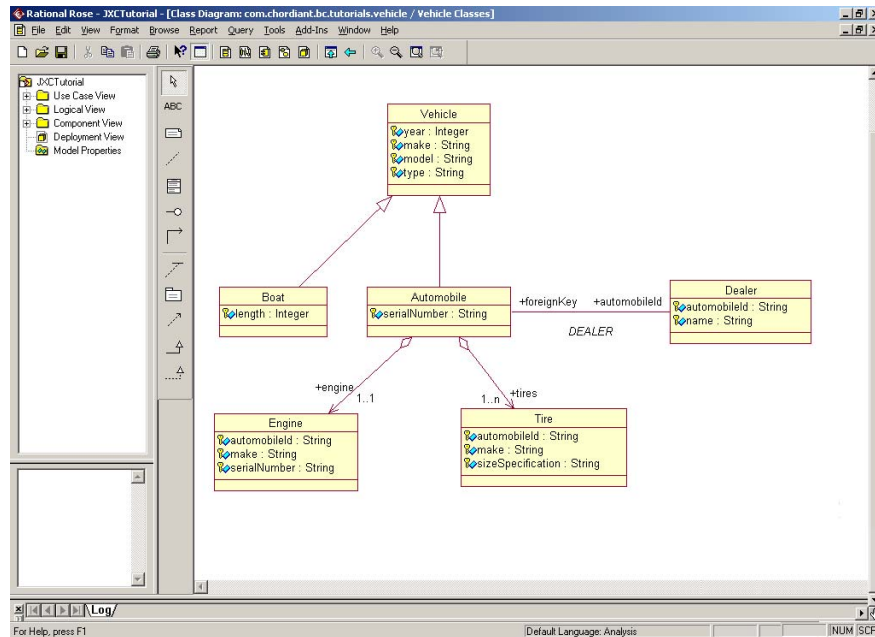


Figure 10-3: Tutorial Model

- **delete** — Deletes a point, that is one specific record, from the database. Deletes the contained object from the database. Returns `true` if the delete was successful and `false` if it was not successful.

```
public boolean delete() throws Exception
```

For example, you can delete a specific automobile object, as shown in [Code Sample 10-2](#).

```
AutomobileBehavior automobilebehavior =
    (AutomobileBehavior)BehaviorFactory.getBehaviorByObject(
        sUserName, sAuthToken, new Automobile() );
automobilebehavior.setId( AUTO_ID );
boolean result = automobilebehavior.delete();
```

Code 10-2: Deleting an Automobile

- **retrieve** — Retrieves a point, that is one specific record, from the database. Replaces the contained object with values from the database. Returns `true` if the retrieval was successful and `false` if it was not successful.

```
public boolean retrieve() throws Exception
```

For example, you can retrieve a specific automobile object, as shown in [Code Sample 10-3](#).

```
AutomobileBehavior automobilebehavior =
    (AutomobileBehavior)BehaviorFactory.getBehaviorByObject(
        sUserName, sAuthToken, new Automobile() );
automobilebehavior.setId( AUTO_ID );
boolean result = automobilebehavior.retrieve();
```

Code 10-3: Retrieving an Automobile

Note: The extended persistence retrieve-type operations are based on the standard Chordiant persistence `retrievePoint` operation. When using retrieve operations, be sure to specify a unique identifier, like the primary key, to specify the object or objects you want to retrieve. If your input is not unique, the business object might map to more than one object in the database. In this case, you will get the following persistence exception:
`UnexpectedMultipleRecordsException` - More than one record returned when only one was expected.

- **retrieveGraph** —Retrieves all of the object’s inheritance and aggregation information. Replaces the contained object with values from the database. Returns `true` if the retrieval was successful and `false` if it was not successful.

```
public boolean retrieveGraph() throws Exception
```

For example, if you are retrieving an automobile object, the detailed information such as engine and tires will also be retrieved, as shown in [Figure 10-3 on page 199](#). [Code Sample 10-4](#) shows the code you would use to find the graph for the contained automobile object.

```
AutomobileBehavior automobilebehavior =
    (AutomobileBehavior)BehaviorFactory.getBehaviorByObject(
        sUserName, sAuthToken, new Automobile() );
automobilebehavior.setId( AUTO_ID );
boolean result = automobilebehavior.retrieve();
```

Code 10-4: Retrieving an Automobile Object

Refer to the note on [page 200](#) about assuring unique retrievals.

- **retrieveGraphToDepth(int)** — Same as `retrieveGraph`, but only retrieves information in the tree to a certain depth, specified by an integer. Depth = 1 means one level below the head object.

Replaces the contained object with values from the database. Returns `true` if the retrieval was successful and `false` if it was not successful.

```
public boolean retrieveGraphToDepth( int depth ) throws Exception
```

In [Figure 10-3](#), using `retrieveGraphToDepth(1)` for a vehicle would retrieve automobile or boat information, but not engine or tire information.

[Code Sample 10-5](#) shows an example of calling the `retrieveGraphToDepth` method on the *vehicle* behavior (not the automobile behavior).

```
AutomobileBehavior automobilebehavior =
    (AutomobileBehavior)BehaviorFactory.getBehaviorByObject(
        sUserName, sAuthToken, new Automobile() );
automobilebehavior.setId( AUTO_ID );
boolean result = vehiclebehavior.retrieveGraphToDepth(1);
```

Code 10-5: Using retrieveGraphToDepth for a Vehicle Behavior

Refer to the note on [page 200](#) about assuring unique retrievals.

- **retrieveAllMatching** — Returns a List of business objects that match the attributes of the contained object. This is similar to retrieving a ray.

```
public List retrieveAllMatching() throws Exception
```

The method call in [Code Sample 10-6](#) will retrieve a List of automobiles matching the attribute of the contained object. Here, we have set the year of the automobile to 1999 and expect to retrieve all automobiles matching that year value of 1999.

```
AutomobileBehavior automobilebehavior =
    (AutomobileBehavior)BehaviorFactory.getBehaviorByObject(
        sUserName, sAuthToken, new Automobile() );
automobilebehavior.setYear(1999);
List result = automobilebehavior.retrieveAllMatching()
```

Code 10-6: Retrieving a List of Automobiles

Refer to the note on [page 200](#) about assuring unique retrievals.

- **retrieveAllMatchingGraphs** — Returns a List of graphs for the **retrieveAllMatching** function described above. Will provide a list of all information—no depth is specified.

```
public List retrieveAllMatchingGraphs() throws Exception
```

Calling the **retrieveAllMatchingGraphs** is similar to calling the **retrieveAllMatching** method described above. However, a list of graphs of information is returned. So in addition to the list of all of the cars returned in **retrieveAllMatching** above, this method would also return lists of their engines and tires will also be returned.

Refer to the note on [page 200](#) about assuring unique retrievals.

- **retrieveAssociation(String)** — Returns associated class information. Returns a List when passed the name of the association, as defined in the object model.

```
public List retrieveAssociation( String associationName )
    throws Exception
```

For example, if passed “Dealer” as shown in [Code Sample 10-7](#), a list of dealers for this automobile will be returned. (Note that in this model, each automobile only has one dealer, so the list would only contain one element. If there were another association defined for Driver, each vehicle could potentially have a list of drivers associated with it.).

```
AutomobileBehavior automobilebehavior =
    (AutomobileBehavior)BehaviorFactory.getBehaviorByObject(
        sUserName, sAuthToken, new Automobile() );
automobilebehavior.setId( AUTO_ID );

// Retrieve this object from the database
boolean result = automobilebehavior.retrieve();

// Find associated dealer
    List associatedDealer =
automobilebehavior.retrieveAssociation("DEALER");
```

Code 10-7: Retrieving an Associated Dealer

Refer to the note on [page 200](#) about assuring unique retrievals.

There are some methods on the Helper class which can assist you in finding the names of the associations. For more information, refer to “[Helpers](#)” on [page 206](#).

- **retrieveAssociation** — Returns all associated class information for all associations. Since no argument is passed, returns a List of Lists for associated class information (a two-dimensional List).

```
public List retrieveAssociation() throws Exception
```

For example, in this model, the method call in [Code Sample 10-8](#) would return a list of all Dealers. If there were also an association of Drivers, then all list of Drivers would also be returned.

```
AutomobileBehavior automobilebehavior =
    (AutomobileBehavior)BehaviorFactory.getBehaviorByObject(
        sUserName, sAuthToken, new Automobile() );
automobilebehavior.setId( AUTO_ID );
List associatedClasses =
    automobilebehavior.retrieveAssociation()
```

Code 10-8: Retrieving a List of Dealers

Refer to the note on [page 200](#) about assuring unique retrievals.

- **retrieveAssociationGraphs(String)** — Returns a List of graphs for the retrieveAssociation function described above. Will provide all information—no depth is specified.

```
public List retrieveAssociation( String associationName )
throws Exception
```

For example, in addition to the list of Dealers returned from the retrieveAssociation method above, all of the Dealer information in the Dealer graph, like contact information, would also be provided.

```
AutomobileBehavior automobilebehavior =
    (AutomobileBehavior)BehaviorFactory.getBehaviorByObject(
        sUserName, sAuthToken, new Automobile() );
automobilebehavior.setId( AUTO_ID );

// Retrieve this object from the database
boolean result = automobilebehavior.retrieve();

// Find associated dealer
List associatedDealer =
    automobilebehavior.retrieveAssociationGraphs("DEALER");
```

Code 10-9: Retrieving the GGraph of Associated Dealers

Refer to the note on [page 200](#) about assuring unique retrievals.

- **update** — Updates any information that has changed since you last retrieved the object. Locking is based on the locking strategy you specified for the object.

```
public boolean update() throws Exception
```

For example, if a model name has changed, you can update this information, as shown in [Code Sample 10-10](#).

```
AutomobileBehavior automobilebehavior =
    (AutomobileBehavior)BehaviorFactory.getBehaviorByObject(
        sUserName, sAuthToken, new Automobile() );
automobilebehavior.setId( AUTO_ID );

// Retrieve this object from the database
boolean result = automobilebehavior.retrieve();

// update object just retrieved
automobilebehavior.setModel("MK II");
result = automobilebehavior.update();
```

Code 10-10: Updating the Automobile Behavior

- **updateGraph** — Updates every object in the graph for this object. Locking is based on the locking strategy you specified for the object. For example, in addition to updating the Dealer's name, you might also need to update its contact information.

If an object has a non-null id field, updateGraph treats it as an existing object and updates the object in the database. If it has a null id field, updateGraph treats it as a new object and inserts the object into the database as a new record. The feature makes it easier to save an object graph containing both new and changed objects.

```
public boolean updateGraph() throws Exception
```

Calling `updateGraph` is similar to the call for the basic `update` method. Refer to the `update` method call in [Code Sample 10-10](#).

- **toString** — Prints out the graph of the object contained in the wrapper. This is mostly used for testing.

```
public String toString()
```

For example, you can print out all of the information about a specific vehicle.

```
AutomobileBehavior automobilebehavior =
    (AutomobileBehavior)BehaviorFactory.getBehaviorByObject(
        sUserName, sAuthToken, new Automobile() );
automobilebehavior.setId( AUTO_ID );
String output = automobilebehavior.toString();
```

Code 10-11: Printing Vehicle Information

- **toFieldValuePairs** — Converts the contained value into a Hashtable representation of the field-value pairs, where the Hashtable key is the attribute name and the Hashtable value is the attribute value. This can be helpful in creating GUI screens with tables of values

```
public Hashtable toFieldValuePairs() throws Exception
```

[Code Sample 10-12](#) provides an example of using the `toFieldValuePairs` method.

```
AutomobileBehavior automobilebehavior =
    (AutomobileBehavior)BehaviorFactory.getBehaviorByObject(
        sUserName, sAuthToken, new Automobile() );
automobilebehavior.setId( AUTO_ID );
Hashtable vehicleValues =
    automobilebehavior.toFieldValuePairs();
```

Code 10-12: Using toFieldValuePairs

Behavior Base Class

All behaviors must subclass from the Behavior Base Class. The Behavior Base Class contains all of the methods described on [page 198](#), which are included in all generated behaviors, as well as those listed below. The following methods of the Behavior Base Class are used to actually encapsulate the business object within the behavior.

- **setContainedObject(CorporateBusinessClass)** — Changes the internally wrapped business object to be the one that you pass in this method. You can set the contained object to be a specific automobile.

```
public void setContainedObject( CorporateBusinessClass bo )
```

- **setUserName** — Sets the `userName` attribute of the business object contained within the behavior. `UserName` and `AuthenticationToken` are used together for security throughout the JX system.

```
public void setUsername( String username )
```

- **setAuthentication** — Sets the authentication token of the business object contained within the behavior. Used with `userName` (see above).

```
public void setAuthentication( String authenticationToken )
```


- **setSecurityInformation** — Sets the security information attribute of the business object contained within the behavior. This is for additional security, should you require it.

```
public void setSecurityInformation(
    ISecurityInformation securityInformation )
```

- **setGenericServiceInstance** — Sets the instance of the generic service, if this behavior was instantiated by the generic service. This way, if the Generic Service will be calling itself through an recursive process, it will use the same instance as it has been using and not jump between different instances of the Generic Service. This improves performance.

```
public void setGenericServiceInstance(
    ServiceBaseClass genericServiceInstance )
```

- **setId** — Sets the ID of the business object contained within the behavior.

```
public void setId( java.lang.String value )
```

- **setToBeDeleted** — Exposes a Boolean flag in the CorporateBusinessClass to the behavior. Enables you to set the flag for whether a method in the business class will be deleted. This flag is inherited from the CorporateBusinessClass base class.

```
public void setToBeDeleted( java.lang.Boolean value )
```

- **getContainedObject** — Returns the contained business object so you can work with it directly. Returns the vehicle object so you can perform operations on it.

```
public CorporateBusinessClass getContainedObject()
```

- **getUsername** — Gets the username attribute of the BehaviorBaseClass object. This can be used for security.

```
public String getUsername()
```

- **getAuthentication** — Gets the authentication attribute of the BehaviorBaseClass object. This can be used for security.

```
public String getAuthentication()
```

- **getSecurity** — Gets the security information attribute of the BehaviorBaseClass object. This is additional security, should you require it.

```
public ISecurityInformation getSecurityInformation()
```

- **getId** — Gets the id attribute of the BehaviorBaseClass object.

```
public java.lang.String getId()
```

- **getGenericServiceInstance** — Gets the generic service instance, if this behavior was instantiated by the generic service. This enables you to continue using the same instance of the Generic Service for recursive calls.

```
public ServiceBaseClass getGenericServiceInstance()
```

- **getToBeDeleted** — Gets the toBeDeleted attribute of the BehaviorBaseClass object.

```
public java.lang.Boolean getToBeDeleted()
```

- **toString(prefix)** — Prints out the graph of the object contained in the wrapper, attaching the specified prefix. For example, you can print out all of the information about a specific vehicle with the String “AUTO-” as a prefix.

```
public String toString( String prefix )
```

- **setAdditionalData(Object additionalData)** — Sets any additional data, not in the object, This enables you to pass more data to the back end, should you require it.

```
public void setAdditionalData( Object additionalData )
```

- **getAdditionalData(Object additionalData)** — Retrieves any additional data that is not in the object.

```
public Object getAdditionalData()
```

Behavior Base Class Exceptions

In the behavior base class, the standard data access methods (Create, Read, Update, Delete) pass an `UnsupportedOperationException`. These methods must be overridden by the subclass or the `UnsupportedOperationException` will be thrown.

Helpers

Helpers are created automatically through the Business Component Generator. Helpers provide several convenience methods and use the Generic Service to fulfill their requests. These helper methods can assist you in calling behavior object methods. For example, you can find the association types for a business object, then use that information to find association information for that business object. You can also use standard business objects with client agents, without using the helpers.

Examples in this section refer to “[Tutorial Model](#)”, shown in [Figure 10-3 on page 199](#).

The methods included on the helper are:

- **getType**—Returns the `typeValue` for this object. If the object is an automobile, the `VehicleHelper` will return `AUTOMOBILE`. Refer to “[Inheritance Tags](#)” on [page 224](#) for more information on `typeValue`. This method only exists if you have specified the `typeValue` in the class.

```
public static String getType()
```

- **lookupClassByType(String)**—Looks through all of the descendant classes for the passed `typeValue`. Returns the object for the `typeValue` specified. For example, the `VehicleHelper` will return the `Automobile` class when passed the `Automobile typeValue`.

```
public static CorporateBusinessClass lookupClassByType(String typeName)
```

- **lookupAllDescendantClasses**—Returns a List of business objects for all descendants of this class in the inheritance hierarchy, including the parent. For example, for the `Vehicle` class, this method will return a list including `boat` and `automobile`.

```
public static List lookupAllDescendantClasses()
```

- **lookupAllDescendantCriteriaClasses**—Returns a List of business object criteria (BOCs) for all descendent classes. For example, for the **Vehicle** class, this method will return all BOCs for boat and automobile.

```
public static List lookupAllDescendantCriteriaClasses()
```

- **retrieveAssociationTypes**—Returns a List of Strings with names of all of the association types for this business object. For example, for the vehicle, this method would retrieve a List including “DEALER”, since that is the name of an association with this class. If there were another association, like “DRIVER”, that would also be included in this list.

```
public static List retrieveAssociationTypes() throws Exception
```

- **getAssociationClassName(String)**—When passed an association name, returns the fully-qualified class name for the associated class. For example, when passed “DEALER” as the association, would return the class name for the dealer class.

```
public static String getAssociationClassName(String association) throws Exception
```

Note: There are additional methods within the helper base class. These are either used internally or are deprecated, so you should not use them.

If you want to create additional helper methods, you can extend the generated Helper object. For example, a method `retrieveAllCustomersByLastName(String)` could easily be added to a subclass of `CustomerHelper`, which could in turn forward the request to the generated `retrieveAllMatching` method.

AccessStrategyInput and AccessStrategy

The access strategy encapsulates the way the data will be manipulated at the lower level. Each generated business object has a corresponding generated access strategy. This access strategy contains the logic for all of the generated operations corresponding to the business object (that is, all CRUD operations).

Access strategies correspond to the standard persistence data accessors, but allow for a richer set of CRUD operations, including loading or updating a hierarchical tree of objects.

Access strategies only use the optimistic locking strategy.

There is a default access strategy that is generated automatically. If you do not want to use the default access strategy, you can specify a different access strategy value in your Rational Rose model. Refer to [“Access Strategy” on page 221](#) for details. You might want to extend an access strategy to connect to an external system and check for security before returning any data to users, or to perform operations other than CRUD operations, including implementing stored procedures.

The class that you specify for the access strategy must implement the `IAccessStrategy` interface. As with all Chordiant customization, we recommend that you extend the generated `AccessStrategy` class and override the desired methods. The generated `AccessStrategy` automatically implements `IAccessStrategy`. `IAccessStrategy` is located in `com.chordiant.bc.interfaces`.

[Code Sample 10-13](#) shows part of the `extendedAutomobileAccessStrategy` used in the Tutorial.

```
public Object execute(IAccessStrategyInput in) throws Exception
{
    String METHOD_NAME = "execute";
    LogHelper.methodEntry( PACKAGE_NAME, CLASS_NAME, METHOD_NAME );

    if (!( in instanceof GeneratedAccessStrategyInputBase ))
    {
        LogHelper.error( PACKAGE_NAME, CLASS_NAME, METHOD_NAME, "AccessStrategyInput passed to
            AutomobileAccessStrategy does not inherit from GeneratedAccessStrategyInputBase" );
        throw new InvalidParameterException( "in", in, IBusinessServiceErrors.BS_RUNTIME_ERROR );
    }

    // Downcast the input so we can use the desired get methods
    GeneratedAccessStrategyInputBase input = (GeneratedAccessStrategyInputBase)in;

    Automobile bo = (Automobile)input.getContainedBo();

    if ( bo == null ) {
        LogHelper.error( PACKAGE_NAME, CLASS_NAME, METHOD_NAME, "Error: passed business object is null in
            AutomobileAccessStrategyInput" );
        return null;
    }

    String username = input.getUsername();
    String authentication = input.getAuthentication();
    ISecurityInformation securityInformation = input.getSecurityInformation();
    Object additionalData = input.getAdditionalData();

    int operation = input.getOperation();
    Object result = null;

    // Using the locally-defined retrieve() call defined below
    if ( operation==GeneratedAccessStrategyBase.RETRIEVE_OPERATION ) {
        if (ValidatorFactory.getValidatorByObject( bo ).retrieveValidate( bo )) {
            result = retrieve( username, authentication, securityInformation, additionalData, bo );
        }
    }
    // Using the default accessStrategy
    else {
        super.execute( in );
    }

    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return result;
}

protected Object retrieve(String username, String authentication, ISecurityInformation securityInformation,
    Object additionalData, Automobile bo) throws Exception
{
    String METHOD_NAME = "retrieve";
    LogHelper.methodEntry( PACKAGE_NAME, CLASS_NAME, METHOD_NAME );
```

Code 10-13: Section of the `extendedAutomobileAccessStrategy`

```

Automobile result = (Automobile)super.retrieve( username, authentication, securityInformation,
    additionalData, bo);

if ( ( result != null ) && ( result.getSerialNumber() != null ) ){
    // Pre-pend the string AUTO- to the front of the serial number
    result.setSerialNumber( "AUTO-" + result.getSerialNumber() );
}

LogHelper.methodExit( PACKAGE_NAME, CLASS_NAME, METHOD_NAME );
return result;
}
}

```

Code 10-13: Section of the *extendedAutomobileAccessStrategy* (Continued)

The *AccessStrategyInput* is instantiated by a Business Object Behavior. It is passed in the Generic Service Client Agent payload through to the Generic Service. Once there, the Generic Service uses the *AccessStrategyInput* to look up the corresponding access strategy.

The access strategy implements logic to fulfill requests from the client. When it is called from the Generic Service, the access strategy is executed to work with the Chordiant Persistence Server to access the data store.

Validation

You can use validators to specify the type of field-level validation that is required for an attribute of a business object. Chordiant provides a framework for you to create your own validation. By default, all provided validators return *true*, meaning that any attribute value will be considered correct.

The Validator Factory (*com.chordiant.bc.factories.ValidatorFactory.java*) is usually called both in the behavior and in the access strategy. The Validator Factory calls are redundant in these two places to ensure that data is validated. Calls to the factory from the behavior check the data as soon as possible, without having to hit the service layer. Since it is possible to call the Generic Service without using a behavior, Validator Factory calls are duplicated in the access strategy. So the access strategy performs the validation that might have been skipped by bypassing the behavior.

The Validator Factory uses the *ValidatorFactoryConfiguration.xml* file to decide which validator class should be used for a specific business object. The *ValidatorFactoryConfiguration.xml* file is generated automatically from your object model through the Business Component Generator.

The Validation Factory contains these two APIs:

- **getValidatorByObject**

```
public static IValidator getValidatorByObject(CorporateBusinessClass data )
```

- **getValidatorByObjectName**

```
public static IValidator getValidatorByObjectName( String className )
    throws Exception
```

If you require a different validator for a specific class, you can make this change in the validation configuration file. You can make changes to this configuration file at runtime. For the tag representing the business object, provide a different value for the desired validator. You can also add tag/value pairs.

To update the Validator Factory with the new settings, use the Administrative Console to first refresh the ConfigurationHelper, then refresh the Generic Service. For information on using the Administrative Console, refer to “Using the Administrative Console” in the *Chordiant 5 Foundation Server Developer’s Guide*.

Code Sample 10-14 shows a portion of the `ValidatorFactoryConfiguration.xml` file for the tutorial model.

```
<Root>

  <Section>ValidatorFactoryConfiguration

    <Tag>com.chordiant.bc.tutorials.vehicle.Vehicle
      <Value>com.chordiant.bc.tutorials.vehicle.validators. VehicleValidator</Value>
    </Tag>
    <Tag>com.chordiant.bc.tutorials.vehicle.Automobile
      <Value>com.chordiant.bc.tutorials.vehicle.validators.AutomobileValidator</Value>
    </Tag>
    <Tag>com.chordiant.bc.tutorials.vehicle.Engine
      <Value>com.chordiant.bc.tutorials.vehicle.validators.EngineValidator</Value>
    </Tag>

  </Section>
</Root>
```

Code 10-14: Portion of the `ValidatorFactoryConfiguration.xml` Configuration File

To create your own validator, implement the `IValidator.java` interface. **Code Sample 10-15** shows an example of an `automobileValidator`, which implements `IValidator`. This is just a portion of the code showing the validation to perform before a create operation is performed. There are validator sections for retrieves, updates, and other operations.

```
package com.chordiant.bc.tutorials.vehicle.validators;
...
public class AutomobileValidator implements IValidator
{
    public static final String PACKAGE_NAME = "com.chordiant.bc.tutorials.vehicle.validators";
    public static final String CLASS_NAME = "AutomobileValidator";

    /**
     * The validation to be performed before performing a create operation
     */
}
```

Code 10-15: Portion of `automobileValidator` Code

```

    *@param bo    The business object to be validated
    *@return      The result of the validation. True if successful
    *@exception   Exception If an internal exception, or a ValidationException
    */
    public boolean createValidate( CorporateBusinessClass bo ) throws Exception
    {
        final String METHOD_NAME = "createValidate";
        String tmpResult;
        ValidationException e = new ValidationException();
        boolean errors = false;

        if (errors) {
            throw e;
        }
        return true;
    }

```

Code 10-15: Portion of automobileValidator Code

BUSINESS OBJECT BEHAVIOR FACTORY

The Behavior Factory (`com.chordiant.bc.factories.behaviorfactory.java`) returns a business object behavior to wrap around a business object that you give it.

The Behavior Factory uses the `BehaviorFactoryConfiguration.xml` file to decide which behavior to instantiate for which business object. The `BehaviorFactoryConfiguration.xml` file is generated automatically from your object model through the Business Component Generator.

If you require a different business object behavior from the Behavior Factory, you can make this change in the behavior factory configuration file. Refer to [“Changing the Behavior Factory Configuration” on page 213](#) for more information.

The Behavior Factory has six interfaces—three each for specifying an object or for specifying the name of the object. The APIs are:

- **getBehaviorByObject** (passing an instance of the object, including passing a new instance)

- Basic

```
public static BehaviorBaseClass getBehaviorByObject(
    String username, String authentication, CorporateBusinessClass data )
```

[Code Sample 10-16](#) provides an example of using this method.

```
bo = new Automobile();
automobilebehavior = (AutomobileBehavior)
    BehaviorFactory.getBehaviorByObject(sUserName, sAuthToken, bo)
```

Code 10-16: Using getBehaviorByObject

Note that you must pass a valid user name and authentication token.

- With increased security functionality

```
public static BehaviorBaseClass getBehaviorByObject(
    String username, String authentication, ISecurityInformation
    securityInformation, CorporateBusinessClass data )
```

Note: ISecurityInformation is a placeholder for your own security object, specific to your own back end system.

- For retrieving a graph or some other functionality having to do with inheritance. Keeps recursive calls to the same generic service instance within the same EJB instance. This is for performance.

```
public static BehaviorBaseClass getBehaviorByObject(
    String username, String authentication,
    ISecurityInformation securityInformation,
    ServiceBaseClass genericServiceInstance,
    CorporateBusinessClass data )
```

- **getBehaviorByObjectName** (passing the fully-qualified name of the Object class, including the package, as a String)

- Basic

```
public static BehaviorBaseClass getBehaviorByObjectName( String username,
    String authentication, String className )
```

[Code Sample 10-17](#) provides an example of using this method.

```
automobilebehavior = (AutomobileBehavior) BehaviorFactory.getBehaviorByName(
    sUserName, sAuthToken, "com.chordiant.bc.tutorials.vehicle.Automobile")
```

Code 10-17: Using getBehaviorByObjectName

- With increased security functionality

```
public static BehaviorBaseClass getBehaviorByObjectName(
    String username, String authentication,
    ISecurityInformation securityInformation, String className )
```


- For retrieving a graph or some other functionality having to do with inheritance. Keeps recursive calls to the same generic service instance within the same EJB instance. This is for performance.

```
public static BehaviorBaseClass getBehaviorByObjectName(
    String username, String authentication,
    ISecurityInformation securityInformation,
    ServiceBaseClass genericServiceInstance, String className )
```

For more information on the Behavior Factory interfaces, refer to the Javadoc. Javadoc is installed on your computer through the installation under `{WORKSPACE}/documentation/FoundationServer/Javadoc`.

Changing the Behavior Factory Configuration

As described above, the Behavior Factory uses the `BehaviorFactoryConfiguration.xml` file to decide which type of behavior to instantiate for a given business object.

Code Sample 10-18 shows an example of a `BehaviorFactoryConfiguration.xml` file.

```
<Root>
  <Section>BehaviorFactoryConfiguration
    <Tag>com.chordiant.poc.businessclasses.Address
      <Value>com.chordiant.poc.businessclasses.behaviors.AddressBehavior</Value>
    </Tag>
    <Tag>com.chordiant.poc.businessclasses.Person
      <Value>com.chordiant.poc.businessclasses.behaviors.PersonBehavior</Value>
    </Tag>
    <Tag>com.chordiant.poc.businessclasses.ContactInfo
      <Value>com.chordiant.poc.businessclasses.behaviors.ContactInfoBehavior</Value>
    </Tag>
  </Section>
</Root>
```

Code 10-18: BehaviorFactoryConfiguration.xml Configuration File

You can change which behavior is used within Rational Rose. Within the object class specification, make changes on the **JXC CMI** tab. Refer to “[useBehavior](#)” on [page 223](#) for details.

LOADING CLASSES DYNAMICALLY

The `DynamicLoaderComponent.xml` enables you to load classes in your development environment dynamically, without recycling your application server. The `DynamicLoaderComponent.xml` is a configuration file in which you specify the location of your class files in your development environment.

There are two tags in this file:

- **runmode**—By default, the runmode is **Production**.

Note: The Dynamic Class Loader is turned off in **Production** mode. If the value of the tag is **Production**, the Dynamic Class Loader will not be used.

- **url**—The location of your classes to be loaded dynamically. You can specify more than one url by duplicating the tag and providing a different value.

Note: These classes should not be on the runtime classpath.

Code Sample 10-19 shows a sample version of the `DynamicLoaderComponent.xml` file. In this sample, the files are located in the generated directory, since they are created by the Business Component Generator.

```
<Section>DynamicLoaderConfiguration
  <Tag>runmode
    <Value>Development</Value>
  </Tag>
  <Tag>url
    <Value>file:///E:/dynamicClasses/bin/generated/</Value>
  </Tag>
  <Tag>url
    <Value>file:///G:/dynamicClasses/bin/generated/</Value>
  </Tag>
</Section>
```

Code 10-19: DynamicLoaderComponent.xml File

This configuration file is used by the `DynamicLoader`. The `DynamicLoader` enables you to load classes into the application server without having to package them as JARs and without having to recycle the application server.

To add classes dynamically, make sure you have specified your settings in the `DynamicLoaderComponent.xml` file.

Use the JX Administrative Console to refresh the `ConfigurationHelper`, then refresh the `Generic Service`. When you refresh the `Generic Service`, the classes in the specified directory will be loaded. For details on using the Administrative Console, refer to the *Chordiant 5 Foundation Server Developer's Guide*.

USAGE MODEL FOR EXTENDED PERSISTENCE COMPONENTS

As their name implies, the extended persistence components should be considered an *extension* of the standard persistence components that are a part of the Chordiant 5 Foundation Server. They are intended to be used in place of the standard persistence components when your implementation requires more than basic create, read, update, delete (CRUD) operations. Do not have your application interface with the extended persistence components, including the Generic Service and the Generic Service Client Agent, directly.

The Chordiant usage model for the extended persistence components prescribes using these components only in conjunction with standard business services for the following reasons:

- Standard services can control a broader scope than the Generic Service and associated components. They handle business logic.
- Standard services and client agents, rather than the generic service, should be tightly tied to your application. The generic service is designed to handle data accessing and not complex business logic and so should not be coupled with the front end.
- Standard services can coordinate transactions that span multiple domain objects.
- Standard services have caching facilities not available to the Generic Service.¹

In this model, you can have a business service (refer to [Chapter 4, “Creating or Modifying Business Services”](#)) make a call to the Generic Service Client Agent. The Generic Service Client Agent then calls the Generic Service which works with the Access Strategies to access the data stores through the Data Accessor. This is the same data accessor included in the standard Chordiant persistence.

1. All Chordiant-provided services extend from the `BusinessDataServiceBaseClass`, which has an internal cache. This cache is used by all Chordiant services in the same JVM.

As shown in [Figure 10-4](#), from your business service, you can choose to use the standard persistence functionality, when you have simple operations with single business objects, or you can use the extended persistence functionality when you are handling graphs of business objects and potentially more complex data operations.

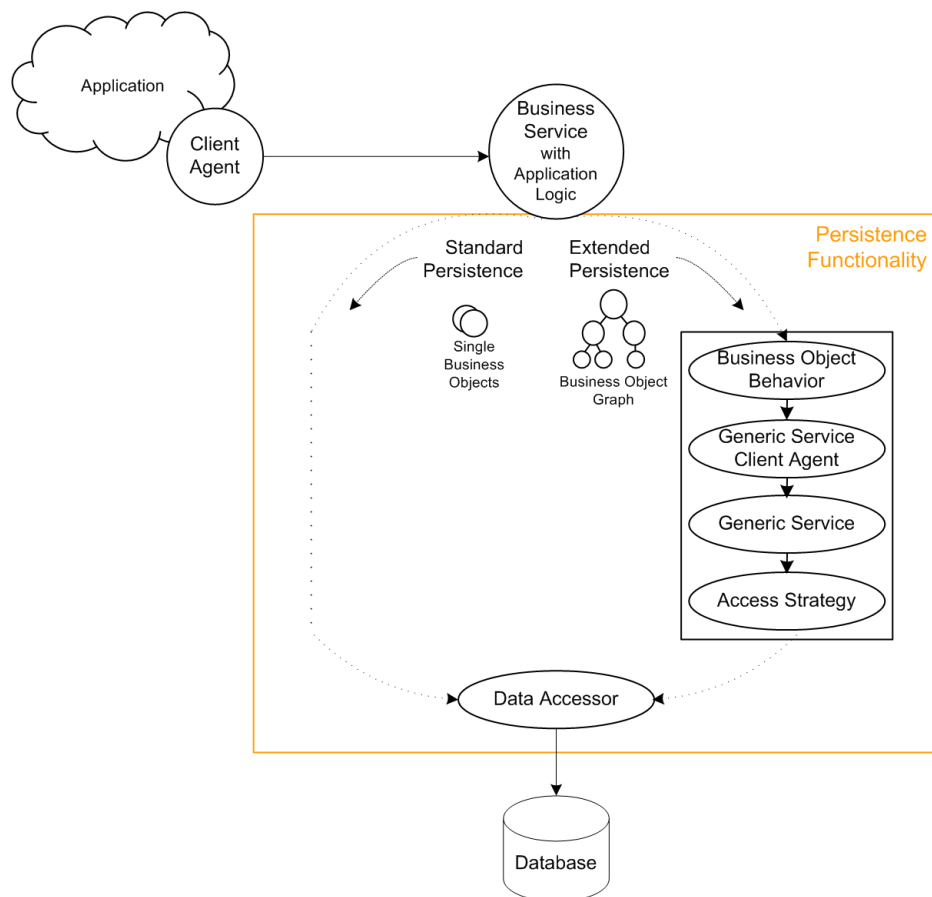


Figure 10-4: Overview of Business Service - Persistence Interaction

The comparison between the Generic Service option and the Business Service façade option is analogous to Entity Bean and Session Bean. Entity Bean offers object-centric CRUD-like APIs to manipulate data. Session Bean offers business-centric APIs to conduct business operations. J2EE best practice recommends a Session Façade pattern for a similar reason:

“Use a session bean as a façade to encapsulate the complexity of interactions between the business objects participating in a workflow. The session façade manages the business objects, and provides a uniform, coarse-grained service access layer to clients.”

Specifying Extended Persistence

If you require additional data manipulation over and above the standard create, read, update, delete (CRUD) interface offered by the standard Chordiant Persistence Server, you can take advantage of the extended persistence functionality. This functionality sits on top of the standard Chordiant Persistence Server and makes use of several business components, including the business object behavior, helper classes, and access strategies. These components are generated automatically by the Business Component Generator.

Notes: This chapter describes customization procedures for *new* business services. Existing Chordiant business services, described in [Chapter 16](#) of this book, do not use the tags described in this chapter.

The Chordiant Data Model does not support Extended Persistence Components.

USING EXTENDED PERSISTENCE COMPONENTS

It is important to remember that these behaviors are only associated with one business object — the business object they encapsulate. This enables you to quickly create business components to perform simple interactions with the database.

However, your deployment requires the use of more complete standard services to control these components. Standard services encompass more than just one aspect of your domain and include caching not available to the generic service. Refer to [“Usage Model for Extended Persistence Components” on page 215](#) for details.

Note: The Generic Service and the Generic Service Client Agent are used exclusively by components generated from the Business Component Generator. Do not expose them directly to the client-facing application. Use them only in conjunction with standard services.

EXTENDED PERSISTENCE COMPONENTS CREATED

You can use the Business Component Generator to create extended persistence components. These components are discussed in [Chapter 10, “Extended Persistence Components”](#).

All of these components are fully formed and do not require additional implementation.

