

BusinessWare Modeling Guide

BusinessWare Version 4.3.2
June 2006



Copyright © 1997-2006
Vitria Technology, Inc.
945 Stewart Drive
Sunnyvale, CA 94085-3913
Phone: (408) 212-2700
Fax: (408) 212-2720

All rights reserved. Printed in the USA.

Vitria, BusinessWare, Vitria Collaborative Application, VCA, VCML, Trading Partner Manager, and XEDI are trademarks or registered trademarks of Vitria Technology, Inc. All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

This document, as well as the software documented in it, is furnished under license and may only be used or copied in accordance with the terms of such license. This product includes technology protected by U.S. Patent No. 6,338,055 B1.

The information in this document is subject to change without notice.

TABLE OF CONTENTS

Preface	xxv
Audience	xxv
Related Reading	xxv
BusinessWare Documentation Set	xxvi
How this Guide Is Organized	xxviii
Conventions	xxx
Command-Line Notation Conventions	xxxi
Contacting Vitria	xxxii
Headquarters	xxxii
Technical Support	xxxii
Documentation	xxxii
Part I Introduction	
Chapter 1 Getting Started	1-1
Modeling BusinessWare Solutions	1-1
Types of Models	1-3
Integration Models	1-3
Process Models	1-4
Transformer Models	1-4
Process Query Models	1-5
Exception Maps	1-6
Communication in Models	1-7
Connectivity with External Systems	1-7
Resources	1-9
Types	1-9
Projects	1-10
Project Modules	1-11
BusinessWare Project	1-12
vtAFCommon Project	1-12

TABLE OF CONTENTS

Deploying BusinessWare Solutions	1-12
Debugging BusinessWare Solutions	1-13
Analyzing BusinessWare Solutions	1-13
Lifecycle of a BusinessWare Solution	1-14
Checklist for Creating BusinessWare Solutions	1-15
Samples and Tutorials	1-19
Chapter 2 Using the Modeling Environment	2-1
Starting the BME	2-1
Getting Help in the BME	2-1
Introduction to the BME Windows	2-2
Main Window	2-3
Explorer	2-5
Properties Window	2-6
Editor	2-9
Output Window	2-11
Quick Reference for Shortcuts in the BME	2-12
Part II Projects	
Chapter 3 Projects	3-1
Introduction to Projects	3-1
Project Content	3-1
Project Objects	3-2
Project Mountpoints	3-3
Local Services	3-4
Remote Services	3-4
Project Properties	3-4
General Properties	3-6
Compilation Properties	3-7
Runtime Properties	3-8
Working with Projects	3-10
Creating a Project	3-10
Saving a Project	3-11
Opening and Renaming a Project	3-11
Deleting a Project	3-11
Sharing Projects	3-11

Project Synchronization	3-13
Exporting and Importing Projects	3-13
Using Project Modules	3-16
Removing a Project Module	3-17
Starting and Stopping a Project	3-17
Using the Project Manager	3-18
Objects and Files in the BME	3-19
Upgrading Objects	3-22
Versioning Projects	3-23
Managing Multiple Project Versions at Design Time	3-23
Runtime Version of a Project	3-23
Building and Compiling Projects	3-24
Compile and Build Behavior	3-25
Deploying a Project	3-25
Animating a Project	3-26
Debugging a Project	3-26
Chapter 4 Design Repository	4-1
Working with the Design Repository	4-1
Enabling the Design Repository	4-2
Publishing to the Design Repository	4-3
Using the Repository Explorer	4-3
Disabling the Design Repository	4-4
Part III Integration Models	
Chapter 5 BusinessWare Types	5-1
Overview	5-1
Document-Centric Project Design	5-2
Defining Types	5-3
Types Summary	5-3
Converting Existing Files to Types	5-5
Creating Defined Types	5-7
Inspecting and Editing Types	5-9
Exporting Types	5-10
Sharing Types	5-10
Connector Types	5-11

TABLE OF CONTENTS

Pre-Defined Types	5-11
Custom Generated Types	5-13
Chapter 6 Integration Model Basics	6-1
Introduction to Integration Models	6-1
Example	6-2
Integration Model Contents	6-3
Creating Integration Models	6-4
Displaying Options, Layers, Labels, and Annotations	6-7
Components	6-9
Types of Components	6-9
Setting Component Properties	6-10
Linking Components to Models	6-11
Linking Integration Components to Multiple Models	6-11
Connectors	6-12
Connector Types	6-12
Guidelines for Configuring Connectors	6-14
Editing Connector Port Event Types	6-14
Linking Connectors to Resources	6-14
Ports	6-15
Input and Output Ports	6-15
Port Properties	6-16
Name Property	6-18
Kind Property	6-18
Port Types Property	6-18
Request Map Property	6-19
Transaction Control Property	6-20
Type Propagation Property	6-20
Dynamic Port Property	6-20
Port Interface Definition	6-21
Port Modeling Constraints	6-26
Adding and Removing Ports	6-26
Wiring Ports	6-26
Matching Interfaces	6-27
Matching Ports	6-28
Best Practices	6-28
Proxies	6-28

Proxy Types	6-29
Using Proxies to Communicate Externally	6-29
Simple Input Proxies	6-29
Simple Output Proxies	6-30
To Add Proxies to an Integration Model.....	6-31
Adding a Proxy to an Integration Model.....	6-32
Proxy Properties.....	6-33
Using Proxies with the Dynamic Map Class	6-35
Working with Integration Models.....	6-37
Designating the Root Integration Model	6-37
Using Nested Integration Models	6-37
Nesting Specification.....	6-40
Reusing Integration Models.....	6-40
Printing Integration Models	6-41
Viewing Large Models	6-41
Chapter 7 Channels and Queues	7-1
Overview	7-1
Asynchronous Communication	7-2
Using Channels and Queues	7-4
Quality of Service	7-6
Comparing Channels and Queues	7-6
Channels and Publish-Subscribe Communication	7-7
Queues and Point-to-Point Communication	7-7
Destination Considerations	7-8
About Channels.....	7-9
Channel Permissions	7-9
Channel Types and Resources.....	7-10
Basic Channel	7-10
BW3x Channel Shortcut	7-10
Composite Channels.....	7-11
Replica Channels.....	7-12
Channel Resources	7-12
Setting Properties for Channel Resources	7-13
Channel Connectors	7-15
Adding a Channel Connector	7-16
Setting Properties for Channel Connectors.....	7-16

TABLE OF CONTENTS

Configuring the Channel Connector for Dynamic Connectivity . . .	7-17
About Queues	7-18
Queue Resources	7-19
Setting Properties for Queue Resources	7-19
Queue Connectors	7-20
Adding a Queue Connector	7-21
Setting Properties for Queue Connectors	7-22
Configuring the Queue Connector for Dynamic Connectivity . . .	7-23
Queue Permissions	7-23
Viewing Event Data on Channels and Queues	7-24
Channel and Queue Connector Exceptions	7-24
Channel Connector Exceptions	7-24
Queue Connector Exceptions	7-25
Chapter 8 Application Server Integration	8-1
Synchronous Integration	8-3
Receiving Requests from a J2EE Application Server	8-4
Making Requests on EJBs in a J2EE Application Server	8-5
IIOP/GIOP Functionality	8-7
Security Considerations	8-7
Asynchronous Integration	8-7
Chapter 9 Web Services	9-1
Introduction to Web Services	9-1
Interoperability	9-2
Supported Standards	9-2
Additional Resources	9-3
Overview of Web Services Objects	9-3
Overview of WSDL Model and UDDI Model	9-4
WSDL Model	9-4
UDDI Model	9-4
Registration and Unregistration	9-5
Mapping the wsdl:portType to the uddi:tModel	9-5
Mapping the wsdl:binding to the uddi:tModel	9-6
Mapping the wsdl:service to the uddi:businessService	9-6
Discovery and Invocation	9-7
Web Service Input Proxy	9-8

Interfaces	9-9
Web Service Input Proxy Properties	9-10
Configuring a Web Service Input Proxy	9-12
HTTP Compression	9-13
Web Service Output Proxy	9-13
Web Service Output Proxy Properties	9-15
Configuring a Web Service Output Proxy	9-18
Configuring the Web Service Output Proxy to use HTTPS/SSL	9-22
Using Handlers	9-22
Sending HTTP Transport Headers	9-24
Exporting a Web Service	9-25
WSDL Dependencies	9-25
Exporting a Web Service from the Project	9-26
Exporting a Web Service from a Web Service Input Proxy	9-27
Exporting a Web Service from an Input Port	9-27
Invoking a Web Service Using the Web Service State	9-27
Using the Wizard to Configure a Web Service State	9-28
Sample Files	9-28

Part IV Process Models

Chapter 10	Process Models: Basic Concepts	10-1
	Introduction to Process Models	10-1
	Example	10-2
	Statechart Diagrams	10-2
	States	10-2
	Transitions	10-3
	Events	10-4
	Ports	10-4
	Exceptions	10-4
	Workflow Models	10-5
	Stateful and Stateless Models	10-5
	Stateless Models	10-5
	Stateful Models	10-5
	Summary of Differences between Stateless and Stateful Models	10-6
	Process Models and Process Components	10-6

TABLE OF CONTENTS

Process Model Link	10-7
Port Synchronization.....	10-7
Business Objects Used in Models	10-8
BPOs	10-8
Data Objects	10-8
Local Data	10-8
Creating Process Models.....	10-9
Before You Start	10-9
Procedure	10-9
Customizing the Display	10-12
Local Settings.....	10-12
Options	10-13
Setting Model Properties	10-14
Adding Action and Condition Code	10-15
Working with Process Models	10-16
Reusing Process Models	10-16
Using Prebuilt Process Model Templates	10-17
Debugging Process Models.....	10-17
Monitoring Process Models	10-18
Printing Process Models	10-18

Chapter 11 **Process Models: Defining and Using Business Objects** 11-1

Using Business Objects in a Process Model.....	11-1
BPO	11-1
DataObject.....	11-4
Local Data	11-5
Extending the Local Data Interface.....	11-6
Defining Business Objects.....	11-7
Defining Complex BPO, DO, and Local Data	11-8
Defining Objects with Struct Attributes	11-9
Defining Objects with Custom Methods.....	11-10
Defining Objects with DataObject Attributes	11-10
Defining Objects with a Sequence Attribute.....	11-11
Large Data Values.....	11-11
Performance Consideration for Persistence when Designing Business Objects	11-12
Structs versus DOs	11-12

Sequences	11-14
Initializing Attributes for BPOs and DOs	11-14
Adding Business Object Definitions to a Project.....	11-14
Coding a Java Implementation for a Business Object	11-15
Setting Business Object Properties.....	11-17
Process Component Properties: Object Table and Local Data Class..	
11-18	
Process Model Properties: BPO and Local Data Interfaces ..	11-19
Chapter 12 Process Models: Ports, States, and Transitions	12-1
Ports	12-1
Input and Output Ports	12-1
Query Ports	12-2
Guidelines for Configuring Ports	12-2
States	12-3
Types of States	12-3
Entry Actions and Exit Actions	12-7
State Properties.....	12-8
Adding and Configuring States	12-10
Transitions.....	12-11
How Transitions Work	12-11
Types of Transitions	12-13
Transition Properties	12-13
Adding and Configuring Transitions	12-14
Special Requirements for Action States	12-15
Default Transitions and Model Actions	12-16
Default Model Action	12-16
Default Transitions	12-17
Deciding Which Transition Triggers	12-18
Use of the defaultEvent on Chained Action States.....	12-19
Example of a Stateful Process Execution.....	12-20
Chapter 13 Process Modeling Techniques	13-1
Timers	13-1
Relative and Absolute Timers	13-2
Setting a Timeout	13-2
Forks	13-5

TABLE OF CONTENTS

Conditional Forks.....	13-6
Joins	13-7
Nested Models.....	13-9
Nesting Terminology	13-10
Understanding Nesting	13-10
Example	13-11
Role of Terminator States in Nested Models	13-13
Ports on Parent Models and Submodels	13-14
Creating Nested Models	13-14
Nesting Type.....	13-15
Iteration	13-20
Concurrent Processing in Models	13-22
Decision Nodes and Decision Trees	13-23
Request Map Class: Dispatching Events and Assigning BPOs.....	13-24
Default Request Map and the BPID Parameter.....	13-24
.....	13-25
RequestMapImpl and Stateless Models	13-25
Request Map Methods.....	13-26
Custom Request Map Implementation.....	13-28
Chapter 14 Process Model Code Construction	14-1
Overview	14-1
Models and Source Code	14-1
Tools for Constructing Code	14-2
Relationship between Code Builders and Editors	14-3
How to Use Action Code	14-3
Accessing the Code Builders	14-4
Accessing Code Builders from the Context Menus	14-4
Accessing Code Builders by Double-Clicking	14-5
Using the Action Builder	14-6
Using the Action Editor	14-10
Using the Condition Builder.....	14-13
Using the XQuery Builder	14-16
Using the Source Code Editor	14-19
Action Categories and Common Actions.....	14-22
Model Variables	14-28
Initializing a Project and Providing Project Global Data.....	14-29

What to Implement in Your Project Init Class	14-30
Accessing Global Data in Your Process Model Code	14-31
Example Implementation.....	14-32
Sample Action Code	14-33
RequestResponseLib vs. ResponseListenerLib	14-36
RequestResponseLib	14-36
ResponseListenerLib.....	14-36
Chapter 15 Process Model Templates	15-1
Router Templates	15-1
RouteByPortName Router	15-3
RouteByPortType Router	15-3
Adding a Router Model	15-4
SimpleTransformer Template	15-4
Adding a SimpleTransformer Template.....	15-5
SimpleXQueryTransformer Template	15-6
Adding a SimpleXQueryTransformer Template	15-6
Creating a Process Model Template.....	15-7
Chapter 16 Workflow	16-1
Workflow Concepts	16-1
Activities	16-1
Subtask Workflow	16-2
Workflow Participants	16-3
BusinessWare Workflow UI.....	16-5
Constructing a Workflow Model	16-5
Process Modeling with Activity States.....	16-6
Activity State Properties	16-7
Dynamically Modifying Properties at Runtime	16-10
Setting Labels for Activity States	16-10
Reference Data Property Settings.....	16-11
BPO Reference Data	16-12
Activity Reference Data	16-14
Creating Role Resources	16-15
Mapping Roles	16-16
Transitions Out of Workflow Activity States.....	16-18
Checking Reference Data Input.....	16-19

TABLE OF CONTENTS

Checking for Approval	16-20
Checking Timeout.....	16-21
Performer and Supervisor Properties	16-22
Performer.....	16-22
Performer Assignment Policy.....	16-23
Supervisor	16-24
Expert Property Settings	16-24
Reassignable	16-24
Task Complete Class Name.....	16-25
Dynamically Setting Performers and Supervisors	16-25
Setting Up the Task Manager Project	16-26
Task Manager Configuration and Deployment	16-26
Advanced Task Manager Configuration.....	16-31
Workflow Performance Tuning	16-33
Configuring a Single Task Manager for Maximum Performance .	16-33
Creating Multiple Task Managers	16-36

Part V Process Analysis and Monitoring

Chapter 17 Process Query Models	17-1
Introduction to Process Query Models.....	17-2
Snapshot Process Query Models	17-2
Real-Time Process Query Models.....	17-2
Process Query Components	17-3
Ports.....	17-3
Creating Process Query Models	17-4
Prerequisites	17-4
Procedure	17-4
Setting Process Query Properties	17-6
Defining Queries	17-7
Query Types	17-7
Using the Query Builder.....	17-9
Building Expressions	17-13
Using the Query Editor.. .	17-14
Generating Types	17-15
Deploying Process Query Models and Components.....	17-16
Customizing SQL Queries	17-17

Retaining Query Customizations	17-20
Purging the Aggregation Results Table.....	17-20
Deploying Clusters Containing Queries	17-20
Runtime Behavior of Process Query Models	17-20
Running the Snapshot Process Query Model.....	17-21
Running the Real-Time Process Query Model	17-21
Creating Update Events	17-22
Processing Update Events	17-23
Using a Downstream Process Component	17-23
Using a Subscriber Application	17-24
Visualizing Update Events.....	17-24
Query Performance at Runtime	17-25
Handling Complex BPO Data	17-25
Filtering and Aggregating Sequence Attributes.....	17-26
CLOB Data Type Limitations.....	17-26
Union Data Type Limitations	17-26
Limitations of Multiple BPOs Sharing Data Objects	17-26
Parameterized (Drilldown) Queries	17-27
Chapter 18 Process Query Language.....	18-1
An Introduction to Queries	18-1
Examples of BPO and Query	18-2
Source and Target Schemas	18-3
Query Components.....	18-4
Aliases.....	18-4
SELECT Clause	18-4
FROM Clause	18-4
WHERE Clause.....	18-5
Quantifiers	18-5
GROUP BY Clause	18-6
HAVING Clause	18-6
Notes on Selected Operators and Attributes.....	18-6
at Operator and bpState	18-6
in Operator and Membership in a Collection	18-7
like Operator and Wildcard String Matching	18-7
Using the oid Attribute	18-8
Query Types	18-8

TABLE OF CONTENTS

Simple Filter Query	18-8
Aggregation Query.....	18-9
Restrictions on Aggregation Queries	18-10
Time-Based Query.....	18-11
Time-Based Filtering.	18-11
Time-Based Aggregation	18-12
Time-Based Filtering with Aggregation.....	18-12
Limitations.	18-12
Filtering and Aggregating Sequence Attributes.....	18-13
Examples	18-13
Query Grammar.....	18-15
Chapter 19 Process Views	19-1
Overview	19-1
Creating Process Views.....	19-2
Creating a Process Query	19-2
Selecting a Folder	19-3
Organizing Views	19-3
Creating a New View	19-3
Deleting Views	19-4
Renaming Views	19-4
Copying Views	19-4
Configuring Process View Properties	19-5
Process Query Model Link.....	19-5
Query	19-6
Integration Model Link	19-6
Component	19-7
Drilldown View	19-7
Resource File.....	19-8
Top Level Visible	19-9
Description	19-9
View Permission	19-10
Configuring Display Properties	19-10
General Attribute Properties	19-11
Name	19-12
Label.....	19-13
Type	19-13

Color	19-13
Expert Attribute Properties	19-13
Date/Time Format	19-14
Numeric Format	19-15
Resource Prefix	19-15
Model State Filters	19-16
XSLT Renderer	19-17
Chart Properties	19-18
CLOB Support	19-20
Displaying an XML File	19-20
Rendering Views	19-21
Adding Custom Renderers	19-21
Using Style Sheets	19-22
Editing a Style Sheet	19-28
Using Single Sign-On	19-28
Enabling Single Sign-On	19-29
Validating Views	19-29
Deploying Views	19-30
Directory Server Namespace	19-30
Security	19-31
Log Files	19-31

Part VI Deployment

Chapter 20 Integration Servers	20-1
Integration Server Resource	20-1
Configuring Integration Servers	20-2
Setting Trace Levels	20-4
Connection Manager	20-6
Service Trace Levels	20-6
Transaction Manager	20-7
Databases	20-8
XA Configuration for Oracle	20-9
Loggers	20-10
Logger Properties	20-12
Web Server	20-13
Web Applications Directory Structure	20-15

TABLE OF CONTENTS

Chapter 21	Load Balancing	21-1
	Introduction to Load Balancing	21-1
	Designing Load Balanced Projects	21-2
	Synchronous Load Balancing	21-2
	Synchronous Call Routing	21-4
	Asynchronous Load Balancing	21-6
	Asynchronous vs. Synchronous Load Balancing	21-8
	Asynchronous Load Balancing	21-8
	Synchronous Load Balancing	21-9
	Configuring a Project for Load Balancing	21-10
	Creating Clusters	21-11
	Configuring a New Cluster	21-13
	Creating Clusters with Web Service Input Proxies	21-14
	Editing Clusters	21-14
	Updating a Deployed Cluster	21-15
	Deleting Clusters	21-16
	Debugging and Animating Clusters	21-16
	Injecting Events	21-16
	Viewing Logs	21-16
	Administering Clusters	21-17
	Sharing Clusters	21-17
	Error Handling	21-17
Chapter 22	Deploying Projects	22-1
	Introduction to Deployment	22-1
	Directory Server and Deployment	22-3
	Schema and Namespace in the Directory Server	22-3
	Deployment Process	22-5
	Before You Begin	22-5
	Deployment Options	22-6
	Creating a Deployment Configuration	22-7
	Partitioning Components and Resources	22-8
	Specifying the Default BusinessWare Server	22-9
	Adding a BusinessWare Server	22-9
	Auto-Partition or Manually Partition	22-10
	Unpartitioning	22-11
	Deploying a Partitioned Project	22-12

Deployment Using the BME	22-12
Using the Project > Start Method	22-13
Using the Project Deploy Method.....	22-14
Stages of Deployment	22-15
Deployment Using vtadmin	22-16
Process Query Deployment	22-17
Executable Project Deployment	22-18
Role Deployment.....	22-19
Undeploying a Project.....	22-19
Channels and Queues.....	22-20
Integration Servers	22-21
Chapter 23 Project Packaging.....	23-1
Introduction to Project Packaging	23-1
Project Parameter Settings	23-3
Properties	23-4
Specifying Project Parameters	23-4
Creating Parameter Groups	23-5
Creating Simple Parameters	23-6
Mapping Parameters.....	23-7
Parameterizing Integration Server Properties	23-7
Parameterizing Nested Process Model Properties.....	23-9
Setting Parameter Values	23-9
Creating Substitution Parameters	23-10
Getting Project Parameters at Runtime.....	23-11
Distributing a Project.....	23-11
Performing a Custom Project Installation	23-12
Generating a Project Parameter XML File	23-13
Project Parameter XML Contents	23-13
Installing a Custom Project	23-13
Part VII Runtime	
Chapter 24 BusinessWare Runtime Architecture	24-1
Runtime Architecture.....	24-1
Processes and Hosts	24-1
Containers.....	24-4

TABLE OF CONTENTS

Components and Ports	24-6
Project Startup	24-7
Activation	24-7
Runtime Services	24-8
Persistence	24-8
Transaction Management	24-8
Connection Management	24-9
Web Server and Servlet Engine	24-9
Security	24-10
Logging	24-11
Chapter 25 Transaction Management	25-1
Transactions Overview	25-1
Transaction Properties	25-3
Transaction Types	25-4
Single-Server	25-4
Multiple-Server	25-5
Transaction Protocols	25-5
Resource and Transaction Managers	25-5
Two-Phase Commit	25-7
One-Phase Commit	25-9
BusinessWare and Transaction Standards	25-10
Transactions and Modeling	25-11
Connector Transactionality	25-11
Transaction Scopes	25-12
Definition	25-13
Batch Commit Limits	25-14
Designing with One-Phase Resources	25-14
Transactions and Integration Servers	25-16
Transaction Manager Properties	25-16
Transaction Control Property	25-17
Transaction Propagation Using OTS	25-17
Integration Server to Integration Server	25-18
BusinessWare Integration Server to Application Server	25-18
Application Server to BusinessWare Integration Server	25-19
TRANSACTION PROPAGATION WITH WEBLOGIC	25-19

Chapter 26 Exception Handling.....	26-1
Exceptions Overview.....	26-1
Kinds of Exceptions.....	26-1
Application Exceptions.....	26-2
System Exceptions.....	26-2
Exception Sources	26-2
Component Exceptions	26-3
Fine-Grained Exception Handling	26-3
External Applications Exceptions via Proxies.....	26-4
Source and Target Connector Exceptions	26-4
Outband Exceptions	26-4
Common Actions for Exception Handling	26-5
Exception Transitions in Process Models	26-7
Types of Exceptions	26-8
Visualizing Error Paths	26-8
Exception Handler Overview.....	26-10
Default Project Exception Handler	26-11
Exception Handler Classes	26-11
ExceptionHandlerImpl Class	26-11
Designing a Custom Exception Handler Class	26-12
PortExceptionHandlerImpl Class.....	26-15
Exception Maps.....	26-16
Creating an Exception Map.....	26-16
Configuring Exception Map Properties.....	26-17
Defining Exception Mappings	26-17
Exceptions	26-18
Using Exception Maps.....	26-23
Additional Exception Handling Properties.....	26-24
Using Port Invocation APIs	26-25
Retrieving Output Ports from Component Exception Handler Classes ..	
26-25	
Retrieving Input Ports from Connector Exception Handler Classes ..	
26	
Retrieving Output Ports from Process Model Action Code	26-27
Chapter 27 Debugging and Animation	27-1
Overview of the Debugging Tools.....	27-2

TABLE OF CONTENTS

Debugging Procedure	27-2
Debugging Your Project Using Auto Deploy	27-3
Stepping Through Your Program	27-5
Debugging Your Project Manually	27-6
Debugging Configuration.....	27-7
Customizing Debug Settings	27-7
Debugger Window	27-9
Working with Breakpoints	27-11
Setting Breakpoints in Models	27-11
Setting Breakpoints in the Code.....	27-12
Enabling and Disabling Breakpoints.....	27-15
Removing Breakpoints	27-15
Working with Watches	27-16
Inspecting Threads	27-16
Inspecting CallStacks.....	27-18
Inspecting Variables.....	27-18
Inspecting Classes.....	27-18
Inspecting Sessions.....	27-18
Event Injector.....	27-19
Using the Event Injector.....	27-20
Using an Externally Created XML File.....	27-24
Event Inspector	27-25
BPO Inspector	27-26
Channel/Queue Inspector	27-27
Using the Channel/Queue Inspector	27-29
Displaying Events	27-30
Log Viewer.....	27-30
Viewing Logs	27-31
Configuring Loggers.....	27-33
Animation	27-34
Animating a Project Using Auto Deploy.....	27-35
Setting Your Project Manually for Animation	27-36
Customizing Animation	27-36
Clearing Animation.....	27-38
Pausing Animation.....	27-38
Animating Integration Models	27-38
Animating Process Models	27-39

Chapter 28 Application Services.....	28-1
Local Service Manager	28-1
Configuring the Service Manager	28-3
Inheriting Services	28-4
Configuring Services	28-4
Document Store Service	28-5
Logging Service.....	28-7
ID Correlation	28-10
Registry Service	28-11
Security Service	28-11
Accessing the Security Service Cipher	28-12
Remote Services.....	28-12
Service Requester Configuration	28-13
Service Provider Configuration	28-15
Synchronous and Asynchronous Endpoints	28-15

Part VIII Appendices

Appendix A Event Interoperability	A-1
C++ Standalone Publisher or Subscriber Interoperability	A-1
BusinessWare Versions 3.x Models	A-1
Appendix B Error, Message, and Tracing Framework	B-1
Overview	B-1
Resource Files	B-2
Resource-File Format	B-2
Example Resource-File Entries	B-3
Module and Message Identifiers	B-3
Message Format	B-4
Message Categories	B-4
Symbolic Message Names.....	B-5
Message Text-Strings	B-5
Message Parameter Format	B-6
Parameter Types	B-6
Modifiers.....	B-7
Localization Utilities.....	B-8
Architecture	B-9

TABLE OF CONTENTS

Stubs	B-10
Using EMT APIs	B-12
TRACE Messages	B-12
Trace Levels	B-12
Using the APIs for TRACE Messages	B-13
Guidelines for Adding Tracing Code	B-14
EXCEPTION Messages	B-15
Using the APIs for EXCEPTION Messages	B-15
Using the APIs for Exception-Handling	B-16
Label Messages	B-18
Using the APIs for LABEL Messages	B-19
Guidelines for Adding LABEL Messages	B-19
Sending Messages and Errors to Other Processes	B-19
Example	B-20
Connector Development	B-22
Message Compatibility Requirements	B-23
Index	Index-1

PREFACE

Welcome to the *BusinessWare Modeling Guide*.

This guide describes how to use BusinessWare to model, debug, and deploy your business application.

This preface contains the following information:

- [Audience](#)
- [Related Reading](#)
- [BusinessWare Documentation Set](#)
- [How this Guide Is Organized](#)
- [Conventions](#)
- [Contacting Vitria](#)

AUDIENCE

This guide is intended for business analysts, architects, and programmers who want to use the BusinessWare Modeling Environment to model, deploy, and debug business applications.

RELATED READING

Refer to the following documents for information related to the topics in this manual:

- [*BusinessWare Introduction*](#)
- [*BME Help*](#)

BUSINESSWARE DOCUMENTATION SET

The BusinessWare documentation set includes the following manuals:

Table 1 BusinessWare Documentation

Document	Description
Read these documents first	
<i>BusinessWare Introduction</i>	Introduces the BusinessWare product suite and architecture and describes the key concepts underlying BusinessWare. Read this manual first.
<i>BusinessWare Release Notes</i>	Provides current information on features, fixes, and known problems for BusinessWare. Read this document before installing BusinessWare.
<i>BusinessWare Installation Guide for Windows</i>	Explains how to install BusinessWare and supporting third-party products and how to perform initial configuration on your operating system. Read this manual before installing BusinessWare.
<i>BusinessWare Installation Guide for Unix</i>	Explains how to install BusinessWare and supporting third-party products and how to perform initial configuration on your operating system. Read this manual before installing BusinessWare.
BusinessWare documents	
<i>BusinessWare Administration Guide</i>	Explains how to administer and manage BusinessWare.
<i>BusinessWare Administration Help</i>	Describes how to perform basic administration tasks using the BusinessWare Web Administration tool.
<i>Application Administration Help</i>	Describes how to administer the registry service for trading partners; users, groups, and roles for your BusinessWare solutions; and the logging service. Access this online help system through the Application Administration UI.
<i>BusinessWare Security Guide</i>	Describes how to configure BusinessWare for security and user management.
<i>BusinessWare High Availability Guide</i>	Describes how to install and configure BusinessWare for a high availability cluster and how to integrate BusinessWare with cluster software.

Table 1 BusinessWare Documentation (Continued)

Document	Description
<i>BusinessWare Modeling Guide</i>	Describes the BusinessWare application integration platform that manages the integration lifecycle with an integrated development environment for designing, developing, deploying, testing, monitoring, and analyzing business solutions.
<i>BME Help</i>	Describes how to model, debug, and deploy projects in the BME. Access this online help system through the BusinessWare Modeling Environment.
<i>BusinessWare Programming Reference</i>	Documents BusinessWare's Java and C++ public APIs.
<i>Business Cockpit Guide</i>	Describes how to access and manage process views using the Cockpit Web interface.
<i>Business Cockpit Help</i>	Context-sensitive help that describes how to access and manage process views using the Cockpit Web interface.
<i>XQuery API Programming Guide</i>	Provides typical use cases and examples of XQueryLib APIs.
<i>BusinessWare Connector SDK Programming Guide</i>	Describes how to develop custom connectors for BusinessWare.
<i>BusinessWare Glossary</i>	Defines key terms associated with BusinessWare.
<i>Email Connector Guide</i>	Describes how to configure and use the Email Connector installed with this release of BusinessWare.
<i>File Connector Guide</i>	Describes how to configure and use the File Connector installed with this release of BusinessWare.
<i>FTP Connector Guide</i>	Describes how to configure and use the FTP Connector installed with this release of BusinessWare.
<i>HTTP Connector Guide</i>	Describes how to configure and use the HTTP Connector installed with this release of BusinessWare.

HOW THIS GUIDE IS ORGANIZED

This book contains the following sections:

Table 2 Contents of this Guide

Section	Contents
Part I: “Introduction”	
Chapter 1, “Getting Started”	Provides an overview of the process of developing, debugging, and deploying BusinessWare solutions. Explains key concepts and provides a checklist of major tasks.
Chapter 2, “Using the Modeling Environment”	Describes the main windows in the BME and how to use them.
Part II: “Projects”	
Chapter 3, “Projects”	Describes how projects are used to organize and manage data. Explains project versioning and project modules.
Chapter 4, “Design Repository”	Describes the BusinessWare design repository. Includes information on sharing projects and publishing to the design repository.
Part III: “Integration Models”	
Chapter 5, “BusinessWare Types”	Provides an overview of the BusinessWare Defined Types and Java Types.
Chapter 6, “Integration Model Basics”	Describes the role of integration models and how to create them, including an explanation of the various components and of component communications via ports and wires.
Chapter 7, “Channels and Queues”	Explains what channels and queues are, how they are used to allow asynchronous communication, and how to add channel and queue resources to projects. Also describes the channel connectors.
Chapter 8, “Application Server Integration”	Explains how BusinessWare integrates with servers in internal and external J2EE-compliant applications.
Chapter 9, “Web Services”	Provides additional information on the elements that comprise process models.
Part IV: “Process Models”	
Chapter 10, “Process Models: Basic Concepts”	Presents an overview of how process models are created and used.

Table 2 Contents of this Guide (Continued)

Section	Contents
Chapter 11, “Process Models: Defining and Using Business Objects”	Describes the business objects used while creating business processes.
Chapter 12, “Process Models: Ports, States, and Transitions”	Describes process components, the elements that comprise a process model.
Chapter 13, “Process Modeling Techniques”	Describes how to use nested models, concurrent models, timers, forks, joins, and other techniques.
Chapter 14, “Process Model Code Construction”	Describes the Action Code Builder, Action Editor, and Condition Builder. Provides sample action code.
Chapter 15, “Process Model Templates”	Describes the prebuilt process models supplied with BusinessWare. Explains how to save process models as templates for reuse.
Chapter 16, “Workflow”	Describes how to set up a workflow system to incorporate tasks performed by individuals and groups in the business process.
Part V: “Process Analysis and Monitoring”	
Chapter 17, “Process Query Models”	Describes how to use process queries to monitor and correct the quality of service delivered by your BusinessWare applications.
Chapter 18, “Process Query Language”	Describes the language used to write queries for monitoring quality of service.
Chapter 19, “Process Views”	Describes how to create process views that display business process data.
Part VI: “Deployment”	
Chapter 20, “Integration Servers”	Describes how to configure BusinessWare to send requests to and receive requests from J2EE-compliant application servers.
Chapter 21, “Load Balancing”	Describes how to configure BusinessWare to support load balancing.
Chapter 22, “Deploying Projects”	Explains how to configure, partition, and deploy projects for use in a production environment.
Chapter 23, “Project Packaging”	Describes how BusinessWare supports custom project installations.
Part VII: “Runtime”	
Chapter 24, “BusinessWare Runtime Architecture”	Describes the BusinessWare runtime architecture and services.

Table 2 Contents of this Guide (Continued)

Section	Contents
Chapter 25, “Transaction Management”	Explains what transactions are and how BusinessWare supports transaction processing.
Chapter 26, “Exception Handling”	Describes how BusinessWare components respond to runtime exceptions and how to modify or extend the default response.
Chapter 27, “Debugging and Animation”	Describes the BME testing and debugging tools and outlines the debugging procedure.
Chapter 28, “Application Services”	Describes how to configure the Application Framework services.
Part VIII: “Appendices”	
Appendix A, “Event Interoperability”	Describes BusinessWare 4 interoperability with standalone publishers and subscribers and BusinessWare 3 models.
Appendix B, “Error, Message, and Tracing Framework”	Describes the Error, Message and Tracing framework (EMT), and describes how to use it.

CONVENTIONS

This manual uses the following conventions:

- **Monospace**

Specifies file names, object names, and programming code.

Example: `C:\VTlicenses`

- ***Italics***

- Identifies a variable.

Example: `installdir/bin/unix_platform`, where

`installdir` indicates the directory where BusinessWare is installed and `unix_platform` indicates the Unix platform, such as Solaris, HP-UX, or AIX.

- Indicates a value that you must enter within examples and command syntax.

Example: `java com.vitria.rta.AnalyzerUtil -n servername`

- Introduces new terminology, highlights book titles, and provides emphasis.

Example: An *operation* is a defined action that is meant to be applied to a given class of software objects.

- **Bold**

Highlights items and indicates specific items in a graphical user interface (GUI).

Example: From the list, select the **Properties** item.

- Path names

- This document uses the variable *installdir* to indicate the directory where BusinessWare is installed. On Windows, the default installation directory is C:\Program Files\Vitria\BW42. On Unix, the default installation directory is /usr/local/bw42.

- While BusinessWare is supported on multiple operating systems, path names are displayed only in the Windows backslash format.

Example for Windows systems:

```
samples\communicator\javaflow\anypub.java
```

- If you are operating in a Unix environment, change to the forward slash format.

Example for Unix systems:

```
samples/communicator/javaflow/anypub.java
```

- Internet URLs follow the standard forward slash convention.

Example:

```
http://www.vitria.com
```

COMMAND-LINE NOTATION CONVENTIONS

This manual uses the following command-line notation conventions:

- { } Braces

Identifies a set of items from which you must choose one.

Example: `java com.vitria.rta.AnalyzerNamedViewUtil -n view { -i file name | -x file name }`

- [] Brackets

Identifies an optional item.

```
java com.vitria.rta.AnalyzerSchemaUtil -n schema-folder
{ [-o ODL-file ... -o ODL-filen | -i schema-file ]
[-x schema-file] }
```

- | Or Operator

Separates items from which you may choose one.

Example: `java com.vitria.rta.AnalyzerNamedViewUtil -n view { -i file name | -x file name }`

CONTACTING VITRIA

If you have any questions regarding Vitria or any of the Vitria products, please contact Vitria using the resources listed in this section.

HEADQUARTERS

Vitria Technology, Inc.
945 Stewart Drive
Sunnyvale, CA 94085
1-408-212-2700

<http://www.vitria.com>

TECHNICAL SUPPORT

Vitria Technical Support provides comprehensive support for all Vitria products through our Technical Support Web site:

<http://help.vitria.com> (login and password required)

If you need a login for help.vitria.com, please send your request to:

contractadmin@vitria.com

Each customer who has purchased a support contract has one or more designated individuals who are authorized to contact Vitria Technical Support. If you have questions about using the BusinessWare products, please have a designated person at your site open a case with Vitria Technical Support.

DOCUMENTATION

If you have any comments on the documentation, please contact the Technical Publications group directly:

documentation@vitria.com

We would like to hear from you.

PART I: INTRODUCTION

This part provides an overview of the developing, debugging, and deploying BusinessWare process. This part also explains key concepts and provides a checklist of major tasks. In addition it describes the main BME windows and tells how to use them.

Chapters include:

- [Getting Started](#)
- [Using the Modeling Environment](#)

PART #: TITLE HERE GOES HERE

1

GETTING STARTED

The BusinessWare Modeling Environment (BME) is an integrated development environment in which you develop, deploy, and debug automated business solutions. This chapter presents an overview of this process and provides a checklist of the major tasks involved. Topics include:

- [Modeling BusinessWare Solutions](#)
- [Deploying BusinessWare Solutions](#)
- [Debugging BusinessWare Solutions](#)
- [Analyzing BusinessWare Solutions](#)
- [Lifecycle of a BusinessWare Solution](#)
- [Checklist for Creating BusinessWare Solutions](#)
- [Samples and Tutorials](#)

MODELING BUSINESSWARE SOLUTIONS

In BusinessWare, business applications are developed by modeling systems graphically. Graphical models make it easy to define and visualize the structure and behavior of complex systems and identify the most efficient solution.

[Figure 1-1](#) shows two models used in a simple order processing system. These illustrations are taken from the Order Process Sample, which is provided with your BusinessWare installation.

GETTING STARTED
Modeling BusinessWare Solutions

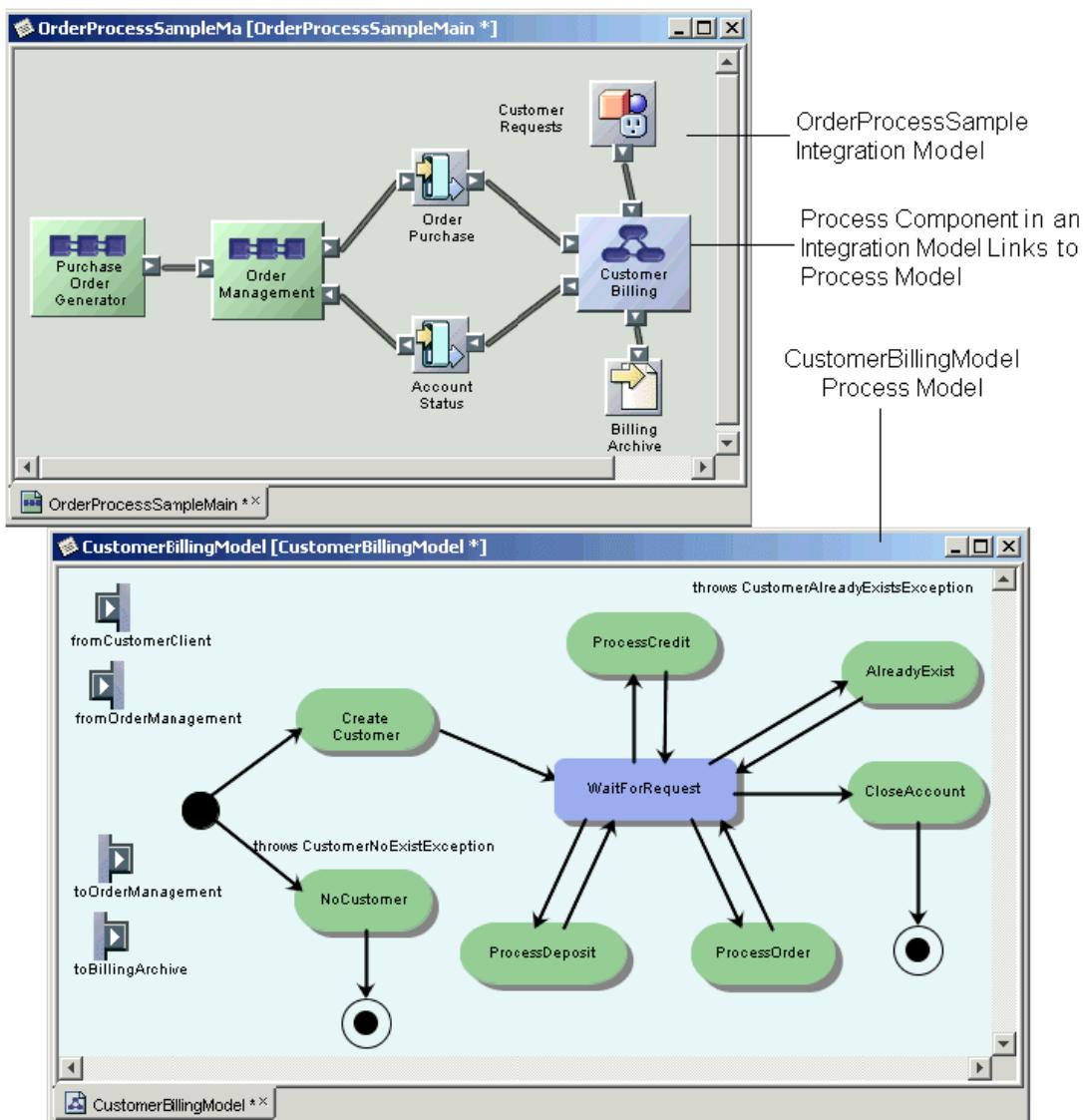


Figure 1-1 Root Integration Model and Linked Process Model in Order Process Sample

As you construct your models by dropping elements onto the canvas, the bulk of the application code is automatically generated. For the remainder, you have a choice of assisted code builders—which basically let you build code by selecting classes and methods from drop-down lists—or a full-featured Java code editor, which lets you write highly specialized, custom code.

TYPES OF MODELS

In designing a BusinessWare solution, you use different types of models for different purposes:

- Integration Models
- Process Models
- Transformer Models
- Process Query Models
- Exception Maps

Integration Models

The highest level model in BusinessWare is an *integration model*. Every BusinessWare solution has one *root integration model* that presents a system-level view of the application, showing the functional components and the means by which they communicate with one another and with external entities (such as a SAP system). The OrderProcessSampleMain model in [Figure 1-1](#) is an example of an integration model.

Nested in the root model may be other integration models that depict individual processes or major subsystems in detail. This hierachal approach not only lets you simplify the visual representation of large, complicated systems and make them more comprehensible; it also allows you to encapsulate subsystems for easy reuse.

Integration models consist of components, connectors, proxies, ports, and wires:

- *Components* represent subsystems within the solution and link to other models that define the structure and operation of those subsystems.
- *Connectors* provide connectivity between BusinessWare components and external entities, such as a file system, database, channel, or an SAP system. A basic set of connectors comes with BusinessWare; other connectors can be purchased separately.
- *Proxies* represent interactions and provide connectivity between components within a project and Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), Enterprise Java Beans (EJB), and Web service objects, outside the project.
- *Ports* and *wires* define the type and directionality of the communication paths between BusinessWare components and the types of data that can be exchanged.

For more information, see [Chapter 6, “Integration Model Basics.”](#)

Process Models

A process model is a statechart diagram that defines a business process. It embodies the business rules by which objects (such as a customer order) are processed across business systems in a well defined, logical manner. The CustomerBilling model in [Figure 1-1](#) is an example of a process model.

Process models consist of a sequence of *states*, which represent distinct phases in the process, and *transitions*, which define how the process instance moves from state to state. Some process models define business processes that are performed in a completely automated fashion with no human interaction. Others, called *workflow models*, coordinate computer tasks and human performer tasks.

A process model component in an integration model links to a process model.

For more information, see [Chapter 10, “Process Models: Basic Concepts”](#) through [Chapter 16, “Workflow.”](#)

Transformer Models

There are two transformer models in BusinessWare:

- **XQuery Transformer Model**—a specification that maps and converts XML data from one schema to another. It is used to convert incoming data to the format required by the next component in the system. The output event can be a byteArray, a stringEvent, or an xmlEnvelopeEvent.

For more information, see the *BusinessWare Advanced XQuery Transformer Guide*.

- **Transformer Model**—a specification that maps events of one type to events of another type. It is represented as a state within a process model. It is used to convert incoming data to the format required by the next component in the system.

For more information, see the *BusinessWare Transformer Guide*.

Process Query Models

Process query models are used to analyze the quality of service (QoS) being delivered by a process model and send information about any violations of service level agreements (SLAs) downstream for corrective action. [Figure 1-2](#) illustrates process query component. The downstream target can be any type of BusinessWare component.

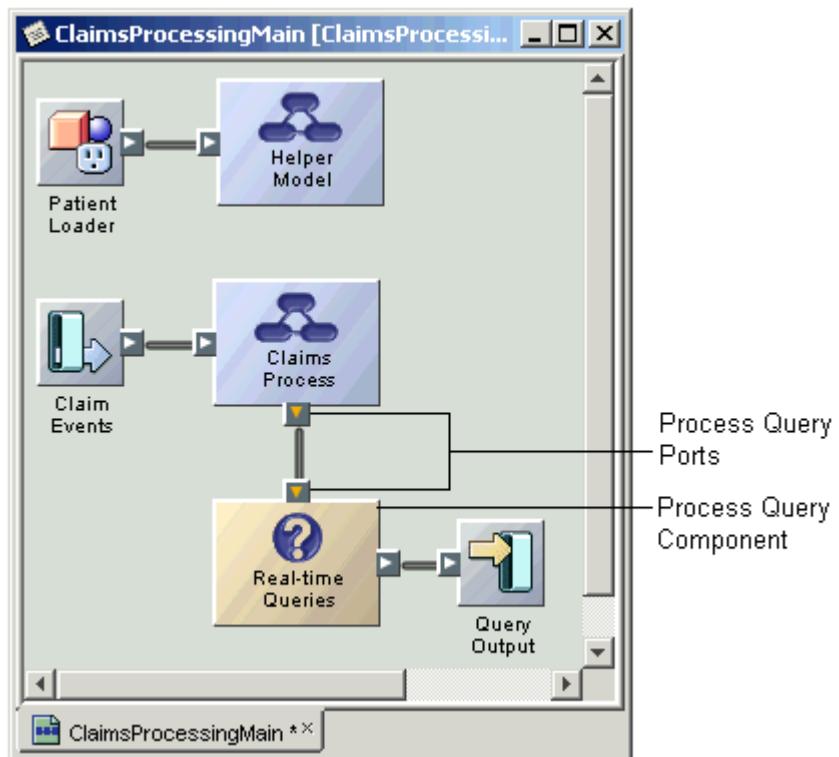


Figure 1-2 Process Query Model

A process query model consists of a set of queries that are run against the business process objects (BPOs) being processed by the process model. Each query checks for certain conditions that violate the QoS constraints and specifies what data is to be passed onto the target if those conditions occur.

[Figure 1-3](#) shows the Query Builder and Query Editor from the Request for Quote Sample. The query shown results in an update event being generated whenever an order larger than 100 is placed for any customer.

For more information, see [Chapter 17, “Process Query Models.”](#)

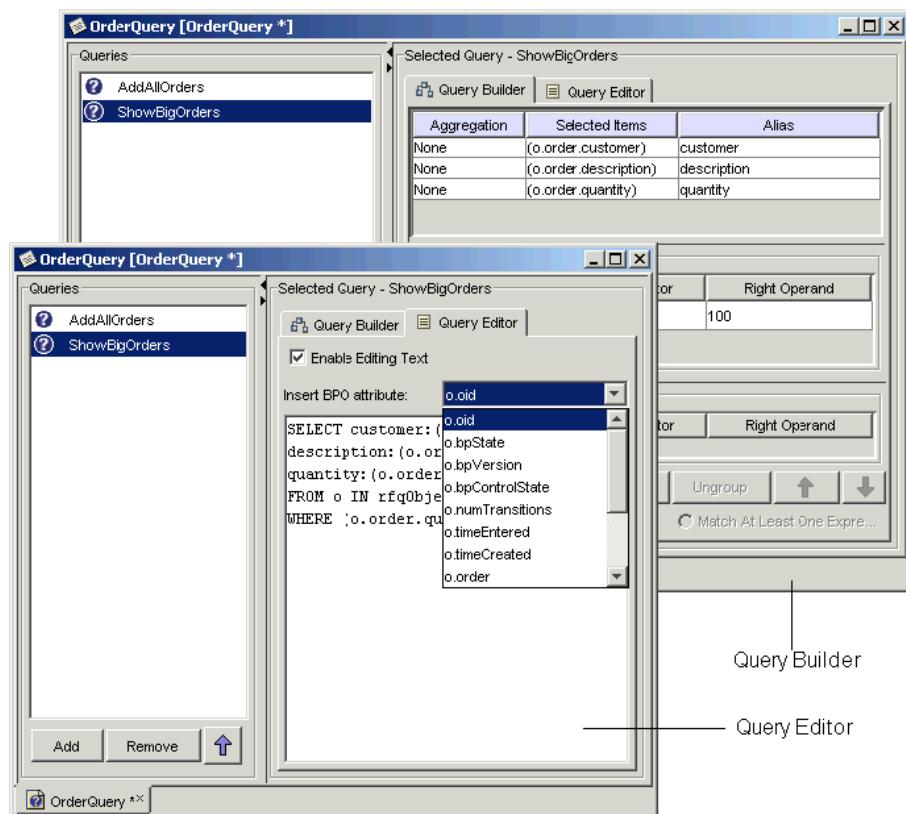


Figure 1-3 Query Builder and Query Editor from Request for Quote Sample

Exception Maps

Exception maps specify an ordered list of exception mappings. Each exception mapping specifies a triggering exception, optional conditional guidelines, and a description of the desired action to handle the exception. The exception mapping also indicates if exception processing is complete with this mapping or should continue down the exception mapping list. For more information, see [Chapter 26, “Exception Handling.”](#)

COMMUNICATION IN MODELS

BusinessWare components can communicate with each other or with external systems using different communication protocols:

- Internet Inter-ORB Protocol (IIOP)
- Web Services Description Language (WSDL)
- Java Remote Method Invocation (RMI)
- Java interfaces
- CORBA Interface Definition Language (IDL)

BusinessWare supports communication in both a synchronous or asynchronous manner. *Synchronous* communication is a more tightly coupled form of messaging, where the caller sends a request to the callee and waits for a response. The caller is blocked until the callee finishes processing the request and returns a response. *Asynchronous* communication is a more decoupled form of messaging where the caller makes data available by publishing it to a channel or a queue. The data is then forwarded to the callee, who subscribes to the channel or queue. The caller does not wait for a response, is not blocked, and can continue processing other data.

For more information on communication styles, see [Chapter 6, “Integration Model Basics.”](#) For information on configuring external systems to interact with BusinessWare, see [Chapter 8, “Application Server Integration.”](#)

CONNECTIVITY WITH EXTERNAL SYSTEMS

Your BusinessWare solutions will need to incorporate data generated by a variety of back-end systems and legacy applications and send data to those entities in a format they understand. Connectors make this possible:

- *Source* connectors receive data from an external source, convert it to BusinessWare events, and forward it to a component in the integration model.
- *Target* connectors receive BusinessWare events from components, convert them into the format required by an external system, and send them to the external target system.

[Figure 1-4](#) illustrates how two target connectors are used in the Request for Quote Sample. A File Target Connector (ERP) receives events output by the Update Inventory component and writes the data to the file system. A Channel Target Connector (Query Output) receives update events from the Order Query component and publishes them to a channel where other applications can retrieve them.

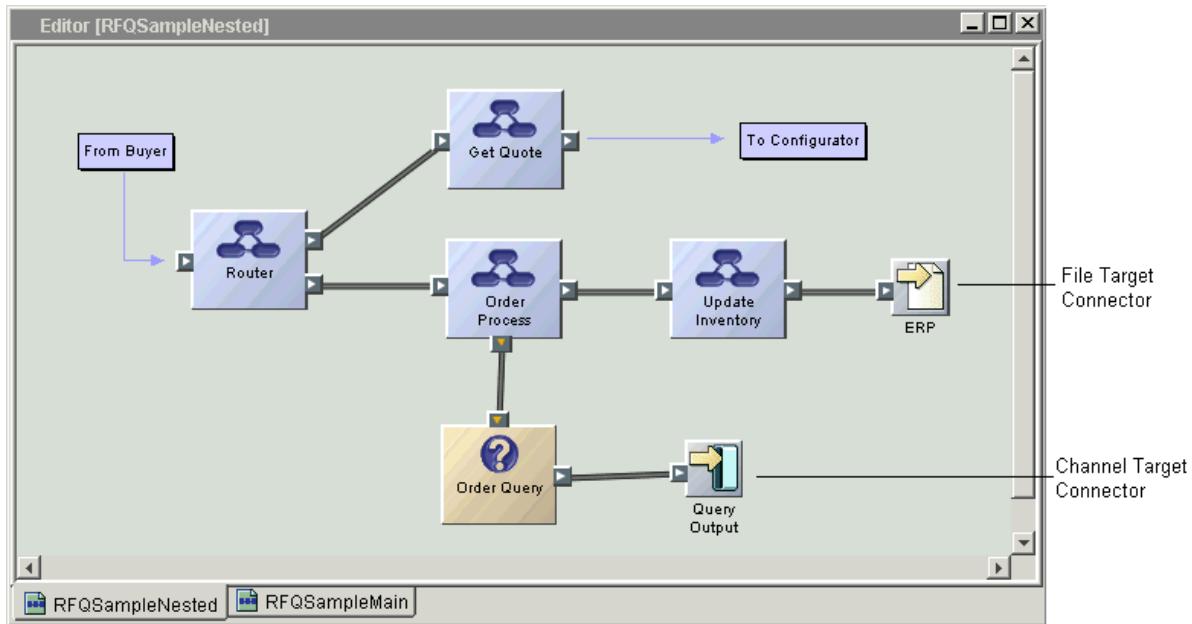


Figure 1-4 Connectors in Request for Quote Sample

BusinessWare connectors are your key to enterprise application integration (EAI). BusinessWare comes with a base set of connectors that provide connectivity to file systems, email systems, Web servers, channels, and queues. Many other connectors designed to meet specific needs can be purchased separately.

Whereas connectors enable interaction between BusinessWare project components and external systems, *proxies* enable interaction with external objects using specific protocols. By including proxies in your model, you have a visual representation of connections to objects outside of the current project and how these objects fit into the complete project.

- *Input* proxies receive data from external objects and forward it to model components.
- *Output* proxies receive data from BusinessWare components and forward it to external objects.

For more information, see [Chapter 6, “Integration Model Basics,”](#) [Chapter 7, “Channels and Queues,”](#) the *File Connector Guide*, the *Email Connector Guide*, and the *HTTP Connector Guide*.

RESOURCES

Your solution will also contain *resources*. In BusinessWare, a resource is a logical representation of a physical entity, such as a channel, database, server, or workflow performer. Resources are used by other modeling objects at design time so that the logical to physical mapping choices may be delayed, shared, and changed easily.

This approach allows you to design without being concerned about implementation details (such as machine names) and to make changes in the production environment (such as moving resources to different machines) without having to make changes in the code.

For more information, see “[Linking Components to Models](#)” on page 6-11.

TYPES

A type is a kind of metadata describing the structure of data. By defining types, you enable BusinessWare to recognize and manipulate the business objects, events, and exceptions that are used in your solution. Many commonly used types are predefined for you and provided in the BusinessWare project and vtAFCommon modules. In addition, you can provide your own type definitions using Java, IDL, WDSL, DTDs, or by structured editing of types provided in the BME.

A BusinessWare type of special interest is the *business process object* (BPO). BPOs are used to represent and store process model data. At design time, each process model can be associated with only one BPO. At runtime, a unique copy of a BPO is instantiated for each instance of that particular process. For example, if you are modeling an order-tracking system and have defined an “order BPO” type, each order is represented by a unique “order BPO.”

For more information on defining interfaces, see [Chapter 6, “Integration Model Basics.”](#) For more information on defining business objects, see [Chapter 11, “Process Models: Defining and Using Business Objects.”](#)

PROJECTS

The BME uses a construct called a *project* to logically contain and manage all the objects associated with a BusinessWare solution.

As you develop your solution, files are generated to store the design of your models, source code, and type definitions. When you compile or deploy your solution, executable files and other types of files are generated. All these files are organized within the project.

Mapping parameters permits values of properties to be established during modeling and to be modified during deployment without affecting the design of the models.

You can share and browse projects in the BME using the *design repository*. The design repository is a mechanism in the BME that enables you to explore project content outside your current project. For more information on the Design Repository, see [Chapter 4, “Design Repository.”](#)

[Figure 1-5](#) shows the folders and objects in the RFQSample project, as they are listed in the Explorer window of the BME.

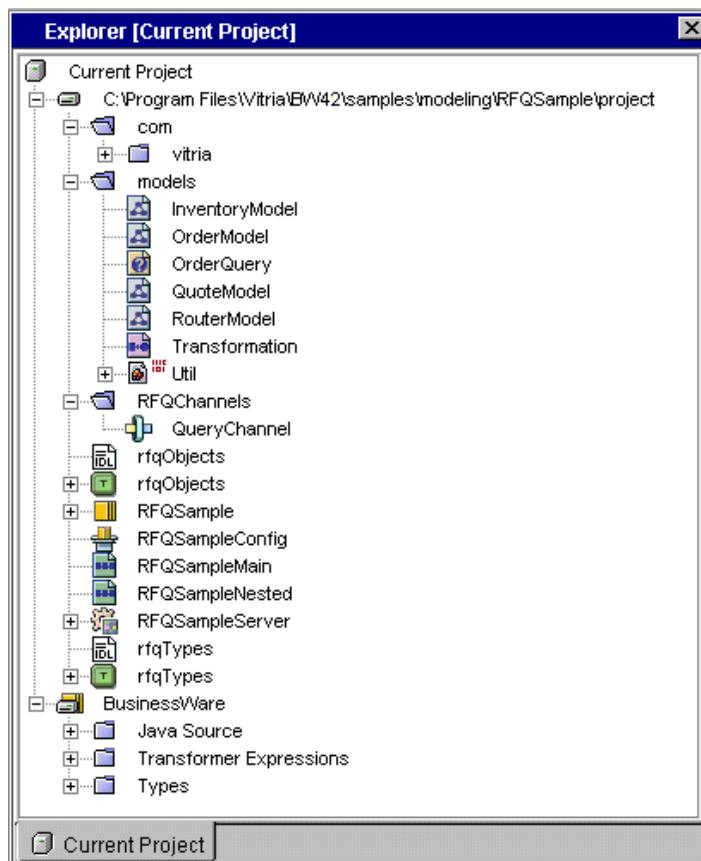


Figure 1-5 Project Contents As Shown in Explorer

The concept of a project thus helps you organize, manage, and share your work. You can easily export and import projects so that others can use them.

For more information, see [Chapter 3, “Projects,”](#) [Chapter 4, “Design Repository,”](#) and [Chapter 23, “Project Packaging.”](#)

PROJECT MODULES

Project modules are a special type of project, containing a “library” of modeling objects that other projects can reference.

BusinessWare Project

The BusinessWare project module is automatically installed in your BME instance and in the directory server when you install BusinessWare. It contains the type definitions for many commonly used interfaces and events. All the projects you create will automatically have a dependency upon the BusinessWare project module, thus giving you access to all the objects it contains. Application connectors are also project modules.

vtAFCommon Project

The vtAFCommon project can be added to your current project as a dependent project if you are developing an application that uses the Application Administration User Interface or that interacts with Vitria Business Collaboration for exchanging messages with trading partners. The vtAFCommon project contains definitions of events that Vitria Business Collaboration uses.

You can install your own project modules (**Tools > Install Project Module**) simply by creating a project that contains the data you want to make reusable and deploying it to a file.

For more information, see “[Using Project Modules](#)” on page 3-16.

DEPLOYING BUSINESSWARE SOLUTIONS

Deployment is the process of moving an application from the BusinessWare Modeling Environment to the runtime environment and ensuring that it will perform as expected. Typically, you first do a test deployment to a single machine and then, after debugging the solution, you deploy to a production environment consisting of one or more machines.

Deployment involves:

- **Partitioning**—specifying the BusinessWare Servers and Integration Servers on which the various components will run. You can partition modeling objects across multiple servers.
- **Deploying**—installing the project data into the directory server. When you start the project, the BusinessWare Servers and Integration Servers retrieve the data they need from the directory server.

You can create multiple deployment configurations for a single project, and you can deploy multiple versions of a project. For more information, see [Chapter 22, “Deploying Projects.”](#)

DEBUGGING BUSINESSWARE SOLUTIONS

BusinessWare provides several powerful tools to help you test and debug your project. Among them are:

- **Debugger**—lets you set breakpoints in models, Java code, target connector input ports, and dynamic connector input ports. When the execution pauses at a breakpoint, you can inspect events, BPO, and Java variables.
- **Event Injector**—lets you create test events and insert them into a running project. It is useful for debugging and for demonstrating projects.
- **Event Inspector**—lets you inspect parameter values when stopped at a breakpoint.
- **BPO Inspector**—lets you inspect business process object (BPO) attribute values when stopped at a breakpoint.
- **Channel/ Queue Inspector**—lets you inspect events as they flow through a channel or queue.
- **Log Viewer**—lets you view the logs sent to text files, binary files, or channels. This diagnostic information about the BusinessWare Server, Integration Servers, or projects can be used to monitor system behavior.

For more information, see [Chapter 27, “Debugging and Animation.”](#)

ANALYZING BUSINESSWARE SOLUTIONS

A process query model gathers information about the performance of a monitored process model as the project runs. You can use this data to alert personnel or an automated recovery system to take corrective action when performance falls below the agreed-upon quality of service.

In addition, process views allow you to visualize the data generated from the process queries using the Business Cockpit, a Web-based tool that displays business data using graphical charts and tables. You may visualize the data as snapshot reports or as continuous real-time updates.

For information, see [Chapter 17, “Process Query Models,” Chapter 19, “Process Views,”](#) and the *Business Cockpit Guide*.

LIFECYCLE OF A BUSINESSWARE SOLUTION

Typically, the lifecycle of a BusinessWare solution involves all the following stages:

- Modeling
- Deploying to a test environment
- Testing and debugging
- Refining the model
- Deploying to a production environment
- Running and monitoring as well as analyzing the performance in production
- Refining the model if performance data or changing business conditions dictate the need

[Figure 1-6](#) shows the progression through these stages.

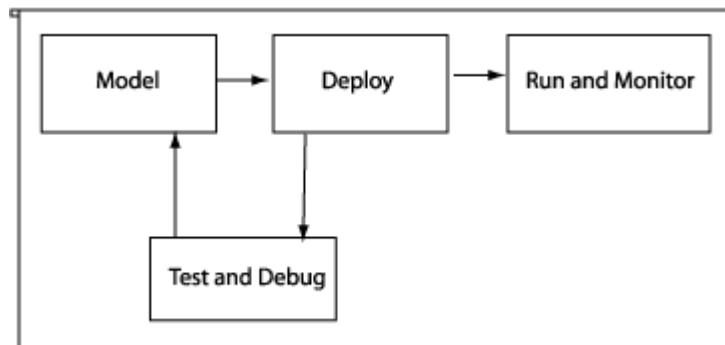


Figure 1-6 Lifecycle of a BusinessWare Solution

CHECKLIST FOR CREATING BUSINESSWARE SOLUTIONS

Table 1-1 lists the major tasks involved in designing, deploying, and debugging a BusinessWare solution. Although these tasks are listed sequentially, the process is iterative, and you have considerable flexibility in how you approach it. This table is intended merely as a general guide and checklist; not all projects require performing all tasks.

Table 1-1 includes all possible tasks. Your solution may not require all of them.

Note: In addition to the references listed in **Table 1-1**, the *BME Help* provides step-by-step instructions for each of these tasks.