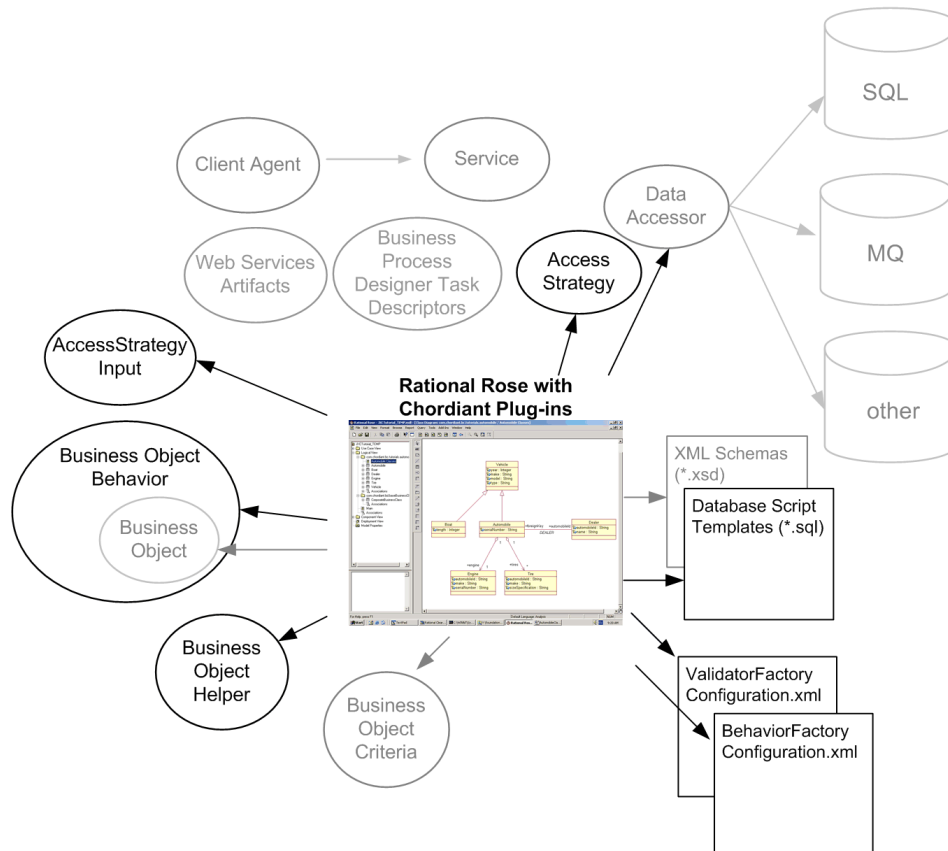


Figure 11-1: Business Component and Persistence Components Created Automatically
Persistence components created in [Chapter 9](#) and Service files created in [Chapter 4](#) are shown in gray.

- **Business Object Behavior** — `{yourobject}Behavior.java`
The Business Object Behavior contains methods which act on the contained Business Object. This is the extended persistence API.
- **Business Object Helper** — `{yourobject}Helper.java`
The Business Object Helper contains additional methods which act on the Business Object.
- **Access Strategy** — `{yourobject}AccessStrategy.java`
The Access Strategy is used by the Generic Business Service to interact with backend datastore through the Data Accessor.
- **Access Strategy Input** — `{yourobject}AccessStrategyInput.java`
The Access Strategy Input is used by the Generic Business Service Client Agent to contact the Generic Business Service, telling it which work needs to be done.
- **Factory Configuration Files**
 - `Behavior FactoryBehaviorFactoryConfiguration.xml`
Tells the Behavior Factory which behavior to supply when a behavior is requested.
 - `Validator Factory ValidatorFactoryConfiguration.xml`
Tells the system which validator to supply from the factory.
- **Database Script Templates** — `{yourobject}.sql`
SQL script templates you can use to run against your database to create schema definitions based on the model you created. Note that these templates require some modification before they can be used.

SPECIFYING EXTENDED PERSISTENCE INFORMATION

Note: Before you can specify these tags, you must configure the system. Refer to [Chapter 2, “Setting Up Your System for Component Generation”](#) for information on configuring your system to generate application components.

All of these tags are specified at the class level. The generated XMI file is shown for each tag described below.

- [“Access Strategy” on page 221](#)
- [“useBehavior” on page 223](#)
- [“Inheritance Tags” on page 224](#)
 - [“xrefTableName” on page 228](#)
 - [“typeValue” on page 229](#)
 - [“typeField” on page 230](#)
- [“Associations” on page 232](#)
 - [“Association Type” on page 232](#)
 - [“Aggregations” on page 234](#)
 - [“Multiple Associations” on page 237](#)

The customizations in this chapter are based on the example model shown in [Figure 11-2](#).

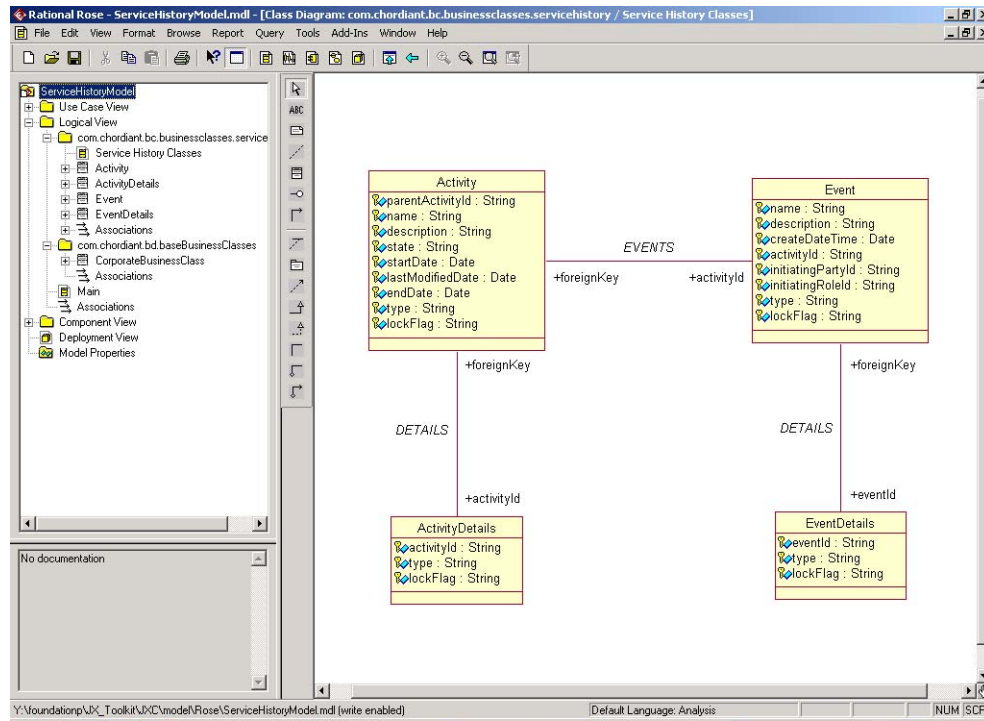


Figure 11-2: Example Model for Customization

Access Strategy

The access strategy contains the logic for all of the generated operations corresponding to the business object (that is, all create, read, update, and delete operations).

There is a default access strategy value that is assigned automatically. If you do not want to use the default access strategy, you can specify a different access strategy value in your Rational Rose model.

To specify a different access strategy:

1. Within Rose, double-click on your class to open the **Class Specifications** window. In the **Class Specifications** window, select the **AccessStrategy** tab.
2. Find the name of the method for which you want to change the access strategy. Then click in the Value column next to that method name. In the box that appears, type the new, fully-qualified class name for the access strategy.

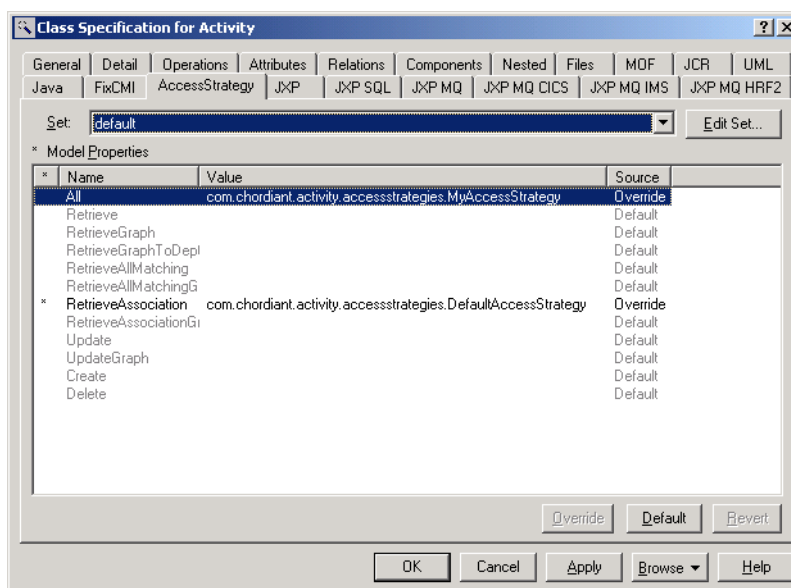


Figure 11-3: Overriding Access Strategy

You can override the access strategy value for one or more individual methods. These methods include:

- All
- Create
- Retrieve
- Update
- Delete
- RetrieveGraph
- RetrieveGraphToDepth
- UpdateGraph
- RetrieveAllMatching
- RetrieveAllMatchingGraphs
- RetrieveAssociation
- RetrieveAssociationGraphs

You can also override the top value of All, which will override all of the other methods underneath. You can override the value for All, but then also specify a different override for methods underneath (as shown in Figure 11-3). In this way, you can make most of the operations use the custom access strategy, but then change just a few of them back to use the default strategy.

3. When you have specified all of the access strategies, click **OK**.

After you run the Business Component Generator, the corresponding section of the CMI file will look like this. There can be one or more **accessStrategy** sections which contain one or more **operation** sections. In [Code Sample 11-1](#), there are two **accessStrategy** sections with one operation each. Each operation section lists the name of the method for which you specified a non-default access strategy. In this example, the first value is **All**, specifying that all operations should use the given strategy. However, one operation (**RetrieveAssociation**) does not use this override, so we must specify the class path of the default access strategy for that operation.

```
<root>
  <class>
    ...
    <accessStrategy>
      <operation>
        <name>All</name>
        <useStrategy>com.chordiant.activity.accessstrategies.MyAccessStrategy</useStrategy>
      </operation>
    </accessStrategy>
    <accessStrategy>
      <operation>
        <name>RetrieveAssociation</name>
        <useStrategy>com.chordiant.activity.accessstrategies.DefaultAccessStrategy</useStrategy>
      </operation>
    </accessStrategy>
    ...
  </class>
</root>
```

Code 11-1: accessStrategy Sections of Generated CMI File

useBehavior

When you instantiate a business object, the behavior factory automatically creates a business object behavior that wraps around the instantiated business object. Instead of using the default behavior created by the behavior factory, you can ask for your own customized behavior to be created. The behaviors of subclasses of this class will also inherit from this custom behavior.

As with other Chordiant customization, we recommend that you extend from the generated behavior class, then make your own customizations.

To specify a different behavior:

1. Within Rose, double-click on your class to open the **Class Specifications** window. In the **Class Specifications** window, select the **JXC CMI** tab.
2. Locate the `useBehavior` row and click in the **Value** column. In the box that appears, type the fully-qualified name of the class for the behavior you want to use.

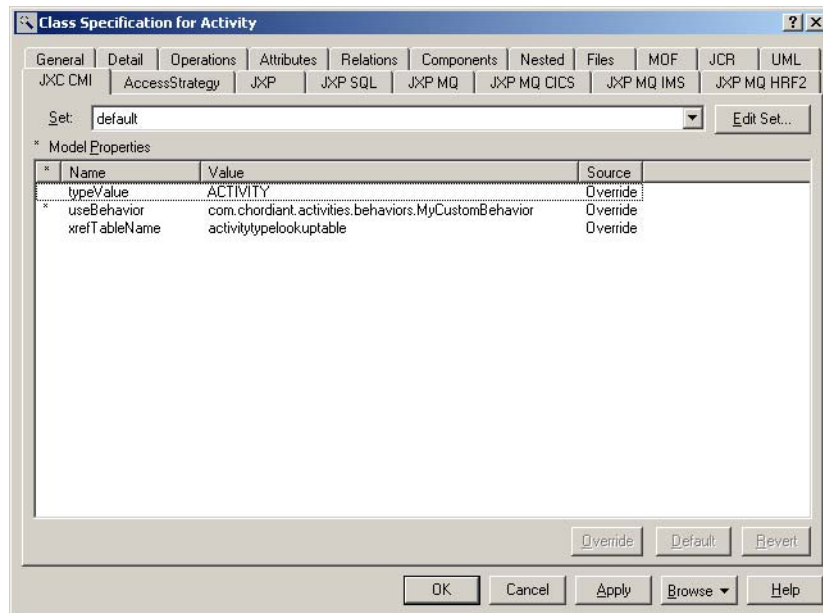


Figure 11-4: Overriding Behavior

3. Click **OK**.

After you run the Business Component Generator, the corresponding section of the CMI file will look like this. Notice the specified behavior to use is listed within the `name` tag within the `useBehavior` tags.

```
<useBehavior>
  <name>com.chordiant.activities.behaviors.MyCustomBehavior</name>
</useBehavior>
```

The Behavior Factory configuration file is also updated, so the Behavior Factory will return the customized behavior class you specify here.

Inheritance Tags

These three tags are used when you have inheritance in your model:

- `xrefTableName`
- `typeValue`
- `typeField`

Specifying Type to Aid Lookup Efficiency

In the Chordiant architecture, each subclass in an object inheritance tree can have its own database table, which includes data from the parent class. Alternatively, you can use a single database table whose columns are the union of the columns that would exist in separate child tables and the parent table.

These three inheritance tags, often used in combination, help to retrieve data for the subclasses in the most efficient manner.

In each class, you choose one attribute to be the `typeField`. This attribute is often called `type`. (This concept is similar to specifying an attribute in an object as the `lockField`.) This `type` attribute has another property called `typeValue`. This specifies the kind of data that is included in the table for that class.

Note: You must specify the `typeField` and `typeValue` in your model if you want to use the parent class behavior to retrieve data from a child table.

In the tutorial in [Chapter 12](#), the `Vehicle` class is a parent class for both `Boats` and `Automobiles`. An attribute in each of the classes, called `type`, is specified as the `typeField`. The `typeField` property is set to `true`. In the `Boat` class, the `typeValue` of the `type` attribute is set to `BOAT`. In the `Vehicle` class, the `typeValue` set to `VEHICLE`, and in the `Automobile` class, the `typeValue` set to `AUTOMOBILE`.

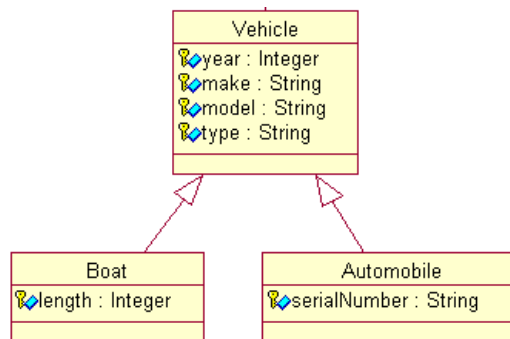


Figure 11-5: Tutorial Model

For these three classes, you could create the four tables shown in [Figure 11-6](#), including the xref_Table.

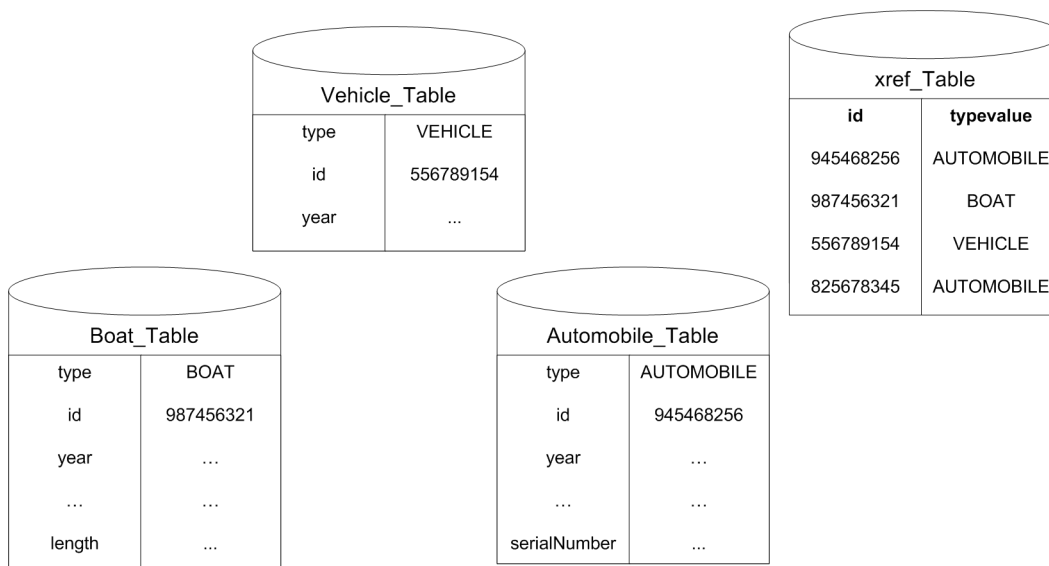


Figure 11-6: Simplified Diagram of Relevant Tables in the Tutorial Model

Alternatively, you can have one large table, including all of the columns from the separate child table, as shown in [Figure 11-7](#).

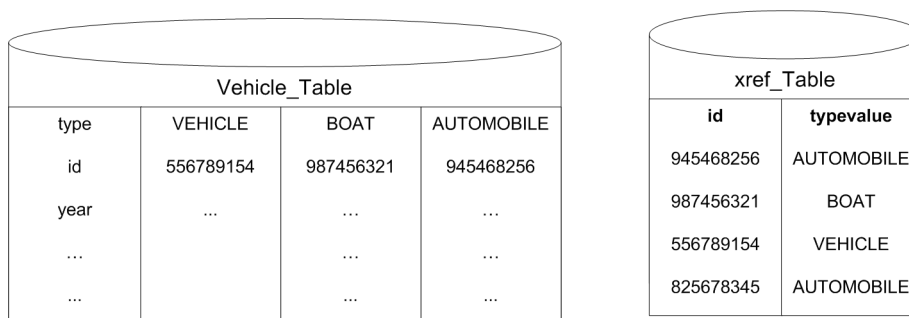


Figure 11-7: Simplified Diagram of Alternative Database Structure for Tutorial Model

Note: The following discussion pertains to the scenario shown in [Figure 11-6](#).

Possible data retrieval scenarios in order of *decreasing* efficiency:

1. In your `retrieve` method, if you set the `type` field to a valid `typeValue`, you will look straight into the appropriate table. This is the most direct approach. The implementation will use the `typeValue` information to load data from the correct table. Refer to [Code Sample 11-2](#) for an example of setting the `type` field.
2. There are two separate scenarios for this step, depending on whether you are using the `retrieve` or `retrieveAllMatching` call.

- a. Using the `retrieve` method:

When using the `retrieve` method of the `VehicleBehavior`, if you do not set the `type` field to a valid `typeValue`, the `retrieve` method will look in the `xrefTable`. The `xrefTable` shows which tables deal with which IDs. Methods can look up the specific ID in the `xrefTable` to find the appropriate data table.

Note: The `xrefTable` is mandatory if you want to use the parent class behavior to retrieve data from a child table.

- b. Using the `retrieveAllMatching` method:

When using the `retrieveAllMatching` method of the `VehicleBehavior`, if you do not set the `type` field to a valid `typeValue`, the method will have to look in all tables to find the desired values. This is the least efficient strategy.

[Code Sample 11-2](#) shows how using the `type` speeds up the retrieval process by avoiding the `xrefTable`.

```
public void retrieve() {
    try {

        AutomobileBehavior automobilebehavior = (AutomobileBehavior)BehaviorFactory.
            getBehaviorByObject( sUserName, sAuthToken, new Automobile() );
        automobilebehavior.setId( AUTO_ID );

        // setting type speeds up retrieval, avoids accessing xrefTable
        automobilebehavior.setType(AutomobileHelper.getType());

        // Retrieve this object from the database
        boolean result = automobilebehavior.retrieve();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Code 11-2: Modified Retrieve Method

xrefTableName

The **xrefTable** is an id-type lookup table that is used to look up the type of an object at runtime. This is important for objects that are part of an inheritance hierarchy. Since each class usually has its own table, this lookup table can save a lot of time. If you want to retrieve information for a specific ID, the **xrefTable** shows which specific table has the information you are looking for.

When you populate a table using the **create** call in the Behavior class, information is automatically added to the **xrefTable** for easier retrieval.

Note: You must specify an **xrefTable** if you want to use the parent class behavior to retrieve data from a child table.

To specify the xrefTableName:

1. Within Rose, double-click on your class to open the **Class Specifications** window. In the **Class Specifications** window, select the **JXC CMI** tab.
2. Locate the **xrefTableName** row and click in the **Value** column. In the box that appears, type the name of the **xrefTable** you want to use.
3. Click **OK**.

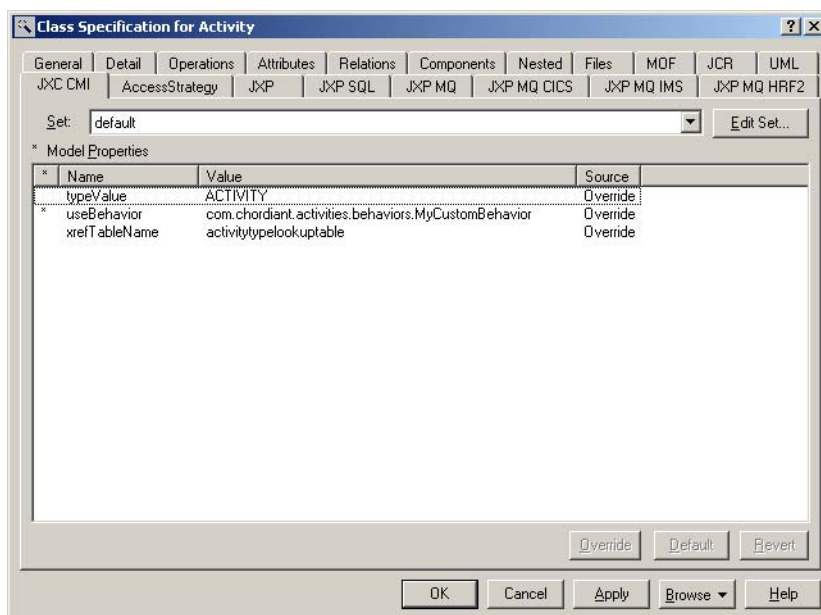


Figure 11-8: Specifying the xrefTableName

After you run the Business Component Generator, the corresponding section of the CMI file will look like this.

```
<xrefTableName>activitytypelookuptable</xrefTableName>
```


4. Within the Business Component Generator, be sure to generate the script for creating the `xrefTable` in the database.

typeValue

When you have a subclass, it is important to specify the `typeValue`. The `typeValue` is a unique identifying token, usually the name of the class typed in all capital letters. This allows the implementation to find which table contains the information for this subclass, so it is easier and faster to find the data. This value will be returned when the `getType` method is called (see [page 206](#)).

Note: You must specify a `typeValue` and `typeField` (refer to “`typeValue`” and “`typeField`” on [page 230](#)) if you want to use the parent class behavior to retrieve data from a child table.

To specify a different `typeValue`:

1. Within Rose, double-click on your class to open the **Class Specifications** window. In the **Class Specifications** window, select the **JXC CMI** tab.
2. Locate the `typeValue` row and click in the `Value` column. In the box that appears, type the new `typeValue` you want to use. For the Activity class, shown in the example in [Figure 11-9](#), the `typeValue` is set to `ACTIVITY`.

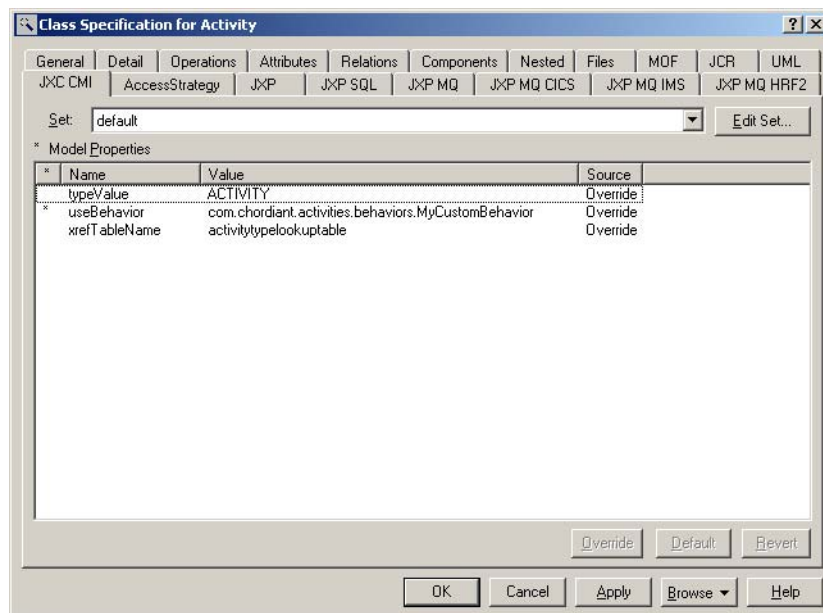


Figure 11-9: Overriding `typeValue`

3. Click **OK**.

After you run the Business Component Generator, the corresponding section of the CMI file will look like this:

```
<typeValue>ACTIVITY</typeValue>
```

typeField

One attribute in the base class of the inheritance hierarchy must be designated as the **typeField**. The **typeField** for an attribute must be set to **true**, so the attribute will be used for inheritance support. This concept is similar to specifying a **lockField** for object locking.

Note: You must specify a **typeField** and **typeValue** (refer to “**typeValue**” and “**typeField**” on page 230) if you want to use the parent class behavior to retrieve data from a child table.

To set the **typeField** to **true**:

1. In your model, locate or create the attribute you want to indicate the type. Usually, you will create a String attribute called **Type**.
2. Double-click the attribute to open the **Class Attribute Specification** window. In the **Class Attribute Specification** window, select the **JXC Properties** tab.
3. Locate the **typeField** row and click in the Value column. In the menu that appears, select **True**.
4. Click **OK**.

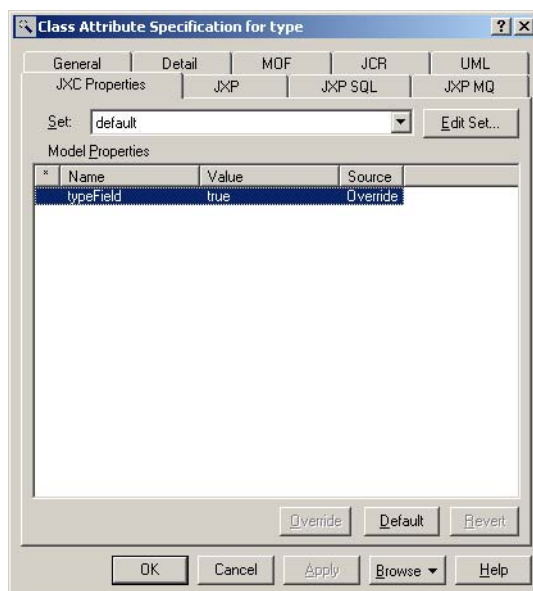


Figure 11-10: Specifying the Attribute's **typeField** as **True**

After you run the Business Component Generator, the corresponding section of the CMI file will look like this.

```
<attribute>
  <name>type</name>
  ....
  <typeField>true</typeField>
</attribute>
```

CMI Example

[Code Sample 11-3](#) shows a section of the CMI file that shows the tags discussed up to this point.

For a full description of all CMI tags, refer to [Chapter 15, “Metadata”](#).

```
<class>
  <name>Activity</name>
  <javaDoc>/**</javaDoc>
  <parentClass>com.chordiant.bd.baseBusinessClasses.CorporateBusinessClass</parentClass>
  <isAbstract>false</isAbstract>
  <rdbPhysicalName>activity</rdbPhysicalName>
  <WherePrefix></WherePrefix>
  <DSN>chordiantXAds</DSN>
  <override></override>
  <persistentType>oracle8</persistentType>
  <LockStrategy>optimistic</LockStrategy>
  <typeValue>ACTIVITY</typeValue>
  <typeField>true</typeField>
  <useBehavior>
    <name>com.chordiant.activity.behaviors.MyCustomBehavior</name>
  </useBehavior>
  <xrefTableName>activitytypelookuptable</xrefTableName>
  <accessStrategy>
    <operation>
      <name>All</name>
      <useStrategy>com.chordiant.activity.accessstrategies.MyAccessStrategy</useStrategy>
    </operation>
  </accessStrategy>
  <accessStrategy>
    <operation>
      <name>RetrieveAssociation</name>
      <useStrategy>com.chordiant.activity.accessstrategies.DefaultAccessStrategy</useStrategy>
    </operation>
  </accessStrategy>
  <association>
    <name>DETAILS</name>
    <type>foreignKey</type>
    <className>com.chordiant.delivery.businessclasses.servicehistory.ActivityDetails</className>
    <foreignKeyAttribute>activityId</foreignKeyAttribute>
  </association>
```

Code 11-3: Section of Generated CMI File

Associations

Association Type

When you associate one class with another, for example associating details with an activity, you must define the association type in your model.

To define the association type in Rational Rose:

1. Create a new class.
2. Click the association tool and draw an association from the main class to the associated class.
3. Right-click on the association line and select **Specifications**.

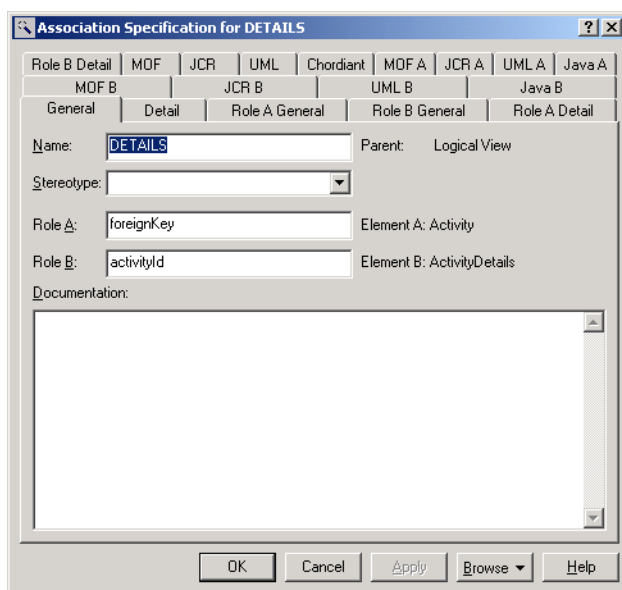


Figure 11-11: Defining Association Specifications

4. Type a name for the association. This name is used on a call to the `Helper.retrieveAssociation` method to retrieve the associated object. In Figure 11-11, the association name is Details.
5. For Role A, type `foreignKey` to designate that there is a foreign key association.
6. For Role B, type the name of the foreign key from the associated class. In this case, it is `activityId`.
7. Click **OK**.

The model will now show the association line between the two classes, the name of the association, as well as the foreign key relationship and the name of the foreign key. In [Figure 11-12](#), the association goes from the `Activity` to `ActivityDetails`.

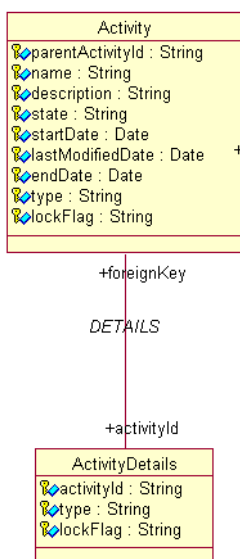


Figure 11-12: Model Showing Association

Note: You must specify a name for the association, otherwise it can become an attribute of the associated class, which is not desirable. The Business Component Generator automatically makes unnamed associations attributes of the associated class. So if you forget to name the association, you might see some unexpected results.

After you run the Business Component Generator, the corresponding section of the CMI file will look like this.

```

<association>
  <name>DETAILS</name>
  <type>foreignKey</type>
  <className>com.chordiant.delivery.businessclasses.servicehistory.ActivityDetails</className>
  <foreignKeyAttribute>activityId</foreignKeyAttribute>
</association>
  
```

Code 11-4: Section of Generated CMI File

Aggregations

The `associationType`, `classAttribute`, and `foreignClassAttribute` tags are used together for contained associated objects, or aggregations. This enables you to retrieve complex objects or Vectors of objects on a call to the `retrieveGraph` method.

In this sample model, the `ActivityDetails` object contains `MyCustomObject`.

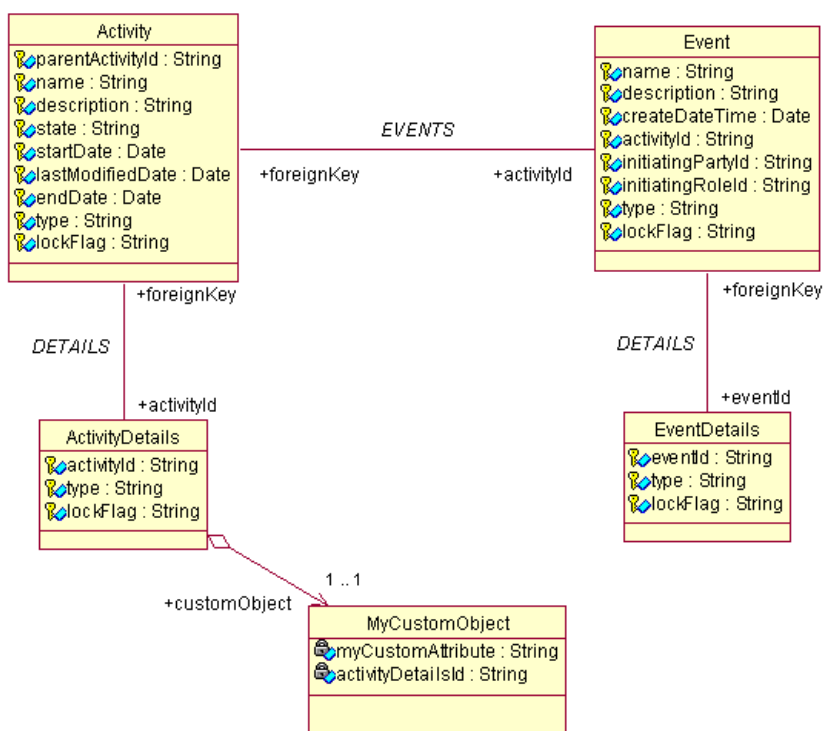


Figure 11-13: Model with Contained Associated Objects

When a single associated objects is contained in an object, the retrieval of the contained object graph is done using the foreign keys.

- The `foreignClassAttribute` tag specifies the remote (associated) object to view.
- The `classAttribute` tag defines the attribute in the container class.

To specify **foreignClassAttribute** and **classAttribute**:

1. In your Rose model, create a new class. In this example, it is called `MyCustomObject`.
2. Draw an aggregation from `ActivityDetails` to `MyCustomObject`. Direction is important.

3. Right-click on the aggregation line and select **Specifications**.

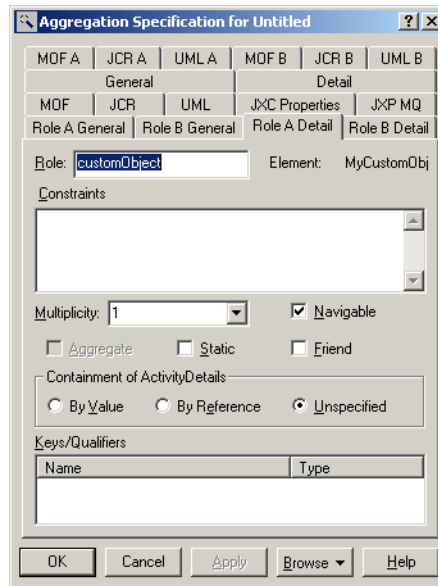


Figure 11-14: Specifying the Role and Multiplicity for a Contained Object

4. In the **Aggregation Specifications** window, select the **Role A Detail** tab. In the **Role** box, type the name of the attribute that corresponds to the contained object. Here, it is `customObject`.
5. In the **Multiplicity** box, type `1..1`, since this is a one-to-one relationship. Click **Apply**.
6. Select the **JXC Properties** tab.

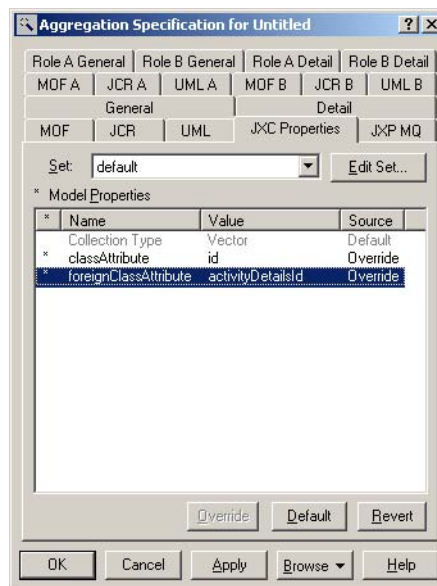


Figure 11-15: Specifying classAttribute and foreignClassAttribute for Contained Object

7. Locate the **classAttribute** row and click in the **Value** column. In the box that appears, type the value of the **classAttribute**. In this case, that value is **id**.
8. Click **Apply**.
9. Locate the **foreignClassAttribute** row and click in the **Value** column. In the box that appears, type the value of the **foreignClassAttribute**. In this case, that value is **activityDetailsId**.
10. Click **OK**.

Notes: The aggregation will automatically become an attribute of the associated class. Do not specify a name for it.

Notice there is no collection type for this attribute, since it is a one-to-one association.

After you run the Business Component Generator, the corresponding section of the CMI file will look like [Code Sample 11-5](#)