Table 1-1 Developing and Deploying a BusinessWare Solution

Task	Description
	For More Information
Launch the BME.	<ul style="list-style-type: none">Windows: Select Start > Programs > Vitria BusinessWare Version > BusinessWare Modeling Environment.Unix: Run <i>installdir/bin/bme.sh</i>. <p>See Chapter 2, “Using the Modeling Environment.”</p>
Create a project.	<p>A BusinessWare project serves as a logical container for all the objects and files associated with a single BusinessWare solution.</p> <p>Select Project > Project Manager</p> <p>See Chapter 3, “Projects.”</p>

GETTING STARTED

Checklist for Creating BusinessWare Solutions

Table 1-1 Developing and Deploying a BusinessWare Solution (Continued)

Task	Description
	For More Information
Define the types that will be used in your project.	<p>BusinessWare types define the structure of data and interfaces used in a project. Many commonly used types are predefined for you in the BusinessWare project module. But you have to define the objects and events that are specific to your project.</p> <p>Several options are available to create types:</p> <ul style="list-style-type: none">• Use the tools in the BME to create a Java Interface, Class, Exception, or Defined Type using the menu items under the File > New... > Types.• Import WSDL using the wizard on the Web service proxy Chapter 9, “Web Services.”• Create simple types using structured editing in the Explorer window: New > Types > DefinedType• Select a Java interface or exception class, and select Tools > Use as Type (or right-click on the file to show the Use as Type menu).• Use an existing Java file and convert it to type by copying it into the project, selecting it in the Explorer, and right-clicking on Tools > Use as Type menu.• Create a new IDL file by using the menu items under the File > New... > Other.• Select an IDL file containing your type specifications and select Tools > Convert to Type... (or right-click on the file to show the Convert to Type... menu). <p>See Chapter 6, “Integration Model Basics” and Chapter 11, “Process Models: Defining and Using Business Objects.”</p>
Create the root integration model: <ul style="list-style-type: none">• Add components.• Add connectors and proxies.• Configure ports.• Wire components together.	<p>The root integration model is the top-level model. It gives an end-to-end view of the solution. All other models ultimately can be traced back to the root model.</p> <p>An empty root integration model is created when you create a new project.</p> <p>Note: Setting key properties as project parameters, such as the database used at runtime, enables seamless transition from the testing environment to the production environment. See Chapter 23, “Project Packaging.”</p> <p>See Chapter 6, “Integration Model Basics.”</p>
Create project resources representing: <ul style="list-style-type: none">• Channels• Queues• Databases• Integration Servers• Workflow performers	<p>A resource is a logical representation of an entity—such as a database, a channel, a workflow performer, or a server—that is mapped to the physical entity during deployment.</p> <p>Right-click the project directory and select New > Resources.</p> <p>See Chapter 3, “Projects,” Chapter 7, “Channels and Queues,” Chapter 6, “Integration Model Basics,” and Chapter 16, “Workflow.”</p>

Table 1-1 Developing and Deploying a BusinessWare Solution (Continued)

Task	Description
	For More Information
Link connectors to appropriate resources.	Select the connector in the integration model, and configure the property values. See Chapter 6, “Integration Model Basics.”
Create process models: <ul style="list-style-type: none">• Add states.• Add transitions from state to state.• Define the triggering events, conditions, and actions for transitions.• Add action code to states.	Process models are statechart diagrams that define a business process. They consist of a sequence of states, each representing a distinct phase in the process, and the transitions between the states. Double-click on a process component. Or select File > New... > Models > Process . See Chapter 10, “Process Models: Basic Concepts,” Chapter 12, “Process Models: Ports, States, and Transitions,” and Chapter 14, “Process Model Code Construction.”
Link process components to process models.	Select the process component in the integration model and set its Process Model Link property. Note: Linking <i>before</i> adding action code to your process model is a good idea since it may result in additional methods being available to you when you generate the code in the Action Builder. See “Linking Components to Models” on page 6-11 and “Port Synchronization” on page 10-7 .
Create transformer models.	A transformer model maps events of one type to events of another type. It is used to convert incoming data to the format required by the next component in the system. See Chapter 15, “Process Model Templates” and the <i>BusinessWare Transformer Guide</i> .
Link transformer states in the process model to transformer models.	Select the transformer state in the process model and set its Nesting Specification property. Double-click to link to a transformation model. See “Linking Components to Models” on page 6-11 .
Optionally, define workflow systems to incorporate human activities in the process. <ul style="list-style-type: none">• Create a workflow process model.• Set up the Task Manager project.• Set up roles.• Set up the Web interface for users.	Workflow systems provide for human interaction with automated systems. When invoked by an activity state in the workflow model, the Task Manager creates and assigns tasks. Users can access their list of assigned tasks via a Web interface. See Chapter 16, “Workflow” and the <i>Application Development Guide</i> .

GETTING STARTED**Checklist for Creating BusinessWare Solutions****Table 1-1 Developing and Deploying a BusinessWare Solution (Continued)**

Task	Description
	For More Information
Optionally, create process query models to monitor BPO processing.	A process query model is a set of queries that are run against the BPOs being processed by a process model. The query results are a means of measuring the quality of service. • Create the queries. • Specify the type of BPO to be monitored. • Specify the module name for the update events used by the model.
Link process query components to process query models for real-time queries.	Select the process query component in the model and set its Process Query Model Link property.
Create process views to allow visualization of process queries through the Business Cockpit.	See “ Linking Components to Models ” on page 6-11, Chapter 19, “Process Views,” and the <i>Business Cockpit Guide</i> .
Compile or build the project.	Compiling generates new machine-executable code for only those objects that have been modified since the last build or compile; building generates new code for all objects. See “ Building and Compiling Projects ” on page 3-24.
Deploy the project in a test environment.	For test purposes, all the components should be configured to run in one Integration Server. Enable Debugging and Animation property must be set to True. See Chapter 22, “ Deploying Projects .”
Run the debugger.	You can insert breakpoints, inject events, and examine BPOs, event data, local variables, etc. when the execution pauses at a breakpoint. See Chapter 27, “ Debugging and Animation .”
Resolve any problems, re-deploy if necessary, and run the debugger again.	Unit testing via the debugger is an important step. However, it is necessary that you also perform system and stress tests in a testing environment before deploying in a production environment. See Chapter 27, “ Debugging and Animation .”
Evaluate the reliability and scalability requirements of the production environment before deploying.	Defining clusters of Integration Servers can improve overall performance characteristics of the solution, but may impose restrictions on the deployment configuration. See Chapter 21, “ Load Balancing .”
Deploy the project in a production environment.	In your deployment configuration, you can assign components to multiple servers running on multiple machines for load-balancing and performance. See Chapter 22, “ Deploying Projects .”

Table 1-1 Developing and Deploying a BusinessWare Solution (Continued)

Task	Description
	For More Information
Use data gathered by process query models to analyze the quality of service being provided. Use process views where applicable to visualize the process query output.	The process query model generates update events reflecting the changes made to BPOs. You can use these results as input to a downstream process model or custom application that takes corrective action when performance falls below the agreed-upon quality of service. In addition to automated corrective action, monitor your business visually using process views with the Business Cockpit. See Chapter 17, “Process Query Models,” Chapter 19, “Process Views,” and the <i>Business Cockpit Guide</i> .
As necessary, update the project by making design changes and redeploying.	You can deploy multiple versions of a project. See Chapter 22, “Deploying Projects.”

SAMPLES AND TUTORIALS

Several sample applications are included with BusinessWare. You can load, build, and deploy these projects to gain a familiarity with the product. In addition, a tutorial guides you step by step through the process of creating and running a sample application.

For descriptions of each of the samples, see `installdir\samples\SamplesOverview.htm`, where `installdir` is your BusinessWare installation directory.

GETTING STARTED
Samples and Tutorials

2

USING THE MODELING ENVIRONMENT

The BME is an integrated development environment through which you can model, deploy, and debug business systems. This chapter explains launching the BME and provides an overview of it.

Topics include:

- Starting the BME
- Getting Help in the BME
- Introduction to the BME Windows
- Quick Reference for Shortcuts in the BME

STARTING THE BME

To start the BME on Windows, choose **Start > Programs > Vitria BusinessWare version > BusinessWare Modeling Environment**, where *version* is the BusinessWare version number.

To start the BME on Solaris, first run the following script to ensure the proper environment setting (you need only run this once per specific shell window):

```
% source installldir/bin/sparc_solaris/vtconfig.csh
```

then run:

```
% installldir/bin/sparc_solaris/bme.sh
```

where *installldir* is the BusinessWare installation directory.

Note: The BME is only supported on Solaris and Windows.

GETTING HELP IN THE BME

The BME features a windows-sensitive Help system called *BME Help* that provides information on items in the BME. To access *BME Help*, select a window or an object in the BME and press F1.

USING THE MODELING ENVIRONMENT

Introduction to the BME Windows

The BME also provides tool tips; rest your mouse cursor over an item in the BME to view a brief description of that item.

INTRODUCTION TO THE BME WINDOWS

The BME is comprised of multiple windows, with each window providing unique functionality. Figure 2-1 displays a view of the BME after initial activation, and calls out five BME windows—*Main*, *Explorer*, *Properties*, *Editor*, and *Output*—which are explained in this section.

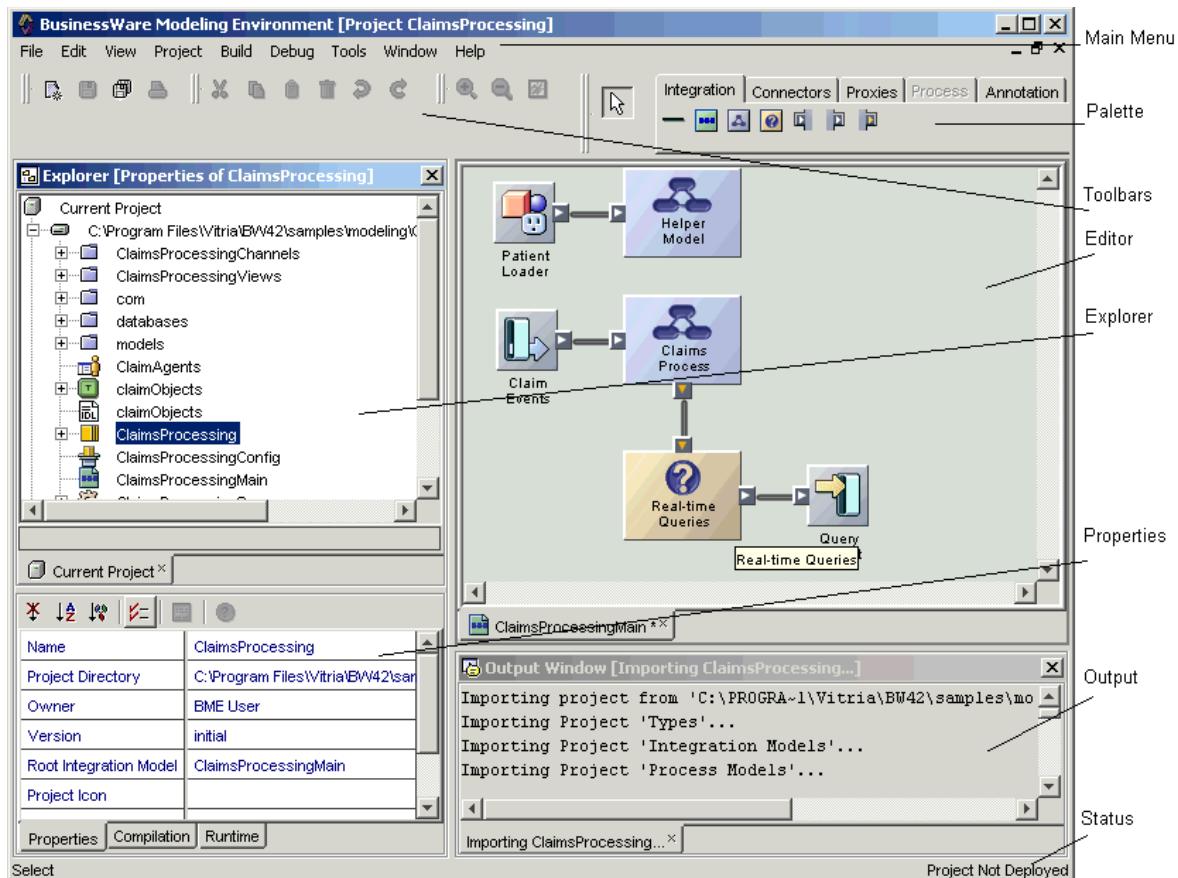


Figure 2-1 Key BusinessWare Modeling Environment Windows

Note: Figure 2-1 shows the view of the BME with a project open. You may size and arrange the BME windows during usage. The BME preserves the window layout for each project upon shutdown and recreates the layout upon restart.

Click a particular window to activate it.

Right-clicking on any object in the BME (including the windows themselves) activates a shortcut menu. Shortcut menus display commands pertinent to the selected object. For example, right-clicking on a tab in the Editor displays commands for saving, closing, etc., the particular model or source code you are editing.

Tip: Shortcut menus are particularly useful for quick command execution. Although both the Main window and the shortcut menus provide access to commands that you can execute against BME objects, shortcut menus only list those commands pertinent to the selected object.

MAIN WINDOW

The Main window includes the following elements, illustrated in [Figure 2-1](#):

- **Main menu**—contains a series of drop-down menus for you to activate commands in the BME.
Note: To view keyboard shortcuts for the menus, press **Alt**.
- **Toolbars**—collections of graphical buttons with which you can activate common commands at the click of a mouse.
- **Palette**—Uses tabs to group related model objects. In the Palette, you can click and drop graphical model objects (for both integration models and process models) into your models in the Editors. Tabs in the palette are enabled based on the currently active editor.
- **Additional windows**—includes the Explorer, Editor, Properties, and Output windows, as described in this chapter.
Note: You must first activate the Editor before clicking and dropping objects into a model.
- **Status**—indicates the current status of the project, including started, stopped, and partially running.

[Table 2-1](#) briefly describes each item in the menu bar, toolbars, and Palette. For detailed information on any item in the Main window, select the item and press **F1** to access *BME Help*.

USING THE MODELING ENVIRONMENT
Introduction to the BME Windows

Tip: To show or hide specific toolbars and the Palette, right-click in the toolbar area and choose which toolbars to show/hide.

Table 2-1 Main Menu, Toolbar, and Palette Items

Item	Brief Description
Main Menu	
File	Commands for creating new items (such as integration models, process models, resources and routers), mounting and un-mounting filesystems, saving selected items, printing, and exiting the BME
Edit	Commands for editing selected items, such as objects in models or source code
View	Commands for finding or viewing information through various BME windows
Project	Commands for creating, managing, and starting and stopping projects
Build	Commands for compiling or building a project
Debug	Commands for debugging the models in your project
Tools	Tools for configuring your BME settings, exporting Web services, and importing XML Metadata Interchange format (XMI) files
Window	Commands for managing windows or selecting specific windows
Help	Provides access to the <i>BME Help</i>
Toolbars	
Build	Buttons for building a project
Data	Buttons for finding information, stepping through window sheets, exploring a filesystem, and viewing properties of a selected item
Debug	Buttons for debugging the models in your project
Edit	Buttons for editing selected items, such as models or source code
Graphics	Buttons for zooming in and out of process models and integration models
System	Buttons for creating new items (such as integration models, process models, resources, and routers), and saving selected items
View	Buttons for activating the Output window, Debugger window, and Execution View
Palette	
Integration	Components that reside in integration models and ports and wires used to connect components together. The integration model components include nested integration component, process component, and process query component. You can click and drop components into integration models in an Editor.
Connectors	Connectors—such as file source and target, channel source and target, Hypertext Transfer Protocol (HTTP) source and target—that you can click and drop into an integration model

Table 2-1 Main Menu, Toolbar, and Palette Items (Continued)

Item	Brief Description
Proxies	Proxies—such as simple input proxy, simple output proxy, EJB output proxy, Web Service input proxy, and Web Service output proxy—that you can click and drop into an integration model
Process	States, transitions, and ports that you can click and drop into process models
Annotation	Annotation elements—such as labels and arrows—that you can click and drop into integration and process models in an Editor to add more information to models

EXPLORER

As you develop a business solution in the BME, you can create one or more projects using objects such as integration models, process models, connectors and proxies. The Explorer displays all projects, objects, and dependencies on other projects and libraries.

During deployment, BusinessWare converts your model designs into executable objects, so that you can run your business system.

The Explorer, shown in [Figure 2-2](#), displays these filesystem objects hierarchically according to the location in which objects are created.

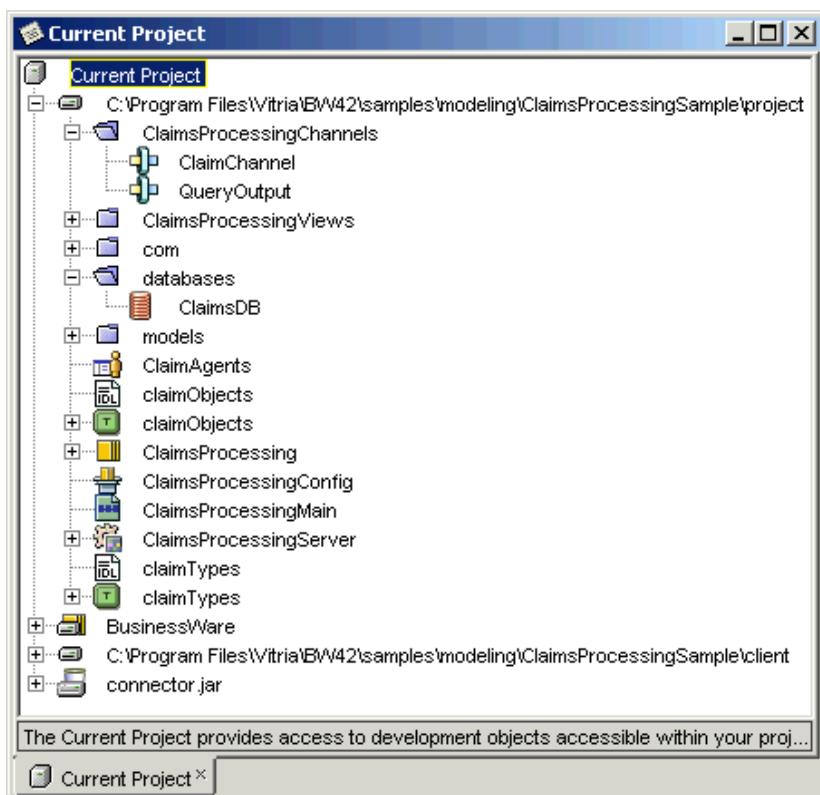


Figure 2-2 BME Explorer

Double-clicking on an item in the Explorer invokes that object's default behavior. For example, double-clicking on a model object in the Explorer launches an Editor that graphically displays that particular model. For more information on the Editor, see "[Editor](#)" on page 2-9.

To view the properties of an item in the Explorer, select the item and the content of the Properties window will update to display the item's property values.

PROPERTIES WINDOW

There are two types of properties windows, both of which display properties for BME objects:

- **The “shared” Properties window**—the “shared” Properties window, [Figure 2-3](#), remains open while you work in the BME. As you select different objects, the shared Properties window changes its contents to display the properties for the currently-selected object. [Figure 2-3](#) shows an example of the “shared” Properties window opened to display the properties of a selected process component.

Using the shared Properties window conserves screen real estate because it uses one window to show each object’s properties as you select different objects.

- **The “anchored” Properties window**—right-click any object in the Explorer or the Editor, and select **Properties** from the shortcut menu to bring up an “anchored” Properties window. In this anchored window, you can view and edit properties for the selected object. This Properties window is “anchored” to the object selected at the time the window is created, and will remain open and not change when you view properties of other objects.

These two types of properties windows show identical information; they are available for you to choose which Properties window you prefer when working in the BME. The “shared” Properties window lets you view properties for multiple objects using a single window. The “anchored” Properties window is useful for comparing properties for multiple objects at the same time.

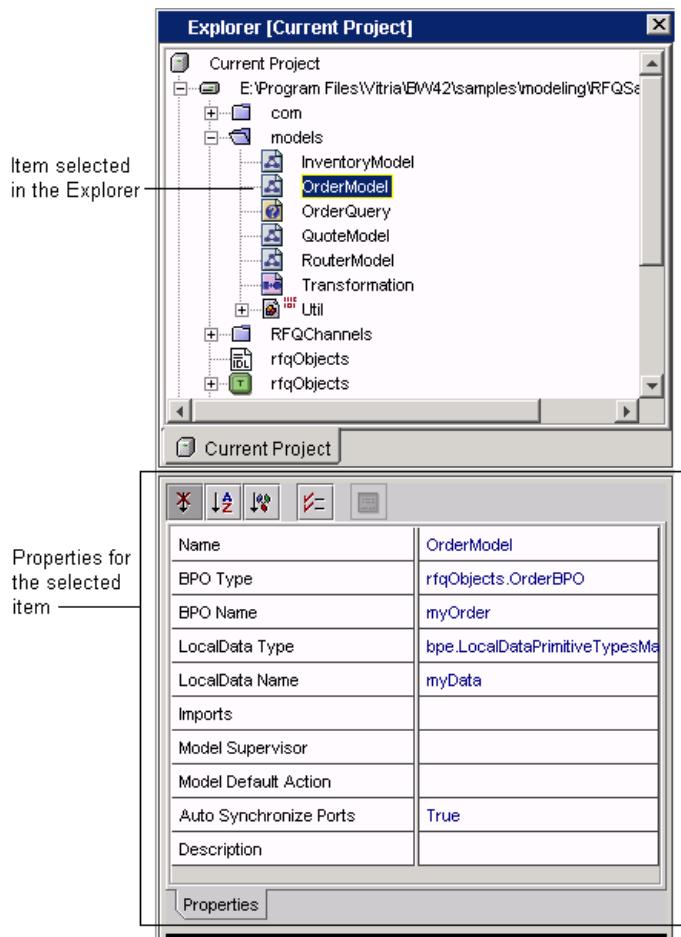


Figure 2-3 Properties Window

The Properties window groups related properties into tabbed sheets. Depending on what item you select in the BME, different property sheets are displayed in the Properties window. For detailed information on a particular property sheet associated with a particular object, access *BME Help*.

Tip: The Properties window features buttons, located just below its title bar; use these to sort properties by various criteria and to hide non-editable properties.

EDITOR

The first step in developing a business solution with BusinessWare is to create a new project in the BME. A *project* is a way to group all the objects that make up your business solution system. For more information about projects, see [Chapter 3, “Projects.”](#)

When you create a new project, the BME automatically creates an empty integration model and displays the model in the Editor. You then build your integration model by clicking and dropping objects from the Integration tab on the Palette into the model in the Editor.

[Figure 2-4](#) illustrates this procedure for an integration model named RFQSampleNested.

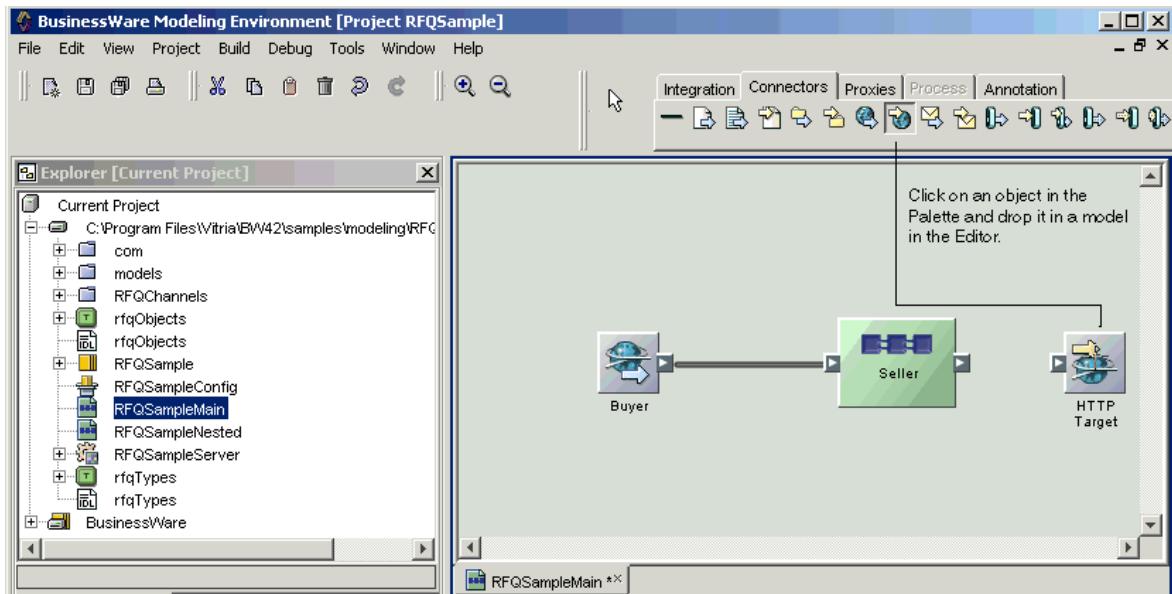


Figure 2-4 Creating an Integration Model in the Editor

USING THE MODELING ENVIRONMENT

Introduction to the BME Windows

After creating the top-level elements of the integration model, double-clicking on a linked object opens the Editor to the linked content. For example, double-clicking on the Update Inventory process component in the RFQ Sample opens the Inventory Model as shown in [Figure 2-5](#). If the component is not linked, a window opens and you can create a link to an existing object or create a new object. If you select to create a new object, the New Wizard opens.

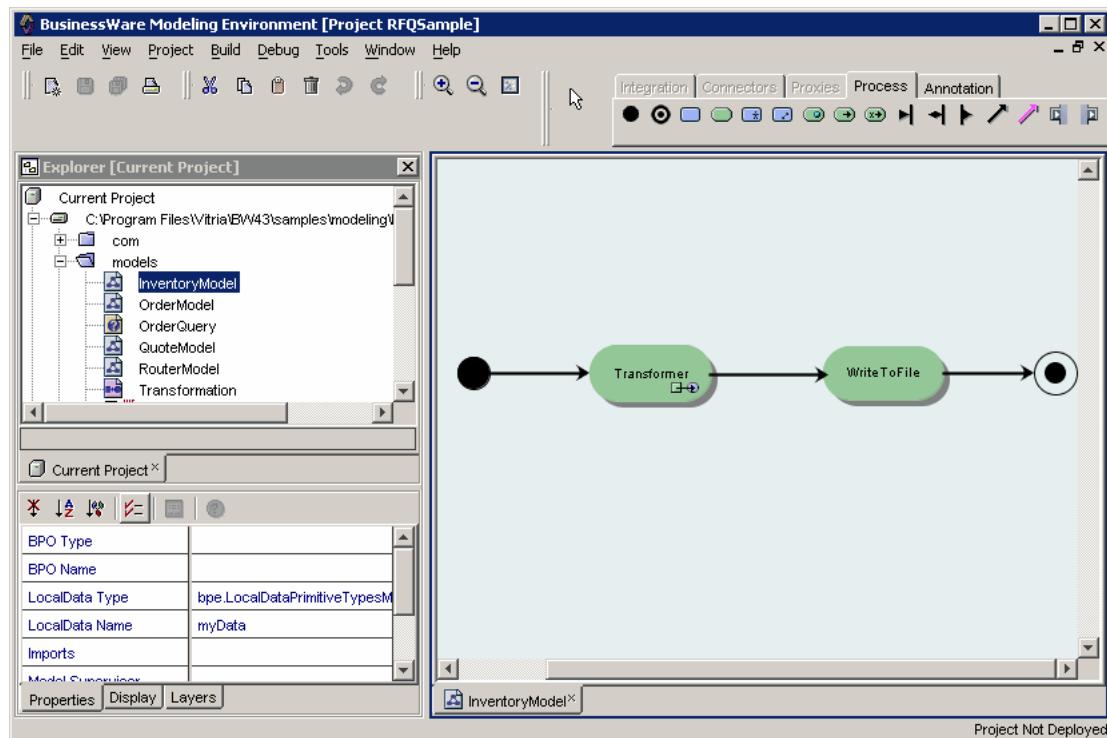


Figure 2-5 Creating a Process Model in the Editor

As you create process models, BusinessWare generates the Java source code necessary to execute (run) the model logic. The BME displays the Java source code, for you to view and edit, in the Source Code Editor, which is another Editor view. To activate the Source Code Editor, right-click on the process model in the Explorer and select **Edit** from the shortcut menu.

Right-clicking on a tab in the Editor brings up a shortcut menu that contains:

- **Save**—saves changes to the process model only.
- **Clone View**—creates a new window for display of the selected view. The originally selected view also remains in the Editor when using the “Clone View” command.

- **Close**—closes the model or source code displayed in that tab.
The close box on the Editor closes *all* models and source code files viewed in that Editor. If you want to close just one model or file, right-click on its tab, and select **Close**.
- **Dock View Into**—removes the selected view from the Editor, and inserts the view into another window.

You may find it helpful to dock or clone a process model Java source code view into a separate window so that you can view the graphical display of a process model concurrently with its corresponding Java source code.

Because you can clone an Editor, and dock an Editor in other frames, multiple Editors can be viewed simultaneously.

OUTPUT WINDOW

The Output window displays error and status messages generated by the BME. The BME creates tabs in the Output window for such actions as compiling, validating, importing, and debugging.

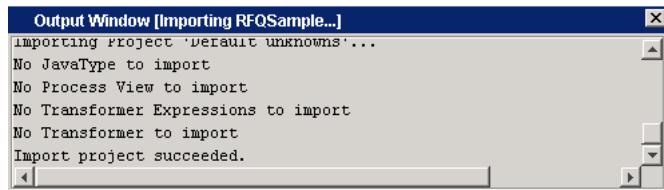


Figure 2-6 Output Window

Tip: The following tips are useful when using the Output window:

- Clicking on highlighted error messages in the Output window automatically moves your mouse cursor to the area of error in the BME.
- Right-clicking in the Output window opens a shortcut menu with Select All and Copy commands. Use these commands to copy contents from the Output window onto your system's clipboard.

QUICK REFERENCE FOR SHORTCUTS IN THE BME

You can double-click on many components in the BME to perform the most common operation on that component. [Table 2-2](#) shows these shortcuts categorized by location in the BME.

Table 2-2 Shortcuts in the BME

Component/Location	Shortcut
Integration Model	<ul style="list-style-type: none"> Double-click on a web service proxy to open the web service wizard. Double-click on a line source or file source connector to set the source file or directory respectively. Double-click on a file target connector to set the output directory. Double-click on a channel/queue source/target to set the channel resource. Double-click on a port to set the port type. To quickly create a transformation, double-click on a process component and use the SimpleTransformer template to quickly link and create a process model with a transformation state that is linked to a BusinessWare Transformer. The Transformer opens so you can immediately begin mapping parameters.
Process Model Double-Click Accelerators for Editing Actions and Code	<ul style="list-style-type: none"> Double-click on a transition to open a single dialog for setting the trigger type, condition and action. Double-click on an action state to edit the action with action builder. Double-click on a simple state to edit the entry action with action builder. Use Ctrl+Double-click to edit the exit action with action builder. Shift-double click on a transition, action, or state to edit the action code with the Source Code Editor instead of the action or condition builder dialogs. Press Ctrl when creating a transition to automatically assign the default Trigger Type or the Trigger Type most recently assigned. You can reverse the double-click/Shift+double-click behavior by setting Tools > Options > BusinessWare Options > Process Modeler Options > Double Click to Code to True.
Process Model Other Accelerators	<ul style="list-style-type: none"> Double-click on an activity state to open the action builder. Double-click on a nested state to open the Nested Model Selector. Double-click on a web service state to bring up the web service wizard.
Integration Model Tab of the Palette and Process Model Tab of the Palette	<ul style="list-style-type: none"> Double-click on a button in the Palette to make that tool/operation “sticky.” That is, when you double-click on the wire tool, you can create multiple wires. Similarly, double click on a process component or action state, you can create those objects multiple times. Click on the selection tool, or press the 'Esc' key to exit “sticky” mode.
Explorer	<ul style="list-style-type: none"> Copy and paste a Java file from one directory to another and the BME will automatically change the package declaration for you.

PART II: PROJECTS

This part describes how project versioning and project modules are used in addition to how projects are used to organize and manage data.

Chapters include:

- [Projects](#)
- [Design Repository](#)

PART #: TITLE HERE GOES HERE

3

PROJECTS

The BME uses a construct called a *project* to logically contain and manage all the objects associated with a single BusinessWare solution. This chapter describes how you can use a *project* to manage a BusinessWare business application and its constituent objects as a unit.

Topics include:

- [Introduction to Projects](#)
- [Working with Projects](#)
- [Using the Project Manager](#)
- [Objects and Files in the BME](#)
- [Upgrading Objects](#)
- [Versioning Projects](#)
- [Building and Compiling Projects](#)

To get detailed help on any of the tasks discussed within this chapter, access the *BME Help*.

INTRODUCTION TO PROJECTS

A project is defined by the following characteristics: *content*, *mountpoints*, *parameters*, and *settings*. This section describes these characteristics.

PROJECT CONTENT

As you develop a BusinessWare solution using the BME, BusinessWare persists data to a number of files:

- At modeling time, the BME creates files to store the design of your models, source code, and type definitions in response to your edits.
- At build-time and deployment-time, the BME uses the design-time files to create executable files used by the runtime environment.

All files associated with a single business system define the *content* of a project.

Tip: To help you locate objects in other projects for reuse, you can browse and share with others using the design repository. For information on the design repository, see [Chapter 4, “Design Repository.”](#)

Project Objects

Project objects are hierarchically arranged in folders, which are equivalent to those on a filesystem. The BME displays the following project objects in the Explorer:

- **Models**—
 - Integration models that logically and graphically define internal and external interactions in the BusinessWare environment.
 - Process models that logically and graphically define a business process or some subcomponent of a business process.
 - Process query models that monitor and report data being processed by process models.
 - Transformer models that contain specifications of how to transform one type of event into another.
 - ExceptionMaps represent a reusable mapping from a specified exception (or set of exceptions) to a distinct corrective action (or actions).
- **Types**—definitions of the format of the data managed and used by a BusinessWare solution for communication between BusinessWare project objects, as well as between BusinessWare project objects and an external entity.
- **Resources**—BME representations of a logical modeling entity—such as a database, a channel, a queue, a workflow performer, or a server—which are mapped to physical entities.
- **Java Source**—user-defined java code used within a BusinessWare solution.
- **Deployment Configurations**—specifications that define the runtime partitioning information that describes how the BusinessWare solution is distributed across BusinessWare and Integration Servers.
- **Project Object**—specifies the project options that control behavior during modeling, compiling, and running of the project. In addition, it specifies project parameterization details. For more information, see [Chapter 23, “Project Packaging.”](#)
- Any other file in a project. This could be a text file that provides documentation about the project for the developer or an XML file that a process model reads at runtime.

PROJECT MOUNTPOINTS

A project's content may span multiple locations on one or more filesystems accessible from a developer's workstation. Each location is referred to as a *filesystem mountpoint* or more commonly as a *mountpoint*.

A mountpoint may represent:

- Directory on a filesystem
- JAR file on a filesystem
- Project module

For more information about project modules, see “[Using Project Modules](#)” on [page 3-16](#).

All projects have at least one mountpoint describing the project directory. The project directory contains the project object which specifies the project options that control behavior during modeling, compiling, and running of the project. When you create a new project, it includes a default integration model located in the project directory. Large projects with content spanning multiple directories may have multiple mountpoints. It is suggested that you choose a directory as your project location that contains only files you want in your project.

To use a class in a JAR file mounted to the current project, you can use `Thread.currentThread().getContextClassLoader()` to call this class, since this can load classes located in the Directory Server, and all the classes related to the current project, including dependent projects.

To make the organization of large projects easier, you can distribute content across multiple mountpoints. For example, you may want to organize content across mountpoints based on the following categories:

- Object type
- Ownership
- Functional units

Typically, the order of the mountpoints in the project is not important; however, mountpoints are used to define the search path used to resolve object references. Objects with the same name are resolved according to the first instance located in the mountpoint search path.

The BME displays the project mountpoints in the Explorer. You may expand or collapse each mountpoint to expose or hide its content.

For more information on creating project mountpoints and mounting filesystems, see the *BME Help*.

LOCAL SERVICES

The Local Services object appears under the Project object in the Explorer and groups all application services of a BusinessWare project.

Local Services consists of the following services:

- **Registry Service**—provides a repository for storing information about trading partners and installed solutions.
- **Document Store Service**—provides persistent storage and versioning of documents at various stages of processing.
- **Logging Service**—provides persistent storage of audit trail data for logging transport, protocol, and application specific information.
- **Security Service**—provides persistent storage of private keys used for signing, encrypting, or decrypting messages and documents.

For more information about the application services, see [Chapter 28, “Application Services.”](#)

REMOTE SERVICES

Remote Services are a collection of properties used to configure a project as a service provider. These properties are also used to customize components that are used to receive responses to service requests made by the current project to other service provider projects. For more information about the remote services, see [Chapter 28, “Application Services.”](#)

PROJECT PROPERTIES

Project properties are a collection of properties that influence a project’s behavior. To access the project properties, select the project object in the Explorer. Project properties are divided into three categories:

- General Properties—control aspects of the project specific to modeling-time activities.
- Compilation Properties—control aspects of the project specific to compiling and deployment.
- Runtime Properties—control aspects of the project specific to runtime activities.

[Figure 3-1](#) shows the Project object selected in the Explorer and the project properties.

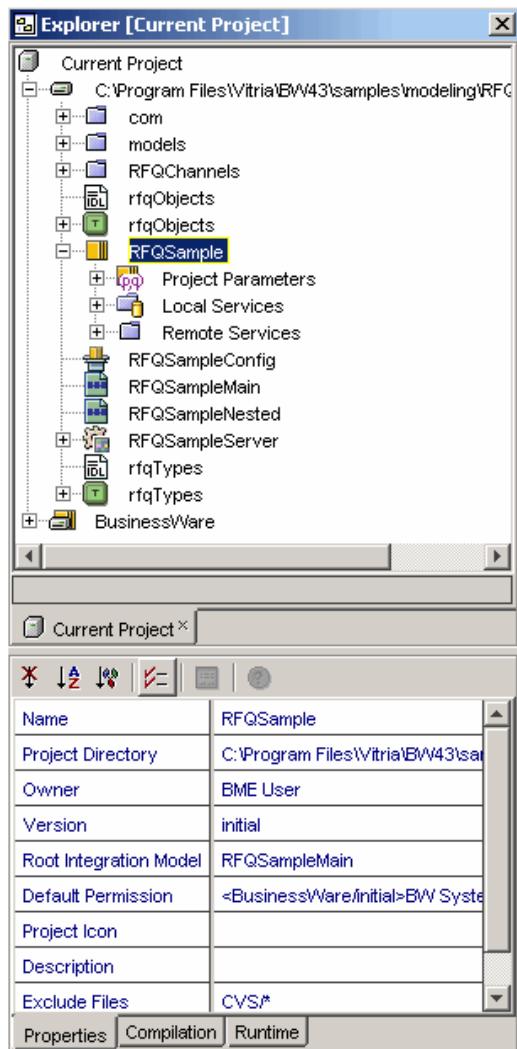


Figure 3-1 Project Object and Project Properties

General Properties

[Table 3-1](#) lists the project general properties.

Table 3-1 General Properties

Property	Description	Default
Name	The project name.	Value is initialized during project creation.
Project Directory	The fully qualified path for the project directory.	Value is initialized during project creation.
Owner	The name of the individual responsible for the project. Generally, this is the original project author.	< <i>user_name</i> > for the project creator
Version	The tag associated with a project for deployment purposes.	initial
Root Integration Model	The highest-level integration model in a BusinessWare solution. The root integration model shows a systems-level view of a project, the functional components, and how the components communicate with each other and external entities.	< <i>project_name</i> >Main
Default Permission	Specifies the value to use for other Permission related attributes in the Project when left unspecified.	<BusinessWare/initial>BW System Administrator, <BusinessWare/initial>BW User
Project Icon	The path to a specific icon (.gif) image inside a project. This icon is used to identify a project after it has been delivered to the design repository when browsing projects in the design repository. It enables users to identify specific projects. If no icon path is specified, the standard project icon will be displayed.	The default project icon is used.
Description	The project description for documentation purposes.	

Table 3-1 General Properties (Continued)

Property	Description	Default
Exclude Files	A list of files that will be excluded when a project is exported or deployed. A regular expression can be used to specify a list of files.	<code>^\.svn/ /\.svn/ ^cvs/ /cvs/ ^GeneratedTypes/</code>
Mountpoints	The project mountpoints and their mappings when a project is exported and subsequently imported. The import mountpoint names can accept environment variables such as \$VITRIA. For more information, see “ Importing Projects ” on page 3-15.	

Compilation Properties

[Table 3-2](#) lists the Java Compilation settings.

Table 3-2 Compilation Properties

Property	Description	Default
Deprecation	Compiler displays deprecation errors.	False
Optimize	Generate optimized Java byte codes.	False
Debug	Embed debugging information in generated byte codes.	True
Encoding	The character encoding the compiler should use when processing source files.	
Max javac Memory	Specifies the maximum memory available for a spawned javac process.	
Source 1.4	Expect Java sources compatible with JDK 1.4.	True

Runtime Properties

[Table 3-3](#) lists all of the runtime properties.

Table 3-3 Runtime Properties

Property	Description	Default
Exception Handler	Fully qualified java class name of the ExceptionHandler implementation class that will be used for components and connectors that do not explicitly define this property. For more information, see Chapter 26, “Exception Handling.”	com.vitria.container.ExceptionHandlerImpl
Request Map Class	Fully qualified java class name of the RequestMap implementation class that will be used by the runtime to help determine how to dispatch a particular request. This class will be used for component input ports for which no explicit class is defined. For more information, see Chapter 13, “Process Modeling Techniques.”	com.vitria.bpe.runtime.ProcessRequestMapImpl
Project Init Class	Fully qualified java class name of the ProjectInit implementation class that implements the init() and cleanup() methods for application specific initialization and cleanup.	None
Project Init Data	Specifies a string to be passed to the ProjectInit.init() method for application-specific initialization.	None
Statistics Log Interval	Time interval (in seconds) at which statistics are generated for each component in the server. Set to 0 to disable statistics logging.	0
Workflow Audit Enabled	Enable or disable workflow auditing.	False
Workflow Audit Channel	The channel on which workflow audit information will be published if it is enabled.	None

Table 3-3 Runtime Properties (Continued)

Property	Description	Default
Web Server Name	Specifies the Web server that is used to display information to supervisors and participants when an activity is running. Format: protocol://Web-server-name[:port-number] where protocol may be http or https. Port-number is optional.	http://localhost:8091 (the Web server running in the Default Task Manager)
Mail Host	Specifies the name of the mail host that will be sending e-mail notification to model supervisors, activity supervisors, and performers.	None
Timer Load Duration	Specifies the amount of time (in seconds) that determines when timers load from the database upon recovery. Timers that are set to expire within this time are loaded.	300
Timer Cache Size	Specifies the maximum size of the cache that contains loaded timers. If set to 0, all timers are loaded.	0
Commit Retry Count	Number of times to retry before shutting down the project, if a commit failure occurs.	3
Commit Retry Interval	The number of seconds to pause between retries.	10
Ignore Source Connector Errors	Ignore source connector errors on start and allow project to be started partially. Appears in Project > Start Settings also.	False
Start Components Only	Start project components only. Source connectors are not started.	False
Admin Permission	Specifies the list of roles allowed to administer the project (such as starting and stopping the project). It can also take a special value, unchecked, to indicate no authorization checks. For more information on security, see the <i>BusinessWare Security Guide</i> .	Value of the Project object Default Permission property.

WORKING WITH PROJECTS

You work with projects in the Explorer of the BME. Unless otherwise noted, all project-related commands are accessible from one of the following:

- **Project** menu in the BME Main menu
- **Project shortcut** menu that activates when you right-click on a project object in the Explorer.

CREATING A PROJECT

The first step to developing a BusinessWare solution is creating a new project. To create a new project, select **Project > Project Manager** from the BME Main menu. In the Project Manager window, click **New...**. When you create a new project, it is displayed in the Explorer, as shown in [Figure 3-2](#).

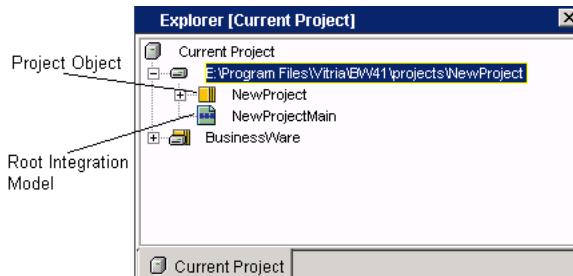


Figure 3-2 New Project in the Explorer

By default, every new project contains three objects:

- **Project object**—an object that allows editing of project properties and lets you synchronize shared projects. (More information about project synchronization is provided in [“Project Synchronization” on page 3-13](#).)
- **Empty root integration model**—a model that you can design to depict your entire business system. For more information about integration models, see [Chapter 6, “Integration Model Basics.”](#)
- **BusinessWare Project**—a default project module that is automatically added as a dependency for every new project you create in the BME. The *BusinessWare* project provides access to a suite of predefined types, which you use to develop and run a BusinessWare business solution.

After creating a new project, you develop your business system by adding and editing objects (such as models and resources) to define the project content. You also may change the project settings at any time. See “[Project Properties](#)” on [page 3-4](#) for more information.

SAVING A PROJECT

Save All saves all modified objects. **Save** saves the currently selected objects.

Save is enabled on the Project object when any project-related information is changed (for example, project properties, project mountpoints and project parameters).

To save your project, click **Save** or **Save All**.

OPENING AND RENAMING A PROJECT

Through the **Project Manager** (select **Project > Project Manager** from the main menu), you can open and rename projects. You work with one project at a time in the BME.

DELETING A PROJECT

Deleting a project by selecting **Project > Project Manager > Delete** removes the reference to the project from the BME. This method of deleting a project does not remove project directories or content from your hard disk. You cannot delete the currently open project.

Note: **Delete** is not the same as **Undeploy**. Delete does not affect the project already deployed in the directory server. You must undeploy the project to remove it from the directory server. See [Chapter 22, “Deploying Projects”](#) for more information on how to undeploy your project.

SHARING PROJECTS

Large business systems are often developed by a team. To support this, BusinessWare allows multiple developers who have their own instance of the BME to share projects.

You can share BusinessWare projects by creating a project JAR file. You create a project JAR file by:

- Exporting a project to create an importable project. An importable project contains the designtime objects as they exist in the BME at the time that the export is initiated. The project can be incomplete.

An Exported Project jar is created using the `ProjectManager | Export` command. The resulting project jar contains a snapshot of the project's current content (be it incomplete, invalid, inconsistent). The only thing you can do with an Exported Project jar is to import it.

- Deploying a project to create a project module. A deployable project contains the designtime and runtime objects in the project. Because a deployable project is created as a result of deployment, it is in a complete state.

A deployed project jar supersedes an exported project jar. In addition to containing a snapshot of the project content, it also contains the runtime artifacts generated by the project contents. Because it contains the runtime items, the deployed project jar can be deployed, and it can be specified on a classpath (in addition to being able to be imported). Notably, a Deployed Project is also known to be consistent, complete and valid (as it is only generated after the project has been validated—a side effect of build/compile).

Only use an exported project when you need to transfer a project “as is” with no guarantee of correctness.

These project sharing methods are described in “[Exporting and Importing Projects](#)” on page 3-13.

If two developers want to co-create a project, they can use any of the following methods:

- **Version Control System (VCS)**—developers work in isolation on their own workstations and periodically commit their changes to a centralized location.
- **Project Export/Import**—one developer exports a snapshot of the project to the other. The two developers cannot work concurrently.
- **Shared filesystem**—both developers mount the same shared network accessible filesystem.
- **Design Repository**—for more information, see [Chapter 4, “Design Repository.”](#)
- **Project Modules**—two developers can share work by creating a project module they both use that has their common items in it, for example, types, resources, and models.

Project Synchronization

To share projects, it is necessary to understand how to synchronize projects. As described in “[Introduction to Projects](#)” on page 3-1, projects are defined by the following characteristics: *content*, *mountpoints*, *parameters*, and *settings*. This information is stored in the project object. This enables the use of a file-based version control system to manage and share project data among multiple developers by storing the project object information in the version control system. As the modified project object is encountered by other developers, updates to the project object’s content, parameters, and settings are transparently received. Modifications to mountpoints require additional manual intervention. Synchronizing a project reconciles the differences in mountpoint settings when a shared project object is extracted from a version control system. The synchronization may result in the addition or the removal of mountpoints. If there are no changes in the mountpoint settings, then the synchronize command has no effect.

To synchronize a project:

1. Select the Project object in the Explorer.
2. Right-click and select synchronize from the shortcut menu.

Exporting and Importing Projects

Exporting a project creates a project JAR file that can be shared via email, FTP, or across the network. Another developer may import the file into their BME instance. The project file contains the contents (models and source code), settings (overall project properties), parameters, and mountpoints (location of the contents) of the original project.

Before a project can be imported, it must be exported.

Exporting Projects

There are two ways an exported project is generated:

1. Select **Project > Project Manager** from the BME Main menu, then click **Export** to export the current project.
Use this method if the project is not complete. The only thing you can do with the resulting project file is import it.
2. During deployment, select the **File:** checkbox.

If the Design Repository is enabled, then another checkbox is available on the deployment dialog, **Publish to Design Repository**. If you select this checkbox, the exported project is published to the design repository in addition to being deployed to file.

Use this method if you want a project file that can be imported, *and* is known to be in a stable, validated, complete state.

Before you export, determine which project objects should be delivered to the importing project. The **Exclude Files** property allows you to exclude content whose name matches the specified values. By default, the Exclude Files property specifies /cvs/ | ^cvs/ | ^GeneratedTypes/ which is useful in environments using concurrent versioning systems to exclude non-project files relevant to the versioning system only.

To exclude files:

1. Select the Project object in the Explorer.
2. Select the **Properties** tab in the Properties window.
3. Click in the Exclude Files field and enter a value.

Also, before you export, determine the mountpoint behavior you desire during subsequent imports. During export, an indication of the project's mountpoints will be exported.

Example: Assume you have a project with two mountpoints:

- f:\devel\samples\project\testProject\A
- f:\devel\samples\project\testProject\B

When other users import this project, the **Project Synchronize** dialog opens displaying these mountpoints, allowing modification in order to accommodate differences across hosts.

To facilitate the synchronization process, you can specify a mapping that will be used during project import. It is common to use environment variables to specify relative path names so that the path names can be automatically customized to the import user's environment.

Select the Project object and in the Properties window, select the **Mountpoints** property to launch the Mountpoints Mapping table. Map your export mountpoints to mountpoints that contain environment variables, for example,
f:\devel\samples\projects\testProject\A maps to
installdir\projects\SampleOneProject\A.

Subsequently, if a user on a machine with a different filesystem imports your project, when the Project Synchronize dialog displays, instead of seeing the structure as it is on your machine

(`f:\devel\samples\projects\testProject\A`), the user will see
`installdir\projects\SampleOneProject\A` and
`c:\ProgramFiles\Vitria\BusinessWare41\projects\SampleOneProject\A`.

For examples of this behavior, try to import the samples that are packaged with BusinessWare. For information on the samples, see
`installdir\samples\SamplesOverview.htm`.

Note: Project settings and parameters are not offered as part of project export or import because all settings and parameters are preserved at all times.

Importing Projects

Importing a project creates a new project and populates it with the previously exported content. At the time of import, if contents already exist in the filesystem that match the content being imported, a *collision policy* is used to define the behavior. The collision policy has three options that you can set:

- **Ignore**—skips the objects that already exist in the new project’s content and moves on with importing objects that do not exist.
- **Overwrite**—automatically overwrites the contents of the new project with that of the exported project regardless of whether the objects already exist.
- **Prompt**—prompts before overwriting the new project’s contents with that of the exported project.

Note: BusinessWare provides several samples, designed to illustrate BusinessWare features, which are simply import-able projects. These samples files are located in `installdir\samples\`, where `installdir` is the BusinessWare installation directory. These samples are exported using the `$VITRIA` environment variable and automatically are mounted to your filesystem at the time of importing a project.

It is considered a best practice to use slashes or forward-slashes “/” to delimit path entries of logical mountpoints within projects. The forward-slash is required if the projects are expected to be shared across platforms since the forward-slash is recognized across all platforms.

Note: Be sure that any environment variables used by any mountpoints of a project exist on the target file system before importing that project so that the logical reference in the export mountpoint will successfully be expanded.

Using Project Modules

Certain aspects of a project may be shared among multiple developers by creating a *project module*. A project module represents a runtime-ready instance of the project. You create a project module by deploying your project to a JAR file. Another developer can install the project module and make use of its functionality by adding it as a dependency to their project.

Adding a project dependency creates a relationship between two projects in which content defined within one project is available for use by another project.

You can share objects such as types, performer resources, channel and queue resources, database resources, Integration Servers, process models, process query models, transformer models, and java code. Objects accessed in dependent projects cannot be modified.

Note: Integration models and deployment configurations existing in project modules are not available for re-use.

Steps to Add a Project Dependency

The procedure for setting up a project dependency is summarized as follows. This procedure assumes that User A has created Project X and populated it with the desired shareable objects.

To create a project module and add a project dependency:

1. In User A's BME instance, deploy Project X, being sure to select the **File:** check box. This creates a project module.
2. User A makes the project module X available to user B by delivering the project JAR file. Delivery may be performed in many ways, including email, shared network drive, and FTP.
3. User B selects **Tools > Install Project Module** to select JAR X provided by user A.
4. User B can now use the command **Project > Add Project Dependency** to establish a linkage with project X.

Note: BME commands to install a project module and add a project dependency establish designtime linkages. In order for the project to be successfully executed at runtime, the dependent project module must be installed on the directory server. The `vtadmin deploy` command can be used to install the dependent project module on the directory server.

Alternatively, the Design Repository can be used to implement project sharing. If users A and B have configured their BMEs to reference the same Design Repository, then user B can establish a dependency directly using the Design Repository command **Add Project Dependency**. For more information on the Design Repository, see [Chapter 4, “Design Repository.”](#)

Removing a Project Module

Select **Tools > Uninstall Project** to remove a project module previously installed by selecting **Tools > Install Project Module**.

Note: You cannot remove a project module that is referenced by the current project. You must first unset the project dependency by selecting the dependent project in the Explorer and selecting **File > Unmount Filesystem** before selecting **Tools > Uninstall Project Module....**

STARTING AND STOPPING A PROJECT

You can start and stop projects several ways:

- **Using the BME**—when you are ready to deploy and start your project, select **Project > Start (Deploy)** from the BME Main menu. To stop your project, select **Project > Stop**. For information on starting and stopping a project during debugging, see [Chapter 27, “Debugging and Animation”](#) and the *BME Help*.
- **Using the BusinessWare Web Administration Console**—after a project is deployed, it is added to a list of deployed projects in the Web Administration Console. Select the project you want started or stopped, and click **Start** or **Stop**, respectively.
- **Using the vtadmin command-line tool**—can be used with the project start/stop option. See the *BusinessWare Administration Guide*.

For more information on BusinessWare Web Administration, see the *BusinessWare Administration Guide*.

USING THE PROJECT MANAGER

The BME maintains a list of known projects. You can manipulate (create, rename, delete, import, export) these projects via the Project Manager, shown in [Figure 3-3](#). To access the Project Manager, select **Project > Project Manager** from the main menu in the BME.

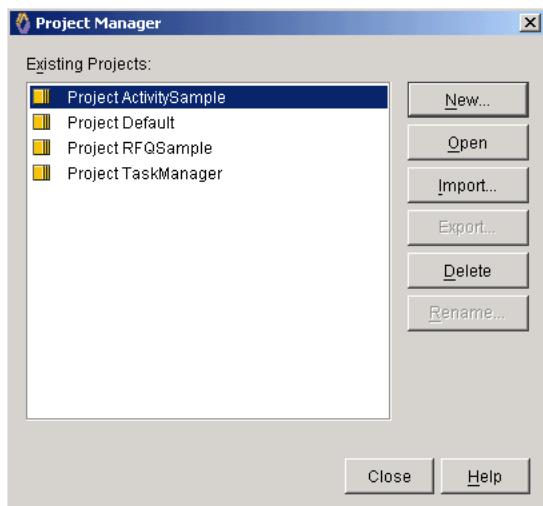


Figure 3-3 Project Manager Dialog

If you delete your project from the BME or install a new version of BusinessWare, you must re-establish the list of projects again within the BME. The project data itself is stored in the source files on your disk so that is not affected by a new install. Note that the project delete functionality will not delete the actual files on disk or from the directory server used by the runtime environment. It will only remove the project from the list of projects known to the BME.

To recognize a project from a previous installation:

1. Select **Project > Project Manager** from the main menu in the BME.
2. In the Project Manager window, select **New...**
3. Specify the main project directory in which your project exists.

Note: This is the directory where your `.project` file exists.

4. Enter the name of your original project.

Note: This name should match the name of your `.project` file.

5. Click **OK**.

6. The **Project Synchronization** dialog displays, asking you to validate the mounted directories in the **Mountpoint Mappings** table. If you have not changed your file directory structure, you do not have to do anything further. If your mounted directories do not look correct for your current machine, you may need to modify them.

Note: If you share a project across multiple machines, your directory locations may not be the same across all of the machines. This can happen if, for example, you have “checked-out” a project last edited by another team member on a different host (or you have multiple machines yourself).

7. Click **OK**.

The project displays normally. You can continue to edit, build, and run your project as before.

OBJECTS AND FILES IN THE BME

The BME manages objects that you can see in the Explorer. Although each of these objects may appear as a single element when browsing the BME, the object may be comprised of multiple files on disk.

There are two categories of files managed by the BME:

- **Source files**—files produced by the model designer. They include all definitions, graphics, and source code files that are edited directly by you (via some editor). For example, process model .java files are included in this definition, but .java files generated for integration models, process query models, or stubs are not.

These are the files that should be checked into a version control system for backup. A list of source files is included in [Table 3-4](#).

Table 3-4 List of Source Files

File Extension	Description
.3channel	3.x channel shortcut definition/properties
.axt	AXT transformation model
.afo	AXT custom function
.bim	Integration model definition/properties
.bpm	Process model definition/properties
.ccchannel	Composite channel resource definition/properties

Table 3-4 List of Source Files (Continued)

File Extension	Description
.cfg	Deployment configuration definition/properties
.channel	Basic channel resource definition/properties
.db2db	DB2 database resource definition/properties
.dtd	Source code for .dtd
.exm	ExceptionMap definition
.graphic	Model graphics information (process model, integration model)
.grp	Group resource definition/properties
.idl	Source code for .idl
	Informix database resource definition/properties
.iserver	Integration Server definition/properties
.java (if directly edited)	Source code for any .java file that the user directly edits including: process model source code, RMI types, implementation classes for BPOs, DOs, LocalData, RequestMap, ExceptionHandler, ProjectInit, other user-defined utility classes or implementation classes that are required by the project
.orcldb	Oracle OCI database resource definition/properties
.orclthindb	Oracle Thin database resource definition/properties
.pqm	Process query model definition/properties
.project	Project properties and mountpoint information
.properties	Source code for .properties – used with views as resource files
.pv	Process view definition/properties
.q	Queue resource definition/properties
.rchannel	Replica channel resource definition/properties
.rmi	A marker for files that must have RMI stubs generated for them; created when specify a java Remote interface to use as a type
.sqlsvrmdb	SQLServer database resource definition/properties
.sybase	Sybase database resource definition/properties
.tml	Transformer model definition/properties

Table 3-4 List of Source Files (Continued)

File Extension	Description
.type	CORBA types imported from .idl or .dtd Sometimes these are generated for process query models and will appear under the /GeneratedTypes folder. In that case, they are not considered source.
.typexd	Design-time type extensibility information
.user	User resource definition/properties
.xml	Source code for .xml
.xsl	Source code for .xsl – used with views where properties are .xml

- **Generated files**—all files that are generated from the source files. If deleted, they can always be re-derived from source files. This includes:
 - .class files
 - generated .java files
 - side-effect files of deployment

Generated files are derived files in your project. These files should *not* be checked into a version control system. The BME will regenerate them as necessary.

A list of generated files is included in [Table 3-5](#).

Table 3-5 List of Generated Files

File Extension	Description
.class	Compiled Java class files
.info	<p>Generated at deployment time, this contains:</p> <ul style="list-style-type: none"> • project parameterization and packaging information • project mountpoint mapping information • a listing of known port bindings that can be used by external clients wanting to know binding names • a list of known servlet names that can be used by external Web applications wanting to know how to invoke servlets in the project <p>The part of this file containing the project parameters can be modified and used to re-deploy the project with different configuration data.</p>

Table 3-5 List of Generated Files (Continued)

File Extension	Description
. jar	Generated when a project is exported or deployed The JAR file is produced by deploying your project represents an internally consistent, validated, executable, deployable version of your project. You may wish to store or backup this file.
. java (if generated and not directly editable)	Java files that the user does not directly edit. The files are generated when a primary object is compiled, including: CORBA stubs, integration model nesting rules, and process query helper classes.
. javahelpers	Generated when building process query models—used to keep track of generated java helper classes associated with a particular model.
. metakeys	Generated when building process query models—used to keep track of generated type information associated with a particular model.
. nbattrs	Generated when files are changed in a given directory— not required by BusinessWare
. stub	Maintains information related to . type files
. type	Considered generated files only if they are generated— typically in the /GeneratedTypes folder from building process query models. Otherwise, if they are created directly by the user as a side-effect of “Convert to Type,” then they are considered source.
. typexpr	Runtime type extensibility information, generated from the . typexprd data
. xql	XQuery files generated by AXT

UPGRADING OBJECTS

When importing a project created in a previous releases of BusinessWare or creating a project that points to a directory containing projects created in previous releases of BusinessWare, BusinessWare automatically upgrades the objects in those projects to the format used in the latest version of BusinessWare. BusinessWare logs information on the updates to bme.log and provides status messages in the Output window.

If you use a version control system, objects are modified even though you do not change them manually. In most cases, the changes BusinessWare makes should be sufficient. In some cases, you may need to manually trigger the upgrade process. This can happen if you are manually copying models into a project folder after the project was created.

To trigger the upgrade process:

1. Right-click the Project object in the Explorer.
2. Select **Upgrade Objects**.

VERSIONING PROJECTS

BusinessWare supports versioning of projects where multiple versions of the same project can run concurrently on a single BusinessWare Server instance.

MANAGING MULTIPLE PROJECT VERSIONS AT DESIGN TIME

At design time, you work with one version of a project at a time. You may maintain separate versions of a project by checking in each version to a version control system.

RUNTIME VERSION OF A PROJECT

The runtime version property defines the location in the directory server where the project files are deployed. When you deploy your project to the directory server, a directory is created under the BusinessWare root directory called `/Projects/<project name>/<project version>`, where `<project name>` is the name of your project and `<project version>` is the runtime version identifier that you assign at the time of configuring the project. Thus, the version identifier helps to avoid namespace collisions.

To modify the project version, select the **Project object** in the Explorer and select the **Version** property in the Properties window.

BUILDING AND COMPIILING PROJECTS

A BusinessWare solution is developed by creating models and other components, which together define the logical architecture and function of your business solution. As you develop a BusinessWare solution, the BME creates source objects on your filesystem. To run the BusinessWare solution, the source objects must be converted to a machine-executable form. This conversion is performed by the BME using the build or compile commands.

Build and compile commands are as follows:

- **Compile or Build**—compiles or builds the object selected in the Explorer and all objects underneath it, if it is a folder.
- **Compile Project or Build Project**—compiles or builds all the objects in the current project.

You can access the compile and build commands by:

- Selecting **Build Project** or **Compile Project** from the **Build** menu in the BME Main menu.
- Selecting an object or folder in the Explorer and selecting **Build** or **Compile** from the **Build** menu in the BME Main menu.
- Right-clicking an object in the Explorer and selecting **Build** or **Compile** from the shortcut menu. This builds or compiles the selected object and all objects underneath it, if it is a folder.

COMPILE AND BUILD BEHAVIOR

The Compile and Build effect on source and generated files is described in [Table 3-6](#).

Table 3-6 Compile and Build Behavior

File Type	Compile	Build
Source Files	<ul style="list-style-type: none"> • Saved if modified • Synchronized if a single BME Object contains two primary files that require it (for example, process model .bpm and .java files) 	Same as Compile
Generated Files	<ul style="list-style-type: none"> • Generated if they don't exist • Regenerated only if the associated primary file was modified 	Deleted and regenerated

Both the build and compile procedures validate the targeted object. Although projects are automatically validated during deployment, building or compiling a project before deployment is a good way to check that your models and other project objects contain no syntax errors prior to formal deploying and debugging. Any errors encountered during compile or build appear in the Output window.

Executing the build command creates all new machine-executable code required to run a project regardless of whether the project or its constituent objects were modified since the previous build.

DEPLOYING A PROJECT

Before you can start, test, or debug a project, you must deploy it.

Deployment includes the following tasks:

- *Partitioning*—process of specifying the BusinessWare Servers and Integration Servers on which BusinessWare components run.
- *Deploying*—process of installing BusinessWare executable files onto the directory server where they can be accessed by distributed systems that will run the BusinessWare solution.

For an extensive discussion of deployment, see [Chapter 22, “Deploying Projects.”](#)

ANIMATING A PROJECT

To demonstrate your project to colleagues or clients, you can run it in *animation mode*. As events flow through the project, the current port, state, or transition is highlighted. You can watch events progress through an integration model and into and through your process models.

See [Chapter 27, “Debugging and Animation”](#) for more information on how to animate projects.

DEBUGGING A PROJECT

It is important to test and debug your project to ensure the project executes correctly. The BME offers a number of handy project runtime tools to help you diagnose problems that can occur at the time of running your project.

For more information, see [Chapter 27, “Debugging and Animation.”](#)

4

DESIGN REPOSITORY

You can share and browse projects in the BME using the *design repository*. The design repository is a mechanism in the BME that enables you to explore project content outside your current project. You can locate objects contained in other projects for reuse.

This chapter describes how you can use the *design repository* to browse and share projects.

Topics include:

- [Working with the Design Repository](#)
- [Enabling the Design Repository](#)
- [Publishing to the Design Repository](#)
- [Using the Repository Explorer](#)
- [Disabling the Design Repository](#)

WORKING WITH THE DESIGN REPOSITORY

The design repository contains a catalog for shared projects and re-usable objects that you can view. It enables you to explore external projects and browse their contents to find objects to use in your current project.

Use the design repository to:

- Reference and/or use resources, types, or models in an external project by adding the project as a dependency.
- Copy and then modify objects from an external project, such as models, types, and resources, for use in your current project.
- Reference or call an external project through its proxies.

When you're browsing the contents of a project outside your current project, you are able to see the objects categorized by their type, that is, process models, integration models, and so on.

With your current project open, you can explore the content of an external project, and browse into the contents of that external project. For example, you can:

- **See an Explorer view**—you can identify each object in the external project by icon, type, name, and textual description. This includes types, models, and resources.
- **See a read-only view**—you can view read-only versions of anything that exists in the project.
- **Specify a project dependency**—select the external project and establish a dependency on it from your current project.

ENABLING THE DESIGN REPOSITORY

The design repository automates the process of delivering new versions of projects across the members of a workgroup. It accomplishes this by taking a snapshot of a project at the time it is deployed and copying it to a central location where it will be picked up by other connected users. Communication is achieved using filesystems. So, to set up the design repository, you need to specify the location your BME will publish deployed projects to, and the location from which you will pick up projects published by others.

When you install BusinessWare, you specify the location where the repository will be maintained. The Installer creates the necessary locations in your environment and sets values you can modify later.

When enabled, the design repository points to a filesystem location. You can also change the location of the design repository at modeling time.

To activate your design repository:

1. From the BME Main menu, select **Tools > Options > BusinessWare Options**.
2. Select the Design Repository property (...). A Design Repository Settings window displays.
3. Select the **Repository Type of FileSystems**.
4. Browse to set the **Local Directory**, as shown in [Figure 4-1](#), and select the directory that will serve as your local cached copy of the repository contents.
5. Browse to set the Repository Directory, as shown in [Figure 4-1](#), and select the directory where the repository contents are located.

6. Click **OK**.

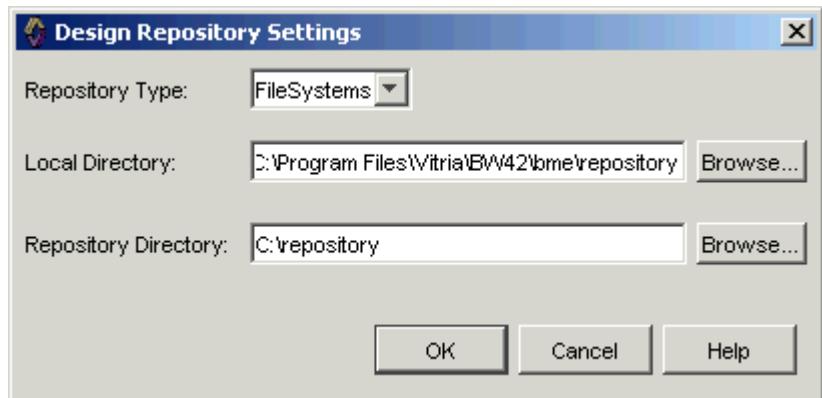


Figure 4-1 Design Repository Settings

PUBLISHING TO THE DESIGN REPOSITORY

During project deployment, you are given the option to publish your project to the design repository. If you have a working project and are ready to share it with others, you can place it in the design repository by selecting the check boxes in the deployment confirmation dialog for **File** and **Publish Project to Design Repository**.

Example: A group of modelers shares a repository server. Projects are published as they are deployed. They automatically appear in the user interface of connected modelers. Content from the shared projects is offered in a tree view categorized by type (process models, integration models, types, process queries, and so on). Objects can be copied and pasted from the shared project into the user's current project.

USING THE REPOSITORY EXPLORER

There are two Project Explorer windows. The first is the current project in the Explorer of the BME. The second is the Project Explorer you use to access the design repository.

To view the Repository Explorer:

1. From the BME Main menu, select **View > Repository**. This activates the Repository Explorer.
2. Click on the **Repository** tab in the Explorer, if it is not automatically selected.
An example Repository is shown in [Figure 4-2](#).

DESIGN REPOSITORY

Disabling the Design Repository

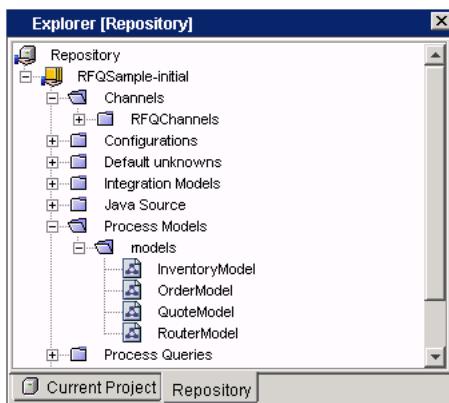


Figure 4-2 Repository Explorer

Shared projects may be expanded to display their content. Selecting objects will display their values in the Properties window. Objects that support opening (such as process models) may be opened for viewing.

The commands available for items in the repository are accessed by right-clicking the projects listed. The available commands are:

- **Add Project Dependency**—available if the selected project is not the base for the current project.
- **Remove Project Dependency**—available if the selected project is a dependent project to the current project.
- **Import**—can be used to create a new project in the BME with editable content initialized to the content of the project in the design repository.
- **Delete**—removes the selected project from the design repository.
- **Properties**—properties of the selected project.

DISABLING THE DESIGN REPOSITORY

Disable the design repository if you do not want to view other projects or publish your project to the repository.

To disable the design repository:

1. Select **Tools > Options > BusinessWare Options** from the BME Main menu.
2. Set the **Design Repository** property to **None**.
3. Click **OK**.

Note: During BME installation, you are prompted for local and remote directory settings. The remote directory is typically on a shared machine. The local directory is the one where copies of the files from the remote directory are stored/cached. The local cache is updated periodically and can be manually updated by selecting **Refresh** from the BME main menu on the repository root node. The default value of the local cache is supplied, *installdir/bme/repository*, and can be changed if required.

DESIGN REPOSITORY

Disabling the Design Repository

PART III: INTEGRATION MODELS

This part begins with an overview of BusinessWare types. It then describes the role of integration models and how to create them in your BusinessWare application. This part also includes an explanation of the various modeling components and of component communications via ports and wires. In addition, it describes channels, queues, and channel connectors as well as Vitria's support for Web services.

Chapters include:

- [BusinessWare Types](#)
- [Integration Model Basics](#)
- [Channels and Queues](#)
- [Application Server Integration](#)
- [Web Services](#)

PART #: TITLE HERE GOES HERE

5

BUSINESSWARE TYPES

BusinessWare *types* describe the structure of application data and the interfaces between project components using Defined Types and Java. BusinessWare stores and deploys these types as it does other project objects. This chapter gives an overview of types.

Topics include:

- [Overview](#)
- [Types Summary](#)
- [Converting Existing Files to Types](#)
- [Creating Defined Types](#)
- [Inspecting and Editing Types](#)
- [Exporting Types](#)
- [Sharing Types](#)

Access the *BME Help* for more information on creating types using type models.

OVERVIEW

BusinessWare models interact with many kinds of data and objects. For example:

- When a model invokes an *operation* on a component, the model passes *parameters* and receives results or throws an *exception*.
- Many process models instantiate and update *business process objects* (such as objects that represent customers or orders).
- BusinessWare Transformers consume *events* of one type (for example, XML events) and produce events of a different type.

Operations, parameters, exceptions, business process objects (BPOs), and events are all examples of BusinessWare *types*. To enable BusinessWare to store, transport, and manipulate your data, you describe it to BusinessWare using types. In addition to a *generic* BPO, you can define *specific* BPOs that describe your data and operations.

The kinds of types you create often depends on the approach you take when developing your BusinessWare project. If your business processes include handling XML documents, your BusinessWare project design could be *document-centric*, processing XMLEnvelope events. If your business processes include exchanging objects in synchronous calls, you could take an *object-centric* approach, passing parameters as Java objects with standard type mapping.

Note: If your BusinessWare project includes stateful models as discussed in [Chapter 10, “Process Models: Basic Concepts,”](#) you must use Defined Types.

DOCUMENT-CENTRIC PROJECT DESIGN

In a document-centric project, there are two data exchange designs:

- **Strongly-typed**—characterized by design time type-checking and better runtime performance characteristics such as:
 - ability to control persistence
 - smaller packets of data
 - typed invocations
 - fine-grained exception handling
 - data hiding
 - security
 - more efficient queries

A strongly-typed, document-centric design usually requires more effort during design, development, and maintenance. Most of the information in this chapter describes how to create a strongly-typed, document-centric project.

- **Loosely-bound**—a generic data mechanism is used to ‘hold’ all information. All information is available at any point during processing. A loosely-bound, document-centric design usually requires less maintenance and development is generally easier.

Loosely-bound, document-centric projects can take advantage of BusinessWare’s XML query functionality. XMLEvents produces an `xmlEnvelope` event that contains data in the payload. For more information on using XMLEvents, see [Chapter 14, “Process Model Code Construction.”](#) The *Order Process Sample* in %VITRIA%\samples provides an example of handling `xmlEnvelopeEvents`.

Note: These two designs are not mutually exclusive and most projects are a combination of the two.

DEFINING TYPES

You can define your types in the following ways:

- You can create types using Java. To do this in the BME, use either the New From Template wizard, or edit the source code using the Source Editor. You also must register existing Java files with the BME so they can be used as Java types.

Java types are commonly used to define operations used by process models, especially when working with XML Envelope events and creating or communicating with web services.

To convert incoming XML events to Java events, use the BusinessWare Transformer. See the *BusinessWare Transformer Guide* for more information.

- You can create an Defined Type by copying an IDL file into a mounted filesystem in the BME and converting it to type.
- You can use the Business Ware design repository to browse and share types across projects. For more information, see [Chapter 4, “Design Repository.”](#)

TYPES SUMMARY

The tables in this section summarize the BusinessWare types.

[Table 5-1](#) summarizes the principal BusinessWare types.

Table 5-1 Major Types Summary

Type	Description
Module	Container for a collection of types, typically interfaces and definitions they use. A module can contain modules. A DefinedType can contain only modules.
Interface	Container for definition of one or more synchronous operations (methods). An interface cannot include events.
Event Interface	Holds the definitions of one or more events. An event interface can only include events.
Business Process Object (BPO)	Defines your business data and operations (such as a customer order) that a stateful process model maintains, including user-defined data, BusinessWare-supplied data that describes the object’s current state, and user-defined operations that change behavior. A customer order is an example of a BPO.

Table 5-1 Major Types Summary (Continued)

Type	Description
Data Object (DO)	An auxiliary object to a BPO that often contains data but no operations (though it can contain operations). For example, a customer order BPO might refer to DOs representing items in the order.
Local Data	An interface that contains user-defined, transient data storage for use in process models (typically stateless process models).

[Table 5-2](#) lists the types you can use to define operations and events.

Table 5-2 Operation, Event, and Related Types Summary

Type	Description
Operation	A method that returns a result (possibly void) or throws an exception.
Event	A message typically transmitted asynchronously through channels or queues. You cannot define a result or an exception for an event. Events include XMLEvents. XMLEvents produces an <code>xmlEnvelope</code> event (Map header, DocumentReference[] payload, Part[] attachments). The event generated is: <code>com.vitria.types.XMLEvents.xmlEnvelopeEvent</code> For more information on XMLEvents, see Chapter 14, “Process Model Code Construction.”
Attribute	<ol style="list-style-type: none"> 1. Instance variable in a BPO or data object. 2. Attribute X is shorthand for two operations <code>getX</code> and <code>setX</code> in an interface. Attributes are not allowed in event interfaces.
Exception	Abnormal condition raised by an operation that cannot return a result.
Parameter	Input data required by an event or operation.

Table 5-3 summarizes user defined types.

Table 5-3 User Defined Types Summary

Type	Description
struct	Ordered collection of named values of primitive or composite types
sequence	Repeating series of the same type, that is, an array; can be bounded or unbounded
union	Type that defines multiple types, one of which is in effect at runtime
enum	Type that can have one of a set of specified values

Table 5-4 summarizes built-in types.

Table 5-4 Built-in Types Summary

Type	Description
boolean	Type whose value can only be True or False
char, wchar	Character and wide character
octet, unsigned short, unsigned long, unsigned long long	An 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the communication system
short, long, long long	Signed integers
float, double, long double	Floating point numbers
string, wstring	Character strings; length can be bounded or unbounded

IMPORTANT: You cannot mix Java and IDL interfaces on a port and you cannot specify more than one Java interface on a port.

CONVERTING EXISTING FILES TO TYPES

Using Interface Definitions from a File

You can use the following types of interfaces:

- Java
- Java RMI

- EJB
- IDL
- WSDL

The procedures for using Java interfaces, Java RMI interfaces, and EJB interfaces from a file are the same. For example, if the interface file you want is called `myInterface.java` and is located in a directory called `c:\MyProjects\MyInterfaces\com\acme\myProject`, you would use the following steps.

Using a Java, Java RMI, or EJB interface:

1. In the BME, mount the directory `c:\MyProjects\MyInterfaces` using **File > Mount Filesystem...** menu option.
2. Right click on the directory folder in the Explorer and choose **Tools > Use as Type**.

Once the above steps are complete, the interface definitions will become available for selection in the Port Types property.

Tip: If you want to use one of the interfaces in your directory as a type, but not all of them, right-click on a specific interface (instead of the folder) and choose **Tools > Use as Type**.

To make the interface definitions unavailable for selection, right click on the type in the Explorer and choose **Tools > Don't Use as Type**.

Using IDL interfaces from a file is slightly different. For example, if the interface file you want is called `myInterface.idl` and is located in a directory called `c:\MyProjects\MyInterfaces`, you would use the following steps.

To import an IDL interface:

1. In the BME, mount the directory `c:\MyProjects\MyInterfaces` using **File > Mount Filesystem....**
2. Right-click on the `myInterface.idl` file and choose **Tools > Convert To Type**.

Following a successful conversion, the BME generates corresponding type files which represent your IDL defined interface.

As shown in [Figure 5-1](#), generated type files are visible in the Explorer and will have the type file icon.

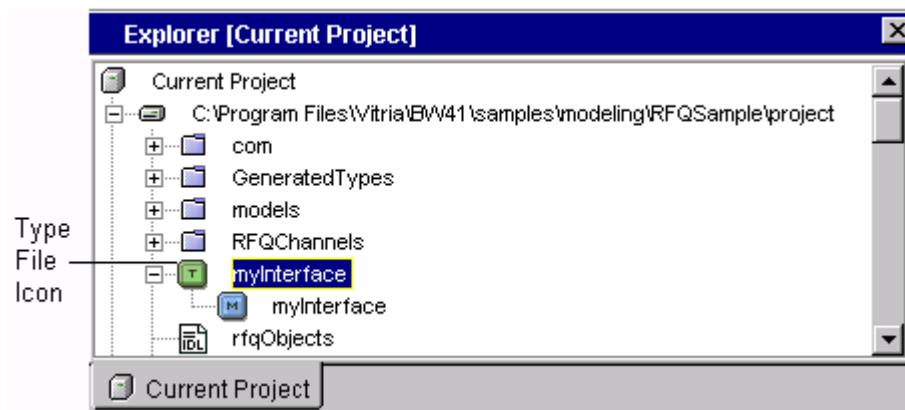


Figure 5-1 Type Icon in the BME Explorer

To undo the type generation, delete the generated type object or unmount the directory by selecting it and using **File > Unmount Filesystem**. For more information, see the *BME Help*.

IMPORTANT: A Java Interface can be used as a type only if it extends directly or indirectly from `java.rmi.Remote`.

CREATING DEFINED TYPES

Objects called **DefinedTypes** are collections of types created either by importing or by right-clicking on a project's file system and then choosing **New > All Templates... > Types > DefinedType**. Within a **DefinedType**, types are arranged in the containment hierarchy summarized in [Figure 5-2](#).

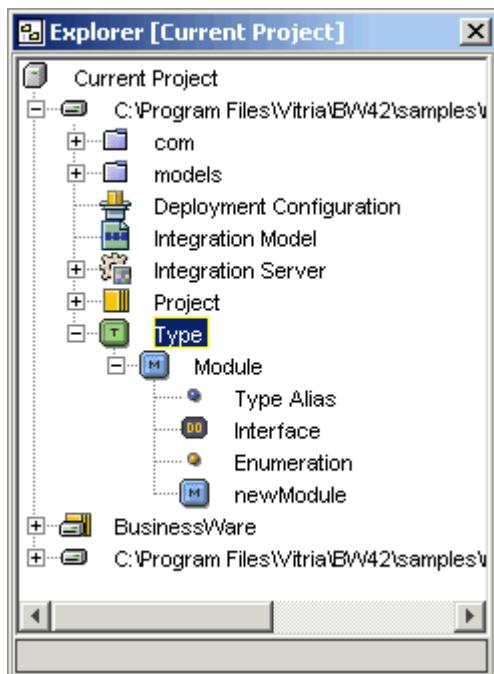


Figure 5-2 Type Containment Hierarchy

A *DefinedType* can contain modules; a module can contain constants, enums, event interfaces, exceptions, module interfaces, structs, *typedefs*, and unions. To add a subordinate type, right-click its superior and choose an item from the shortcut menu to show the legal subordinates. For example, if you right-click an interface, you see *New Attribute* and *New Operation*. For detailed instructions on adding types, access the *BME Help*.

You cannot create struct, enum, union, sequence, or exception types with the Type Modeler; nor can you create *typedefs* (type aliases), or constants. To create these, copy IDL or DTD files into a mounted filesystem and convert to type. After converting these types, you can refer to them in other type definitions you create with the Type Modeler.

If you roll the mouse over a type, a tool tip reveals whether it is an attribute, a struct, etc. To examine a type's properties (for example, what an operation returns), select the type and look in the Properties window. You also can change a type's property values in the Properties window.

INSPECTING AND EDITING TYPES

You can inspect and edit types by double-clicking on a port in the Editor. Figure 5-3 shows the Edit Types dialog that opens after double-clicking on a port.

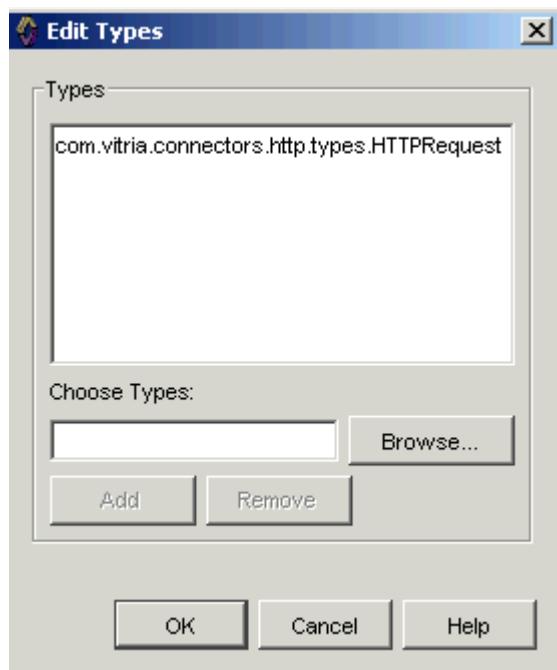


Figure 5-3 Edit Types Dialog

The Types field displays the currently assigned port type.

To specify additional types:

1. Click **Browse...** to open the Choose Interface dialog (Figure 5-4).
2. Select an interface and click **OK**.

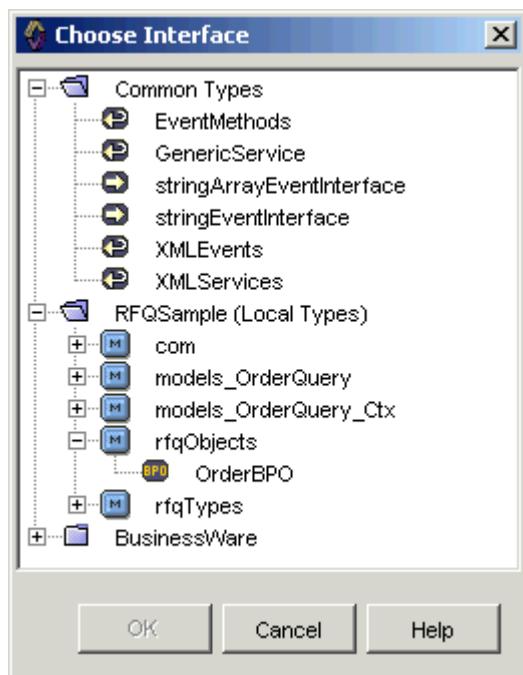


Figure 5-4 Choose Interface Dialog

For more information on ports, see “[Process Models: Ports, States, and Transitions](#)” on page 12-1.

EXPORTING TYPES

You can export a **DefinedType** as an IDL file or an interface as a Java file.

To export to IDL, right-click a **DefinedType** and choose **Export Type to IDL**. For more details on exporting types, see the *BME Help*.

Note: If you create a type from a DTD file, you cannot export it as an IDL file.

SHARING TYPES

You can share types between projects by defining the shared types in one project (known as a project module) and adding a dependency on the project module within other projects. For more information about project modules, see “[Using Project Modules](#)” on page 3-16.

Tip: When you create a types-only project, do not delete the default integration model that is created for the project. The default integration model is used in the validation phase of deployment. If the integration model is not present during the validation phase, you will not be able to deploy the project.

CONNECTOR TYPES

Most connectors have a set of pre-defined types and/or custom generated types that they accept or emit via their input or output ports respectively. The following sections outline the relationships and give guidelines for connectors and their types and ports.

PRE-DEFINED TYPES

Connectors may have either Java or Defined Types associated with them. You cannot mix Java types and Defined Types on a port and you cannot specify more than one Java interface on a port.

Pre-defined types for bundled connectors are included in the BusinessWare project. For example, the FTP connector's pre-defined types are in the module vtFTPConnectorEvents, and its Java types are located in the types directory ([Figure 5-5](#)).

BUSINESSWARE TYPES
Connector Types

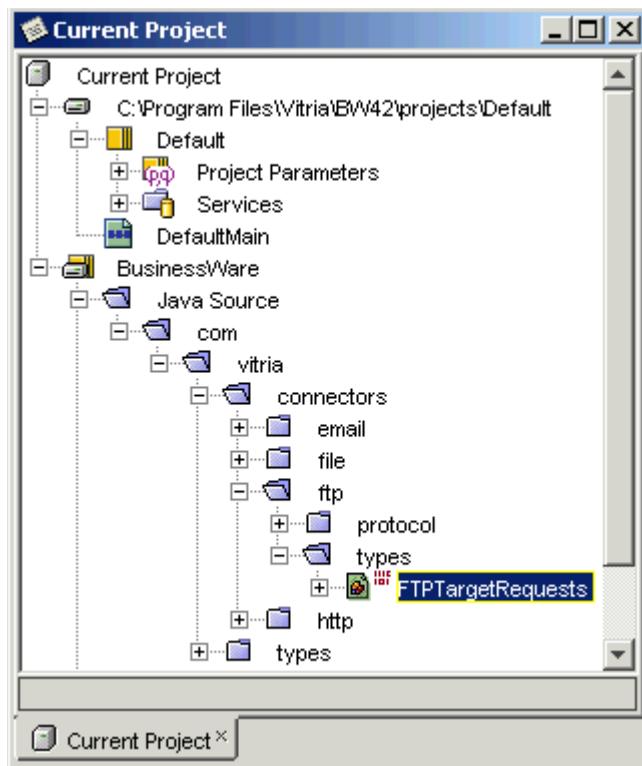


Figure 5-5 FTP Connector Types

For unbundled connectors, pre-defined types are included in the individual connector's project. For example, the RDBMS connector's project is `vtRDBMSConnector` and its pre-defined types are in the module `vtRDBMSEvents` ([Figure 5-6](#)).

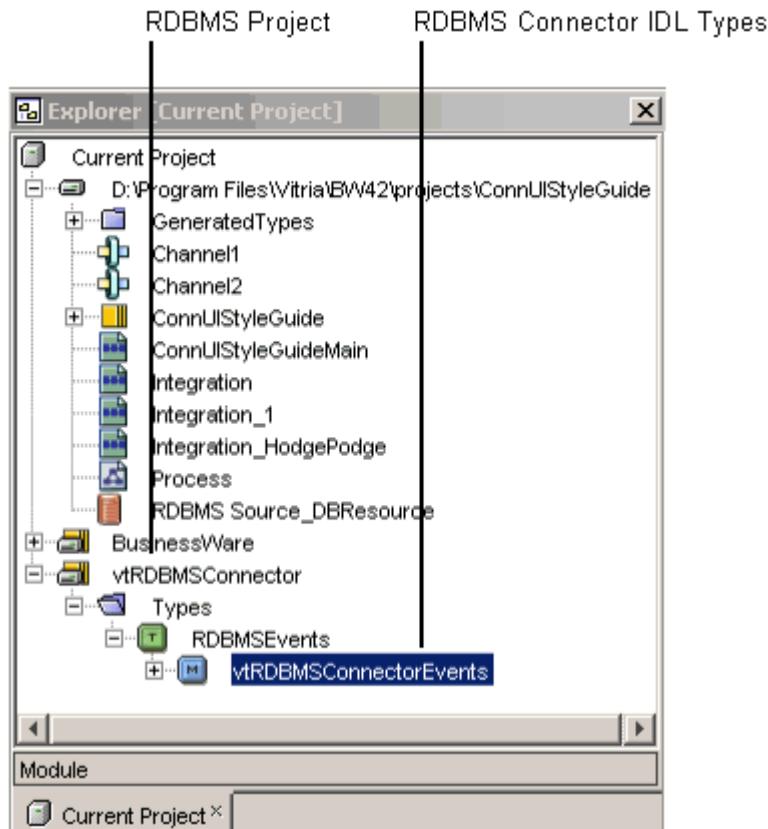


Figure 5-6 RDBMS Connector Pre-defined Types

CUSTOM GENERATED TYPES

Many connectors support custom types for interacting with specific objects in their external application. Typically, the connector provides a wizard for selecting and generating these types. The wizard creates these custom types in the current project's GeneratedTypes folder ([Figure 5-7](#)). For example, the RDBMS Connector Wizard lets you generate types corresponding to operations on specific tables and fields in a database.

BUSINESSWARE TYPES
Connector Types

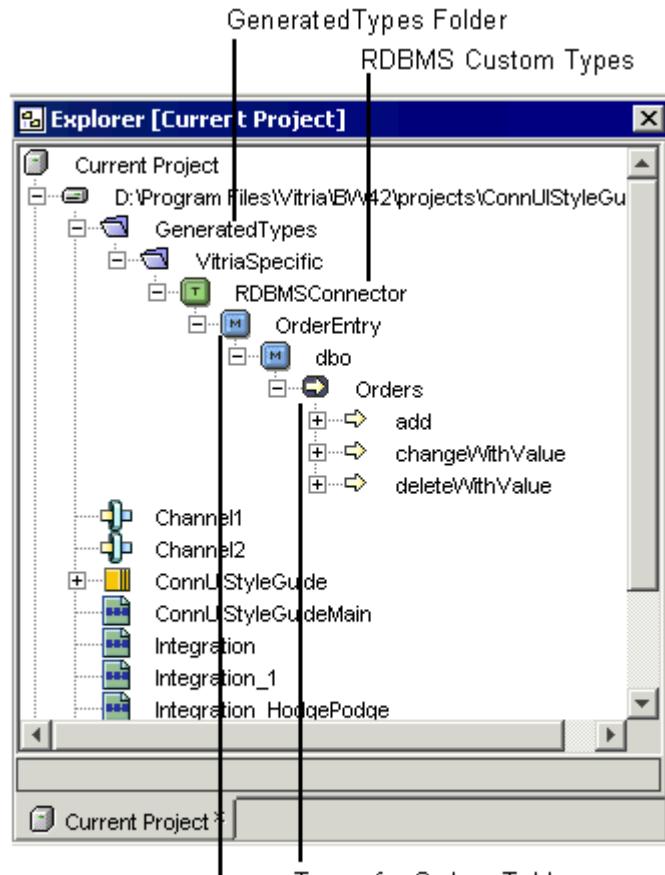


Figure 5-7 RDBMS Connector Generated Types

6

INTEGRATION MODEL BASICS

This chapter discusses the basics of *integration models*, which are graphical models that depict the overall logical structure of a business application. Topics include:

- [Introduction to Integration Models](#)
- [Creating Integration Models](#)
- [Components](#)
- [Connectors](#)
- [Ports](#)
- [Proxies](#)
- [Working with Integration Models](#)

INTRODUCTION TO INTEGRATION MODELS

Integration models are high-level models that present an end-to-end view of a business system or major subsystem. They identify the main components of the system and describe how those components interact with one another and with external entities. Integration models provide a graphical means of:

- Understanding the overall structure and operation of a business system
- Logically partitioning a business system into subsystems
- Identifying the major components of a business application
- Showing the relationships between those components across internal and external systems
- Specifying the data exchanged by the components
- Showing the direction of data flow

Every BusinessWare project has one *root integration model*, which is the top-level model in the project. Depending on the complexity of the application, there may also be nested integration models, which represent individual subsystems.

EXAMPLE

As an illustration, consider two integration models that are used to define a business-to-business application in the Request for Quote Sample ([Figure 6-1](#)).

To access instructions for installing and running this sample, see *installdir\samples\SamplesOverview.htm*.

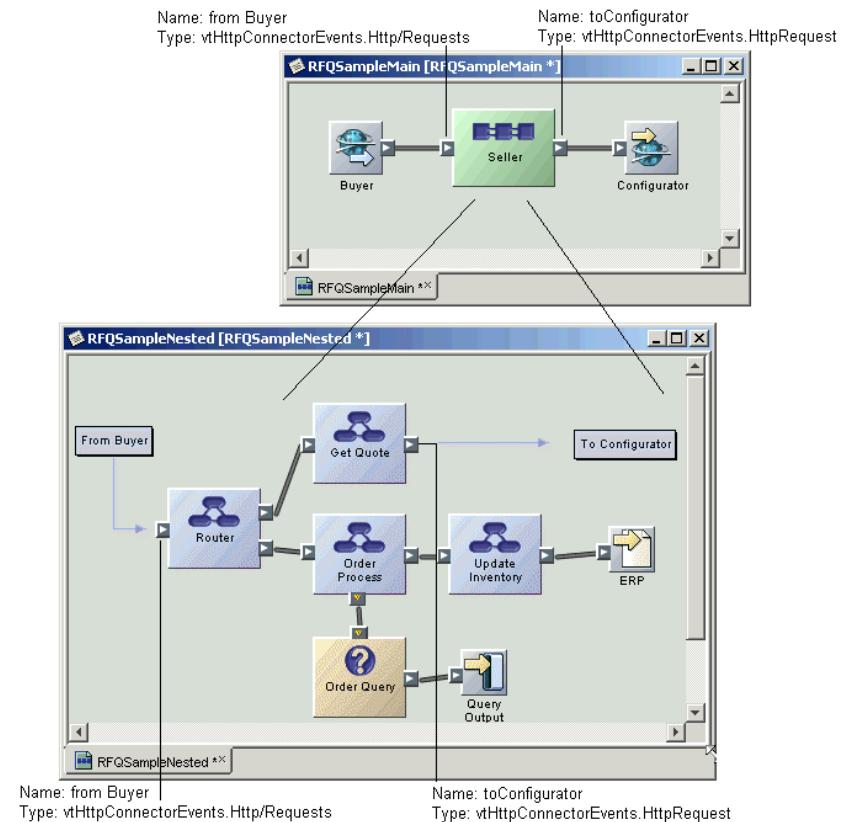


Figure 6-1 Integration Models in Request for Quote Sample

In this example, the company distributes memory modules to customers who request quotes and place orders over the Web. The root integration model, RFQSampleMain, shows the three main components of the system:

- **Buyer**—receives the requests for quotes and orders that are placed by customers over the Web and initiates requests on the Seller component.

- **Seller**—processes the requests for quotes and orders. Processing involves a variety of tasks, which are shown in the nested integration model. The Seller component in the root model links to the nested integration model.
- **Configurator**—communicates with the manufacturer who produces the memory modules.

By relegating the details of the Seller subsystem to a nested integration model, you are able to present a clear, uncluttered view of the overall system in the root model.

INTEGRATION MODEL CONTENTS

Figure 6-2 shows the types of objects that can be included in integration models.

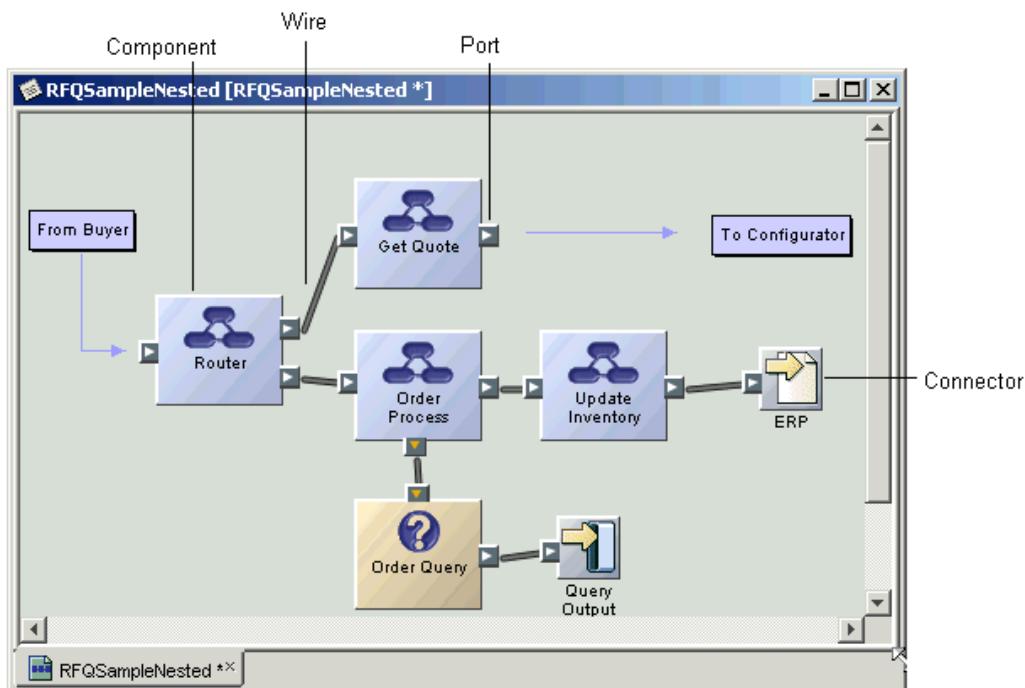


Figure 6-2 Objects in an Integration Model

Integration models consist of the following:

- **Components**—are modeling representations of a subsystem within the overall system. Components in the integration model are linked to other models that actually define the structure and operation of the subsystem.

For example, the RFQSampleNested model shown in [Figure 6-2](#) contains four process components, each of which is linked to a process model that defines a particular application, and one process query.

- **Connectors**—provide connectivity between BusinessWare components and external entities, such as a file system, database, channel, or SAP system. A basic set of connectors comes with BusinessWare; other connectors can be purchased separately.

Two connectors are included in the RFQSampleNested model: a Channel Target Connector, which publishes data output by the Order Query component to a channel, and a File Target Connector, which writes data output by Update Inventory component to a file.

- **Proxies**—represent interactions and provide connectivity between components within a project and objects outside of the project using either Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), Enterprise Java Beans (EJB), or a Web service. See [Figure 6-12](#) for an example of a proxy in an integration model.
- **Ports**—define the interfaces to components and indicate the direction in which communication occurs. By specifying the kinds of events or operations that a port can handle, you control what type of data a component can receive or output.
- **Wires**—connect an input port on one component to an output port on another component. Wires are graphic representations of the communication links between specific components.

CREATING INTEGRATION MODELS

In the BME, you construct integration models graphically by inserting and configuring various elements in the Integration Model Editor. Basically, there are two approaches you can use:

- **Top-down approach**—Begin by designing the root integration model to get a clear understanding of the logical structure of the application. Then create all the subordinate models and other objects that comprise the application.
- **Object-centric approach**—Begin by creating a “library” of modeling objects and models of individual processes; then construct a hierarchy of integration models that show the relationship and interaction of those elements.

Either approach is valid, and you may well find that you use a mixture of both.

Before creating an integration model, you should complete the following tasks:

- Create a project
- Define or choose event interfaces and operation interfaces

You must create a project first since all models are created within a project.

Interfaces must be defined so that you can specify the kinds of data that can be accepted or output by a given component.

The basic procedure for creating an integration model is outlined below. Where appropriate, cross-references to additional information are provided. In addition, the *BME Help* offers detailed instructions for using the BME.

To create an integration model:

1. Create a new project by selecting **Project > Project Manager**. Select **New** from the Project Manager dialog box and enter the name and location of the new project. The BME automatically creates an empty root integration model, names it *projectMain*, and displays it in the Editor.
2. Create a new integration model using either of these techniques:
 - In the Explorer, right-click on the project directory and select **New > Integration**. When prompted, provide an object name for the integration model. Click **Finish**.
 - Select **File > New...** to display the New Wizard. Expand **Models**, select **Integration**, and click **Next**. When prompted, provide an object name for your integration model and select a folder in which to create it. Click **Finish**.

Note: No spaces or special characters are allowed in the name because it will be used as a valid Java class name.

Add components to the model to represent individual subsystems.

To add a component:

1. Click on a component button  in the Integration tab of the Palette, and then click in the Editor.
2. Set the component properties by selecting the component and entering values in the Properties window.

See “[Components](#)” on page 6-9 for information on the types of components and their properties.

Add connectors or proxies to allow interaction with external systems.

To add a connector or proxy:

1. To add a connector, click on a connector button in the Connectors tab of the Palette and click in the Editor window.



Figure 6-3 Connector Palette

To add a proxy, click on a proxy button in the Proxies tab of the Palette and click in the Editor window.



Figure 6-4 Proxies Palette

2. Click in the Editor window to add the connector or proxy.
3. Set properties by selecting the connector or proxy and entering values in the Properties window.
4. If appropriate, you can link connectors to resources by setting the resource link property. Channel connectors, for example, must be linked to channel resources.

See “[Connectors](#)” on page 6-12 for information on the types of connectors and their properties. See “[Proxies](#)” on page 6-28 for information on the types of proxies and their properties.

Attach additional ports to components if needed.

To add a port:

1. Click on a port button    in the Integration tab of the Palette.
2. Click on the component.
3. Configure ports by naming them and specifying events.

For more information, see “[Ports](#)” on page 6-15.

Wires connect the input and output ports of components.

To wire components:

1. Click on the Wire button  in the Integration tab of the Palette.
2. Click on the output port of one component; then click on the input port of another component.

Tip: Press the **Shift** key and click while drawing a wire to create right-angled, segmented wires. To remove a segment, press the **Ctrl** key and select the segment origin.

For more information, see [“Wiring Ports” on page 6-26](#).

Link components to models to define the logical structure of your business system.

Link components to models using any of these techniques:

- Double-click on the component. Then choose an existing model from the selection box or click **New** to create a new model.
- Right-click the component and select **Tools > Open Nested Model**. Then choose an existing model from the selection box or click **New** to create a new model.
- Select the component and set the Link property or Nesting Specification in the Properties Windows.

See [“Linking Components to Models” on page 6-11](#).

To save a model and build it into a project:

1. Save the integration model and the other objects you have created by selecting **File > Save All**.
2. Compile the project at this point to validate your work by choosing **Build > Build Project**. For more information, see [Chapter 3, “Projects.”](#)

DISPLAYING OPTIONS, LAYERS, LABELS, AND ANNOTATIONS

You have considerable control over the appearance of your integration models.

To customize an *individual integration model*, use any of these options:

- **Display Options**—For the model itself and for each element in the model, the Properties window has a Display tab. For components, connectors, and proxies, you can change the background or inner color, add a logo (any image in .gif format), or add a relief or black border. For wires, you can change the wire width.

- **Layers**—The Layers tab in the model’s property sheet lets you hide or show port names, port types, annotation, and wire names.
- **Labels**—Elements in the model are automatically labeled with the names you give them. To edit the label or change its placement, right-click on the object and select **Enable Label Selection**. You can then select the label, move or resize it graphically in the Editor, and use the Properties window for changing the font, editing the label text, “pinning” the label so when moved it stays connected to a component with a line, and setting the text (foreground) color.
- **Annotation**—Using the Annotation tab of the Palette, you can add labels (for tooltips), shadowed labels, lines, filled rectangles, filled ellipses, or logos (any image in .gif format) to your model. For example, you might want to list the names of the external systems that feed data to or get data from the BusinessWare connectors. The property sheet for each of these annotation elements lets you change the color, change the font or line width, and add a tooltip annotation.

To set display options globally for *all integration models*, select **Tools > Options**. In the Options window (shown in [Figure 6-5](#)) select **Integration Modeler Options** and make the changes you want.

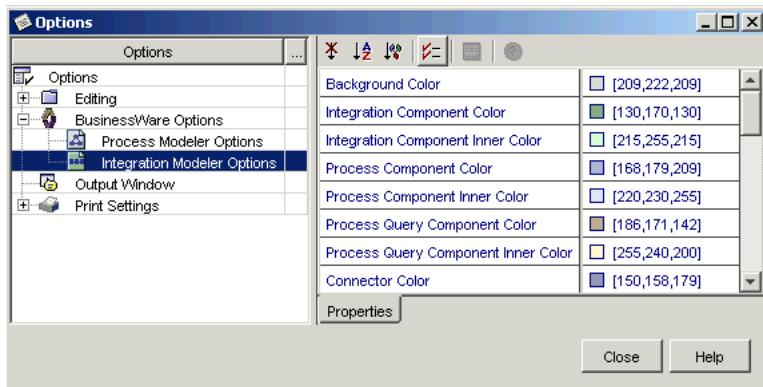


Figure 6-5 Setting Display Options for Integration Models

COMPONENTS

In BusinessWare, *components* are graphic representations of the subsystems in the solution. In a sense, components are “stand-ins” for the actual subsystem models. They are used to simplify the integration model so that anyone can easily grasp how all the various subsystems are related and interact.

Components are linked to models, which encapsulate the logic for that particular subsystem.

TYPES OF COMPONENTS

Integration models can contain the following types of components:

- **Integration component**—links to a nested integration model. A nested integration model graphically defines the logical structure of a major subsystem or closely related subsystems of a business solution. By using a hierarchy of nested integration models, you can simplify and clearly model the most complex business systems. The link between component and model is dynamic so you can specify several different integration models that may be used, depending on conditions that are evaluated at runtime. For more information, see [“Using Nested Integration Models” on page 6-37](#).
- **Process component**—links to a process model. A process model is a statechart diagram that defines a business process. It embodies the business rules by which business objects (such as a customer order) are processed across business systems in a well defined, logical manner. For more information, see [Chapter 10, “Process Models: Basic Concepts.”](#)
- **Process query component**—links to a process query model. A process query model is a set of queries that are run against the BPOs (business process objects) that are processed by a process model. The query results are a means of measuring the quality of service delivered by the business application. For more information, see [Chapter 17, “Process Query Models.”](#)

SETTING COMPONENT PROPERTIES

To set component properties, select the component and then enter the values you want in the Properties window.

[Table 6-1](#) describes the properties for the various types of components.

Table 6-1 Properties of Components

Property	Description
Integration Component	
Name	Name of the integration component.
Nesting Specification	List of one or more integration models associated with this component. The decision of which model to invoke is made at runtime, based on an evaluation of the triggering events and conditions specified for each model in the Nesting Specification. For more information, see “ Nesting Specification ” on page 6-40.
Exception Handler Class	The Exception Handler implementation class is used to process component system exceptions and application exceptions. Enter the fully qualified Java class name. For more information, see Chapter 26, “Exception Handling.”
Process Component	
Name	Name of the process component.
Process Model Link	Link to the process model associated with this component. Each process component must link to one process model.
Local Data Class	The Local Data class is a default class used to hold transient data during the course of a single event trigger. The data is no longer valid when the processing is completed. Local data is used primarily in stateless models across a series of chained action states. Enter the fully qualified Java class name.
Object Table	<p>The Object Table maps each object implementation class (for the BPO and its associated DOs) to the database table where the BPO data or DO data is persisted. Based on this mapping, the runtime can locate the data for the appropriate BPO instance.</p> <p>This table is filled in by default. For additional information on the object table, refer to Chapter 11, “Process Models: Defining and Using Business Objects.”</p>
Exception Handler Class	<p>The Exception Handler implementation class is used to process component system exceptions and application exceptions that are not caught in the model. Enter the fully qualified Java class name.</p> <p>The default value is blank. It is not necessary to specify a value for this property. For additional information on exception handling, refer to Chapter 26, “Exception Handling.”</p>
Process Query Component	

Table 6-1 Properties of Components (Continued)

Property	Description
Name	Name of the process query component.
Process Query Model Link	Link to the process query model associated with this component.

LINKING COMPONENTS TO MODELS

Components, as graphic representations of subsystems, include a link from the component to the model it represents.

You can link the component to an existing model, or you can create a new, empty model while defining the link. There are several methods for doing this and all of them set the component's "link" property.

- Double-click on the component. If a link has been established already, a picker opens from which you can select an existing model or create a new model. Double-clicking on an existing link opens the linked model.
- Select the component, right-click on it, and select **Tools > Open Nested Model**. Then choose an existing model from the selection box, or click **New** to create a new model.
- Select the component, and set its "link" property in the Properties window by entering the name of the model.
- To change an existing link, select the component and edit its "link" property.

Linking Integration Components to Multiple Models

By default, integration components are linked to a single integration model. However, integration components can be linked to multiple integration models. The decision of which model to invoke is made at runtime, based on the triggering events and conditions you specify. These specifications are defined when you set the Nesting Specification property.

To specify the integration models and the conditions under which they are invoked:

1. Select the Integration component.
2. In the Properties window, click the Nesting Specification property and then click .
3. Click **Add**. In the row you created, specify the model that should be invoked. Edit the Trigger and Condition columns to specify when the integration model should be used.

4. Add as many rows as necessary to specify all the integration models and conditions that you require.
5. Click **OK**.

CONNECTORS

Connectors enable BusinessWare components to interact with external systems. They are the foundation for enterprise application integration (EAI), allowing your BusinessWare solution to accept data from and send data to your back-end systems and legacy applications.

BusinessWare comes with a base set of connectors that provide connectivity to file systems, email systems, Web servers, channels, and queues. Many other connectors designed to meet specific needs can be purchased separately. For example, there are database connectors, SAP connectors, and Oracle Applications connectors. For a complete list of connectors, visit the Vitria Technical Support Web site at <http://help.vitria.com>.

CONNECTOR TYPES

All connectors act as either source or target connectors:

- *Source connectors* receive data from an external source, channel, or queue, convert it to BusinessWare events, and forward it to a component in the model.
- *Target connectors* receive BusinessWare events from model components, convert it to the format required by an external system, and send it to that external target, channel, or queue.

Table 6-2 describes the connectors that are packaged with BusinessWare.

Table 6-2 Connectors Included with BusinessWare

Icon	Connector	Function
	File Source	Monitors a particular directory and all of its subdirectories for changes to files and produces events to reflect detected changes.
	Line Source	Monitors a particular ASCII file for the addition of new records or lines and produces events to reflect detected changes. The Line Source Connector is best used for monitoring a log file or other types of streamed output data.
	File Target	Receives file data events and writes copies of the file data into files in a target directory. The data received can be written into separate files or into a single file.
	FTP Source	Monitors a given directory on an FTP server for changes to files or the creation of new files and publishes those changes as events.

Table 6-2 Connectors Included with BusinessWare (Continued)

Icon	Connector	Function
	FTP Target	The FTP Target Connector receives events (with file content inside or pointing to the local disc) and transfers them as files into the specified directory on an FTP server.
	HTTP Source	Accepts HTTP requests and translates them into BusinessWare events. The HTTP Source Connector is actually a servlet that runs within the context of a servlet engine
	HTTP Target	Translates BusinessWare HTTP events into HTTP requests, sends the request to a target Web server, and receives the HTTP response.
	Email Source	Monitors an IMAP4 or POP3 mailbox (Internet Message Access Protocol version 4 or Post Office Protocol version 3). Retrieves new messages and sends them out as events.
	Email Target	Receives events, turns them into email messages, and then mails them out using a Simple Mail Transfer Protocol (SMTP) server.
	Channel Source	Subscribes to a BusinessWare channel for events.
	Channel Target	Publishes events to a BusinessWare channel.
	Channel TargetSource	Publishes events to a BusinessWare channel and subscribes to the same channel for events.
	Queue Source	Subscribes to a BusinessWare queue for events.
	Queue Target	Publishes events to a BusinessWare queue.
	Queue TargetSource	Publishes events to a BusinessWare queue and subscribes to the same queue for events.

For additional information on the File, HTTP, and Email connectors, see the following documents, which can be accessed from the BusinessWare documentation index (*installdir\doc\docindex.htm*):

- *File Connector Guide*
- *FTP Connector Guide*
- *HTTP Connector Guide*
- *Email Connector Guide*

For additional information on channel and queue connectors, see [Chapter 7, “Channels and Queues.”](#)

GUIDELINES FOR CONFIGURING CONNECTORS

Keep these points in mind as you add connectors to your integration model:

- Connectors have a predefined number of ports. You cannot add ports to a connector. For more information, see “[Ports](#)” on page 6-15.
- Most connectors are designed to handle specific event types. It is possible to edit the default event types that a connector handles. See “[Editing Connector Port Event Types](#)” on page 6-14 for more information.
- Channel and queue connectors can be configured to handle all types of events.
- You cannot wire a source connector directly to a target connector. Some other component, such as a process component, must be placed between the two connectors.

EDITING CONNECTOR PORT EVENT TYPES

You can edit the list of event types that a connector handles by selecting from a number of interfaces that are available to the project.

1. Double click a connector port, or click  in the Port Types field in the connector’s Properties Window to open the Edit Types dialog box.
2. Click **Browse...** to open the Choose Interface dialog box. The chooser will show you a list of interfaces known to the current project. The list of interfaces will include interfaces in projects that are referenced as dependent projects. This includes the BusinessWare project, which contains many useful interfaces.
3. Select an interface.
4. Click **OK** to close the Choose Interface dialog box.
5. Click **OK** again to close the Edit Types dialog box.

LINKING CONNECTORS TO RESOURCES

In BusinessWare, a *resource* is a logical representation of a physical entity, such as a channel, database, server, or role.

Some connectors must be linked to resources during the modeling phase so that they can utilize the corresponding physical resources during runtime. For example, the channel connectors must be linked to a channel resource and the RDBMS connector must be linked to a database resource.

To link a connector to a resource:

1. Create the resource, using either of these techniques:
 - In the Explorer, right-click on the project directory, and select **New > All Templates...** and expand **Resources**. Select the resource and click **Next**. Provide a name and click **Finish**.
 - Select **File > New...** to display the New Wizard, and expand **Resources**. Select the resource and click **Next**. Provide a name and folder location and click **Finish**.
- Note:** For more information, see the *BME Help*.
2. Select the connector in the Editor.
3. In the Properties window, select resource properties for the selected connector.

POR TS

Ports are used to define the communication interfaces between components. These communication interfaces are software protocols supported by BusinessWare. Through these interfaces application data can be exchanged between modeling components, third-party applications, and software objects. Supported protocols include Java RMI, J2EE EJB, CORBA IIOP, CORBA GIOP, and WSDL with HTTP binding.

INPUT AND OUTPUT PORTS

There are two functional categories of ports: input ports and output ports. Ports define the input and output interfaces of a modeling component. Some components such as connectors have a fixed number of ports (typically one), while other components such as process models can have multiple ports.

To add a port to an integration or process model component, select the port from the Integration tab of the Palette (shown in [Figure 6-6](#)) and place it on an edge of the component. The arrow on the port shows the direction of the data flow across it.

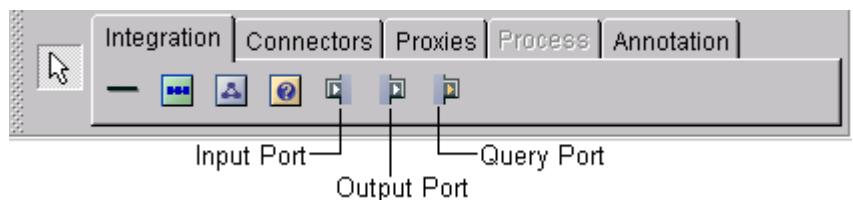


Figure 6-6 Palette Integration Tab

Besides input and output ports, BME has another kind of port called a query port. A query port is a special output port that can be added only to process model components. The interface on query ports is fixed and you cannot change it. See the RFQSample (in *installdir\samples\modeling\RFQSample*) for an example of how a query port can be used.

To remove a port, select it in your model and then use the Delete key. Ports cannot be added to or removed from connectors.

PORT PROPERTIES

To view port properties in the shared Properties window, select a port. [Table 6-3](#) shows the port properties available for input ports.



Figure 6-7 Port Property Window

In [Table 6-3](#), “X” marks the properties that apply to each kind of port. See the following sections for in-depth descriptions of each property.

Table 6-3 Port Properties

Property	Description	Input Port	Output Port	Query Port
Name	Name of the port. Port names can contain no spaces or special characters since they must be valid Java variable names. All the ports attached to a single component must have unique names.	X	X	X
Kind	Indicates whether the port is an input or output port; read-only.	X	X	X

Table 6-3 Port Properties (Continued)

Property	Description	Input Port	Output Port	Query Port
Port Types	<p>Interface definitions that define what kind of information flow into or out of the port.</p> <p>You can select existing interfaces, import them from other sources, or create your own. See “Port Types Property” on page 6-18 for more information</p>	X	X	X
Request Map Class	<p>The implementation class used by the runtime to determine how to dispatch a particular event. The RequestMap class checks the incoming event for an <code>oid</code> or other primary key that the runtime can use to retrieve the correct BPO record from the database. It also groups events so that all events related to a particular BPO instance are processed in the same thread. See “Request Map Property” on page 6-19 for more information.</p>	X		
Transaction Control	<p>Defines one of several types of transaction controls for input ports of components.</p> <ul style="list-style-type: none"> • Required—Use the incoming transaction context if one exists or create a new transaction if one does not exist. • RequiresNew—Always create a new transaction context whether one exists or not. • Mandatory—The incoming request must propagate a transaction context. <p>The default is Required.</p>	X		
Dynamic Port	<p>If set to <code>True</code>, specifies that the port is to be connected to a dynamic proxy. The Dynamic Map Class property of the proxy must be configured. Default is <code>False</code>.</p> <p>For more information on the Dynamic Map Class, see “Using Proxies with the Dynamic Map Class” on page 6-35.</p>		X	
Type Propagation	<p>If set to <code>True</code>, the types assigned to this port will be automatically assigned to the port it is wired to (if that port is currently untyped). Default is <code>True</code>.</p>	X	X	
Invoke Permission	<p>Valid for integration and process components only. Specifies the list of roles allowed to invoke the port. Leaving this property unchecked specifies that no authorization checks need to be completed.</p> <p>Note: This property value cannot be changed on a query port.</p> <p>For more information on security, see the <i>BusinessWare Security Guide</i>.</p>	X		X

Name Property

Every port must be given a name and that name must be unique within a model. The syntax of the name you enter for the Name property must obey Java variable syntax (because port names are converted to Java variable names when projects are built).

Kind Property

The Kind property is read-only and identifies whether the port is an input or output port.

Port Types Property

The Port Types property defines the interface of the port. You can use the following types of interfaces on ports:

- CORBA Interface Definition Language (IDL)
- J2EE Enterprise Java Beans (EJB)
- Java Remote Method Invocation (RMI)
- Java interfaces
- Web Service Definition Language (WSDL)

You can learn more about these interface definitions by searching the Web. A good starting point for your research is <http://www.java.sun.com>.

You can find examples of how these interface definitions are used in BusinessWare by exploring the following samples included in your package:

- A sample IDL file called `orderTypes.idl` can be found in the `installdir\samples\protocols\SimpleOrderSample` directory. This sample can also be accessed from the CORBA interoperability sample web page (in `installdir\samples\protocols`).

Note: If you are familiar with the IDL syntax for event interface definition in BW 3.x, you should note that the use of the “event” keyword has become optional in BW 4.x. For more information, see [Appendix A, “Event Interoperability.”](#)

- A sample of an RMI interface file called `OrderRequestsRMI.java` can be found in the `installdir\samples\protocols\SimpleOrderSample` directory. This sample can also be accessed from Java RMI interoperability sample Web page (`installdir\BW41\samples\protocols\SimpleOrderSample`)

- Sample EJB interface files `OrderRequestsEJB.java` and `OrderRequestsEJBHome.java` can be found in `installdir\samples\protocols\SimpleOrderSample`. These samples can also be accessed from the WebLogic interoperability sample web page `installdir\BW41\samples\protocols\`)
- Sample WSDL files can be found in the `installdir\samples\modeling\WebServiceSample` directory.

You do not need to be familiar with all the interface definitions supported by the BME in order to set up interface definitions for ports. You are free to choose the interface definition that best meets your project needs.

IMPORTANT: You are not required to select an interface for the Port Type property. If the Port Type property is left unset, the port is known as untyped. Otherwise, it is called a typed port.

See “[Port Interface Definition](#)” on page 6-21 for information on interface definitions.

Request Map Property

The RequestMap class is the implementation class used by the runtime to determine how to dispatch a particular event. It decides whether the request should be processed or dropped, identifies the correct BPO instance to associate with the request, and ensures that requests for a single object or group of objects are processed in the proper sequence. The Request Map property allows you to define the class that specifies how the Integration Server services events. Two classes are provided by BusinessWare:

- `com.vitria.bpe.runtime.ProcessRequestMapImpl`—This map should be used if the first parameter of the event is a BPID or string object and the port is on a process model component linked to a stateful model.
- `com.vitria.container.map.RequestMapImpl`—This map should be used if the port is on a process model component linked to a stateless model.

For more information on how the map causes the Integration Server to service events against the same BPO in the order they arrive, and for information on customizing the maps, see [Chapter 13, “Process Modeling Techniques.”](#)

Transaction Control Property

The Transaction Control property tells the Integration Server whether or not it should initiate a transaction when it receives an event on this port. This property can have one of three values: Required, RequiresNew, or Mandatory.

- **Required**—the Integration Server is to use the transaction context in the event, if it exists, or create a new transaction context (start a new transaction) if the event does not have a transaction context.
- **RequiresNew**—the Integration Server should always start a new transaction with each event.
- **Mandatory**—the Integration Server should not start new transactions and must use the transaction context in the event.

Required is the default value, which should suffice for most usages. For more information on Transaction Management performed by the Integration Server, see [Chapter 25, “Transaction Management.”](#)

Type Propagation Property

The Type Propagation property can be set to either `True` or `False`. When its value is `True` and the current port is wired to an untyped port, BME will set the Port Types property of the untyped port to match the value of the same property in the current port. When the value is `False`, the BME does not automatically propagate the interface definition, so you must manually coordinate the types on connected ports before they will pass validation. The Type Propagation default property value is `True`.

Dynamic Port Property

The Dynamic Port property can be set to either `True` or `False`. Use this property in conjunction with the Dynamic Map Class property in the Simple Output proxy. See [“Using Proxies with the Dynamic Map Class” on page 6-35](#) for more information.

At model execution time, the proxy determines which connected input port will receive the event passing through this output port. (See [“Using Proxies with the Dynamic Map Class” on page 6-35](#) for more information.)

When this value is set to `False`, the Dynamic Map Class property of the attached Simple Output Proxy is ignored. The Dynamic Port default property value is `False`.

Port Interface Definition

As explained in the following sections, when configuring the Port Types property described in “[Port Types Property](#)” on page 6-18, you can use existing interface definitions or you can import or create your own.

Selecting an Interface for the Port

You can select from a number of interfaces that are available to the project.

1. Click on  in the Port Types field to open the Edit Types dialog box.
2. Click Browse to open the Choose Interface dialog box. The chooser will show you a list of interfaces known to the current project. The list of interfaces will include interfaces in projects that are referenced as dependent projects. This includes the BusinessWare project, which contains many useful interfaces.
3. Select an interface.
4. Click **OK** to close the Choose Interface dialog box.
5. Click **OK** again to close the Edit Types dialog box.

Using Interface Definitions from a File

You can use the following types of interfaces:

- Java
- Java RMI
- EJB
- IDL
- WSDL

The procedures for using Java interfaces, Java RMI interfaces, and EJB interfaces from a file are the same. For example, if the interface file you want is called `myInterface.java` and is located in a directory called `c:\MyProjects\MyInterfaces\com\acme\myProject`, you would use the following steps.

Using a Java, Java RMI, or EJB interface:

1. In the BME, mount the directory `c:\MyProjects\MyInterfaces` using **File > Mount Filesystem...** menu option.
2. Right click on the directory folder in the Explorer and choose **Tools > Use as Type**.

Once the above steps are complete, the interface definitions will become available for selection in the Port Types property.

Tip: If you want to use one of the interfaces in your directory as a type, but not all of them, right-click on a specific interface (instead of the folder) and choose **Tools > Use as Type**.

To make the interface definitions unavailable for selection, right click on the type in the Explorer and choose **Tools > Don't Use as Type** or un-mount the directory by selecting it and using **File > Unmount Filesystem**.

Using IDL interfaces from a file is slightly different. For example, if the interface file you want is called `myInterface.idl` and is located in a directory called `c:\MyProjects\MyInterfaces`, you would use the following steps.

To import an IDL interface:

1. In the BME, mount the directory `c:\MyProjects\MyInterfaces` using **File > Mount Filesystem...**
2. Right-click on the `myInterface.idl` file and choose **Tools > Convert To Type**.

Following a successful conversion, the BME generates corresponding type files which represent your IDL defined interface.

As shown in [Figure 6-8](#), generated type files are visible in the Explorer and will have the type file icon.

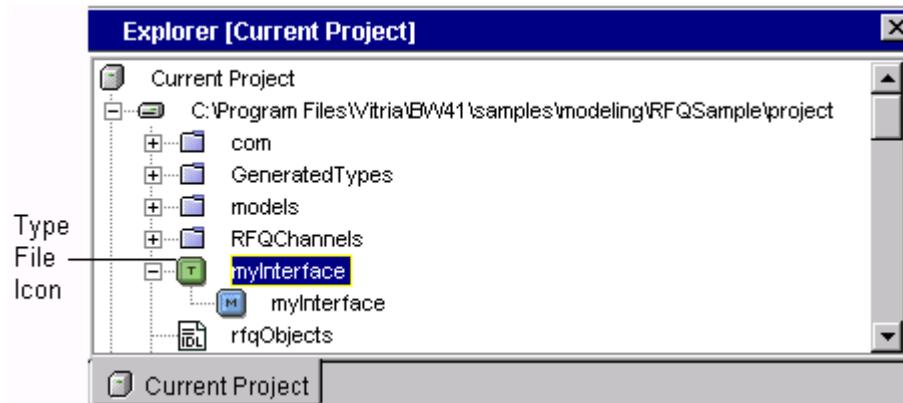


Figure 6-8 Type Icon in the BME Explorer

To undo the type generation, delete the generated type object or unmount the directory by selecting it and using **File > Unmount Filesystem**.

Using a WSDL Interface Definition

Use the Web Service Wizard as explained in [Chapter 9, “Web Services.”](#) Java stub files will be generated by BME following a successful import of WSDL interfaces. To undo the import, delete the generated Java stub files.

Using Interface Definitions from Another Project

Interface definitions from one project can be reused in another project. To do this, you establish a dependency on another project by using **Project > Add Project Dependency**. All interface definitions defined in the added project will become available to the current project and can be used for the Port Types property. When a project is unmounted, its interface definitions become unavailable to the current project. See [Chapter 3, “Projects”](#) for more information on project dependency.

Using Interface Definition from a Java JAR File

Another possible source of interface definitions is Java interface files contained in a standard Java JAR file. The JAR file can be created by BusinessWare components or third-party tools.