```
<class>
  <name>ActivityDetails</name>
  ...
  <attribute>
    <name>customObject</name>
    <javaType>com.chordiant.bc.businessclasses.servicehistory.MyCustomObject</javaType>
    <multiplicity>1..1</multiplicity>
    <classAttribute>id</classAttribute>
    <foreignClassAttribute>activityDetailsId</foreignClassAttribute>
    <javaDoc>/**/</javaDoc>
  </attribute>
</class>
```

Code 11-5: Section of Generated CMI File

For a full description of all CMI tags, refer to [Chapter 15, “Metadata”](#).

Note that you must still define the persistence information for the **MyCustomObject** class. Refer to [“Creating or Modifying Business Objects” on page 163](#) for information on specifying persistence metadata.

Multiple Associations

If the associated object contains a Vector of other objects, you must specify additional information for the retrieveGraph method. In the model shown in Figure 11-16, there is a 1..n relationship between Details and myCustomObjects.

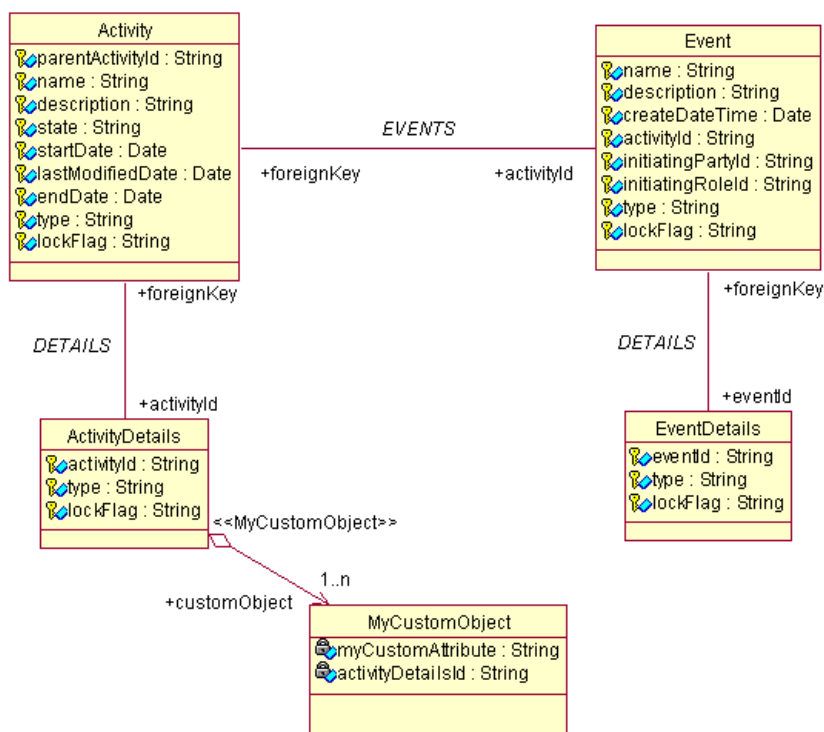


Figure 11-16: Associated Object with Contained Vector

1. In your Rose model, create a new class. In this example, it is called **MyCustomObject**. (This is identical to the class you created in [Step 1 on page 234](#).)
2. Draw an aggregation from **ActivityDetails** to **MyCustomObject**. Direction is important. (This is identical to the class you created in [Step 2 on page 234](#).)

3. Right-click on the aggregation line and select **Specifications**.

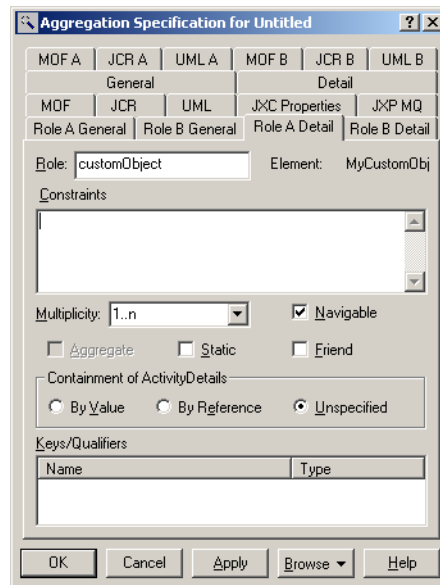


Figure 11-17: Specifying Multiplicity for a Contained Vector

4. In the **Aggregation Specifications** window, select the **Role A Detail** tab. In the **Role** box, type the name of the attribute that corresponds to the contained object. Here, it is `customObject`.
5. In the **Multiplicity** box, select **1..n**, since this is a one-to-many relationship. Click **Apply**.

6. Select the **JXC Properties** tab.

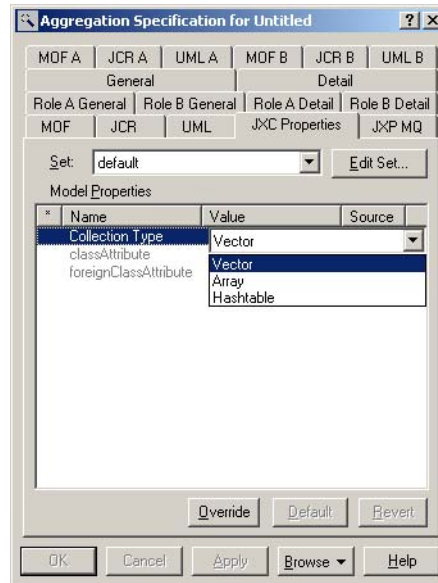


Figure 11-18: Specifying the Association for a Contained Vector

7. Locate the **Collection Type** row and click in the **Value** column. In the menu, select **Vector**. You could also select either array or hash table, depending on what you are adding to the class.

Note: Step 8 - Step 11 below are identical to Step 7 - Step 10 on page 236 for specifying a contained object.

8. Locate the **classAttribute** row and click in the **Value** column. In the box that appears, type the value of the classAttribute. In this case, that value is **id**.
9. Click **Apply**.
10. Locate the **foreignClassAttribute** row and click in the **Value** column. In the box that appears, type the value of the foreignClassAttribute. In this case, that value is **activityDetailsId**.
11. Click **OK**.

12. Select the **General** tab.

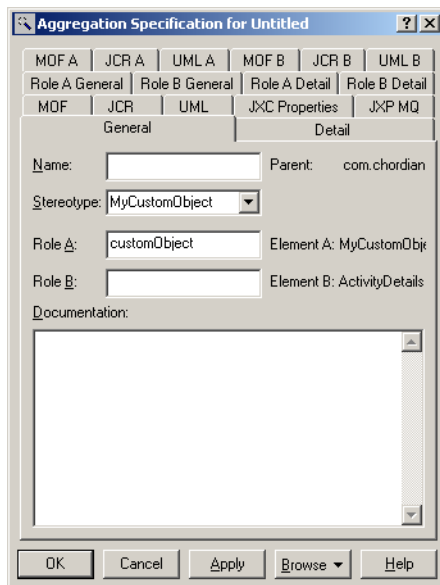


Figure 11-19: Specifying the Stereotype for a Contained Vector

13. In the **Stereotype** box, select the name of the contained class, **MyCustomObject**. This tag translates to the **associationType** tag in the CMI file.
14. The **Role A** box should already be populated with the name of the attribute which will contain the Vector. You already specified this information in [Step 4 on page 238](#).

[Code Sample 11-6](#) shows the corresponding section of the CMI file, after you run the Business Component Generator.

```
<class>
  <name>ActivityDetails</name>
  ...
    <attribute>
      <name>customObject</name>
      <jakartaType>java.util.Vector</jakartaType>
      <associationType>com.chordiant.bc.businessclasses.servicehistory.
        MyCustomObject</associationType>
      <multiplicity>1..*</multiplicity>
      <classAttribute>id</classAttribute>
      <foreignClassAttribute>activityDetailsId</foreignClassAttribute>
      <jakartaDoc>/**</jakartaDoc>
    </attribute>
  </class>
```

Code 11-6: Section of Generated CMI File

For a full description of all CMI tags, refer to [Chapter 15, “Metadata”](#).

ADDING DOCUMENTATION

If you want to annotate the code for these Java classes you are creating, you must put it in a specific location. The documentation you add will appear as Javadoc in the Java code that is generated.

Notes: Do not use the **Javadoc** tab for either class or attributes.

Do not type “/” or “*/” tags. Your comments will appear as Javadoc in the generated code.

You can use HTML tags in your documentation if you want. Refer to the figures in this section for examples.

To add documentation in your Rational Rose model:

For a class:

1. Double click on the class you want to document to open the **Class Specification** window.
2. Enter documentation for the class in the **Documentation** box as shown in [Figure 11-20](#).

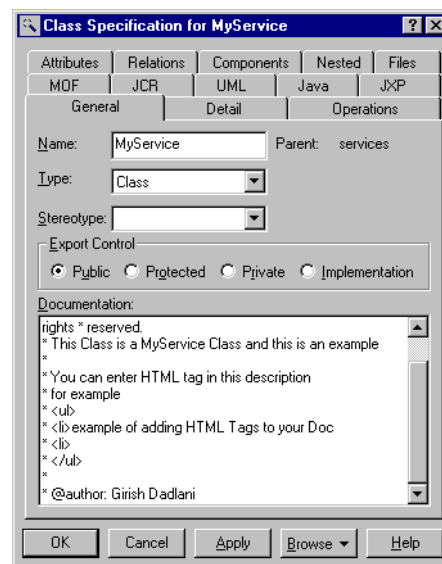


Figure 11-20: Adding Description for a Class

For an attribute:

1. Right-click on the attribute you want to document. Select **Open Specification**.
2. Enter documentation for the attribute in the **Documentation** box, as shown in [Figure 11-21](#).

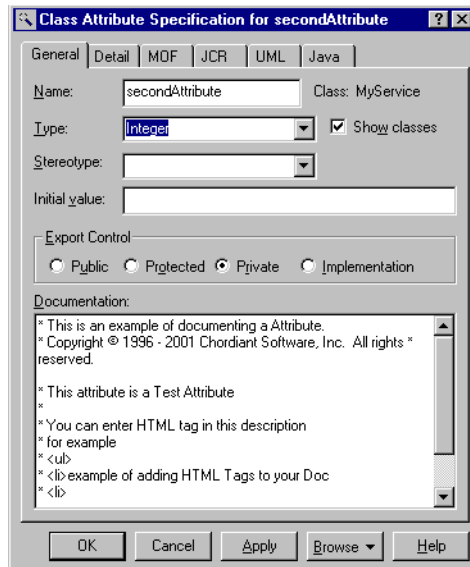


Figure 11-21: Adding Description for an Attribute

Tutorial: Extended Persistence Components

This tutorial is designed to walk you through a simple model using inheritance, containment, and association through the extended persistence components. Customization points will be described within each of the sections of this chapter.

The model created in this tutorial is available in its final form in `{eclipse_root}/plugins/{data_model_plugin}/rosemodels/JXCTutorial.mdl`. You can also find it through the Business Component Generator.

You can refer to it while you step through the tutorial.

This tutorial assumes that you have a working knowledge of object modeling using Rational Rose.

Note: This tutorial describes customization procedures for *new* business services. Existing Chordiant business services, described in [Chapter 16](#) of this book, do not use the tags described in this tutorial.

Refer to “[Usage Model for Extended Persistence Components](#)” on page 215 for details on when and how to use extended persistence components.

THE TUTORIAL MODEL

Before you begin, you might want to open up the finished tutorial model to see what it looks like.

Figure 12-1 illustrates the final model.

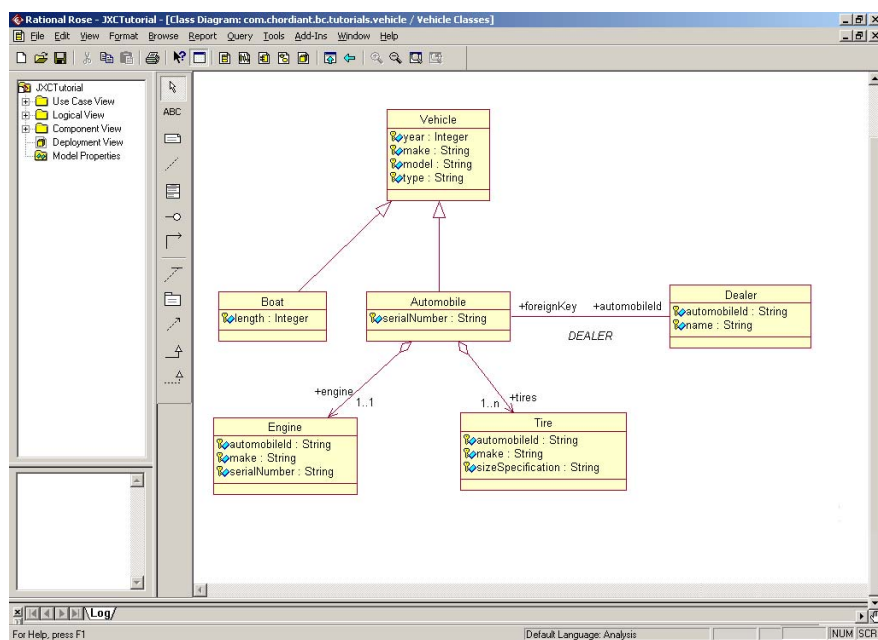


Figure 12-1: Vehicles: A Sample Customized Model

This model shows three main concepts. Each of these concepts is discussed in specific sections of the tutorial.

- **Inheritance**—The Automobile class is a subclass of the Vehicle class. Refer to the next section, [“Inheritance”](#).
- **Containment**—The Automobile class contains the Engine class and the Tire class. Refer to [“Containment” on page 255](#).
- **Association**—The Automobile class is associated with the Dealer class. Refer to [“Associations” on page 261](#).

Notes: Early steps in the tutorial affect later steps. You will likely want to follow these procedures from start to finish.

It is a good idea to generate your code frequently as you progress through the tutorial. This can help you find any mistakes you might have made more easily than if you wait until the end.

Inheritance

The Chordiant Corporate Business Class is at the base of this model. The Vehicle class subclasses from the base class, then the Automobile class subclasses from the Vehicle.

The tags that are pertinent to inheritance in this model are:

- typeField
- typeValue
- xrfefitable

Creating a Model and the Base Class

1. Create a new model in Rational Rose.
2. Open Chordiant's base object model, JXB0Base.mdl.

This model is located in

`{eclipse_root}/plugins/{data_model_plugin}/rosemodels.`

Note: You can open the model from anywhere or create a new project through the Business Component Generator, using this model. Then you can customize the model from within your project.

3. Create a new package for this tutorial, called `com.chordiant.bc.tutorials.vehicle`.
4. Create a new class diagram in this package called **Vehicle Classes**.
5. Open the **Vehicle Classes** class diagram.
6. Drag the **CorporateBusinessClass** into the **Vehicle Classes** class diagram.

Notice that in the diagram, Rose shows that the base class is from a different package, specifically `com.chordiant.bd.basebusinessclasses`.

7. In the base class, add a new protected String attribute called `id`.
8. Open the specifications for the base class.
9. In the **Class Specifications** window, select the **Attribute** tab. Double-click on `id` to see the **Class Attribute Specification for id**.

10. In the **Class Attribute Specifications for id** window, select the **JXP SQL** tab, as illustrated in Figure 12-2.

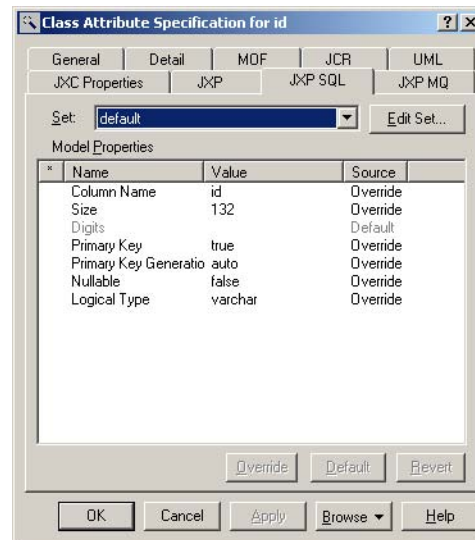


Figure 12-2: Specifying JXP SQL Information for id Attribute

11. On the **JXP SQL** tab, click in the Value column next to the specifications listed here and type the values:
 - Column Name = id
 - Size = 132
 - Primary Key = true
 - Primary Key Generation = auto
 - Nullable = false
 - Logical Type = varchar
12. Click **OK** to return to the **Vehicle Classes** view.

Adding a New Class, Subclassing from Base Class

13. Add a new class called Vehicle.
14. Draw a generalization arrow from vehicle to the base class.

15. Create these four new protected attributes:

- year: Integer
- make: String
- model: String
- type: String

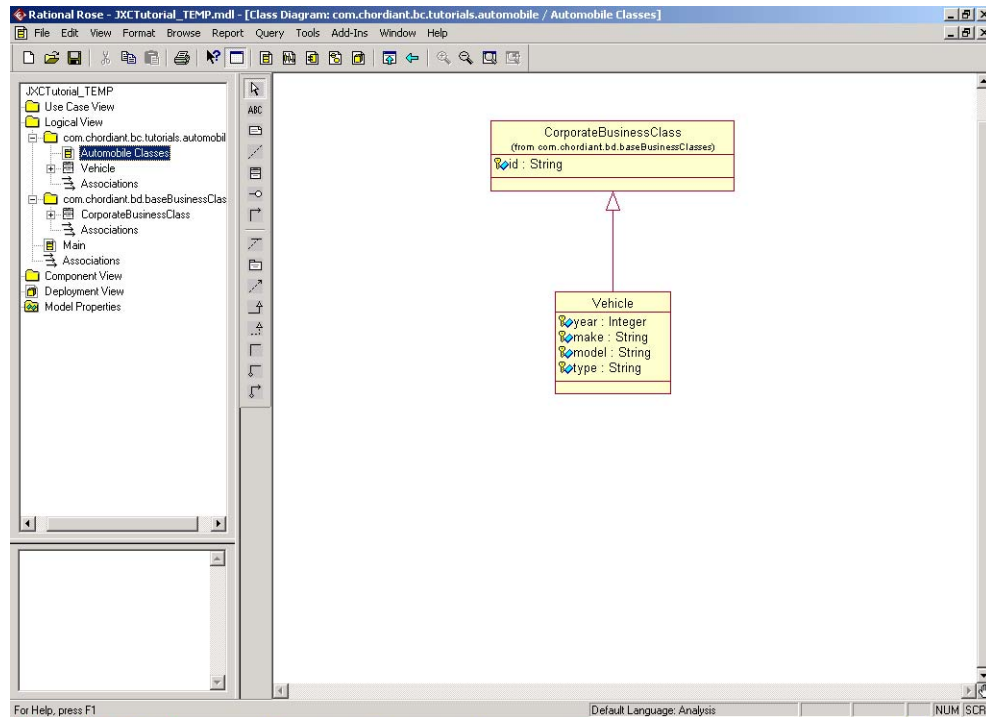


Figure 12-3: New Vehicle Class Created

16. Open the **Class Specifications for Vehicle** and select the **JXP** tab.

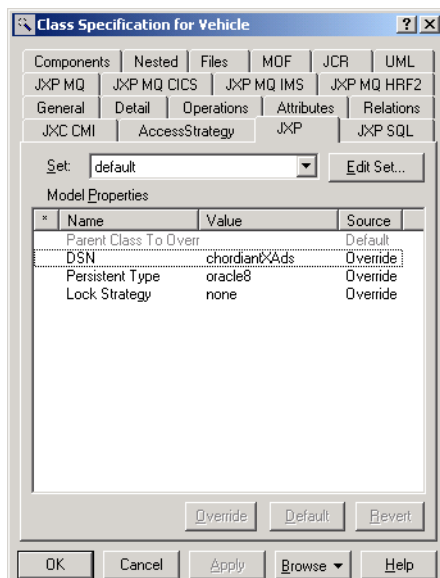


Figure 12-4: Specifying JXP Information for the Vehicle Class

17. Click in the Value column associated with each of these three properties and type in the white box to override the specifications:
- DSN = chordiantXAds
 - Persistent Type = oracle8
 - Lock Strategy = none

Note: The **DSN** and **Persistent Type** are environment-specific and can be different, depending on your setup. The **DSN** value must match the data source resource name specified in your service configuration file, in this case, `GenericService.xml`.

18. Select the **JXP SQL** tab.

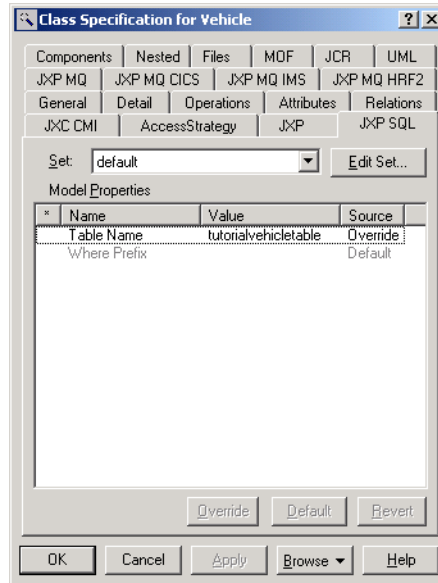


Figure 12-5: Specifying JXP SQL Information for the Vehicle Class

19. Click in the Value column next to Table Name and type tutorialvehicletable.
 20. Select the **JXC CMI** tab.

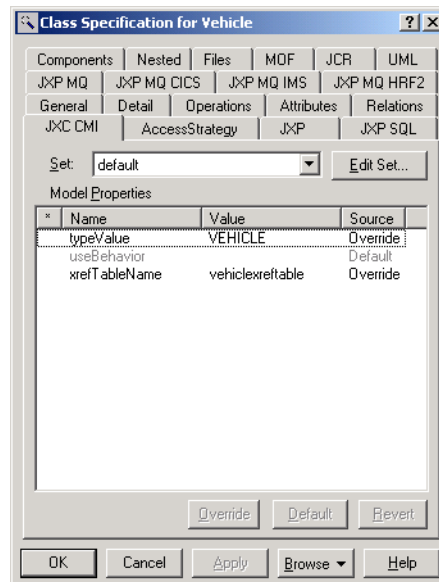


Figure 12-6: Specifying JXC CMI Information for the Vehicle Class

21. Type the values for these two JXC CMI properties:
 - typeValue =VEHICLE (all capital letters)
 - xreftablename = vehiclexreftable

For an explanation of the typeValue and the xreftablename, refer to [“Inheritance Tags” on page 224](#).
22. Select the **Attributes** tab. Double-click on the **Make** attribute to see its specifications.
23. Select the **JXP SQL** tab.
24. On the **JXP SQL** tab, click in the **Value** column next to these three specifications and type the values, as you did in [Step 11 on page 246](#):
 - Column Name = make
 - Size = 132
 - Logical Type = varchar

Accept the default values for the other specifications.
25. Repeat [Step 24](#) for Year, Model, and then Type.
 - Do **Type** last, since you must only complete [Step 26](#) for **Type**.
 - You do not have to enter JXP SQL information for **id**, since that information is inherited from the parent class.
 - As shown in [Figure 12-7](#), the size for the Year attribute is 4, since this attribute is an integer signifying a 4-digit year.

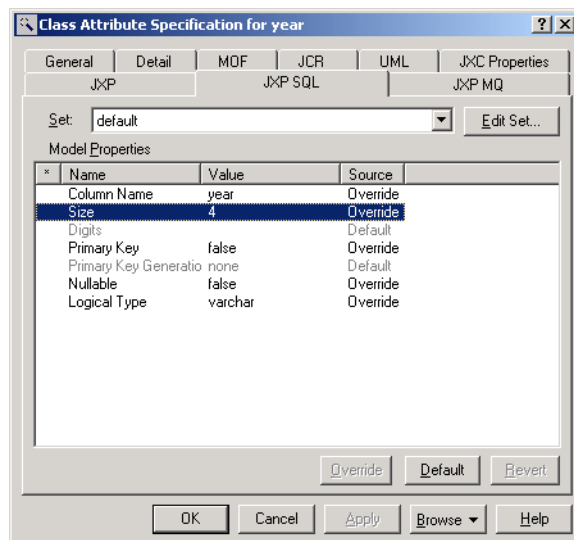


Figure 12-7: Specifying Size for Year Attribute

26. For the *Type* specifications only, select the **JXC Properties** tab. For other attributes, proceed to [Step 30 on page 252](#).

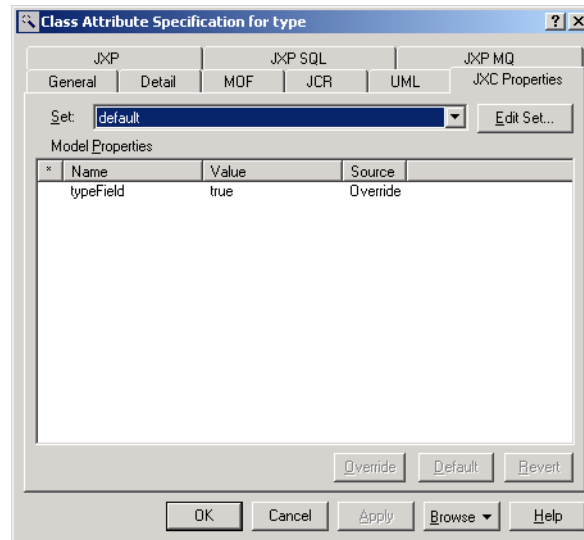


Figure 12-8: Specifying the typeField for Type

27. Click in the Value column next to typeField and type true.
28. Click **OK** to return to the **Vehicle Classes** class diagram.
29. Run the Business Component Generator to make sure that you have completed the steps to this point correctly. If you have problems, check that you filled in all of the information for all of the attributes of this class. Continue when you receive the desired, error-free output from the Business Component Generator.

For instructions on running the Business Component Generator, refer to [Chapter 3, “Using the Business Component Generator”](#).

Creating a New Subclass

30. Create a new class called Automobile.
31. Draw a generalization arrow from Automobile to Vehicle. The direction of this arrow is important. Create a new protected String attribute called `serialNumber`. Your diagram should look like [Figure 12-9](#).

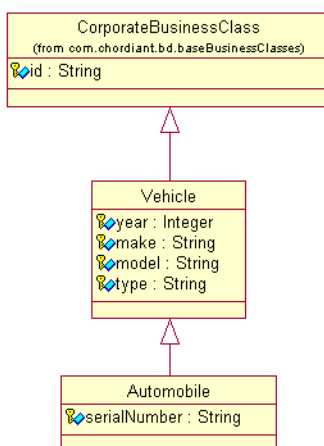


Figure 12-9: Automobile Classes Class Diagram with Automobile Class Created

32. Open the class specifications for Automobile and select the **JXC CMI** tab.

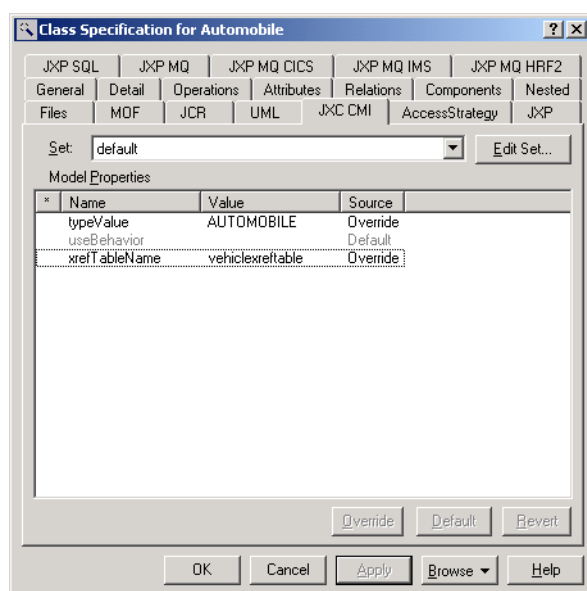


Figure 12-10: Specifying JXC CMI Information for Automobile Class

33. Type these two values, as you did in [Step 21 on page 250](#).
 - typeValue = AUTOMOBILE (using all capital letters)
 - xreftablename = vehiclexreftable
 34. Select the **JXP** tab. Type these three values, as you did earlier [Step 16](#) and [Step 17 on page 248](#).
 - DSN = chordiantXAds
 - Persistent Type = oracle8
 - Lock Strategy = none
 35. Click **OK** to close this window.
 36. View the attribute specifications for serialNumber.
 37. In the **Specifications for serialNumber** window, select the **JXP SQL** tab. Type the values for these three properties, as you did in [Step 11 on page 246](#):
 - Column Name = serialnumber
 - Size = 132
 - Logical Type = varchar
- Accept the default values for the other specifications.

Note: This step only needs to be done for serialNumber. The Automobile object inherits all other attributes from the parent Vehicle class.

38. Run the Business Component Generator to make sure you get the desired, error-free output. If everything looks correct, proceed to the next step.

39. Create a new class called **Boat**. Follow all of the instructions from [Step 30 on page 252](#) to [Step 38 on page 253](#). Most of the information will be the same, except:
 - Create a single protected integer attribute for the **Boat** class called **length**.
 - On the **JXC CMI** tab, specify the **typeValue = BOAT**, as shown in [Figure 12-11](#).

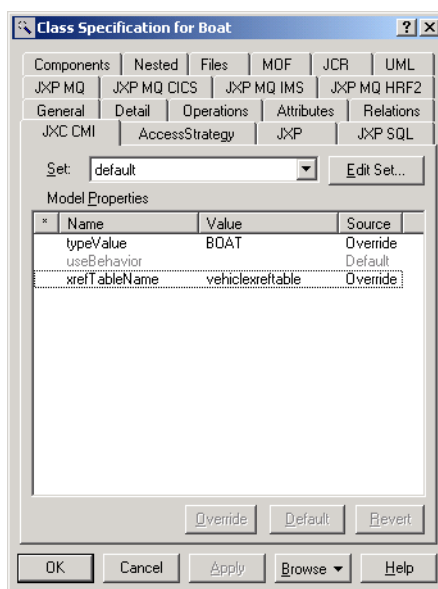


Figure 12-11: Specifying JXC CMI Information for Boat Class

- On the **JXP SQL** tab for the **Attribute Specifications for Length**, specify the **length = 4**.

At this point, your model should look like [Figure 12-12](#).

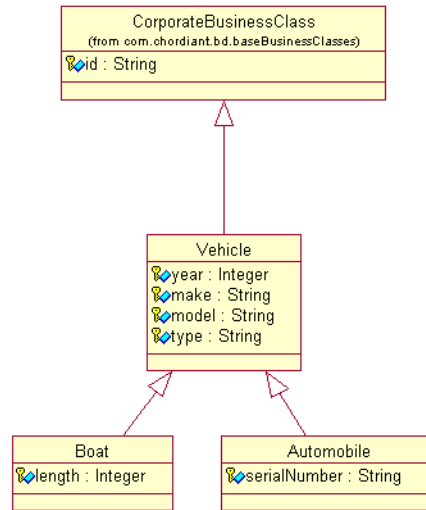


Figure 12-12: Model including Vehicle, Automobile, and Boat

Containment

This section of the tutorial illustrates these tags and concepts:

- class attribute
- foreign class attribute
- unidirectional aggregation

Note: Although this section starts with step 1, the procedure is continued from previous section.

1. Create a new class called **Engine** give it these three protected String attributes:
 - `make`
 - `serialNumber`
 - `automobileId`—This attribute will be the foreign key to the automobile table.

2. Follow instructions from previous steps:

Class Specifications for Engine: [Step 16 on page 248](#) through [Step 19 on page 249](#).

- **JXP** tab: All info same as other classes, since all attributes are Strings.
- **JXP SQL** tab: Set table name to tutorialenginetable.

Attribute Specifications for Engine — repeat for each attribute [Step 36 on page 253](#) through [Step 38 on page 253](#):

- **JXP SQL** tab: Set column name to the name of the attribute (make, serialNumber, and automobileId respectively).

3. Draw a generalization arrow from Engine to the Corporate Business Class.
4. Draw a unidirectional aggregation from Automobile to Engine. Direction is important.

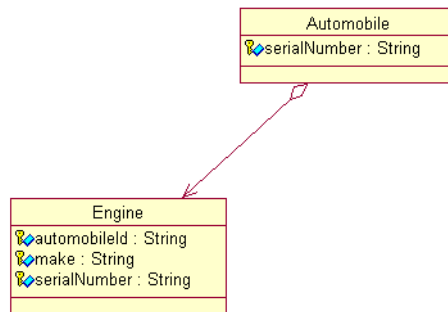


Figure 12-13: Unidirectional Aggregation from Automobile to Engine

5. Double-click the unidirectional aggregation to open its properties.
6. Select the **Role A Details** tab. In the **Role** box, type engine.

7. In the **Multiplicity** box, type 1..1. Then click **OK**.

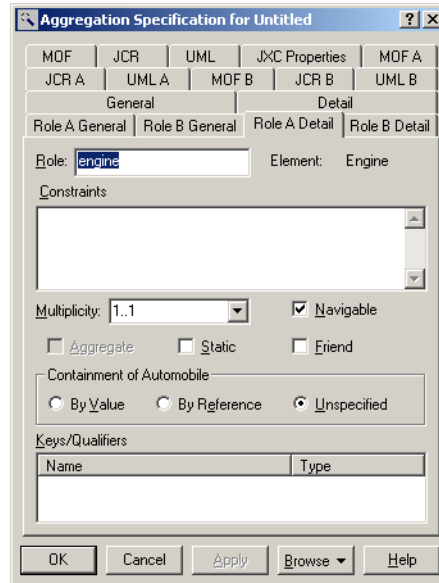


Figure 12-14: Specifying the Role and Multiplicity for the Engine

8. Select the **JXC Properties** tab.
- Set the `classAttribute` = `id`
 - Set the `foreignClassAttribute` = `automobileId`
 - Notice there is no collection type for this attribute, since it is a one-to-one association.

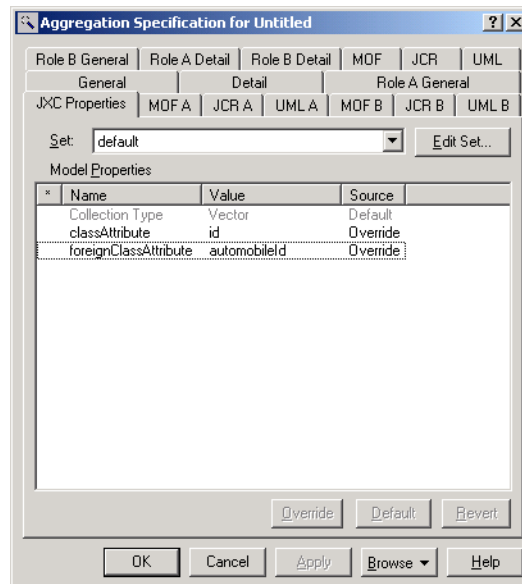


Figure 12-15: Specifying JXC Properties for One to One Relationship

9. Click **OK** to return to the class diagram. You will notice in your class model that the role of engine has been added to the aggregation line.

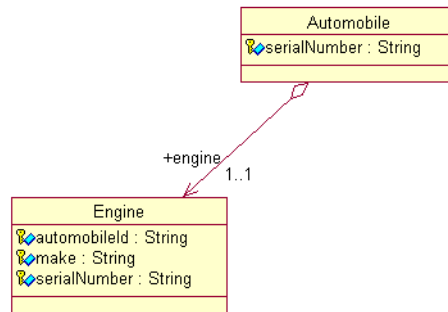


Figure 12-16: Unidirectional Aggregation with Role and Multiplicity

10. Run the Business Component Generator to make sure that you have completed the steps to this point correctly. If you have problems, check that you filled in all of the information for all of the attributes of this class. Continue when you receive the desired, error-free output from the Business Component Generator.
11. Create new class called Tire.
12. Create these three protected attributes:
 - automobileId: String This will be the foreign key.
 - make: String
 - sizeSpecification: String
13. Draw a generalization arrow from tire to the Corporate Business Class.

Note: If you choose, you can remove the Corporate Business Class from the diagram to keep the diagram neat. If you want to put it back, just drag the Corporate Business Class back onto the main window again. All of your arrows will reappear.

14. Draw a unilateral aggregation from Automobile to Tire. Direction is important.
15. Open the properties of the aggregation. Select the **Role A Detail** tab. In the **Role** box, type tires.

16. In the **Multiplicity** box, select 1..n. Click **Apply**.

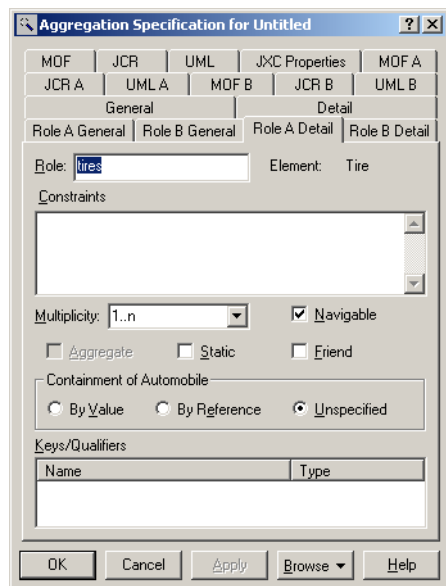


Figure 12-17: Specifying Role and Multiplicity for the Tires

17. Select the **JXC Properties** tab.

- Set the collectionType = Vector
- Set the classAttribute = id
- Set the foreignClassAttribute = automobileId

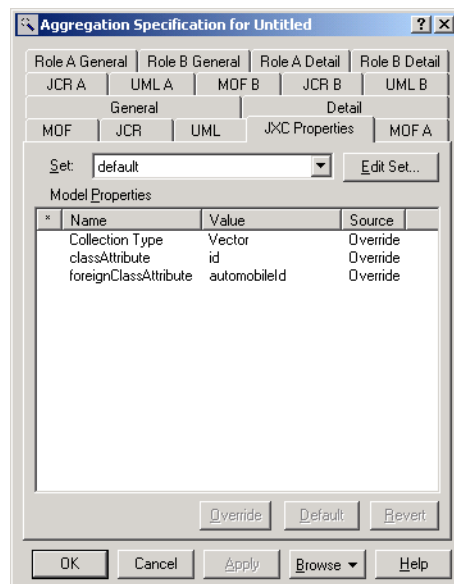


Figure 12-18: Specifying JXC Properties for One to Many Relationship

18. Go to the attribute specifications for each attribute in the Tire class. Follow the directions in [Step 36 on page 253](#) through [Step 38 on page 253](#) for each attribute:

JXP SQL tab:

- Set column name to the name of the attribute (automobileId, make, and sizeSpecification respectively).
- Size is the same for all attributes except automobileId. automobileId size = 47.

19. Click **OK** to return to the class model. At this point, your model should look like [Figure 12-19](#).

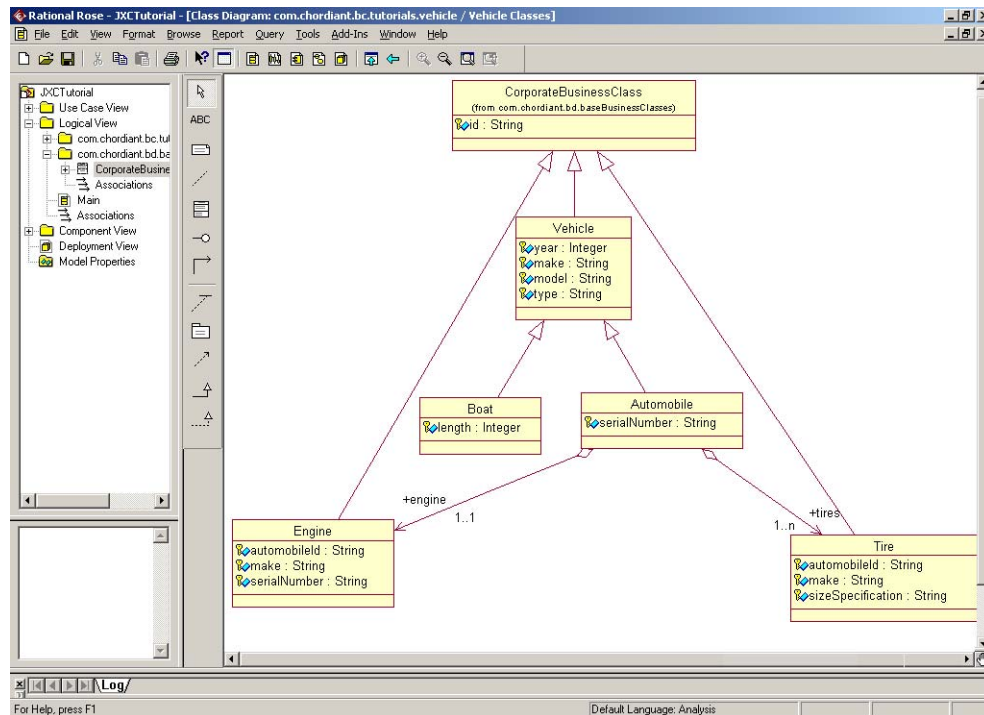


Figure 12-19: Object Model with Tire Class Created

20. Run the Business Component Generator to make sure that you have completed the steps to this point correctly. If you have problems, check that you filled in all of the information for all of the attributes of this class. Continue when you receive the desired, error-free output from the Business Component Generator.

Associations

This section of the tutorial illustrates the concept of association.

Note: Although this section starts with step 1, the procedure is continued from previous section.

1. Create a new class called Dealer.
2. Draw a generalization arrow from Dealer to the Corporate Business Class.

Note: If you choose, you can remove the Corporate Business Class from the diagram to keep the diagram neat. If you want to put it back, just drag the Corporate Business Class back onto the main window again. All of your arrows will reappear.

3. Draw an association from Automobile to Dealer.
4. Add these two protected String attributes to the Dealer class:
 - automobileId
 - name
5. Fill in the specifications for the Dealer class and its attributes, as you have in previous steps in this tutorial.

Class Specifications for Dealer: [Step 16 on page 248](#) through [Step 19 on page 249](#).

- **JXP** tab: All info same as other classes, since all attributes are Strings.
- **JXP SQL** tab: Set table name to tutorialdealertable.

Attribute Specifications for Dealer — Repeat for each attribute [Step 36 on page 253](#) through [Step 38 on page 253](#):

- **JXP SQL** tab: Set column name to the name of the attribute (automobileId and name).

6. Open the association properties. Select the **General** tab.

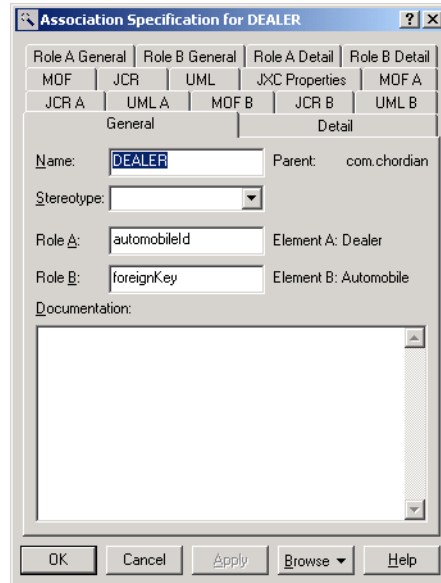


Figure 12-20: Association Specifications for Dealer

7. Type these three specifications for the association:
 - name = DEALER
 - role A = automobileID
 - role B = foreignKey

8. Click **OK** to return to the object model. It will look like [Figure 12-21](#).

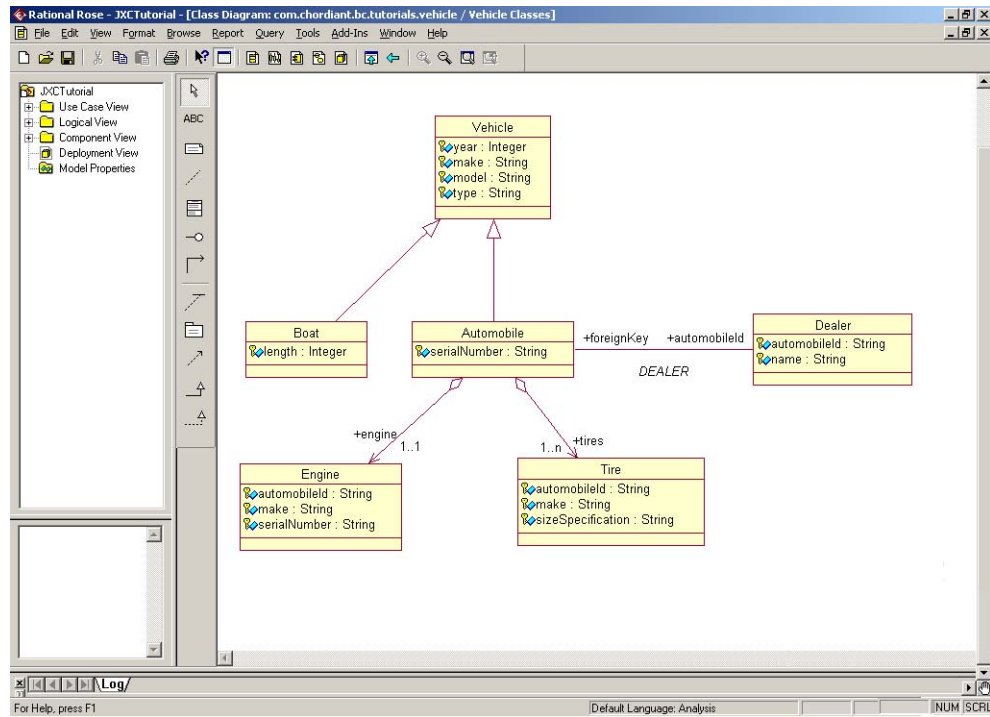


Figure 12-21: Object Model with Dealer Class and Association Created

9. Run the Business Component Generator to make sure that you have completed the steps to this point correctly. If you have problems, check that you filled in all of the information for all of the attributes of this class.

The Business Component Generator will create components in the directories shown in [Figure 12-22](#).

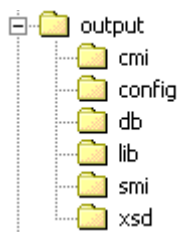


Figure 12-22: Output Directories

- **cmi**: contains the CMI file. A Chordiant-based XMI file which is used to create the generated components.
- **config**: contains the behavior factory and validation configuration files.
- **db**: contains the SQL files for creating database schema for your model.
- **lib**: contains the JAR file for the generated code
- **smi**: contains the SMI file. A Chordiant-based XMI file which is used to create the service framework files.
- **xsd**: the schema created from your model

Run the Schema

1. Run the SQL script in SQLplus. You might need to modify the script before you run it. Examine it first to make sure it suits your needs.

Note: The first time you run the SQL script, you will get errors because of drop table and drop synonym commands. Run it again and it should run smoothly.

Overriding Behavior

When you customize a behavior, use the same general rule: subclass the existing class and override or add methods.

Note: Although you can add functionality to a behavior, it is important to note that this behavior will only be accessible through Java clients. Clients accessing services through IIOP or sockets will not have access to methods on behaviors.

1. Make sure your model is accurate. Run the Business Component Generator to create all of the code for the model.
2. Open the generated behavior, `automobileBehavior.java`. It will be located in the `{project}/src/generated/{object package name}/behaviors` directory.
3. Save the behavior in a different location (in a custom directory) with a different name (prepended with extended), so you can make modifications and so it will not be overwritten when you run the Business Component Generator again. Save it as:

```
com.chordiant.bc.tutorial.vehicle.behaviors.custom.  
ExtendedAutomobileBehavior.java
```

4. Extend the parent behavior.
5. Delete all basic functions. They are already inherited from the parent behavior.
6. Add a `getDealer` method. Make it return a dealer *behavior*.

Tip: It is more convenient to return a behavior instead of returning the object itself. There are several methods on the behavior, so you can do something with the dealer immediately, without first having to get a behavior from the factory.

[Code Sample 12-1](#), we have included logic to make sure only one dealer is returned. Refer to [Code Sample 12-1](#), and the complete example code installed with this tutorial.

```
public DealerBehavior getDealer() throws Exception  
{  
    final String METHOD_NAME = "getDealer";  
    LogHelper.methodEntry( PACKAGE_NAME, CLASS_NAME, METHOD_NAME );  
    List dealerList = retrieveAssociation("DEALER");  
    // Check that only a single dealer was returned  
    if ( dealerList.size() == 0 ) {  
        LogHelper.methodExit( PACKAGE_NAME, CLASS_NAME, METHOD_NAME );  
    }  
}
```

Code 12-1: getDealer Method Logic

```

        return null;
    }
    if ( dealerList.size() > 1 ) {
        LogHelper.error( PACKAGE_NAME, CLASS_NAME, METHOD_NAME,
            "Multiple Dealers returned. Returning the first one");
    }
    Dealer dealer = (Dealer)dealerList.get(0); // Get first entry in list
    DealerBehavior result = (DealerBehavior)BehaviorFactory.
        getBehaviorByObject( getUsername(), getAuthentication(), dealer );
    LogHelper.methodExit( PACKAGE_NAME, CLASS_NAME, METHOD_NAME );
    return result;
}

```

Code 12-1: *getDealer Method Logic (Continued)*

7. Compile your code to make sure it is accurate.
8. Go to the model. View the Automobile Class Specifications and select the **JXC CMI** tab.
9. For useBehavior, specify the fully-qualified class path for the new behavior you just created.

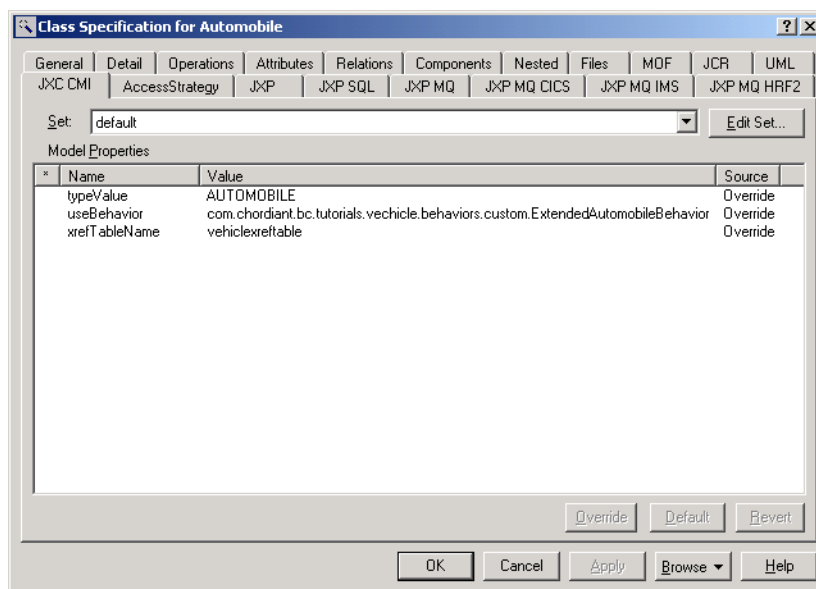


Figure 12-23: *Overriding the Automobile Behavior*

10. Click **Override**. Click **OK**.

Overriding Access Strategy

When you override an access strategy, it affects data every time the access strategy is called. You will not override access strategies just for certain circumstances or geographies. For the purposes of this tutorial, we created a new access strategy that will prepend “Auto-” to each tire serial number. This is not a very powerful example, but illustrates how to perform the override.

Note: A sample of the `ExtendedAutomobileAccessStrategy.java` file is provided with the model. It is located in `{eclipse_root}/plugins/{data_model_plugin}/tutorial_source/src/com/chordiant/bc/tutorials/vehicle/accessstrategies/ExtendedAutomobileAccessStrategy.java`. We recommend that you save it under a separate root directory, such as `src/custom/com/chordiant/bc/tutorials/vehicle/accessstrategies`, so it doesn't get overwritten upon file generation.

1. Make sure your model is correct and run the Business Component Generator to generate your code.
2. Open the generated access strategy, `AutomobileAccessStrategy.java`. It will be located in the `{project}/src/generated/{object package name}/accessstrategies` directory.
3. Save the access strategy in a different location (in a custom directory) with a different name (prepended with `extended`), so you can make modifications and so it will not be overwritten when you next run the Business Component Generator. Save it as:
`com.chordiant.bc.tutorial.vehicle.accessstrategies.
custom.ExtendedAutomobileAccessStrategy.java`
4. Extend the parent access strategy.
5. In your new access strategy, remove everything except the `execute` and `retrieve` methods. Everything else is inherited from the parent.
6. For the `execute` method, leave everything, but specify that if it makes a call to `retrieve`, it calls the special `retrieve` in this access strategy (refer to [Step on page 268](#)). Otherwise, call the super class methods.

```
public Object execute(IAccessStrategyInput in) throws Exception
{
    String METHOD_NAME = "execute";
    LogHelper.methodEntry( PACKAGE_NAME, CLASS_NAME, METHOD_NAME );

    if (!( in instanceof GeneratedAccessStrategyInputBase ))
    {
        LogHelper.error( PACKAGE_NAME, CLASS_NAME, METHOD_NAME,
            "AccessStrategyInput passed to AutomobileAccessStrategy does not
            inherit from GeneratedAccessStrategyInputBase" );
        throw new InvalidParameterException( "in", in,
            IBusinessServiceErrors.BS_RUNTIME_ERROR );
    }
}
```

Code 12-2: Execute Method Logic

```

// Downcast the input so we can use the desired get methods
GeneratedAccessStrategyInputBase input =
    (GeneratedAccessStrategyInputBase)in;

Automobile bo = (Automobile)input.getContainedBo();

if ( bo == null ) {
    LogHelper.error( PACKAGE_NAME, CLASS_NAME, METHOD_NAME, "Error: passed
        business object is null in AutomobileAccessStrategyInput" );
    return null;
}

String username = input.getUsername();
String authentication = input.getAuthentication();
ISecurityInformation securityInformation = input.getSecurityInformation();
Object additionalData = input.getAdditionalData();

int operation = input.getOperation();
Object result = null;

if ( operation==GeneratedAccessStrategyBase.RETRIEVE_OPERATION ) {
    if (ValidatorFactory.getValidatorByObject( bo ).
        retrieveValidate( bo )) {
        result = retrieve( username, authentication, securityInformation,
            additionalData, bo );
    }
}
else {
    super.execute( in );
}

LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
return result;
}

```

Code 12-2: Execute Method Logic (Continued)

7. Replace almost all of the `retrieve` method to specify your customization. In [Code Sample 12-3](#), we prepend “AUTO-” to the tire serial number. Note that this `retrieve` method is actually calling `super.retrieve`.

```

protected Object retrieve(String username, String authentication,
    ISecurityInformation securityInformation, Object additionalData,
    Automobile bo) throws Exception
{
    String METHOD_NAME = "retrieve";
    LogHelper.methodEntry( PACKAGE_NAME, CLASS_NAME, METHOD_NAME );

    Automobile result = (Automobile)super.retrieve( username, authentication,
        securityInformation, additionalData, bo);

    if ( ( result != null ) && ( result.getSerialNumber() != null ) ){
        // Pre-pend the string AUTO- to the front of the serial number
        result.setSerialNumber( "AUTO-" + result.getSerialNumber() );
    }

    LogHelper.methodExit( PACKAGE_NAME, CLASS_NAME, METHOD_NAME );
    return result;
}

```

Code 12-3: Updating the Retrieve Method

8. Compile your code to make sure it is accurate. Continue below.

Modifying accessStrategyInput

Once you change the access strategy, you must create a corresponding accessStrategyInput for it.

Note: A sample of the ExtendedAutomobileAccessStrategyInput.java file is provided with the model. It is located in `{eclipse_root}/plugins/{data_model_plugin}/tutorial_source/src/com/chordiant/bc/tutorials/vehicle/accessstrategies/inputs/ExtendedAutomobileAccessStrategyInput.java`. We recommend that you save it under a separate root directory, such as `src/custom/com/chordiant/bc/tutorials/vehicle/accessstrategies/inputs`, so it doesn't get overwritten upon file generation.

9. Open the generated accessStrategyInput, automobileAccessStrategyInput.java. It will be located in the `{project}/src/generated/{object package name}/accessstrategies` directory.
10. Save the accessStrategyInput in a different location (in a custom directory) with a different name (prepended with extended), so you can make modifications. Save it as:

```
com.chordiant.bc.tutorial.vehicle.accessstrategies.  
custom.inputs.ExtendedAutomobileAccessStrategyInput
```

The accessStrategyInput must have the same name as the access strategy, with "input" at the end.

11. Return to your object model. Open the **Class Specifications for Automobile**. Select the **AccessStrategy** tab. Override the access strategy for the retrieve method, specifying the fully-qualified name of the new access strategy you created.

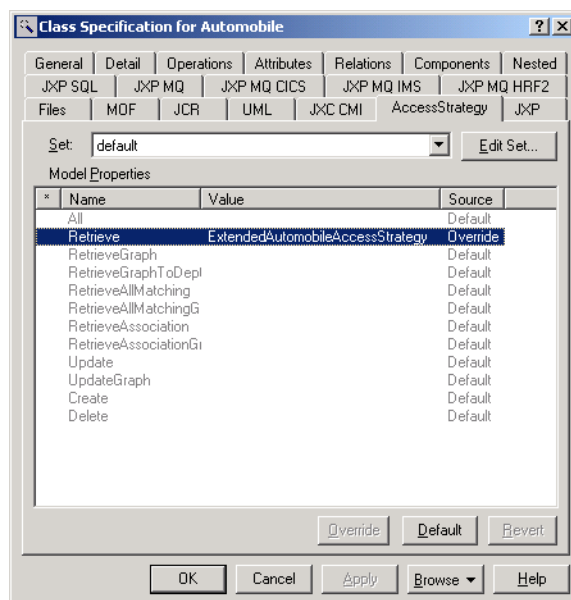


Figure 12-24: Overriding the Access Strategy for the Retrieve Method

12. Click **Override**. Click **OK**.
13. Run the Business Component Generator to create your new components.

CREATING TESTERS

The Business Component Generator automatically creates JUnit testers for every object in your model. These testers are skeletons and must be filled in to be useful. You must also seed data in the database so your testers will have something to work against. Scripts to add data to your database are included with this tutorial.

Since you will probably not use all of the testers, you might choose to delete the ones you will not need. In this tutorial, we are only interested in the vehicle and automobile testers. We deleted the other tester files.

Note: When you are making customizations, be sure to move your customized components to a different location so they will not be overwritten the next time you use the Business Component Generator.

A generated `TestSuite.java` file is also created by the Business Component Generator. This file calls each of the tester files, in turn, from a single location.

To customize the testers:

1. Delete any testers you will not be using.
2. In the `com.chordiant.bc.testsuites.generatedTestSuite.java` file, remove lines corresponding to the deleted testers you will not be using. Save and close this file.
3. In the Vehicle tester, add code for the new access strategy and new Behavior you created. Refer to the `vehicleBehaviorTest.java` file included with the example code.
4. Before you run the testers, you must seed the data, so your tester will have something to run against. The seed data is located in the generated file `populateValuesORACLE.sql` in the `\tutorial_source\seed` directory. Run this script to populate the database.

Note: The first time you run the SQL script, you will get errors because of drop table and drop synonym commands. Run it again and it should run smoothly.

5. In the `vehicleBehaviorTest.java`, check that the `retrieveAllMatching` and `retrieveAllMatchingGraphs` tests work. Both Boat and Automobile should be returned.
6. Run the `automobileBehaviorTest`. The test contains seed data and tests the `retrieve`, `retrieveGraph`, and `retrieveGraphToDepth` functions. For the full graph, the automobile, as well as its engine and tires should be returned.

This tester also tests the `getDealer` behavior you created, with its `retrieveAssociation` and `retrieveAssociationGraph`. Check that this returns a dealer for the specified car.

This is the end of the tutorial. Please feel free to explore the tutorial more. If you choose to make your own customizations to it, as with all customizations, we suggest that you make your changes within a separate directory so you will not lose the functionality of the original and so your components will not get overwritten when you run the Business Component Generator.

Party Management Facility

Managing parties and their respective roles is a key requirement of enterprise software. A party is a person or organization that can participate in a business relationship, while a role is the business view of the person or organization, such as customer and prospect.

Chordiant 5 Foundation Server provides a Party/Role application component that you can use as a Party Management Facility, enabling you to model and manage the various parties that participate in a business relationship with your enterprise. Party Management Facility (PMF) is an OMG standard.

The Party Management Facility consists of these two entities:

- The Party/Role object model
- The Party/Role Service

This enables your applications to treat persons and organizations as customers of your enterprise. This significantly differs from the earlier Customer object model, which assumed that a customer was always a person.

Notes: You can use the `PartyRoleService` directly from client applications. You can also use components, such as the `PmfCustomerService`, which extends facilities provided in the `PartyRoleService` and adds role-specific behavior.

An extension of the `PmfCustomerService`, which is itself an extension of the `PartyRoleService`, is called the `PmfDelegateService`. The `PmfDelegateService` was created to include the concept of a *delegate* — an existing customer who is able to act on behalf of another customer. You can use this service as is or you can customize it for your own application. For additional details on the `PmfDelegateService`, refer to [“PmfDelegateService” on page 350](#) and to the `PmfDelegateService`’s Javadoc.

Figure 13-1 illustrates the object model for the Party Management Facility.

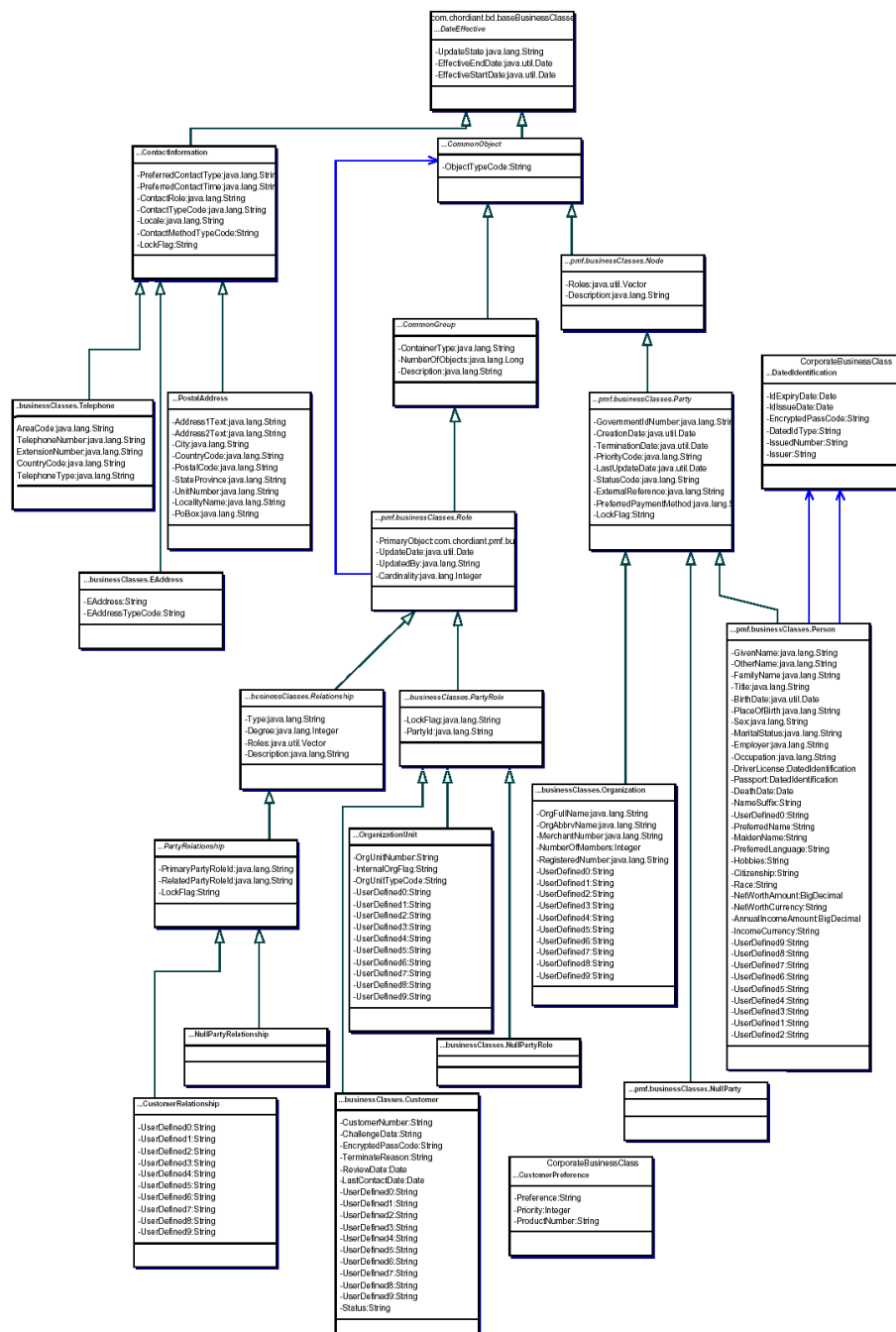


Figure 13-1: Party Management Facility Object Model

Take note of these points about the Party/Role object model:

- **CommonObject** is the common base class for business objects that can serve as groups and the business objects that can be organized in groups.
- **CommonGroup** serves as the base class for objects that are not only business objects, but also serve to aggregate business objects.
- **Node** serves as a base class providing a richer level of aggregation and relationship traversal for business objects that can be grouped.
- **Role** acts as the base class enabling applications to express roles for other business objects.
- **PartyRole** is the base class for business objects that express roles played by Party objects.
- **Relationship** is a specialization of Role, and is used to aggregate roles with participation constraints.
- **PartyRelationship** provides constrained aggregation of **PartyRole** objects
- **ContactInformation** acts as the base class to **PostalAddress**, **Telephone**, and **EAddress** objects. **ContactInformation** is related to **PartyRole** as well as **Party** objects.

The Party Management Facility offers significant advantages. In earlier versions of Foundation Server, a **Customer** object, or example, had no relationship to the **Party** object model. Now, using the Party/Role object model, a **Customer** object is a subtype of **PartyRole**, and a **PartyRole** is related to a **Party**.

In addition, you can customize the behavior of the Party/Role Application Component using the Chordiant 5 Foundation Server customization model. For more information, refer to the *[“Customizing the PartyRole Service” on page 297.](#)*

USING THE PARTY MANAGEMENT FACILITY API

The Party Management Facility API can be logically partitioned into these two categories:

- Business object methods
- Manager (business service) methods

The behavior for the business objects and managers is implemented through the Party Role service. This means that client applications use a client agent to access the behavior, similar to every other Chordiant service.

In addition, for business object interfaces, you must supply a reference to the intended target of the operation as a parameter when using the Party Role methods.

This section describes the Party Management Facility interface, and includes the following topics:

- [“Using the CommonObject Interface” on page 277](#)
- [“Using the Role Interface” on page 279](#)
- [“Using the Node Interface” on page 281](#)
- [“Using the Party Interface” on page 284](#)
- [“Using the PartyRole Interface” on page 288](#)
- [“Using the Manager Interface” on page 292](#)
- [“Using the PartyManager Interface” on page 292](#)
- [“Using the RoleManager Interface” on page 293](#)
- [“Using the PartyRoleManager Interface” on page 294](#)
- [“Using the RelationshipManager Interface” on page 294](#)
- [“Example Scenarios” on page 295](#)

Tip: For all of these APIs, you can also refer to the Javadoc for more information.

Using the CommonObject Interface

The abstract `CommonObject` interface offers the core behavior inherited by most of the remaining Party Management interfaces. The `CommonObject` is a subclass of the `DateEffective` object, enabling the Party Management Facility classes to record date and time stamps.

Figure 13-2 illustrates the attributes for the `CommonObject` object, as well as the attributes for related objects.

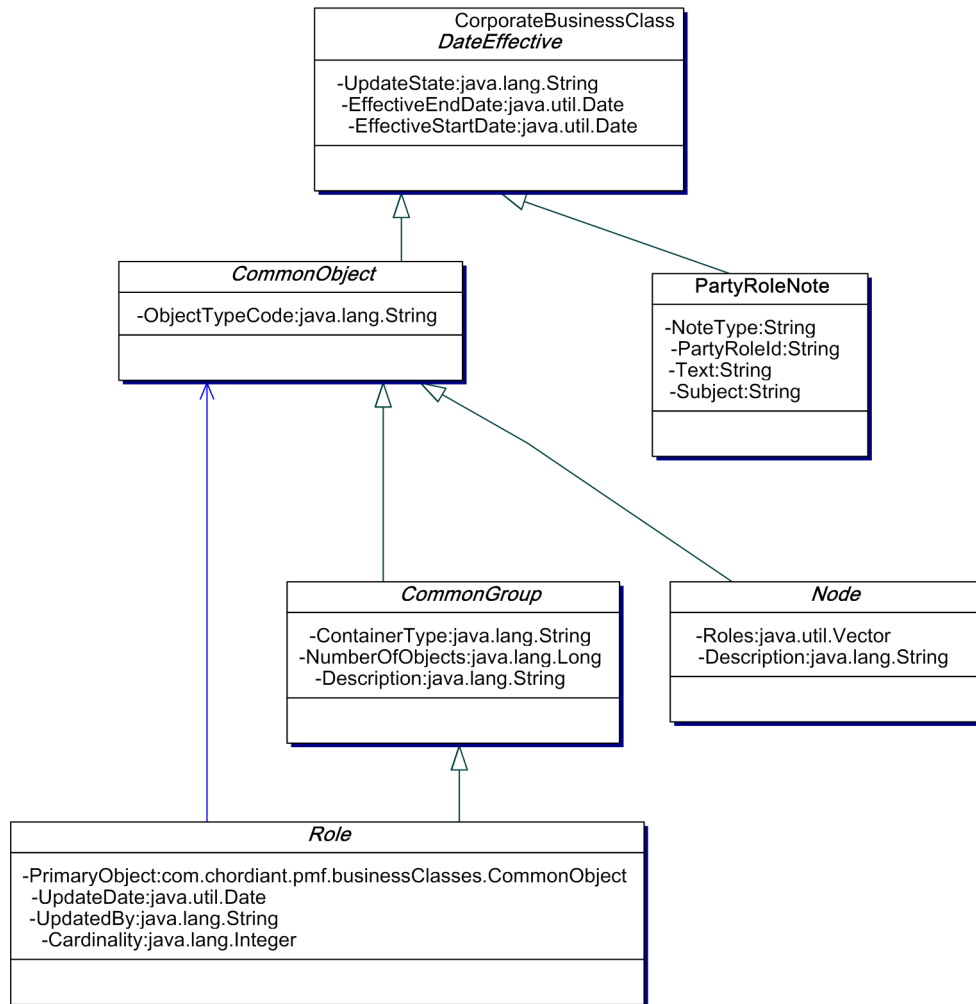


Figure 13-2: CommonObject Object

The `CommonObject` interface offers the following functionality:

- **getCommonObject**—Retrieves the `CommonObject` based on the attributes of the specified `CommonObject` object.

```
public CommonObject getCommonObject(  
    String username,  
    String authentication,  
    CommonObject theBO)  
    throws ObjectNotFoundException, Exception
```

Code 13-1: The getCommonObject Method

Note: theBO is the leaf node of the party|partyRelationship|partyRole and should have enough information to uniquely identify itself.

- **updateCommonObject**—Updates the `CommonObject` based on the attributes of the specified `CommonObject` object.

```
public CommonObject updateCommonObject(  
    String username,  
    String authentication,  
    CommonObject theBO)  
    throws Exception
```

Code 13-2: The updateCommonObject Method

Using the Role Interface

The **Role** interface serves as a base interface for **PartyRole**, along with all related roles that parties can play. [Figure 13-3](#) illustrates the attributes for the **Role** object, as well as the attributes for related objects.

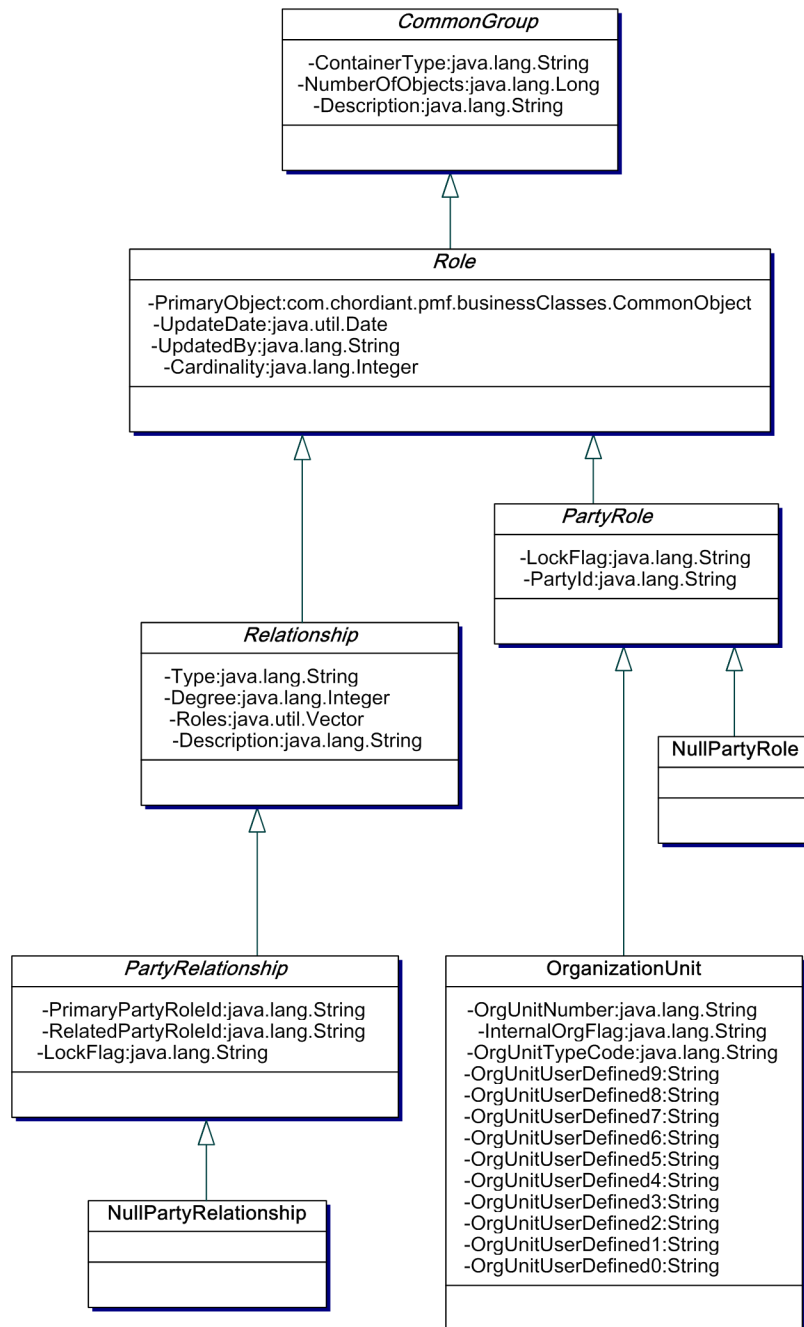


Figure 13-3: Role Object

The Role Interface offers the following functionality:

- **addRelatedObject**—Creates a relationship between a Role and another CommonObject.

```
public abstract void addRelatedObject(  
    String username,  
    String authentication,  
    Role theBO,  
    Role addOtherRole)  
    throws  
        IsDuplicateException,  
        InvalidRoleException,  
        InvalidAggregationException,  
        MaximumCardinalityExceededException,  
        ParameterValueException,  
        Exception;
```

Code 13-3: The addRelatedObject Method

- **getRelatedObject**—Returns an object based on the role of a related object. For example, you can use this method to obtain a reference to a CommonObject representing the person to which someone was married at the time. You can use this method to determine 1:1 relationships.

The **getRelatedObject** method throws a **MoreThanOneRelated** exception in cases when the relationship is determined to be 1:n, as would be the case for an employer/employee relationship. Likewise, the method throws an **InvalidRelatedRole** exception in cases when there is no relationship between the objects (as would be the case if a person does not have a spouse).