To use a JAR file interface definition:

1. In the BME, mount the JAR file by using **File > Mount Filesystem....**
2. In the Explorer, open the folder that corresponds to the mounted file. The first item in the opened folder should typically be a folder labeled `com` (the top level Java package name in the JAR file).
3. Right-click on the `com` folder and select **Tools > Use as Type** to complete the import.

To undo the import, right-click on the type and select **Tools > Don’t Use as Type** or unmount the JAR file by selecting it and using **File > Unmount Filesystem**.

Creating an Interface Definition in the BME

If you like, you can compose interface definitions directly in the BME instead of importing them. For example, to compose a Java interface, RMI interface, or EJB interface with a package named `com.acme.myProject`, you use the following steps.

To create an interface definition:

1. Create a new folder in the Explorer called com. Inside of the com folder create an acme folder, and inside of the acme folder create a myProject folder.
[Figure 6-9](#) shows how the folders should appear in the BME Explorer.

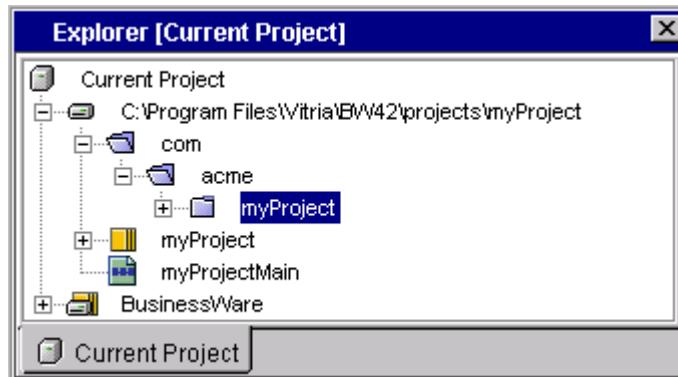


Figure 6-9 Creating Folders for a New Interface Definition

2. Right-click on the com folder and choose **New > All Templates...** and in the New Wizard, select **Types > Interface**. This brings up the wizard (shown in Figure 6-10) to guide you through defining a Java interface.

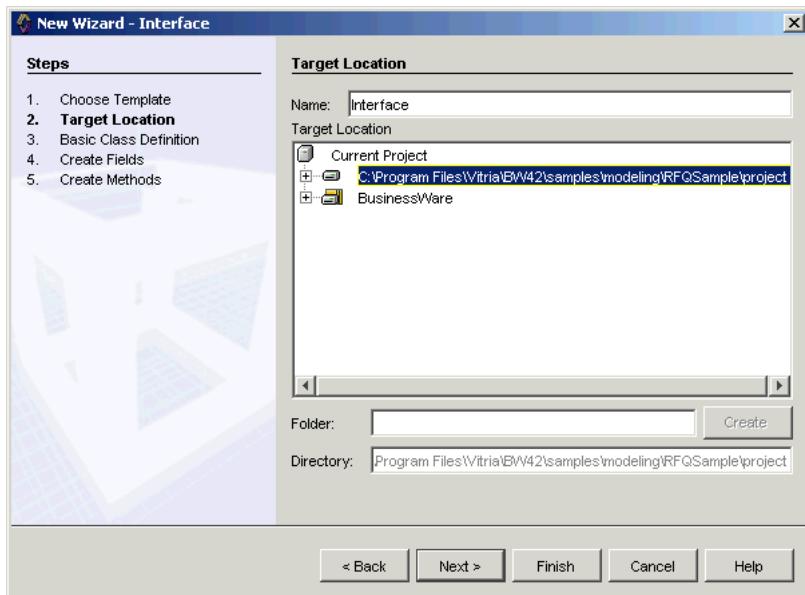


Figure 6-10 Using the New Wizard to Create a New Interface Definition

To create new WSDL interface definitions, use a third-party authoring tool to create an interface in a file, and then reference the interface as described in “[Using Interface Definitions from a File](#)” on page 6-21.

To create new IDL interface definitions, do one of the following:

- Right-click on a folder in the BME Explorer and select **New > All Templates...**. In the New Wizard, select **Other > IDL Source**. When you finish editing the file, use the same technique to import the definition into the project.
- From the File menu, select **New > All Templates...**. In the New Wizard, select **Types > DefinedType** to create the Defined Type object.

IMPORTANT: You should not mix Java and Defined Types within a single definition. Java based types should not inherit from or contain methods that either accept arguments or return values which are based on IDL Defined Types. Similarly, Defined Types should not make references to Java Types.

PORT MODELING CONSTRAINTS

The BME enforces the following constraints for setting up ports in your projects.

Adding and Removing Ports

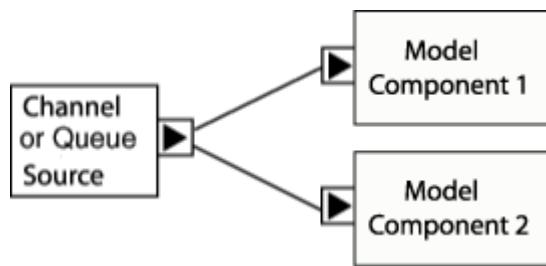
Keep in mind the following when adding and removing ports:

- Process and integration components each have one input port and one output port by default, but you can add or remove any number of ports.
- Process query components have exactly one input port and one output port. Neither port can be removed.
- The number of ports on connectors varies by connector or proxy type. You cannot add or remove ports to or from connectors and proxies.

Wiring Ports

Keep in mind the following when wiring ports:

- Query ports can be added to process components only. They must be wired to the input ports of process query components only and cannot be wired to input ports of other components.
- The ports of two connectors cannot be wired together. An intervening process component separating the two connectors is required.
- BME components are not reentrant, meaning that the same component cannot be visited more than once in one transaction. You need to avoid creating reentrant execution when wiring ports. For example, wiring the reentrant component. This should be avoided. For more information, see [Chapter 25, “Transaction Management.”](#)
- An output port cannot be wired to two or more input ports unless the output port belongs to the Channel or Queue Source connector or the Channel or Queue TargetSource connector. [Figure 6-11](#) shows how you could wire the output port on a Channel or Queue source connector to input ports on two different components.

**Figure 6-11 Wiring an Output Port to Multiple Input Ports**

- In general, one input port may be wired to multiple output ports. If an output port belongs to an input proxy, the input port can only be wired to that one output port. (For more information, see “[Proxies](#)” on page 6-28.)
- When you violate a wiring constraint when connecting ports, BME will display the wire as invalid by changing its color. By default, invalid wires are displayed in red and valid wires are displayed in black. For information on changing the default colors for valid and invalid wires, see “[Displaying Options, Layers, Labels, and Annotations](#)” on page 6-7.

Matching Interfaces

Interface matching refers to the comparison the BME makes between the port type properties of ports wired together.

- A typed output port matches a typed input port if and only if:
 - Neither port is placed on a proxy and the interface of the input port is the same as or an extension of the interface of the output port
 - One of the ports is placed on a proxy and the interfaces of the ports are identical
- A typed port with an IDL interface matches ports with IDL interfaces; it does not match ports with Java interfaces.
- When a typed (input or output) port is wired to a channel or queue connector, its interface must conform to the following restrictions:
 - All the methods (Java) or operations (IDL) in the interface must return void.
 - If the interface is Java, it must be a RMI interface. That is, it cannot be a plain Java interface.
 - All methods (Java) or operations (IDL) in the interface must not throw exceptions.
 - All methods or operations in the interface cannot throw or raise exceptions.

- The parameters in an IDL interface must be “IN” parameters only.
- Untyped ports match anything.

Matching Ports

The BME synchronizes between the ports you add to process components and the ports you use in process models linked to the process components. See “[Port Synchronization](#)” on page 10-7.

- Should you choose to disable automatic port synchronization, you must make sure that the ports of process components are consistent with those in the linked process models.
- The ports of integration components must be matched by ports inside their corresponding nested integration models. To illustrate, consider an integration component with one input port called A and one output port called B. If this component is linked to another integration model (a nested integration model) the port matching requirement means that the nested integration model must also contain an input port called A and an identical output port called B. The types specified for both A ports must match and the types specified for both B ports must match.
- The most effective way to make sure port matching occurs between integration components and their corresponding nested integration models is to use copy and paste operations on ports. An example of the use of a nested integration model can be found in the RFQSample (see the sample in `\installdir\samples\modeling\RFQSample`).

Best Practices

Keep in mind the following best practices when setting up ports:

- When setting the port type property, it is best to use just one interface instead of multiple ones.
- Although an untyped port matches any typed or untyped port, it best not to wire typed and untyped ports together.

PROXIES

Whereas connectors enable interaction between BusinessWare project components and external systems such as Oracle Applications and SAP, proxies enable interaction with external projects or software objects using specific protocols.

By including proxies in your model, you get a visual representation of port connections to objects outside of the current project and how these external objects fit into the complete project.

PROXY TYPES

All proxies act as either input or output proxies:

- *Input proxies* receive data from external objects and forward it to the model component.
- *Output proxies* receive data from BusinessWare components and forward it to external objects.

Table 6-4 Proxies Included with BusinessWare

Icon	Proxy	Function
	Simple Input Proxy	Specifies a binding name (in the directory server) for receiving requests from external CORBA or Java RMI based applications.
	Simple Output Proxy	Specifies a binding name (in the directory server) that will be resolved at runtime to locate external CORBA, RMI, and Java Local objects (typically components in other projects).
	EJB Output Proxy	Specifies information required at runtime to resolve a third-party EJB object in an application server.
	Web Service Input Proxy	Specifies information that allows the connected component to be invoked as a Web service.
	Web Service Output Proxy	Specifies information that allows the connected component to invoke another Web service.

For more information about proxies, see [Chapter 8, “Application Server Integration”](#) and [Chapter 9, “Web Services.”](#)

USING PROXIES TO COMMUNICATE EXTERNALLY

You can communicate with third-party applications or other BusinessWare projects by using Simple Input proxies and Simple Output proxies. The proxies can be configured to use any JNDI-compliant naming service instead of the BusinessWare namespace (such as, a third-party CORBA naming service).

Simple Input Proxies

Third-party applications can invoke BusinessWare components by making RMI or CORBA calls on an input port of a process model through an input proxy. The main steps required to prepare for and implement this are described below.

To prepare a Simple Input Proxy in the BME:

1. Set the input port's interface.
2. Wire a Simple Input proxy to the component's input port.
3. Set the Binding Path Name property of the Simple Input Proxy to the location in the namespace where the proxy should be bound. See [Figure 6-14](#).
4. Optionally, configure the Expert properties of the Simple Input Proxy to use a different name service. See [Figure 6-16](#).

Note: By default, proxies are bound into the BusinessWare namespace.

To invoke a Simple Input Proxy from a standalone client:

1. Generate and compile stubs from the input port's interface (IDL or Java).
2. When the project is running, resolve a reference to the proxy using the Binding Path Name specified in the proxy properties.
3. Invoke the operation by calling the corresponding method. The following example is from the *Order Process Sample*:

```
private static final String CUSTOMER_REQUESTS =
    "/Projects/OrderProcessSample/initial/CUSTOMER_REQUESTS";
customerReq_ = (CustomerRequests) PortLib.getRMIBinding(CUSTOMER_REQUESTS);
```

Simple Output Proxies

A component can synchronously invoke an external application that supports RMI or CORBA operations. This includes third-party applications and other BusinessWare projects. With such an invocation, a component can obtain information from the external process and/or pass information to it.

To prepare a Simple Output Proxy in the BME:

1. Set the output port's interface. If the interface is that of a third-party system, the IDL or Java defining the interface must first be imported into the BusinessWare project.
2. Wire a Simple Output Proxy to the component's output port.
3. Set the Binding property, shown in [Figure 6-15](#), of the Simple Output Proxy to the location in the namespace where the external system can be resolved. The location can be set as one of the following:
 - **Static Binding**—sets a static location
 - **Dynamic: Class**—sets the location based on a Class file
 - **Dynamic XQuery**—sets the location using an XML file or array
 - **Dynamic Java**—sets the location using a piece of Java code

When dynamic binding rules are specified (either in XML or Java), BusinessWare auto-generates the code.

4. Optionally, configure the Expert properties, shown in [Figure 6-16](#), of the Simple Output Proxy to use a different name service.

Note: By default, proxies are resolved in the BusinessWare namespace.

To invoke an external system through a Simple Output Proxy:

1. Bind a reference to the external system into the namespace that the Simple Output Proxy has been configured to use.
2. In the process component's code, get the output port and invoke operations through the port by calling the methods available on the port, just as with ports wired to other components. See [Chapter 6, “Integration Model Basics”](#) for more information.

To communicate with another BusinessWare project using proxies:

1. Share the interface types between the two projects, preferably by placing these types in a shared dependent project.
2. Configure a Simple Output Proxy in the calling project and a Simple Input Proxy in the receiving project with the same port types and the same Binding Path Name property.
3. If the projects are running on instances of BusinessWare that use a different namespace (that is, a different `bw_root`), configure the Expert properties of the Simple Output Proxy to use the directory server name service of the receiving project. In particular, set the Provider URL to the VTPARAMS value of `bw_root` of the receiving projects' BusinessWare instance.

To Add Proxies to an Integration Model

To add a proxy to an integration model component, select a proxy tool from the Proxies tab of the Palette (as shown in [Figure 6-12](#)) and click in the Editor.

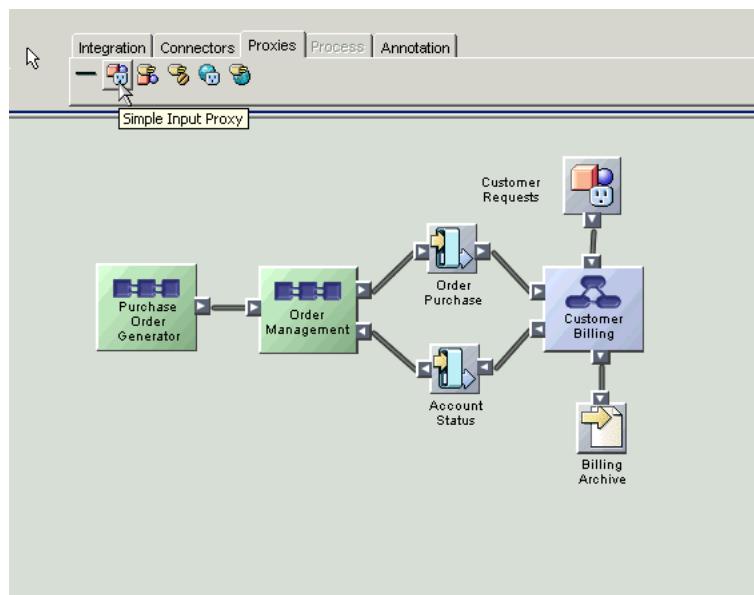


Figure 6-12 Selecting a Proxy Tool from the Proxies Tab of the Palette

Adding a Proxy to an Integration Model

Once you've added a proxy to the Editor, you can wire it to an integration model component by using the Wire tool to connect the output port on the proxy to an input port on the component. [Figure 6-13](#) shows the Wire tool on the Proxies tab of the Palette.

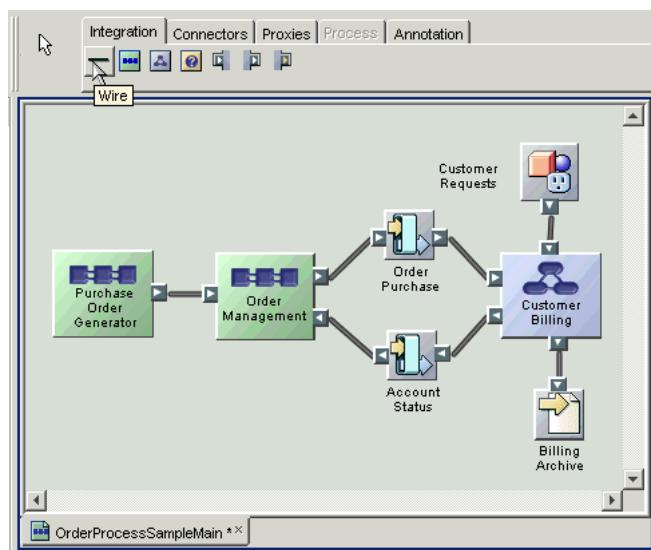


Figure 6-13 Selecting the Wire Tool from the Proxies Tab of the Palette

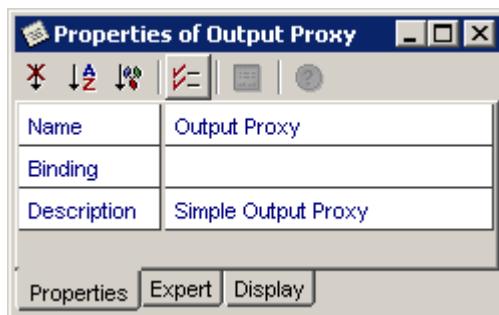
Proxy Properties

Click on a Simple Input Proxy to bring up the proxy's properties. These properties are shown in [Figure 6-14](#).

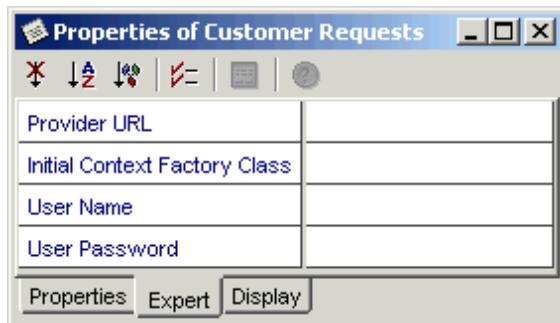


Figure 6-14 Simple Input Proxy Properties

Click on a Simple Output Proxy to bring up the proxy's properties. These properties are shown in [Figure 6-15](#).

**Figure 6-15 Simple Output Proxy Properties**

The Expert properties for a Simple Output Proxy are shown in [Figure 6-16](#).

**Figure 6-16 Expert Properties for Simple Input and Output Proxies**

In [Table 6-5](#), “X” marks the properties that apply to each kind of proxy.

Table 6-5 Simple Input and Simple Output Proxy Properties

Property	Description	Simple Input Proxy	Simple Output Proxy
Name	The name of the input proxy.	X	X
Binding Path Name	The fully qualified name where the object can be found in the namespace.	X	X
Dynamic Map Class	The fully qualified Java class name that dynamically chooses the Binding Path Name.		X
Description	Describes the proxy.	X	X

Table 6-5 Simple Input and Simple Output Proxy Properties

Property	Description	Simple Input Proxy	Simple Output Proxy
Provider URL	URL that specifies a foreign name server. If blank, defaults to <code>bw_root</code> of the Integration Server's VTPARAMS file.	X	X
Initial Context Factory Class	A class used to create JNDI contexts for binding and resolving servants.	X	X
User Name	A security principal. If blank, throws <code>javax.naming.NamingException</code> .	X	X
Password	A security credential. If blank, throws <code>javax.naming.NamingException</code> .	X	X

USING PROXIES WITH THE DYNAMIC MAP CLASS

Process model components communicate with other process model components and connectors through ports. The output port of an invoking process model component can be wired to the input port on the called process model component or connector. This type of connection represents static connectivity—you clearly know the calling relationships of the components at modeling time.

However, there are situations where these calling relationships are known only at runtime and you cannot wire the components statically.

BusinessWare allows process model components to dynamically choose the components they invoke at runtime depending on incoming data. This requires the use of specific properties on the output port and output proxy to which it is wired.

To enable dynamic runtime binding:

1. Wire an output port to a Simple Output proxy. The output port's Dynamic Port property must be set to `True` and the Simple Output proxy's Dynamic Map Class property must be set to a valid dynamic map class.

The Dynamic Map Class enables the process component to dynamically choose the component it invokes at runtime by computing the namespace binding of the input port to invoke. Specifically, the dynamic map class specified in the proxy property must implement the interface `com.vitria.container.map.DynamicPortMap`. This interface contains the following method that must be implemented to return the JNDI name of the input port that should be invoked:

```
String mapToPort (ComponentContext processModelContext,
Object mapInfo);
```

The ComponentContext is the base interface for ProcessModelContext. It provides access to the incoming event and the BPO (for stateful models). The mapInfo object can be anything you choose.

2. Include a class with the project and implement it to look similar to the following:

```
import com.vitria.container.map.DynamicPortMap;
import com.vitria.container.client.ComponentContext;

public class MyDynamicPortMapImpl implements
    DynamicPortMap {

    public MyDynamicPortMapImpl() {
        // requires public default constructor
    }

    public String mapToPort(ComponentContext ctx, Object
        mapInfo) {
        String supplier = (String)mapInfo;
        if (supplier.equals("supplier1")) {
            return <fully qualified name of supplier1 ref in
            namespace>
        } else if (supplier.equals("supplier2")) {
            return <fully qualified name of supplier2 ref in
            namespace>
        }
    }
}
```

The BME generates code for resolving output ports from within action code (See [Chapter 14, “Process Model Code Construction.”](#)). When the Dynamic Port property of an output port is set to True, the generated code for looking up dynamic ports takes an additional mapInfo argument in addition to the output port name. This allows you to pass in custom data to the mapToPort method. You can then invoke the generated lookup code for myOutPort as follows.

```
// assume some data is retrieved from some objects
available
//in the context - e.g. event, BPO, LocalData
String mapInfo = myData.getMyInfo();
// Use this data to pass into the call to the dynamic map
//class to resolve the output port
```

```
MyInterface myPort = getMyOutPort(mapInfo);
// Use the port to make the invocation
myPort.doSomething();
```

For more information on port actions from within process models, see [Chapter 14, “Process Model Code Construction.”](#) and the SimpleOrderSample in `installdir/samples/protocols`.

WORKING WITH INTEGRATION MODELS

This section describes some ways in which you can use integration models.

DESIGNATING THE ROOT INTEGRATION MODEL

The root integration model is the highest-level model in the project. All other models in the project can be traced back to the root model.

When you create a new project, BusinessWare automatically creates an empty root integration model and names it `projectMain`, where `project` is the name of your project. Most often you populate this empty model and keep it as the root model. However, you can designate another integration model as the root at any time during the modeling phase.

To designate the root integration model:

1. Select the **Project** object in the Explorer.
2. Select the **Properties** tab in the Properties window and select the **Root Integration Model** property.
3. Configure the property.

USING NESTED INTEGRATION MODELS

Integration models can be nested within other integration models. Nesting enables:

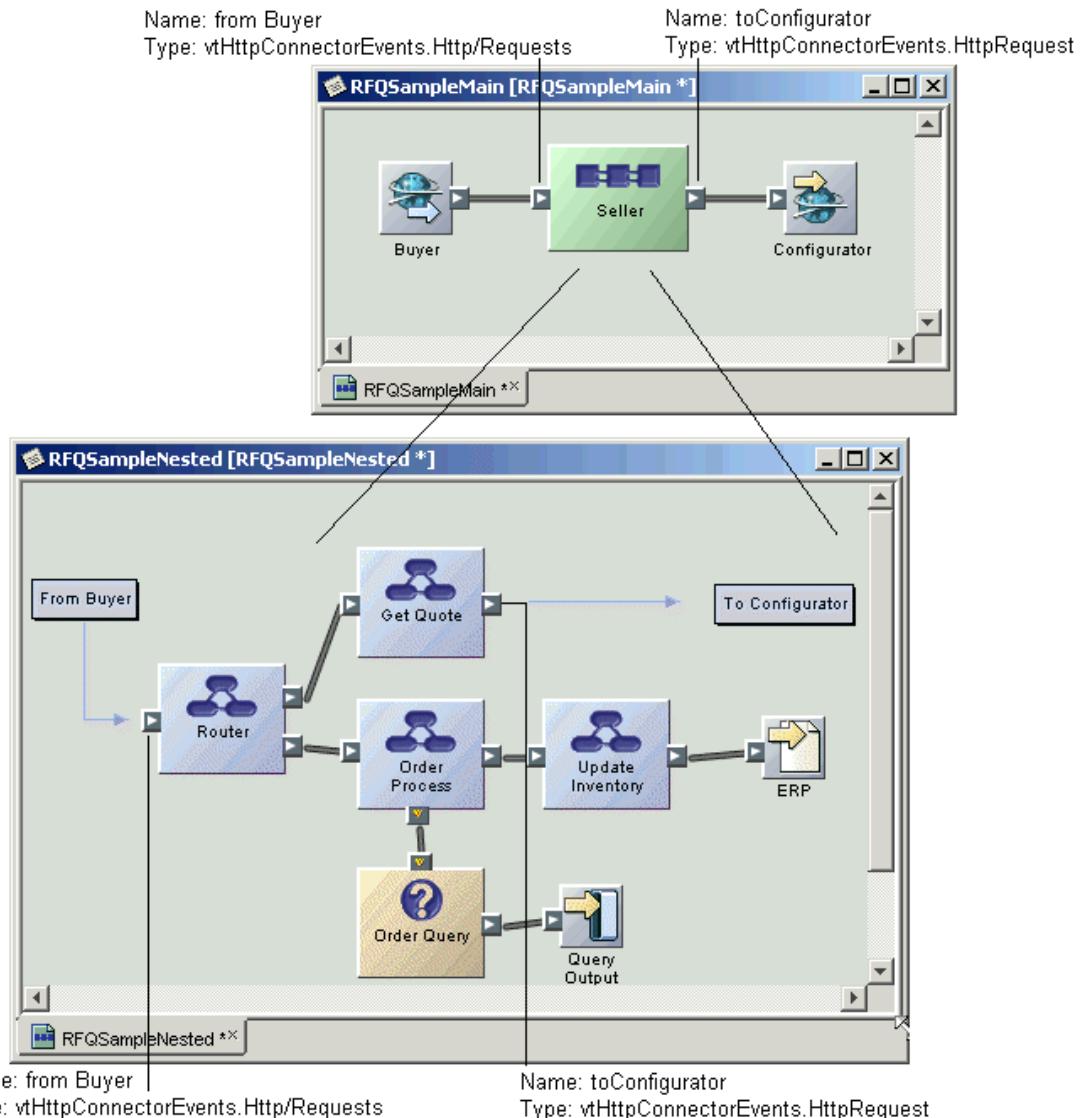
- Hierarchical modeling of complex systems
- Logical encapsulation and reuse of an existing model design
- Building complex components from simpler ones
- Flexibility to invoke any of several nested models based on conditions at runtime

You create a nested model by placing an integration component in one model (the parent) and linking it to a separate integration model (the child). The key thing to remember is that *the port configurations on the integration component must be included in the nested integration model, and the port names and types must match exactly*. Matching ports are required because at runtime:

- Invocations made on the input port of the integration component in the parent model are propagated to the identically named input port within the child integration model.
- Invocations made on the output port of a component within the child integration model are propagated to the identically named output port of the integration component in the parent model.

You can copy and paste ports to easily reproduce the port configurations.

As an example, in [Figure 6-17](#) the ports on the integration component and corresponding ports in the integration model have been labeled.


Figure 6-17 Corresponding Ports on Integration Component and in Integration Model
To create a nested integration model:

1. Create the integration component in the parent model.
2. Configure the ports on the integration component.
3. Create the nested integration model.

4. Configure the ports on components in the nested model.
5. Link the component to the model:
 - a. Select the integration component in the parent model.
 - b. Set the Nesting Specifications property, listing one or more nested models and the triggering events and conditions under which each one should be invoked.

NESTING SPECIFICATION

The decision of which model to invoke is made at runtime, based on the triggering events and conditions you specify. These specifications are defined when you set the Nesting Specification property. Specify the integration models and the conditions under which they are invoked as follows:

1. Select the Integration component.
2. In the Properties window, click the Nesting Specification property and then click .
3. Click **Add**. In the row you created, specify the model that should be invoked. Edit the Trigger and Condition columns to specify when the integration model should be used.
4. Add as many rows as necessary to specify all the integration models and conditions that you require.
5. Click **OK**.

REUSING INTEGRATION MODELS

You can easily copy the integration model you made in one project to other projects by saving the model as a template and then opening it in the other project.

To reuse an integration model:

1. Right-click on the model in the Explorer, and select **Save As Template**.
2. In the Save as Template dialog, select the Models folder and click **OK**.
3. Open the second project and select **File > New....**
4. In the New Wizard, select your integration model template.
5. Set the link property for components in the integration model to point to objects that exist in the current project.

PRINTING INTEGRATION MODELS

You can print a hardcopy of a integration models. Hard copies can be useful as hand-outs during design planning discussions.

To print hardcopy of an integration model:

1. Display the model in the Editor.
2. With the Editor window active, select **File > Print**.

VIEWING LARGE MODELS

If you have large models that you cannot view easily in the Editor, you can use the Overview window to alter the focus of your model. The Overview window is a scaled, mirror image of the model displayed in the Editor. Move the focus line in the Overview window to where you want it positioned in the model, and the Editor changes your view accordingly.

To view the Overview window:

1. From the View menu, select **Overview Window**.

The window will scale the model currently selected in the Editor to size as you drag.

INTEGRATION MODEL BASICS

Working with Integration Models

7

CHANNELS AND QUEUES

Channels and channel connectors are the mechanisms that support the publish-subscribe approach of distributed computing, and queues and queue connectors are the mechanisms that support the point-to-point approach of distributed computing. This chapter describes how and why you might use these approaches in your solution and includes the following topics:

- [Overview](#)
- [Asynchronous Communication](#)
- [Using Channels and Queues](#)
- [Quality of Service](#)
- [Comparing Channels and Queues](#)
- [About Channels](#)
- [Channel Types and Resources](#)
- [Channel Connectors](#)
- [About Queues](#)
- [Queue Resources](#)
- [Queue Connectors](#)
- [Viewing Event Data on Channels and Queues](#)
- [Channel and Queue Connector Exceptions](#)

OVERVIEW

BusinessWare supports communication in a synchronous or asynchronous manner. Synchronous communication is a more tightly coupled form of messaging, where the caller sends a request to the callee and waits for a response. The caller is blocked until the callee finishes processing the request and returns a response. Asynchronous communication is a decoupled form of messaging where the caller makes data available by publishing it to a channel or queue. The data is then forwarded to the callee, who subscribes to the channel or queue. The caller does not wait for a response, is not blocked, and can continue processing other data.

To implement this asynchronous form of distributed computing, you need to incorporate channel or queue resources and their connectors during the design phase. At deployment, the channel or queue resource is partitioned to a BusinessWare Server where the physical channel or queue is instantiated.

ASYNCHRONOUS COMMUNICATION

As shown in [Figure 7-1](#), with the asynchronous form of communication, an application makes information available by placing it on a channel or queue. Other applications subscribe to the channel or queue to receive information. Channels and queues buffer the published data so subscribers need not be available at the moment that the publishers produce it, allowing publishers and subscribers to proceed at their own rate.

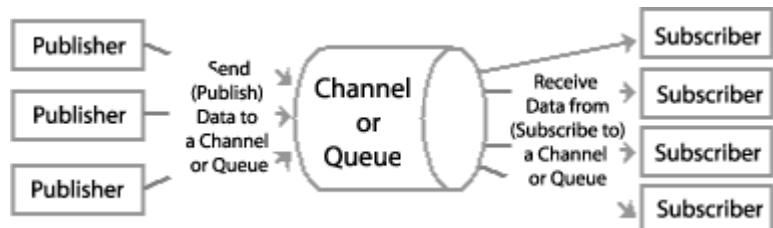


Figure 7-1 Communication Through a Channel or Queue

The publisher and subscriber must agree on the format of the data being exchanged. As shown in [Figure 7-2](#), BusinessWare data is formatted and transmitted as events.

An event is a notable occurrence in a business system. BusinessWare solution developers decide which occurrences are interesting enough to be events.

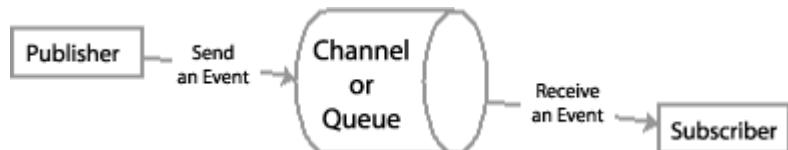


Figure 7-2 Publishing and Receiving Events

An event is a generic data-exchange format that contains a block of bytes. Developers create specialized types of events to represent the important occurrences in their systems. Instead of simply sending and receiving a block of bytes, publishers and subscribers can use these special events to exchange typed data. Exchanging typed data means that publishers and subscribers can send and receive strings, integers, or whatever data types best represent the information they are exchanging. [Figure 7-3](#) shows how the specialized event of sending an order (`OrderEvent`) is exchanged through a channel or queue with a publisher and subscriber.

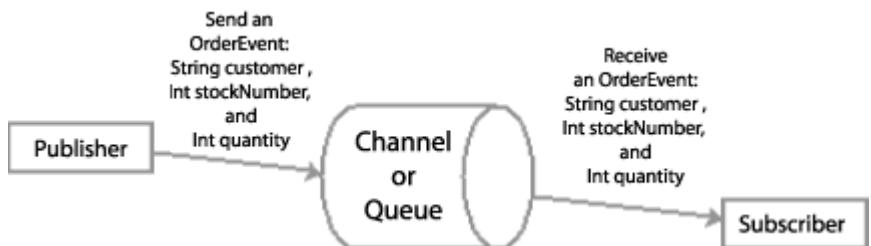


Figure 7-3 Publishing and Receiving Specialized Events

The ability to exchange typed data offers many advantages. It means less work for developers, who would otherwise have to package the data into a block of bytes and then unpack it themselves. It also makes BusinessWare solutions highly portable. Different kinds of platforms may handle raw data (blocks of bytes) in different ways, but because BusinessWare provides typed data, developers do not have to rewrite a publisher or subscriber that has been moved to a different machine.

Asynchronous communication offers the following major advantages:

- **Incremental, dynamic extensibility**—Channels and queues keep track of their publishers and subscribers, but the publishers and subscribers are not aware of each other's identities. Because publishers and subscribers are decoupled, you can transparently add new publishers and subscribers at any time. Similarly, you can provide substitutes for existing publishers and subscribers without impacting the remaining processes.
- **Failure isolation**—A publisher can continue processing even if one or more of its subscribers fails. Because events can be retransmitted, a failed subscriber need not miss any data when it restarts. Subscriber failures do not impact a publisher's performance. Conversely, publisher failures do not impact a subscribers performance.
- **Transparent information exchange**—Publishers and subscribers must agree on the format of event data, but they do not have to have any further knowledge of one another's interfaces or services.

USING CHANNELS AND QUEUES

You use channels and queues in an integration model to allow asynchronous communication and set transactional boundaries.

When two components communicate synchronously, the caller who sends a request is blocked until the callee processes the request and returns a response.

On the other hand, when two components communicate via a channel or queue, the caller (the publisher) sends a request to a channel or queue. The transaction ends after the event is pushed to a channel or queue. A second, separate transaction begins when the callee (the subscriber) receives the request from the channel or queue. It may take minutes, hours, or even days for the callee to process the request, but the original caller is not blocked during this time and can continue processing other data. This is the benefit of asynchronous communication.

You can publish and subscribe to channels and queues programmatically; however, using the prebuilt channel and queue connectors is recommended to achieve the proper transactional behavior in the modeling runtime. With a connector, communicating with channels or queues is a simple matter of configuring the connector properties. In addition to the basic publish-subscribe capability, the channel and queue connectors automatically provide transaction support to ensure that there will be no duplicate or missing events.

[Figure 7-4](#) illustrates how the Order Process sample uses Channel TargetSource Connectors to provide asynchronous communication between the Order Management component and the Customer Billing component. The order processing system sends `AccountPurchaseEvents` to the billing system via one channel, and the billing system sends `AccountStatusEvents` to the order processing system via another channel. This decouples the two systems so they can run at their own rate.

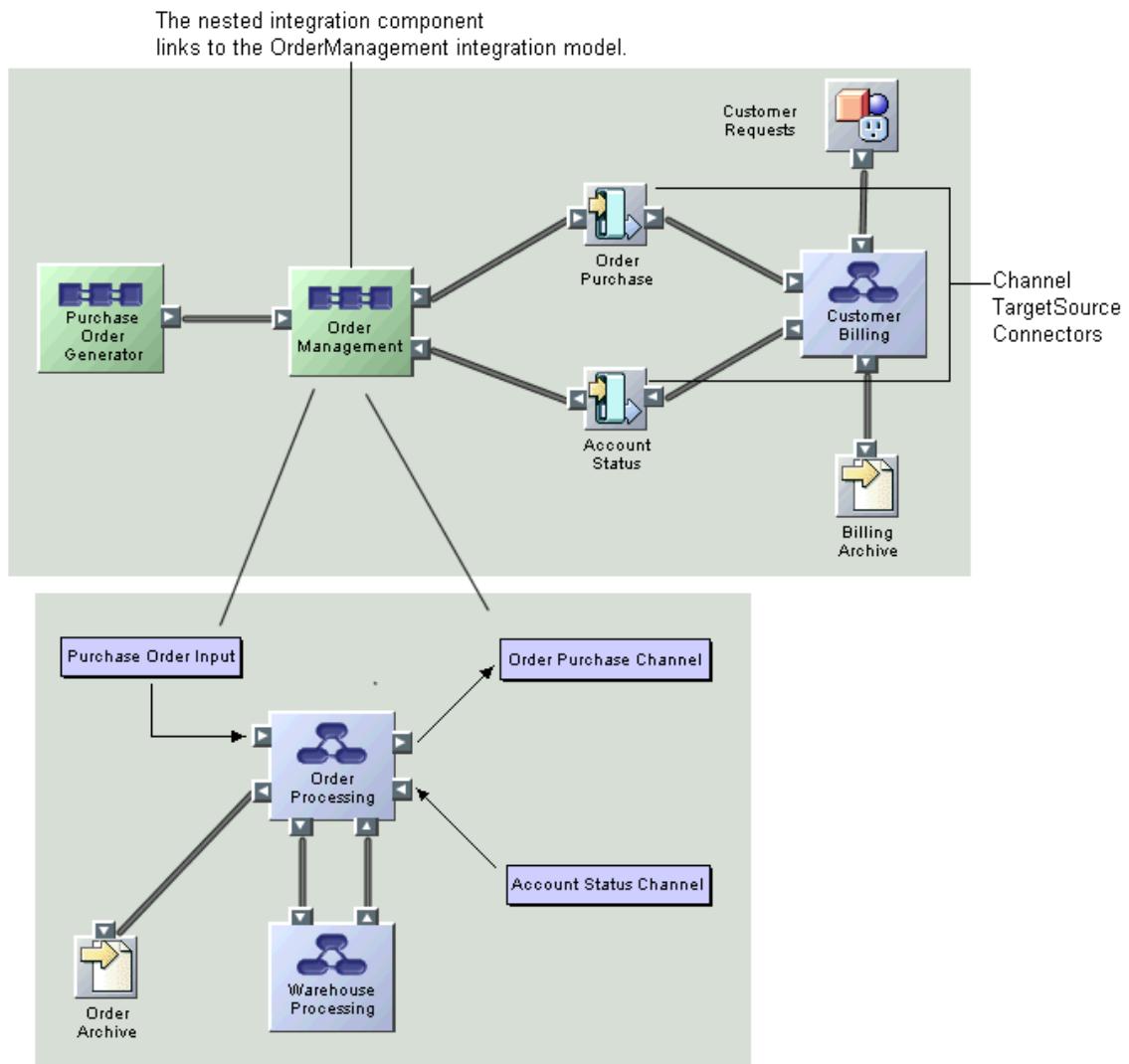


Figure 7-4 Using Channel Connectors for Asynchronous Communication

QUALITY OF SERVICE

All channels and queues have an associated quality of service for message delivery which specifies how events are stored on the server.

Quality of service is set when the channel or queue is created and cannot be changed later. There are two possibilities:

- **Reliable**—events from reliable channels and queues are stored in memory. They can be resent if transient subscriber failures or network failures prevent delivery. In the event of a server failure (for example, a shutdown, machine failure, or power outage), all events in a reliable channel or queue are lost.
- **Guaranteed**—events from guaranteed channels or queues are stored on disk. So, in the event of a server failure, the events are safe and can later be sent to subscribers.

Use reliable channels and queues for applications, such as delivering news, where message loss may be inconvenient but will not seriously impact your business.

Use guaranteed channels and queues for business-critical information, where loss of information could affect the operations or revenues of your business.

With both reliable and guaranteed channels and queues, events are purged when they exceed the age limit or capacity limit you have set. See [Table 7-1](#) for information about setting age and capacity limits.

COMPARING CHANNELS AND QUEUES

There are two common styles of asynchronous communication: publish-subscribe and point-to-point. With the publish-subscribe style, each message published to a destination (either a channel or queue) is received by all subscribers to that destination. With the point-to-point style, each message published to a destination is received by (at most) one subscriber to that destination. In BusinessWare, channels implement the publish-subscribe style of communication while queues implement the point-to-point style.

CHANNELS AND PUBLISH-SUBSCRIBE COMMUNICATION

The publish-subscribe style is appropriate for distributing each unit of information to multiple subscribers. As shown in the example in [Figure 7-5](#), a financial application might have a channel for each stock symbol that receives events representing quotes (such as 5.2 and 5.4) for that stock. An application interested in quotes for a set of stocks (for example, those owned by a specific customer) could subscribe to the corresponding channels. Each quote would be sent to all subscribers interested in that stock.

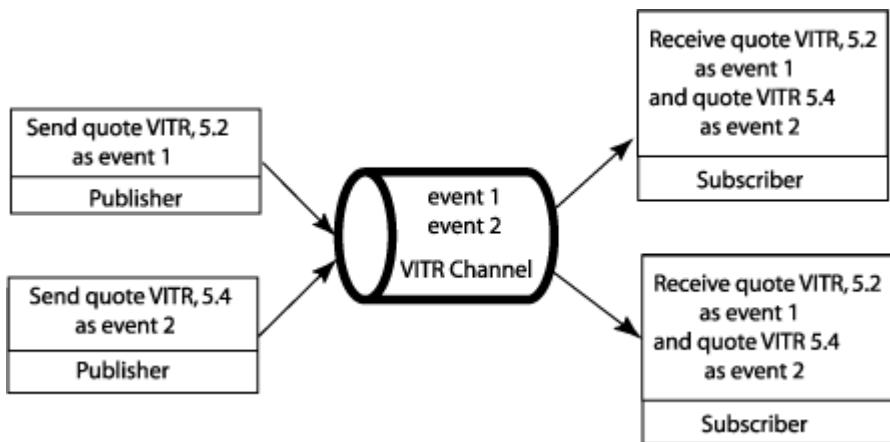
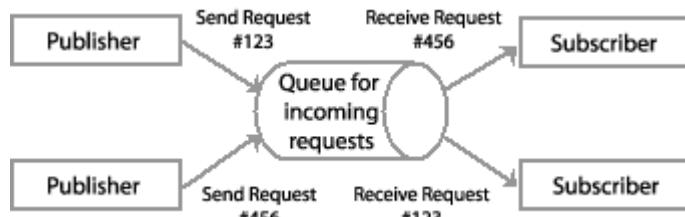


Figure 7-5 Publish-Subscribe Communication

QUEUES AND POINT-TO-POINT COMMUNICATION

With the point-to-point style, each message published to a destination is received by, at most, one subscriber to that destination and the message is removed after it is received. In BusinessWare, queues implement the point-to-point style of communication.

The point-to-point style is appropriate for distributing each unit of information to exactly one of several subscribers. As shown in the example in [Figure 7-6](#), an order processing system might have a queue that receives incoming requests. Several servers might subscribe to this queue to retrieve work, and each request is processed by exactly one server. The queue allows load balancing across the servers.

**Figure 7-6 Point-to-Point Communication**

DESTINATION CONSIDERATIONS

When choosing the type of destination for asynchronous communication, you should consider the following.

- **How many subscribers will the destination have?**— If the destination (a channel or queue) has multiple subscribers, you must choose which style is appropriate (should each event go to one or all subscribers?). This dictates your choice of destination. For publish-subscribe, you must use a channel. For point-to-point, you must use a queue. If a destination has only one subscriber (the most common case), then both styles are equivalent and either a channel or queue may be used.
- **How quickly will I need to retrieve events?**— The main advantage of using a channel is performance. Retrieving events from a channel is non-destructive and therefore faster than retrieving events from a queue. The rate at which events can be retrieved from a queue is roughly 20-30 percent less than the rate events can be retrieved from a channel.

Another advantage is lower latency. A channel will push a newly arrived event to its subscribers immediately, whereas queue subscribers must periodically poll the queue to check for new events, introducing some latency.

- **What will I need to know to configure channels and queues?**— The main advantage of using a queue is that it is simple to guarantee that events are not purged until they are received and processed by the subscriber; this is the default behavior.

Channels are more difficult to configure since you must set attributes (capacity and age limit) that determine when events are purged. The maximum subscriber backlog must be estimated based on the average event rate through the channel and maximum subscriber down time. Since you do not want to purge any events before they are processed, these estimates are often quite conservative, so most events are kept in the channel long after they are processed, causing the server to use more resources (memory or disk space) than necessary.

- **Will I be using a BusinessWare 3.x server?**—If you are using a BusinessWare 3.x server, you can use channels for exchanging data but not queues since BusinessWare 3.x servers do not support queues.

ABOUT CHANNELS

In BusinessWare, a channel is an object that manages a flow of events on a first-in-first-out (FIFO) basis. Channels accept events from publishers, deliver the events to subscribers, and maintain the events for a specified period of time. Thus, if a transmission fails (due to network outages or a subscriber being offline, for example), the data can be resent later. In addition, channels can generate and maintain performance statistics about their operation. Like other BusinessWare resources, channels use permissions to control access to the channel and its contents.

A publisher specifies the names of the channels to which it wants to send data. A subscriber can specify, by name, the channel or channels from which it wants to receive data. Channels can receive data from multiple publishers and send data to multiple subscribers.

BusinessWare is typical of publish-subscribe systems in that its channels immediately send newly published data to its subscribers. The immediate forwarding of data from a channel to its subscribers is called *near real-time* communication.

CHANNEL PERMISSIONS

The following security permissions can be set on channels to specify authorization and access:

- **Publish Permission**—specifies the list of roles allowed to publish events to the channel. Leaving this property unchecked specifies that no authorization checks need to be completed.
- **Subscribe Permission**—specifies the list of roles allowed to subscribe to the channel. Leaving this property unchecked specifies that no authorization checks need to be completed.
- **Admin Permission**—specifies the list of roles allowed to administer the channel, such as changing channel attributes post-deployment, purging, etc. Leaving this property unchecked specifies that no authorization checks need to be completed.

The default values match the value of the project object's default permission property.

To set channel permissions:

1. Open the **Properties Window** for the BusinessWare channel whose security permissions you want to set.
2. In the **Properties Window**, set the appropriate Permissions property.

For more information on security permissions and security in BusinessWare, see the *BusinessWare Security Guide*.

CHANNEL TYPES AND RESOURCES

BusinessWare supports four types of channels:

- Basic channels
- BW3x channel shortcuts
- Composite channels
- Replica channels

Each of these channel types has resources which are logical representations of channels that are mapped to BusinessWare Servers during deployment. Once you create a channel resource, you change the way it handles events by changing its properties.

BASIC CHANNEL

A basic channel is a general purpose channel suitable for most applications. It is the most commonly used type of channel.

BW3X CHANNEL SHORTCUT

The BW3x Channel Shortcut provides interoperability with previous versions of BusinessWare. The shortcut channel is needed because BusinessWare 3.x does not use the directory server to manage the namespace; so the 3.x channel names are not immediately available to BusinessWare 4.x.

Using this shortcut channel, you can either subscribe to events from channels or publish events to channels that reside on previous versions of BusinessWare.

IMPORTANT: The BW3x Channel Shortcut is limited to BusinessWare versions 3.0.2 and later. Interoperability is not supported for versions of BusinessWare older than 3.0.2.

To configure the BW3x Channel Shortcut, you must specify the machine name and port number for the BusinessWare Server where the 3.x channel resides, and the pathname for the channel. The transport protocol (read only) is TCP. Interoperability with BusinessWare 3.x using SSL is not currently supported.

To share types between BusinessWare 3.x and 4.x, you must use the 3.x types in your 4.x project. For more information, see “[Ports](#)” on page 6-15.

COMPOSITE CHANNELS

Composite channels consolidate information from other channels. They are used to gather information from multiple channels that are commonly accessed together. An application that needs information from multiple “source” channels can simply subscribe to the composite channel, rather than subscribing to all of the individual channels. Similarly, if you create a new channel and add it as a source for the composite channel, all of the composite channel’s subscribers automatically get the information from the new channel; you don’t have to add each one as a subscriber to the new channel.

A composite channel can have basic, replica, BW3x channel shortcut, or other composite channels as its sources. Sources for composite channels can be defined in local or library projects. Composite channels cannot take events directly from publishers. When a composite channel receives events from a source channel, it assigns them new event identifiers. It never reorders events from a particular source channel, although it does interleave events from its source channels.

IMPORTANT: If you specify a source channel that is defined in a library project, and the library project is subsequently redeployed to a different BusinessWare Server, the composite channel will stop functioning.

Although composite channels derive their data from other channels, they exist independently. Even if all its source channels are destroyed or become unavailable, the composite channel still exists.

Example: Suppose that each piece of equipment in a manufacturing plant publishes status information on its own dedicated channel. An application that monitors the equipment for a given area of the plant gets its information from a composite channel that consolidates information from all of the dedicated channels for that area. Similarly, composite channels could be created to gather status information for all the equipment in a particular building or in the entire plant. When new equipment is added, along with new dedicated channels, the definitions of the relevant composite channels are updated to include the new source channels.

Replica Channels

A replica channel is a duplicate of another channel. It copies the information from its source channel and redistributes that information to its subscribers. Replica channels are used to effectively segment consumer populations, disseminate information to a large clientele efficiently, and optimize wide-area network (WAN) information flows. For example, you might use replica channels running in Japan, Australia, and England to disseminate information to subscribers in those countries, rather than have each subscriber subscribe directly to the source channel running in the United States. This way, only one copy of each event is sent across the WAN. In addition, using replica channels increases fault-tolerance across unreliable network links.

The source for a replica channel can be a basic channel, a composite channel, a BW3x channel shortcut, or another replica channel. Replica channels cannot take events directly from a publisher. *Once you specify the source for a replica channel, you cannot change it*, but you can create more than one replica channel from the same source.

When a replica channel gets events from its source channel, it maintains the original event identifiers and the sequence of events. Once an event is in a replica channel, it can be kept or deleted independent of the event's existence in the source channel. Therefore, the source channel can delete the event without affecting the replica. Similarly, the replica channel can delete the event without affecting the source.

A replica channel depends on the source channel. If its source becomes unavailable or is destroyed, the events currently on the replica channel remain, but no new events are received. If the source channel is recreated, all the old events on the replica channel are purged.

CHANNEL RESOURCES

When you model business solutions in BusinessWare, you can choose to create “channel resources,” which are logical representations of channels. During deployment, you map these channel resources to BusinessWare Servers where the physical channels are instantiated. Therefore, you can develop the model without being concerned about configuration details such as machine names.

When you create a channel resource, you set several properties that specify how the physical channel will operate at runtime.

To add a channel resource:

1. Use either of these methods to add a channel resource to your project:
 - In the BME Explorer, right-click on the directory where you want to place the new resource; then select **New > All Templates... > Resources > Channel** and choose the type of channel you want.
 - Select **File > New...** to display the New Wizard; then expand the **Resources** node and choose the channel type you want.
2. In the Properties Window, set the properties for the new channel resource as described in [Table 7-1](#).

SETTING PROPERTIES FOR CHANNEL RESOURCES

To set properties for a channel resource, right-click the resource in the **Explorer** as described in [Table 7-1](#).

Table 7-1 Channel Properties Set in the BME

Property	Description
PROPERTY TAB	
Properties for Basic, Replica, and Composite Channels	
Name	The name of the Channel Resource. This will be partitioned to a real channel with this name during deployment.
Quality of Service	Determines how reliable message delivery is. There are two options: <ul style="list-style-type: none"> • Guaranteed—Events are logged to disk and survive any crashes or reboots of the BusinessWare Server. • Reliable—Events are stored in memory and may be lost if the BusinessWare Server goes down.
Capacity	Maximum number of events the channel can hold. When a channel reaches its capacity, BusinessWare removes the oldest events to make room for new events. A value of -1 indicates no limit.
Age Limit	Time (in seconds) that an event remains in the channel before being removed. A value of -1 indicates no age limit.
Priority	Indicates the importance of the channel's events. BusinessWare gives more time to the delivery of the events on higher priority channels under heavy loads. Possible values: <ul style="list-style-type: none"> • High • Medium • Low

CHANNELS AND QUEUES

Channel Types and Resources

Table 7-1 Channel Properties Set in the BME (Continued)

Property	Description
Publish Permission	Specifies the list of roles allowed to publish events to the channel. Leaving this property unchecked specifies that no authorization checks need to be completed.
Subscribe Permission	Specifies the list of roles allowed to subscribe to the channel. Leaving this property unchecked specifies that no authorization checks need to be completed.
Admin Permission	Specifies the list of roles allowed to administer the channel, such as changing channel attributes post-deployment, purging, etc. Leaving this property unchecked specifies that no authorization checks need to be completed. Note: You must have been assigned the role of BusinessWare System Administrator in order to change this property.
Additional Property for Composite Channels	
Channel Source List	List of source channels from which a composite channel gets its data. Source channels may be basic channels, composite channels, replica channels, or BW3x shortcut channels.
Additional Property for Replica Channels	
Name	The name of the Channel Resource.
Channel Source	Name of the source channel which is duplicated by this replica channel. The source channel may be a basic channel, composite channel, a BW3x shortcut channel, or another replica channel. Once you set the source channel, you cannot change it.
Properties for BW3X Channel Shortcut only	
Server Host Name	Host name of the BusinessWare Server that contains the channel.
Server Port	Port number for the BusinessWare Server that contains the 3.x channel.
Protocol (Read Only)	Transport protocol (TCP).
Channel Name	Fully qualified pathname of the 3.x channel on the BusinessWare 3.x server.
EXPERT TAB	
Properties for Basic, Composite, and Replica Channels	
Wait Limit	Time (in seconds) to wait for a response before dropping a subscriber's connection.
Batch Size	The maximum number of events that are sent to a subscriber in each batch.
Channel Prune Time	Time (in seconds) that a channel can remain idle before it is deleted. A value of 0 indicates there is no maximum time limit.
Active Sub Prune Time	Time (in seconds) that a channel can remain idle before all its subscribers are asked to acknowledge receiving events.
Inactive Sub Prune Time	Polling interval (in seconds) for determining whether subscribers to an inactive channel are still active. If a channel is idle, the Communicator Server polls the channel's subscribers at this interval to determine their status.
MULTICAST TAB	

Table 7-1 Channel Properties Set in the BME (Continued)

Property	Description
From the Multicast tab, set these properties for Basic, Composite, and Replica channels	
Enable Multicast	Enable Multicast (True or False)
Port	Port number for the multicast port.
IP Multicast Group Address	Leave blank to let the transport code pick an address.
Service ID	The service ID of the server. Set to 0 to let the server pick an ID.
Protocol (read-only)	Displays the name of the transport protocol being used.
REMOTE TAB	
Provider URL	Full directory server URI of the remote channel.
User Name	The security Principal used for accessing the foreign BusinessWare's repository LDAP.
User Password	The password used for accessing the foreign BusinessWares repository LDAP.
BW User Name	The security Principal used for authentication and authorization in the foreign BusinessWare.
BW User Password	The password used for authentication and authorization in the foreign BusinessWare.

CHANNEL CONNECTORS

Channel connectors provide the link between channels and other components in a BusinessWare solution. Three channel connectors are available:

- **Channel Target Connector**—publishes events to a channel.
- **Channel Source Connector**—subscribes to a channel to receive events.
- **Channel TargetSource Connector**—publishes events to a channel and subscribes to the same channel for events.

All the channel connectors have the following characteristics:

- Channel connectors have a predefined number of ports through which they accept data from or send data to other components in the integration model. You cannot add more ports.
- Channel connectors support a full and open exchange of events with the channel. So target connectors publish all the events they receive to their designated channel and source connectors accept all the events sent by their designated channel.

- Channel connectors set transactional boundaries and support two-phase commits. See [Chapter 25, “Transaction Management”](#) for more information.

ADDING A CHANNEL CONNECTOR

When you add a channel connector to an integration model, you must link it to the channel resource that the connector will publish to and/or subscribe to.

To add a channel connector:

1. Select the type of connector you want from the Connector tab of the Palette:
 - Channel Source Connector
 - Channel Target Connector
 - Channel TargetSource Connector
2. Click in the Editor where you want the connector to appear.
3. Wire the port on the connector to the port on the component that it receives data from or sends data to.
4. If you have not already done so, create the channel resource that this component will publish to or subscribe to.
See [“Channel Resources” on page 7-12](#).
5. Link the channel connector window.
6. Set the other connector properties.
See [Table 7-2](#) for a list of properties.

SETTING PROPERTIES FOR CHANNEL CONNECTORS

[Table 7-2](#) describes the properties associated with channel connectors.

Table 7-2 Channel Connector Properties

Property	Description	Channel Target	Channel Source	Channel TargetSource
Name	Name of the connector.	X	X	X
Batch Commit Number	Number of events to process in each transaction.		X	X
Transactionality	Specifies the transactional nature of the connector. Channel source connectors are two-phase (read-only). Channel target connectors are local or two-phase.	X	X	

Table 7-2 Channel Connector Properties (Continued)

Property	Description	Channel Target	Channel Source	Channel TargetSource
Exception Handler Class	Fully qualified Java class name of the <code>ExceptionHandler</code> implementation class that is used to process exceptions.	X	X	X
Max Concurrency	Maximum number of threads used by the source flow to process events. Note: If you increase a connector's concurrency, and the connector is wired to a stateless model, you should use the <code>RequestMapImpl</code> to enable concurrency on the model. See “RequestMapImpl and Stateless Models” on page 13-25 for more information.		X	X
Channel Resource	Name of the channel that a Channel Target Connector publishes to. Name of the channel that a Channel Source Connector subscribes to. Name of the channel that a Channel TargetSource Connector both publishes to and subscribes to.	X	X	X
Permitted Roles	List of roles authorized to deliver events to the channel source connector.		X	
Source Transactionality	Specifies the transactional nature of the connector. (read-only)			X
Target Transactionality	Specifies the transactional nature of the connector. Can be local or two-phase.			X
Verification Batch Size	Number of events to publish before verifying that the Communicator received them. Note: Used only if the Target channel is a BusinessWare 3.x Shortcut Channel			

CONFIGURING THE CHANNEL CONNECTOR FOR DYNAMIC CONNECTIVITY

You can configure a Channel Target Connector to connect to different destinations dynamically based on information within events.

1. In the process model, select the **output port** and set the **Dynamic Port** property to **True**.
2. Select the process model and in the Action Builder, enter the following code:

```
com.vitria.connectors.channel.DynamicChannelTargetInfo info =  
    com.vitria.connectors.channel.ChannelConnectorLib.createDynamicTargetInfo();  
info.setChannelName( "/Servers/bserv1/Channels/TargetChannel2" );
```

ABOUT QUEUES

Queues are like channels in many ways. They buffer a flow of events on a first-in first-out (FIFO) basis, receiving events from publishers and sending them to subscribers. The key differences are:

- Each event in a queue is sent to (at most) one subscriber.
- Events can be retrieved from a queue transactionally. When the transaction commits, the event is typically purged from the queue as part of the transaction.
- Queue subscribers must poll the queue to check for new events (the Queue Source Connector is a pollable connector, while the Channel Source Connector is a notifiable connector).
- There is only one type of queue.

Immediately purging events as soon as they are consumed by a subscriber may not always give you the results you want. For example, in some cases an administrator may want to replay events that were already sent in order to recover from a failure downstream.

A queue can be configured to hold events for a given amount of time after they are consumed by setting the Replay Limit property. This property specifies the number of seconds consumed events should be held before they are purged.

By default, the Replay Limit of a queue is zero (events are purged once consumed). If the Replay Limit of a queue is greater than zero, the command `vtadminreplay` can be used to replay a range of events that have been consumed but not yet purged.

QUEUE RESOURCES

When you model business solutions in BusinessWare, you can choose to create “queue resources,” which are logical representations of queues. During deployment, you map these queue resources to BusinessWare Servers where the physical queues are instantiated. Therefore, you can develop the model without being concerned about configuration details such as machine names.

When you create a queue resource, you set several properties that specify how the physical queue will operate at runtime.

To add a queue resource:

1. Use either of these methods to add a queue resource to your project:
 - In the BME Explorer, right-click on the directory where you want to place the new resource; then select **New > All Templates... > Resources > Queue**.
 - Select **File > New...** to display the New Wizard; then expand the **Resources** node and choose the desired type of channel.
2. In the Properties window, set the properties for the new queue resource as described in [Table 7-3](#).

SETTING PROPERTIES FOR QUEUE RESOURCES

To set properties for a queue resource, right-click the resource in the BME Explorer as described in [Table 7-3](#).

Table 7-3 Queue Properties Set in the BME

Property	Description
PROPERTIES TAB	
Name	The name of the Queue Resource. This will be partitioned to a real channel with this name during deployment.
Quality of Service	Determines how reliable message delivery is. There are two options: <ul style="list-style-type: none"> • Guaranteed—Events are logged to disk and survive any crashes or reboots of the BusinessWare Server. • Reliable—Events are stored in memory and may be lost if the BusinessWare Server goes down.
Capacity	Maximum number of events the queue can hold. When a queue reaches its capacity, BusinessWare removes the oldest events to make room for new events. If the queue contains consumed events, these will be purged first. A value of -1 indicates no limit.

CHANNELS AND QUEUES

Queue Connectors

Table 7-3 Queue Properties Set in the BME (Continued)

Property	Description
Age Limit	Time (in seconds) that an event remains in the queue before being removed. A value of -1 indicates no age limit.
Priority	Indicates the importance of the queue's events. BusinessWare gives more time to the delivery of the events on higher priority queues under heavy loads. Possible values: <ul style="list-style-type: none">• High• Medium• Low
Publish Permission	Specifies the list of roles allowed to publish events to the queue. Leaving this property unchecked specifies that no authorization checks need to be completed.
Subscribe Permission	Specifies the list of roles allowed to subscribe to the queue. Leaving this property unchecked specifies that no authorization checks need to be completed.
Admin Permission	Specifies the list of roles allowed to administer the queue, such as changing queue attributes post deployment, purging, etc. Leaving this property unchecked specifies that no authorization checks need to be completed. Note: You must have been assigned the role of BusinessWare System Administrator in order to change this property.
EXPERT TAB	
Wait Limit	Time (in seconds) to wait for a response before dropping a subscriber's connection.
Replay Limit	The number of seconds that consumed events are kept in the queue before being purged.
Inactive Sub Prune Time	Polling interval (in seconds) for determining whether subscribers to an inactive queue are still active. If a queue is idle, the Communicator Server polls the queue's subscribers at this interval to determine their status.
REMOTE TAB	
Provider URL	Full directory server URI of the remote channel.
User Name	The security Principal used for accessing the foreign BusinessWare's repository LDAP.
User Password	The password used for accessing the foreign BusinessWare's repository LDAP.
BW User Name	The security Principal used for authentication and authorization in the foreign BusinessWare.
BW User Password	The password used for authentication and authorization in the foreign BusinessWare.

QUEUE CONNECTORS

Queue connectors provide the link between queues and other components in a BusinessWare solution. Three queue connectors are available:

- **QueueTarget Connector**—publishes events to a queue

- **Queue Source Connector**—subscribes to a queue to receive events
- **Queue TargetSource Connector**—publishes events to a queue and subscribes to the same queue for events

All queue connectors have the following characteristics:

- Queue connectors have a predefined number of ports through which they accept data from or send data to other components in the integration model. You cannot add more ports.
- Queue connectors support a full and open exchange of events with the queue. That is, target connectors publish all the events they receive to their designated queue; source connectors accept all the events sent by their designated queue.
- Queue connectors set transactional boundaries and support two-phase commits. See [Chapter 25, “Transaction Management”](#) for more information.

Adding a Queue Connector

When you add a queue connector to an integration model, you must link it to the queue resource that the connector will publish to and/or subscribe to.

To add a queue connector:

1. Select the type of connector you want from the Connector tab of the Palette:
 -  Queue Source Connector
 -  Queue Target Connector
 -  Queue TargetSource Connector
2. Click in the Editor where you want the connector to appear.
3. Wire the port on the connector to the port on the component that it receives data from or sends data to.
4. If you have not already done so, create the queue resource that this component will publish to or subscribe to.

See [“Queue Resources” on page 7-19](#).

5. Link the queue connector to a queue by setting the Queue Resource properties in the Properties window.
6. Set the other connector properties.

See [Table 7-4](#) for a list of properties.

SETTING PROPERTIES FOR QUEUE CONNECTORS

[Table 7-4](#) describes the properties associated with queue connectors.