```
public CommonObject getRelatedObject(  
    String username,  
    String authentication,  
    Role theBO,  
    Role relatedRole)  
    throws  
        MoreThanOneRelatedException,  
        InvalidRelatedRoleException,  
        ParameterValueException,  
        ParameterRequiredException,  
        Exception
```

Code 13-4: The getRelatedObject Method

- **getAllRelatedObjectsByRole**—Returns all related CommonObjects based on a specified role. You can use this method to determine 1:1 relationships.

For example, you could use this method in cases when the CommonObject represents an employer, and the client needs to determine all related employee objects.

```
public Vector getAllRelatedObjectsByRole(  
    String username,  
    String authentication,  
    Role theBO,  
    Role relatedRole)  
    throws  
        InvalidRoleException,  
        ParameterValueException,  
        InvalidRelatedRoleException,  
        Exception
```

Code 13-5: The getAllRelatedObjectsByRole Method

- **removeRelatedObject**—Breaks the relationship between a Role and another CommonObject.

```
public void removeRelatedObject(
    String username,
    String authentication,
    Role theBO,
    Role relatedRole)
    throws
        ObjectNotFoundException,
        ParameterValueException,
        ParameterRequiredException,
        Exception
```

Code 13-6: The removeRelatedObject Method

Using the Node Interface

The **Node** interface derives from the **CommonObject** object enabling applications to obtain all the roles that a particular object is playing. [Figure 13-4](#) illustrates the attributes for the **Node** object, as well as the attributes for related objects.

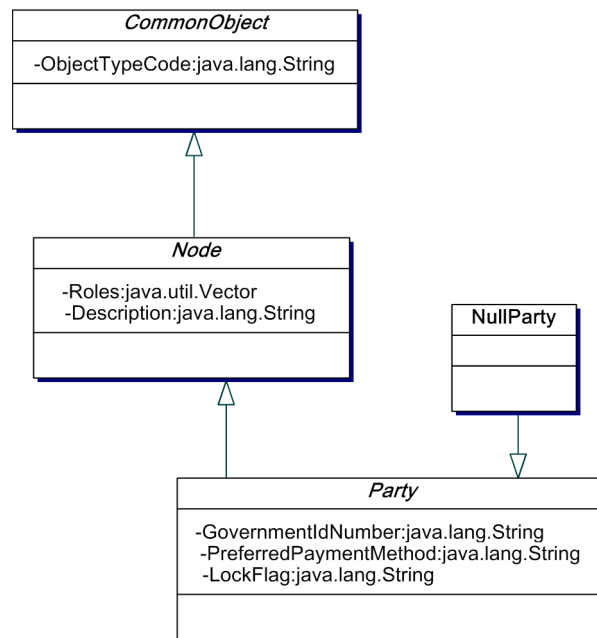


Figure 13-4: Node Object

The **Node** Interface offers the following functionality:

- **addRole**—Assigns a new role to an existing node.

```
public abstract void addRole(  
    String username,  
    String authentication,  
    Node theBO,  
    Role newRole)  
    throws  
        TypeNotSupportedException,  
        ParameterValueException,  
        ParameterRequiredException,  
        BusinessServiceException,  
        Exception
```

Code 13-7: The addRole Method

- **getAllRoleNames**—Returns a sequence of strings representing all of the roles this object currently plays.

```
public abstract Vector getAllRoleNames(  
    String username,  
    String authentication,  
    Node theBO)  
    throws  
        NotSupportedException,  
        Exception
```

Code 13-8: The getAllRoleNames Method

- **getAllRoles**—Returns a list of all roles played by a specific **CommonObject**. These roles represent the relationships created using methods in the **CommonGroup** class, a subclass of **CommonObject**.

For example, when a **CommonObject** represents a person, this method could return one or more of: **Husband**, **Claimant**, **Lienholder**, **Attorney**.

```
public abstract Vector getAllRoles(  
    String username,  
    String authentication,  
    Node theBO)  
    throws  
        NotSupportedException,  
        Exception
```

Code 13-9: The getAllRoles Method

- **getRoles**—Returns the roles associated with the specified role name. For example, the **getRoles** method can return a single reference or multiple role references, depending on the relationship.

```
public abstract Vector getRoles(  
    String username,  
    String authentication,  
    Node theBO,  
    Role theRole)  
    throws  
        UnknownRoleNameException,  
        NotSupportedException,  
        ParameterValueException,  
        ParameterRequiredException,  
        Exception
```

Code 13-10: The getRoles Method

- **removeRole**—Breaks the relationship between a Node object and a role.

```
public abstract void removeRole(  
    String username,  
    String authentication,  
    Node theBO,  
    Role removeRole)  
    throws  
        UnknownRoleNameException,  
        RoleNotFoundException,  
        ParameterValueException,  
        ParameterRequiredException,  
        Exception
```

Code 13-11: The removeRole Method

Using the Party Interface

The Party interface serves as the base interface for **Person** and **Organization** objects, enabling applications to obtain party-related information. [Figure 13-5](#) illustrates the attributes for the Party object, as well as the attributes for related objects.

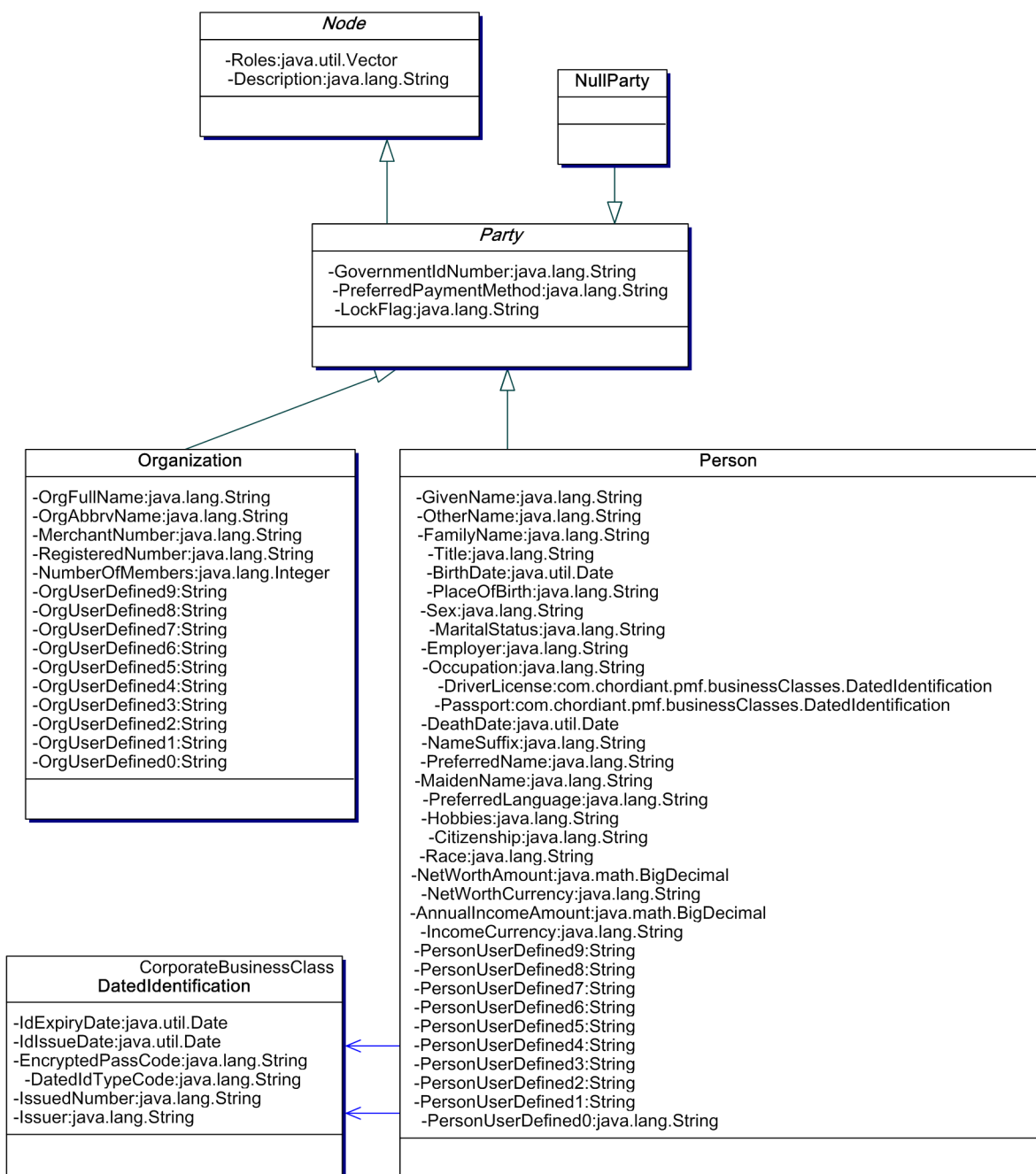


Figure 13-5: Party Object

The **Party** object extends the **Node** interface, and offers the following interface:

- **getAllPaymentMethods**—Returns all payment methods for the specified Customer.

```
public Vector getAllPaymentMethods(
    String username,
    String authentication,
    Party theB0)
    throws
        LockUnavailableException,
        UnexpectedMultipleRecordsException,
        BusinessServiceException,
        DataSourceException,
        ParameterRequiredException,
        ParameterValueException,
        Exception
```

Code 13-12: The getAllPaymentMehtods Method

- **getContactInformation**—Returns a sequence of references to all Contact Information objects related to a party.

```
public Vector getContactInformation(
    String username,
    String authentication,
    Party theB0)
    throws Exception
```

Code 13-13: The getContactInformation Method

- **getParties**—Returns the **Party** objects for the specified **Party** object.

```
public Vector getParties(
    String username,
    String authentication,
    Party theB0)
    throws
        UnexpectedMultipleRecordsException
        LockUnavailableException
        BusinessServiceException
        DataSourceException
        ParameterRequiredException
        ParameterValueException,
        Exception
```

Code 13-14: The getParties Method

- **getPartiesByAddress**—Returns the parties associated with a given postal address.

```
public Vector getPartiesByAddress(
    String username,
    String authentication,
    PostalAddress postalAddress)
    throws
        UnexpectedMultipleRecordsException,
        LockUnavailableException,
        BusinessServiceException,
        DataSourceException,
        ParameterRequiredException,
        ParameterValueException,
        Exception
```

Code 13-15: The getPartiesByAddress Method

- **getPartiesByDatedIdent**—Returns the parties associated with a specified dated identification.

```
public Vector getPartiesByDatedIdent(  
    String username,  
    String authentication,  
    DatedIdentification datedIdent)  
    throws  
        UnexpectedMultipleRecordsException,  
        LockUnavailableException,  
        BusinessServiceException,  
        DataSourceException,  
        ParameterValueException,  
        Exception
```

Code 13-16: The getPartiesByDatedIdent Method

- **getPartiesByEAddress**—Returns the parties associated with a specified electronic address.

```
public Vector getPartiesByEAddress(  
    String username,  
    String authentication,  
    EAddress anEAddress)  
    throws  
        UnexpectedMultipleRecordsException,  
        LockUnavailableException,  
        BusinessServiceException,  
        DataSourceException,  
        ParameterRequiredException,  
        ParameterValueException,  
        Exception
```

Code 13-17: The getPartiesByEAddress Method

- **getPartiesByPartyIds**—Returns the Party objects based on the specified party identifiers.

```
public Vector getPartiesByPartyIds(  
    String username,  
    String authentication,  
    Vector partyIds)  
    throws  
        UnexpectedMultipleRecordsException,  
        LockUnavailableException,  
        BusinessServiceException,  
        DataSourceException,  
        ParameterValueException,  
        ParameterRequiredException,  
        Exception
```

Code 13-18: The getPartiesByPartyIds Method

- **getPartiesByTelephone**—Returns the parties associated with a specified telephone number.

```
public Vector getPartiesByTelephone(  
    String username,  
    String authentication,  
    Telephone aTelephoneNumber)  
throws  
    UnexpectedMultipleRecordsException,  
    LockUnavailableException,  
    BusinessServiceException,  
    DataSourceException,  
    ParameterRequiredException,  
    ParameterValueException,  
    Exception
```

Code 13-19: The getPartiesByTelephone Method

- **setContactInformation**—Uses the behavior object associated with the specified business object to associate a list of contact information objects.

```
public Vector setContactInformation(  
    String userName,  
    String authentication,  
    Party theBO,  
    Vector collectionOfCIObjcts)  
throws Exception
```

Code 13-20: The setContactInformation Method

Using the PartyRole Interface

A **PartyRole** represents a **Party** (person or an organization) in a relationship. The **PartyRole** interface simplifies the **Role** interface by providing higher-level wrappers to the base functionality of the **Role** interface.

Figure 13-6 illustrates the attributes for the **PartyRole** object, as well as the attributes for related objects.

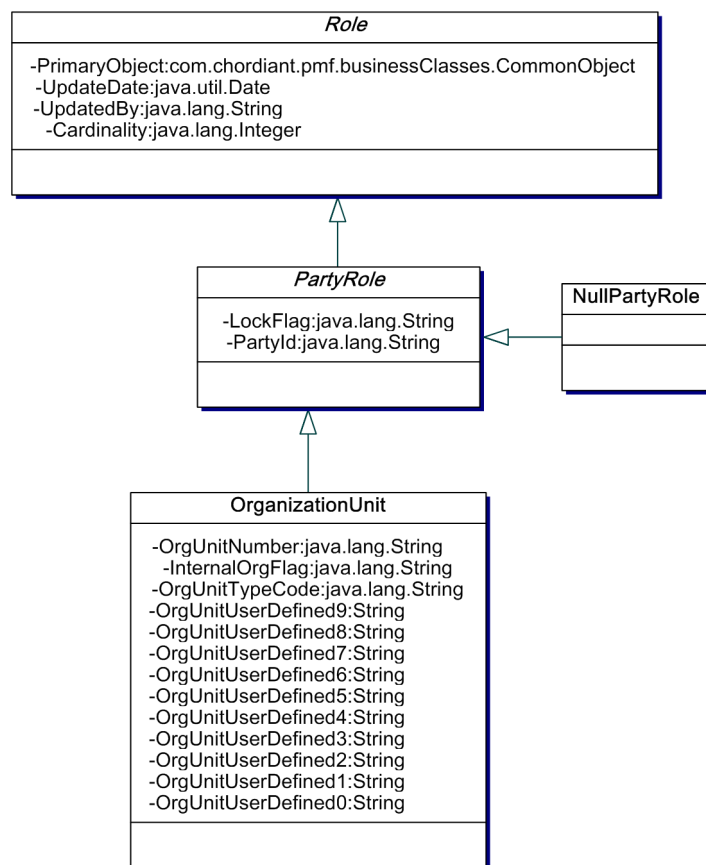


Figure 13-6: PartyRole Object

The `PartyRole` object extends the `Role` interface, and offers the following interface:

- **addNote**—Adds a note.

```
public PartyRoleNote addNote(
    String username,
    String authentication,
    PartyRoleNote newNote)
    throws
        ParameterValueException,
        Exception
```

Code 13-21: The addNote Method

- **addRelatedPartyRole**—Associates a party role with another party role.

```
public void addRelatedPartyRole(
    String username,
    String authentication,
    PartyRole theB0,
    PartyRole relatedPartyRole)
    throws
        IsDuplicateException,
        InvalidRoleException,
        InvalidAggregationException,
        MaximumCardinalityExceededException,
        ParameterValueException,
        Exception
```

Code 13-22: The addRelatedPartyRole Method

- **getAllNotes**—Returns all notes.

```
public Vector getAllNotes(
    String username,
    String authentication,
    PartyRole theB0)
    throws ParameterValueException, Exception
```

Code 13-23: The getAllNotes Method

- **getAllRelatedPartyRoles**—Returns all related party role objects for a specified role name. The `getAllRelatedPartyRoles` method has two signatures, shown in [Code Sample 13-24](#) and [Code Sample 13-25](#). They return null where there are no related `PartyRoles` for the `PartyRole` object passed in.

```
public Vector getAllRelatedPartyRoles(
    String username,
    String authentication,
    PartyRole theB0,
    Role relatedRole)
    throws
        InvalidRoleException,
        InvalidRelatedRoleException,
        ParameterValueException,
        ParameterRequiredException,
        Exception
```

Code 13-24: The getAllRelatedPartyRoles Method Signature 1

```
public Vector getAllRelatedPartyRoles(  
    String username,  
    String authentication,  
    PartyRole theB0,  
    Role theOtherRole,  
    CommonObject theCommonObject)  
    throws  
        InvalidRoleException,  
        ParameterValueException,  
        ParameterRequiredException,  
        BusinessServiceException,  
        Exception
```

Code 13-25: The *getAllRelatedPartyRoles* Method Signature 2

- **getPartyRolesByPartyIds**—Returns the *PartyRole* objects based on the specified party identifiers. There are two signatures for this method.

```
public Vector getPartyRolesByPartyIds(  
    String username,  
    String authentication,  
    Vector partyIds)  
    throws  
        UnexpectedMultipleRecordsException,  
        LockUnavailableException,  
        BusinessServiceException,  
        DataSourceException,  
        ParameterValueException,  
        Exception
```

Code 13-26: The *getPartyRolesByPartyIds* Method Signature 1

```
public Vector getPartyRolesByPartyIds(  
    String username,  
    String authentication,  
    Vector partyIds,  
    String roleType)  
    throws  
        UnexpectedMultipleRecordsException,  
        LockUnavailableException,  
        BusinessServiceException,  
        DataSourceException,  
        ParameterValueException,  
        Exception
```

Code 13-27: The *getPartyRolesByPartyIds* Method Signature 2

- **getRelatedPartyRole**—Returns access to a related party role object given its role relative to this object.

```
public PartyRole getRelatedPartyRole(  
    String username,  
    String authentication,  
    PartyRole theB0,  
    Role relatedRole)  
    throws  
        MoreThanOneContainedException,  
        TypeNotSupportedException,  
        InvalidRoleException,  
        ParameterValueException,  
        Exception
```

Code 13-28: The *getRelatedPartyRole* Method

- **removeNote**—Removes a note.

```
public void removeNote(  
    String username,  
    String authentication,  
    PartyRoleNote theNote)  
    throws ParameterValueException,  
    Exception
```

Code 13-29: The removeNote Method

- **removeRelatedPartyRole**—Removes the relationship between two party roles.

```
public void removeRelatedPartyRole(  
    String username,  
    String authentication,  
    PartyRole theB0,  
    PartyRole relatedRole)  
    throws  
        InvalidRelatedRoleException,  
        ObjectNotFoundException,  
        ParameterValueException,  
        Exception
```

Code 13-30: removeRelatedPartyRole Method

Using the Manager Interface

The **Manager** interface contains the **create** method, which creates a new **CommonObject** or **CommonGroup**, or suitable derivation, representing the specified type. The **create** method throws a **TypeNotSupported** exception when requested to create an unsupported type of object.

```
public CommonObject create(
    String username,
    String authentication,
    CommonObject theBO)
    throws
        TypeNotSupportedException,
        InvalidInitializationTypeException,
        InvalidInitializationValueException,
        IsDuplicateException,
        ParameterValueException,
        ParameterRequiredException,
        Exception
```

Code 13-31: The create Method

Using the PartyManager Interface

The **PartyManager** interface offers the capability to create objects derived from **Node** and **Party**, such as **Person** and **Organization** objects.

The **PartyManager** object extends the **Manager** interface, and consists of these methods:

- **createParty**—Initializes contact information for a party.

```
public Party createParty(
    String username,
    String authentication,
    Party theBO,
    Vector contactInfo)
    throws
        TypeNotSupportedException,
        InvalidInitializationTypeException,
        InvalidInitializationValueException,
        IsDuplicateException,
        ParameterValueException,
        ParameterRequiredException,
        Exception
```

Code 13-32: The createParty Method

- **getSupportedContactTypes**—Returns a sequence of strings representing all of the contact types with which a party can be associated.

```
public Vector getSupportedContactTypes(
    String username,
    String authentication)
    throws Exception
```

Code 13-33: The getSupportedContactTypes Method

- **getSupportedParties**—Returns the types of parties that this manager is capable of creating.

```
public Vector getSupportedParties(
    String username,
    String authentication)
    throws Exception
```

Code 13-34: The *getSupportedParties* Method

- **removeParty**—Removes the **Party** object and the party's associated roles.

```
public void removeParty(
    String username,
    String authentication,
    Party theBO)
    throws
        ParameterValueException,
        Exception
```

Code 13-35: The *removeParty* Method

- **updateParty**—Updates the **Party** object and its contact information using the specified information.

```
public Party updateParty(
    String username,
    String authentication,
    Party theBO,
    Vector contactInfo)
    throws Exception
```

Code 13-36: The *updateParty* Method

Using the RoleManager Interface

The **RoleManager** interface offer the ability to create and manage **Role** objects. The **RoleManager** object extends the **Manger** interface, and consists of the following methods:

- **createRole**—Given a primary object and the requested type, the **createRole** method creates a **CommonObject** object and associates it with the primary object.

```
public Role createRole(
    String username,
    String authentication,
    Role theBO,
    CommonObject thePrimaryObject)
    throws
        IsDuplicateException,
        TypeNotSupportedException,
        ParameterValueException,
        Exception
```

Code 13-37: The *createRole* Method

- **getSupportedRoles**—Returns all role types, as a string, that this manager is capable of creating.

```
public Vector getSupportedRoles(
    String username,
    String authentication)
    throws Exception
```

Code 13-38: The *getSupportedRoles* Method

Using the PartyRoleManager Interface

The `PartyRoleManager` interface extends the `RoleManger` interface, and contains the `getSupportedPartyRoles` method that returns all party role types, as a string, that this manager is capable of creating.

```
public Vector getSupportedPartyRoles(  
    String username,  
    String authentication)  
    throws Exception
```

Code 13-39: The `getSupportedPartyRoles` Method

Using the RelationshipManager Interface

The `RelationshipManager` interface offer the ability to create and manage relationships and roles. The `RelationshipManager` object extends the `Manger` interface, and consists of the following methods:

- **createRelationship**—Creates a new relationship object from the specified role objects and relationship type.

```
public Relationship createRelationship(  
    String username,  
    String authentication,  
    Relationship theB0,  
    Role primeRole,  
    Role relatedRole)  
    throws  
        RoleTypeErrorException,  
        UnknownRoleException,  
        InvalidRoleException,  
        ParameterValueException,  
        Exception
```

Code 13-40: The `createRelationship` Method

- **getSupportedRelationships**—Returns all of the types of relationships that this manager is capable of creating.

```
public Vector getSupportedRelationships(  
    String username,  
    String authentication)  
    throws Exception
```

Code 13-41: The `getSupportedRelationships` Method

- **getSupportedRolesForRelationship**—Returns the allowed role names based on a relationship type for a specified relationship.

```
public Vector getSupportedRolesForRelationship(  
    String username,  
    String authentication,  
    Relationship theB0)  
    throws Exception
```

Code 13-42: The `getSupportedRolesForRelationship` Method

EXAMPLE SCENARIOS

This section describes the use of the Party Management Facility for the following common scenarios:

- “Creating a Customer” on page 295
- “Establishing Marriage Between Two People” on page 295
- “Converting a Prospect to a Customer” on page 296

Creating a Customer

Here is a sample execution flow for creating a customer:

1. The client application calls the `PartyRoleClientAgent.createRole` and passes the required parameters. Refer to the `createRole` signature on [page 293](#).
2. The `PartyRoleClientAgent` calls `PartyRoleService.createRole`, and passes the party `Person` and role `Customer`.
3. The `PartyRoleService` uses the Business Object Behavior (BOB) Factory to determine the behavior for `Customer` business object.
4. The `PartyRoleService` calls the `CustomerBOB` to create a role `Customer` for the specified party `Person`.

The `Customer` business object is returned to the client, and a customer number has been assigned to the person.

Establishing Marriage Between Two People

Here is a sample execution flow for establishing a marriage between two people:

1. The client application calls `PartyRoleService.createRelationship` and passes two `Roles` (`Husband` and `Wife`) and a relationship (`Marriage`).
2. The `PartyRole` Service calls the `MarriageBOB` (behavior) to create a `Marriage` relationship between the two `Role` objects.
3. The `MarriageBOB` creates the `Marriage` relationship between the two roles and returns the `Marriage` business object to client.

Converting a Prospect to a Customer

Here is a sample execution flow for establishing a converting a prospect to a customer:

1. The client application calls `ProspectService` to convert the role `Prospect` to a role `Customer`.
The client passes the `Prospect` role to `ProspectService`.
2. The `ProspectService` calls the `ProspectBOB` (behavior) to remove the `Customer` role and add the new `Prospect` role.
The `ProspectService` passes the `Prospect` role to `ProspectBOB`.
3. The `ProspectBOB` invokes `creates a Customer`.
4. The `ProspectBOB` calls `ProspectBOB.complete` method to update the `Prospect` role to indicate that the it has become a `Customer`.
The `Customer` business object is returned to client.

Customizing the PartyRole Service

This chapter describes moving the business logic from the business service to the business object behavior and the customization associated with that move.

Note: This strategy is currently used only for PartyRole customizations.

BUSINESS OBJECTS AND BUSINESS OBJECT BEHAVIOR

Note: The concept of the Business Object Behavior (BOB) for the PartyRole service, described in this chapter, is different from the Extended Persistence BOB described in [Chapter 10](#). Here, the BOB does not contain the business object, but rather has logic and methods that act on the business object.

Chordiant Business Objects (BO) typically represent modeled entities within the domain of your organization. Business objects can also have associated behavior which defines the operations that can be performed on the business object data.

You can generate BOB classes when generating the business object classes using Rational Rose and the Business Component Generator. Using business object behavior enables you to decouple business logic implementation with the associated data, resulting in an application environment that is more easily customizable, more robust, and is enabled for multiple-channel accessibility across the enterprise.

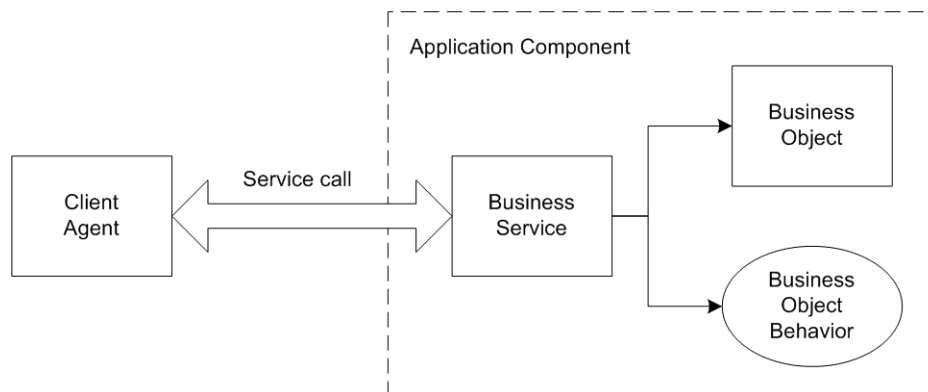


Figure 14-1: Application Component Architecture for the PartyRole Service

A business object behavior object contains behavior that is specific to the associated BO data. Many methods on BOB objects accept a business object as an input parameter.

Note: Business object behavior is associated only with business objects, not with services.

During the modeling process, you can define the attributes within an object that are to be persistent. Calls to the data accessor are then coded directly into the BOB object. Within the business object, you can declare any number of APIs with public visibility. These objects will have corresponding BOB objects generated for them.

A business object behavior object is deployed through the Foundation Server, thereby preventing applications from accessing restricted behavior and providing the same behavior to all channels. BOB classes also typically implement persistence directly, employing the data accessor components.

BUSINESS SERVICES

Services operate at a higher level than business objects, and typically act as a controller, calling the methods that implement business object behaviors. Services often implement domain-specific behavior instead of the object-specific behavior, and generally manage a collection of related business objects.

Services can also perform caching and provide connectivity to other systems within Foundation Server and with third-party products. Services centralize portions of the domain API. Without services, the domain APIs could be distributed across potentially hundreds of BOB objects instead of a handful of services.

When a service requires a BOB object, it uses an Object Factory method on the Resource Manager to vend a business object behavior instance. [Figure 14-2](#) illustrates the relationship between a service and a business object behavior object.

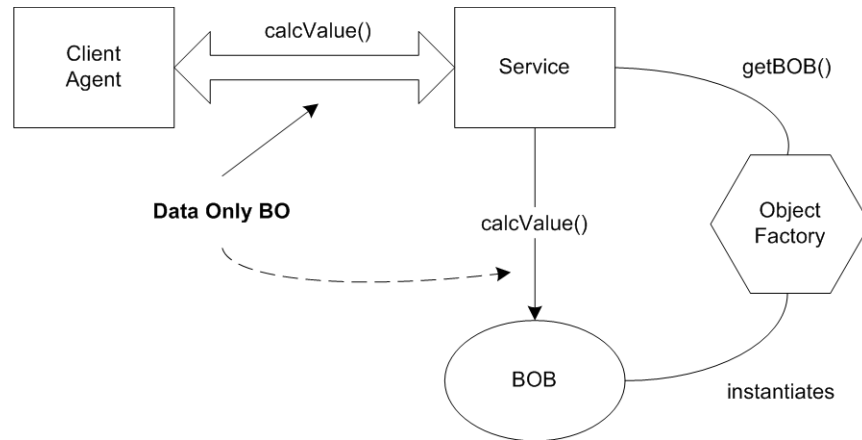


Figure 14-2: Service Object and Business Object Behavior Object Relationship

The Resource Manager is discussed in the *Chordiant 5 Foundation Server Developer's Guide*.

The BOB object is a suitable home for persistence behavior. [Figure 14-3](#) illustrates a sequence diagram in which the BOB implements persistence.

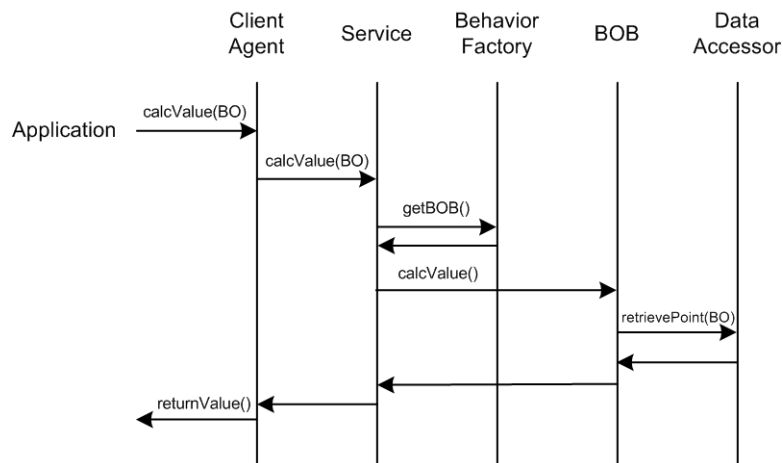


Figure 14-3: Service and BOB Sequence Diagram

INTEGRATING BUSINESS OBJECT BEHAVIOR

You are not required to integrate business object behavior into your Foundation Server environment. In fact, the PartyRole Service is the only Chordiant business service that provides this type of business object behavior. Business services that do not use BOB can co-exist with BOB-enabled services without problem. Likewise, since client applications and peer services both interact with business services through client agents, whether functionality is implemented through business object behavior is transparent to the calling entity.

Customizing business object behavior objects for the PartyRole Service uses the same general customization guidelines as other customizations:

1. Subclass the original entity.
2. Add, override, or overload specific attributes and behavior within the derived object.

Business Object Behavior Considerations

The business object behavior typically implements calculations and other data manipulation on business objects, as well as performing persistence operations. The specific implementation of your business object behavior is necessarily domain specific.

Take note of the following points about BOB objects:

- Do not have BOB objects maintain state using the Resource Manager or any other resource. BOB objects should only read resource information.
- You can, however, have BOB objects manage data in a cache, since this information is typically an extension of data in a database.
- Use the BOB object to implement as much of the object-specific behavior as possible, and thereby avoid implementing helper objects.
- Business object behavior objects should not contain setup methods.
- Business object behavior objects should maintain a reference to a Resource Manager. This is done automatically by the factory methods in the BusinessObjectResourceManager. Be sure your BOB object's constructor calls its parent's constructor.
- Usage of BOB objects should not cross domain boundaries. For example, Product Service should not call the BOB Factory to obtain a Customer BOB. BOBs can call other BOBs within the same domain. If a BOB requires operations outside of its own domain, it should use the client agent corresponding to the other domain.

Customizing Business Object Behavior

This section describes how to customize business object behaviors. For information on related customizations, refer to:

- [“Modifying Service Framework Components” on page 46](#)
- [“Customizing Business Objects” on page 165](#)

You can customize business object behavior in the PartyRole Service to add or modify methods. You might want to do this in the following situations:

- To manipulate new or existing data and object attributes
- To update the capabilities of an existing methods (behavior)
- To add new behavior to an existing object

To customize business object behavior in the PartyRole Service:

1. Subclass the business object to which you want to add, or modify, behavior.

You can use Rational Rose to update the model for the business object. As with any customization, you should generally derive a new object instead of modifying an existing object directly.

2. In Rational Rose, define which class you are overriding. This will appear as an **override** metadata tag in the CMI file.
3. Add, or override, the method in the derived business object using the modeling tool.

[Figure 14-4](#) illustrates adding a new attribute and a new method (behavior) to a derived object.

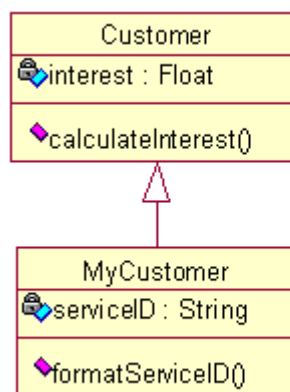


Figure 14-4: Adding Behavior and an Attribute to a Derived Business Object

4. Use the Business Component Generator to automatically generate the components listed below. For instructions, refer to [Chapter 3, “Using the Business Component Generator”](#).

Note: To use the Business Component Generator, you must create a descriptor file. Refer to [“Creating the Descriptor File” on page 16](#) for details.

- Business Object Class
- Business Object Criteria Class
- Data Accessor Class
- Business Object Behavior Class (skeleton)
Only used with the PartyRole Service

The skeleton BOB class has empty methods for all methods defined in Rose. For example, `myCustomerBehavior` will have a single method, `formatServiceID()` and extend from `CustomerBehavior`.

The business services use a factory to obtain a BOB. If the input business object is a customized BO, the factory will return a customized BOB.

Note that when you add new behavior, you must create an access point for the new API you created. You might choose to:

- add a new API to your customized, subclassed service, or
- override an existing service or BOB API and call to your new API from there.

If you add a new API to your customized service, you will have to regenerate the client agent and service code for it.

CUSTOMIZATION EXAMPLE: PMFCUSTOMERSERVICE

In this release, we have included a customization of the **PartyRole** service called **PmfCustomerService**. *Pmf* stands for Party Management Facility, which is an OMG standard.

Figure 14-5 shows the relationship between **PmfCustomerService**, **PartyRoleService**, their client agents, and the client agents for past releases.

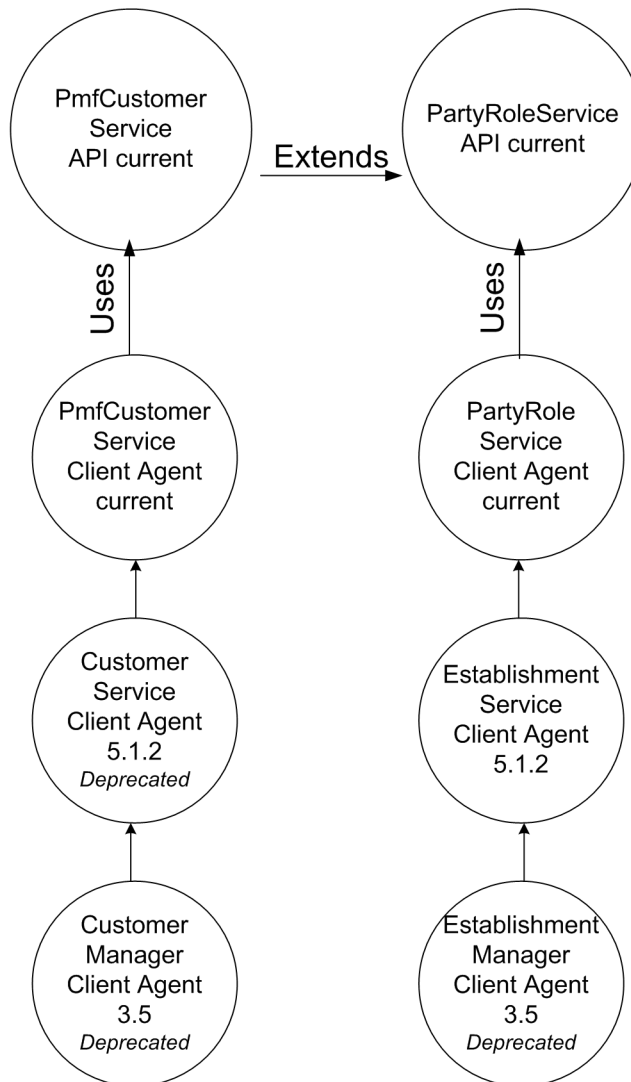


Figure 14-5: *PmfCustomerService* and *PartyRoleService*

The **PmfCustomerService** extends the **PartyRole** service to specialize in customer issues. Before you begin your own extensions to the **PartyRole** service, we recommend that you look at the code for the **PmfCustomerService** to see how the customization was done.

Customization Example: PmfCustomerService

Metadata

Metadata is data about data. Metadata is typically used to describe other data, however, the metadata that applies to one application can also serve as the data for another application.

Metadata can include information such as how data is formatted, the meaning applied to the data in various contexts, and how the data is to be manipulated. Metadata is essential for understanding and interpreting information stored in advanced data systems.

You can use metadata to describe various components of a system, with each component relying on its own metadata. For example, the Chordiant Rule System and Chordiant Workflow System each uses separately defined metadata pertinent to the particular system.

Likewise, the Chordiant system uses specific Business metadata to capture information about the data pertaining to applications.

Note that while the metadata can be completely different for each component within the system, all metadata is stored in XML format.

Metadata is entered in a model within Rational Rose, as shown in the previous chapters. (Refer to [“Specifying Persistence Metadata” on page 173](#) and [“Specifying Extended Persistence Information” on page 220](#) for examples.) From the Rational Rose model, you can export an industry-standard XMI file, a XML file which describes the model. Chordiant uses stylesheets to transform the XMI file into a Chordiant-proprietary CMI or SMI file, which is then used by the Business Component Generator to create business components.

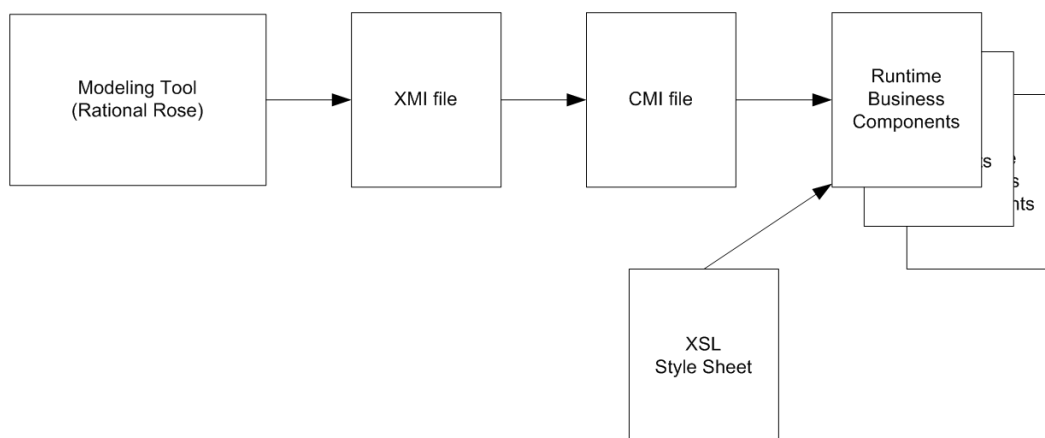


Figure 15-1: Metadata from Model to Business Components

UNDERSTANDING THE METADATA FORMAT

You store metadata information for Foundation Server in a Chordiant Metadata Information (CMI) file. A CMI file is a text file that contains the following type of information:

- **Class-level metadata**

Identified in the CMI file using the `<class>` `</class>` tags, class-level metadata includes information about the Java class.

- **Attribute-level metadata**

Identified in the CMI using the `<attribute>` `</attribute>` tags, attribute-level metadata includes any other information that might be required by the Business Component Generator.

Various components within Foundation Server will typically require different information to be associated with the class. For example, Persistence Server stores information such as the table name, the logical type of the field, and primary key information using attribute-level metadata.

- **Operation-level metadata**

Identified in the CMI file using the `<operation>` `</operation>` tags, operation-level metadata includes information about the defined methods associated with a business object.

[Code Sample 15-1](#) shows the structure of the CMI file.

```
<?xml version="1.0" ?>
  <root>
    <package>
      <name>test.jx.simple</name>
      <class>
<!--
        class-level metadata -->

        <attribute>
<!--
          attribute-level metadata -->
        </attribute>

        <attribute>
<!--
          attribute-level metadata -->
        </attribute>

        <operation>
<!--
          operation-level metadata -->
        </operation>

        <operation>
<!--
          operation-level metadata -->
        </operation>

      </class>
    </package>
  </root>
```

Code 15-1: Structure of a CMI File

You can choose any arbitrary name for the file, and store it anywhere within your system. Once created, you specify the name and location of a CMI file as a parameter in the component configuration file.

Note: Case is very important when specifying tag names in the CMI file. Always use the case described in this document. Substituting `<Class>` for `<class>`, for example, will cause the processing of your CMI file to fail.

BUSINESS METADATA

Business metadata is data about business data. Business metadata defines business objects and their attributes. For example, business metadata can include information about the format of a customer account number, or a rule specifying the minimum or maximum balance of an account.

Business metadata can also include a description of where the data resides within an information system.

Note: Business metadata can include any information about business data itself, however, it does not include descriptions of business behavior.

Exploring the Core Business Metadata

Chordiant 5 Foundation Server includes the following information in the CMI file to capture the Core Business metadata:

- **Package name**—The name of the Java package.

```
<package><name>package_name</name></package>
```

- **Class name**—The name of the Java class.

```
<class><name>class_name</name></class>
```

- **Parent Class**—The name of the parent class object.

You must include the package as part of the parent class name, for example, `com.chordiant.test.baseclass`.

```
<parentClass>parent_class_name</parentClass>
```

- **Abstract**—Specifies whether the class is abstract.

```
<isAbstract>true|false</isAbstract>
```

- **Override**—Specifies the fully-qualified name of the class that this class overrides, if applicable. Be sure to include the package as part of the class name.

```
<override>overridden_class_name</override>
```

- **Javadoc**—Specifies Javadoc for the object.

```
<javadoc> /** descriptive text goes here */ </javadoc>
```

- **Attribute**—Any value attribute required by the application.

In defining attributes, you need to include the name of the attribute, the Java type, and the database multiplicity.

```
<attribute> </attribute>
```

- **Name**—The name of the attribute.

```
<name>attribute_name</name>
```

- **Java type**—The fully-qualified Java type of the attribute, such as `java.lang.String` or `java.lang.Integer`.

```
<javaType>java_type</javaType>
```

- **Multiplicity**—The relationship between the attribute and the associated object, encoded as 1..n. For example, in the case of the relationship between a customer object and associated aliases, you might specify 1..3 to indicate that a customer can have as many as three potential aliases.

```
<multiplicity>1..n</multiplicity>
```

- **Operation**—Any operations (behavior) defined associated with the business object.

In defining operations, you need to include the name of the operation, the visibility, associated Javadoc, and whether the operation is static or final.

```
<operation> </operation>
```


- **Name**—The name of the method implementing the operation (behavior).
`<name>doOperation</name>`
- **Visibility**—Specifies the visibility of the method implementing the operation (behavior).
`<visibility>public</visibility>`
- **Static**—Specifies whether the method implementing the operation (behavior) is static (a class method).
`<static>true|false</static>`
- **Final**—Specifies whether the method implementing the operation (behavior) is defined as final (preventing overrides).
`<final>true|false</final>`

Figure 15-2 illustrates a CMI file containing only the Core Business metadata.

```
<?xml version="1.0" ?>
<root>
  <package>
    <name>test.jx.simple</name>
    <class>
      <name>Thinger</name>
      <javaDoc>/**/</javaDoc>
      <parentClass>Object</parentClass>
      <isAbstract>false</isAbstract>
      <override></override>
      <attribute>
        <name>Type</name>
        <javaDoc>/**/</javaDoc>
        <javaType>java.lang.String</javaType>
        <multiplicity>1..1</multiplicity>
      </attribute>
      <operation>
        <name>doOperation</name>
        <visibility>/**/</visibility>
        <javaDoc>/**/</javaDoc>
        <Static>java.lang.String</Static>
        <Final>true</Final>
      </operation>
    </class>
  </package>
</root>
```

Code 15-2: Sample CMI File with Core Business Metadata

Note: There are additional metadata tags corresponding to extended persistence metadata. Refer to [“Extended Persistence Tags” on page 318](#) for details.

Persistence Metadata

The persistence metadata defines information required by Persistence Server to interact with arbitrary data store systems. As part of the persistence metadata, Enterprise Integration supports two well-known tags, enabling you to encode database connector information for the following systems:

- RDBMS using SQL
- WebSphere MQ

SQL Persistence Tags

Use the SQL persistence tags to define the information you need to access in a Relational Database Management System using SQL. [Table 15-1](#) describes the SQL persistence tags available in the CMI file.

CMI TAG	DESCRIPTION
<DSN> </DSN>	<p>Specified at the class level, the data source name, used by the application server to establish a connection to the database. It specifies the resource name in the XML configuration file.</p> <p>Note: The CMI file and XML configuration files provided by Chordiant specify "chordiantXAds" as the data source name. Generated persistence components (like the Data Accessor) will have this name in the code. If you want to use a different data source name, you must change the CMI file and regenerate the persistence components. You must also make sure that the resource name in the configuration files of the production system matches the DSN specified in the persistence components.</p> <p>Example: chordiantXAds</p> <p>For more information on DSN, refer to "Specifying DSN" on page 313.</p>
<rdBPhysicalName> </rdBPhysicalName>	As class-level metadata, the tag defines the name of the database table. Example: CUSTOMER
<persistentType> </persistentType>	<p>The type of back end data system being used. Possible values are: <i>mqseries</i>, <i>mqseriesxml</i>, <i>oracle8</i>, and <i>db2udb</i></p> <p>Note: If you will be using a DB2UDB database with the Chordiant-provided business services, be sure to use the DB2UDB-specific CMI file (<i>chordiantcmi_db2udb.xml</i>) to generate the persistence components. If you are using Oracle, use the <i>chordiantcmi.xml</i> file.</p>

Table 15-1: SQL Persistence Tags

CMI TAG	DESCRIPTION
<XMLType> </XMLType> <i>* for mqseriesxml persistent type</i>	Specifies the type of the XML document generated. This document type must match the type expected by the receiving application. Possible values: <ul style="list-style-type: none"> • Literal • Encoded
<XMLRootName> </XMLRootName> <i>* for mqseriesxml persistent type</i>	(Default) root The high level node of the XML document.
<XMLObjectName> </XMLObjectName> <i>* for mqseriesxml persistent type</i>	Specifies the name of a particular object within the XML file. XML files can contain multiple objects.
<XMLNamespacePrefix> </XMLNamespacePrefix> <i>* for mqseriesxml persistent type</i>	(Optional) Specifies the prefix to use to create a fully qualified name space within the XML document.
<LockStrategy> </LockStrategy>	Defines the locking mechanism for the object. Possible values: <ul style="list-style-type: none"> • pessimistic—Establishes an exclusive lock on the record • optimistic—Establishes a non-exclusive (shared) lock on the record • none—no locking mechanism
<LockField> </LockField>	Specifies that an attribute is used for locking for either optimistic or pessimistic locking Possible values: true or false.
<WherePrefix> </WherePrefix>	A full conditional statement, using table.column notation, to specify the conditional relating identifiers for a join operation.
<rdbPhysicalName> </rdbPhysicalName>	As attribute-level metadata, this tag specifies the name of the column in the table.
<rdbLogicalType> </rdbLogicalType>	The data type of the column, as defined in the database.
<rdbSize> </rdbSize>	The size of the column in the database.
<rdbDigits> </rdbDigits>	The number of digits following the decimal point, as defined in the database.

Table 15-1: SQL Persistence Tags (Continued)

CMI TAG	DESCRIPTION
<rdBNotNull> </rdBNotNull>	<p>Specifies whether the column can assume a null value or not.</p> <p>Possible values:</p> <ul style="list-style-type: none"> • true—Indicates that the column value cannot have a null value • false—Indicates that the column value can include the null value
<rdBPrimaryKey> </rdBPrimaryKey>	<p>Indicates whether the column serves as the primary key for the table.</p> <p>Note: You might have more than one primary key per table. Each primary key must have the <code>rdBPrimaryKey</code> set to "true" and have an associated <code>rdBPrimaryKeyGenerationType</code> (see below).</p>
<rdBPrimaryKeyGenerationType> </rdBPrimaryKeyGenerationType>	<p>Indicates whether the system should autogenerate the primary key.</p> <p>Possible values:</p> <ul style="list-style-type: none"> • auto—Autogenerate the primary key for the table using the Chordiant 5 Foundation Server GUID (For more information on GUIDs, refer to the <i>Chordiant 5 Foundation Server Developer's Guide</i>.) • none—Do not autogenerate the primary key

Table 15-1: SQL Persistence Tags (Continued)

[Code Sample 15-3](#) illustrates a CMI file containing Core Business metadata together with a definition of associated SQL persistence metadata.

```
<?xml version="1.0" ?>
<root>
  <package>
    <name>test.jx.simple</name>
    <class>
      <name>Thinger</name>
      <parentClass>Object</parentClass>
      <DSN>chordiantXAds</DSN>
      <rdbPhysicalName>THINGER</rdbPhysicalName>
      <persistentType>oracle8</persistentType>
      <LockStrategy>pessimistic</LockStrategy>
      <WherePrefix>Cust.cust_id=Profile.prof_id</WherePrefix>
      <attribute>
        <name>Type</name>
        <javaType>java.lang.String</javaType>
        <multiplicity>1..1</multiplicity>
        <rdbPhysicalName>locktokentext</rdbPhysicalName>
        <rdbLogicalType>VARCHAR</rdbLogicalType>
        <rdbSize>80</rdbSize>
        <rdbDigits>0</rdbDigits>
        <rdbNotNull>true</rdbNotNull>
        <rdbPrimaryKey>false</rdbPrimaryKey>
      </attribute>
      <attribute>
        <...>
      </attribute>
    </class>
  </package>
</root>
```

Code 15-3: Sample CMI File Including Core Business and SQL Persistence Metadata

Specifying DSN

The data source name (DSN) in Chordiant is the resource name known to the Chordiant Resource Manager. It is not the same as the JNDI name known to the application server, it is a layer on top of the application server. Coincidentally, the out-of-the-box Chordiant DSN is usually the same as the JNDI name.

This DSN is specified in the CMI file, in the XML configuration file, and is part of the generated Data Accessor. These three settings are interrelated (as shown in the description of [Code Sample 15-4 on page 314](#)) and should not be altered.

Note: If the DSN in the CMI file does not exactly match the resource name in the XML configuration file, you will have problems finding your data sources.

This example shows why it is important for the DSN in the CMI file to match the resource name in the XML configuration file.

1. When the system starts up and reads from the XML configuration files, the resource name is read into the Resource Manager.
2. The Data Accessor is generated from a CMI file, so it has the resource name it needs.
3. When the generated Data Accessor is called, it calls the Resource Manager to get the data source it requires, using the resource name.
4. The Resource Manager looks up the value and returns the appropriate data source.

For example, the value of the resource name in the `{component}.xml` configuration file must exactly match the DSN in the CMI file, for example, the CMI code shown on [page 313](#). [Code Sample 15-4](#) shows this section for the `GenericService.xml` file.

```
<Section>com.chordiant.bc.services.GenericService
  <Tag>ResourceName
    <Value>chordiantXAds</Value>
  </Tag>
  ...
</Section>
...
<Section>com.chordiant.bc.services.GenericService.chordiantXAds
  <Tag>ResourceType
    <Value>JNDI</Value>
  </Tag>
  <Tag>ResourceValue
    <Value>&JXB_XA_DATASOURCE;</Value>
  </Tag>
</Section>
```

Code 15-4: Resource Name in the `GenericService.xml` File

MQ Persistence Tags

Use the MQ persistence tags to define the information you need to access in MQ Series data stores. [Table 15-2](#) describes the MQ persistence tags available in the CMI file.

CMI TAG	DESCRIPTION
<DSN> </DSN>	The data source name. This is the name of the section in the configuration file with connection data. The out-of-the-box Chordiant CMI file specifies "chordiantXAds" as the datasource name. Generated persistence components (like the Data Accessor) will have this name in the code. If you want to use a different data source name, you must change the CMI file and regenerate the persistence components. Refer to "Specifying DSN" on page 313 for more information.
<persistentType> </persistentType>	Use the value MQ Series for this field.

Table 15-2: MQ Persistence Tags

CMI TAG	DESCRIPTION
<XMLType> </XMLType> <i>* for mqseriesxml persistent type</i>	Specifies the type of the XML document generated. This document type must match the type expected by the receiving application. Possible values: <ul style="list-style-type: none"> • Literal • Encoded
<XMLRootName> </XMLRootName> <i>* for mqseriesxml persistent type</i>	(Default) root The high level node of the XML document.
<XMLObjectName> </XMLObjectName> <i>* for mqseriesxml persistent type</i>	Specifies the name of a particular object within the XML file. XML files can contain multiple objects.
<XMLNamespacePrefix> </XMLNamespacePrefix> <i>* for mqseriesxml persistent type</i>	(Optional) Specifies the prefix to use to create a fully qualified name space within the XML document.
<LockStrategy> </LockStrategy>	(Default) none Defines the locking mechanism for the object. Possible values: <ul style="list-style-type: none"> • none—Optimistic locking for MQ Series is not supported • pessimistic—Establishes an exclusive lock
<mqMessageModel> </mqMessageModel>	(Default) RequestReply The message model. Possible values: <ul style="list-style-type: none"> • RequestReply—Send a request and wait for a response • FireForget—Send a request with no reply expected
<mqmdencoding> </mqmdencoding>	Specifies the encoding format.
<mqGetMessageSize> </mqGetMessageSize>	The size of the response message. You need to specify this numeric field only if you expect response messages greater in size than 4096 bytes.
<mqmdencodingGet> </mqmdencodingGet>	Specifies the encoding used for processing response messages.
<mqmdformat> </mqmdformat>	(Default) MQC.MQFMT_STRING
<mqmdcharacterSet> </mqmdcharacterSet>	(Default) MQC.MQCCSI_DEFAULT

Table 15-2: MQ Persistence Tags (Continued)

CMI TAG	DESCRIPTION
<mqmdmessageType> </mqmdmessageType>	(Default) MQC.MQMT_REQUEST
<mqmdpriority> </mqmdpriority>	(Default) 8
<mqmdcorrelationId> </mqmdcorrelationId>	Set this property if you select a value other than the default for the mqmdformat property.
The following tags (mqStartingOffset through mqDateFormat) can be used for either attributes or aggregations	
<mqStartingOffset> </mqStartingOffset> * for either attributes or aggregations	Numeric offset of data in the message relative to zero.
<mqLength> </mqLength> * for either attributes or aggregations	Numeric length of the data in the message.
<mqAlignment> </mqAlignment> * for either attributes or aggregations	Alignment of the data in the message. Possible values: <ul style="list-style-type: none"> • Left • Right
<mqFillCharacter> </mqFillCharacter> * for either attributes or aggregations	Fill characters to add or remove from aligned data message. Possible values: <ul style="list-style-type: none"> • Space • Zero • None
<mqInput> </mqInput> * for either attributes or aggregations	Parse response messages using this attribute or aggregation. Possible values: <ul style="list-style-type: none"> • True • False
<mqOutput> </mqOutput> * for either attributes or aggregations	Build request messages using this attribute or aggregation. Possible values: <ul style="list-style-type: none"> • True • False

Table 15-2: MQ Persistence Tags (Continued)

CMI TAG	DESCRIPTION
<code><mqPrimaryKeyGenerationType></code> <code></mqPrimaryKeyGenerationType></code> <i>* for either attributes or aggregations</i>	<p>Automatically generate a unique nine digit key for this attribute or aggregation, which must be a <code>JavaType</code> of <code>java.lang.String</code> or <code>java.lang.Integer</code>. Possible values:</p> <ul style="list-style-type: none"> • Auto • None <p>One attribute (and no more) must have this tag set to Auto. Key generation only occurs in the <code>createPoint</code> or <code>createSet</code> methods.</p>
<code><mqDateFormat></code> <code></mqDateFormat></code> <i>* for either attributes or aggregations</i>	<p>Defines the data format of the data in the message if <code>javaType</code> is <code>java.util.Date</code>; otherwise specify None. Possible values:</p> <ul style="list-style-type: none"> • None • yyyyMMdd • MM/dd/yyyy

Table 15-2: MQ Persistence Tags (Continued)

Extended Persistence Tags

If you are using extended persistence in your application components, the following class-level metadata flags are available to you:

- **accessStrategy**—The access strategy tags contain tags for operations for which you are overriding the access strategy.

```
<accessStrategy> </accessStrategy>
```

The following tags are included within the access strategy markers:

- **operation**: Wraps around the name of the operations used for the override.

```
<operation>
```

- **name**: The name of the method using the access strategy override. If all methods are using the override, specify **All**.

```
<name>RetrieveAssociation</name>
```

- **useStrategy**: The fully-qualified name of the access strategy you are using for the override.

```
<useStrategy>com.chordiant.activity.accessstrategies.MyAccessStrategy</useStrategy>
```

- **useBehavior**—The fully-qualified name of the business object behavior to use in place of the default.

```
<useBehavior>
  <name>com.chordiant.activity.behaviors.MyCustomBehavior</name>
</useBehavior>
```

- **xrefTableName**—The name of the lookup table to use for inheritance

```
<xrefTableName>lookuptablename</xrefTableName>
```

- **typeValue**—Used for inheritance, this tag specifies which table contains the information for this subclass. Usually the name of the class, typed in all capital letters. This value will be returned when the **getType** method is called

```
<typeValue>ACTIVITY</typeValue>
```

- **typeField**—Used to support inheritance, if the **typeField** is set to **true**. Used in conjunction with the name of an attribute.

```
<typeField>true|false</typeField>
```

- **association**—This tag wraps around additional tags to show association information.

```
<association></association>
```

The following tags are included within the association markers:

- **name**: The name of the association between the two classes, usually written in all capital letters.

```
<name>DETAILS</name>
```

- **type**: The type of association, usually **foreignKey**. This is related to specifying a stereotype (refer to [page 240](#)).

```
<type>foreignKey</type>
```

- **className:** The fully-qualified name of the associated class.

```
<className>com.chordiant.delivery.businessclasses.  
servicehistory.ActivityDetails</className>
```

- **foreignKeyAttribute:** The name of the foreign key from the associated class.

```
<foreignKeyAttribute>activityId</foreignKeyAttribute>
```

- **foreignClassAttribute:** Specifies the remote (associated) object to view.

```
<foreignClassAttribute>activityDetailsId</foreignClassAttribute>
```

Code Sample 15-5 is a section of the CMI file that shows all of the extended persistence tags.

```
<class>  
  <name>Activity</name>  
  <javaDoc>/**</javaDoc>  
  <parentClass>com.chordiant.bd.baseBusinessClasses.  
    CorporateBusinessClass</parentClass>  
  <isAbstract>false</isAbstract>  
  <rdbPhysicalName>activity</rdbPhysicalName>  
  <WherePrefix></WherePrefix>  
  <DSN>chordiantXAds</DSN>  
  <override></override>  
  <persistentType>oracle8</persistentType>  
  <LockStrategy>optimistic</LockStrategy>  
  <typeValue>ACTIVITY</typeValue>  
<accessStrategy>  
  <operation>  
    <name>All</name>  
    <useStrategy>com.chordiant.activity.accessstrategies.MyAccessStrategy</useStrategy>  
  </operation>  
</accessStrategy>  
<accessStrategy>  
  <operation>  
    <name>RetrieveAssociation</name>  
    <useStrategy>com.chordiant.activity.accessstrategies.  
      DefaultAccessStrategy</useStrategy>  
  </operation>  
</accessStrategy>  
  <useBehavior>  
    <name>com.chordiant.activity.behaviors.MyCustomBehavior</name>  
  </useBehavior>  
  <xrefTableName>activitytypelookuptable</xrefTableName>  
  <accessStrategy>  
    <operation>  
      <name>All</name>  
      <useStrategy>com.chordiant.activity.accessstrategies.  
        MyAccessStrategy</useStrategy>  
    </operation>  
  </accessStrategy>
```

Code 15-5: Sample CMI File Showing All Extended Persistence Tags

```
<accessStrategy>
  <operation>
    <name>RetrieveAssociation</name>
    <useStrategy>com.chordiant.activity.accessstrategies.
      DefaultAccessStrategy</useStrategy>
  </operation>
</accessStrategy>
<association>
  <name>DETAILS</name>
  <type>foreignKey</type>
  <className>com.chordiant.delivery.businessclasses.servicehistory.
    ActivityDetails</className>
  <foreignKeyAttribute>activityId</foreignKeyAttribute>
</association>
```

Code 15-5: Sample CMI File Showing All Extended Persistence Tags (Continued)

SERVICE METADATA

You store metadata information for JX Services in a Chordiant Service Metadata Information (SMI) file. The SMI file is a text file that contains the following type of information:

- **Class-level metadata**

Identified in the SMI file using the `<class>` `</class>` tags, class-level metadata includes information about the Java class.

- **Attribute-level metadata**

Identified in the SMI file using the `<attribute>` `</attribute>` tags, attribute-level metadata includes any other information that might be required by the code generation tool.

- **Operation-level metadata**

Identified in the SMI file using the `<operation>` `</operation>` tags, operation-level metadata includes information about the methods within the service.

[Code Sample 15-6](#) shows the SMI file structure.

```
<?xml version ="1.0" ?>
  <root>
    <package>
      <name>test.jx.simple</name>
      <class>
        <!--      class-level metadata -->
          <attribute>
            <!--      attribute-level metadata -->
              </attribute>
            <attribute>
              <!--      attribute-level metadata -->
                </attribute>
            <operation>
              <!--      operation-level metadata -->
                <parameter>
                  <!--      parameter-level metadata -->
                    </parameter>
                </operation>
              </class>
            </package>
          </root>
```

Code 15-6: SMI File Structure

Notes: All services should derive from a base Chordiant service.

Case is very important when specifying tag names in the SMI file. Always use the case described in this document. Substituting `<Class>` for `<class>`, for example, will cause the processing of your SMI file to fail.

Chordiant 5 Foundation Server includes the following information in the SMI file to capture the metadata for services. Many of these tags are similar to those in the CMI file.

- **Package name**—The name of the Java package.

```
<package><name>package_name</name></package>
```

- **Class name**—The name of the Java class.

```
<class><name>class_name</name></class>
```

- **Parent Class**—The name of the parent class object.

You must include the package as part of the parent class name, for example, `com.chordiant.test.baseclass`.

The parent class is encoded using these tags, within the `<class></class>` tags.

```
<parentClass>parent_class_name</parentClass>
```

- **isService**—All services should have this value set to `true`. Business objects and other objects in the model do not have this tag at all.

The `isService` value is encoded using these tags, within the `<class></class>` tags

```
<isService>true</isService>
```

- **Javadoc**—Javadoc documentation associated with the class.

```
<javadoc>/** [text goes here] */</javadoc>
```

- **Attribute**—Any value required by the application.

In defining attributes, you need to include the name of the attribute and the Java type.

```
<attributes> </attributes>
```

- **name**—The name of the attribute.

```
<name>attribute_name</name>
```

There are two required attributes for service generation, based on transaction type. Their names are:

TX_SEMANTICS: value is `EJBBMT` or `EJBCMTRequired`

STUBTYPE: value is `EJBStub` or other value you specify

- **Java type**—The fully-qualified Java type of the attribute, such as `java.lang.String` or `java.lang.Integer`.

```
<javaType>java_type</javaType>
```

- **Operation**—Information describing the methods in the service.

In defining operations, you need to include the name of the operation (method), the visibility, Javadoc, the return type, and the parameters.

```
<operation> </operation>
```

- **Operation Name**—The name of the method in the service.

Method names are encoded using these tags, within the `<operation></operation>` tags.

```
<name>doGuitar</name>
```

- **Visibility**—Whether the method is public, private, or protected.

Visibility is encoded using these tags, within the `<operation></operation>` tags.

```
<visibility>public</visibility>
```

- **Javadoc**—Javadoc documentation associated with the method.

The Javadoc is encoded using these tags, within the `<operation></operation>` tags.

```
<javadoc>/** [text goes here] */</javadoc>
```

- **Return Type**—What the method returns (for example, an integer or string).

The Return Type is encoded using these tags, within the `<operation></operation>` tags.

```
<returnType>java.lang.String</returnType>
```

- **Parameter**—What the method takes as a parameter (for example, an integer or a string).

When defining a Parameter, you must define the Parameter Type and the Parameter Name.

```
<parameter></parameter>
```

- **Parameter Type**—The fully-qualified Java type of the parameter, such as `java.lang.String` or `java.lang.Integer`.

The Parameter Type is encoded using these tags, within the `<parameter></parameter>` tags.

```
<parameterType>java.lang.String</parameterType>
```

- **Parameter Name**—The name of the parameter.

The Parameter Name is encoded using these tags, within the `<parameter></parameter>` tags.

```
<parameterName>authenticationToken</parameterName>
```

[Code Sample 15-7](#) illustrates a sample SMI file.

```
<?xml version = '1.0' ?>

<root>
<package>
  <name>com</name>
</package>
<package>
  <name>com.chordiant</name>
</package>
<package>
  <name>com.chordiant.guitar</name>
  <class>
    <name>GuitarService</name>
    <parentClass>com.chordiant.services.BusinessDataServiceBaseClass</parentClass>
    <isService>true</isService>
    <javadoc>/**</javadoc>
  <operation>
```

Code 15-7: Sample SMI File

```

    <name>doString</name>
    <visibility>public</visibility>
    <javaDoc>/**/</javaDoc>
    <returnType>java.lang.String</returnType>
    <parameter>
      <parameterType>java.lang.String</parameterType>
      <parameterName>userName</parameterName>
    </parameter>
    <parameter>
      <parameterType>java.lang.String</parameterType>
      <parameterName>authenticationToken</parameterName>
    </parameter>
    <parameter>
      <parameterType>java.lang.String</parameterType>
      <parameterName>param0</parameterName>
    </parameter>
  </operation>
</operation>
<operation>
  <name>doGuitar</name>
  <visibility>public</visibility>
  <javaDoc>/**/</javaDoc>
  <returnType></returnType>
  <parameter>
    <parameterType>java.lang.String</parameterType>
    <parameterName>userName</parameterName>
  </parameter>
  <parameter>
    <parameterType>java.lang.String</parameterType>
    <parameterName>authenticationToken</parameterName>
  </parameter>
  <parameter>
    <parameterType>java.lang.String</parameterType>
    <parameterName>param0</parameterName>
  </parameter>
</operation>
<attribute>
  <name>attribute1</name>
  <javaDoc>/**/</javaDoc>
  <javatype>java.lang.String</javatype>
</attribute>
<attribute>
  <name>TX_SEMANTICS</name>
  <value>EJBCTRequired</value>
  <javaDoc>/**/</javaDoc>
  <javatype>java.lang.String</javatype>
</attribute>
<attribute>
  <name>STUBTYPE</name>
  <value>EJBStub</value>
  <javaDoc>/**/</javaDoc>
  <javatype>java.lang.String</javatype>
</attribute>
</class>
</package>

```

Code 15-7: Sample SMI File (Continued)