Table 7-4 Properties of Queue Connectors

Property	Description	Queue Target	Queue Source	Queue TargetSource
Properties Tab				
Name	Name of the connector.	X	X	X
Transactionality	Specifies the transactional nature of the connector. Queue source connectors are two-phase (read-only). Queue target connectors are local or two-phase.	X	X	
Batch Commit Number	Number of events to process in each transaction.		X	X
Exception Handler Class	Fully qualified Java class name of the <code>ExceptionHandler</code> implementation class that is used to process exceptions.	X	X	X
Max Concurrency	Maximum number of threads used by the source flow to process events. Note: If you increase a connector's concurrency, and the connector is wired to a stateless model, you should use the <code>RequestMapImpl</code> to enable concurrency on the model. See “RequestMapImpl and Stateless Models” on page 13-25 for more information.		X	X
Queue Resource	Name of the queue that a Queue Target Connector publishes to. Name of the queue that a Queue Source Connector subscribes to. Name of the queue that a Queue TargetSource Connector both publishes to and subscribes to.	X	X	X

CONFIGURING THE QUEUE CONNECTOR FOR DYNAMIC CONNECTIVITY

You can configure a Queue Target Connector to connect to different destinations dynamically based on information within events.

1. In the process model, select the **output port** and set the **Dynamic Port** property to **True**.
2. Select the process model and in the Action Builder, enter the following code:

```
com.vitria.connectors.channel.DynamicQueueTargetInfo info =  
    com.vitria.connectors.channel.QueueConnectorLib.createDyn  
    amicTargetInfo();  
info.setQueueName( "/Servers/bserv1/Queues/TargetQueue2" );
```

QUEUE PERMISSIONS

The following security permissions can be set on queues to specify authorization and access:

- **Publish Permission**—specifies the list of roles allowed to publish events to the queue. Leaving this property unchecked specifies that no authorization checks need to be completed.
- **Subscribe Permission**—specifies the list of roles allowed to subscribe to the queue. Leaving this property unchecked specifies that no authorization checks need to be completed.
- **Admin Permission**—specifies the list of roles allowed to administer the queue, such as changing queue attributes post deployment, purging, etc. Leaving this property unchecked specifies that no authorization checks need to be completed.

The default values match the value of the project object's default permission.

To set queue permissions in the BME:

1. Open the **Properties Window** for the BusinessWare queue whose security permissions you want to set.
2. In the **Properties Window**, set the appropriate Permissions property.

For more information on security permissions and security in BusinessWare, see the *BusinessWare Security Guide*.

VIEWING EVENT DATA ON CHANNELS AND QUEUES

After the project is deployed and running, you can use the Channel/Queue Inspector to view the event data on a given channel or queue. To launch the Channel/Queue Inspector, use either of these:

- Select **View > Channel/Queue Inspector**.
- Right-click a channel connector in the integration model and select **Tools > Channel/Queue Inspector**.

For instructions on configuring and using the Inspector, see [Chapter 27, “Debugging and Animation.”](#)

CHANNEL AND QUEUE CONNECTOR EXCEPTIONS

Exception handlers provide a systematic and customizable way to respond to exceptions. Connector exceptions are handled by the following classes which provide the necessary interfaces and methods:

- `com.vitria.container.ExceptionHandlerImpl` class
- `com.vitria.modeling.runtime.PortExceptionHandlerImpl` class

The following sections describe the exceptions for channels and queues.

CHANNEL CONNECTOR EXCEPTIONS

There are three types of Channel Connector exceptions:

- **ChannelConnectorConnectionException**—This exception occurs when a Channel Connector is unable to connect to the specified channel. Reasons for this exception include the following:
 - The channel server is down or unreachable.
 - The channel does not exist.
- **ChannelConnectorPushException**—This exception occurs when a Channel Connector is unable to send an event to a channel. Reasons for this exception include the following:
 - The channel server is down or unreachable.
- **ChannelConnectorException**—`ChannelConnectorConnectionException` and `ChannelConnectorPushException` inherit `ChannelConnectorException`.

QUEUE CONNECTOR EXCEPTIONS

There are three types of Queue Connector exceptions:

- **QueueConnectorConnectionException**—This exception occurs when a Queue Connector is unable to connect to the specified queue. Reasons for this exception include the following:
 - The queue server is down or unreachable.
 - The queue does not exist.
- **QueueConnectorPushException**—This exception occurs when a Queue Connector is unable to send an event to a queue. Reasons for this exception include the following:
 - The queue server is down or unreachable.
- **QueueConnectorException**—QueueConnectorConnectionException and QueueConnectorPushException inherit QueueConnectorException.

For more detailed information on exceptions, see [Chapter 26, “Exception Handling.”](#)

CHANNELS AND QUEUES

Channel and Queue Connector Exceptions

8

APPLICATION SERVER INTEGRATION

This chapter describes how BusinessWare integrates with servers in internal and external J2EE-compliant applications.

Topics include:

- [Synchronous Integration](#)
- [Asynchronous Integration](#)

BusinessWare can interact with J2EE-compliant application servers in a bidirectional fashion (send/receive). BusinessWare receives requests from an application server, and also makes requests to an application server. [Figure 8-1](#) shows how an environment where BusinessWare and application servers coexist might work:

- A Web server handles authentication and presentation logic.
- An application server performs front-end business logic (for example, validation).
- BusinessWare performs the back-end integration logic, data transformations, application connectivity and process analysis.

APPLICATION SERVER INTEGRATION

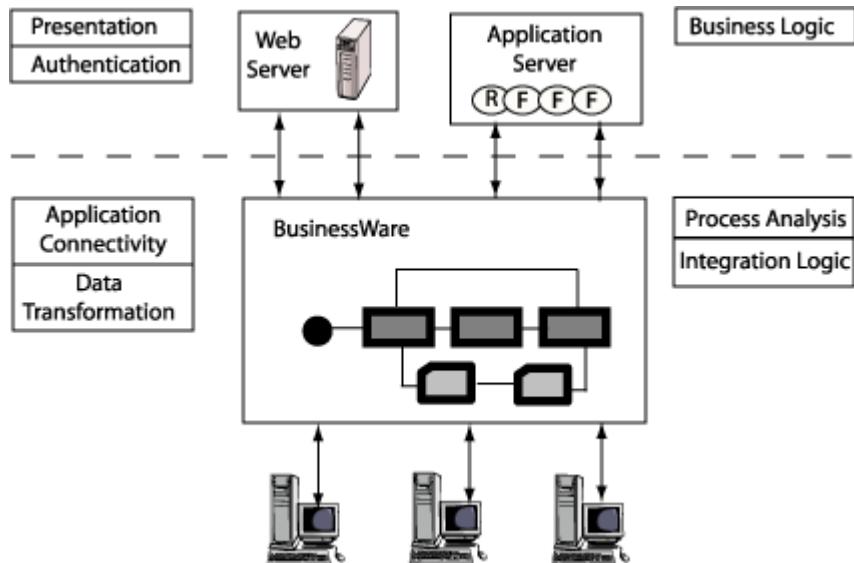


Figure 8-1 Interoperability between BusinessWare and Application Servers

For example these functions might be divided as shown in [Figure 8-2](#).

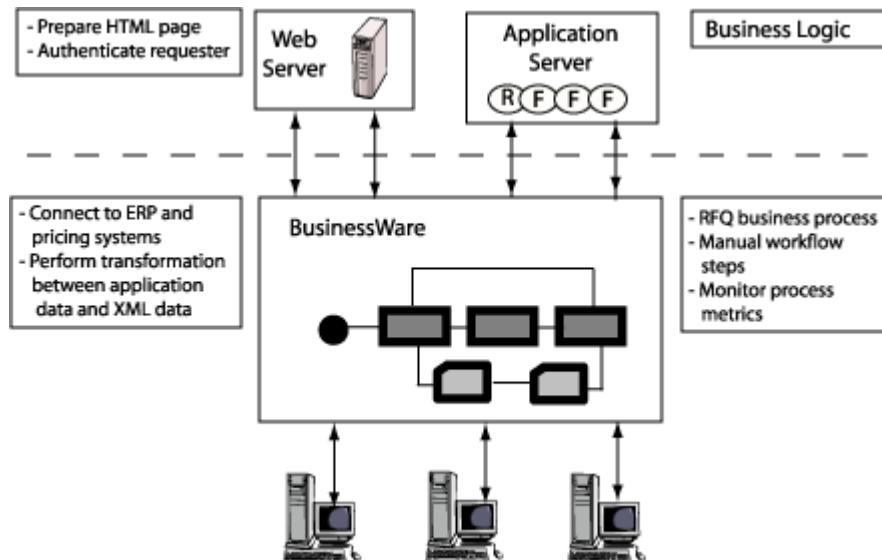


Figure 8-2 Application Server Interoperability Example

The integration between BusinessWare and application servers can be synchronous (request/reply), or asynchronous. In addition, BusinessWare can integrate between internal and external applications using RMI, EJB, and Web Services proxy objects.

The following sections describe how to use these interactions in the BusinessWare Modeling Environment (BME).

SYNCHRONOUS INTEGRATION

A BusinessWare component is allowed to receive a request from and make a request to EJBs residing in external application servers. As with all other kinds of external communication in BusinessWare, ports are used to enable synchronous invocation between components and EJBs. In the Synchronous section of the SimpleOrderSample shown in [Figure 8-3](#), calls are made in and out of BusinessWare from EJBs that run inside of an application server.

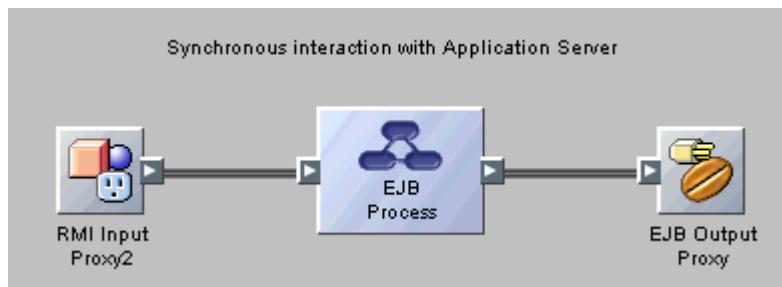


Figure 8-3 Synchronous Invocation between a Component and an EJB

In particular, setting up and configuring ports and proxies are almost all that is necessary to implement an integration, as described in the following sections.

Synchronous integration between BusinessWare projects or external J2EE application server objects and BusinessWare indicates one or more of the following interactions:

- **BusinessWare receiving requests from an application server**—An Enterprise Java Bean (EJB) deployed in a third-party application server can make requests against a model with a Simple Input proxy deployed in BusinessWare and wait synchronously for the request to return a reply.
- **BusinessWare making requests to an application server**—A BusinessWare model with an EJB Output Proxy makes requests against an EJB deployed in a third-party application server and wait synchronously for the request to return a reply.

- **BusinessWare projects making RMI requests to each other**—A BusinessWare component connected to an Simple Output Proxy makes requests against another BusinessWare component, in a different project, connected to a Simple Input Proxy.

For more information on proxies, see [Chapter 6, “Integration Model Basics.”](#)

You should also refer to [Chapter 25, “Transaction Management”](#) for information on transaction propagation.

RECEIVING REQUESTS FROM A J2EE APPLICATION SERVER

At model design time, one or more input ports can be associated with a component to define the set of requests the component may receive from external application server objects and the set of results the component returns in response to these requests. An input proxy is added to provide an externally accessible connection point so the external call can be made to the input port.

The following lists the general steps for setting up input ports to receive requests from EJBs. For specific information, see *BME Help*.

To set up input ports to receive requests from EJBs:

1. Assign a Java-defined type that extends `java.rmi.Remote` to the input port.
2. Add an Simple Input Proxy and wire it to the input port.
3. Set the Binding Path Name property of the proxy to the location in the BusinessWare namespace where the proxy will be bound.
Note: Leaving the Binding Path Name field empty defaults to the full path name of the port to which it is connected.
4. Make sure that the guest credential for the BusinessWare Server is configured and has access to the port. See *BusinessWare Samples Overview* for information on configuring the guest credential.

To make calls on a port from an EJB running in an J2EE application server:

1. Make sure that the Java classes that define the port type are in the J2EE application server’s CLASSPATH.
2. Make sure that the any classes required to access the directory server are on the J2EE application server’s CLASSPATH. This allows the bean to use JNDI to access the BusinessWare directory server namespace.

Once configured, the EJB code will be able to make synchronous Remote Method Invocation/Internet Inter-ORB Protocol (RMI/IOP) invocations on the BusinessWare port as follows:

- Creates an initial naming context from the directory server provider URL, directory server user, and password (taken from the BusinessWare VTPARAMS file).
- Using the naming context, look up the name specified in the Binding Path property of the RMI Simple Input Proxy.
- Using the PortableRemoteObject.narrow () method, narrow the object retrieved to the remote interface on the BusinessWare port.
- Makes calls on the resulting reference.

See the code in the *SimpleOrder Sample* for more details. To access instructions for installing and running this sample, see
installdir\samples\protocols\SimpleOrderSample.htm.

MAKING REQUESTS ON EJBs IN A J2EE APPLICATION SERVER

At model design time, one or more output ports can be associated with a component. These ports define the set of requests the component may make and the types of results these requests return. The ports can be wired to output proxies that specify an external object to invoke.

The following lists the general steps for requests on EJBs. For specific information, see *BME Help*.

To make requests on EJBs through output ports:

1. Ensure that the remote Java interfaces of the EJB and other associated classes are available to the project:
 - a. Mount the directory that contains the interfaces.
 - b. Right-click each interface.
 - c. Select **Use As Type** for each interface. You can select Use As Type on the directory, and it will include all of the interfaces in the Types directory.
2. Assign an EJB interface as the type for the output port.
3. Create an EJB Output Proxy and wire it to the output port.
4. Configure the proxy:

- a. Set the **Home Path Name** property of the proxy to the path where BusinessWare can resolve the EJB home interface in the application server's namespace. This path is configured in the EJB's deployment descriptor.
- b. Set the **Factory Class** property to the fully-qualified name of a class that, given a reference to the EJB's home interface, will invoke the appropriate methods and return a reference to the correct bean.
- c. Set the **Expert** properties, shown in [Figure 8-4](#) and described in [Table 8-1](#), to specify the name service of the application server. In particular, the Provider URL should specify the host and port at which the application server is running. For a description of all of the EJB Output Proxy properties, see the *BME Help*.

Once configured, a BusinessWare model can invoke a method on an EJB by invoking the method on the corresponding output port.

See the code in the SimpleOrderSample for detail. To access instructions for installing and running this sample, see *installdir\samples\protocols\SimpleOrderSample\SimpleOrderSample.htm*.

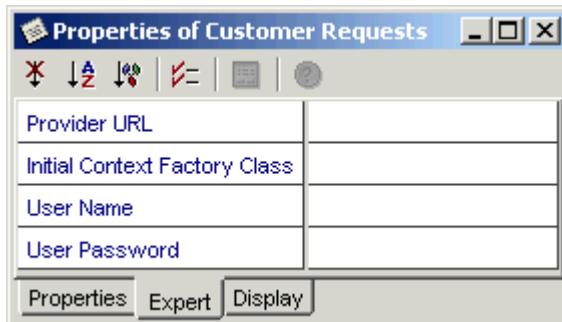


Figure 8-4 Expert Property Sheet

[Table 8-1](#) provides Expert property sheet descriptions.

Table 8-1 Expert Properties

Property	Description
Provider URL	URL that specifies a foreign name server.
Initial Context Factory Class	A class used to create JNDI contexts for binding and resolving servants.
User Name	A security principal.
Password	A security credential.

IIOP/GIOP FUNCTIONALITY

IIOP maps GIOP messages to TCP/IP. IIOP specifies the CORBA ORB which allows applications to communicate with each other regardless of location. GIOP enables interoperable communications between disparate ORB implementations over TCP/IP connections. For more information on IIOP/GIOP, go to the Object Management Group™ (OMG™) Web site: <http://www.omg.org>.

Note: The BusinessWare CORBA ORB does not support sending or receiving GIOP chunked data. If BusinessWare receives GIOP chunked data, the message is rejected and an error is logged. By default, BusinessWare uses GIOP 1.1 for sending all data. To ensure that messages are not rejected, you can specify that BusinessWare accept only GIOP 1.1 replies.

To specify GIOP 1.1:

1. Enter the following line in your VTPARAMS file:

```
giop_java_version = "1.1"
```
2. Save and close the VTPARAMS file.

SECURITY CONSIDERATIONS

See the *BusinessWare Security Guide* for information on all BusinessWare security issues.

ASYNCHRONOUS INTEGRATION

In addition to synchronous integration, you can also asynchronously integrate BusinessWare and external J2EE application servers. Asynchronous integration is a loosely coupled way to exchange data between the systems: the data provider is unaware of when its data is received or processed by the data recipient. There are a number of ways to achieve this using the various connectivity options provided by the Vitria connectors including:

- An EJB can persist its data into an RDBMS (Relational Database Management Systems). BusinessWare models can get the changed data through an RDBMS connector (see the *RDBMS Connector Guide*).
- An EJB in the J2EE application server can put messages into BusinessWare channels and queues using the publish/subscribe APIs available in BusinessWare. See the *BusinessWare Programming Reference* for more information.
- BusinessWare can push messages into a JMS (Java Messaging Service) queue or topic in the J2EE application server. (See [Figure 8-6](#).)

As shown in the SimpleOrderSample example in [Figure 8-5](#) this can be modeled in the BME like any other connector interaction.

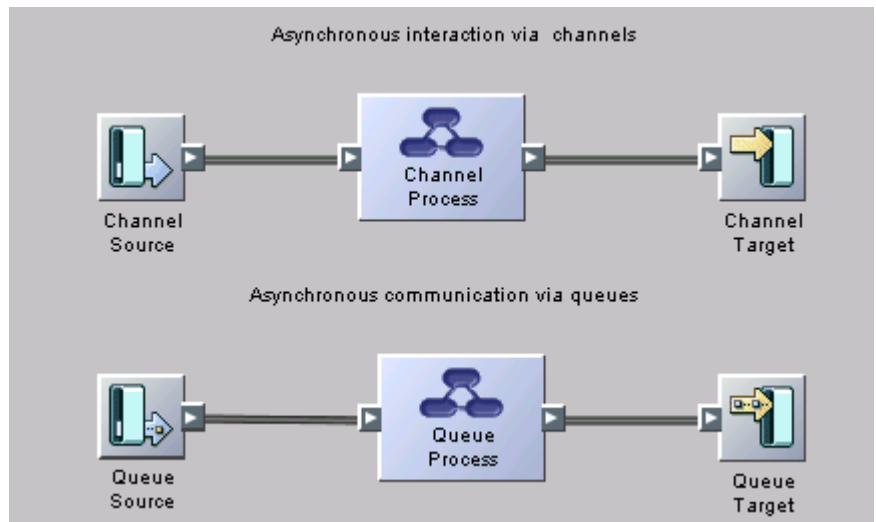


Figure 8-5 Asynchronous Integration with Channels and Queues

See the code in the SimpleOrderSample for detail. To access instructions for installing and running this sample, see *installdir\samples\protocols\SimpleOrderSample\SimpleOrderSample.htm*.

[Figure 8-6](#) shows connectivity using a JMS queue or topic.

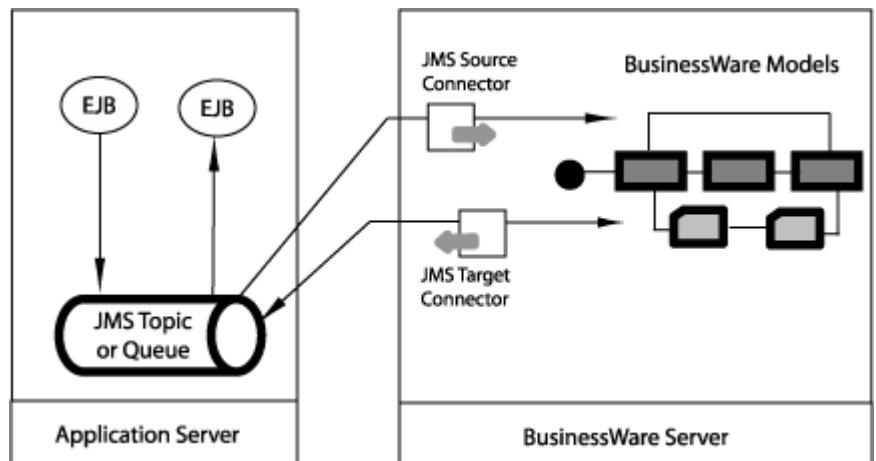


Figure 8-6 BusinessWare Messages and JMS Queues

This chapter describes BusinessWare’s Web services capabilities. It includes the following topics:

- [Introduction to Web Services](#)
- [Web Service Input Proxy](#)
- [Web Service Output Proxy](#)
- [Exporting a Web Service](#)
- [Invoking a Web Service Using the Web Service State](#)
- [Sample Files](#)

For information on migrating a Web services solution from BusinessWare 3.x to BusinessWare 4.x, see the *BusinessWare Migration Guide*.

For detailed instructions on performing any of the tasks discussed in this chapter, see the *BME Help*.

INTRODUCTION TO WEB SERVICES

The term *Web services* is used throughout the Internet community to refer to a collection of standards that allow clients to access services programmatically over the Internet by exchanging XML data through the Simple Object Access Protocol (SOAP). Web services enable BusinessWare to connect to other applications over the Internet, to provide application services to clients over the Internet, and to communicate with other instances of BusinessWare over the Internet.

The Web Services Description Language (WSDL) is the XML format used for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into services.

BusinessWare uses the JAX-RPC specification (see “[Supported Standards](#)” on page [9-2](#)).

INTEROPERABILITY

BusinessWare's Web services feature is built on open standards (see “[Supported Standards](#)” below). It interoperates with other Web services implementations, such as Microsoft.Net. That is, it can be invoked by clients with different implementations as well as invoke services with different implementations at runtime.

However, since Web services technologies as a whole are still loosely defined and continuing to evolve, interoperability among different implementations on different languages, operating systems, and platforms remains a challenge. You should take extra care when designing a system that involves different Web services implementations.

Interoperability is limited by different programming models in different implementations. Certain Web services capabilities may be available in one programming model but not in another programming model. Thus, WSDL generated from one implementation might not be fully understandable in a different implementation. A common issue here is that certain complex XML schemas are mappable in one language but not in another.

SUPPORTED STANDARDS

BusinessWare's Web services feature supports the following standards:

- SOAP 1.1 and 1.2 (Simple Object Access Protocol)—a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML.
- WSDL 1.1 (Web Services Description Language)—XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.
- XML Schema 1.0—language for describing the structure and constraining the contents of XML documents.
- JAX-RPC 1.1 (Java API for XML-Based RPC)—JAX-RPC enables Java developers to build Web applications and Web services by incorporating XML-based RPC functionality according to the SOAP specification.
- XML (Extensible Markup Language)—specification for creating markup languages that define the structure of documents.
- UDDI 3.0 (Universal Description, Discovery & Integration)—a platform-independent way of describing and discovering Web services and Web service providers.

Additional Resources

You can find additional information about these standards at the following sites:

- **SOAP:**
 - **Version 1.1:** <http://www.w3.org/TR/SOAP/>
 - **Version 1.2:**
 - <http://www.w3.org/TR/soap12-part1>
 - <http://www.w3.org/TR/soap12-part2>
 - SOAP Messages with Attachments: <http://www.w3.org/TR/SOAP-attachments>
- **WSDL:** <http://www.w3.org/TR/wsdl>
- **XML schema:**
 - <http://www.w3.org/TR/xmlschema-0/>
 - <http://www.w3.org/TR/xmlschema-1/>
 - <http://www.w3.org/TR/xmlschema-2/>
- **JAX-RPC:** <http://java.sun.com/xml/jaxrpc/>
- **XML:** <http://www.w3.org/TR/REC-xml>
- **UDDI:** <http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v2.htm>

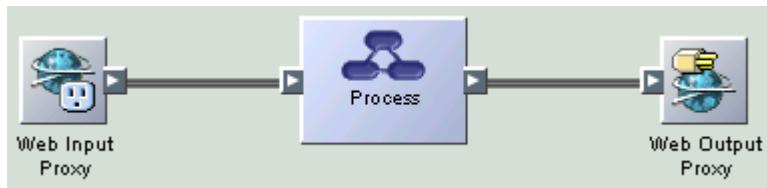
OVERVIEW OF WEB SERVICES OBJECTS

In the BusinessWare Modeling Environment (BME), the Web services functionality is comprised of the following modeling objects:

- Web service input proxy
- Web service output proxy
- Web service state

The Web service input proxy accepts Web service invocations on behalf of a process component or integration component and translates them into BusinessWare operations. For more information, see “[Web Service Input Proxy](#)” on page 9-8.

The Web service output proxy acts as a client to invoke other Web services over the Internet. For more information, see “[Web Service Output Proxy](#)” on page 9-13.

**Figure 9-1 Web Services Integration Model**

For general information about proxies, see “[Proxies](#)” on page 6-28.

The Web service state lets you invoke a Web service in a process model with a minimum amount of programming. For more information, see “[Invoking a Web Service Using the Web Service State](#)” on page 9-27.

**Figure 9-2 Web Services Process Model**

These modeling objects enable you to build a Web service in BusinessWare and invoke other Web services from BusinessWare.

OVERVIEW OF WSDL MODEL AND UDDI MODEL

WSDL Model

A service can be described using a WSDL Document. In WSDL a service is defined in three distinct sections:

- portType—a set of operations provided by the services described by those operations.
- binding—wire protocol and message formats offered by the service.
- service—ports for using the service, each port possibly with several bindings.

UDDI Model

The data structure within UDDI is comprised of four constructions:

- businessEntity—representing a Web Service provider with company name, contact details, and other business information.
- businessService—a logical group of one or several Web Services, contained by the business entity.

- bindingTemplate—the technical information needed to bind and interact with the target Web Service. Contains the access point (for example, URI to invoke a Web Service).
- tModel—Representing the technical specification; typically a specifications pointer, or metadata about a specification document, including a name and a URL pointing to the actual specification.

REGISTRATION AND UNREGISTRATION

The service implemented by web service input proxy is registered as a UDDI business service record. Any ports defined in the WSDL are published in a binding template record. The bindings and portTypes defined in the WSDL are published as tModels. WSDL file exported from the web service input proxy is a complement WSDL containing only one port. You must publish the wsdl:Service, wsdl:binding, and wsdl:portType portions of the WSDL to the UDDI server.

A business entity will be created if no business entity is found with that name.

Note: It is recommended that UDDI tools be used to register business entities in the UDDI registry, so detailed information can be registered.

The WSDL file exported from WSIP is a complementary WSDL containing only one port. The following parts of WSDL should be published in the UDDI server: wsdl:Service, wsdl:binding, wsdl:portType.

Mapping the wsdl:portType to the uddi:tModel

The following table maps the wsdl:portType to the uddi:tModel.

Table 9-1 wsdl:portType is modeled as a uddi:tModel

WSDL	UDDI
portType	tModel (categorized as portType)
Namespace of portType	keyedReference with a tModelKey of the XML Namespace category system and a keyValue of the target namespace of the wsdl:definitions element that contains the wsdl:portType in categoryBag
Local name of portType	tModel name
Location of WSDL document	overviewURL

Mapping the wsdl:binding to the uddi:tModel

The following table maps the wsdl:binding to the uddi:tModel.

Table 9-2 wsdl:binding is modeled as a uddi:tModel

WSDL	UDDI
binding	tModel (categorized as binding and wsdlSpec)
Namespace of binding	keyedReference in categoryBag
Local name of binding	tModel name
Location of WSDL document	overviewURL
portType binding relates to	A keyedReference with a tModelKey of the WSDL portType Reference category system and a keyValue of the tModelKey that models the wsdl:portType to which the wsdl:binding relates in categoryBag
Protocol from binding extension	keyedReference in categoryBag
Transport from binding extension (if there is one)	keyedReference in categoryBag

Mapping the wsdl:service to the uddi:businessService

The following table maps the wsdl:service to the uddi:businessService.

Table 9-3 wsdl:service is modeled as a uddi:businessService

WSDL	UDDI
Service	businessService (categorized as service)
Namespace of Service	keyedReference in categoryBag
Local Name of Service	keyedReference in categoryBag; optionally also the name of the service

The bindingTemplates element of the businessService will include bindingTemplate elements that model the ports of the service.

The following table maps the wsdl:port to the uddi:bindingTemplate.

Table 9-4 wsdl:port is modeled as a uddi:bindingTemplate

WSDL	UDDI
Port	bindingTemplate
Namespace	Captured in keyedReference of the containing businessService
Local Name of port	instanceParms of the tModelInstanceInfo relating to the tModel for the binding
Binding implemented by port	tModelInstanceInfo with tModelKey of the tModel corresponding to the binding
portType implemented by port	tModelInstanceInfo with tModelKey of the tModel corresponding to the portType
wsdl:port Address Extensions	uddi:accessPoint of bindingTemplate
wsdlsoap: address	AccessPoint with the value of location attribute of the soap: address element and with a useType of endpoint.

For more information on registering to a UDDI registry or unregistering from a UDDI registry, see the *BusinessWare Online Help*.

Discovery and Invocation

At modeling time, the web service input proxy, the web service output proxy, and the web service state can locate a remote WSDL by querying the UDDI registry through UDDI keys or qualified names.

At runtime, the web service output proxy can discover the web service access point.

The service URL can take the form of UDDI : <inquiryurl>?bindingkey=.... This bindingkey can be specified at design time when WSDL is located or at runtime a dynamic class can return the service key with the URL.

The web service endpoint URL is looked up from UDDI registry when the web service output proxy or web service state makes the first call and will then be cached. The web service endpoint will be retrieved again if a failure occurs.

Note: Query binding through UDDI-Key and qualified names are supported.

When the real endpoint of the web service is not available, such as when the service provider has moved the service into another location and has updated the address in the WSDL file, the WSDL implement document is reaccessed to get the real endpoint.

The following pre-defined useTypes are supported:

- endPoint—designates that the accessPoint points to the actual service endpoint, such as the network address at which the Web service can be invoked.
- bindingTemplate—designates that the accessPoint contains a bindingKey that points to a different bindingTemplate entry. The value in providing this facility is seen when a business or entity wants to expose a service description that is actually a service that is described in a separate bindingTemplate record. This might occur when many service descriptions could benefit from a single service description.
- wsdlDeployment—designates that the accessPoint points to a remotely hosted WSDL document that already contains the necessary binding information, including the actual service endpoint.

WEB SERVICE INPUT PROXY

The Web service input proxy accepts Web services invocations on behalf of a BusinessWare process component or integration component. When you attach a Web service input proxy to an input port of a BusinessWare process component or integration component, you enable that component to receive Web services invocations via that input port. The Web service input proxy accepts SOAP requests, converts them to BusinessWare operations, converts the BusinessWare return value to a SOAP reply, and then returns it to the client.

Each Web service input proxy has one output port. This port must be typed by a Service Endpoint Interface or com.vitria.types.XMLServices interface. These interfaces are described in the section below.

Note: The vtWebServices.GenericService interface is deprecated in this release. See the *BusinessWare Migration Guide* for migration issues.

Normally, any SOAP request is converted by the proxy into a Java invocation of the service endpoint interface to BusinessWare based on the JAX-RPC specification. In most cases, you only need to be aware of the Java types during modeling. However, if you want to manipulate the SOAP request natively (that is, in XML format), you must set the **Event Type** property to **XML Envelope** on the input proxy. By setting the property to XML Envelope, the input proxy automatically converts a Web service request using the built-in `com.vitria.types.XMLServices` interface and handles the native SOAP envelope as a document reference. The interfaces are described below.

INTERFACES

Below is a description of the interfaces used with BusinessWare's Web services feature.

- Service Endpoint Interface

To map a Java interface to WSDL definitions, the Java interface must be a Java Remote interface and follow a set of rules defined in JAX-RPC specification section 5.2. This Java interface is called Service Endpoint Interface. Please refer to the JAX-RPC specification for more details.

You can either generate a Service Endpoint Interface from WSDL or manually write a Java Remote interface that conforms to the rules described below. For more information on generating Java interfaces from a WSDL, see [“Configuring a Web Service Input Proxy” on page 9-12](#).

In addition to the requirements in JAX-RPC, BusinessWare requires the following rules on the Java interface:

- If you want to use Event Injection capabilities on ports, all of the method parameters must be Java serializable.
- If you want to define service-specific Java exceptions, the exception class must extend from `com.vitria.types.ServiceException`.
- `com.vitria.types.XMLServices` Interface

To handle a native SOAP envelope, set the **Event Type** property of the Web Service Input Proxy to XML Envelope. The proxy and proxy port types are then set to `com.vitria.types.XMLServices` and the `com.vitria.types.XMLServices.XMLEnvelopeService` method is invoked when a web service invocation is received. The event produced is an `xmLEnvelopeService` event (Map header, `DocumentReference[]` payload, `Part[]` attachments). The header map contains all MIME headers of the SOAP Part.

The Soap envelope is contained in the first element of the DocumentReference array. Soap attachments are contained in com.fc.data.Part array attachments []. Part consists of java.util.Map (Attachment Headers) and DocumentReference (Attachment Data). See “[Web Service Output Proxy](#)” on page 9-13 for more information on Soap attachments.

- com.vitria.types.ServiceException

To define service-specific Java exceptions, the exception class must extend from com.vitria.types.ServiceException. Custom exceptions can extend from java.lang.Exception, but best practice is to use com.vitria.types.ServiceException.

WEB SERVICE INPUT PROXY PROPERTIES

Table 9-5 describes the properties associated with a Web service input proxy. The properties are also described in the *BME Help*.

Table 9-5 Web Service Input Proxy Properties

Property	Description	Default
Properties Tab		
Name	The simple name of the input proxy, which must be unique within the integration model.	Web Input Proxy (2, 3, 4,...)
Mode	Specify the operation style and encoding representation of the service. Note: In some cases (for example, when the service endpoint interface has complex beans and arrays of complex beans), BusinessWare does not support document/literal message encoding. In these cases, you should use the RPC mode which is RPC/Encoded style.	Document/Literal(Wrapped)
SOAP Version	Specify which version of the SOAP standard—either 1.1 or 1.2—is used on this input proxy.	1.1

Table 9-5 Web Service Input Proxy Properties (Continued)

Property	Description	Default
Event Type	<p>Defines the input event type:</p> <ul style="list-style-type: none"> • WSDL or Existing Type—original service endpoint interface is used • XML Envelope—port type is changed to XMLServices. • XML - Generic Web Service—the generic service interface is used. <p>Note: Generic Web Service is deprecated in this release and is only included for backward compatibility.</p>	WSDL or Existing Type
Handlers	JAX-RPC Handlers	
Description	Description of the proxy for documentation purposes.	Web Service Input Proxy
UDDI Tab		
Enable UDDI	Indicates whether to register the web service implemented by the input proxy into UDDI registry.	False
Inquiry URL	The URL that projects use to inquire on the UDDI registry.	
Publish URL	The URL that projects use to publish to the UDDI registry.	
Security URL	The URL that projects use to log into the UDDI registry.	
User Name	The User Name of a UDDI "Authorized Name" with update access to this registry.	

Table 9-5 Web Service Input Proxy Properties (Continued)

Property	Description	Default
Password	The Password of a UDDI "Authorized Name" with update access to this registry.	
Business Name	<p>The registered Business Name under which WSDL will be published.</p> <p>This property is optional. It is recommended that customer should use UDDI tools to register business entity into UDDI, so detailed information can be registered.</p>	
Publisher Key	<p>The Publisher key that is assigned by the UDDI registry.</p> <p>This property is optional.</p>	

CONFIGURING A WEB SERVICE INPUT PROXY

Like other kinds of proxies in BusinessWare, you can configure the Web service input proxy by setting the properties and port type. In addition, the Web service input proxy provides a wizard to help you properly configure the proxy.

To configure the Web service input proxy:

1. Open the integration model where you want to include the proxy.
2. From the **Proxies** tab, select the Web service input proxy () and click in the Editor where you want to place the proxy.
3. Double-click on the Web service input proxy (or select the Web service input proxy, then right-click and select **Tools > Configure...**). The Web Service Configuration Wizard appears.
4. Define the proxy's name, type, and protocol configuration. For detailed instructions, see the *BME Help*.

HTTP Compression

Inbound HTTP requests are processed by your Web server. The Tomcat server bundled with BusinessWare accepts gzip compressed HTTP messages. For information on configuring your Web server, see “[Web Server](#)” on page 20-13.

WEB SERVICE OUTPUT PROXY

The Web service output proxy acts as a client to invoke other Web services. When you attach a Web service output proxy to the output port of a BusinessWare process component or integration component, you enable that component to invoke other Web services via the output port.

Each Web service output proxy has one input port. This port must be typed by a WSDL type or by the built-in type `XMLServices`. The WSDL types are the Java types generated by using the Web Service Configuration Wizard. Any other RMI type (one that is not generated by importing a WSDL) is not supported on the output proxy because the type of the output proxy must exactly match the service definition of the Web service you want to invoke.

The Web service output proxy supports different operation modes based on the value specified in WSDL. If the operation mode specified in WSDL is something other than RPC style with encoded representation or document style with literal representation, the value for the **Mode** property is set to “Defined in WSDL”.

Normally, the Java invocation of the service endpoint interface is converted into the SOAP request. In most cases, you only need to be aware of the Java types during modeling. However, for dealing with attachments or Soap envelopes in raw XML format, you must set the Event Type property to XML Envelope. By setting the Event Type property to XML Envelope, the port type automatically changes to the built-in `XMLServices` interface.

You can manipulate the SOAP Envelope as a Document Reference to obtain the raw XML. In addition, you can add attachments as a `com.vitria.fc.data.Part` object. You can manipulate the response as a `com.vitria.fc.data.Envelope`.

To configure the output call to send a SOAP message to the Service URI specified in the Web Service Output Proxy properties, your process model code must correctly define the parameters for the output call. Refer to the following guidelines:

- The header parameter should contain the header items of the SOAPPart of the outgoing SOAP message. If the header is empty, the default header elements that are created during creation of the SOAPPart are used.

- The first element of the payload parameter is used as the content of the outgoing SOAP Envelope. If no payload is specified, the default empty envelope is used.
- The SOAP attachments are constructed from each element of the attachment parameter. Each Part object should be constructed in the following way:
 - The `data_` attribute should contain the contents of the attachment.
 - The `header_` attribute should contain the attachment headers. At a minimum, it should contain the Content-Type key.
 - One entry of the `header_` attribute must contain the instance of the DataHandler that is used as the DataHandler of the attachment. The key to this entry (as specified in `com.vitria.fc.data.Envelope`) is `ATTACHMENT_HANDLER`.

Note: If the Web Service Output Proxy's Service URI property has a URL in the form of `http(s)://<username>:<password>@<hostname>:<port>/<service path>`, then HTTP Basic Authentication will be enabled using the credentials specified in the given URL. If the URL used by an Output Proxy is "HTTPS," the VTPARAMS file defined by the VTPARAMS environment variable determines the configuration for the HTTPS connection. In the VTPARAMS file, `ssl_capath` specifies the name of a directory containing trusted CA certificates. For more information on the VTPARAMS file, see the *BusinessWare Installation Guide*. For more information on security, see the *BusinessWare Security Guide*.

For more information, see "[Proxies](#)" on page 6-28.

WEB SERVICE OUTPUT PROXY PROPERTIES

[Table 9-6](#) describes the properties associated with a Web service output proxy. The properties are also described in the *BME Help*.

Table 9-6 Web Service Output Proxy Parameters

Property	Description	Default
Name	The simple name of the output proxy, which must be unique within the integration model.	Web Output Proxy (2, 3, 4,...)
Mode (read-only)	The value is based on the operation style and encoding representation value in WSDL. If the binding for this port is RPC style with encoded representation in WSDL, then RPC is selected. If the binding for this port is document style with literal representation in WSDL, then Document/Literal(Wrapped) is selected. For all other values in WSDL, Defined in WSDL is selected.	As defined in WSDL
Service URI	The SOAP address of the Web service you want to invoke. This value, if present, was set when you imported the WSDL. Service URI can be a static URI or a dynamic map.	
SOAP Version	Identifies which version of the SOAP standard is used on this output proxy.	1.1

Table 9-6 Web Service Output Proxy Parameters (Continued)

Property	Description	Default
Event Type	<p>Defines the input event type:</p> <ul style="list-style-type: none"> • WSDL or Existing Type—original service endpoint interface is used • XML Envelope—port type is changed to XMLServices. • XML - Generic Web Service—the generic service interface is used. 	WSDL or Existing Type
Handlers	JAX-RPC Handlers	
Use HTTP 1.1	If set to True , makes HTTP 1.1 requests, otherwise makes HTTP 1.0 requests.	True
Socket Timeout	Socket timeout in seconds. A value of 0 indicates no timeout.	60
Compression Type	<p>Specifies whether a compressed HTTP request is sent. Choices include:</p> <ul style="list-style-type: none"> • IDENTITY—no compression • GZIP—use gzip compression 	IDENTITY

Table 9-6 Web Service Output Proxy Parameters (Continued)

Property	Description	Default
Transfer Encoding	The body of the message is modified in order to transfer it as a series of chunks, each with its own size indicator, followed by an optional trailer containing entity-header fields. If it is set to None, the outgoing request will not be chunked and the Web service output proxy will not ask that the incoming reply be chunked. If it is set to Chunk, the outgoing request is sent in chunks and the Web service output proxy tells the target Web server that it can receive a chunked reply.	None
Description	Description of the proxy for documentation purposes.	Web Service Output Proxy
UDDI Tab		
Inquiry URL	The URL that projects use to inquire on the UDDI registry and invoke the web service.	
Binding Key	The UDDI key of the binding template.	
SSL Tab		
SSL		
SSL Enabled	Specifies whether to use SSL with HTTP protocol.	False
Client Authentication Enabled	Specifies whether client authentication will be used as part of the SSL connection.	False

Table 9-6 Web Service Output Proxy Parameters (Continued)

Property	Description	Default
Private Key Passphrase	The pass phrase for the private key. This value is only needed when client authentication is enabled.	
Private Key File	The full path name of the PKCS8 private key file to use. This value is only needed when client authentication is enabled.	
Client Certificate File	The full path name of the PEM-encoded certificate file to use. This value is only needed when client authentication is enabled.	

CONFIGURING A WEB SERVICE OUTPUT PROXY

Like other kinds of proxies in BusinessWare, you can configure the Web service output proxy by setting the properties and port type. In addition, the Web service output proxy provides a wizard to help you properly configure the proxy.

To configure the Web service output proxy:

1. Open the integration model where you want to include the proxy.
2. From the **Proxies** tab, select the Web service output proxy () and click in the Editor where you want to place the proxy.
3. Double-click on the Web service output proxy (or select the Web service output proxy, then right-click and select **Tools > Configure...**). The Web Service Configuration Wizard appears.
4. Complete the wizard to configure the proxy properties. For detailed instructions, see the *BME Help*.

Note: The Web service output proxy supports dynamic binding of the Service URI. The location can be set as one of the following:

- **Static Binding**—sets a static location
- **Dynamic: Class**—sets the location based on a Class file

Note: This implementation is similar to that of the dynamic map for the Simple Output Proxy. For more information, see “[Using Proxies with the Dynamic Map Class](#)” on page 6-35.

- **Dynamic XQuery**—sets the location using an XML file or array
- **Dynamic Java**—sets the location using a piece of Java code

When dynamic binding rules are specified (either in XML or Java), BusinessWare auto-generates the code.

Example: The following is an example of how to configure the Service URI using Dynamic XQuery. The file that you use to parse the incoming payload (XML file) must match the structure of the incoming file. This example assumes that the incoming payload adheres to the structure defined in the example file serviceuri.xml.

To configure the Service URI property:

1. Create an XML file called **serviceuri.xml** that contains the XML schema used to parse the incoming payload and save the file in your project directory. For the purposes of this example, the **serviceuri.xml** file is based on the following XML payload.

```
<?xml version="1.0" ?>
<CUSTOMER id="1234">
    <NAME>Customer One</NAME>
    <ORDER>
        <NUMBER>ab122</NUMBER>
        <DATE>May 6, 2005</DATE>
        <ITEM>
            <SERIAL>421424</SERIAL>
            <Description>Large Widget</Description>
            <PRICE>321.00</PRICE>
        </ITEM>
        <CONFIRMATION
            uri="http://www.company.com:80/orders/confirm.cgi" />
    </ORDER>
</CUSTOMER>
```

2. Select the Web Service Output Proxy in the Editor. In the Properties window, click in the **Service URI** field and select **Query By String**. Click **OK**.
3. In the Service URI window, click **Edit...**
4. In the XQuery Builder, enter **serviceuri.xml** in the **Source Schema:** field.
5. Use the XQuery editor to construct the XQuery which will return the Service URI as a String.

- In the Source Schema pane (Figure 9-3), expand /CUSTOMER/ORDER/CONFIRMATION/Attributes and drag URI to the Mapping pane. Click **Ok** to exit the XQuery Builder. Click **OK** to exit the Service URI window.

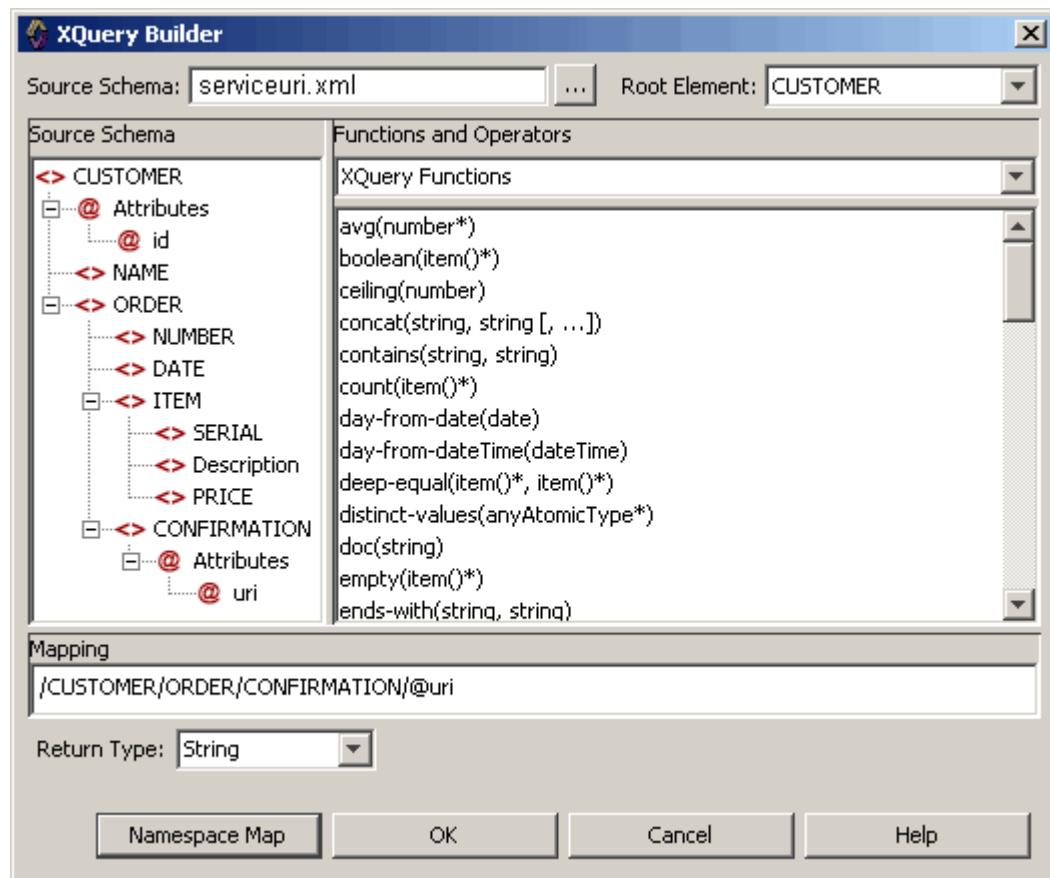


Figure 9-3 Mapping the URI Attribute in the XQuery Builder

After building the project, the XQuery is automatically generated in a file called *integration model.java*, where *integration model* is the name of the model that contains the Web Service Output Proxy.

The generated .java file will look similar to the following:

```
// GEN-BEGIN:CodeGen_Begin
/*
 *Generated Code for IntegrationModel 'CustomerOrder'
```

```

        */

import com.vitria.fc.flow.EventBody;
import com.vitria.container.client.ModelContext;
import
    com.vitria.modeling.integration.IntegrationModelNestingRu
le;
import com.vitria.container.map.DynamicPortMap;
import com.vitria.container.client.ComponentContext;
import com.vitria.xquery.XQueryActionLib;
import com.vitria.fc.data.DocumentReference;
import com.vitria.fc.diag.DefaultLogger;
import com.vitria.fc.diag.DiagLogger;
import com.vitria.fc.diag.DiagError;
import com.vitria.af.common.ContentLib;

public class CustomerOrder {

/*
 *Generated Code for 'Dynamic: XQuery -- DynamicBinding_ProxyToConvertor_0'
 */
public static class DynamicBinding_ProxyToConvertor_0
implements DynamicPortMap {
    public DynamicBinding_ProxyToConvertor_0() {
        //required public default constructor
    }

    public String mapToPort(ComponentContext
modelContext, Object mapInfo) {
        try {
            DocumentReference[] payload = null;
            if("xmlEnvelopeEvent".equals(modelContext.ge
tEvent().getDef().getName())) {
                payload =
(DocumentReference[]) (modelContext.getEvent().getJavaPara
meters()[1]);
            }
            // Query XML data
            return
XQueryActionLib.queryAsString("/CUSTOMER/ORDER/CONFIRMATI
ON@uri",payload[0]);
        }
        catch(Exception e) {
            if (DefaultLogger.traceLevelContainer_ >=
DiagLogger.ERROR) {
                DiagError.logWholeException(e);
            }
        }
        return null;
    }
}

```

```
        }
    }
}
//GEN-END:CodeGen_Begin
```

IMPORTANT: You should never modify this auto-generated .java file.

During runtime, the project will automatically parse the XML payload that is part of the `xmlEnvelopeService` event described in “[Interfaces](#) on page 9-9” and determine the Service URI used by the Web Service Output Proxy.

CONFIGURING THE WEB SERVICE OUTPUT PROXY TO USE HTTPS/SSL

The Web service output proxy can be used to exchange data using HTTPS/SSL. When the Web service output proxy Service URI property is set to an HTTPS address, HTTPS is automatically enabled.

To configure the Web service output proxy SSL properties:

1. Select the Web service output proxy in the Editor.
2. In the properties window, select the **SSL** tab.
3. Configure the SSL properties. For a description of the Web service output proxy properties, see “[Web Service Output Proxy Properties](#)” on page 9-15.

USING HANDLERS

A SOAP Message handler intercepts SOAP messages between the client and the server and accesses a SOAP message that represents either a SOAP request or a SOAP response. Message handlers are tied to the Web service endpoints and are used to provide additional processing mechanisms when invoking a service using RPC.

[Figure 9-4](#) shows the Handlers dialog for a Web service input proxy. Here, the developer has specified two handlers, Handler_1 and Handler_2. These two handlers constitute a handler chain, in the specified order, for the input proxy. The handler must be specified using the fully qualified class name of a JAX-RPC compliant handler implementation present in the project.

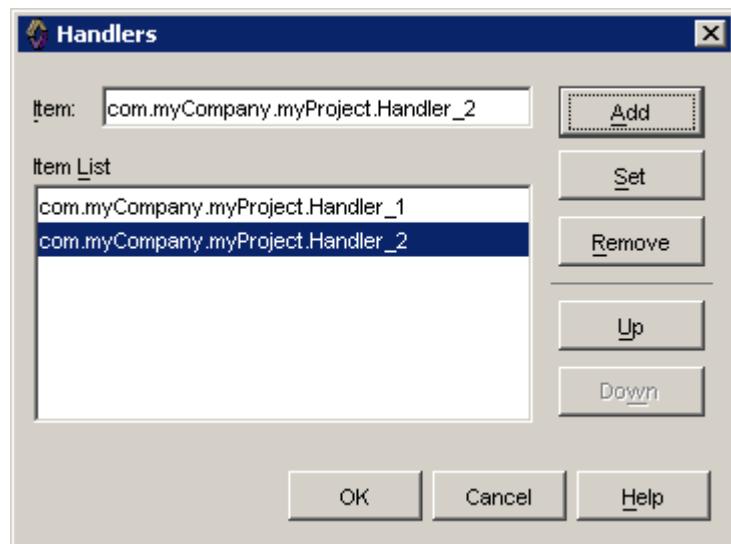


Figure 9-4 Web Service Input Proxy Handlers

When this project is deployed and a Web Service call comes into this input proxy, handler-specific methods are invoked before and after reaching the process model. This is illustrated in [Figure 9-5](#).

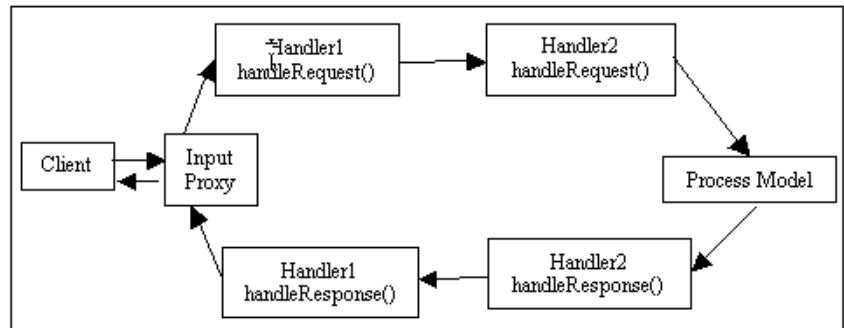


Figure 9-5 Web Service Request-Response on an Input Proxy

You can add any number of class names. Each class must implement the JAX-RPC Handler interface and be a part of the project. The order of the classes in the Item List is preserved when invoking the handlers (for example, if Handler_1 is above Handler_2 for an input proxy, the handleRequest of Handler_1 is called first and then the corresponding method of Handler_2 is called).

SENDING HTTP TRANSPORT HEADERS

After setting a property on the Stub, you can send HTTP transport headers. Values are set in all HTTP requests against the Stub until the property is set to null or the Stub is discarded.

To send values in the HTTP transport headers on outbound requests, do the following:

1. Create a java.util.Hashtable that will contain the HTTP header identifiers and the associated values.
2. Set the Hashtable entry key to a string that exactly matches the HTTP header identifier.

Note: HTTP header identifiers such as Host, Content-Type, SOAPAction, Content-Length, Cookie, Cookie2, Authorization, Proxy-Authorization and Connection are ignored by HTTP Transport.

3. Set the Hashtable on the Stub by using the property com.vitria.webservices.output.HTTPHeaders. When the HTTPHeaders property value is set, that Hashtable is used to set the header values in the outgoing requests.

You can now issue remote method calls against the Stub. Headers and the associated values from the Hashtable are added to outgoing HTTP requests.

Note: A JAXRPCException may occur if the property is not set correctly. The following must be true:

- The property value set on the Stub is a Hashtable object or null.
- The Hashtable is not empty.
- Each key in the Hashtable is a String.
- Each value in the Hashtable is a String.

Example: The following code is from a sample Hashtable:

```
Object port = getOutPort();
Stub stub = (javax.xml.rpc.Stub)port;
Hashtable headers = new Hashtable();
headers.put("TE", "chunked");
headers.put("Accept-Encoding", "gzip");
stub._setProperty("com.vitria.webservices.output.HTTPHeaders",
",headers);
```

Note that this example meets the requirements listed above.

The HTTP headers may override properties such as Content Encoding and Transfer Encoding configured in the BME.

IMPORTANT: Using Axis properties to specify the HTTP transport headers is not supported. You should not use Axis properties to specify HTTP transport headers.

EXPORTING A WEB SERVICE

Exporting a Web service enables you to make the BusinessWare interface for the models available from the external applications that will invoke it.

When you export a Web service, you actually export the WSDL file associated with a Web service input proxy. WSDL files can be generated for each Web service input proxy based on its properties and port type (for more information, see “[WSDL Dependencies](#)” on page 9-25). Each proxy generates one WSDL file.

There are three ways to export a WSDL file:

- [Exporting a Web Service from the Project](#)
- [Exporting a Web Service from a Web Service Input Proxy](#)
- [Exporting a Web Service from an Input Port](#)

Note: If you export the WSDL after the project has been deployed, BusinessWare uses the configuration information from the last deployment that was made from the BME.

WSDL DEPENDENCIES

An exported Web service’s WSDL file includes the host name and port where the Web service is configured. BusinessWare provides a Web server to deploy Web services over HTTP and HTTPS protocols. From the Integration Server properties, you can choose whether you want to use BusinessWare’s internal Web server or your own external Web server. Within the Web server properties, you can select HTTP or HTTPS protocols for the configured Web service. For more information, see [Chapter 20, “Integration Servers.”](#)

BusinessWare generates different types of WSDL depending on whether the project is deployed and the settings you specify for the project and BusinessWare Server.

- If the project is deployed to a directory server, BusinessWare generates concrete WSDL. If the proxy is partitioned on an Integration Server with **Internal Web Server** enabled, BusinessWare generates the host and port of the SOAP address for the proxy based on the settings. If the proxy is partitioned on an Integration Server with **External Web Server** enabled, BusinessWare cannot determine the host and port of the SOAP address for the proxy. In this case, BusinessWare generates a temporary value for the host (“unknown-host”) and a temporary value for the port (“9999”). You must modify the generated WSDL to provide the correct external Web server’s host name and port number.
- If the project is deployed to a file, the host is unknown even when the proxy is partitioned on an Integration Server with **Internal Web Server** enabled. In this case, you must modify the generated WSDL to provide the correct host name of the Integration Server.
- If the project is never deployed, BusinessWare only generates abstract WSDL. That is, the generated WSDL does not include SOAP address information.

Vitria recommends that you deploy the project correctly before exporting the WSDL.

Note: Only the exported WSDL accurately describes how to properly invoke the Web service input proxy. The client should always invoke the Web service input proxy based on the exported WSDL, even if you design the project based on an existing WSDL.

EXPORTING A WEB SERVICE FROM THE PROJECT

To export from the project:

1. From the main menu, select **Tools > Export Web Service....** The Export Web Service Wizard appears.
2. Select the proxies you want to export and specify the export location.

Note: The export directory you specify must already exist. The default directory is a subdirectory of the project directory whose name corresponds to the name of the integration model. That is, files are exported to *ProjectDir/IntegrationModel/ProxyName.wsdl*.

For detailed instructions, see the *BME Help*.

EXPORTING A WEB SERVICE FROM A WEB SERVICE INPUT PROXY

To export from a Web Service Input Proxy:

1. Select a Web service input proxy on a process component or integration component and, from the context menu, select **Tools > Export Web Service....** The Export Web Service Wizard appears.
2. Specify the export location. For detailed instructions, see the *BME Help*.

EXPORTING A WEB SERVICE FROM AN INPUT PORT

To export from an input port:

1. Select an input port on a process component or integration component and, from the context menu, select **Tools > Export Web Service**. The Export Web Service Wizard appears.
2. Specify the export location and protocol configuration. For detailed instructions, see the *BME Help*.

INVOKING A WEB SERVICE USING THE WEB SERVICE STATE

The Web service state is a specialized action state through which you invoke a Web service. For more information about action states, see [Chapter 12, “Process Models: Ports, States, and Transitions.”](#) To help you configure a Web service, BusinessWare provides the Web Service State Wizard, which automates many of the tasks necessary to configure a Web service and requires minimal programming.

In the BME, you typically work in a top-down fashion—that is, you create an integration model before you create process models; you define proxies and ports before you model the states that will use them. The Web Service State Wizard, however, lets you work from the bottom up.

The Web Service State Wizard automates the following actions:

- Invoke Web service.
- Set type for inbound and outbound transitions; passing in parameters to the invocation call.
- Determine how to handle return values from the Web service.
- Create the Web service output proxy and wire ports at the parent level.
- Create the output port and configure the port type.

If your proxies and ports are already established, or if you are a power user who prefers to write your own code, you can configure the Web service state manually using the Action Builder or a Java editor.

USING THE WIZARD TO CONFIGURE A WEB SERVICE STATE

To invoke a Web service using the Web Service State Wizard:

1. In the process model, double-click on the Web service state (or right-click on the Web service state and select **Tools > Configure...**). The Web Service Configuration Wizard appears.
2. Define the Web service by providing the following information:
 - State Name
 - Type
 - Operation
 - Method Invocation
 - Port Invocation
 - Protocol Configuration

For detailed instructions, see the *BME Help*. The wizard is also described in the Web services tutorial. For more information, see “[Sample Files](#)” on page 9-28.

SAMPLE FILES

A sample of the Web services functionality is located in the following directory:

installdir\samples\modeling\WebServiceSample

For instructions on manually creating the project provided in the sample above, including step-by-step instructions for using the Web services wizards, see the Web services tutorial, which is available in the following directory:

installdir\samples\modeling\WebServiceTutorial

Both the sample and tutorial provide examples of the WSDL and Java that is used in BusinessWare’s Web services functionality. In addition, the tutorial provides instructions on how to attach JAX-RPC handlers.

PART IV: PROCESS MODELS

This part describes the role of process models and how to create and use them in your BusinessWare application.

Chapters include:

- [Process Models: Basic Concepts](#)
- [Process Models: Defining and Using Business Objects](#)
- [Process Models: Ports, States, and Transitions](#)
- [Process Modeling Techniques](#)
- [Process Model Code Construction](#)
- [Process Model Templates](#)
- [Workflow](#)

PART #: TITLE HERE GOES HERE

A process model contains the logic for initiating and managing a business process. This chapter presents an overview of how process models are created and used. Topics include:

- [Introduction to Process Models](#)
- [Business Objects Used in Models](#)
- [Creating Process Models](#)
- [Setting Model Properties](#)
- [Adding Action and Condition Code](#)
- [Working with Process Models](#)

For more detailed information on ports, states, transitions, and modeling techniques, see [Chapter 12, “Process Models: Ports, States, and Transitions”](#) and [Chapter 13, “Process Modeling Techniques.”](#) For step-by-step instructions for creating and configuring process models, access the *BME Help*.

INTRODUCTION TO PROCESS MODELS

A process model is a graphical means of defining a business process. Process models embody the business rules by which objects (such as a customer order) are processed across business systems in a well defined, logical manner. For example, a model for a simple order processing system might manage interactions with inventory, billing, and shipping systems. It might even incorporate human interaction, assigning a credit-risk analyst to approve orders over a certain amount.

Process models are event-driven; that is, they define the behavior of an automated system in response to well defined events that occur in the business scenario. Common examples would be receipt of a new order, a quote, or an invoice.

EXAMPLE

As an example, [Figure 10-1](#) shows an order processing model. When this model receives a newOrder event, it begins a multi-step process that includes verifying each item in the order, processing the order, billing the customer, shipping the order, and updating the account. Each step in the process is triggered by an incoming event or operation.

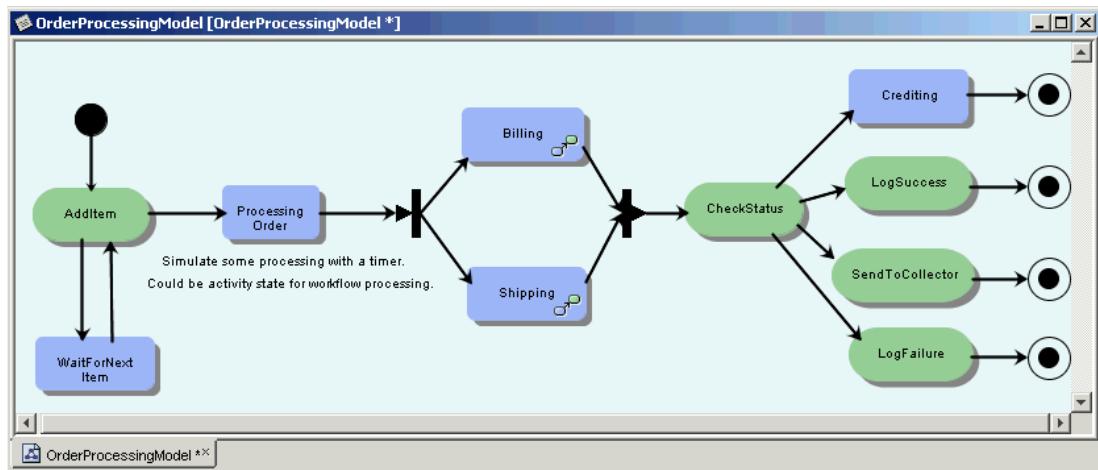


Figure 10-1 OrderProcessingModel from OrderProcessSample Project

STATECHART DIAGRAMS

As illustrated in [Figure 10-1](#), a process model is a statechart diagram based on the Unified Modeling Language (UML).

A statechart diagram specifies:

- Sequence of *states* that a business object goes through as it is processed
- *Transitions* that move it from state to state
- *Events* or *operations* that trigger those transitions

These concepts are described briefly here and are explained in more detail in [Chapter 12, “Process Models: Ports, States, and Transitions.”](#)

States

A *state* represents a distinct phase in a business process. Each state has a unique name and at least one transition leading into or out of it. Often, states have multiple transitions.

Basically, there are two types of states:

- **Resting States**—The process pauses in a resting state and waits for an event or operation to trigger its transition to the next state. Resting states are shown in the model as blue rectangles with rounded corners.
- **Action States**—The process completes the action associated with the state and immediately transitions to the next state. Action states are shown in the model as green ovals.