```
<package>
  <name>com.chordiant.services</name>
  <class>
    <name>BusinessDataServiceBaseClass</name>
    <parentClass>java.lang.Object</parentClass>
    <javaDoc>/***/</javaDoc>
  </class>
</package>
</root>
```

Code 15-7: Sample SMI File (Continued)

GENERATING CODE FROM CMI OR SMI

The Business Component Generator uses the CMI or SMI to create application components or service framework components.

For information on running the Business Component Generator, refer to [Chapter 3, “Using the Business Component Generator”](#).

Chordiant 5 Application Components

Chordiant Application Components provide a rich set of integrated business services that can be accessed from applications running on a multiple communication channels. This chapter provides a summary of the Chordiant Application Components (business services) included with Chordiant 5 Foundation Server.

Most of the application components provided with Chordiant 5 Foundation Server are also able to be deployed as web services. Refer to [Chapter 6, “Creating Web Services”](#) for more information.

Some APIs exist only for backward compatibility. They are listed at the end of this chapter.

For specific details on the Chordiant Application Components, refer to the Javadoc for the business services.

TASK DESCRIPTORS FOR BUSINESS PROCESS DESIGNER

The public APIs listed here are available as task descriptors within the Chordiant 5 Business Process Designer. Follow the links to find out more about these APIs.

- [“Account Service”](#)
- [“EBC Interaction Service”](#)
- [“Guide Service”](#)
- [“Location Service”](#)
- [“Offering Service”](#)
- [“Order Fulfillment Service”](#)
- [“Order Generation Service”](#)
- [“Order Tracking Service”](#)
- [“Party Role Service”](#)
- [“Product Service”](#)

ACCOUNT SERVICE

The Account Service offers facilities to authenticate, retrieve, create, and close an account. The Account Service offers the following additional functions:

- Returns all accounts for a business entity or customer
- Returns all accounts for a party
- Gets all parties for an account
- Gets all party roles for an account
- Adds a party to an account
- Generates a new pass code and updates the account with the new pass code
- Returns an account entry list for a given account
- Returns an account statement for a given account and date
- Updates an account with new data specified in the account data and account entry
- Negates an amount previously specified in the account entry
- Adds and updates an account entry memo for a given account entry

The main business area for the Account Service involves processing account and account transaction (account entry) information. An account can be related to one or more customers.

Figure 16-1 illustrates the hierarchy for account-related objects.

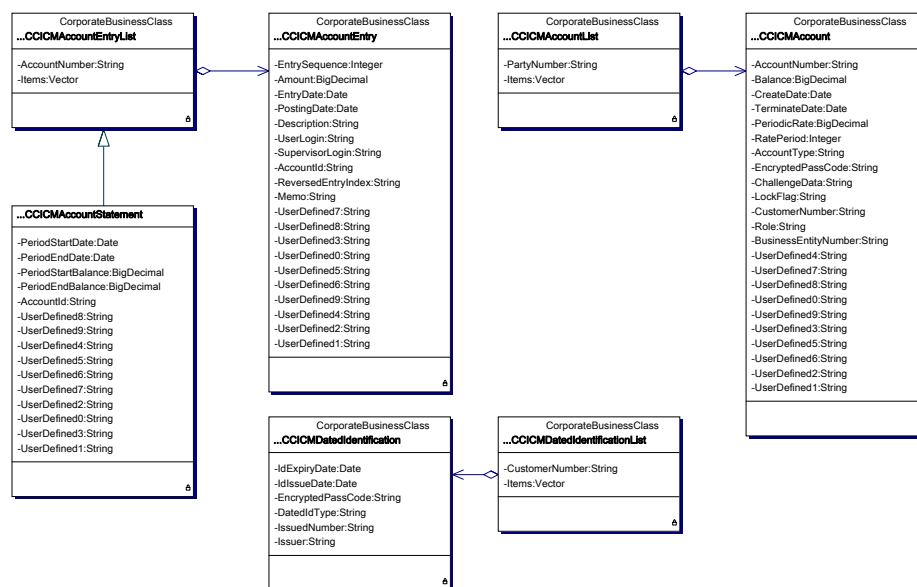


Figure 16-1: Account Objects

Customizable Components

The Account Service contains these customizable components:

Package `com.chordiant.bd.services`:

- `AccountService`

Package `com.chordiant.bd.clientAgents`:

- `AccountClientAgent`

Package

`com.chordiant.businessServices.customerInformationCorporateBusinessClasses`:

- `CCICMAccount`
- `CCICMAccountEntry`
- `CCICMAccountEntryList`
- `CCICMAccountList`
- `CCICMAccountStatement`
- `CCICMDatedIdentification`
- `CCICMDatedIdentificationList`

Package `com.chordiant.bd.numberGeneration`:

- `AccountNumberGenerator`
- `CustomerNumberGenerator`

Package `com.chordiant.bd.constants`

- `AccountServiceConstants`

The `AccountClientAgent` offers the following methods with which you can interact with the Account Service. All of these methods are available as task descriptors for the Business Process Designer.

- **addAccountEntry**—Adds an account entry to the persistent data store for the specified account. The method returns `true` when the account entry is added successfully.
- **addPartyToAccount**—Adds a party with a specified role to the persistent data store for the specified account. The method return is `void`.
- **authenticateAccount**—Authenticates the account by matching the pass code provided in the account against the stored pass code. The method returns `true` when the account has been authenticated, or when the stored pass code for the account is null.
- **closeAccount**—Closes an account, and adds an account entry specifying the closure date. Note that the account will not be closed if the outstanding balance is not zero.
- **createAccount**—Creates an account based on a customer number or business entity ID, and adds an account entry specifying the initial account transaction. The method returns the account object when the account has been created successfully; null in case of failure.
- **createAccount**—Creates an account based on a Party ID, and adds an account entry specifying the initial account transaction. The method returns the account object when the account has been created successfully; null in case of failure.
- **getAccount**—Returns an account based on the `AccountId` or the `AccountNumber`.
- **getAccountEntryList**—Returns the account entry list for a specified account.
- **getAccountStatementForDate**—Returns an account statement for a specified account and date.
- **getAllAccounts**—Returns all accounts for a specified customer.
- **getAllAccounts**—Returns all accounts for a specified business entity.
- **getAllAccounts**—Returns all accounts for a specified customer or business entity.
- **getAllAccountsForParty**—Returns all accounts for a party based on the object id.
- **getAllPartiesForAccount**—Returns all parties associated with the account.
- **getAllPartyRolesForAccount**—Returns all party roles associated with the account.
- **renewPassCode**—Generates a new pass code and updates the account with the new pass code.
- **reverseAccountEntry**—Reverses an account entry for a specified account (negates the amount previously specified in the account entry).
- **updateAccount**—Updates an account with new data specified in the account data and the account entry.
- **updateAccountEntryMemo**—Updates an account entry memo for a specified account entry.

Credit Card Account Service

The Credit Card Account Service is a subclass of the Account Service. This service extends the Account Service to handle logic and tasks associated with credit card disputes.

The Credit Card Account Service is provided for use in Chordiant's vertical applications. You can also use it or customize it for your own application.

For details on the Credit Card Account Service, refer to its Javadoc, located in package `com.chordiant.bd.services`.

DELIVERY SERVICE (STUB)

Chordiant has a stubbed service called Delivery service. It was originally intended to work with our order fulfillment service. However, it was not completed. The Harmony Bank application uses this service for check orders.

The Delivery service client agent has a single method, `processRequest`. Unlike other client agents, the Delivery service client agent does not have any regular business APIs. The service implementation has three methods with empty implementation.

- `approveDelivery`
- `cancelDelivery`
- `changeDeliveryMethod`

Package `com.chordiant.bd.constants`

- `DeliveryServiceConstants`

You are welcome to use this service as a starting point for creating your own delivery service.

EBC INTERACTION SERVICE

EBC Interaction Service captures information about customer interactions with an Enterprise Business Center (EBC). For each interaction, a case can be created to track requests and record information about the communication, its communication events, and communication participants.

You can query this information to determine the history of a customer's interactions with the EBC. The service is designed to be channel-independent. You can also manage request states to track the progress of fulfillment.

The main business area for the EBC Interaction Service involves processing Case, Request, and Communication information. [Figure 16-2](#) illustrates the hierarchy of EBC objects.

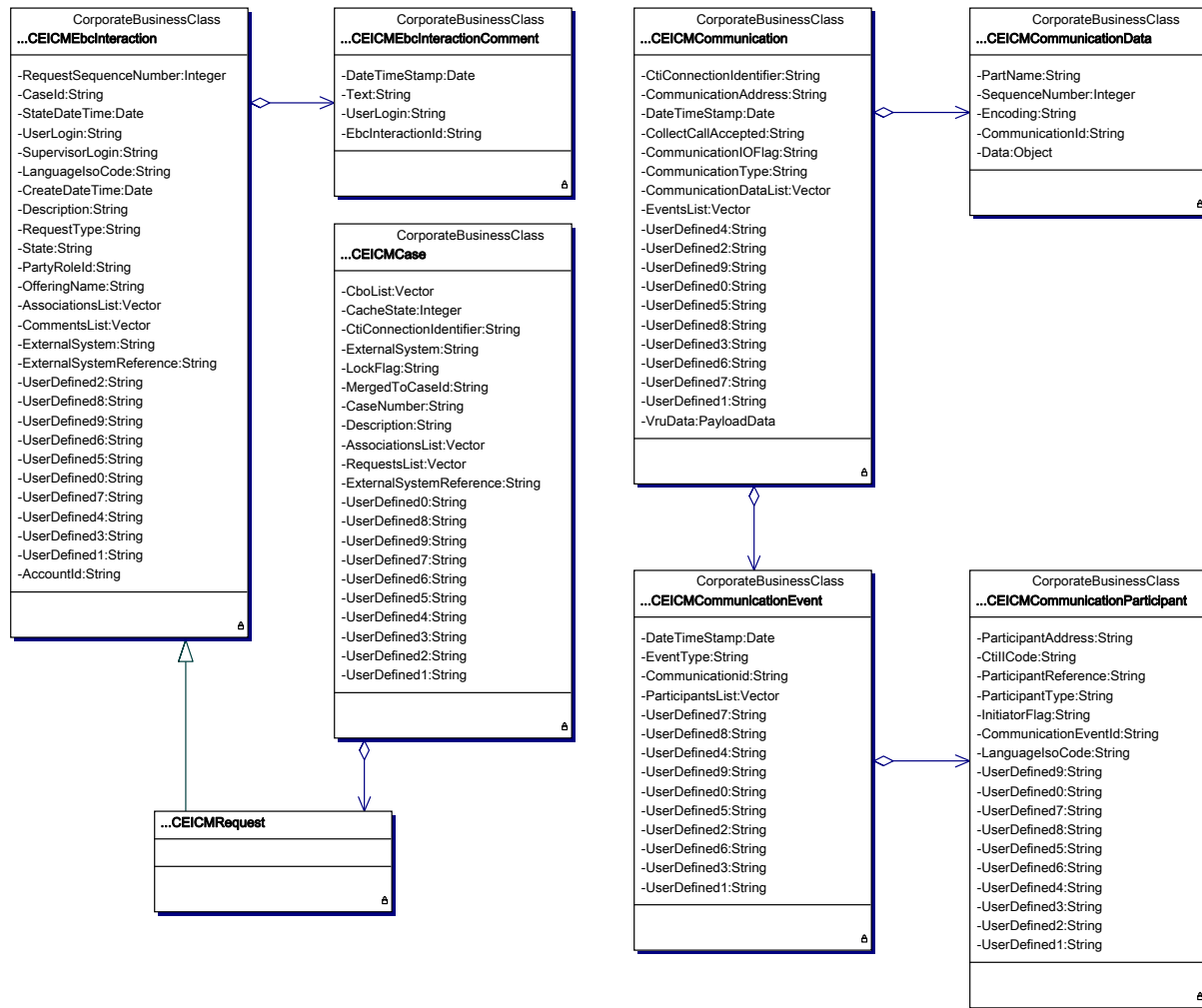


Figure 16-2: EBC Objects

The EBC Interaction Service is related to other services in the following ways:

- Offerings are related to Product Groups
- Cases and Requests are owned by Customers
- Requests can be associated with Accounts

Customizable Components

Package `com.chordiant.bd.helpers`:

- `CaseBuilderHelper`

Note: This is the real client. Use this instead of the `EbcInteractionClientAgent`.

Package `com.chordiant.bd.services`:

- `EbcInteractionService`

Package `com.chordiant.bd.clientAgents`:

- `EbcInteractionClientAgent`

Note: Do not use this client agent. Use the `EbcInteractionClientAgent` (described above) instead.

Package
`com.chordiant.businessServices.ebcInteractionCorporateBusinessClasses`:

- `CEICMCase`¹
- `CEICMCommunication`
- `CEICMCommunicationData`
- `CEICMCommunicationEvent`
- `CEICMCommunicationParticipant`
- `CEICMCustomerHistory`¹
- `CEICMEbcInteraction`
- `CEICMEbcInteractionComment`
- `CEICMRequest`
- `CEICMSearchContext`
- `CEICMSearchResult`
- `DateRange`

1. This business object has been manually modified for backward compatibility. If you regenerate it directly from the Rose model provided, compare it with the source code provided and make the appropriate changes. Or just delete the generated object and use the modified business object provided with the source code.

Package `com.chordiant.bd.jxp`:

- `CommunicationEventTypeTable`
- `CommunicationParticipantTypeTable`
- `CommunicationTypeTable`
- `EbcInteractionCommunicationView`
- `EbcInteractionCommunicationTable`
- `EbcInteractionTypeTable`
- `EiCommunicationData`

Package `com.chordiant.bd.numberGeneration`:

- `CaseNumberGenerator`

Package `com.chordiant.bd.constants`:

- `EbcInteractionServiceConstants`

The `EbcInteractionClientAgent` offers the following methods with which you can interact with the EbcInteraction Service. All of these methods are available as task descriptors for the Business Process Designer.

- **continueCase**—Retrieves the whole case object graph based on the `caseNumber`. The case is usually then available for updates and additions, such new requests.
- **getCase**—Retrieves the whole case object graph based on the `caseNumber`.
- **getCommunicationsForInteraction**—Retrieves communications for a given `interactionId`.
- **getInteractionComments**—Retrieves comments for a given `interactionId`.
- **getInteractionHistory**—Retrieves the `interactionHistory` for a given `partyRoleId`.
- **mergeCase**—Combines one or more cases. The resulting `Case` object will include all of the requests and communication details of the `fromCases`.
- **saveOrUpdateCase**—Saves a new `Case` or updates an existing one.
- **searchForCase**—Retrieves all `Cases` that match the search criteria.
- **startNewCase**—Creates an initialized `Case` with the inputs provided and stores it the database.
- **startNewCommunication**—Creates and initializes a new communication item with the input provided.

GUIDE SERVICE

The Guide Service provides a customer service agent with guidance when interacting with a customer. Guidance can consist of instruction steps, prompts, and scripts.

Specifically, the instruction set contains work steps that advise the agent. Likewise, prompts and scripts are UI components can be read by an agent. The Guide Service also provides standard values for pick lists within the application, such as marriage status, gender and customer title.

The update of instruction sets, prompts, and scripts can be managed by the Chordiant Business Data Manager. Refer to the *Chordiant 5 Tools Platform Administration Manager Guide* for details.

Customizable Components

Package `com.chordiant.bd.services:`

- `GuideService`

Package `com.chordiant.bd.clientAgents:`

- `GuideClientAgent`

Package

`com.chordiant.businessServices.applicationSupportCorporateBusinessClasses:`

- `CASCMGuide`
- `CASCMGuideContext`
- `CASCMInstructionSet`¹
- `CASCMInstructionStep`¹
- `CASCMPrompt`¹
- `CASCMPromptSet`
- `CASCMScript`
- `PickListItem`

Package `com.chordiant.bd.jxp:`

- `PickListTable`

Package `com.chordiant.bd.constants:`

- `GuideServiceConstants`

1. This business object has been manually modified for backward compatibility. If you want to use this object, subclass it and modify it, if necessary. When you generate the code, use only your modified code. Do not use the generated Chordiant classes from the provided object.

The `GuideClientAgent` offers the following methods with which you can interact with the Guide Service. All of these methods are available as task descriptors for the Business Process Designer.

- **getGuide**—Returns a guide of the specified type for the specified guide key and language.
- **getInstructionSetForActivityName**—Returns an instruction set for the specified context.
- **getInstructionSetForOfferingName**—Returns an instruction set for the specified context.
- **getPickList**—Returns a pick list for the specified list name.
- **getPrompt**—Returns the prompt within a prompt set, for a specified language, based on the guide key and prompt name.
- **getPromptForActivityName**—Returns a prompt for the specified activity name.
- **getPromptForField**—Returns a prompt for the specified field name.
- **getScriptForActivityName**—Returns a script for the specified context and language.
- **getScriptForOfferingName**—Returns a script for the specified context and language.

LOCATION SERVICE

The Location Service retrieves country, province, and language-related information based on the specified country code, ISO code, or other location information. The main business area for the Local Service is location information (country, state, province, postal code, and currency).

The Location Service is a stand-alone service.

Customizable Components

Package `com.chordiant.bd.clientAgents`:

- `LocationClientAgent`

Package

`com.chordiant.businessServices.applicationSupportCorporateBusinessClasses`:

- `CASCMCountry`
- `CASCMFieldInteractor`
- `CASCMInstructionSet`
- `CASCMInstructionStep`
- `CASCMLanguage`¹
- `CASCMLocation`

1. This business object has been manually modified for backward compatibility. If you want to use this object, subclass it and modify it, if necessary. When you generate the code, use only your modified code. Do not use the generated Chordiant classes from the provided object.

- CASCMPrompt
- CASCMScript
- CASCMSStateProvince

Package `com.chordiant.bd.constants`:

- LocationServiceConstants

The `LocationClientAgent` offers the following methods with which you can interact with the Location Service. All of these methods are available as task descriptors for the Business Process Designer.

- **getAllCountries**—Returns all countries.
- **getAllLanguages**—Returns all languages.
- **getAllStatesProvinces**—Returns all states and provinces for the specified country.
- **getLanguage**—Returns a language for the specified language attributes
- **getLanguageNameForIsoCode**—Returns the language name for the specified ISO language code.

OFFERING SERVICE

The Offering Service manages these entities:

- Offerings, which can be any product or service provided by an Enterprise Business Center (EBC)
- Offering categories, which is a grouping of one or more offerings or offering categories
- Offering views, which is a named hierarchical structuring of offering categories

Typically, each application in an EBC has its own offering view. Offering information is populated through the Chordiant Business Data Manager. Refer to the *Chordiant 5 Tools Platform Administration Manager Guide* for details. The main business area of the Offering Service consists of product and service information.

The Offering Service has the following relationship to other services:

- Offerings are related to product groups
- Offerings are captured as requests in the EBC Interaction Service

Customizable Components

Package `com.chordiant.bd.clientAgents`:

- OfferingClientAgent

Package

`com.chordiant.businessServices.establishmentCorporateBusinessClasses:`

- `CESCMOffering`
- `CESCMOfferingCategory`
- `CESCMOfferingHierarchyItem`
- `CESCMOfferingView`

Package `com.chordiant.jxp:`

- `OfferingHierarchyView`
- `OfferingHierarchyViewComparator`
- `OfferingTable`
- `OfferingViewTable`

Package `com.chordiant.bd.constants:`

- `OfferingServiceConstants`

The `OfferingClientAgent` offers the following methods with which you can interact with the Offering Service. All of these methods are available as task descriptors for the Business Process Designer.

- **`getAllOfferingsForView`**—Returns a graph consisting of all offering categories and offerings that make up the view, based on a supplied view name.
- **`getOfferingWithId`**—Returns a corresponding offering for a specified ID value.

ORDER FULFILLMENT SERVICE

The Order Fulfillment Service enables a user to start the fulfillment of an order. It also provides a way to find the delivery methods available for an order, based on the business entity of the party that made the order. The main business area of the Order Fulfillment Service consists of order information.

Customizable Components

Package `com.chordiant.bd.services:`

- `OrderFulfillmentService`

Package `com.chordiant.bd.clientAgents:`

- `OrderFulfillmentClientAgent`

Package `com.chordiant.businessServices.orderCorporateBusinessClasses:`

- `CODCMOrder`

The `OrderFulfillmentClientAgent` offers the following methods with which you can interact with the Order Fulfillment Service. All of these methods are available as task descriptors for the Business Process Designer.

- **getAvailableDeliveryMethods**—Returns all available delivery methods. (deprecated)
- **startFulfillment**—Starts the fulfillment process.

Package `com.chordiant.bd.constants`:

- `OrderFulfillmentServiceConstants`

ORDER GENERATION SERVICE

The Order Generation Service provides the main database operations for order objects, enabling orders to be created, retrieved, submitted, updated, canceled, and deleted using service calls. The Order Generation Service also provides the capability to determine cross-sell products for items in an order, and to substitute order line items that are not currently available.

The main business area of the Order Generation Service consists of order information. Figure 16-3 illustrates the hierarchy for order-related objects.

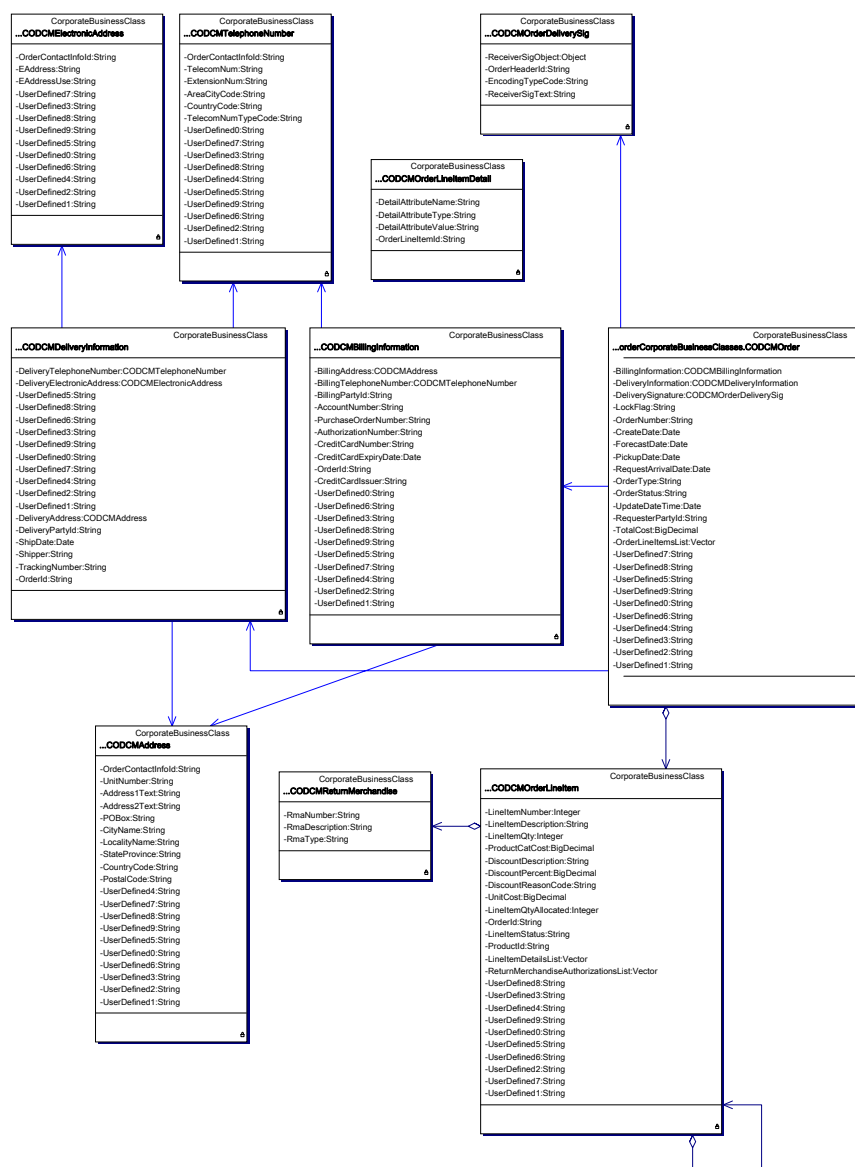


Figure 16-3: Order Objects

Customizable Components

Package `com.chordiant.bd.services`:

- `OrderGenerationService`

Package `com.chordiant.bd.clientAgents`:

- `OrderGenerationClientAgent`

Package `com.chordiant.businessServices.orderCorporateBusinessClasses`:

- `CODCMAddress`
- `CODCMBillingInformation`
- `CODCMDeliveryInformation`
- `CODCMElectronicAddress`
- `CODCMOrder`
- `CODCMOrderDeliverySig`
- `CODCMOrderLineItem`
- `CODCMOrderLineItemDetail`
- `CODCMReturnMerchandise`
- `CODCMTelephoneNumber`

Package `com.chordiant.bd.numberGeneration`:

- `OrderNumberGenerator`
- `RmaNumberGenerator`

Package `com.chordiant.bd.constants`:

- `OrderGenerationServiceConstants`

The `OrderGenerationClientAgent` offers the following methods with which you can interact with the Order Generation Service. All of these methods are available as task descriptors for the Business Process Designer.

- **cancelOrder**—Requests the order to be canceled through the Delivery Service.
- **createNewOrder**—Creates and initializes a new order.
- **determineCrossSellOption**—Searches for a cross-sell option, using the Product Service, based on the specified order and a line item number.
- **determineOrderAvailability**—Determines whether a specified order is available using the Inventory Service. In cases when a line item is not available, substitute items are returned through the Product Service.
- **getOrder**—Searches for a matching order.
- **submitOrder**—Sends an order for processing.
- **updateOrder**—Updates an order.

ORDER TRACKING SERVICE

The Order Tracking Service enables a user to retrieve the current status of an order (for example, `CREATED` or `DELIVERED`). You can also use this service to find all order numbers for a specified customer, or to locate the order associated with a specific order number.

The main business area for the Order Tracking Service is order information.

Customizable Components

Package `com.chordiant.bd.services`:

- `OrderTrackingService`

Package `com.chordiant.bd.clientAgents`:

- `OrderTrackingClientAgent`

Package `com.chordiant.businessServices.orderCorporateBusinessClasses`:

- `CODCMOrder`

The `OrderTrackingClientAgent` offers the following methods with which you can interact with the Order Tracking Service. All of these methods are available as task descriptors for the Business Process Designer.

- **findAllOrders**—Locates all order numbers based on either a specified order criteria, or all orders belonging to a customer. The `findAllOrders` method uses the Party Management Facility when locating order numbers by customer. (deprecated)
- **getOrderStatus**—Retrieves the status of the order, and updates the status field.

Package `com.chordiant.bd.constants`:

- `OrderTrackingServiceConstants`

PARTY ROLE SERVICE

The Party Role Service enables you to manage the various parties that participate in a business relationship with an enterprise. A party is a person or organization that can participate in a business relationship, while a role is the business view of the person or organization, such as customer or prospect.

You can use the service to create and manage parties and roles, and to establish relationships between them. You can also use the Party Role Service to manage contact information associated with individual parties.

Note: `BusinessEntity` (5.0 Establishment BO) is mapped to `Organization` and `OrgUnit` (current versions of `PartyRole` BOs).

Figure 16-4 illustrates the hierarchy of Party Role objects.

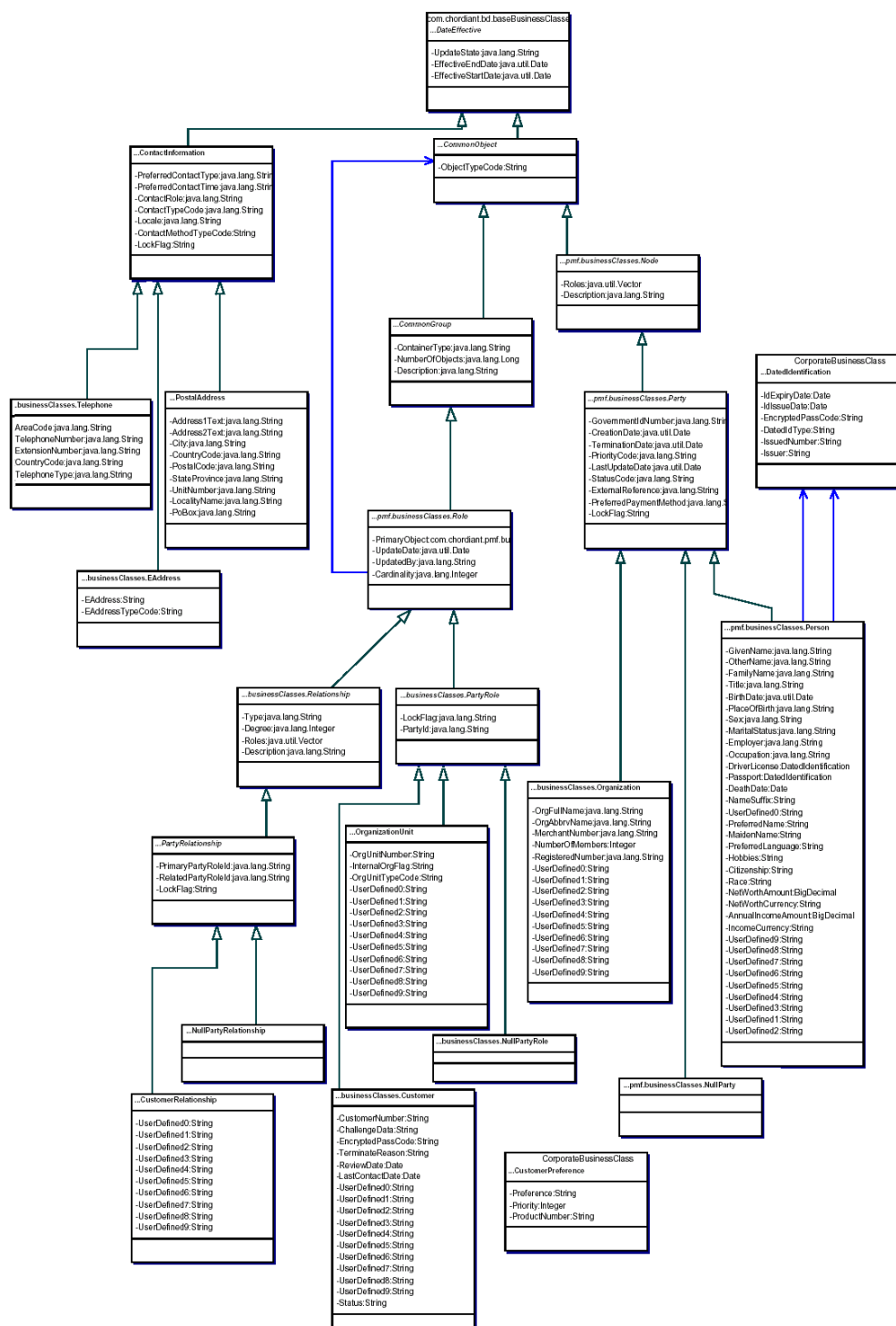


Figure 16-4: Party Role Facility

Customizable Components

Package `com.chordiant.pmf.service`:

- `PartyRoleService`

Package `com.chordiant.customer.service`:

- `PmfCustomerService`
- `PmfDelegateService` (refer to [“PmfDelegateService” on page 350](#))

Note: `PmfCustomerService` is an extension of `PartyRoleService`. Calls made to `CustomerManagerService` (Foundation Server 3.5 API) are delegated to `CustomerService` (Foundation Server 5.0 API) and then to the current version’s `PmfCustomerService`.

Package `com.chordiant.pmf.client`:

- `PartyRoleClientAgent` (refer to [“PartyRoleClientAgent” on page 347](#))

Package `com.chordiant.customer.client`:

- `PmfCustomerClientAgent` (refer to [“PmfCustomerClientAgent” on page 349](#))
- `PmfDelegateClientAgent`

Notes: `PmfCustomerClientAgent` is an extension of `PartyRoleClientAgent`. Calls made to `CustomerManagerClientAgent` (Foundation Server 3.5 API) are delegated to `CustomerClientAgent` (Foundation Server 5.0 API) and then to the current version’s `PmfCustomerClientAgent`.

`PartyRoleClientAgent` also handles calls to the old `EstablishmentClientAgent`.

Package `com.chordiant.businessServices.partyBusinessClasses`:

- `ContactInfo`
- `ContactInfoList`

Package `com.chordiant.pmf.businessClasses`:

- `CommonGroup`
- `CommonObject`
- `ContactInformation`
- `DatedIdentification`
- `DateEffective`
- `Node`
- `NullParty`
- `NullPartyRelationship`
- `NullPartyRole`
- `Organization`
- `OrganizationUnit`
- `Party`
- `PartyRelationship`
- `PartyRole`
- `PartyRoleNote`
- `Person`
- `PostalAddress`
- `Relationship`
- `Role`
- `Telephone`

Package `com.chordiant.pmf.constants`:

- `PMFConstants`

Package `com.chordiant.customer.constants`:

- `PmfCustomerConstants`
- `PmfDelegateConstants`

PartyRoleClientAgent

The `PartyRoleClientAgent` offers the following methods with which you can interact with the Party Role Service. All of these methods are available as task descriptors for the Business Process Designer.

- **addNote**—Adds a note.
- **addRelatedObject**—Creates a relationship between a role and another `CommonObject`.
- **addRelatedPartyRole**—Associates a party role with another party role.
- **addRole**—Assigns a new role to an existing node.
- **create**—Creates a new `CommonObject` or `CommonGroup`, or derivation thereof, that represents the type requested.
- **createParty**—Initializes contact information on the party.
- **createRelationship**—Creates a new first class relationship object from two role objects and a relationship type.
- **createRole**—Given a primary object and the requested type, creates a `CommonObject` object and associates it with the primary.
- **getAllNotes**—Retrieves all notes.
- **getAllPaymentMethods**—Returns all payment methods for a specified customer.
- **getAllRelatedObjectsByRole**—Returns all related `CommonObjects` based on the role.
- **getAllRelatedPartyRoles**—Returns all related party role objects for a specified role name.
- **getAllRelatedPartyRoles**—Returns all related party role objects for a specified role name.
- **getAllRoleNames**—Returns a sequence of strings representing all of the roles this object currently assumes.
- **getAllRoles**—Returns a list of all the roles assumed by a specific `CommonObject`.
- **getCommonObject**—Retrieves the `CommonObject` based on the attributes of the given `CommonObject` object.
- **getContactInformation**—Returns a sequence of references to all contact information related to a party.
- **getContactInformation**—Returns a sequence of references to all contact information related to a `PartyRole`'s primary object.
- **getContactInformationHistory**—Returns the contact information history.
- **getContainedObject**—Returns the contained object.
- **getParties**—Returns the Party objects for the specified Party object.
- **getPartiesByAddress**—Returns the parties associated with a specified postal address.

- **getPartiesByDatedIdent**—Returns the parties associated with a specified dated identification.
- **getPartiesByEAddress**—Returns the parties associated with a specified electronic address.
- **getPartiesByPartyIds**—Returns the Party objects based on the specified party identifiers.
- **getPartiesByTelephone**—Returns the parties associated with a specified telephone number.
- **getPartyRolesByPartyIds**—Returns the PartyRole objects based on the specified party identifiers.
- **getPartyRolesByPartyIds**—Returns the PartyRole objects based on the specified party identifiers and role type.
- **getRelatedObject**—Provides a mechanism to obtain a related object based on the role of the related object.
- **getRelatedPartyRole**—Enables access to a related party role object based on its role relative to this object.
- **getRoles**—Returns the roles associated with the string role name in the specified Role object.
- **getSupportedContactTypes**—Returns a sequence of strings that represents all of the contact types with which a party could be associated.
- **getSupportedParties**—Returns the types of parties that this manager is capable of creating.
- **getSupportedPartyRoles**—Returns all party role types, in the form of a string, that this manager is capable of creating.
- **getSupportedRelationships**—Returns all of the types of relationships that this manager is capable of creating.
- **getSupportedRoles**—Returns all role types, in the form of a string, that this manager is capable of creating.
- **getSupportedRolesForRelationship**—Provides the role names that are allowed on the specific relationship type in the specified Relationship object.
- **removeNote**—Removes a note.
- **removeParty**—Removes a Party object.
- **removeRelatedObject**—Breaks the relationship between a role and another CommonObject.
- **removeRelatedPartyRole**—Tears down the relationship between two party roles.
- **removeRole**—Breaks the relationship between a Node object and a role.
- **setContactInformation**—Uses the behavior object associated with the input business object to associate the list of contact information objects with it.
- **setContactInformation**—Uses the behavior object associated with the input business object to associate the list of contact information objects with its primary object.
- **updateCommonObject**—Updates the CommonObject based on the attributes of the specified CommonObject object.

- **updateParty**—Updates the Party object and its contact information with the specified information.

PmfCustomerClientAgent

PmfCustomerClientAgent offers the following methods with which you can interact with the PmfCustomerService.

- **addCustomerPreferences** — Adds product preferences to an existing Customer object.
- **addOnFilePaymentMethods** — Adds the given Onfilepaymentmethods to an existing Customer object. Each dated ID item on the list must contain the following attributes: dated identification, type, issuer name, issuer number.
- **authenticateCustomer** — Authenticates the customer based on the given customer information.
- **getAllDeliveryMethods** — Returns a list of delivery methods for a customer.
- **getCustomerPreferences** — Returns a customer's product preferences.
- **getCustomers**—Returns a list of customers based on attributes of the input object.
- **getCustomersByAddress** — Returns the customer's information based on the given postal address.
- **getCustomersByDatedIdent** — Returns a list of customers based on customer's dated identification object.
- **getCustomersByEAddress**—Returns the customer's information by the given electronic address.
- **getCustomersByTelephone** — Returns the customer's information by the given telephone number.
- **getOnFilePaymentMethods** — Returns all onfilepayment methods for the given Customer object.
- **updateCustomerPreferences** — Updates the product preferences of the given Customer object.
- **updateOnFilePaymentMethods** — Updates the on file payment methods for the given Customer object.

PmfDelegateService

The PmfDelegateService is a subclass of the PmfCustomerService, which is itself, a subclass of the PartyRoleService. This service extends the PmfCustomerService to include the concept of a *delegate* — a person who is able to act on behalf of another customer. One customer can have more than one delegate and one person can be a delegate for one or more other customers.

Note: In this implementation, the person filling the role of Delegate must also fill the role of an existing Customer.

As illustrated in [Figure 16-5](#), a new Party Role called **Delegate** is included in this service to fill the new Role of **Customer Delegate**. This Party Role is similar to the Customer and OrganizationUnit Party Role classes already implemented the Party Role service.

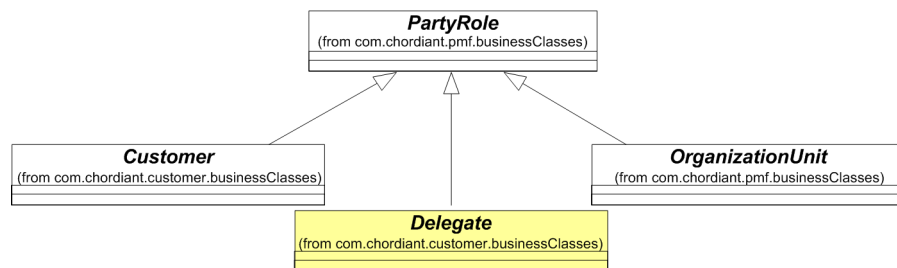


Figure 16-5: Delegate Role

As illustrated in [Figure 16-6](#), a **DelegateRelationship** class is subclassed from the **PartyRelationship** class. It models the **Delegate** relationship between a **Customer** and a **Delegate**.

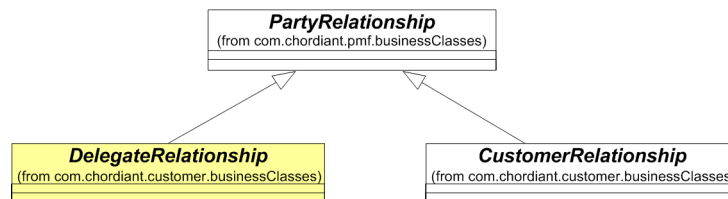


Figure 16-6: DelegateRelationship

The PmfDelegateService is provided for use in Call Center Advisor - Browser Edition. You can also use it or customize it for your own application.

For details on the PmfDelegateService, refer to its Javadoc.

PRODUCT SERVICE

The Product Service provides information on products, product details, product catalogs, product category, and packages. A product can be associated with a product catalog or product category. A product catalog contains a list of available products, while a product category is a collection of products grouped for the convenience of identifying similar kinds of products. An offering in an application is often associated with a product category.

To retrieve a product, you can search for a product either by a product number or by a list of products such as product catalog or product category. Products can be offered individually or as part of a product package. A product package consists of other products called sub-products.

For example, a promotion introduction sale is often associated with opening a new account. Products can be cross-sold for a product. If a product is not available, an alternate product can substitute for a product.

Product detail describes the product details. A product detail value contains the characteristic of the product detail. It is used to accurately describe a product line or all variations of a product. For example, a product detail might contain the value “Green” for a detail of “Check Color”.

You can use the Chordiant Business Data Manager to manage updates to product categories, catalogs, and quantities. Refer to the *Chordiant 5 Tools Platform Administration Manager Guide* for details. [Figure 16-7](#) illustrates the hierarchy of Product objects.

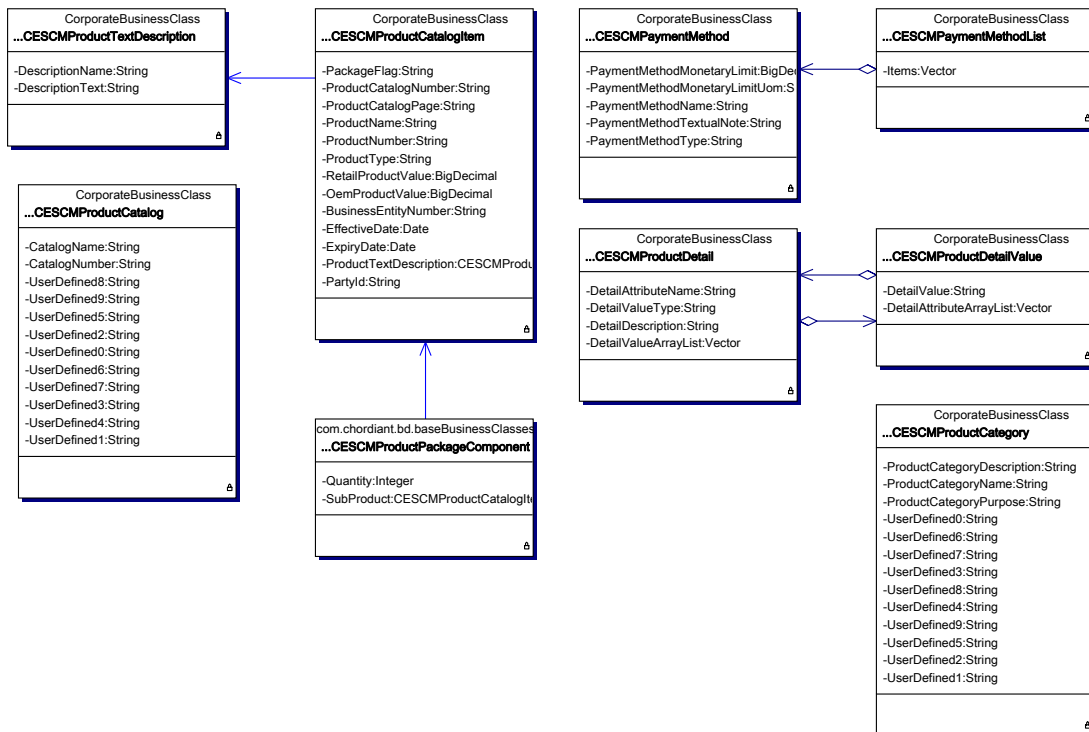


Figure 16-7: Product Objects

The main business area for the Product Service is product and product catalog information. Products are associated with order line items and customer preferences. Product groups can also be referenced in offerings.

Customizable Components

Package `com.chordiant.bd.services`:

- `ProductService`

Package `com.chordiant.bd.clientAgents`:

- `ProductClientAgent`

Package

`com.chordiant.businessServices.establishmentCorporateBusinessClasses`:

- `CESCMPProductCatalog`
- `CESCMPProductCatalogItem`
- `CESCMPProductCategory`
- `CESCMPProductDetail`
- `CESCMPProductDetailValue`
- `CESCMPProductPackageComponent`

Package `com.chordiant.bd.jxp`:

- `ProductByCatalogView`
- `ProductByCategoryView`
- `ProductCatalogItemTable`
- `ProductCatalogTable`
- `ProductCrossSellTable`
- `ProductDetailHierarchy`
- `ProductDetailTable`
- `ProductDetailValueTable`
- `ProductPackageComponentTable`
- `ProductSubstituteTable`
- `ProductTable`

Package `com.chordiant.bd.constants`:

- `ProductServiceConstants`

The `OrderTrackingClientAgent` offers the following methods with which you can interact with the Order Tracking Service. All of these methods are available as task descriptors for the Business Process Designer.

- **addSubProductToProduct**—Retrieves a sub-product for a specified product.
- **getProduct**—Retrieves a product.
- **getProductCatalog**—Retrieves a product catalog for a specified business entity.
- **getProductDetailNode**—Retrieves a product detail node by detail index.
- **getProductDetails**—Retrieves products details.
- **getProductDetailValues**—Retrieves product detail views by detail index.
- **getProductPackages**—Retrieves product packages.
- **getProductsForCategory**—Retrieves products for a specified category.
- **identCrossSellLineItems**—Locates cross-sell line items for a product.
- **identifySubstituteProductItem**—Locates substitute product items for a specified product.
- **isProductPackage**—Determines whether a specified product is a package.

BACKWARD COMPATIBILITY APIs

The following APIs exist for backward compatibility only.

Customer Service Client Agent

The Customer Service has been deprecated. The Client Agent for the Customer service still exists for backward compatibility.

Please use the `PMFCustomerService` for all customizations. Refer to [“PmfCustomerClientAgent” on page 349](#).

EstablishmentClientAgent