You can specify a set of actions to be performed as the process enters the state or as it exits the state. These blocks of code are referred to as *entry actions* and *exit actions*. BusinessWare provides tools to help you easily generate this Java code.

For more information, see [Chapter 12, “Process Models: Ports, States, and Transitions.”](#)

Transitions

Transitions control the progression from state to state by specifying:

- The events or operations that can trigger the transition
- Any additional conditions that must be satisfied to trigger the transition
- The actions to be performed during the transition

These three specifications, event, condition, and action, are commonly referred to as the ECA specifications.

A single state may have multiple transitions leading to different successor states. When an event is received, it is evaluated against the ECA specifications of each transition to determine which transition should fire.

Transitions are shown in the model as arrows. Black arrows indicate normal transitions; pink arrows indicate transitions that are triggered by exceptions.

For more information, see [Chapter 12, “Process Models: Ports, States, and Transitions.”](#)

Events

Events are the stimuli that trigger transitions in a process model. An event indicates a significant occurrence in the business environment, such as receipt of a purchase order or issuance of an invoice. You specify which types of events can trigger a given transition when you set that transition's properties.

Typically, you also define one or more interfaces for your project. In each interface, you define one or more related events. For example, a simple order system might require definitions for events such as placing an order or cancelling an order. These interfaces are defined on input and output ports.

For more information on events, see [Chapter 6, “Integration Model Basics.”](#)

Ports

Ports on the model control what event is received and how the event is produced. Ports are the model's gateways to other BusinessWare components and external systems. When you configure a port, you give it a unique name and specify the types of events that it will accept by defining an interface.

For more information on ports, see [Chapter 6, “Integration Model Basics”](#) and [Chapter 12, “Process Models: Ports, States, and Transitions.”](#)

Exceptions

An exception is an abnormal condition raised by an operation that cannot return a result. Like events, exceptions can trigger transitions. Exception transitions are used to provide for compensating action when an abnormal condition occurs. For example, in an order processing model you might provide an exception transition to respond when a customer's request exceeds the current stock in inventory.

For more information on exception handling in BusinessWare, see [Chapter 26, “Exception Handling.”](#)

The Order Process Sample also illustrates how exceptions can be handled effectively and provides a discussion of this topic. The Order Process Sample resides here:

*installdir\samples\modeling\OrderProcessSample\
OrderProcessSample.htm*

For information on transitions in process models, see [“Transitions” on page 10-3.](#)

WORKFLOW MODELS

Some business processes can be performed in a completely automated fashion with no human interaction. A common example is a process for updating inventory as orders are shipped.

Other business processes require the coordination of tasks performed by computers and tasks performed by individuals or groups. For example, a credit manager may have to review and approve purchase orders of a certain amount before the order management system proceeds to check the inventory, bill the customer, and ship the product. Such processes are referred to as *workflow*.

In the BME, you can construct workflow models that define the tasks to be performed, the individuals or groups who will perform the tasks, the deadlines for completing tasks, and the priorities of tasks. Individuals are automatically informed of their pending tasks, and supervisors are notified when a task is not completed on time.

This chapter describes how to create models that do not require the workflow functionality. For a detailed discussion of workflow models, see “[Constructing a Workflow Model](#)” on page 16-5.

STATEFUL AND STATELESS MODELS

A process model is said to be *stateful* if it persists information about the current state of a process and its associated data. A model is *stateless* if it does not persist such information.

Stateless Models

In stateless models, the process executes completely, from start to finish, when invoked by the initial incoming event. The process never comes to rest; stateless models consist only of action states and nested stateless models. As soon as the computation in one state is completed, the process moves to the next state. Hence, there is no need to persist data about the current state. Stateless models typically are used to model data transformations.

Stateful Models

A model is stateful if it satisfies one of the following conditions:

- A model is stateful if the process enters one or more “resting states,” where it waits for an event to trigger a transition to the next state. Consequently, the model must keep track of the location and condition of each process instance over time.

BusinessWare tracks the location and condition of each process instance by storing state information and other data as attributes of a business process object (BPO). The BPO is persisted in a relational database and is accessed as needed when an event is received.

- A model is stateful if it contains one or more nested states and the BPO type is defined.

Stateful models are used to model long-running business processes. They are especially useful when the process involves asynchronous interactions.

For additional information on BPOs refer to [Chapter 11, “Process Models: Defining and Using Business Objects.”](#)

Summary of Differences between Stateless and Stateful Models

In summary:

- Stateless models cannot contain resting states. They consist solely of action states, or nested states which represent nested stateless models. They cannot nest stateful models.
- Stateless models are not associated with a BPO.
- Any model that contains a resting state is a stateful model.
- Stateful models must be associated with a BPO.

PROCESS MODELS AND PROCESS COMPONENTS

Process components are used in integration models and link to a process model. The component shows how the process model fits into the total business solution, and it defines the flow of data into and out of the process model with its configuration of ports and wires. The process model itself is not bound to other components, and it has no knowledge of where the input data it receives came from or where the output data it produces goes.

This approach allows for modularity and flexibility. A single process model can easily be used in multiple solutions, and changing the configuration of other components in the solution does not impact the process model. As long as the ports on the process component and the process model are synchronized, the model can receive, process, and output data.

Process Model Link

The link between a process component and a process model is defined by the Process Model Link property. This property is set at the component level and can be changed if you later decided to use a different model. Remember:

- A process component must be linked to one process model.
- A process model can be linked to one or more process components.
- The link can be changed at any time.

Port Synchronization

The port settings, port name and port type, on the component and on the model must be exactly the same to allow data flow into and out of the model. By default, BusinessWare provides automatic synchronization of ports on process components and their linked process models:

- When you create a new process model and simultaneously link it to a process component, the new model inherits the port configurations from the component.
- When you link an existing process model to a process component, the BME checks the port configurations and asks if you want it to synchronize ports. You can choose to apply the port configurations from the component to the model or vice versa.
- If you add, change, or delete ports on either a process model or a process component, the BME will detect a mismatch between the component and its linked model and ask if you want it to synchronize ports. You can choose to apply the port configurations from the component to the model or vice versa.

For more information on ports, see “[Ports](#)” on page 6-15.

BUSINESS OBJECTS USED IN MODELS

BusinessWare must be able to recognize and manipulate the business objects, events, and exceptions that are used in your process model. Many commonly used interfaces are predefined for you in the BusinessWare project module. However, you may need to define additional interfaces that are specific to your application. Also, if your model uses BPOs to persist data, you must define the BPO.

BPOs

BPOs are used to represent and persist modeling data. Each process model can be associated with only one BPO. Then, for each instance of that particular process, a unique BPO is instantiated. For example, if you are modeling an order-tracking system, each order is represented by a unique “order BPO.”

For additional information on BPOs refer to [Chapter 11, “Process Models: Defining and Using Business Objects.”](#)

DATA OBJECTS

A data object (DO) is an auxiliary object used in combination with a BPO. Like a BPO, a data object contains data (in the form of attributes), but it contains no process state information and usually no methods.

For additional information on DOs refer to [Chapter 11, “Process Models: Defining and Using Business Objects.”](#)

LOCAL DATA

The Local Data interface provides a means for you to store temporary, transient data within your model. Local data is not persisted in a database (as BPO data is). When the current request completes, all local data is deleted. You can reliably use local data across multiple chained action states, for example, to get and hold data from an external system as your process progresses.

For additional information on local data refer to [Chapter 11, “Process Models: Defining and Using Business Objects.”](#)

CREATING PROCESS MODELS

In the BME, you construct process models graphically by inserting and configuring various elements in the Process Model Editor. As you do so, the Java code segments to implement your design are automatically generated. This visual approach makes it easy to design, understand, and maintain process models.

BEFORE YOU START

Before creating a process model, you should complete the following tasks:

- Create a project.
- Create an integration model.
- Configure a process component in the integration model.
- Define the types (event interfaces and operation interfaces).
- Define the business objects used in your model.

Technically, you can create the process model before configuring the process component, and you can define types and BPOs while constructing the model. However, it is more efficient to do these tasks first, if possible.

PROCEDURE

Developing a process model is an iterative process. Typically you work on the overall layout first, adding states and transitions. Then you go back and define the action code. As you work, you continually refine and enhance the model.

The basic procedure for creating a process model is outlined below. Where appropriate, cross-references to additional information are provided. In addition, the *BME Help* offers detailed instructions for using the BME.

This procedure does not include the workflow elements that incorporate human activity into an automated process. For information on creating a workflow process model, see “[Constructing a Workflow Model](#)” on page 16-5.

Tip: Linking the process component and process model propagates the component ports to the model and makes additional methods available to you when you construct code in the Action Builder. Therefore, it’s a good idea to establish the link as early as possible.

To construct a process model:

1. Create an empty process model using any of the following techniques:
 - Right-click on a process component in an integration model, and select **Tools > Open Nested Model**. Click **New....**
 - Select a process component. Use the **Browse** button in the Process Model Link property to bring up the Select Process Model dialog. Click **New....**
 - Double-click on a process component. Click **New....**
 - Right-click on a folder in the Explorer and select **New > Process**.
 - From the main menu, select **File > New...** to display the New Wizard; then open the Models node and choose **Process**.

The first three methods automatically link the new process model to the process component. With the last two methods, you have to manually link the model to the component, as described in [step 6](#).

2. When prompted, provide a name for your process model. No spaces or special characters are allowed in the name; it must be a valid Java class name.
3. Set model properties:
 - a. Click in the background of the Process Model Editor to display the model's property sheet and enter the desired values.
 - b. Be sure to associate a BPO with the model if you need to persist state data (always required for a stateful model). Specifying a BPO automatically generates code that you can later use when creating your action code.
 - c. Specify a Local Data object if you cannot use the default for the Local Data Type property.

See “[Setting Model Properties](#)” on page 10-14 for information.

4. Add states to the model:
 - a. Click on a state button  in the Process tab of the Palette, and then click in the Editor.
 - b. Set the state properties by selecting the state and entering values in the Properties window.

See “[Process Models: Ports, States, and Transitions](#)” on page 12-1 for information on the types of states and their properties.

5. Add transitions from state to state:
 - a. Draw the transition by clicking on a transition button  in the Process tab of the Palette, then on the source state, and finally on the target state.
 - You can create an angled line by clicking at each vertex before clicking on the target state.

- Exception transitions can originate in action and nested states.
- b. Specify what type of event can trigger the transition (or exception transition) and set other properties in the Properties window.
 - c. Specify any conditions that must be met for the transition to occur. Right-click the transition and select **Tools > Condition...** to generate code with the Condition Builder or **Tools > Condition Code** to add code in the Java code editor. You can also launch the Condition Builder from the Condition property in the Properties window for the transition.

See [Chapter 12, “Process Models: Ports, States, and Transitions”](#) for information on the types of transitions and their properties. See [“Adding Action and Condition Code” on page 10-15](#) and the *BME Help* for information on using the Condition Builder.

6. If necessary, manually link the process model to the process component. (If you created the process model by double-clicking on the process component, the link was automatically set, and you do not have to perform this step.)
 - a. Select the process component in the integration model.
 - b. In the Properties window, set the Process Model Link by selecting your new model.
7. If necessary, add and configure ports.

Note: By default, ports are not shown in the process model. To view ports, set the Port property in the Layers tab of the Properties window to Show. Manually adding a port to a process model automatically enables the port layer.

- a. Click on the input port button  or output port button 

See [“Process Models: Ports, States, and Transitions” on page 12-1](#) for information on ports and their properties.

Note: Manually creating ports is necessary only if you are creating a process model that will be nested within another process model (not linked to a process component). If you have linked the model to a process component, the component ports are automatically replicated on the model. See [“Nested Models” on page 13-9](#) for more information.

8. Add action code to states and transitions.
 - a. Specify any actions to be executed as the process instance enters or exits the state. Double-click the state, or right-click the state and select **Tools > Entry/Exit Action...** to generate code with the Action Builder or **Tools > Entry/Exit Action Code** to add code in the Java code editor. You can also launch the Entry/Exit Action Builder from the Entry Action or Exit Action property in the Properties window for the state.
 - b. Add the action code to be executed during the transition. Double-click the transition, or right-click the transition and select **Tools > Condition...** to generate code with the Action Builder or **Tools > Condition Code** to add code in the Java code editor. You can also launch the Action Builder from the Action property in the Properties window for the transition. See [“Adding Action and Condition Code” on page 10-15](#) and [Chapter 14, “Process Model Code Construction.”](#)
9. Save the process model and integration model by selecting **File > Save All**.

Note: If you have models that terminate in a resting state with no outbound transitions, you should modify those models so that the process completes in a terminator state.

CUSTOMIZING THE DISPLAY

You have considerable control over the appearance of your process models.

Local Settings

To customize an *individual process model*, use any of these options:

- **Display Options**—For the model itself and for each element in the model, the Properties window has a Display tab. You can change the background color of the model, add a grid, zoom in or out, change the color of the states, add a logo or other image to a state, or change the line width of transitions. You can also resize states by selecting the state and dragging one of the square handles that appear.
- **Tip:** Activate the Grid Visible and Snap to Grid features in the model’s property sheet to make aligning model elements easy.
- **Layers**—The Layers tab in the model’s property sheet lets you do the following:
 - hide or show annotations
 - hide or show ports
 - hide or show the port types that trigger the various transitions

- hide or show transition details
- **Labels**—Elements in the model are automatically labeled with the names you give them. To edit the label or change its placement, right-click on the object and select **Enable Label Selection**. You can then select the label, move or resize it graphically in the Editor, and change its font, color, or text in the Properties window.
- **Annotation**—Using the Annotation tab of the Palette, you can add labels, boxed labels, images, rectangles, ovals and arrows to your model. For example, you might want to draw an arrow from a terminator state to an output port. The property sheet for each of these annotation elements lets you change the color, change the font or line width, and add a tooltip.

Options

To set display options globally for *all process models*, select **Tools > Options**. In the Options window, select **BusinessWare Options > Process Modeler Options** and make the desired changes. Altering global Options will not change existing models; it will alter new models and new objects created after you change the options.

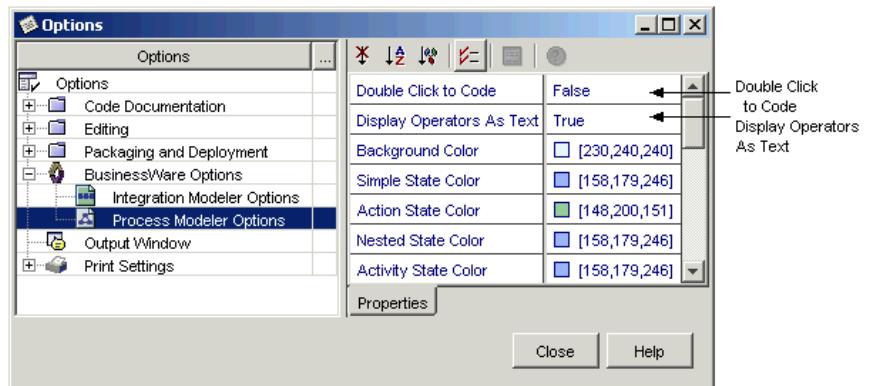


Figure 10-2 Setting Display Options Globally for Process Models

SETTING MODEL PROPERTIES

Properties that control the operation of process models are set at the component level ([Table 10-1](#)), and some are set at the model level ([Table 10-2](#)).

Note: In all property windows, you can display properties arranged in a logical, task-oriented order or in alphabetical order by clicking one of the sort buttons shown below. Throughout this chapter, the task-oriented order is used.

- Click the first button  for task-oriented order.
- Click the second button  for alphabetical order.

To view or set the component properties, select the component in the Integration Model Editor.

Table 10-1 Properties of the Process Component

Property	Description
Name	Name of the process component.
Process Model Link	Link to the process model associated with this component. Each process component must link to one process model.
Local Data Class	The Local Data class is used to hold transient data during the course of a single invocation or request. The data is deleted when the request is completed. Local data is used primarily in stateless models, across a series of chained action states. Enter the fully qualified Java class name. The default value is <code>com.vitria.modeling.runtime.LocalDataPrimitiveTypesMapImpl</code> .
Object Table	The Object Table maps each object implementation class (for the BPO and its associated DOs) to the database table where the BPO data or DO data is persisted. Based on this mapping, the runtime can locate the data for the appropriate BPO instance. This table is filled in by default. For additional information on the object table, refer to Chapter 11, “Process Models: Defining and Using Business Objects.”
Exception Handler Class	The Exception Handler implementation class is used to process component system exceptions and application exceptions that are not caught in the model. Enter the fully qualified Java class name. The default value is blank. It is not necessary to specify a value for this property. For additional information on exception handling, refer to Chapter 26, “Exception Handling.”

To view or set model properties, click in the background of the Process Model Editor.

Table 10-2 Properties of the Process Model

Property	Description
BPO Type	<p>Interface type of the BPO. If this type is specified, conditions and actions can access the attributes and methods of the object. You can specify only one BPO per model. Stateless models do not have to be associated with a BPO.</p> <p>Note: The following IDL event types are not supported:</p> <ul style="list-style-type: none"> • long double • void • any • Object
BPO Name	Name attribute of the BPO. Used as a local variable to refer to the BPO's attributes in action code and condition code.
LocalData Type	<p>Interface type of the Local Data object.</p> <p>The default value is <code>bpe.LocalDataPrimitiveTypesMap</code>.</p>
LocalData Name	<p>Name attribute of the Local Data object; used in source code.</p> <p>The default value is <code>myData</code>.</p>
Imports	List of packages to import; used for generated source code.
Model Supervisor	The user or group that acts as the supervisor for the process. The supervisor is the user who receives notifications. Used with workflow models.
Model Default Action	Action executed when an incoming event causes none of the transitions to fire.
Auto Synchronize Ports	If true, the BME automatically synchronizes the port configurations on the process component and process model. The default is <code>true</code> .
Description	Description of the model; used in generated source code comments.

ADDING ACTION AND CONDITION CODE

You can add action code to a state or transition in two ways:

- **Action Builder**—with this code generator, you simply select the event types and methods you want to use from predefined lists, and then specify parameters for the methods. This tool also includes an editor so you can edit small snippets of generated code.
- **Source Code Editor**—this full-fledged Java code editor gives you the flexibility to write your own code. This editor automatically opens at the line where you should insert your code.

Similarly, you can add a condition to a transition in two ways:

- **Condition Builder**—with this code generator, you build expressions by selecting methods, variables, or event parameters from a list, choosing the operator (equals, less than, etc.), and specifying the required value.
- **Source Code Editor**—the Java code editor gives you the flexibility to write your own condition code. This editor automatically opens at the line where you should insert your code.

[Chapter 14, “Process Model Code Construction,”](#) provides instructions on how to use the Action Builder and Condition Builder.

WORKING WITH PROCESS MODELS

This section describes some ways you can use and monitor process models.

REUSING PROCESS MODELS

You can reuse process models across multiple projects in several different ways.

- You can place your process model (and its associated type information) in a project module for use by other projects.

By creating a project dependency on the project module, you can reference process models within it, from your current project. With this design, you maintain a reference to an existing model, but cannot modify the model itself. If the model changes, your project will automatically run with the new changes. For additional information on projects, see [Chapter 3, “Projects.”](#)

- You can also copy a process model from another project.

Using the design repository, you may browse published projects and copy models from them into your own project.

Note: You must also make sure to copy any associated type information for the process model you copied.

With this design, your current project will have its own copy of the process model, and you can change it to accommodate your requirements. For additional information on projects, see [Chapter 4, “Design Repository.”](#)

- You can make a template from a process model you created.

This allows you to start with an existing model template anytime you want to create a new process model in the BME. You can do this as follows:

- a. Right-click on the model object in the Explorer, and select **Save As Template...**
- b. In the **Save as Template** dialog box, expand the Models node and select a folder for your template.

The template name will be your existing model name, so make sure to name your model with a meaningful name. If you name the model with an existing template name, the new template will be created with the same name with a version appended to the name, for example, newtemplate_1.

- c. Subsequently, any time you select **File > New...**, the customized model is available.

USING PREBUILT PROCESS MODEL TEMPLATES

Templates for several prebuilt process models are installed when you install BusinessWare. These models are provided to help you with common tasks. They include the following:

- **SimpleTransformer**—contains a simple model with a transformer state and allows you to create the transformer model you desire to perform the transformation. It also pushes the output event out to all the output ports defined in the parent process component.
- **RouteByPortName**—routes each event to a single port based on the port name specified in the event parameters.
- **RouteByPortType**—routes each event to all the ports that accept that particular event type.

For more information, see [Chapter 15, “Process Model Templates.”](#)

DEBUGGING PROCESS MODELS

To test your process model, you must build and deploy the project. You then can use the BusinessWare debugger to set breakpoints and trace the execution of your model. At each breakpoint, you can examine the event data and BPO data using the Event Inspector or BPO Inspector.

In a single debugging session, you can debug an individual process model, multiple models, components, or the entire project. See [Chapter 27, “Debugging and Animation,”](#) for more information.

MONITORING PROCESS MODELS

You can monitor a process model to ensure that BPOs are being processed in accordance with your service level agreements (SLAs). To do this, wire a process query component to the process component in the integration model and create a set of queries to be run against the BPO data for the linked process model. When a situation occurs that violates your service level agreements, the process query component detects the violation and sends notification so that corrective action can be taken.

In addition, you can create process views to be used with the Business Cockpit. Process views are used in conjunction with process queries and allow you to visualize the output using a Web front-end.

See [Chapter 17, “Process Query Models”](#) and [Chapter 19, “Process Views”](#) for information.

PRINTING PROCESS MODELS

You can print a hardcopy of the process model diagram or the underlying code.

1. Display the model diagram or the code in the Editor.
2. With the Editor window active, select **File > Print**.

To specify print settings such as page footers or headers, select **Tools > Options**. Select Print Settings to access the property sheet for print settings (see [Figure 10-2](#)).

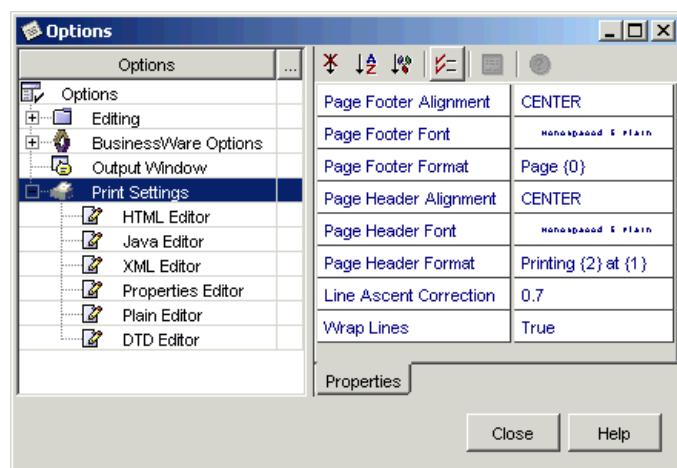


Figure 10-3 Print Settings Property Sheet

The Process Modeler Options and Integration Modeler Options settings contain a Print Fit To Page property. When set to “true”, all components are printed on one page. The component sizes are automatically enlarged or reduced to fit on one page.

PROCESS MODELS: BASIC CONCEPTS

Working with Process Models

A process model contains the logic for initiating and managing a business process. This chapter presents an overview of the business objects used while creating business processes.

Topics include:

- [Using Business Objects in a Process Model](#)
- [Defining Business Objects](#)
- [Performance Consideration for Persistence when Designing Business Objects](#)
- [Adding Business Object Definitions to a Project](#)
- [Coding a Java Implementation for a Business Object](#)
- [Setting Business Object Properties](#)

USING BUSINESS OBJECTS IN A PROCESS MODEL

The following sections describe when to use and how to create business objects in the BME. Three types of business objects are used to define a process model:

- Business Process Object (BPO)
- Data Object (DO)
- Local Data

BPO

In any stateful process model (a model with resting states) the current data for an instance of the model must be persistently stored. This data includes the business process state of the instance (that is, which step of the business process the instance is currently in), plus any user-defined variables that might be necessary to persist the data.

For example, an Order business process might need to persist the order ID, the order amount, and the current state of the order. In BusinessWare, a BPO is defined to persist this information. After the BPO is defined with the associated data values, it is linked to the process model.

As shown in [Figure 11-1](#), a BPO object is instantiated during process model execution. This object instance is automatically persisted in a database and is automatically updated by BusinessWare as needed. A unique identifier, an `oid`, must be assigned to the BPO. The BPO `oid` is then used as the primary key of the object instance for subsequent database lookup and retrieval of the object instance.

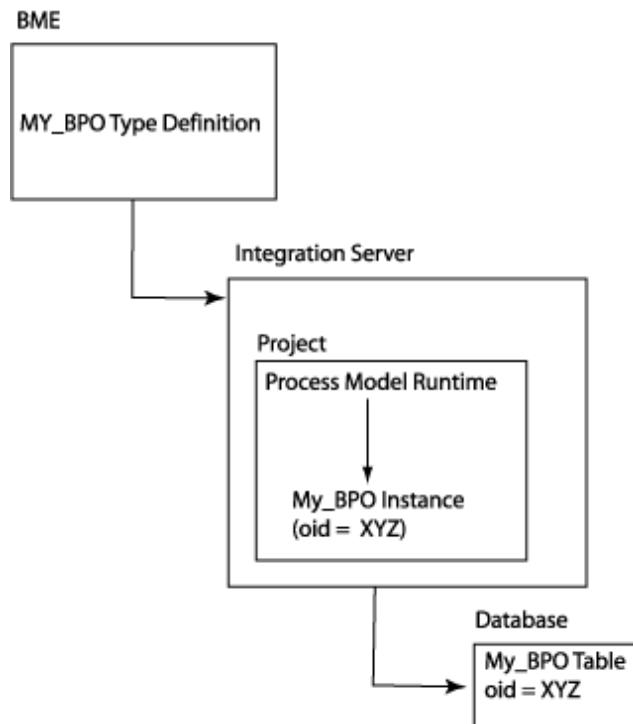


Figure 11-1 BPO Lifecycle: From Definition to Stored Instance

For information on how `oids` are assigned, see [“Request Map Class: Dispatching Events and Assigning BPOs” on page 13-24](#).

In addition to the `oid` attribute, a BPO has a set of attributes and methods that BusinessWare automatically provides. The standard attributes and methods are shown in [Figure 11-2](#). Custom attributes and methods can be added to the base set.

MyBPO

```
Standard BPO Attributes:  
    string oid  
    readonly string modelName  
    StatesGroup bpState  
    string bpVersion  
    ControlState bpControlState  
    long numTransitions  
    TimeStamp timeEntered  
    TimeStamp timeCreated  
  
Standard BPO methods:  
    BPID bpid()  
    string type()  
    void setTimer(in string state,  
        in string model,  
        in Duration duration)  
    void setRelativeToStart(in string state,  
        in string model, in Duration duration)  
    void setAbsoluteTimer(in string state,  
        in string model, in TimerData data)  
    void set AbsoluteRelative(in string state,  
        in string model, in TimerData data,  
        in long days)  
  
myBPO Custom Attributes (optional)  
    string myAttribute1  
    long myAttributeN  
  
myBPO Custom Methods (optional)  
    myMethod1()  
    myMethodN()
```

Figure 11-2 BPO Definition

As shown in [Figure 11-2](#), the custom attributes required for a BPO design must be defined. This definition specifies the persistent data to be stored with the BPO. Custom methods that access custom BPO attributes and/or perform further processing may also be defined. An attribute can be data of any type. For example, primitives, standard BusinessWare types, structs, and sequences of these data types may form an attribute. Care should be used when naming attributes to avoid database specific keywords as the attribute name can be mapped to the database column name.

IMPORTANT: Only one BPO can be linked to each stateful process model. However, multiple process models can use the same BPO definition. Two BPO instances cannot share the same `oid`, even if they have different definitions.

DATAOBJECT

Typically, persisted data is distributed across a BPO, plus one or more *data object* (DOs). A DO is similar to a BPO in that it can be persisted, but a DO does not store the business process state of the process model. However, the DO can be referred to from the BPO as an attribute of the BPO, thus allowing access to the data stored in the DO.

Instead of defining all information in a BPO, a DO definition is useful when information will be re-used across multiple BPO definitions. For example, the same Part DO that is used in an Order business process can be used in a Manufacturing business process since each process refers to the same part information. In some cases, we suggest that DOs be used to store data to increase performance, as described in [“Performance Consideration for Persistence when Designing Business Objects”](#) on page 11-12.

Similar to BPOs, a unique identifier, the `oid`, must be assigned to the DO. The `oid` is used as the primary key of the object instance for purposes of subsequent database lookup and retrieval of the object instance.

IMPORTANT: Two DO instances cannot share the same `oid`, even if they have different definitions. Also, even though a BPO and DO are stored in different tables in a database, the same `oid` cannot be used between any two BPO or DO instances.

LOCAL DATA

As described above, BPOs and DOs are used to persist information about a business process. However, sometimes *transient* data, that is, data that does not need to be persisted, must be defined and used. Transient data is useful in passing information from one action state in a process model to a subsequent action state.

You can see an example of transient data use in the BankModel (Figure 11-3) of the Web Service Sample (`installdir\samples\modeling\WebServiceSample`). The Calculate Rate action state computes an exchange rate, and passes that exchange rate information to the subsequent LookupAccount or UpdateAccount state using a Local Data object.

The Web Service Tutorial

(`installdir\samples\modeling\WebServiceTutorial`) describes how to store information as local data and then retrieve the information later.

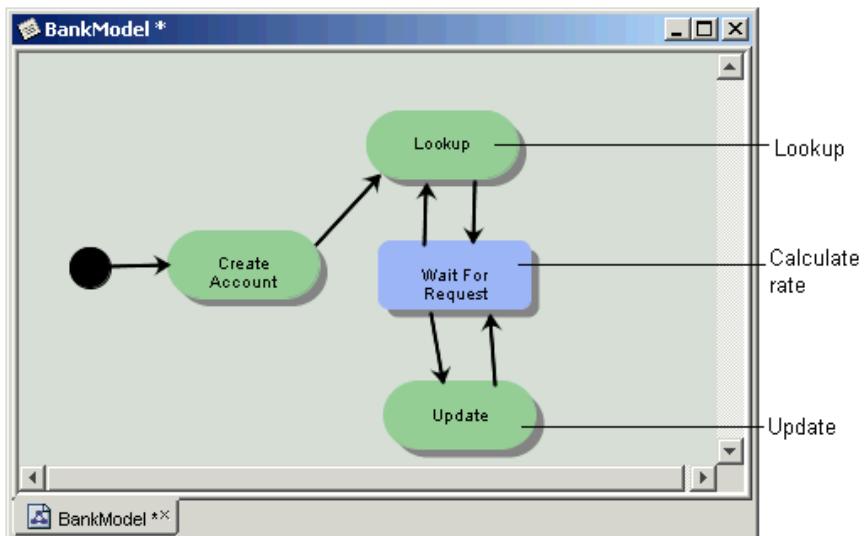


Figure 11-3 Web Service Sample Bank Model

Since Local Data is not persisted in a database, all data is eliminated when the current request on the process model completes. In particular, Local Data information is not retained while a BPO waits in a resting state. Local Data is for use only across multiple chained action states. This makes Local Data extremely useful in stateless models, which consist solely of action states.

Every process model contains a default Local Data definition of type `bpe.LocalDataPrimitiveTypesMap`. The Local Data class is defined in the process model component properties (see [Table 10-1](#)) and the Local Data type is defined in the process model properties (See [Table 10-2](#)). Most process models can simply use this built-in type definition. However, new Local Data definitions can also be used.

To use new Local Data definitions:

1. Define the objects in an IDL file.
2. Add the definitions to the project by right-clicking on the IDL file and selecting **Tools > Convert to Type**.

All the definitions will be added to the project.

IMPORTANT: You should never define an object to be both a BPO or DO and a Local Data object because a BPO or DO is meant to be persisted.

Extending the Local Data Interface

It is possible to extend the Local Data interface. For example, you can create custom classes that extend the `com.vitria.modeling.runtime.LocalDataImpl` interface as the following example shows.

```
public class MyLocalDataImpl extends
    com.vitria.modeling.runtime.LocalDataImpl
    implements <LocalDataType>
{
    // put your variables here, for storing your local data
    public LocalDataMapImpl()
    {
        // need a zero-argument public constructor
    }
    // put your getter & setter methods here
}
```

For example, if you create a type called `com.company.types.OrderLocalDataType`, then add this class as the interface:

```
implements com.company.types.OrderLocalDataType
```

If you are using a custom local data class, then the class must be available to your project at compile-time in the BME and at runtime.

Example: Assume that you have created a project with a process model consisting of several action states for which you do not need to persist data. However, you would like to use information gathered in the first action state in subsequent action states. You can use a local data object to pass the data from one action state to the next.

To use a local data object:

1. Create a DefinedType by either importing the definitions or by creating it in the BME as described in “[Creating Defined Types](#)” on page 5-7.
2. Create a new module under the DefinedType. See the *BME Help* for more information.
3. Create a new LocalData object interface under the module. See the *BME Help* for more information.
4. Create a new operation or attribute under the LocalData object as necessary.
5. Compile the project to generate the types. For more information, see “[Building and Compiling Projects](#)” on page 3-24.
6. In the Explorer, select the process model containing the action states for which local data will be used. In the Properties window, set the **LocalData Type** by browsing to the local types contained within your project. Select the interface under the module that you created in step 3 and click **OK**.
7. In the Explorer, select the integration model that links to the process model selected in step 6. In the Properties window, set the Local Data Class for the new LocalData object. The class should be in the format *module._type*. For example, your Local Data Class could be
`DefinedType._MyLocalDataObject.`

DEFINING BUSINESS OBJECTS

BPOs, DOs, and Local Data can be defined using the structured editor in the BME. However, the editor only allows simple business objects to be defined. In particular, you are restricted to using primitive types as the object attributes when using the structured editor. Note that BusinessWare provides an additional mechanism using CORBA IDL to define complex business objects as shown below.

To access the structured editor, use **New > All Templates...** and click **Types > DefinedType**. For more information on the structured editor, refer to the *BME Help*. For more information on types, see “[BusinessWare Types](#)” on page 5-1.

DEFINING COMPLEX BPO, DO, AND LOCAL DATA

To define complex BPO, DO, or Local Data:

- Create a file for object definitions by selecting **File > New...** and clicking **Other > IDLSource**. Follow the standard naming convention, *myProjectObjects*, where *myProject* is the name of the project.
- Add an include statement in the file to include the BusinessWare definition file, *bpe.idl*. Also add include statements to include any other IDL definition files that the object definition file uses. For example, if the BPO uses an attribute type, *Foo*, which is defined in a different IDL file, *myFoo.idl*, *myFoo.idl* must be included.
- Add the module declaration after the include statement(s). The module name should match the file name. Note that although multiple module declarations per definition file are allowed, one module per file may be the optimum for type management purposes.

The syntax is as follows:

```
#include <bpe.idl>
module moduleName {
};
```

- Add one or more BPO, DO, or Local Data definitions inside the module brackets. Each BPO, DO, or Local Data definition should contain one or more attributes and (optionally) custom methods. The attribute type must be a:
 - Supported primitive type
 - Standard BusinessWare type
 - Type defined in the current file
 - Type defined in an #included file

The example below shows the syntax for BPO, DO, or Local Data with one attribute of a standard BusinessWare type.

```
#include <bpe.idl>
module moduleName {
    interface BPOName: bpe::BusinessProcessObject {
        attribute TypeName variableName;
    };

    interface DOName: bpe::DataObject {
        attribute TypeName variableName;
    };
};
```

```
interface LocalDataName:bpe::LocalData {
    attribute TypeName variableName;
}
};
```

Note: Any BPO definition needs to extend the interface `bpe::BusinessProject` that is defined in the file `bpe.idl`. Similarly, any DO definition needs to extend the interface `bpe::DataObject`, and any Local Data definition needs to extend the interface `bpe::LocalData`.

For a list of IDL primitive types, refer to the CORBA standards specification. BusinessWare types are provided in the BusinessWare project module that is automatically added to each project.

Note: Although a separate definition file for each object can be used, it is advantageous to use a single definition file that contains all BPO, DO, and Local Data used by the project.

DEFINING OBJECTS WITH `Struct` ATTRIBUTES

Complex custom types can be defined as structs for object attributes. Struct data is written to the same database table as the BPO and/or DO data whenever the BPO and/or DO data is written or updated (recall that Local Data is not persisted.)

For example:

```
#include <bpe.idl>
module moduleName {
    struct PO {
        string description;
        string customer;
    };
    interface BPOName: bpe::BusinessProcessObject {
        attribute PO myPO;
    };
};
```

Note: Structs and unions cannot have null values.

DEFINING OBJECTS WITH CUSTOM METHODS

Custom methods definitions for BPO, DO and Local Data can be invoked in custom action code or transition code. However, in addition to defining the methods, they must be implemented as described in the section, “[Coding a Java Implementation for a Business Object](#)” on page 11-15.

The following example shows a BPO with two custom methods defined.

```
#include <bpe.idl>
module moduleName {
    interface BPOName: bpe::BusinessProcessObject {
        attribute string myVar;
        void setSomething(in string s);
        string getSomething();
    };
}
```

IMPORTANT: If a custom BPO method is defined but not implemented, BusinessWare will throw the `AbstractMethodError` runtime error when the method is invoked.

DEFINING OBJECTS WITH DATAOBJECT ATTRIBUTES

Business objects can use `DataObject` as an attribute. For example:

```
#include <bpe.idl>
module moduleName {
    interface BillingRecord: bpe::DataObject{
        attribute string customerName;
        attribute float balance;
        attribute string status;
    };
    interface BPOName: bpe::BusinessProcessObject {
        attribute BillingRecord myRecord;
    };
}
```

Since Local Data objects are not persisted, a BPO and/or DO cannot have a Local Data as an attribute.

DEFINING OBJECTS WITH A SEQUENCE ATTRIBUTE

Sequences are useful for an unknown number of a certain type of data items, such as order items in a customer's order. To define objects with a sequence attribute, a list of items such as the following can be used:

- Simple types
- Standard BusinessWare types
- User-defined types (for example, an attribute in the object or in the custom methods)

For example:

```
#include <bpe.idl>
module moduleName {
    typedef sequence<string> MyStringList;
    interface BPOName: bpe::BusinessProcessObject {
        attribute MyStringList strList;
    };
}
```

LARGE DATA VALUES

To store large data values in the database as attributes of a BPO or DO, store them as a BLOB or a CLOB in the database by first defining a BPO or DO as follows:

```
#include <bpe.idl>
module moduleName {
    typedef sequence<octet> Blob;
    interface BPOName: bpe::BusinessProcessObject {
        // this stores data as BLOB
        attribute Blob data;
        // this stores document as a CLOB
        attribute string<10000> document;
    };
}
```

Whether or not an attribute is stored as a CLOB depends on the maximum length of the attribute (10,000 in the above example) and the maximum length that can be stored in a database for a VARCHAR column. If the maximum length of the attribute exceeds the length of the VARCHAR, then the attribute is stored as a CLOB, otherwise, it is stored as a VARCHAR.

Note: For Oracle databases, BusinessWare currently only supports storing sequence<octet> attributes as Long Raw.

PERFORMANCE CONSIDERATION FOR PERSISTENCE WHEN DESIGNING BUSINESS OBJECTS

When designing an application, it is important to understand exactly how the runtime service maps persistent object data to relational tables. Simplicity of object design must be balanced with speed and efficiency of storage. Use the following guidelines when designing object schema:

- Any struct defined as an attribute of a BPO or DO is flattened in the same table as the object.
- Any sequence defined as an attribute of a BPO or DO is placed in a separate table (called a *sequence* table) with pointers back to the original object so they can be joined.
- The DO attributes of a BPO are stored in a separate table. The BPO contains a pointer to the DO.
- Each sequence of DOs that is an attribute of a BPO is stored in a separate table. A third table (the sequence table mentioned above) maintains the mapping between the BPO instances and their DO instances.

Increasing complexity adds more data tables and may decrease performance. However, if the application has large amounts of data, complex DO classes may provide more control of *when* additional data updates occur and *what* must be updated. This may improve performance as discussed below.

STRUCTS VERSUS DOs

Related data may be stored in a DO or stored as a `struct` in a BPO. It may be easier to define a `struct` to contain a group of related information than to define a DO class. It is common to use a `struct` to pass information in events, so it may be natural to store the same incoming `struct` data in the BPO.

A `struct` in a BPO is stored automatically. It is written to the database every time the BPO state changes. From a coding perspective, it is much easier to use a `struct`, but there are some trade-offs. As a general rule, choose `struct` rather than a DO in the following situations:

- The data set is small, perhaps a few elements. Storing additional columns each time a BPO is committed may not significantly affect performance.

- The data set is constantly changing, so that updates to the data occur on almost every transition. Maintaining the data as a `struct` in the BPO may perform better. Only a single table must be updated. Also, it is certainly easier from a coding perspective.
- Data will not be shared by more than one BPO. Data defined in a `struct` cannot be accessed by more than one object.

Unlike data in a `struct`, a modeler can decide when the data in a DO needs to be written to a database. Typically, one will only do that when that DO changes, and not when any other part of the BPO changes.

As a general rule, choose a DO over a `struct` in the following situations:

- The data set is large. Storing additional columns each time a BPO is committed will significantly affect performance.
- The data set rarely changes. Updates to the data occur infrequently, perhaps only on the initial transition. Maintaining the data as a DO in the BPO may perform better.
- The data will be shared by more than one BPO. Data defined in a DO can be accessed by more than one object.

DOs are persisted in the database only when indicated as such by you. To persist the DOs in the database, you must mark the DO for create, update, or delete in the database according to your design specification. These methods are contained in the context object (the `ctx` object) that is available in all of your model action code. The context object is of type

`com.vitria.bpe.process.ProcessModelContext` and contains the following methods:

- `void createObject(DataObject obj) throws ContainerException;`
- `void updateObject(DataObject obj) throws ContainerException;`
- `void deleteObject(DataObject obj) throws ContainerException;`
- `Storable loadObject(String interfaceName, String id) throws RStoreException, ContainerException;`

For more information on methods, see the *BusinessWare Programming Reference*.

SEQUENCES

Understanding sequence behavior is extremely important because performance problems can occur if the incorrect object design for sequences is used. The above discussion on the trade-offs of `struct` vs. `DataObject` applies to sequences as well. However, it is important to note the following with regard to sequences:

- If a BPO contains a sequence, then each time any element of the sequence is changed, all the entries in the sequence table for that BPO must be re-written. In fact, BusinessWare deletes all the old entries, and then inserts new entries. However, this only happens if:
 - Any element in the sequence is changed
 - A new element is added to the sequence
 - An element is removed from the sequence
- Significant performance degradation can occur if:
 - The number of elements in the sequence grows
 - The sequence is constantly modified

As a general rule, choose sequences of `DataObject` over sequences of `structs` when the number of elements in the sequence can be very large. Even though the sequence table will be updated every time the sequence changes, you maintain control over which DOs in the sequence must be updated in the database; for example, only those that have changed.

INITIALIZING ATTRIBUTES FOR BPOS AND DOs

Each custom attribute in a BPO or a DO must be initialized with a valid value. No attribute should be `NULL` when the BPO or DO is written to the database.

Attributes can be initialized in a constructor or in the action code of the states and transitions.

IMPORTANT: If a sequence attribute that has no elements is referenced, the value may be `NULL`, rather than a zero-length array.

ADDING BUSINESS OBJECT DEFINITIONS TO A PROJECT

After objects are defined in an IDL file, add the definitions to the project by right-clicking on the IDL file. Select **Tools > Convert to Type**. All the definitions will be added to the project.

CODING A JAVA IMPLEMENTATION FOR A BUSINESS OBJECT

[Figure 11-4](#) shows when an implementation Java file *may* be required for an object definition. When a project is built, various stub files and _Schema files are generated automatically. A separate _Schema implementation class file is generated for each BPO, DO, or Local Data defined in the project.

If an object has only attributes and no methods, then a custom implementation file is not needed; the generated _Schema file is sufficient. The following code is from the Customer_Schema.java file. To see the complete file, refer to:

```
$VITRIA\samples\modeling\OrderProcessSample\project\GeneratedTypes\orderProcessObjects.
```

```
// DO NOT EDIT -- Automatically generated with the Vitria IDL
compiler
// DO NOT COPY -- Use of internal APIs used in this code is
not supported
package orderProcessObjects;
//? "Interface" "IDL:orderProcessObjects/Customer:1.0"
public class Customer_Schema extends
    com.vitria.bpe.process.StorableBPO implements Customer,
    com.vitria.rstore.LazyStorable{
    protected com.vitria.rstore.FieldFlags _dirty_flags_ =
com.vitria.rstore.FieldInfoLib.createFieldFlags(this);
    public com.vitria.rstore.FieldFlags getFieldFlags(){
    return _dirty_flags_; }
    public com.vitria.rstore.Database getDatabase() { return
    _db_; }
    protected boolean _dirty_ = false;
    public boolean updated() { return _dirty_; }
    public void updated(boolean __b) { _dirty_ = __b; }
    protected boolean _loaded_ = true;
    public boolean loaded() { return _loaded_; }
    public void loaded(boolean __b) { _loaded_ = __b; }
    protected com.vitria.rstore.Database _db_ = null;
    public void setDatabase(com.vitria.rstore.Database __db)
{__db_ = __db ; }
    protected String firstName_ = null;
    public String firstName(){
        if (!loaded_) {
            try { _db_.loadObject(this);
            } catch (com.vitria.rstore.RStoreException __ex)
{
                throw new
com.vitria.rstore.RStoreError(__ex.getMessage(), __ex);
            }
        }
    return firstName_;
```

PROCESS MODELS: DEFINING AND USING BUSINESS OBJECTS

Coding a Java Implementation for a Business Object

```
        }
        public void firstName(String __o) {
            if (!loaded_) {
                try { _db_.loadObject(this);
                } catch (com.vitria.rstore.RStoreException __ex)
{
                throw new
com.vitria.rstore.RStoreError(__ex.getMessage(),__ex);
            }
            firstName_ = __o;
            _dirty_ = true;
            _dirty_flags_.makeDirty("firstName");
        }
protected String lastName_ = null;
public String lastName(){
    if (!loaded_) {
        try { _db_.loadObject(this);
        } catch (com.vitria.rstore.RStoreException __ex)
{
            throw new
com.vitria.rstore.RStoreError(__ex.getMessage(),__ex);
        }
    }
    return lastName_;
}
```

However, custom methods can be added as needed when a BPO is defined. If custom methods are added, a custom implementation class (a subclass of the `_Schema` class) that defines those methods must be created. In addition, the `_Schema` implementation class for the BPO, DO, or Local Data in the implementation file must be extended. The BME provides a facility for this operation:

- Select **File > New...** and choose **JavaSource**.
- Make sure this class extends the corresponding `_Schema` class.
- Enter the Java statements that define the method in the structured editor.

For information on accessing BPO, DO, and Local Data methods in action code, see [Chapter 14, “Process Model Code Construction.”](#)

MyBPO Interfaces (IDL Definition)

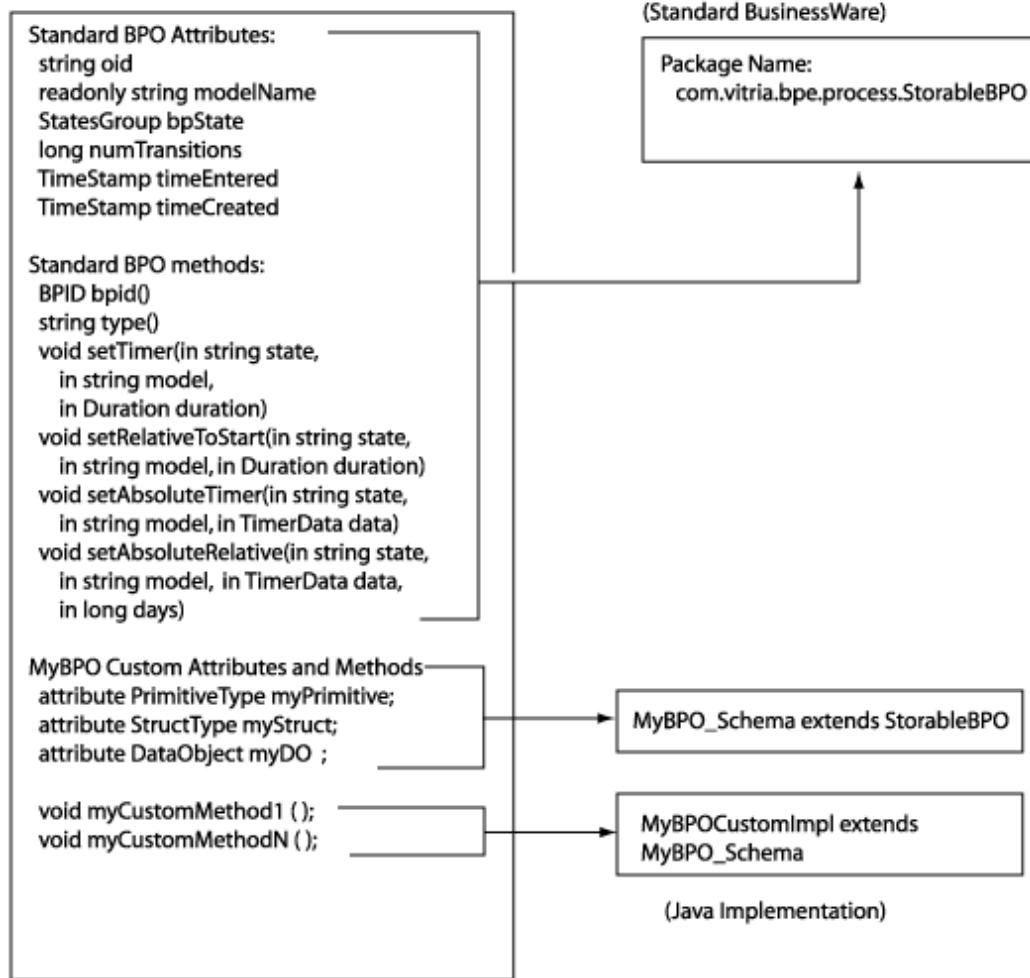


Figure 11-4 BPO Definition and Java Implementation

SETTING BUSINESS OBJECT PROPERTIES

To use business objects in a project, set both process component and process model properties. See the *BME Help* for further description of how to set these properties.

PROCESS COMPONENT PROPERTIES: OBJECT TABLE AND LOCAL DATA CLASS

The process component contains properties that define implementation classes for BPOs, DOs, and Local Data. The process component properties are shown in [Figure 11-5](#). The BPOs and DOs are defined in the Object Table property. This information maps implementation classes to the database tables and is used by the runtime to persist the data in a database. Local Data does not require any database table and has a separate property, Local Data Class.

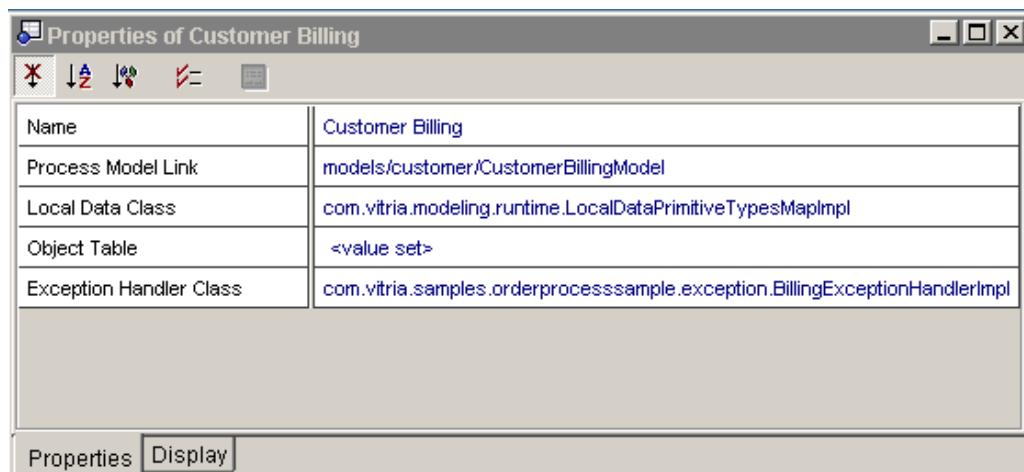


Figure 11-5 Process Component Property Sheet

Clicking the browse button in the Object Table property opens a table in which the implementation classes and table names for BPOs and DOs are specified. For example, the Object Table in [Figure 11-6](#) shows the Customer_Schema class, a BPO, and the Account_Schema class, a DO.

IMPORTANT: For stateful components, the first Object Table entry must be the BPO that is used to maintain the state information. All subsequent entries are for the auxiliary DOs used by the component. For stateless components, all entries are for DOs. A given implementation class can only be mentioned once.

BusinessWare only supports one-to-one mapping between the implementation class and the table name in a single schema. However, you can map the same DO implementation class to different tables in the same schema.

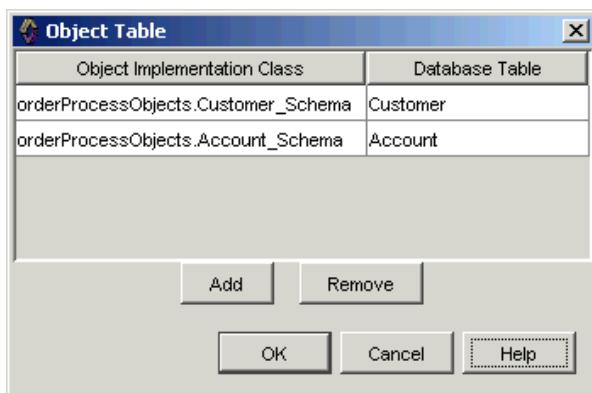


Figure 11-6 Object Table

PROCESS MODEL PROPERTIES: BPO AND LOCAL DATA INTERFACES

The process model defines the interface type and name to be used in the generated code for BPOs and Local Data. The properties, shown in [Figure 11-7](#), are:

- BPO Type
- BPO Name
- LocalData Type
- LocalData Name

The interfaces specified for BPO Type and Local Data Type must be implemented by the classes specified in the parent process component, as defined in the Object Table and the Local Data Class properties.

DOs are not part of generated code and are not defined in a property sheet. For information on using the BPOs, DOs, and Local Data in process model action code, refer to [Chapter 14, “Process Model Code Construction.”](#)

PROCESS MODELS: DEFINING AND USING BUSINESS OBJECTS

Setting Business Object Properties

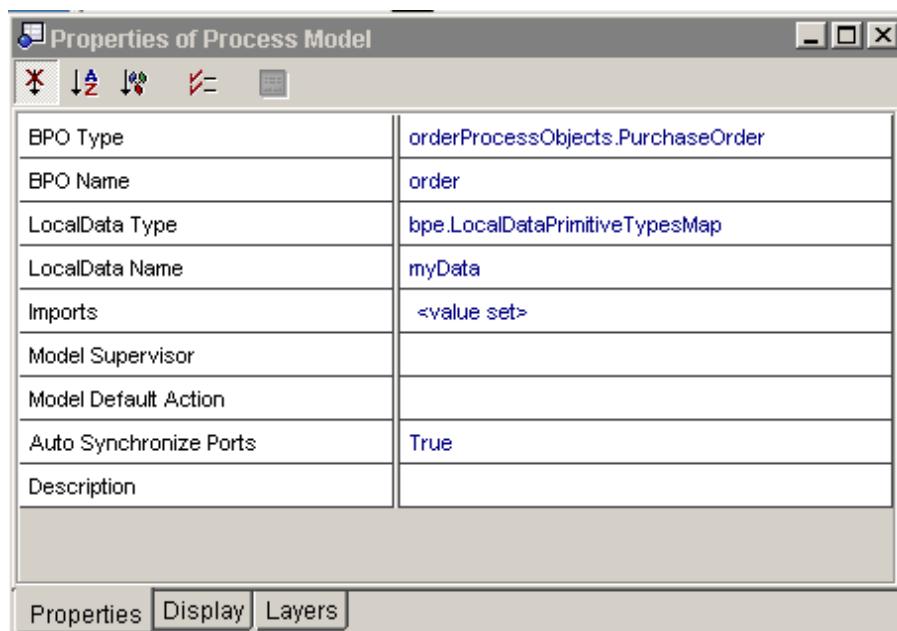


Figure 11-7 Process Model Property Sheet

This chapter provides additional information on each of the elements that comprise a process model. Topics include:

- Ports
- States
- Transitions
- Default Transitions and Model Actions
- Example of a Stateful Process Execution

PORtS

A process component, you recall, is an object in an integration model that links to a process model. Both the component and the model have a set of ports, which must match.

Ports function as the entry and exit points to the component and to its linked model. The *port type* specifies the interfaces that port supports and thus the kinds of information that can flow through it. In the integration model, *wires* connect the output port of one component to the input port of another component, thus indicating the flow of data between components.

You can also configure the Invoke Permission property for a port that authorizes access to the port. For a full description of ports, see “[Ports](#)” on page 6-15. Port properties, including Invoke Permission, are listed in [Table 6-3 on page 6-16](#).

INPUT AND OUTPUT PORTS

Input ports receive events. *Output* ports push events to other components.

Each input and output port on a process component must have a corresponding port on the process model that is configured exactly the same. This pairing of ports is essential because having an output port in the process model causes code generation that enables you to send events out to the port (i.e., using the `getMyPort()` methods). See the Order Process Sample for examples.

The BME automatically synchronizes the ports on process components and process models for you. If it detects a mismatch, the BME displays a dialog box asking if you want to synchronize ports based on the model's port configuration or based on the component's port configuration.

Automatic port synchronization is activated by default for all process models except the prebuilt process model templates (see [Chapter 15, “Process Model Templates”](#)). You can turn off this feature for an individual model by setting the model's Auto Synchronize Ports property to false.

QUERY PORTS

Query ports are special output ports used to send data to a process query component that is monitoring the performance of the process model. The query port on a process component is wired to the input port on a process query component.

Unlike input and output ports, query ports on a process component have no counterpart on the process model itself.

For additional information, see [Chapter 6, “Integration Model Basics.”](#)

GUIDELINES FOR CONFIGURING PORTS

Follow these guidelines for configuring ports on process components and process models:

- You can add as many input and output ports as you need.
- You should define a single interface per port. If you need more interfaces, add ports.
- You cannot mix Java and IDL interfaces on a port. You cannot specify more than one Java interface on a port.
- Each input/output port on a process component must have a corresponding port on the process model and the names of these ports must match.
- The query port on a process component has no counterpart on the linked process model. You cannot attach query ports to a process model.
- The query port on a process component must be wired to the input port on a process query component.
- Query ports have one interface: `processanalyzer.BPOTransition`. You cannot specify any other interfaces.

Note: Do not be confused by the port's Type Propagation property. This property controls whether the types associated with the port will be automatically associated with the port it is wired to in the integration model. It has nothing to do with synchronizing port configurations from a component to a model.

For more information on configuring ports, see “[Ports](#)” on page [6-15](#) and the *BME Help*.

STATES

Basically, a state is a distinct phase in the business process. States can have associated actions that are executed as the process enters or exits the state, and they have properties that control their behavior.

TYPES OF STATES

There are two broad categories of states:

- **Resting States**—process waits in the state for an event to trigger a transition to the next state. Resting states are used, for example, when your business process requires data input from an external system.

In the process model, resting states are shown as blue rectangles with slightly rounded corners as shown in [Table 12-1](#).

- **Action States**—process transitions immediately to the next state. Action states are used when the process can continue without any additional external input.

In the process model, action states are shown as green ovals as shown in [Table 12-1](#).

[Table 12-1](#) describes the various states you can use in constructing your process model and indicates whether you can specify action code to be executed as the business process enters or exits the state.

Table 12-1 States Used in Process Models

Icon	Type of State	Description	Action
	Start State	<p>An initial state in a process model. A start state is not a resting state; an outgoing transition must fire immediately.</p> <p>Every model must have at least one start state, and models may have more than one start state for simultaneous processing of the same BPO. See “Concurrent Processing in Models” on page 13-22.</p> <p>A start state has no transitions leading into it and one or more transitions leading out of it.</p>	none
	Terminator State	<p>An end state in a process model. A terminator state signals the completion of the model or of a concurrent processing flow in the model.</p> <p>In a nested process model, a terminateEvent is sent from the terminator state to the parent model. This event may cause the automatic exit of the nested parent state, or it may contain a value that the parent state evaluates to determine which state to transition to.</p> <p>Every model must have at least one terminator state; complex models may have multiple terminator states.</p> <p>A terminator state has at least one transition leading into it and no transitions leading out of it.</p>	none
	Simple State	<p>A resting state where the BPO waits for some triggering event to cause it to transition to the next state.</p> <p>A simple state has at least one transition leading into it and at least one transition leading out of it.</p>	entry action exit action

Table 12-1 States Used in Process Models (Continued)

Icon	Type of State	Description	Action
	Action State	<p>A state that executes a set of actions and then transitions immediately to another state. With action states, you must provide for an immediate transition out of the state <i>in all circumstances</i>. Use default transitions and default exception transitions to cover all possibilities. See “Default Transitions” on page 12-17.</p> <p>Models for data transformation are typically composed of action states. You also can use action states to model decision nodes, to build decision trees, to manage concurrent model processing, and to handle exceptions.</p> <p>An action state has at least one transition leading into it and at least one transition leading out of it.</p>	entry action
	Activity State	<p>Activity states are used to incorporate workflow activities—tasks that must be performed by individuals—into a process model. An activity state defines the activity to be performed, the role (performer) that will perform the activity, the supervisor of the activity, and other aspects, as appropriate, such as deadline, priority, and cost.</p> <p>An activity state has at least one transition leading into it and at least one transition leading out of it.</p> <p>See “Constructing a Workflow Model” on page 16-5.</p>	entry action exit action
	Nested State	<p>A state that contains a link to a separate process model. Using submodels allows you to abstract lower-level details from the parent model and to package individual subprocesses so that they can be easily reused.</p> <p>A single nested state can route BPOs to any of several submodels, depending on the type of triggering event and conditions.</p> <p>An nested state has at least one transition leading into it and at least one transition leading out of it.</p> <p>Typically a nested state is used in stateful models as a resting state. However, nested states can also be used in stateless models where the nested models enter and exit the nesting in one external triggering call.</p> <p>See “Nested Models” on page 13-9.</p>	entry action exit action

PROCESS MODELS: PORTS, STATES, AND TRANSITIONS

States

Table 12-1 States Used in Process Models (Continued)

Icon	Type of State	Description	Action
	Web Service State	A state that allows clients to programmatically access services over the Internet. Right click to select the Tools > Configure... context menu item and launch the Web Services wizard. See Chapter 9, “Web Services.”	entry action
	Transformer State	A state that contains a link to the BusinessWare Transformer, which allows you to perform data transformation and manipulation by mapping fields in the input event to fields in the output event. Transformer states support one-to-many transformations by wrapping multiple outgoing events in a special wrapper event, eventSequence. A transformer state has at least one transition leading into it and at least one transition leading out of it. See the <i>BusinessWare Transformer Guide</i>.	none
	Advanced XQuery Transformer State	A state that contains a link to the Advanced XQuery Transformer, which allows you to graphically define and execute transformations on XML. For more information please see the <i>Advanced XQuery Transformer Guide</i> .	none
	Fork	A “pseudo state” that splits a single task into multiple parallel tasks. A fork enables a source state to have simultaneous transitions to multiple destination states. The output transitions from a fork fire as soon as the input transition fires. A fork has one transition leading into it and multiple transitions leading out of it. See “Forks” on page 13-5.	none
	Conditional Fork	A conditional fork enables a model to transition out of a fork state based on the trigger type and condition code of each of the outbound transitions. See “Conditional Forks” on page 13-6.	none
	Join	A “pseudo state” that reunites multiple transitions from multiple parallel tasks into a single task. A join completes only when <i>all</i> the transitions into the join have fired. A join has multiple transitions leading into it and one transition leading out of it. See “Joins” on page 13-7.	none

Note: You can customize the icon for a state by using the Display tab in the Properties window. You can change the background color, add an image, or turn off the shadow border.

ENTRY ACTIONS AND EXIT ACTIONS

In a process model, you can attach action code to states. States may have entry actions and/or exit actions:

- **Entry action**—Java code that is executed immediately before a process instance enters the state. First the action code of the incoming transition is executed; then the entry action is executed. Use entry action when there is an action that must *always* be performed when a state is entered, regardless of which transition fired.
- **Exit action**—Java code that is executed when an event triggers a transition out of the state. First the exit action is executed; then the action code of the outgoing transition is executed. Use exit action when there is an action that must *always* be performed when a state is exited, regardless of which transition fired.

The BME provides three facilities to help you generate action code:

- **Action Builder**—lets you generate code simply by selecting events, methods, and parameters from lists. It also provides an editor and Action Picker to assist you in writing snippets of code. To access the Action Builder, right-click a state and select **Tools > Entry/Exit Action....** Alternatively, click in the Entry Action or Exit Action field of the Properties window to display the Action Builder.
- **Action Editor**—assists in writing code by supplying method names; requires a minimal knowledge of Java syntax. As you write your Java statements, you can choose and insert methods from the Action Picker.
- **Source Code Editor**—is a full-featured Java code editor. When you right-click a state and then select **Tools > Entry/Exit Action Code**, the Source Code Editor opens at the line where your entry or action code should be inserted.

For more information, see [Chapter 14, “Process Model Code Construction,”](#) or access the *BME Help*.

STATE PROPERTIES

Table 12-2 describes the state properties and indicates which properties apply to which states.

Note: In all property windows, you can display properties arranged in a logical, task-oriented order or in alphabetical order by clicking one of the sort buttons shown below. The task-oriented order is the default and is used in this book.

- Click the first button  for task-oriented order.
- Click the second button  for alphabetical order.

Table 12-2 Properties of States (Excluding Activity States)

Property	Description	Start State	Terminator State	Simple State	Action State	Nested State	Web Service State	Transformer State	Fork State	Conditional Fork State	Join State
Name	Name of the state.	X	X	X	X	X	X	X	X	X	X
ID	Generated ID of the state, read-only.	X	X	X	X	X	X	X	X	X	X
Timer	A timeout value indicating when the object must transition to the next state. This value can be relative (amount of time after entering the state) or absolute (a specified date and time).			X		X					
Nesting Specification	Table specifying which submodels are linked to this nested state. The decision of which submodel to use is made at runtime, based on the type of triggering event and its conditions. See “ Nested Models ” on page 13-9.					X		X			
Entry Action	Action code that is executed <i>after</i> the action code of the incoming transition and <i>before</i> entering the state. Entry action code must be valid Java statements and can refer to the BPO attributes (in stateful models) or LocalData (typically in stateless models) or to port methods. For instructions on setting action code, see Chapter 14, “Process Model Code Construction.”			X	X	X	X				

Table 12-2 Properties of States (Excluding Activity States) (Continued)

Property	Description	Start State	Terminator State	Simple State	Action State	Nested State	Web Service State	Transformer State	Fork State	Conditional Fork State	Join State
Exit Action	<p>Action code that is executed <i>after</i> an event triggers a transition out of the state and <i>before</i> the action code of the transition is executed. Exit action code must be valid Java statements and can refer to the BPO attributes (in stateful models) or LocalData (typically in stateless models).</p> <p>For instructions on setting action code, see Chapter 14, “Process Model Code Construction.”</p>			X		X					
Termination Value	String value for evaluation on parent model terminateEvent.		X								
Description	Description of the state, to be used in generated source code comments.	X	X	X	X	X	X	X	X	X	X

Activity states have many of the same basic properties as all other states plus several properties that control how a task is to be performed. For information on activity state properties, creating activity states, and using activity states, see [Chapter 16, “Workflow.”](#)

ADDING AND CONFIGURING STATES

You add states to a process model by clicking on the appropriate state button in the Process tab of the Palette and then clicking in the Editor.

You configure states by setting state properties and, optionally, by specifying actions to be performed as the process enters or exits the state.

- **Properties**—to view and set properties, use either of these techniques:
 - Select the state to display its properties in the Properties window. This window shows properties for the selected item, and its contents change when you select a different item.
 - Right-click the state and select **Properties** to display its properties in a separate Properties window. The state properties remain posted in this window until you close it.
- **Action Code**—to generate code for entry or exit actions, use any of these techniques:
 - Right-click the state and select **Tools > Entry/Exit Action...** to use the Action Builder to generate your code.
 - Right-click the state and select **Tool > Entry/Exit Action Code** to use the Source Code Editor to write your code.
 - Click in the Entry Action or Exit Action field of the Properties window to display the Action Builder, or double-click on the state to view its entry actions.

For more information, see [Chapter 14, “Process Model Code Construction.”](#)

TRANSITIONS

A business process changes state over time according to a set of well-defined rules, which are embodied in the transitions.

HOW TRANSITIONS WORK

Transitions specify how the process instance reacts to events received by the model. This specification has three parts—trigger type, condition, and action:

- **Trigger Type**—defines the type of event that can trigger the transition.
A trigger is a special grouping of data within BusinessWare that indicates a significant occurrence outside the model and carries with it the data related to that occurrence.
Events and operations can trigger a normal transition.
Exception transitions can be triggered by exceptions.
- **Condition**—defines a set of conditions that must be satisfied by the incoming data in order for the transition to occur.
Conditions are typically used as a decision parameter. A state may have multiple transitions leading out of it, all of which are triggered by the same type of event. *Which* transition to trigger is determined by evaluating the condition specifications of each transition. The transition whose condition specification is met is the one that is triggered. If conditions on two transitions originating from the same state evaluate to `true`, an error occurs.
- **Action**—defines a set of atomic tasks that are performed when the transition fires and before the process enters the next state.

When an event is received, the specifications are evaluated and executed in the following order:

1. The event type is evaluated against each transition leading out of the current state.
2. If the event is found to trigger the transition, then the transition's condition specifications are evaluated.
3. If the transition's condition specifications are met, its action specifications are performed.
4. The transition completes, and the process instance enters the next state.

Transitions

Example: As shown in [Figure 12-1](#), an event, called Event1, can trigger three possible transitions, each leading to different states. Each transition has different condition specifications, and the conditions determine whether the process transitions to State A, State B, or State C in response to Event1. For any other event type, the process transitions to state Default.

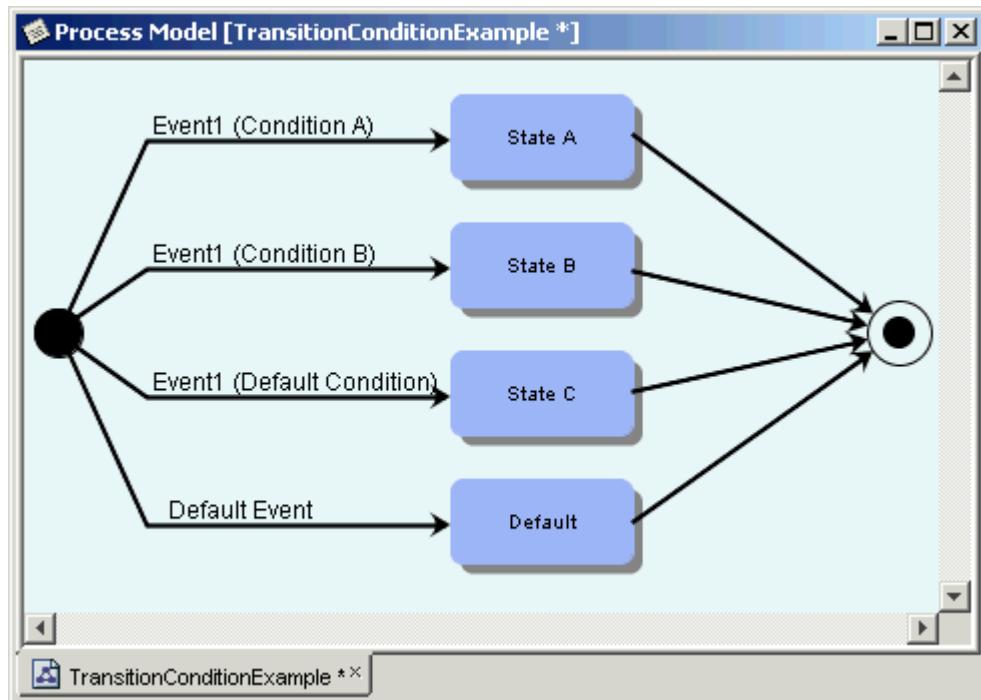


Figure 12-1 Transition Conditions Used in Evaluation

Note: A single event or operation may satisfy the requirements for more than one transition, but only one transition can fire. In this case, all the possible transitions are evaluated in the order described in “[Deciding Which Transition Triggers](#)” on page 12-18.

TYPES OF TRANSITIONS

Basically, there are two types of transitions:

- **Normal transitions**—triggered by an event. They are indicated in the model by a black arrow.
- **Exception transitions**—triggered by an exception event. They are indicated in the model by a pink arrow. Exception transitions can originate in an action or a nested state.

Two commonly used normal transitions are the timeout transition and the self-transition:

- **Timeout transitions**—triggered by the passage of time. You must set a timer on the state and configure the transition to be triggered by a `timerEvent`.
- **Self-transitions**—originate and end in the same state. They are useful for iterative calculations or recurring actions. When a self-transition occurs, any actions associated with the transition are executed before the process re-enters the state, and any timer associated with the state is reset.

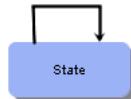


Figure 12-2 A State with a Self-Transition

TRANSITION PROPERTIES

[Table 12-3](#) describes the properties associated with transitions and exception transitions. [Figure 12-3](#) displays the transition window.

Table 12-3 Transition Properties

Property	Description	Normal Transition	Exception Transition
Name	Name of the transition.	X	X
ID	Generated ID of the transition, read-only.	X	X
Trigger Type	Type of event that triggers the transition. See Figure 12-3 .	X	
Exception Trigger Type	Type of exception that triggers the transition.		X

Table 12-3 Transition Properties (Continued)

Property	Description		Normal Transition		Exception Transition
Default Condition	If set to true, there are no qualifying conditions required for the event to trigger the transition. That is, the “default condition” accepts all conditions in the data. If set to false, there may be qualifying conditions specified in the code.		X		X
Condition	Condition code for the transition. Clicking in the field displays the Condition Builder.		X		X
Action	Action code for the transition. Clicking in the field displays the Action Builder.		X		X
Description	Description of the transition, used in generated source code comments.		X		X

ADDING AND CONFIGURING TRANSITIONS

You add transitions by clicking first on the transition button in the Process tab of the Palette, then on the source state, and finally on the target state. The transition does not have to be a straight line; simply click at each vertex before clicking in the target state to draw a line with angles.

Tip: Press the **Shift** key and click while drawing a wire to create right-angled, segmented wires. To remove a segment, press the **Ctrl** key and select the segment origin.

You configure transitions by specifying the triggering event, defining any required conditions, and specifying actions to be performed during the transition.

- **Trigger Type**—in the Properties window, specify which type of event or operation can trigger the transition and set other properties as defined in [Table 12-3](#).
- **Conditions**—specify any conditions that must be met for the transition to occur, using the Condition Builder or the Source Code Editor. See [Chapter 14](#), “Process Model Code Construction.”
- **Actions**—generate the action code to be executed during the transition, using the Action Builder or the Source Code Editor. See [Chapter 14](#), “Process Model Code Construction.”

Figure 12-3 shows how to set the Trigger Type property.

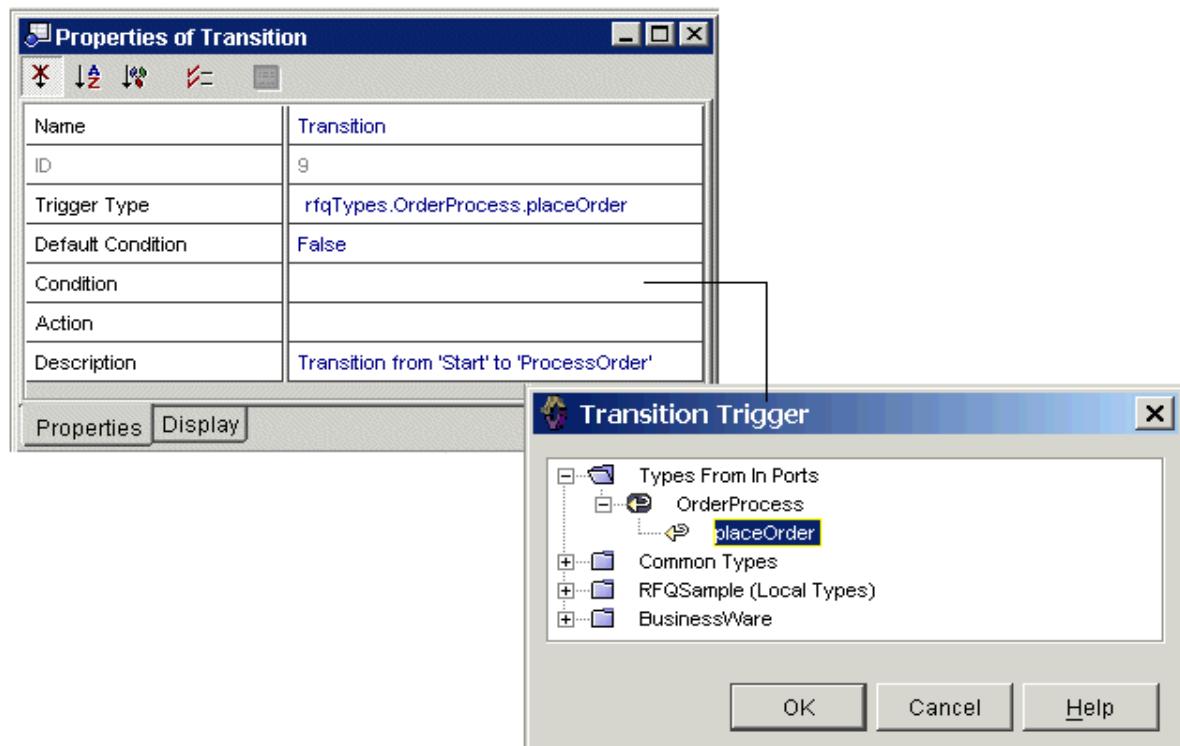


Figure 12-3 Setting the Transition Trigger Type

SPECIAL REQUIREMENTS FOR ACTION STATES

With action states, you must provide for an immediate transition out of the state *in all circumstances*. The process instance cannot rest or pause in an action state. Use default transitions and default exception transitions to cover all possibilities. See “[Default Transitions](#)” on page 12-17 for more information.

You can trigger exception transitions with exceptions you define and are accessible to your project. In addition, you can use any existing Java exception. By creating more than one exception transition from an action state, you can provide appropriate compensating actions for different types of failure conditions.

IMPORTANT: If there are too many transitions in a row (without coming to a resting state), `StackOverflowException` errors are possible. This can happen if you have a loop (a self-loop, or a loop that iterates through several states) that includes no resting states and executes many times (such as a self-loop on an action state, where the action increments a counter, and the loop goes until the counter hits 1000). The model should not be used to implement loops without any resting states in the loop.

For more information on exception handling, refer to [Chapter 26, “Exception Handling.”](#)

DEFAULT TRANSITIONS AND MODEL ACTIONS

To provide appropriate response to all events or exceptions that the model might receive, you can use default transitions and a default model action.

DEFAULT MODEL ACTION

A model default specifies an action to be taken if an event does not cause any transition to trigger. That action could include performing additional computations or throwing an exception if the event was received in error. Note that the default always triggers an action; it cannot cause any transition to trigger.

A process model can have only one model default action.

To specify the model default, right-click in the model background, and select:

- **Model Default Action** to display the Action Builder
- **Model Default Action Code** to display the Source Code Editor

Note: The mere fact that no transitions can be triggered does not cause an exception to be thrown unless the process is in a start state. In any resting state, the process can remain indefinitely. In such a case, an exception is thrown only if you have defined a model default action to trap errors.

DEFAULT TRANSITIONS

A default transition is designed to fire when an event or exception triggers no other transition out of the state. You create a default transition by configuring the transition to accept all events, to accept all conditions, or both. Two transition properties are involved:

- **Trigger Type**—if set to `defaultEvent`, any event can trigger the transition, provided it meets the condition requirements.
- **Default Condition**—if set to `true`, any condition is acceptable for triggering the transition, provided the event is of the required type.

You can use any combination of settings to get precisely the result you want. For example, if you set Trigger Type to `stringEvent` and Default Condition to `true`, any `stringEvent` that does not cause another transition to fire will trigger this default transition. If you set Trigger Type to `defaultEvent` and Default Condition to `true`, any event that does not cause another transition to fire will trigger this default transition.

The same logic applies to exception transitions, where you can set Exception Trigger Type to `defaultEvent` or to a specific type and Default Condition to `true` or `false`.

IMPORTANT: The `defaultEvent` is only a placeholder for an actual event in a process model. You should not create a `defaultEvent` to use in situations where a real event should be used (for example, when injecting events into a model or as a type for a port).

[Table 12-4](#) summarizes how you can control when a transition triggers using various combinations of these property settings.

Table 12-4 Property Settings for Non-Default and Default Transitions

Transition Properties	Transition Fires If . . .
Trigger Type = a particular event type Default Condition = <code>false</code>	The event is of the correct type and satisfies the conditions specified for this transition.
Trigger Type = a particular event type Default Condition = <code>true</code>	The event is of the correct type and all other transitions for this event depend on a condition that is not met.

Table 12-4 Property Settings for Non-Default and Default Transitions

Transition Properties	Transition Fires If . . .
Trigger Type = defaultEvent Default Condition = false	The event is of any type and satisfies the conditions specified for this transition and no transitions for the particular event type fired.
Trigger Type = defaultEvent Default Condition = true	The event is of any type, all other transitions depend on a condition that is not met, and no transitions for the particular event type fired.

To create a default transition, simply select the transition in the Editor and set the Trigger Type and Default Condition properties in the Properties window. Keep the following points in mind:

- You can create more than one default transition leading out of a state. However, only one transition with the combination of `defaultEvent` and Default Condition is allowed.
- Exception transitions can originate in action and nested states.

DECIDING WHICH TRANSITION TRIGGERS

During runtime, an incoming event is checked against the transitions leading out from the current state. This evaluation begins with the most restrictive transitions (that is, transitions that specify both a particular event type and conditions) and proceeds through progressively less restrictive transitions. As soon as a match is found, the evaluation stops and the eligible transition is triggered.

The order for checking transitions is as follows:

1. Test transitions with events and conditions defined. Neither default events nor default conditions are used.
2. Test transitions with defined events and default conditions.
3. Test transitions with default events and defined conditions.
4. Test transitions with default events and default conditions.
5. If the event matches none of the transitions, the default model action is executed.

USE OF THE DEFAULTEVENT ON CHAINED ACTION STATES

As described earlier, the `defaultEvent` is a special type of wrapper event defined on a transition to mean “all other events.” It is used when you need to make decisions such as if `event1` (`e1`) occurs then do special processing, otherwise do something different. The `defaultEvent` is defined on the transition instead of a particular event you may have defined. However, in your action code, you have access to the real event. To access the real event, you should use the following code:

```
EventBody event = ctx.getEvent();
if (event.getName().equals("defaultEvent")) {
    EventBody realEvent =
        (EventBody) event.getJavaParameters()[0];
    ...
}
```

This continues throughout all subsequent chained action states as shown in [Figure 12-4](#). It is even possible to switch back and forth between the use of defaults and the real event underneath.

IMPORTANT: The `defaultEvent` is only a placeholder for an actual event in a process model. You should not create a `defaultEvent` to use in situations where a real event should be used (for example, when injecting events into a model or as a type for a port).

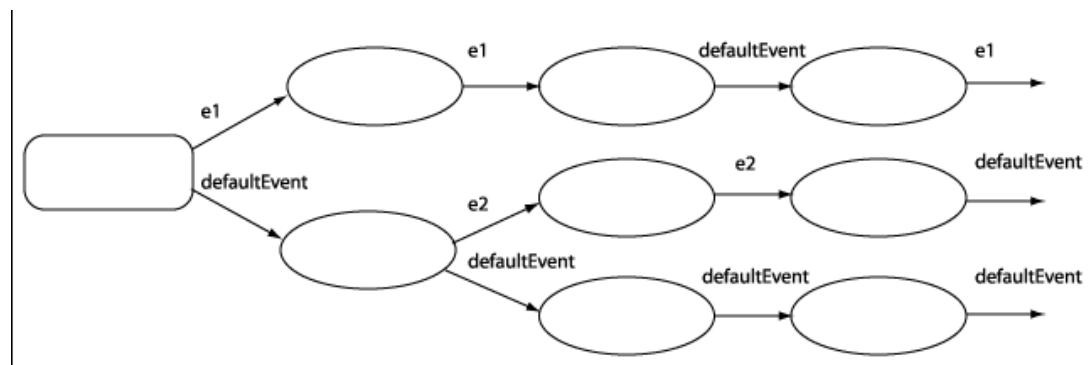


Figure 12-4 defaultEvent in a Transition

Exception transitions work the same way, such that the `defaultEvent` defined on exception transitions means “all other exceptions.” For more information on exception transitions refer to [Chapter 26, “Exception Handling.”](#)

EXAMPLE OF A STATEFUL PROCESS EXECUTION

Stateful processes are generally long-lived processes that execute over time. They can be complex and may contain multiple input ports, multiple events, and multiple states with many transitions. To maintain process data over time, each stateful process is linked to a BPO instance. It is necessary to know how events and transitions cause changes in the BPO instance to understand stateful processes. This section describes process model execution once an event arrives at an input port and a BPO instance is defined.

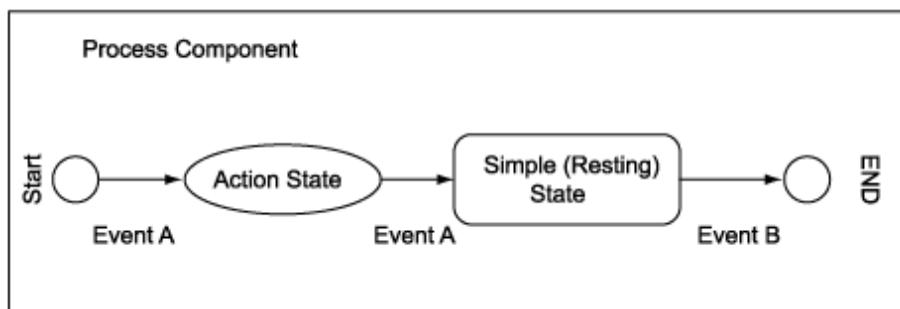
For additional information on BPOs, refer to [Chapter 11, “Process Models: Defining and Using Business Objects.”](#) For additional information on how the BPO is identified for an incoming event, refer to [“Request Map Class: Dispatching Events and Assigning BPOs” on page 13-24.](#)

Events are processed serially for a specific BPO instance. That is, regardless of how many threads are available for processing or how many input ports exist on a process component, a specific BPO instance (as identified by its `oid`) can be triggered by one thread at a time. Subsequent events for the same BPO are queued until the previous event is processed and the model reaches its resting state.

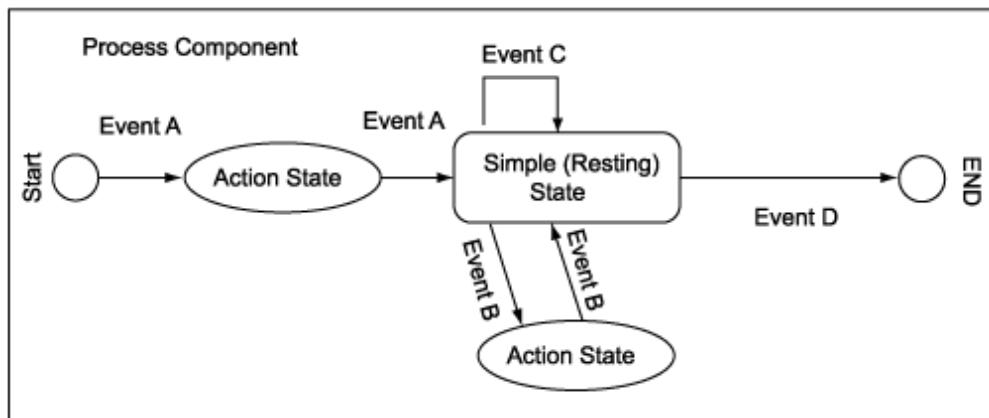
However, processing of a single event can cause multiple transitions to trigger. This kind of modelled concurrency can be implemented in a variety of ways, such as multiple start states, forks, and joins, as described [Chapter 13, “Process Modeling Techniques.”](#)

Each event moves the BPO instance from its current set of resting states to the next set of resting state(s). The originating state for every BPO instance is the start state. The event that causes the instance creation is the first event that can occur, and this originating event moves the BPO instance from the start state via a transition to the first resting state.

As shown in [Figure 12-5](#), Event A is the event that creates a new BPO instance. For example, for a new customer account, the occurrence of Event A fires the Event A transition from the start state to the action state. Because no resting state has been reached yet, the Event A transition continues from the action state to the resting state. Notice that no more transitions from the resting state are triggered from Event A after the resting state is reached.

**Figure 12-5** State Transitions via Events

The BPO instance remains in this resting state until, at some future time, another event (Event B) triggers the model to move it to the next resting state, which in this example is the terminator state. The BPO instance could be persisted for any length of time: hours, days, weeks, or more. Whenever it is retrieved in response to an event it will “remember” its last state, and therefore will be triggered by an event that can move it to the next state (as described in the section [“Deciding Which Transition Triggers” on page 12-18](#)). For example, as shown in [Figure 12-5](#), it will only respond to Event B, which will move it to the next resting state, the terminator state. A more complex example is shown in [Figure 12-6](#).

**Figure 12-6** Transitioning from a Simple State Back to that Simple State

In [Figure 12-6](#), when the BPO instance is in the resting state, three possible events can cause it to transition to another state:

- Event B, which causes a transition from the resting state to an action state and back to the resting state.
- Event C, which causes a transition from the resting state back to the resting state; this kind of transition is called a self transition.
- Event D which causes the transition to the terminator state.

Events B and C have the same effect. A transition is triggered to cause the process instance to exit the simple state and subsequently re-enter the state. The process will come to rest, and the BPO will persist the same state as before. Consequently, both Events B and C will trigger this behavior until Event D triggers the transition to the terminator state. Once the process instance reaches the terminator state, no further events will trigger transitions out of the state. The BPO is not deleted, but subsequent events will not cause any state transitions.

IMPORTANT: When a BPO instance reaches the terminator state, the BusinessWare runtime does not delete it from the persistent store. You are responsible for periodically archiving and/or deleting terminated BPO instances from your database.

This chapter describes several techniques you might use in your process models. Topics include:

- [Timers](#)
- [Forks](#)
- [Joins](#)
- [Nested Models](#)
- [Iteration](#)
- [Concurrent Processing in Models](#)
- [Decision Nodes and Decision Trees](#)
- [Request Map Class: Dispatching Events and Assigning BPOs](#)

TIMERS

Timers are used to ensure that the process transitions to the next state after a specified amount of time passes (relative timer) or when a specific time is reached (absolute timer). For example, timers are commonly used on activity states to impose a deadline for the performer to complete the task.

Keep these points in mind as you work with timers:

- You can set timers on resting states: simple states, activity states, and nested states.
- You cannot set timers on action states, transformer states, or Web service states.
- You can also configure one or more transitions coming out of the state to trigger on a `timerEvent`. Transition conditions and defaults determine which timeout transition triggers. See [Chapter 12, “Process Models: Ports, States, and Transitions”](#) for information on transitions.

RELATIVE AND ABSOLUTE TIMERS

There are two types of timers:

- **Relative timer**—when a BPO enters a state, a timer is initiated and set to the amount of time you specified:
 - If the object is still in that state when the timer expires, a timeout event is sent and a timeout transition is triggered.
 - If the object exits the state before the timer expires, the timer is automatically cancelled.
- **Absolute timer**—a timeout event is sent if the time you specified is reached while the BPO is still in that state.

Note: Time calculations are based on the operating system time clock and follow the operating system's handling of the daylight savings time switch. Consult your operating system documentation for information on how it handles time switches.

If a BPO transitions back to the originating state through a self-transition, any timers for the state are cancelled and reset.

IMPORTANT: Use timers carefully, especially absolute timers. Any simultaneous influx of a large number of events will affect performance, and multiple timers firing all at once could cause such a problem. Also, if timers in process models running in a single Integration Server could fire simultaneously, you may need to increase the maximum number of connections allowed by that server. See “[Connection Manager](#)” on page 20-6.

SETTING A TIMEOUT

When you set a timer on the state, you need to provide at least one timeout transition leading from the state.

1. Select the state to display its properties in the Property window.
2. Select the Timer property to display the Timer dialog box.
3. Check **Has Timeout**.
4. To set a relative timer:
 - a. Set the Timer Type to **Relative**.
 - b. Click **Set**.

- c. Enter the maximum amount of time you want the BPO to remain in the state as shown in [Figure 13-1](#).

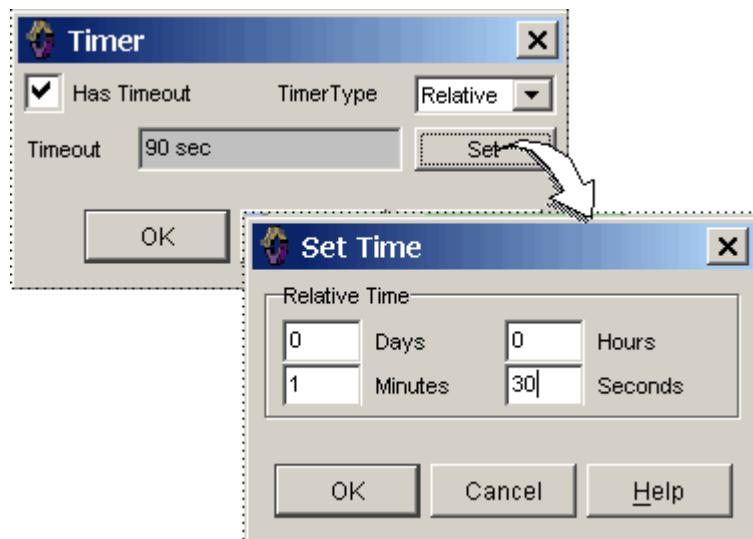


Figure 13-1 Setting a Relative Timer

5. To set an absolute timer (see [Figure 13-2](#)):
- Set the Timer Type to **Absolute**.
 - Click **Set**.
 - Choose the desired month, year, date, and hour.
 - Indicate whether you want to use Greenwich Mean Time (GMT) or local time.

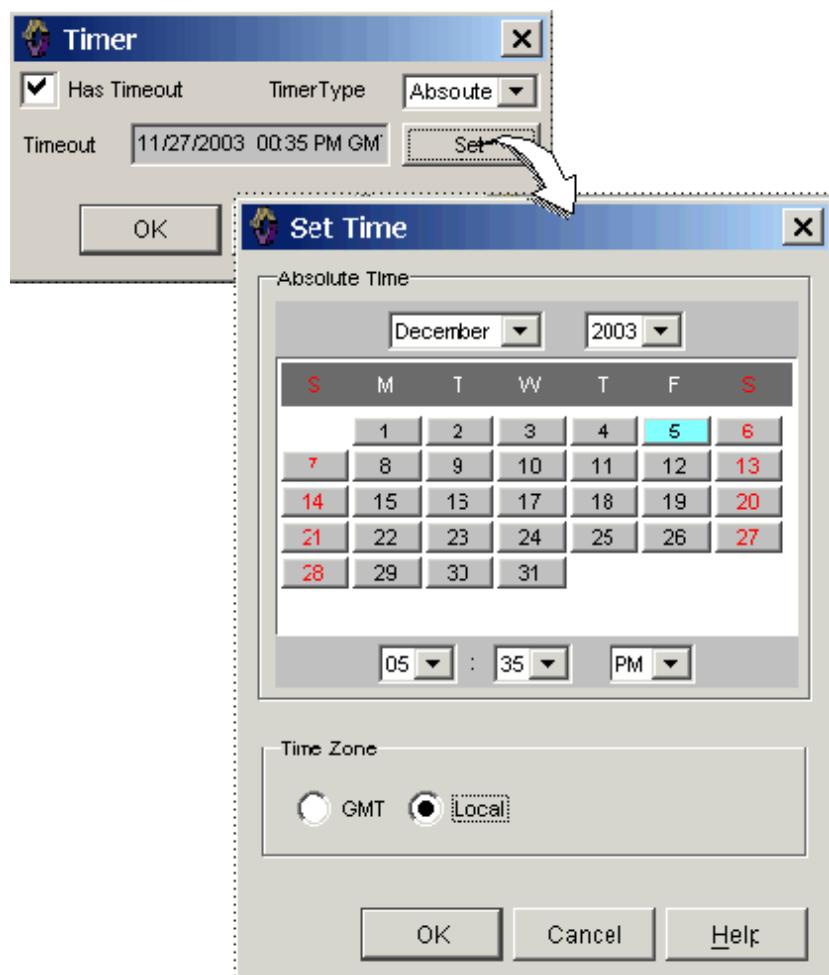


Figure 13-2 Setting an Absolute Timer

6. Configure one or more timeout transitions out of the state by setting the **Trigger Type** to **timerEvent** and specifying the conditions to be met and the action to be executed.

Note: You can also set a timeout programmatically. For instructions, see the *BusinessWare Programming Reference*.

FORKS

A *fork* is used to split a single task into parallel tasks. A fork has one incoming transition and two or more outgoing transitions, which fire simultaneously and lead to multiple destination states. For example, in the ContractModel shown in Figure 13-3, a fork is used to send approved contracts simultaneously to two groups for further processing. Both transitions out of the fork fire as soon as the input transition fires.

Keep these points in mind as you work with forks:

- Timers and actions cannot be associated with a fork.
- All the transitions out of a fork must be default transitions (both default event and default condition).
- Outgoing transitions can have no action code. If action code is warranted, use action states as successor states.

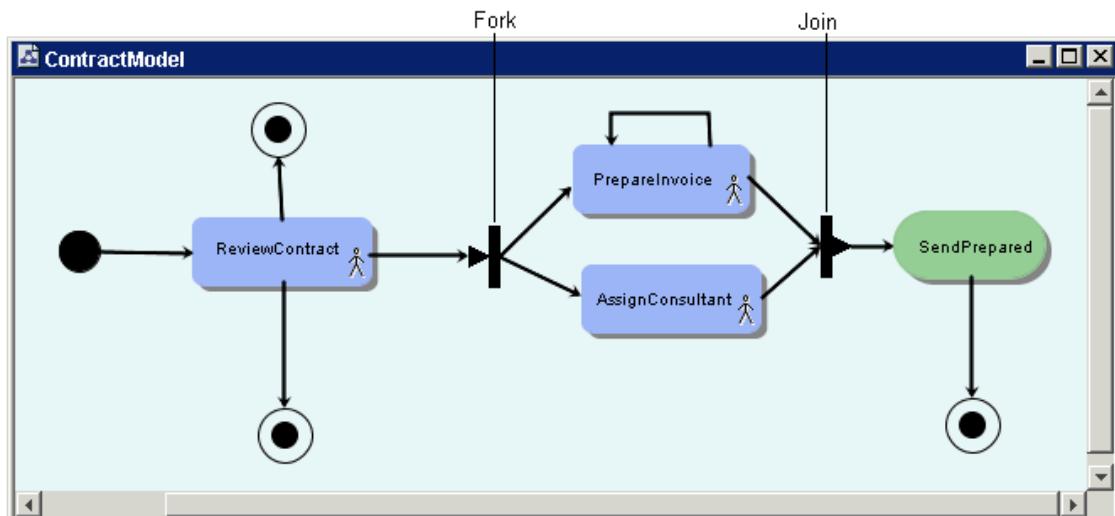


Figure 13-3 Fork and Join Used in ContractModel

To create a fork:

1. Create the predecessor state.
2. Add a fork to the model by clicking on the button  in the Process tab of the Palette and then in the Editor.

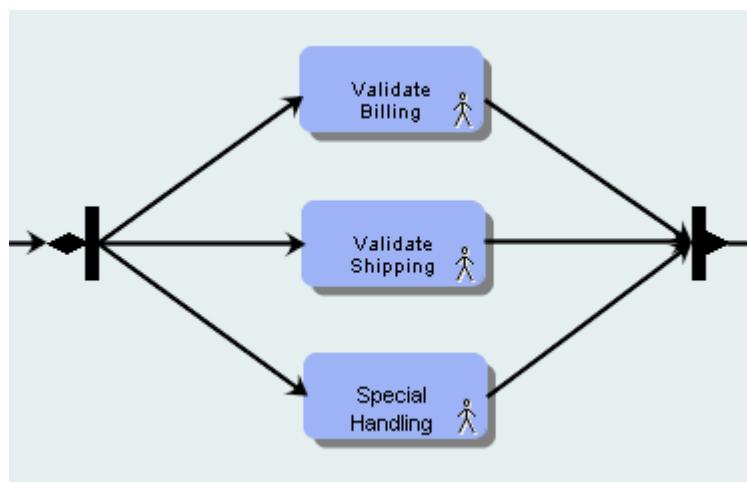
You can rotate the fork 90 degrees by right-clicking it and selecting **Rotate Left** (counter-clockwise) or **Rotate Right** (clockwise).

3. If desired, enter a name and description for the fork in the Properties window. The name is not displayed but may be helpful for debugging purposes.
4. Create a transition from the predecessor state to the fork.
5. Create two or more successor states.
6. Connect the fork to each of its successors with a transition. The settings are automatically provided when you connect a transition to a fork. For each transition, the properties are set as follows:
 - a. Trigger Type = defaultEvent
 - b. Default Condition = True

CONDITIONAL FORKS

Conditional Forks allow more freedom than normal forks. A conditional fork enables a model to transition out of a fork state based on the trigger type and condition code of each of the outbound transitions. That is, transitions out of a conditional fork can contain any trigger type or condition definition (rather than being restricted to the default).

Example: Suppose incoming orders require special handling under specific circumstances and additional tasks must be completed for all orders. The special handling is a manual step that can take several days to complete. If the special handling is processed serially, it could delay the order several days before further processing can be completed. Using a conditional fork, you can create the special handling activity immediately and other processing can be done in parallel, saving time for the overall order process. [Figure 13-4](#) shows what this process model might look like.

**Figure 13-4 Conditional Fork Example****To create a fork:**

1. Create the predecessor state.
2. Add a fork to the model by clicking on the button  in the Process tab of the Palette and then in the Editor. You can rotate the fork 90 degrees by right-clicking it and selecting **Rotate Left** (counter-clockwise) or **Rotate Right** (clockwise).
3. If desired, enter a name and description for the fork in the Properties window. The name is not displayed but may be helpful for debugging purposes.
4. Create a transition from the predecessor state to the fork.
5. Create two or more successor states.
6. Connect the fork to each of its successors with a transition. For each transition, set the Trigger Type and Condition properties. At least one transition should be configured with a non-default trigger type and condition.

JOINS

A *join* is used to reunite multiple parallel tasks into a single task. A join has multiple incoming transitions and only one outgoing transition. The outgoing transition fires only after transitions from *all* the predecessor states into the join have fired. For example, the join in [Figure 13-3](#) does not complete and transition to the terminator state until both the invoice has been prepared and the consultant has been assigned.

The process model maintains a list of all the join's predecessor states and the firing status of their transitions to the join. This list is checked at runtime to determine if all the predecessor states have transitioned to the join. Only then can the join's outgoing transition fire.

When a join is associated with a conditional fork, the outgoing transition only fires when the number of fired transitions into the join equals the number of fired transitions originally from the conditional fork. The Process model maintains the list of the number of transitions that were fired from the conditional fork.

To ensure that the join does eventually complete, you can use timeouts. Use either of these techniques:

- Add a timeout transition from *each* of the predecessor states to the join. When these timeout transitions fire, the join is forced to transition to its destination state.
- Nest the model with the join inside a parent model. Place the timer on the nested state in which the model with the join exists. Timers on nested states cause the entire nested model to exit.

Keep these points in mind as you work with joins:

- Timers and actions cannot be associated with a join.
- The transition out of a join must be triggered by a `joinEvent` and must have a default condition (i.e., it allows all conditions). These settings are automatically provided when you connect a transition to a join.
- The outgoing transition can have no action code. If action code is warranted, use action states as successor states.

To create a join:

1. Create all the predecessor states.
2. Add a join to the model by clicking on the button  in the Process tab of the Palette and then in the Editor.
You can rotate the join 90 degrees by right-clicking it and selecting **Rotate Left** (counter-clockwise) or **Rotate Right** (clockwise).
3. If desired, enter a name and description for the join in the Properties window. The name is not displayed but may be helpful for debugging purposes.
4. Create a transition from each predecessor state to the join.
5. Create the successor state.

6. Connect the join to its successor with a transition. Transition properties are automatically set as follows:

- a. Trigger Type = joinEvent
- b. Default Condition = True

NESTED MODELS

Using nested states, you can create a process model that contains other process models. Nesting models help you develop and manage complex business processes more efficiently. Parts of the system can be abstracted, either to encapsulate business logic or to facilitate model reuse.

For example, in the OrderProcessing Model (Figure 13-5), the billing and shipping processes are nested inside of two states, thus simplifying the model layout and increasing its readability.

In addition, having those processes defined in separate models means that billing and shipping can easily be “plugged into” other models. The only requirement for reuse is that the submodel and the parent model both process compatible BPOs.

Note: The parent model's BPO may be a subclass of the nested model's BPO.

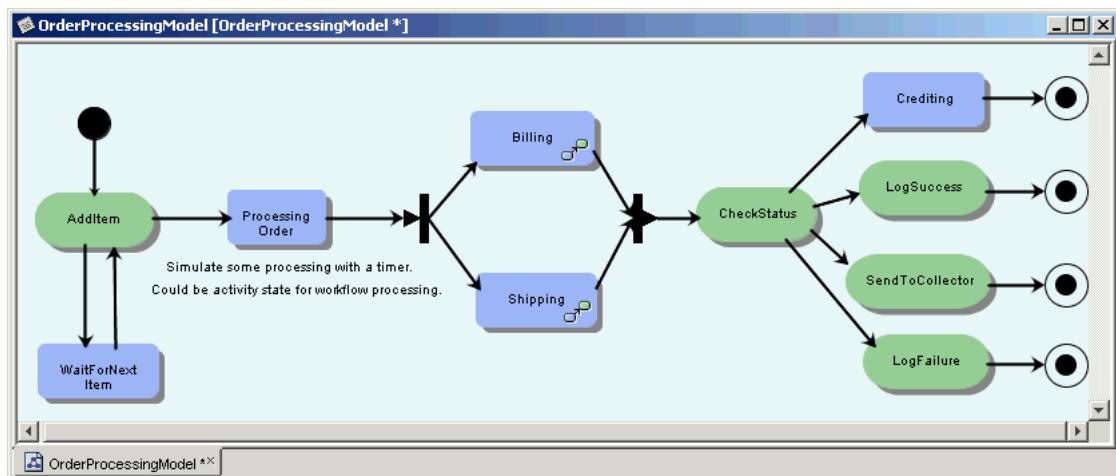


Figure 13-5 OrderProcessing Model: Nested States Contain Billing and Shipping Models

A nested state can be linked to more than one submodel. In the Nesting Specification, you define all the possible submodels and the conditions under which each one may be chosen at runtime. This *dynamic nesting* is a Vitria extension of standard UML. This nesting enhances the power of nested models by enabling the process to adapt to a rapidly changing environment at runtime.

Note: BusinessWare does not support reentrant process models. A submodel cannot invoke its parent model. That is, if model B is nested in model A, then model A cannot be nested in model B. At runtime, the container in which the components run will detect reentrant models and abort.

NESTING TERMINOLOGY

Several key terms used in describing nesting are defined in [Table 13-1](#).

Table 13-1 Nesting Terminology

Term	Definition
nested state	State that contains a separate model.
submodel (or nested model)	Model contained by a nested state.
level 1 model (or top-level model)	“Master” model being run; that is, the only model that is not currently a submodel of another model.
level n model	Submodel at the n th level; that is, if the level n state is a nested state, its submodel is the level $n+1$ model.
level 1 state	Current state in the level 1 model.
level n state	Current state in the level n model.
BPO state	Model state or collection of states that completely describes the state of the BPO.

UNDERSTANDING NESTING

Nesting can extend several levels deep, and the layered processes are managed as follows:

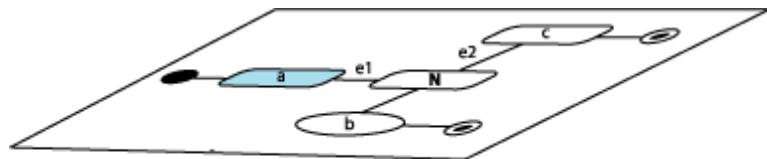
- When a BPO is in a nested state, its “current state” is actually a list of *all* the states in *all* the nested models and top-level model where the BPO currently resides.
- Events are processed top-down. When an event is received, transitions are examined beginning with the top-most level:

- If the event can trigger a transition out of the level 1 nested state, the BPO moves to the next state in the level 1 model and simultaneously exits all submodels.
- Otherwise, the event is passed through to subsequent levels of nesting until a transition that can be triggered by the event is found.
- If all levels are evaluated and no transition can fire, the model default action is executed. Again, the approach is top-down: if the level 1 model executes a model default action, the default actions of the submodels are not needed and not executed.
- When a BPO transitions out of a level n nested state, it simultaneously exits all the submodels below that state (that is, the level $n+1$ model, level $n+2$ model, and so on).

Example

To illustrate these concepts, a simple example of nesting is shown in [Figure 13-6](#) through [Figure 13-9](#). In this example, **N** is a nested state, and **en** is used to represent “a transition that triggers on event *en*.” The state or states in which the BPO currently resides are indicated with color fill.

[Figure 13-6](#) shows the top-level model. The BPO’s current state is state **a**.



before entering nested state: BPO state = {a}

Figure 13-6 Top-Level Model Containing Nested State N

When the model receives event e1, it causes a transition to the nested state N (Figure 13-7). Based on the nesting specifications and the conditions of the event, the appropriate submodel is invoked. The BPO simultaneously moves to state N in the level 1 model and through the start state of the level 2 model to state d. The BPO state is now state N plus state d.

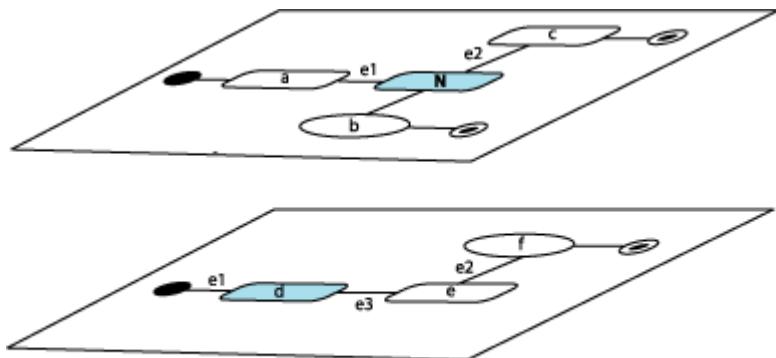


Figure 13-7 Event e1 Causes Transition to Nested State, which Invokes Submodel

Event e3 arrives and is evaluated, first to see if it can trigger a transition from state N (it cannot) and then to see if it can trigger a transition from state d (it can). The BPO remains in state N in the top-level model *and* moves to state e in the submodel. The BPO state is now state N plus state e (Figure 13-8).

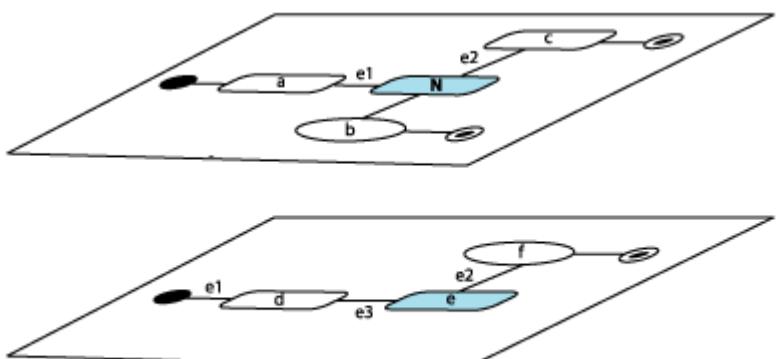
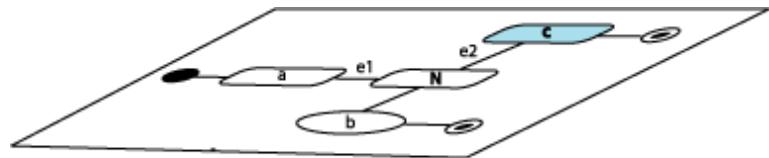


Figure 13-8 Event e3 Causes Transition in Submodel

Event e2 arrives and can trigger a transition at the top level. The BPO moves to state c in the level 1 model and simultaneously exits the submodel. The BPO state is now state c ([Figure 13-9](#)).



after event e2: BPO state = {c}

Figure 13-9 Event e2 Causes Transition in Top-Level Model

Role of Terminator States in Nested Models

As illustrated in the previous example, subprocesses are exited prematurely when an event causes a transition out of the parent nested state. But what happens when a subprocess completes normally? How is the parent nested state informed and how does it react? Terminator states fill that role.

Terminator states in submodels can maintain a static *termination value*. When a BPO enters a terminator state, a `terminateEvent` containing that state's termination value is immediately sent to the parent state in the parent model. The termination value is used to determine which transition out of the parent nested state can trigger. (A nested state may have multiple outgoing transitions that specify a `terminateEvent` and test the termination value in the conditions.)

Thus, a nested billing model might have two terminator states that return values of “BillPaid” or “BillOverdue” to the nested parent state. The nested state then transitions to the “SendThankYou” or “CallCollectionAgency” states, depending on the value it received.

The `terminateEvent` is defined by

```
event void terminateEvent(in string value);
```

IMPORTANT: If a nested model has multiple terminator states, the first one that is reached causes the `terminateEvent` to be sent. The model does not wait for all terminator states to be reached.

Tip: If a nested model has concurrency (for example BPO is in multiple states at the same time in the nested model), and if it should not exit until all concurrent paths have completed, a join can be used immediately before a single terminator state. For more information on joins, see “[Joins](#)” on page [13-7](#).

You should always use terminator states rather than resting states with no outbound transitions.

POR TS ON PARENT MODELS AND SUBMODELS

Ports on the submodel must have matching ports on the parent model. You have to create the ports on the submodel manually; the BME does not replicate ports to nested models.

The parent model, of course, may contain more ports than each of its nested submodels. The parent model contains a superset of *all* the ports on *all* its submodels.

CREATING NESTED MODELS

You create nested process models by creating a nested state within the parent model and specifying which submodels the nested state can invoke:

1. Add a nested state to the parent model by clicking on the  button in the Process tab of the Palette and then in the Editor.
2. Connect the nested state to its predecessor states with transitions.
3. Specify which submodels the nested state can invoke under which conditions by setting the **Nesting Specification** property. Using the Nesting Specification dialog, map a triggering event type and required conditions to the name of a process model. For more information on specifying the nested model, see “Nesting Type” on page 13-15.

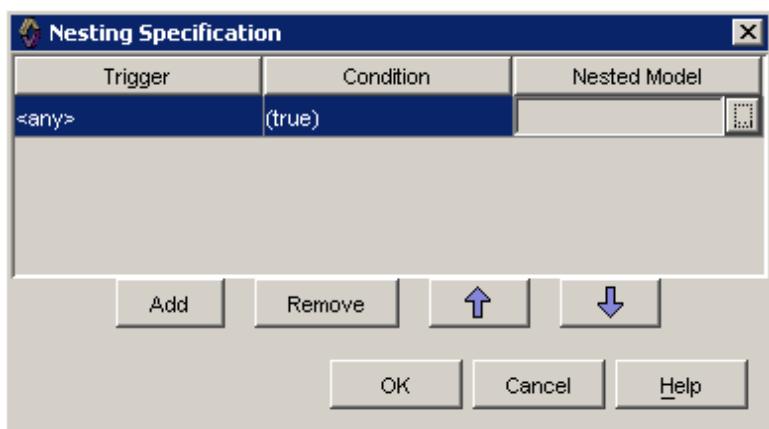


Figure 13-10 Setting Nesting Specifications

Nesting Type

When configuring the Nesting Specification property of a nested state, you can choose from the following options when specifying the nested model:

- Static nesting
- Dynamic nesting using XQuery
- Dynamic nesting using Java

Static Nesting

Static nesting binds the nesting specification to a specific model name at design time.

To select a model that will be statically nested:

1. Click in the Nested Model field in the Nesting Specification dialog.
2. Select **Static Model** from the Nesting Type drop-down box.
3. In the Nested Model dialog, click **Browse....**
4. In the Select Process Model dialog, select a model from the project (or dependent project) that will be statically bound. See [Figure 13-11](#).

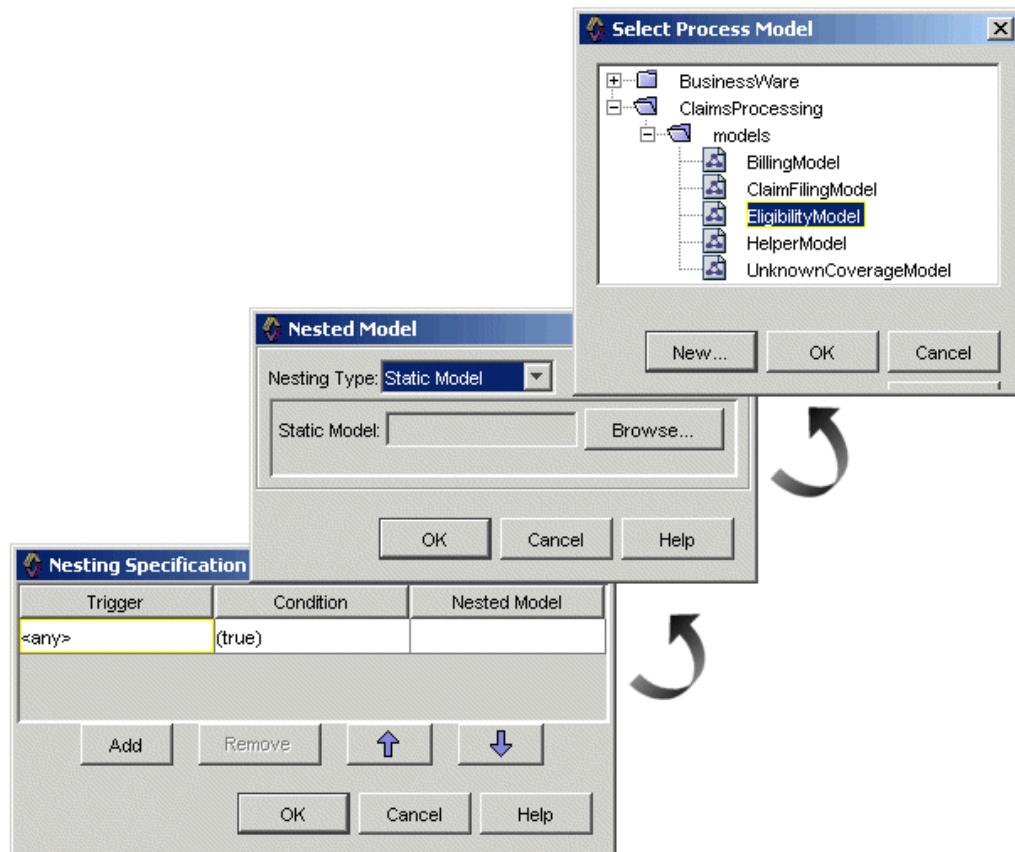


Figure 13-11 Selecting a Static Nested Model

Dynamic Nesting

Dynamic nesting allows the choice of model to be made at runtime, based on parameters not known at design time (for example, based on data contained within an incoming event, document, or other data object). There are two ways to specify dynamic binding:

- XQuery
- Java

Dynamic XQuery enables you to choose a nested model by querying XML data. This is useful when information necessary to make the decision is stored in documents that are part of the process.

To use XQuery to define a model that will be dynamically selected:

1. Click in the Nested Model field in the Nesting Specification dialog.
2. Select **Dynamic XQuery** from the Nesting Type drop-down box.
3. In the Nested Model dialog, Payload indicates the DocumentReference or DocumentReference array used in
`com.vitria.xquery.XQueryActionLib.queryAsString
(java.lang.String,
com.vitria.fc.data.DocumentReference). This method executes
the XQuery expression and returns the results as a string. (See the
BusinessWare Programming Reference for information on BusinessWare
APIs.)`

When the event is an `xmlEnvelopeEvent`, the payload is initialized using:

```
payload =  
(com.vitria.fc.data.DocumentReference[]) (ctx.getEve  
nt().getJavaParameters()[1]);
```

The DocumentReference can also be retrieved programmatically if the `docRefId` is known using:

```
ContentLib.createDocument(docRefId)
```

4. Select Query By File or Query By String.

- To load a file containing an XQuery string, select **Query By File**, click **Browse...** and select a file in the Choose XQuery File dialog.
- To build an XQuery string using the XQuery Builder, select **Query By String** and click **Edit...** to open the XQuery Builder. For more information on the XQuery Builder see “[Using the XQuery Builder](#)” on page 14-16.

PROCESS MODELING TECHNIQUES

Nested Models

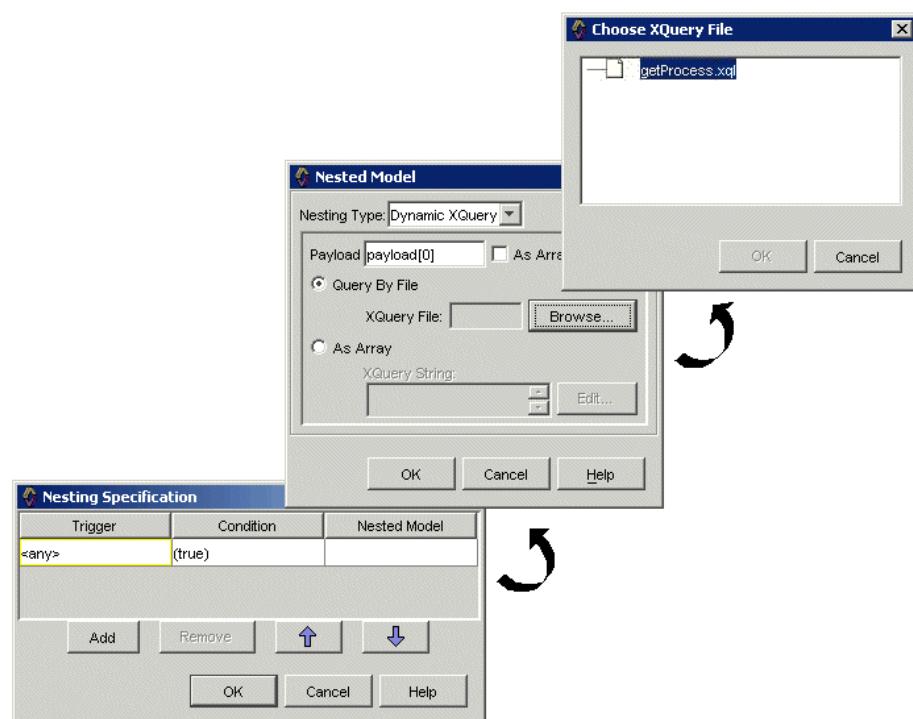


Figure 13-12 Specifying a Process Model - Query By File

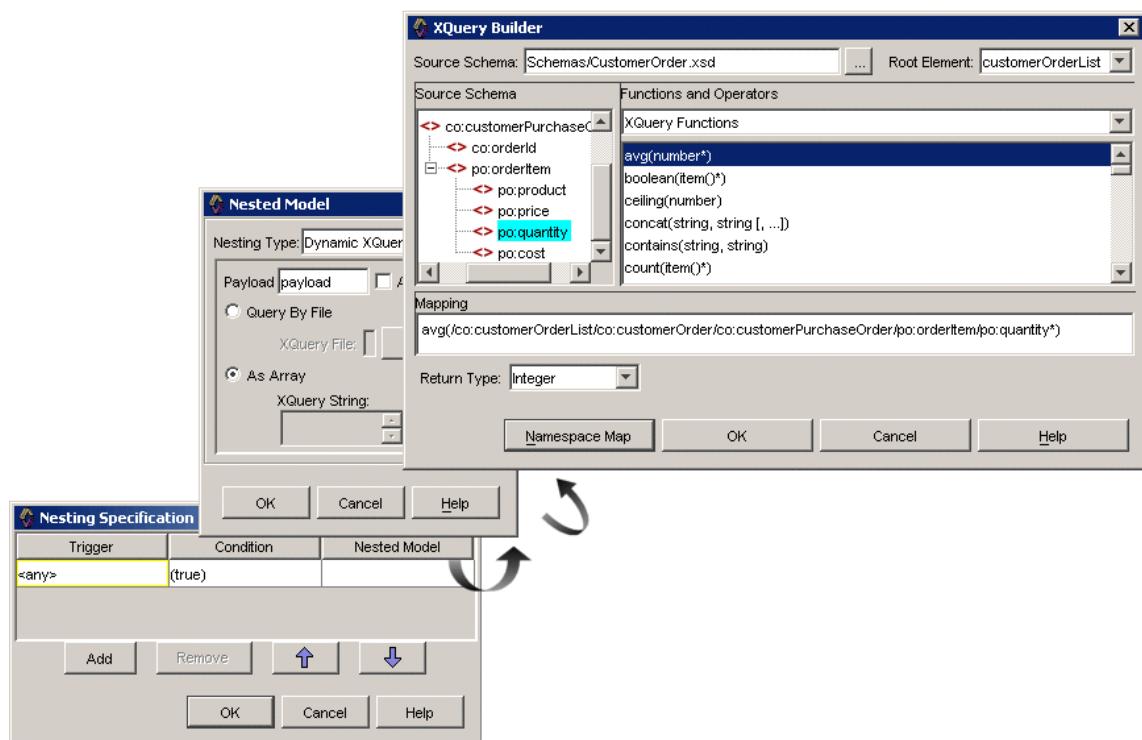


Figure 13-13 Specifying a Process Model - Query By String

Dynamic Java enables you to choose a nested model by writing Java code that returns the model name (see fig 13-12). (Adam: figure 13-12 should show the Nesting Type is Dynamic Java and should have an overlay popup of some valid Java code with a “return” statement in it that returns a model name dynamically.)

To use Java to define a model that will be dynamically selected:

1. Click in the Nested Model field in the Nesting Specification dialog.
2. Select **Dynamic Java** from the Nesting Type drop-down box.
3. Enter the Java code and specify the name of the model in a return statement. Click **Insert...** to add code using the Action Picker.

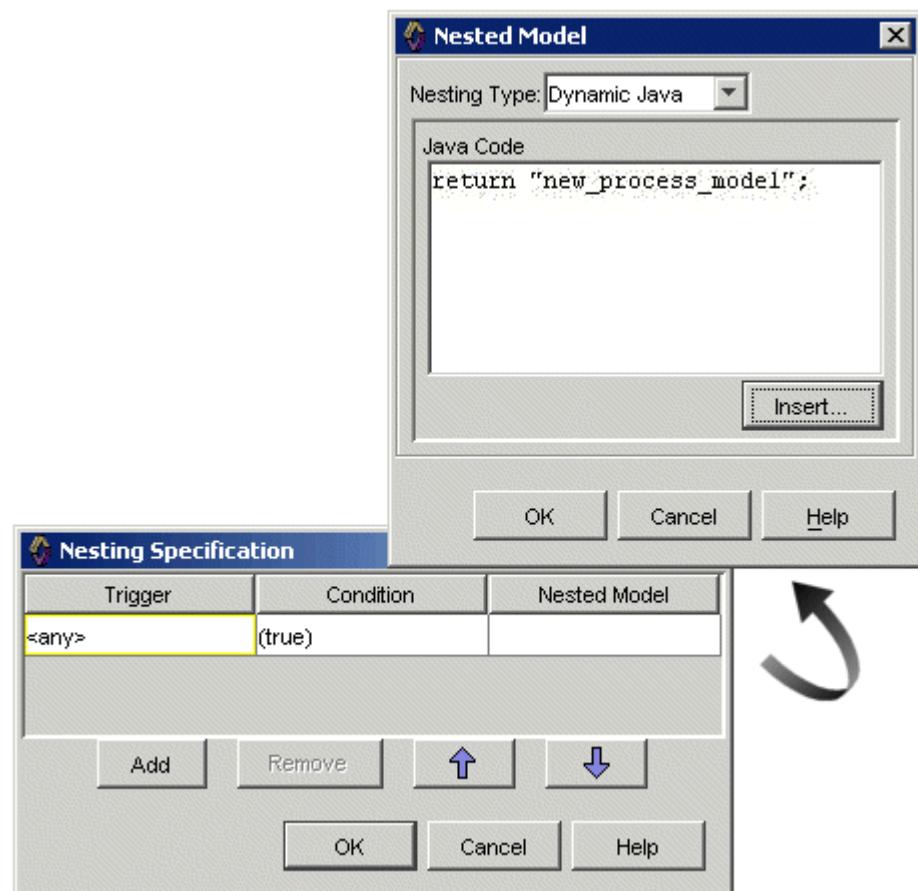


Figure 13-14 Specifying a Process Model Using Dynamic Java

ITERATION

Use iteration when you are dealing with a data collection and want to process the collection one data element at a time. During iteration, the action code in a state executes repeatedly until all of the data in a collection has been processed. When processing a collection, the model iterates through the elements sequentially, one at a time. For example, if the collection is a list of ten elements, iteration begins with the first element and proceeds in order to the last element.

You can configure iteration on non-resting states (action states and nesting states that nest stateless models without any resting states).

The following data types can be iterated:

- array
- java.util.Collection
- eventSequence(EventBody[])
- xmlEnvelopeEvent

To configure a state for iteration:

1. Click on the Iteration property of an action or nested state.
2. Check the **Enable Iteration** checkbox.
3. Select one of the following data sources from the drop-down list:
 - Inbound Transition
 - BPO
 - Local Data
4. Specify the event parameters (for inbound transitions) or BPO/Local Data parameters (For BPO or local data) through which the state will iterate.