The `EstablishmentClientAgent` remains for backward compatibility. Calls to `EstablishmentClientAgent` are delegated to the `PartyRoleClientAgent`. For details on the `PartyRoleClientAgent`, refer to [“Party Role Service” on page 343](#).

`BusinessEntity` (Foundation Server version 5.0 `Establishment Business Object`) is mapped to `Organization` and `OrgUnit` (current versions of the Foundation Server `PartyRole Business Objects`).

Index

A

- abstract, metadata tag 308
- access strategy
 - called from Generic Service 194
 - CMI example 223
 - generated 219
 - in sequence flow 195
 - overriding 221
 - overriding in tutorial 267
 - overview 207
 - specifying 221
- AccessStrategyInput
 - generated 219
 - in sequence flow 195
 - modifying 269
 - overview 207
- Account Service 328, 331
- AccountNumberGenerator 52
- ActivityDetails 234
- Add External JAR 63
- addNote 289
- addRelatedObject 280
- addRelatedPartyRole 289
- addRole 282
- aggregation
 - attribute of class 236
 - automobile to engine 256
 - Automobile to Tire 258
 - classAttribute 257
 - CMI example 236
 - drawing 234
 - foreignClassAttribute 257
 - in tutorial 234
 - JXP MQ specifying 190
 - MQ persistence tags for 316
 - unilateral 258
- alignment 190
- all, for access strategy override 223
- annotation
 - attribute-level 242
 - class-level 241
 - in Rational Rose 241
- Ant scripts
 - ctpbuid.xml 35
 - generate-tasks.xml 154
 - generating business components 32
 - generation targets 37
 - web services generation 97
- ws-deploy.xml 106
- ws-java2wsdl.xml 97, 100
- ws-runChordiantSample.xml 126
- ws-run-nonchordiant-
 {application_server}.xml 142
- ws-undeploy.xml 109
- ws-wsdl2java.xml 103
- ws-wsdl2java-dynamic.xml 112, 114
- antcall, task descriptor generation 157
- APIs excluded from task descriptors 155
- application components, definition 4
- Application Packaging Manager 68
- architecture
 - distributed 173
 - extended persistence 196
- arguments, creating new in service 61
- association
 - Automobile to Dealer 261
 - CMI example 240
 - drawing 232
 - foreign key 232
 - getAssociationClassName 207
 - in Rational Rose 233
 - model 234
 - naming 233
 - retrieveAssociation 202, 232
 - retrieveAssociation(String) 202
 - retrieveAssociationGraphs(String) 203
 - retrieveAssociationTypes 207
 - specifying 232
 - stereotype 240
 - tutorial 261
 - type 232
 - Vector example 237
- associationType 234
- attribute
 - adding to business object 166
 - adding to derived object 301
 - adding to subclass 6
 - aggregation 236
 - annotation 242
 - creating in tutorial 247
 - data types 6

- defined in CMI 171
- do not remove 6
- from association 233
- JXP SQL tab 250
- like 172
- metadata 322
- metadata tag 308
- precision 6
- private 6
- public 6
- size 6
- transferring 172
- attribute-level metadata
 - CMI 306
 - SMI 321
- attributes tab
 - class specification 186
 - in tutorial 250
- attributes, MQ persistence tags for 316
- authentication
 - getAuthentication 205
 - setAuthentication 204
 - web services 91
- authenticationToken 48
- autogenerate primary key 189
- automatic code generation, disabling 17
- Automobile class created 252
- automobileBehavior.java 265
- AutomobileBehaviorTest.java 198
- available web services 113
- Axis, web services infrastructure 99
- AxisServlet 113

B

- base
 - object model 59, 173, 245
 - service model 41
- behavior
 - adding 7
 - do not remove 7
 - extended persistence
 - adding functionality 265
 - architecture 197
 - base class
 - exceptions 206
 - overview 204
 - factory 211

- generated 219
- getBehaviorByObject 212
- getBehaviorByObjectName 212
- getters and setters 198
- IIOP and sockets, no access 265
- instantiating 198
- methods 198
- overriding in tutorial 265
- overview 198
- retrieving behavior 265
- sequence flow 195
- specifying 223
- overloading 7
- overriding 7
- removing, avoid 7
- standard persistence
 - considerations 300
 - constructor 300
 - customization 300
 - customizing 301
 - diagram 297
 - factory 302
 - sequence diagram 299
 - service relationship 298
- BehaviorFactoryConfiguration.xml 211, 213, 219
- best practices
 - customizing business components 1
 - customizing services 49
 - extended persistence usage 215
 - web services 115
- bindings for task descriptors 153
- blended models 5
- BO Base Model 23
- Boat class created 254
- BOB. *See* business object behavior
- BOC. *See* business object criteria
- build path, Java 63
- build.xml for web services 113
- Business Component Generator
 - Ant script alternative 32
 - Ant-based targets 37
 - before you begin 15
 - components created 2
 - deploying components 38
 - error messages 18
 - JDK 1.4 instructions 19
 - overview 15
 - slow generation 19

- business logic 46
- business metadata 307
- business object
 - See also* object
 - adding attribute 166
 - adding locking 170
 - attribute, adding 166
 - CorporateBusinessClass 6
 - created automatically 164
 - criteria, created automatically 164
 - customization criteria 6
 - customizing 165
 - definition 6
 - getters and setters 6
 - helper, generated 219
 - locking 169, 170
 - no logic 6
 - overriding 6, 167
 - requirements for web services 96
 - subclassing existing 166, 174
 - synchronize with database 168
 - transferring attributes 172
 - use object factory 50
- business object behavior
 - extended persistence
 - base class 204
 - generated 219
 - instantiating 198
 - overview 198
 - specifying 223
 - standard persistence
 - cache 300
 - called by service 50
 - considerations 300
 - constructor 300
 - created automatically 164
 - customization 300
 - customizing 301
 - diagram 297
 - do not maintain state 300
 - factory 302
 - from Resource Manager 50
 - logic in 297
 - no setup method 300
 - sequence diagram 299
 - service relationship 298
 - skeleton code generated 168
 - use object factory 50

- business object criteria
 - created automatically 164
 - lookupAllDescendantClasses 207
 - use object factory 50
- business object helper, generated 219
- Business Process Designer task descriptors 44, 151
- business service. *See* service
- BusinessDataServiceBaseClass 5
- BusinessObjectResourceManager 300

C

- cache
 - in services 215
 - manager, caution 165
 - standard persistence business object behaviors 300
 - updating to match data 173
- CAFE tasks 151
- callback 45
- CaseNumberGenerator 52
- cases, in EBC Interaction Service 332
- case-sensitivity
 - in CMI files 307
 - in SMI files 321
- caution, changing serialized objects 165
- changing data types 6
- Chordiant 5 UML Extender for Rational Rose 2
- Chordiant BO Base Model 23
- Chordiant Data Model 23
- Chordiant JARs 18
- Chordiant Metadata Information. *See* CMI
- Chordiant Service Model 23
- CHORDIANT_RUNTIME 18
- ChordiantUtils project 100
- chordiantXAds 248
- CICS, JXP MQ 180
- class
 - annotation 241
 - creating new 246
 - diagram, creating 245
 - loading dynamically 214
 - metadata tag 308
 - subclassing from base class 246
- CLASS_NAME 48
- classAttribute 234, 257, 259
- class-level metadata
 - CMI 306

- SMI 321
- classpath variables 18
- client agent
 - callback to 45
 - created automatically 43
 - Generic Service 194
 - tester 68
- client task descriptor 152
- CMI
 - access strategy 223
 - aggregation example 236
 - association example 240
 - attribute definition 171
 - business metadata 307
 - case-sensitivity 307
 - core business metadata 308, 309
 - definition 306
 - directory 264
 - example code 231, 233, 314, 319
 - missing values 13
 - modeling overview 2
 - MQ persistence metadata 314
 - persistence metadata 310
 - SQL persistence metadata 310
 - SQL persistence sample 313
 - structure 306
- code
 - CMI 314
 - CMI example 231, 233, 319
 - generation
 - Ant scripts 32
 - disabling 17
 - slow 19
- collection type 236, 239, 259
- column
 - name 188, 246, 250
 - size 188
- CommonGroup 275
- CommonObject 275, 277
- components
 - application, definition 4
 - extended persistence 3
 - persistence 3
 - service framework 3
- config directory 264
- configuration
 - behavior factory 211, 213
 - DynamicLoaderComponent.xml 214
 - files, factory 219
 - refreshing 210
 - Resource Manager settings 44
 - service 5
 - settings, updating in tutorial 82
 - validator factory 209
 - web services 89
- CONFIGURATION_SECTION_NAME_SUFFIX 48
- ConfigurationHelper 210, 214
- configuring
 - Rational Rose for Java components 9
 - web services 91
 - XMI export 9
- constants
 - created automatically 43
 - creating in tutorial 60
 - for overloading 51
- constructor
 - avoiding 5
 - default, in business object 96
 - for ResourceManager reference 300
 - required for web services 96
- ContactInformation 275
- contained
 - associated objects 234
 - vector 237
- containment, in tutorial 255
- controller 44
- copyright symbol 12
- core business metadata
 - exploring 308
 - sample 309
- CorporateBusinessClass 6
- create method 198
- createParty 292
- createRelationship 294
- createRole 293, 295
- Credit Card Account Service 331
- criteria
 - business service 5
 - customizing business objects 6
- CRUD logic in access strategy 197
- ctpbuid.xml 35
- customer
 - converting prospect to 296
 - creating 295
- Customer Service (deprecated) 353
- CustomerNumberGenerator 52

- customization guidelines 5
- customizing
 - business objects 165
 - criteria for business objects 6
 - deeper level 7
 - guidelines for 5
 - introduction 4
 - levels 6
 - PartyRole strategy 297
 - philosophy of 4
 - standard persistence business object behavior 300, 301
 - web services 98
- customObject 235

D

- DA. *See* data accessor
- data
 - accessor
 - created automatically 164
 - use object factory 50
 - logical type 189
 - matching cache 173
 - seeding 271
 - synchronizing across JVMs 173
 - type changing 6
 - type error 12
- Data Model, Chordiant 23
- database
 - synchronizing with attribute 168
 - XSD created automatically 164
- date format 191
- DateEffective 277
- db directory 264
- DB2UDB, Persistent Type 176, 310
- Dealer
 - Dealer class created 261
 - getDealer method 265
- default
 - constructor, web services requires 96
 - values in CMI 13
- delegate service 350
- deleting, setToBeDeleted 205
- Delivery service 331
- dependency, JAR 69

- deploying
 - business component code 38
 - web services 106
- deployment descriptor files 87
- derived object, adding attribute 301
- descriptor files
 - Ant-based generation 34
 - for business component generator 16
- development
 - environment memory 19
 - runmode 214
- digits 188
- directories, output 264
- disabling automatic code generation 17
- dispatcher 44
- distributed architecture 173
- documentation, adding in Rational Rose 241
- domain
 - boundaries 300
 - logic 44
- DSN
 - JXP class-level 175
 - JXP class-level, in tutorial 248
 - metadata tag 310, 314
 - value 248
- duplicating a tester 85
- dynamic
 - web service invocation 117, 144
 - WSDL distribution 97
- DynamicLoaderComponent.xml 214

E

- EAddress 275
- EAR file, web services 113
- EBC Interaction Service 331
- ebs script 10
- efficiency of lookups in inheritance 225
- EJB Modules 69
- empty methods
 - in business object behavior 168
 - in generated code 50
- encoded XML type 185
- endpoint object, web services 146
- Engine class created 255

- error
 - generation, due to copyright symbol 12
 - handling, web services 114
 - messages in Business Component Generator 18
- EstablishmentClientAgent, use PartyRoleClientAgent 353
- example code
 - CMI access strategy 223
 - CMI association 233
 - CMI attribute definition 171
 - CMI extended persistence 319
 - CMI for extended persistence 231
 - CMI for Generic Service 314
 - dynamic web service invocation 144
 - proxy for non-Chordiant web service 132
 - SAAJ 144
 - typeField 231
 - web services proxy 124
 - web services, installing 118
- exceptions, behavior base class 206
- excluded APIs for task descriptors 155
- execute method 267
- exporting
 - JAR 68
 - model to XMI 15
 - XMI, configuration 9
- extended persistence
 - AccessStrategy 207
 - AccessStrategyInput 207
 - architecture 196
 - Chordiant Data Model 217
 - components 3
 - flexibility with 194
 - Generic Service 194
 - Generic Service Client Agent 194
 - helpers 206
 - IAccessStrategy 207
 - tutorial 244
 - typeField 230
 - typeValue 229
 - usage model 215
 - validator factory 209
 - validators 209
- extended persistence behavior. *See* business object behavior, extended persistence
- Extended Persistence Tutorial Model 23
- ExtendedAutomobileAccessStrategy 208, 267

- extending
 - code to prevent overwriting 63
 - generated service 63
- external web services, calling 115

F

- façade service 216
- factory
 - behavior 211
 - configuration files 219
 - object 50
 - returning behavior 265
 - standard persistence business object behavior 299, 302
 - validator 209
- field value pairs 204
- fill character 190
- final, metadata tag 309
- FireForget 178
- flexibility with extended persistence 194
- foreignClassAttribute 234, 257, 259
- foreignKey association 232
- foreignKeyAttribute 233
- FOUNDATION_LIB 18, 19

G

- generated service, extending 63
- GenerateTaskDescriptor tool 153
- generate-tasks.xml script 154
- Generic Service
 - architecture 196
 - getGenericServiceInstance 205
 - overview 194
 - setGenericServiceInstance 205
 - use of 217
 - use with standard services 215
 - when to use 216
- Generic Service Client Agent
 - and Generic Service 194
 - in sequence flow 195
 - overview 196
 - use of 217
- getAdditionalData 206
- getAllNotes 289

- getAllPaymentMethods 285
- getAllRelatedObjectsByRole 280
- getAllRelatedPartyRoles 289
- getAllRoleNames 282
- getAllRoles 282
- getAssociationClassName 207
- getAuthentication 205
- getBehaviorByObject 212
- getBehaviorByObjectName 212
- getBehaviorForName 50
- getCommonObject 278
- getContactInformation 285
- getContainedObject 205
- getDealer 265
- getGenericServiceInstance 205
- getId 205
- getParties 285
- getPartiesByAddress 285
- getPartiesByDatedIdent 286
- getPartiesByEAddress 286
- getPartiesByPartyIds 286
- getPartiesByTelephone 287
- getPartyRolesByPartyIds 290
- getQuickStockQuote method 134
- getRelatedObject 280
- getRelatedPartyRole 290
- getRoles 283
- getSecurity 205
- getSupportedContactTypes 292
- getSupportedParties 293
- getSupportedRelationships 294
- getSupportedRoles 293
- getSupportedRolesForRelationship 294
- getters and setters
 - business object 6
 - in behaviors 198
 - required for web services 96
- getToBeDeleted 205
- getType 206
- getUsername 205
- getValidatorByObject 210
- getValidatorByObjectName 210
- graph
 - getBehavior 212
 - locking 172
 - retrieve 200
 - retrieveAllMatchingGraphs 201
 - retrieveAssociationGraphs(String) 203

- retrieveGraphToDepth 200
- updateGraph 203
- updatePointOptimistic 172
- GUID 189
- Guide Service 335
- guidelines, customization 5

H

- Hashtable, toFieldValuePairs 204
- helper
 - generated 219
 - methods 206
 - overview 206
 - retrieveAssociation 232
- hierarchy
 - typeField 230
 - typeValue 229
 - using retrieve method 227
- HRF2, JXP MQ tab 183
- Husband 295

I

- IAccessStrategy 207
- IIOP, no access to behavior methods 265
- imports, organize 66
- IMS, JXP MQ 179, 182
- inheritance
 - efficiency of lookups 225
 - hierarchy
 - typeField 230
 - typeValue 229
 - using retrieve method 227
 - in tutorial 245
 - lookupAllDescendantClasses 206
 - retrieveGraph 200
 - retrieveGraphToDepth 200
 - table examples 226
 - tags 225
 - three tags 224
- installing web services samples 118
- interfaces as parameters, avoid 5
- introduction to customization 4
- InvalidRelatedRole exception 280
- ISecurityInformation 212
- isService, metadata 322

J

- J2EE session facade analogy 216
- j2ee.jar 18, 63
- J2EE_LIB 18
- JAR
 - business component code 38
 - dependency editor 69
 - exporting 68
- Java
 - build path 63
 - task descriptor 152
 - type, metadata 322
 - type, metadata tag 308
- Javadoc
 - adding in Rational Rose 241
 - metadata 322, 323
 - metadata tag 308
- javaType tag 12
- JDK 1.4
 - business component generation 19
 - schema files and 26
 - task descriptors 158
- joins 177
- JUnit testers 270
- JVM
 - specifying 19
 - synchronizing data across 173
- JXBBOBase model 173
- JXBServiceBase model 41, 59
- JXC CMI tab
 - properties in tutorial 249
 - typeValue 229
 - useBehavior 224
 - xrefTableName 228
- JXC Properties tab
 - for contained object 235
 - for contained vector 239
 - properties for tutorial 251
 - typeField 230
- JXC support 10
- JXCTutorialDescriptor.xml 16
- jxextensions.jar 63
- JXP attribute-level metadata, specifying 187
- JXP class-level metadata, specifying 175
- JXP MQ CICS tab 180
- JXP MQ HRF2 tab 183
- JXP MQ IMS tab 182

- JXP MQ tab
 - aggregation 190
 - always displayed 176
 - attribute-level 187, 189
 - figure 178
 - overriding preset values 179
 - properties 178
- JXP MQ XML tab 185
- JXP SQL tab
 - always displayed 176
 - attribute-level 187, 250
 - figure 177
 - properties for tutorial 246, 249
 - table name 177
 - where prefix 177
- JXP tab
 - attribute 187
 - DSN 175
 - figure 175
 - lock
 - field 171
 - strategy 170, 175
 - persistent type 175
 - properties for tutorial 248

K

- key autogeneration 189

L

- last name 207
- levels of customization 6
- lib directory 68, 264
- like attributes 172
- listing web services 113
- literal 185
- loading classes dynamically 214
- location of
 - models 41
 - WSDL file 100
- Location Service 336
- lock field
 - only one 171
 - specifying 171, 187
- lock strategy 175, 248
- LockField metadata tag 311
- LockFlag 171

- locking
 - adding to object 170
 - graph of objects 172
 - strategy 170
 - using object locking 169
- LockStrategy metadata tag 170, 311, 315
- locktokentext 171
- logic
 - in service 46, 50
 - in standard persistence business object behavior 297
 - not in business object 6
- logical type 189, 246, 250
- lookupAllDescendantClasses 206
- lookupAllDescendantCriteriaClasses 207
- lookupClassByType 206

M

- Manager 292
- manifest.mf 69
- mapping, persistent 6
- marriage, establishing 295
- memory of development environment 19
- message, SOAP-encoded 165
- MessageFactory, web services 145
- metadata
 - attribute name 308, 322
 - attribute-level
 - CMI 306
 - SMI 321
 - attributes 308, 322
 - business 307
 - class name 308, 322
 - class-level
 - CMI 306
 - SMI 321
 - core business metadata 308, 309
 - description 305
 - final 309
 - isService 322
 - Java type 308, 322
 - Javadoc 308, 322, 323
 - JXP attribute-level, specifying 187
 - JXP class-level, specifying 175
 - MQ attribute-level, specifying 189
 - MQ CICS, specifying 179
 - MQ class-level, specifying 178
 - MQ HRF2, specifying 183
 - MQ IMS, specifying 182
 - MQ persistence 314
 - multiplicity 308
 - operation 308, 322
 - operation name 323
 - operation-level
 - CMI 306
 - SMI 321
 - override 308
 - package 308
 - package name 308, 322
 - parameter 323
 - parameter type 323
 - parent class 308, 322
 - persistence 310
 - persistence, specifying 174
 - return type 323
 - services 322
 - SQL attribute-level, specifying 188
 - SQL class-level, specifying 177
 - SQL persistence 310
 - static 309
 - visibility 309, 323
- methods
 - empty in generated code 50
 - excluded from task descriptors 155
 - extended persistence business object behavior 198
- missing default values 13
- mixed models 5
- models
 - associated object with contained vector 237
 - base object model 59, 245
 - Chordiant BO Base Model 23
 - Chordiant Data Model 23, 217
 - Chordiant Service Base Model 23
 - Chordiant Service Model 23
 - contained associated objects 234
 - export to XMI 15
 - extended persistence 215
 - Extended Persistence Tutorial Model 23
 - in data model plugin 23
 - JXBBOBase 173
 - JXBServiceBase 41
 - location of provided 41, 46
 - mixing types 5
 - Party Management Facility 274

- QueueItem Model 23
- service components from 42
- Service Creation Tutorial Model 23
- Service Extension Tutorial Model 23
- System Task Processor Service Model 23
- tutorial 57, 243
- modeltype, override for Ant generation 34
- modifying
 - AccessStrategyInput 269
 - business objects 165
 - existing service 46
- MOF, Rational Rose 9
- MoreThanOneRelated exception 280
- move code to prevent overwriting 49, 168
- MQ attribute-level metadata 189
- MQ CICS metadata 179
- MQ class-level metadata 178
- MQ HRF2 metadata 183
- MQ IMS metadata 182
- MQ persistence metadata
 - alignment 316
 - character set 315
 - correlation id 316
 - data source name 314, 315
 - date format 317
 - encoding 315
 - encoding get 315
 - fill character 316
 - format 315
 - get message size 315
 - input 316
 - length 316
 - lock strategy 315
 - message model 315
 - message type 316
 - output 316
 - persistent type 314
 - primary key generation type 317
 - priority 316
 - starting offset 316
 - tags 314
- mqAlignment metadata tag 316
- mqDateFormat metadata tag 317
- mqFillCharacter metadata tag 316
- MQFMT_CICS 179
- MQFMT_IMS 179
- MQFMT_RF_HEADER_2 179
- mqGetMessageSize metadata tag 315
- mqInput metadata tag 316
- mqLength metadata tag 316
- mqmdcharacterSet metadata tag 315
- mqmdcorrelationid metadata tag 316
- mqmdencoding
 - metadata tag 315
 - persistence 178
- mqmdencodingGet metadata tag 315
- mqmdformat 179
- mqmdformat metadata tag 315
- mqmdmessageType metadata tag 316
- mqmdpriority metadata tag 316
- mqMessageModel metadata tag 315
- mqOutput metadata tag 316
- mqPrimaryKeyGenerationType metadata tag 317
- mqseries persistent type 176, 310
- mqseriesxml
 - JXP MQ XML tab 185
 - persistent type 176, 185, 310, 311, 315
- mqStartingOffset metadata tag 316
- multiple services warning 59
- multiplicity
 - aggregation, 1 .. 1 235
 - aggregation, 1 .. n 238
 - in tutorial 257
 - tag 187
- multiplicity, metadata tag 308
- MyCustomObject 234

N

- name, specifying for association 233
- namespace 185
- network presence key 45
- new
 - operator, do not use 50
 - service tutorial 58
- Node 275, 281
- non-Chordiant web services 129
- nullable 189, 246
- number generation 46
- NumberGeneratorHelper 52

O

- object
 - base model 173
 - base object model 59, 245

- contained 234
- definition 6
- factory
 - and overrides 50
 - in business service 5
 - standard persistence business object behavior 299
- getContainedObject 205
- locking 169
- name, XML 185
- object-centric CRUD operations 216
- serialized, caution 165
- setContainedObject 204
- subclassing existing 174
- See also* business object
- objectIsEmpty 172
- Offering Service 337
- offerings
 - in EBC Interaction Service 332
 - in Offering Service 337
- offset, MQ 190
- OMG, Party Management Facility 303
- one-to-many relationship 238
- one-to-one relationship 235
- operation
 - creating new 60, 78
 - metadata 322
 - metadata tag 308
 - name metadata 323
- operation-level metadata
 - CMI 306
 - SMI 321
- optimistic locking 169, 175, 187
- oracle8 persistent type 176, 310
- Order Fulfillment Service 338
- Order Generation Service 340
- Order Tracking Service 342
- OrderNumberGenerator 52
- Organization 284
- organize imports 66
- output directories 264
- output\lib directory 68
- overloading
 - behaviors 7
 - methods in services 51
- override, metadata tag 167, 308
- overriding
 - access strategy 221

- and object factory 50
- behavior 7, 223
- business object behavior class 301
- business object class 6, 167
- typeField 230
- typeValue 229
- overwriting code by Business Component Generator 49, 63, 168

P

- package, metadata tag 308
- PACKAGE_NAME 48
- packages, location of services 45
- parameter
 - name metadata 323
 - type metadata 323
- parentClass, metadata tag 308
- parse, MQ 191
- Party 284
- Party Management Facility
 - customization example 303
 - object model 274
 - overview 273
- PartyManager 292
- PartyRelationship 275
- PartyRole
 - architecture diagram 297
 - base class 275
 - customizations 297
 - interface 288
 - marriage 295
 - PmfCustomerService 303
 - Service 45, 273, 343
- PartyRoleManager 294
- PayloadData in sequence flow 196
- peer service 45
- Perform build automatically 18
- persistence
 - business objects 6
 - business service interaction 216
 - components 3
 - files created automatically 163, 218
 - metadata
 - overview 310
 - persistent type 175
 - services 5
 - specifying metadata 174

- persistent
 - cache manager, caution 165
 - mapping 6
 - type
 - in Rose 175
 - in tutorial 248
 - metadata 310
 - type description 176
- persistentType metadata tag 310, 314
- Person 284
- pessimistic locking 169, 175, 191
- PmfCustomerClientAgent 345, 349
- PmfCustomerService
 - customization of PartyRole 303
 - package 45
- PmfDelegateService 350
- populateCriteria 172
- PostalAddress 275
- precision attribute 6
- primary key
 - generation 246
 - in tutorial 246
 - specifying in metadata 312
 - specifying in Rational Rose 188
- private attributes 6
- process flow, task descriptor 151
- processRequest method 44
- production environment, deploying web services 113
- production runmode 214
- prospect, converting to customer 296
- provided
 - models 41
 - web services 90
- proxy code
 - creating files 103
 - dynamic invocation with SAAJ 144
 - incorporating for external service 132
 - project 91
 - web services 88, 124
- proxy code project 91
- public attributes 6

Q

- QueueItem Model 23

R

- Rational Rose
 - configuring for Java components 9
 - error 12
 - MOF 9
 - UML 9
 - UML Extender 2
 - XMI 9
- rdbDigits metadata tag 311
- rdbLogicalType metadata tag 311
- rdbNotNull metadata tag 312
- rdbPhysicalName metadata tag 310, 311
- rdbPrimaryKey metadata tag 312
- rdbPrimaryKeyGenerationType metadata tag 312
- rdbSize metadata tag 311
- recursive process 196, 205
- referenced classes 12
- references, updating in tutorial 82
- refreshing configuration 210
- relationship
 - marriage 295
 - one-to-many 238
 - one-to-one 235
 - specialization of Role 275
- RelationshipManager 294
- relocate code to prevent overwriting 49, 168
- removeNote 291
- removeParty 293
- removeRelatedObject 281
- removeRelatedPartyRole 291
- removeRole 283
- removing attributes discouraged 6
- request messages, MQ 191
- RequestReply 178
- requests, in EBC Interaction Service 332
- requirements, web services 96
- Resource Manager
 - business object behavior 300
 - configuration settings 44
 - getBehaviorForName method 50
 - vends business object behaviors 50

- retrieve
 - efficiency of lookups 227
 - extended persistence behavior 199
 - inheritance hierarchy 227
- retrieveAllCustomersByLastName 207
- retrieveAllMatching
 - extended persistence behavior 201
 - inheritance hierarchy 227
 - with xrefTable 227
- retrieveAllMatchingGraphs 201
- retrieveAssociation 202, 232
- retrieveAssociation(String) 202
- retrieveAssociationGraphs(String) 203
- retrieveAssociationTypes 207
- retrieveGraph
 - aggregation 234
 - extended persistence behavior 200
- retrieveGraphToDepth 200
- retrieving graph as single entity 172
- return type, metadata 323
- RmaNumberGenerator 52
- Role
 - A Detail 235, 238
 - A, foreign key 232
 - B, foreign key name 232
 - Husband 295
 - in Party/Role 275
 - interface 279
 - Wife 295
- RoleManager 293
- root name 185
- RpcEnc suffix 100
- RPC-style web services 90
- runmode 214
- runtime environment, setting up 68

S

- SAAJ
 - invocation of web services 117
 - proxy code 144
- Samples project 118
- schema
 - defined automatically 164
 - generation and JDK 1.4 26
 - running the SQL script 264
 - XSD created automatically 164

- scripts
 - ebs 10
 - generating business components 32
 - web services generation 97
- Secure Sockets Layer, with web services 91
- security
 - getSecurity 205
 - ISecurityInformation 212
 - setSecurityInformation 205
- seeding data 271
- sequence diagram
 - standard persistence business object behavior 299
- serialized objects, caution 165
- server-config.wsdd
 - deleting 103
 - deploying web services 106
 - editing 111
 - listing deployed web services 113
- service
 - adding behaviors 7
 - as controller 44
 - base model 41
 - behaviors 7
 - BusinessDataServiceBaseClass 5
 - callback 45
 - calling peer service 45
 - calls business object behavior 50
 - configuration 5
 - definition 5
 - domain logic 44
 - exploring 322
 - façade 216
 - files created automatically 43
 - framework components 3
 - Generic Service 194
 - logic 46, 50
 - modifying existing 46
 - overloading
 - behavior 7
 - methods 51
 - overriding behavior 7
 - package 45
 - persistence 5
 - persistence interaction 216
 - service to service 45
 - skeleton creation 43

- standard persistence behavior 298
 - subclassing existing 47
 - transactions 45
 - tutorial 57
 - using object factory 5
 - using with Generic Service 215
 - warning about multiple 59
- Service Base Model 23
- Service Creation Tutorial Model 23
- Service Extension Tutorial Model 23
- Service Metadata Information. *See* SMI
- service wrapper for web services 115
- serviceconfig.xml 44
- ServiceLocator.java 106
- setAdditionalData 206
- setAuthentication 204
- setContactInformation 287
- setContainedObject 204
- setGenericServiceInstance 205
- setID 205
- setSecurityInformation 205
- setToBeDeleted 205
- setup
 - method, resource manager 44
 - not in business object behaviors 300
- setUserName 204
- seurity, web services 91
- size
 - attribute 6
 - column, specifying 188
 - in tutorial 246, 250
- Skeleton.java 106
- slow generation 19
- SMI
 - case-sensitivity 321
 - directory 264
 - metadata, attribute-level 321
 - metadata, class-level 321
 - metadata, operation-level 321
 - modeling overview 2
 - overview 321
 - sample 323
 - services metadata 322
- SOAP with Attachments API for Java. *See* SAAJ
- SoapBindingImpl.java 106
- SoapBindingStub.java 106
- SoapConnectionFactory, web services 145
- SOAP-encoded message 165
- SoapFactory, web services 145
- sockets, no access to behavior methods 265
- SQL attribute-level metadata 188
- SQL class-level metadata 177
- SQL persistence metadata
 - data source name 310
 - lock field 311
 - lock strategy 311
 - persistent type 310
 - relational database digits 311
 - relational database logical type 311
 - relational database not null 312
 - relational database physical name 310, 311
 - relational database primary key 312
 - relational database primary key generation type 312
 - relational database size 311
 - sample 313
 - tags 310
 - where prefix 311
 - XMLNamespacePrefix 311
 - XMLObjectName 311
 - XMLRootName 311
 - XMLType 311
- SQL script, running 264
- SSL. *See* Secure Sockets Layer
- state, business object behaviors and 300
- stateful task descriptor 152
- static
 - metadata tag 309
 - web service invocation 117
 - WSDL distribution 97
- stereotype 240
- stock quote web service 129
- strategy, locking 170
- strategy. *See* access strategy
- STUBTYPE 48
- style sheets 2
- subclassing
 - business object 166
 - existing business objects 174
 - existing service 47
 - from base object class 246
 - new class in tutorial 252

- symbol, copyright 12
- synchronizing
 - data 173
 - database and objects 168
- System Task Processor Service Model 23

T

- table
 - inheritance examples 226
 - name, specifying 177
- targets, Ant-based generation 37
- task descriptors
 - _tdgen_client target 155
 - _tdgen_java target 157
 - antcall 157
 - bindings 153
 - excluded APIs 155
 - GenerateTaskDescriptor tool 153
 - generate-tasks.xml 154
 - intermediate WSDL file 155
 - list of provided 327
 - location 151
 - overview 151
 - post service generation 44
 - types 152
- taxcalculator tutorial 57
- tdgen_client target 155
- tdgen_java target 157
- Telephone 275
- testers
 - creating 270
 - customizing 271
 - duplicating in tutorial 85
 - in service tutorial 68
- TestSuite.java 270
- THIRD_PARTY_LIB 18
- time stamps 277
- toFieldValuePairs 204
- token, authentication 48
- topology
 - web services design time 88
 - web services runtime 89
- toString 204, 206
- transaction
 - and services 215
 - locking within a block 169
 - performing 45

- TransferAttributesHelper 172
- transferLikeAttributes 172
- transferLikeAttributesAndSetCriteria 173
- transformation 2
- tutorials
 - creating a service 57
 - extended persistence 243
 - Extended Persistence Tutorial Model 23
 - extending a service 76
 - invoking Chordiant web services 120
 - Non-Chordiant web services 129
 - Service Creation Tutorial Model 23
 - Service Extension Tutorial Model 23
 - services 57
 - web services 117
- TX_SEMANTICS 48
- type
 - association 232
 - error 12
 - getType 206
 - logical 189
 - lookupClassByType 206
 - persistent 175
 - XML 185
- typeField
 - example code 231
 - specifying 230
- typeValue
 - in tutorial 250
 - specifying 229

U

- UML Extender for Rational Rose 2
- undeploying web services 108
- unidirectional aggregation 256
- unilateral aggregation 258
- unique number generation 46
- UnsupportedOperationException 206
- update method 203
- updateCommonObject 278
- updateGraph 203
- updateParty 293
- updatePoint 172
- updatePointOptimistic 172
- updateRay 172
- updateSegment 172
- updateSet 172

- URL for dynamic loading 214
- usage model for extended persistence 215
- useBehavior
 - extended persistence behavior 223
 - JXC CMI tab 224
- user name
 - getUsername 205
 - required for service API 48
 - setUserName 204

V

- validator
 - factory 209
 - overview 209
- ValidatorFactoryConfiguration.xml 209, 219
- Vector, in association 237
- Vehicle class created 246
- vehicleBehaviorTest.java 271
- vehicletax tutorial 57
- visibility, metadata tag 309, 323

W

- WAR file, web services 113
- warning, multiple services 59
- web services
 - authentication for 91
 - best practices 115
 - calling external 115
 - Chordiant samples 118
 - configuration through WSDD 89
 - configuring for 91
 - creating WSDL files 100
 - customizations note 98
 - deploying 106
 - deploying to production 113
 - designtime topology 88
 - distributing WSDLs 97
 - dynamic script 121
 - EAR file 113
 - external, using 129
 - generating proxy files 103
 - incorporating proxy code 124, 132
 - infrastructure 89, 99
 - invoking 117
 - Java code v. 115
 - listing 113

- non-Chordiant, using 129
- overview 87
- provided by Chordiant 90
- proxies
 - incorporating 124
 - to services 88
- proxy code project 91
- regenerating files 115
- requirements 96
- RPC-style 90
- runtime topology 89
- SAAJ overview 117
- Secure Sockets Layer 91
- security 91
- topology 88
- tutorials 117
- undeploying 108
- WAR file 113
- WSDD files 87
- WSDL files 87
- weblogic.jar 18
- WebServicesProxyProject 93, 119
- WebServicesSamples_Chordiant 119
- WebServicesSamples_Non-Chordiant 119
- where prefix 177
- WherePrefix metadata tag 311
- Wife 295
- workflow context, warning 165
- wrapper for web services calls 115
- WSDD files
 - creating 103
 - definition 87
- ws-deploy.xml 106
- WSDL files
 - definition 87
 - editing 111
 - generating 100
 - intermediate for task descriptors 155
 - regenerating 115
 - specifying location 100
 - static v. dynamic distribution 97
 - task descriptors 151
- ws-java2wsdl.xml 100
- ws-runChordiantSample.xml 126
- ws-run-nonchordiant-{application_server}.xml 142
- ws-undeploy.xml 109
- ws-wsdl2java.xml 103, 121
- ws-wsdl2java-dynamic.xml 112, 114, 121

X

Xalan, specifying version 19

XMI

- configuring for export 9

- export tip 62

- exporting model 15

- Rose 9

- slow code generation 19

XML

- namespace 185

- object name 185

- root name 185

- type 185

XMLNamespacePrefix metadata tag 311, 315

XMLObjectName metadata tag 311, 315

XMLRootName metadata tag 311, 315

XMLType metadata tag 311, 315

xrefTableName

- in tutorial 250

- specifying 228

XSD

- created automatically 164

- directory 264