For nested states, in addition to passing the iterated element as part of IterationContext, you can nest the iteration element as an event.

5. Check **Nest iterated item as event**.
6. Click **Browse...** and pick an event type.
7. Specify the event parameters in the Event Parameters table.
8. Click **OK**.

Note: When a state is configured for iteration, it must have exactly one trigger type coming into it. The state can have multiple inbound transitions as long as they are of the same type. This ensures that there is only one set of inbound event or operation parameters per iteration construct.

Figure 13-15 shows an action state configured with iteration enabled.



Figure 13-15 Action State with Iteration Enabled

After completing the iteration configuration, `IterationContext` is created in the process model code. `IterationContext` returns four methods:

- `getCurrentElement()`

- `getCollection()`
- `hasNext()`
- `next()`

These methods are available as a group of actions under “Iteration” in the Action Picker.

You can obtain the context as part of the process model context by making a call to `ProcessModelContext.getIterationContext()`.

Note: When a multi-dimensional array is used as a data source, only the first dimension is iterated. For multi-dimensional arrays, you should use nested models to successively break down the array. For example, `String[][]` is first iterated as `String[]` in a process model. Next, a nested model further iterates through `String[]` to process the individual String.

Note: When an iteration object is null, no exception is raised because BusinessWare assumes it is iterating through an empty collection or array.

CONCURRENT PROCESSING IN MODELS

In a model with concurrent processing flows, the process can be in more than one state at a time. This is useful for modeling tasks that can be completed independently of each other. For example, in an order management system, billing and shipping are two separate operations that can be done in parallel. The OrderProcessing Model, shown in [Figure 13-5](#), separates these two tasks with a fork and then reunites the process with a join.

Forks and joins are one way to achieve concurrency in a model. Multiple start states are another. With multiple start states, you can build a model that has two or more distinct chains of activity. For example, perhaps one part of the process performs billing operations and another part enables customer queries about their bills. Queries can occur during any phase in the billing operation. They are parallel requests, always.

If you use multiple start states, follow these guidelines:

- An incoming event that triggers a new process instance is sent simultaneously to all the start states in the model.
- The process must exit all the start states immediately and transition to successor states. There is no resting in a start state.
- The graphs emanating from each start state must be distinct. They cannot overlap, intersect, or share states.

As with nested models, the BPO state in a concurrent model is a list of all the states where the BPO currently resides.

Note: Do not confuse branching into multiple states with multithreading. A single trigger into the model is processed in a single thread context. When the model receives an event, all the current states are checked, one by one, to see if they can consume the event. Note also, the order in which states are evaluated is non-deterministic.

If there are multiple requests or events for the same BPO, they are processed in sequence by the same thread. This can restrict the concurrency by requiring a particular set of requests to be processed serially. Specifically, stateless models typically have fewer ordering requirements, and can use a less restrictive RequestMap implementation. For more information, see “[RequestMapImpl and Stateless Models](#)” on page 13-25 for information on setting the Request Map class for stateless models.

DECISION NODES AND DECISION TREES

You can use an action state to model a decision node or a series of chained action states to model a decision tree.

In a *decision node*, a choice must be made between two or more transitions. Typically this is accomplished as follows: Calculations performed by the action state set values that can then be tested by the transition conditions to determine which transition fires. These values could be attributes of the BPO, attributes of a Local Data object used by the model, or model variables. You can use whatever is appropriate for your application.

In a *decision tree*, a complex decision is made by breaking it into a series of smaller decisions and stepping through each of those in succession. You model a decision tree with a series of action states. Each action state functions as a decision node, with outgoing transitions leading to other decision nodes.

Regardless of the number of action states you use to represent the decision-making process, all processing should be *short in duration*. The thread allocated to processing the action will not process any more events until this code completes.

REQUEST MAP CLASS: DISPATCHING EVENTS AND ASSIGNING BPOs

A Request Map class is associated with input ports in order to determine how to dispatch incoming requests (events). The class is particularly significant for process models that store data and state information for a particular process instance in a BPO.

The Request Map class:

- Decides whether the request should be processed or dropped.
- Identifies the correct BPO instance to associate with the request. Later it is created or loaded from the database by the runtime services.
- Ensures that requests for a single object or group of objects are processed in the proper sequence

The default Request Map class for all projects is the `com.vitria.bpe.runtime.ProcessRequestMapImpl` class. This section describes the default implementation of this class and explains why, in certain cases, you may want to create a custom implementation class.

DEFAULT REQUEST MAP AND THE BPID PARAMETER

The default Request Map class, `com.vitria.bpe.runtime.ProcessRequestMapImpl`, is designed to work with:

- Requests that incorporate the `bpe.BPID` structure

The BPID is a structure defined as follows:

```
struct BPID {
    string oid;
    string type;
};
```

Where `oid` is the unique identifier for the BPO instance and `type` is any value that uniquely identifies the type of object (such as, Order or Customer). For more information on the `bpe.BPID` structure, see the *BusinessWare Programming Reference*.

The type is useful for routing decisions used with the default routers provided with BusinessWare. For more information on routers see [Chapter 15, “Process Model Templates.”](#)

- Requests where the first parameter is a string

The default implementation expects and looks for a BPID.oid (object identifier) as the primary key for locating the correct record in the database. If it does not find a BPID structure, and the first parameter is a string, it will assume that the string is the oid you want to use as the primary key for lookup. If it cannot find a BPID structure, or a first parameter that is a string, then you may need to provide a custom Request Map class based on the type of model and characteristics you want to define (see [Figure 13-16](#)).

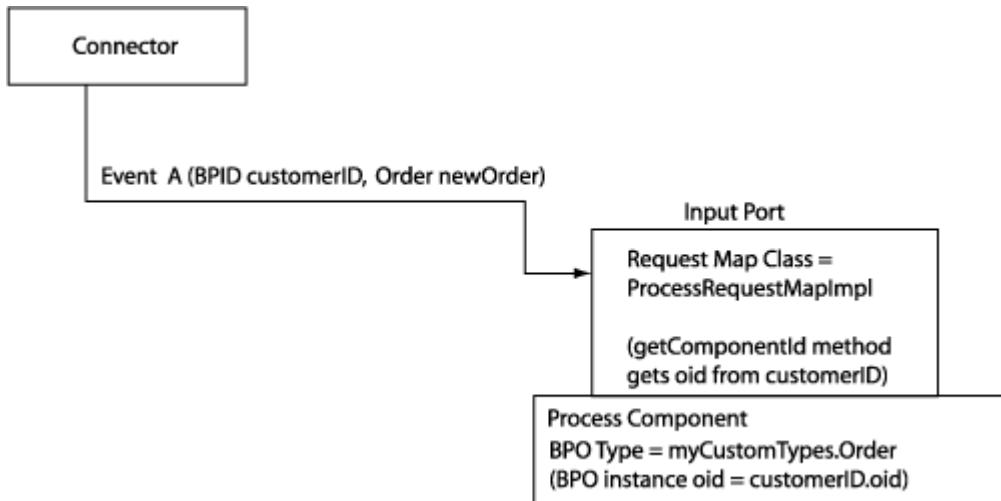


Figure 13-16 Using the Default Request Map to Get a BPO Instance

As shown in [Figure 13-16](#), Event A has a BPID as its first parameter.

The default request map gets the oid from the BPID parameter and returns it so that the persistent store can create or retrieve the BPO instance. The getComponentID, where componentID is the oid, is the Request Map class method that performs this function.

REQUESTMAPIMPL AND STATELESS MODELS

To ensure that events are handled concurrently in stateless models, you should use the `com.vitria.container.map.RequestMapImpl` class. If you use the default request map, events may be handled sequentially.

For example, if an event enters a stateless model, and the first string parameter only handles a small number of values (such as region codes), the default request map, `ProcessRequestMapImpl`, forces all events with the same region code to be sequential instead of concurrent. To handle events in stateless models concurrently, set the Request Map Class property of the port on the process model handling the events to `com.vitria.container.map.RequestMapImpl`.

REQUEST MAP METHODS

To dispatch a request, the infrastructure uses a data structure called a *request descriptor*. The request descriptor contains such information as the project, version, component, BPO identifier, and group identifier. It is built partly from information derived from the port wiring and partly from values returned by the following methods:

```
boolean canProcess(ReadableRequestDescriptor r, ProjectExecutionContext ctx);
String getComponentId(ReadableRequestDescriptor r, ProjectExecutionContext ctx));
String getGroupId(ReadableRequestDescriptor r ProjectExecutionContext ctx));
String getComponentType(ReadableRequestDescriptor r ProjectExecutionContext ctx));
```

If necessary, you can override the default implementation of these methods as described below:

- `canProcess()`—returns `True` if the component can process the incoming request and returns `False` if it cannot. The default implementation always returns `True`.

You can customize the `canProcess()` method to allow the runtime to identify certain requests that should be dropped by the component. This capability is especially useful if an event is pushed to multiple downstream components from a single Channel or Queue Source Connector. In effect, it provides subscriber-side filtering. However, you can accomplish the same goal with less overhead by placing a router between the invoking component and the invoked components. See [Chapter 15, “Process Model Templates”](#) for more information.

- `getComponentId()`—returns the database key that is used to retrieve the BPO instance associated with the incoming event. (Note that “component” in this method refers to the BPO associated with the request, not to a component in the model.)

The default implementation assumes that the incoming event includes a BPID structure or a request where the first parameter is a string. The event can return the `oid` field from either. If the incoming event does not have a BPID structure or a request where the first parameter is a string, or if the invoked component is stateless, the default implementation returns null. If your event has some other way of identifying the BPO associated with an event, you must provide your own implementation of this method that returns the BPO's database key.

- `getGroupId()`—returns an identifier that is used to ensure that all requests for a single object or group of objects are processed by the same thread in the sequence in which they arrive. Multi-threaded source connectors call this method to group and serialize related outgoing events. This ensures that multiple threads running in parallel on a Channel Source Connector do not process events in a different order than their sequence on the channel.

The default implementation returns the same value as the default implementation of `getComponentId()` which ensures that events related to the same BPO instance arrive in sequence. If you want to serialize events differently you need to provide your own implementation of this method.

The following example demonstrates the different uses of `getComponentId()` and `getGroupId()`. If a process has two stateful models that share a DO, where one model takes order header events and one model takes order line events (all events from the same queue) then you should insure the order of processing to reduce issues with the shared DO. The `getComponentId()` method for the order header returns the “order ID”, and the `getComponentId()` for the order line event returns “order ID + line number”. Use `getGroupId()` to return the “order ID” for both the header and line events and they will all be dispatched to the same thread for processing.

- `getComponentType()`—returns a value that is used by the `RouteByPortName` router to determine where to send the request. By default, this router matches the type value in the request to the name of one of the router's output ports. The routing is automatically done by the underlying logic in the model. You need only define the type for the incoming request and design output ports on the router with names that match the possible types.

The default implementation of this method assumes that incoming events contain a BPID structure; the method returns the `type` field from this structure. If the events that a `RouteByPortName` router can receive do not include a BPID structure, you must provide an implementation of this method that returns a value corresponding to an output port name.

CUSTOM REQUEST MAP IMPLEMENTATION

If you develop a custom Request Map class, you can associate it with all input ports in a project or with individual input ports:

- **Default for all input ports**—select the project object in the Explorer. Select the Runtime tab in the Properties Window.

Set the Request Map Class property to the name of your class. All input ports that do not have the Request Map Class property set locally will use this project-level setting. The default value is
`com.vitria.bpe.runtime.ProcessRequestMapImpl`.

- **For individual input ports**—select the port on the component and set the Request Map Class property in the port’s Properties window. This local setting overrides the value in the project properties.

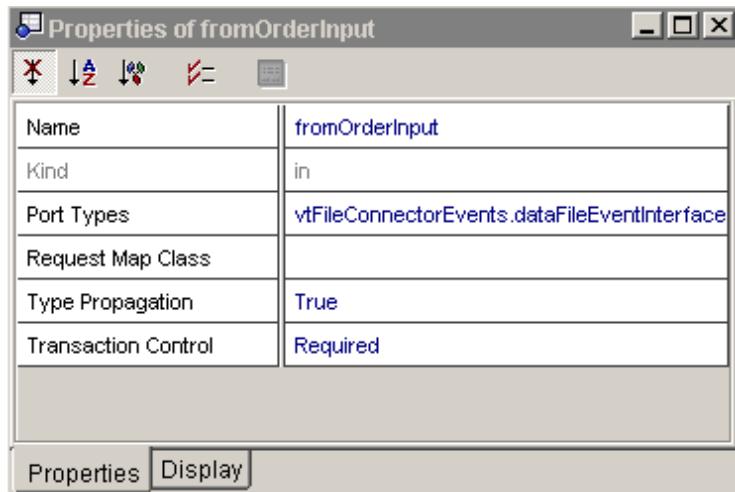


Figure 13-17 Request Map Class Property

To specify a custom Request Map class, extend the `RequestMapImpl` or `ProcessRequestMapImpl`. In this custom class you must override the following class method:

```
String getComponentId (ReadableRequestDescriptor r,
ProjectExecutionContext ctx)
```

For example, in [Figure 13-18](#), a custom request map is specified for the input port. The overridden method `getComponentID` pulls the social security number from the event parameter for use as the `oid`.

For a further discussion of the Request Map class and examples of custom implementations, see the Order Process Sample, which is located in the following directory:

installdir\samples\modeling\OrderProcessSample

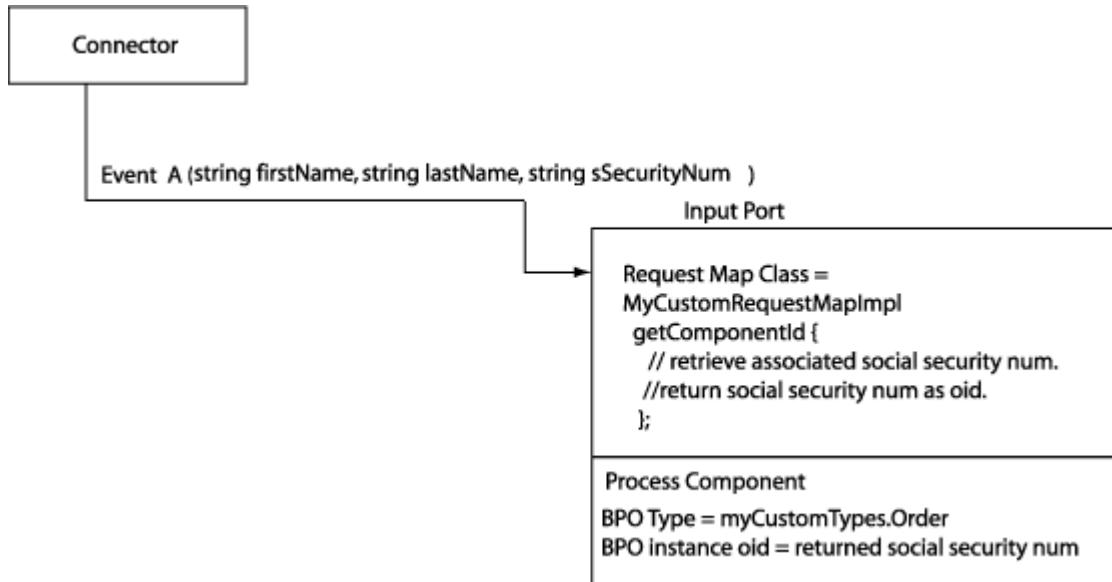


Figure 13-18 Using a Custom Request Map to Get a BPO Instance

Note: You should use the `RequestMapImpl` class with stateless model to ensure that events are handled concurrently, unless your stateless model has special sequencing requirements. See “[RequestMapImpl and Stateless Models](#)” on page 13-25 for more information.

PROCESS MODELING TECHNIQUES

Request Map Class: Dispatching Events and Assigning BPOs

This chapter describes the code builders and code editors that are provided in the BME. It also presents code examples. Topics include:

- [Overview](#)
- [Accessing the Code Builders](#)
- [Using the Action Builder](#)
- [Using the Action Editor](#)
- [Using the Condition Builder](#)
- [Using the XQuery Builder](#)
- [Using the Source Code Editor](#)
- [Action Categories and Common Actions](#)
- [Model Variables](#)
- [Initializing a Project and Providing Project Global Data](#)
- [Sample Action Code](#)
- [RequestResponseLib vs. ResponseListenerLib](#)

OVERVIEW

This section explains the relationship between graphical process models and the source code that underlies them. It also introduces several tools that help you create that code.

MODELS AND SOURCE CODE

A process model is a graphic representation of a business process. It shows the distinct phases in the process (states) and the logical connections between them (transitions). The actual business rules that control the process and the progression through it are in the source code.

Much of that code is automatically generated as you add objects to the graphical model and set their properties. However, you must explicitly define:

- **Triggering rules for transitions**—that is, the type of *event* that can trigger the transition, the *conditions* that must be met for the transition to fire, and the *action* executed when the transition fires. For more information on transition specifications, see “[How Transitions Work](#)” on page 12-11.
- **Entry actions for states**—the actions executed as the process enters the state. First the action code of the incoming transition is executed; then the entry action is executed. Use entry action when there is an action that must *always* be performed when a state is entered, regardless of which transition fired.
- **Exit actions for states**—the actions executed when an event triggers a transition out of the state. First the exit action is executed; then the action code of the outgoing transition is executed. Use exit action when there is an action that must *always* be performed when a state is exited, regardless of which transition fired.
- **Model default action**—the action executed when an incoming event causes none of the transitions to fire. Use of model default actions is optional; you do not have to define one for your process model. See “[Default Transitions and Model Actions](#)” on page 12-16.

The BME provides several tools to assist you in constructing this code.

TOOLS FOR CONSTRUCTING CODE

You don’t have to be a programmer to write action code in BusinessWare. The BME provides several code builders and editors, each offering a different level of assistance. Choose the tool that suits your skill level and experience:

- **Action Builder**—provides a point-and-click approach to building code; requires little or no programming knowledge. You simply select from prebuilt actions and set parameters to suit your model and your purposes.
- **Action Editor**—assists in writing code by supplying method names; requires a minimal knowledge of Java syntax. As you write your Java statements, you can choose and insert methods from the Action Picker.
- **Condition Builder**—provides point-and-click building and assisted editing of transition conditions; requires little or no programming knowledge. Like the Action Builder and Action Editor, the Condition Builder lets you pick from prebuilt actions to build a list of conditional expressions, or choose and insert methods as you write code.

- **Condition Editor**—assists in writing code by supplying method names; requires a minimal knowledge of Java syntax. As you write your Java statements, you can choose and insert methods from the Action Picker.
- **XQuery Builder**—provides point-and-click building and assisted editing of XPath and XQuery expressions; requires little or no programming knowledge.
- **Java Source Code Editor**—allows you to take full advantage of the flexibility and power of the Java language and requires programming knowledge. Features like code completion and color highlights for syntax help you code correctly and quickly.

RELATIONSHIP BETWEEN CODE BUILDERS AND EDITORS

[Table 14-1](#) and [Table 14-2](#) describe the relationship between the code builders and code editors. Generally speaking, code generated in a code builder cannot be edited in a code editor. You must edit generated code in the tool you used to generate it.

Table 14-1 Generated Code

Code Generated Here	Can Be Viewed Here	Can Be Edited Here
Action Builder	Action Editor, Condition Editor, and Source Code Editor	Action Builder
Condition Builder	Source Code Editor	Condition Builder

Table 14-2 Written Code

Code Written Here	Can Be Viewed Here	Can Be Edited Here
Action Editor	Source Code Editor	Source Code Editor
Condition Editor	Source Code Editor	Source Code Editor
Source Code Editor	Action Editor	Action Editor

HOW TO USE ACTION CODE

When you build action code, several types of data are readily available to you, including the following:

- **Event data**—data associated with the event that triggered the transition
- **Attribute data**—data stored in the BPO’s attributes
- **Local data**—transient computational data that is held in a Local Data object

Vitria provides a large library of actions (Java methods) for accessing and manipulating these data. These actions are listed in the code builders. For descriptions of each action, see the *BusinessWare Programming Reference*.

In addition, *model variables* can be used as “shorthand” for accessing various types of information, such as current state. [Table 14-6](#) lists the model variables you can use in your code.

A typical sequence of actions in a transition might be as follows:

- Get data from the incoming event and store these values in temporary storage variables
- Manipulate temporary storage variables to calculate new data
- Modify an attribute of the BPO with new data
- Push an event to a port, along with associated data
- Mark associated data as “modified” to be written to persistent storage

ACCESSING THE CODE BUILDERS

You can access the various code builders and editors from context menus or by double-clicking on the state or transition. The code builders are also available by selecting their corresponding fields in the Properties window for states or transitions.

ACCESSING CODE BUILDERS FROM THE CONTEXT MENUS

You can display the code builders or the Java Source Code Editor by right-clicking the state, transition, or model background and choosing the appropriate tool from the context menu.

Right-click the state, and select:

- **Tools > Entry Action...** or **Exit Action...** to display the Action Builder
- **Tools > Entry Action Code** or **Exit Action Code** to display the Source Code Editor

Right-click the transition, and select:

- **Tools > Action...** to display the Action Builder
- **Tools > Action Code** to display the Source Code Editor
- **Tools > Condition...** to display the Condition Builder
- **Tools > Condition Code** to display the Source Code Editor

Right-click in the model background, and select:

- **Model Default Action** to display the Action Builder
- **Model Default Action Code** to display the Source Code Editor

ACCESSING CODE BUILDERS BY DOUBLE-CLICKING

By default, BusinessWare is configured to display a code builder when you double-click on a state or transition.

To change the BME default behavior to use the Java Source Code Editor:

1. Select **Tools > Options**.
2. In the Options window (Figure 14-1), expand the BusinessWare Options and select Process Modeler Options.
3. Set **Double Click to Code** to True.

Note: You also can control how operators are displayed in the Condition Builder by setting the **Display Operators As Text** property in this dialog box.

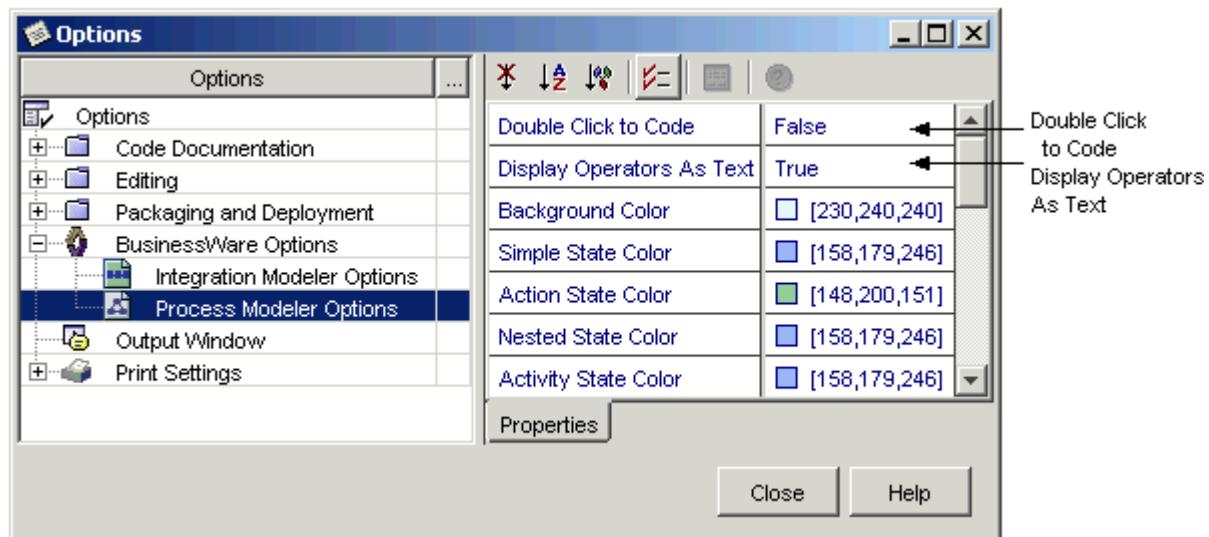


Figure 14-1 Setting Double-Click to Display Source Code Editor

PROCESS MODEL CODE CONSTRUCTION

Using the Action Builder

Table 14-3 summarizes the double-click accelerators you can use to quickly display the Action Builder, Condition Builder, or Source Code Editor.

Table 14-3 Shortcuts for Displaying Code Builders or Source Code Editor

Accelerator	Global Option Setting for Double Click to Code	
	False	True
Double-click on state	Action Builder for entry action	Source Code Editor at entry action method
Shift+double-click on state	Source Code Editor at entry action method	Action Builder for entry action
Control+double-click on state	Action Builder for exit action	Source Code Editor at exit action method
Shift+Control+double-click on state	Source Code Editor at exit action method	Action Builder for exit action
Double-click on transition	Builder dialog for editing the Trigger Event, Condition and Action	Source Code Editor at action method
Shift+double-click on transition	Source Code Editor at action method	Action Builder
Control+double-click on transition	Condition Builder	Source Code Editor at condition method
Shift+Control+double-click on transition	Source Code Editor at condition method	Condition Builder

Note: You cannot use these shortcuts on a nested state or a transformer state. Double-clicking on one of those states displays its nested model. Use the context menu to add action code to a nested state. Action code is not permitted on transformer states.

USING THE ACTION BUILDER

With the Action Builder, you construct code simply by selecting from lists of prebuilt actions (Figure 14-2). If the action includes parameters, you set the parameter values, either manually or with the aid of the Action Picker.

You may or may not need to specify the Incoming/Outgoing Events type:

- **State actions**—When you are constructing action code for a state, you can specify whether the action should be applied to all events that transition to/from the state or only to events of a certain type. This step is necessary because a state may have multiple transitions leading into or out of it, each triggered by different event types.

- **Transition actions**—When you are constructing action code for a transition, the type of event that triggers that particular transition is already defined, and the action is applied to those triggering events.
- **Model default action**—When you are constructing the default action code for a model, the action is applied to all events that fail to trigger any of the transitions out of a state. You do not specify an event type.

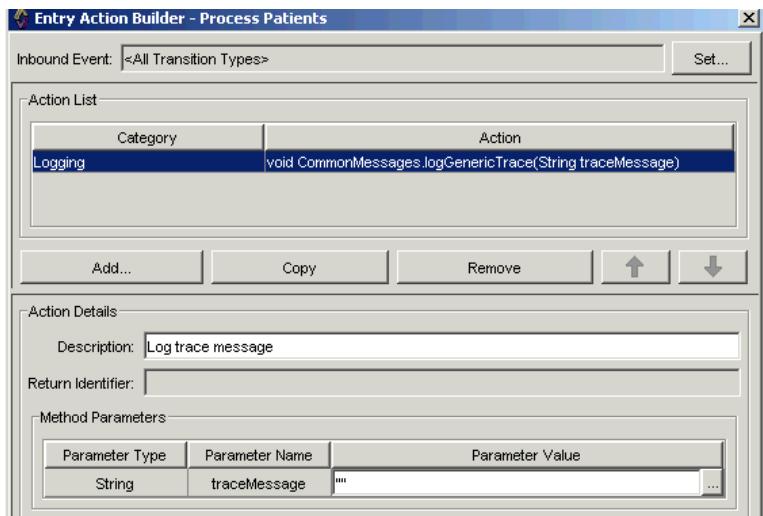


Figure 14-2 Action Builder

The Action Builder is context-sensitive. Depending on where you are in the model when you launch the Action Builder, it will display the appropriate actions and parameters for that context. For example, the actions listed for publishing to a port will include the names of the ports available in your model.

IMPORTANT: Code that you generate using the Action Builder can be edited or deleted only in the Action Builder. Although you can view this code in the Action Editor and in the Java Source Code Editor, you cannot edit it in those tools.

To build action code:

1. For state actions, select the **Incoming/Outgoing Events** you want the action code applied to.
 - <All Transition Types> applies the code to all events that trigger any transition to the state (entry action) or from the state (exit action).

- Selecting an event type applies the code to events that trigger one particular transition. It also sets the scope for other options in the builder. For example, event parameters for that event type will now be available for selection in the Action Picker.
- 2.** Click **Add...** and select an action from one of the categories in the Action Picker.
- A single row is displayed in the Action List, showing the selected action. For a description of the various action categories, see “[Action Categories and Common Actions](#)” on page 14-22.
- If the action you choose takes parameters, fields for entering those parameters are displayed in the Action Details pane, along with information on the data type of each parameter.
- Select Xquery/XPath to query the XML payload of an XML envelope. See “[Using the XQuery Builder](#)” on page 14-16 for more information.
- 3.** Enter values for each of the parameters. You can type in the parameter value or use the Action Picker to locate and insert a method that will set the parameter value or a model variable for the parameter value ([Figure 14-3](#)). To use the Action Picker:
- a. Click in the **Parameter Value** field.
 - b. Click the browse button  to open the Action Picker.

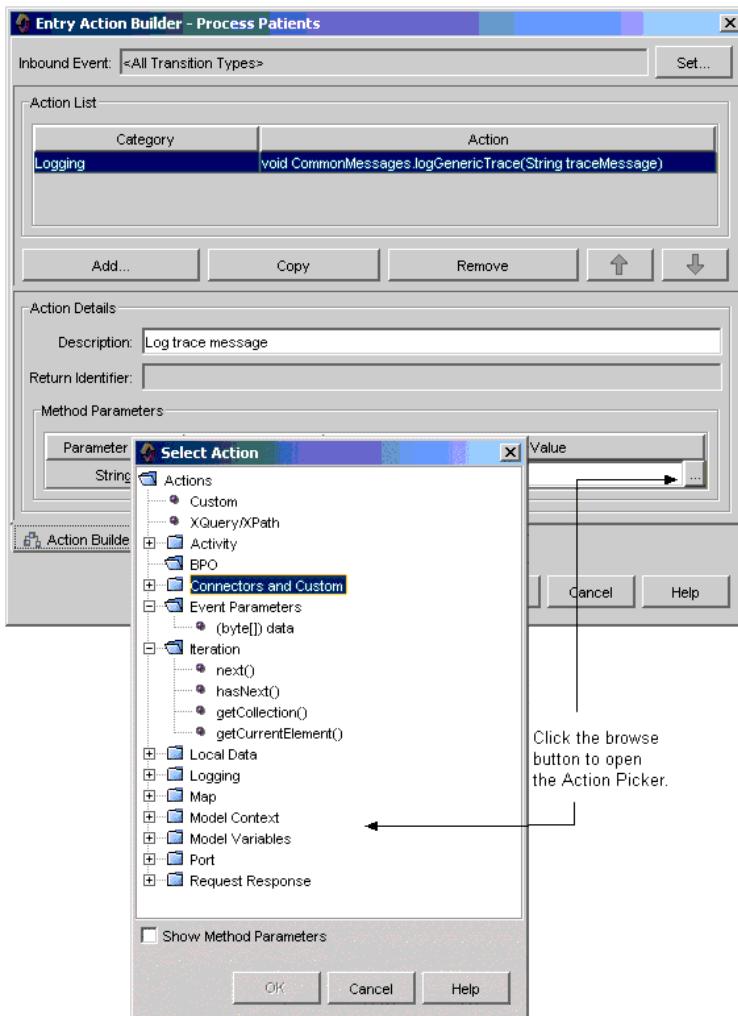


Figure 14-3 Insert Code Using the Action Picker

- c. Select a **Category**. See “Action Categories and Common Actions” on page 14-22 for descriptions of each category.

Tip: Move the mouse over an action to view details for that action.

- d. Select (or double-click) an action or variable from that category.

Note: For String parameters, click between the quotation marks before entering the value unless you are inserting a variable or another action. Code actions and variables should never be enclosed in quotes.

If you add an XQuery/XPath action to the Action List, you can specify an XQuery file or and XQuery string as the payload in the Action Details.

When specifying an XQuery string, click **Edit...** to open the XQuery/XPath Builder. See “[Using the XQuery Builder](#)” on page 14-16 for more information.

4. Optionally, enter a **Description** for your action.

Descriptions are written as comments in the code.

The **Return Identifier** is an auto-generated variable name that is used to store the return value of an action. All variable names are unique. You can override the auto-generated variable name if you want to specify your own variable name.

5. Add more actions to the Action List as desired, repeating [step 1](#) through [step 4](#).

Actions will be executed in the order listed (top action first). You can reorder the list using the arrow buttons or delete an action using the **Remove** button.

6. When your list of actions is complete, click **OK**.

Tip: As you build your action list, you can view the generated code by clicking the Action Editor tab. Your code is displayed at the top in the Generated Action Code pane and cannot be edited in the Action Editor.

USING THE ACTION EDITOR

The Action Editor is designed for modelers with a basic knowledge of Java syntax. It allows you to use additional Java constructs such as if-then-else and for statements. As you write code, you can identify and insert the appropriate methods and XPaths with the aid of the Action Picker.

The top pane displays code you have constructed using the Action Builder (if there is any). You can copy and paste to your user code from this generated code, but you cannot edit the generated code.

The lower pane is where you write your code. Anything you write here will be inserted after the generated code displayed in the top pane.

IMPORTANT: Code that you write in the Action Editor can be viewed and edited in the Java Source Code Editor. Code that you write in the Java Source Code Editor can be viewed and edited in the Action Editor.

To write code in the Action Editor:

1. For state actions, select the **Incoming/Outgoing Events** you want the action code applied to.
 - <All Transition Types> applies the code to all events that trigger any transition to the state (entry action) or from the state (exit action).
 - Selecting an event type applies the code to events that trigger one particular transition. It also sets the scope for other options in the builder. For example, event parameters for that event type will now be available for selection in the Action Picker.
2. Click in the User Action Code pane and begin writing your Java statements.
3. To use the Action Picker to insert a method name or Xpath:
 - a. Click **Insert....**
 - b. Select a **Category**. See “[Action Categories and Common Actions](#)” on [page 14-22](#) for descriptions of each category.
 - c. Select (or double-click) an action or variable from that category.
4. When finished writing your action code, click **OK**.

PROCESS MODEL CODE CONSTRUCTION
Using the Action Editor

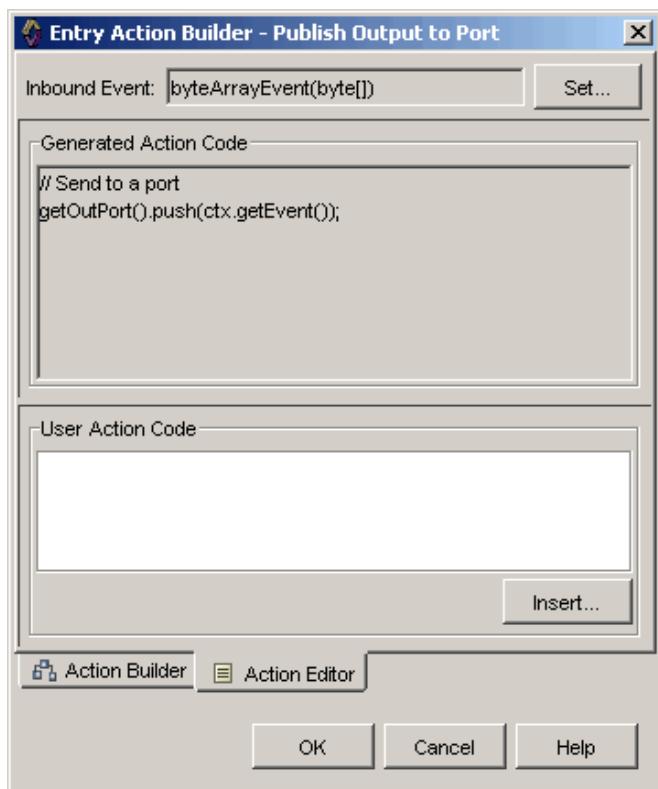


Figure 14-4 Action Editor

USING THE CONDITION BUILDER

You use the Condition Builder to build a list of conditional expressions for a given transition. The transition will fire only if the event is a valid trigger *and* the conditions are met.

To access the Condition Builder:

1. Select the transition whose condition you want to create or change.
 2. In the Properties window, select the Condition property, and browse to open the Condition Builder ([Figure 14-5](#)).
- or
- Select the transition whose condition you want to create or change.
- a. Right-click.
 - b. Select **Tools > Condition....**

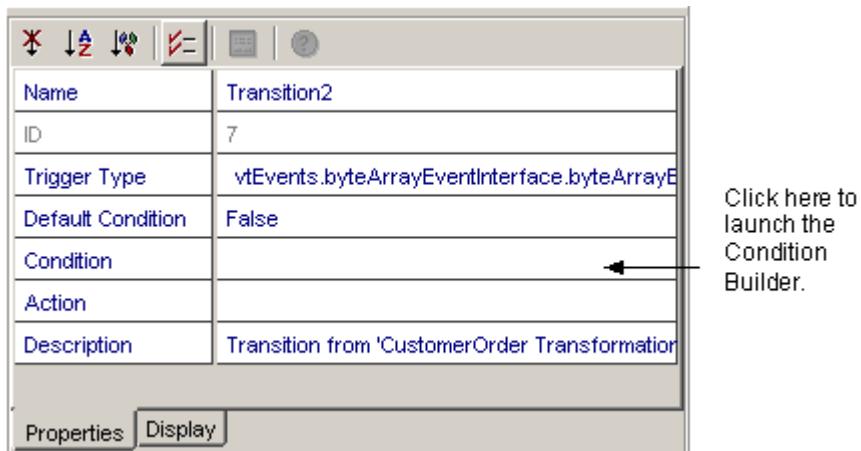


Figure 14-5 Condition Builder Access

By default, the Condition Builder displays operators as text. If you prefer symbols, select **Tools > Options > BusinessWare Options > Process Modeler Options** and set Display Operators As Text to false. The Options dialog box is shown in [Figure 14-1](#).

You build expressions in the upper pane, using the Action Picker to define the operands and selecting the operator from the Operator Picker. Optionally, you can edit or write condition expressions in the Operand Details pane.

IMPORTANT: The edits you make in the Operand Details pane are not reflected in the expression list in the top pane. Any code you create in the Condition Builder can be viewed in the Source Code Editor, but it can only be changed or deleted in the Condition Builder.

To build conditional expressions:

1. Select **Match All Expressions** or **Match At Least One Expression** in the Code Builder ([Figure 14-5](#)).
 - Match All Expressions joins all expressions in the list with AND.
 - Match At Least One Expression joins all expressions in the list with OR.

To use both AND and OR joins in the expression list, group expressions as explained in [step 5](#).
2. Click **Add**.
3. Use the Action Picker to define the operands in the expression:
 - Click in an operand field.
 - Select a category of actions.
 - Select one of the methods available from that category of actions.
 - Click **OK**.
4. To specify the operator, click on the displayed operator and then click the browse button. In the Operator Picker, select an operator and click **OK**.

Condition code is displayed under Operand Details as you build the list of expressions.

[Figure 14-6](#) shows the Operator Picker.

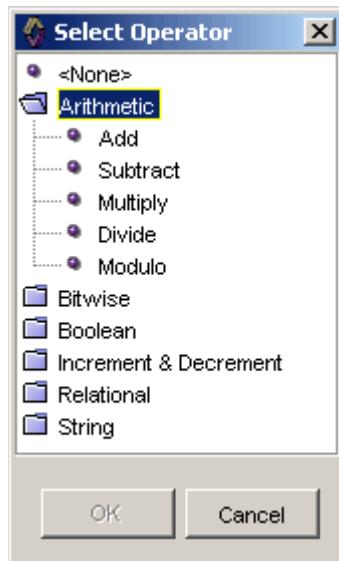


Figure 14-6 Operator Picker

5. To group expressions and form a compound expression:
 - a. Select two adjacent expressions.
 - b. Click **Group**.
Grouping maintains the AND/OR relationship within the group.
 - c. Select the operator for the new compound expression.

Note: To separate the expressions, select the group and click **Ungroup**.
6. Continue adding expressions as needed.

Conditions will be evaluated in the order listed (top expression first). You can reorder the list using the arrow buttons or delete an expression using the **Remove** button.
7. Optionally, edit the condition code in the Operand Details pane.
 - Edits you make here are not reflected in the expression list at top.
 - You can use the Action Picker to insert methods or model variables in the condition code as you edit.
 - If you add an XQuery/XPath action to the Action List, you can specify an XQuery file or and XQuery string as the payload in the Action Details. When specifying an XQuery string, click **Edit...** to open the XQuery/XPath Builder. See “[Using the XQuery Builder](#)” on page 14-16 for more information.

PROCESS MODEL CODE CONSTRUCTION
Using the XQuery Builder

8. When satisfied with your condition expressions, click **OK**.

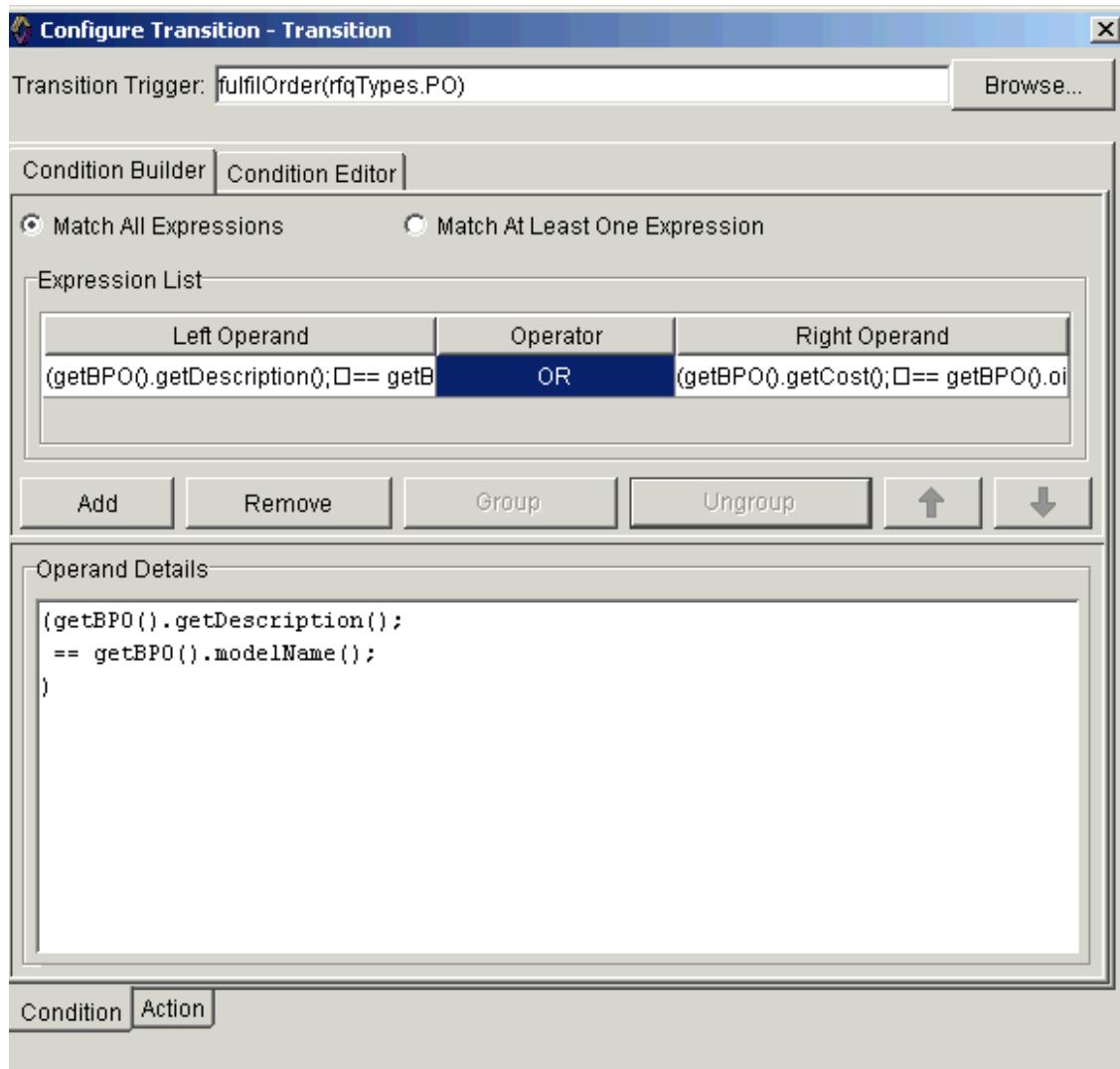


Figure 14-7 Condition Builder

USING THE XQUERY BUILDER

Use the XQuery Builder to build and edit XPath and XQuery expressions. You can use the XQuery Builder when creating action code for the following components in the BME:

you can use the XQuery Builder with the following BME components:

- Action states
- Simple states
- Transitions

To access the XQuery Builder from the Action Builder:

1. Click **Add...** and select the **XQuery/XPath** action from the Select Action dialog. The selected action appears in the Action List pane Action Details pane is populated.
2. In the Action Details pane, select **XQuery String** and click **Edit....** The XQuery Builder opens.

To open the XQuery Builder from the Condition Builder:

1. Click **Add...** to add an expression to the Expression List pane.
2. Click the browse button in either of the Operand fields and select the **XQuery/XPath** action from the Select Action dialog. Click **OK**.
3. With an Operand selected, click **Edit...** in the Operand Details pane. The XQuery Builder opens.

Note: In both cases, you can also select the XQuery File option and click **Browse...** to select an XQuery file from the Choose XQuery File dialog box.

To build expressions:

1. Click the browse button next to the Source Schema box to open the Select XML Schema / DTD / Instance dialog. Select an XML schema, DTD, or an XML instance.

The structure for the selected input appears in the Source Schema pane and the Root Element list is populated with all top-level elements from the selected input.

2. From the **Root Element** drop-down list, select the root element.

Note: There are no visual clues identifying the root element. You must either know what the root element is or read the source file to determine the root element.

3. Select XQuery functions and operators from the Functions and Operators pane.
4. Drag and drop the selected function or operator from the Functions and Operators pane to the Mapping pane.
5. Replace function parameters with the XPath expression by dragging and dropping elements from the Source Schema pane to the Mapping pane.

6. Click **Namespace Map** to edit the Namespace for the selected source schema. See the ????? for more information on editing namespaces.
7. Click OK.

Note: AXT validates the statements in the Mapping pane when you click OK. In case of an error, a message box is displayed with a brief description of the error message.

If there are no errors, the text area corresponding to the XQuery String option in the Entry Action Builder or Condition Builder dialog is populated with the Namespace definition (if any) and XPath expressions specified in the Mapping pane in the XQuery Builder dialog.

Figure 14-8 shows the XQuery Builder.

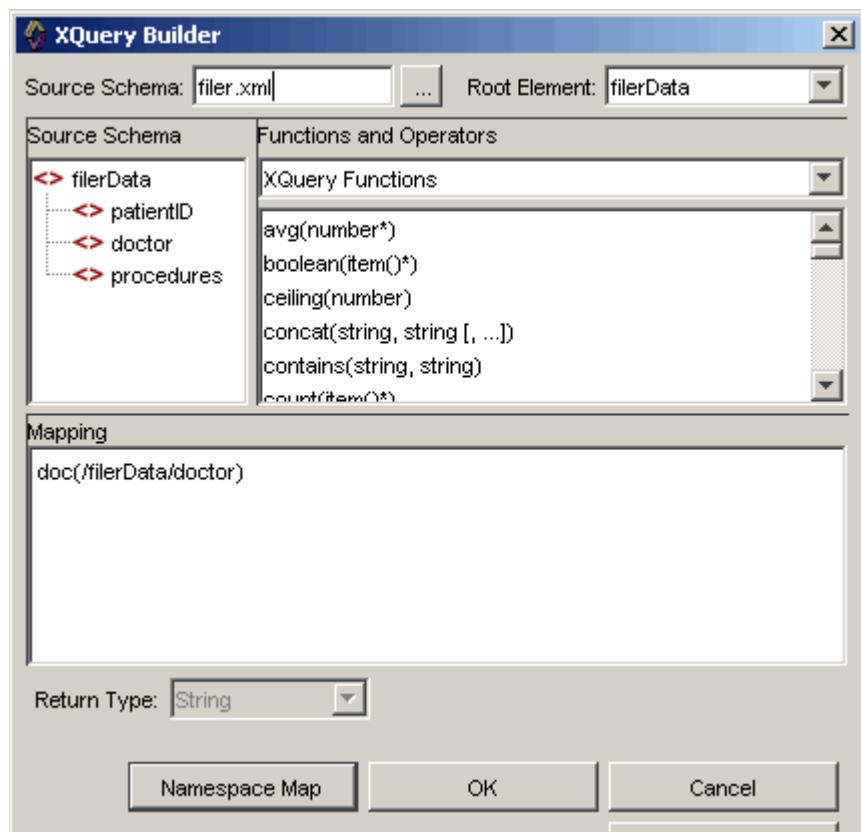


Figure 14-8 XQuery Builder

Figure 14-9 shows the result of the XQuery Builder in the Action Builder.

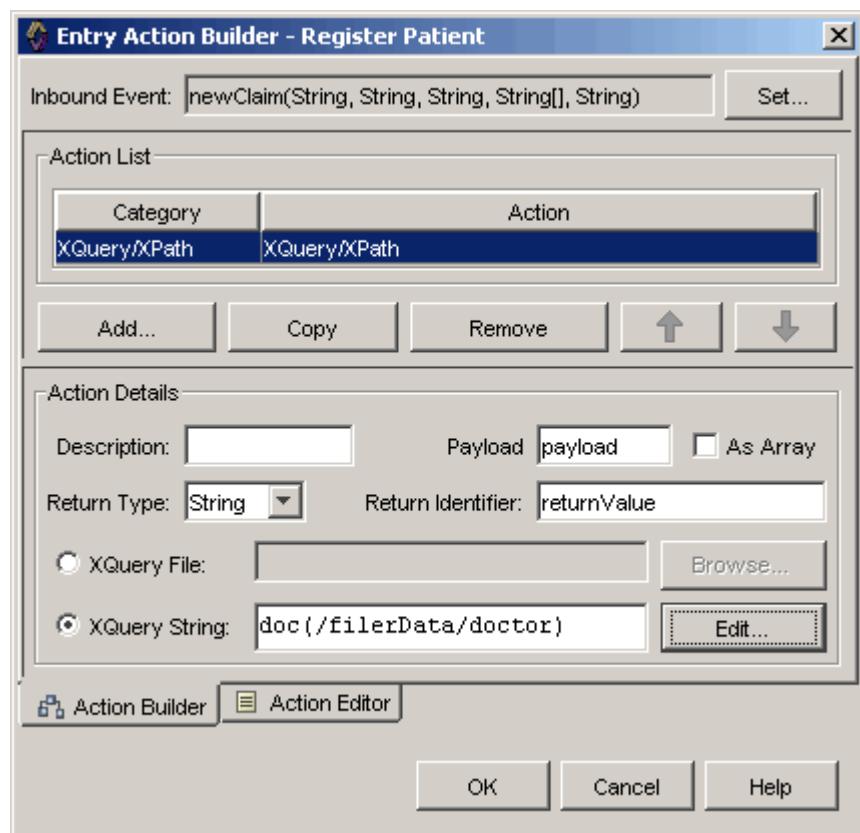


Figure 14-9 XQuery Expression in the Action Builder

USING THE SOURCE CODE EDITOR

The Java Source Code Editor offers you the freedom to write code blocks that meet your special needs and goals. It also provides features to help you code quickly and correctly. However, you should be an experienced Java programmer to use the Source Code Editor.

To access a section of code quickly, navigate the source code using the context-sensitive **Tools** menu for a particular state or transition from within the graphical Process Model Editor, or use the shortcuts listed in [Table 14-3](#).

Two especially helpful features of the Source Code Editor are illustrated in **Figure 14-10**:

- **Code completion**—you type the first few characters of an expression, and the editor displays a list of possibilities for completing the expression. Simply select the appropriate one. For information on updating the parser database, see the *BusinessWare Online Help*.
- **Color highlighting**—code is colored according to syntax. For example, keywords are shown in blue and comments in light grey. Generated code (read-only) has a light blue background. You can customize the color highlights.

To open the Source Code Editor:

1. Double-click a Java Source, IDL Source, or other text file object in the Explorer.
or
Right-click on a process model in the Explorer and select Edit.

To use code completion:

1. Type the first few characters of an expression.
2. Press <Ctrl+space> or simply pause after typing a period.
A code completion box appears, listing all the expressions that begin with the string you typed.
3. Select the expression you want, and press **Enter**.
4. If the method you selected takes parameters:
 - a. Edit the replaceable text that is supplied for the first parameter.
 - b. Type a comma to reopen the list of parameters.

To customize color highlighting:

1. Select **Tools > Options** and select **Editing > Editor Settings > Java Editor**.
2. Click in the value field for **Fonts and Colors**.
3. Make desired changes in the Property Editor for Fonts and Colors.

Figure 14-10 Source Code Editor

The default color settings for some key syntactical elements are listed in [Table 14-4](#).

Table 14-4 Default Colors for Syntactical Elements

Syntax	Foreground Color	Background Color
comment	grey	white
guarded block (read-only)	black	light blue (225,236,247)
Java keyword	blue	white
Java method call	black	white
error in sources	white	red
string literal	magenta	white
character literal	green (0,178,0)	white
numerical literal	red	white
selected text	white	light grey

ACTION CATEGORIES AND COMMON ACTIONS

Table 14-5 briefly describes the action categories used in the Action Builder, Action Editor, and Condition Builder. The actions available in each category will vary somewhat, depending on the context in which you launch the tool. For each category, a few of the most commonly used actions are listed. Detailed descriptions of all the actions are provided in the *BusinessWare Programming Reference*.

Note: If you have installed other Vitria products, such as the RDBMS Connector, Vitria automatically installs additional action categories with supplemental components, which appear in the code builders.

To add custom actions in the code builders, see the *BusinessWare Administration Guide* for information about the registeraction command-line tool.

Table 14-5 Action Categories and Some Representative Actions

Item	Description and Examples ^a
BPO Actions	
Description	Methods for manipulating BPO data, including methods for the dynamic overrides on activity states. These methods are available only if a valid BPO is specified in the model.
Examples	void getBPO().setTimer(String state, String model, long duration); dynamically sets the timer on a state
	void getBPO().setPerformerLast(); one of the dynamic overrides for activity states, used to set the performer for the activity
Connectors and Custom	
ChannelConnectorLib	
Description	Transformation library for common Channel Connector events. For details, see com.vitria.connectors.common in the <i>BusinessWare Programming Reference</i> .
Examples	EventBody ChannelConnectorLib.createDynamicTargetInfo()

Table 14-5 Action Categories and Some Representative Actions (Continued)

Item	Description and Examples ^a (Continued)
CommonTransformationLib	
Description	<p>Transformation library for common BusinessWare events.</p> <p>For details, see <code>com.vitria.connectors.common.CommonTransformationLib</code> in the <i>BusinessWare Programming Reference</i>.</p>
Examples	<pre>EventBody CommonTransformationLib.byteArrayToString(EventBody eventBody) converts a byteArrayEvent to a stringEvent</pre> <pre>EventBody CommonTransformationLib.eventParamsToStringArray(EventBody eventBody) converts the parameter values of an incoming event into a string array. Basic types are mapped into a single array element and nested structures are flattened into multiple array elements.</pre>
EmailTransformationLib	
Description	<p>Transformation library for Email Connector events.</p> <p>For details, see <code>com.vitria.connectors.email.EmailTransformationLib</code> in the <i>BusinessWare Programming Reference</i>.</p>
Examples	<pre>EventBody EmailTransformationLib.emailMessageToString(EventBody, String delimiter) converts a structured event to a stringEvent</pre> <pre>EventBody EmailTransformationLib.stringToEmailMessage(EventBody, String delimiter) converts a stringEvent to a structured event, using the delimiter to separate the message</pre>
FileTransformationLib	
Description	<p>Transformation library for File Connector events.</p> <p>For details, see <code>com.vitria.connectors.file.FileTransformationLib</code> in the <i>BusinessWare Programming Reference</i>.</p>
Examples	<pre>EventBody FileTransformationLib.asciiFileToString(EventBody eventBody) converts an asciiFileEvent to a stringEvent</pre> <pre>EventBody FileTransformationLib.bytesToFile(EventBody eventBody, String fileName) converts a byteArrayEvent to a dataFileEvent</pre>

Table 14-5 Action Categories and Some Representative Actions (Continued)

Item	Description and Examples ^a (Continued)
HttpTransformationLib	
Description	Transformation library for HTTP Connector events. For details, see <code>com.vitria.connectors.http.HttpTransformationLib</code> in the <i>BusinessWare Programming Reference</i> .
Examples	<code>EventBody HttpTransformationLib.postToXmlEvent(EventBody eventBody)</code> converts an HTTP post event to an XML event
	<code>EventBody HttpTransformationLib.xmlEventToPost(EventBody eventBody)</code> converts an XML event to an HTTP post event
QueueConnectorLib	
Description	
Examples	<code>createDynamicTargetInfo()</code>
Event Parameters	
Description	
Local Data	
Description	Methods for manipulating data in a Local Data object. These methods are available only if the model is configured to store transient data in a Local Data object. The base interface Local Data does not have any methods. The following is an example of the special <code>bpe.LocalDataPrimitiveTypesMap</code> which is provided as the default type. The implementation provides a HashMap with a key-value pair for different primitive types.
Examples	<code>void getLDO().setString(String keyStr, String valStr);</code> sets the key-value pair in the map with a string value.
	<code>void getLDO().setStringArray(String keyStr, String[] valStr);</code> sets the key-value pair in the map with a string array value.
	<code>void getLDO().setInt(String keyInt, int valInt);</code> sets the key-value pair in the map with an integer value.
	<code>void getLDO().setIntArray(String keyInt, int[] valInt);</code> sets the key-value pair in the map with an integer array value.
	<code>String getLDO().remove(String keyStr);</code> removes the entry from the map.

Table 14-5 Action Categories and Some Representative Actions (Continued)

Item	Description and Examples ^a (Continued)
Logging	
Description	Methods for generating trace, warning, error, and exception messages in the Integration Server logger(s). Typically, however, you will want to place these log messages in a conditional “if” statement to test log levels. You do not want to be logging too many messages if the loggers are turned down to low levels. The samples provided with BusinessWare illustrate this approach.
Example	<pre>void CommonMessages.logGenericTrace(String traceMessage);</pre> used to log a simple String message to the logger(s)
Map	
Description	Library for manipulating instances of java.util.Map classes.
Example	<pre>boolean containsKey(Map map, Object key)</pre> Returns if a key is found in a map. <pre>containsValue(Map map, Object value)</pre> Returns if a value is found in a map.
Iteration	
Description	Common iteration-related actions available during process execution. These actions are accessible for condition and action code when iteration is enabled on a state.
Example	<pre>Object getCurrentElement()</pre> Get the current element within an iteration, or null if there one does not exist. <pre>Collection getCollection()</pre> Get the original collection of the iteration. <pre>Object next()</pre> Advance to the next element. <pre>boolean hasNext();</pre> Determine if there is another element.

Table 14-5 Action Categories and Some Representative Actions (Continued)

Item	Description and Examples ^a (Continued)
Model Context	
Description	Methods for setting information in the current invocation context and for acting on the current execution state.
Examples	<pre>void ctx.setEvent(EventBody event);</pre> <p>changes the current event that will be used to evaluate outgoing transitions, useful for chained action states</p> <pre>void ctx.setReturnValue(Object returnValue);</pre> <p>sets the return value expected on the original request into the component. This must be set in some action code before the model comes to rest if the original request has a non-void return type. Be careful to set the original return value expected from the incoming component request. If you mutate the current event (e.g., with the <code>setEvent()</code> call), you still need to return the value originally expected by the runtime.</p> <pre>void ctx.createObject(DataObject obj);</pre> <p>marks a DataObject for creation in the database when the transaction commits</p> <pre>void ctx.deleteObject(DataObject obj);</pre> <p>marks a DataObject for deletion from the database when the transaction commits</p> <pre>void ctx.updateObject(DataObject obj);</pre> <p>marks a DataObject for update in the database when the transaction commits</p> <pre>void ctx.abort();</pre> <p>aborts the current process</p> <pre>void ctx.skip();</pre> <p>specifies that the current event not be redelivered in case of an abort. An exception indicates that the source flow is not capable of providing the skip functionality, for example, if it is a synchronous invocation.</p> <pre>void ctx.stopCurrentProject();</pre> <p>stops the project of which the component is a part. Only the part of the project that is running in the current server is stopped. Stop is asynchronous.</p> <pre>void ctx.restartCurrentProjectSourceFlows();</pre> <p>stops and restarts the source flows in the project of which the component is a part. Restart is asynchronous.</p>

Table 14-5 Action Categories and Some Representative Actions (Continued)

Item	Description and Examples ^a (Continued)
Port	
Description	<p>Methods for making invocations on ports. In the case of a typed port, these methods will return the typed object on which to make the invocation. For untyped ports, this will return a <code>RequestListener</code> on which you can push an <code>EventBody</code> object.</p> <p>For each output port in a process model, a method is generated that matches the properties given to the port:</p> <pre>Interface getName()</pre> <p>where <code>Interface</code> is the type defined in the Port Types property, and <code>Name</code> is the port name; for typed invocations.</p> <p>Alternatively, for untyped ports:</p> <pre>RequestListener getName();</pre> <p>where <code>Name</code> is the port name; for untyped invocations.</p> <p>The methods above describe the static port case. If the port is connected to a Simple Output Proxy and you choose to make your output port a “dynamic” port, by setting the “Dynamic Port” property True, the generated methods will be slightly different:</p> <pre>Interface getName(Object mapInfo)</pre> <pre>RequestListener getName(Object mapInfo);</pre> <p>where <code>mapInfo</code> is an object that can be used to pass data from your action code to the callback method in your Dynamic Map Class defined on your connected Simple Output Proxy. See Chapter 6, “Integration Model Basics” for additional information.</p>
Request Response	
Description	<p><code>void getToAccountManagement().creditAccount(String customerId, String orderId, long quantity, float amount);</code> typed invocation</p> <p><code>RequestListener getToAccountManagementListener().push(EventBody event);</code> untyped invocation-if the port was not typed</p>
XQuery/XPath	
Description	For more information on the XQuery Builder, see “Using the XQuery Builder” on page 14-16

- a. The `EventBody` interface, which is used in many of the actions listed here, is in the `com.vitria.fc.flow` package. For more information, see the *BusinessWare Programming Reference*.

MODEL VARIABLES

Vitria provides several model variables that you can use in your action code to retrieve and insert various types of information about the event. These variables are available from the Action Picker in the Action Builder, Action Editor, Condition Builder, and Condition Editor. You can also use them if you write code in the Source Code Editor.

Table 14-6 defines the model variables and shows how they are listed in the code builders. It also indicates whether you can use the variable in transition actions and conditions, state actions, or model default actions.

Table 14-6 Model Variables Used in Action Code

Variable (in Code Builders)	Definition	Transition	State	Model Default Action
<code>_thisModel</code> (String) This Model	String <code>_thisModel</code> Name of the model	X	X	X
<code>_thisState</code> (String) This State	String <code>_thisState</code> Name of the current state. Used only in the entry or exit actions of states.		X	
<code>_fromState</code> (String) This State	String <code>_fromState</code> Name of the origin state of a transition. Used only in the action or condition code for transitions.	X		
<code>_toState</code> (String) This State	String <code>_toState</code> Name of the destination state of a transition. Used only in the action or condition code for transitions.	X		
<code>_eventBody</code> (EventBody) Event	EventBody <code>_eventBody</code> Event object in raw, unmarshalled format. Useful when you do not need to recreate the event and want to push it to a port for processing by a downstream component. For a description of the EventBody structure, see the <i>BusinessWare Programming Reference</i> .	X	X	X
<code>_eventParams</code> (Object[]) Event Parameters	Object[] <code>_eventParams</code> Array of objects passed in the original event. Useful with default transitions where the event is not known at compile time.	X	X	X

INITIALIZING A PROJECT AND PROVIDING PROJECT GLOBAL DATA

There may be a need to maintain some global data storage within your project for use at runtime. This information can be shared by all threads running within the context of the project. Sometimes it is convenient to cache information that will be accessed often. For example, if a small lookup table containing region codes for assignment during the processing of new orders will be necessary to access each time a new order arrives, this information can be loaded on project startup and be easily accessed during the processing of each order. Be aware that the decision to maintain a global data cache must be carefully constructed, as to ensure that:

- Memory requirements are met
- The code is thread-safe if any data is to be modified
- Data recovery is not necessary

This section describes how to provide the appropriate support for the project global data for use within your project. It involves providing a special class within your project, configuring it, and accessing the data within it from action code. The interface that must be implemented is `com.vitria.container.client.ProjectInit`, which contains methods for initialization on project startup and cleanup on project shutdown.

You must provide an implementation for this interface, which is accessible from within the project, and link it to your project via the project properties ([Figure 14-11](#)). In the Explorer, click on the Project object to display the Properties Window and select the Runtime tab. There are two properties of interest:

- Project Init Class defines the fully qualified class name of your implementation class.
- Project Init Data defines a configurable string that will be passed to the `initialize` method on startup. Typically this may contain a path name from which further information may be loaded.

IMPORTANT: Global data loaded via `ProjectInit` remains in memory until the project is shutdown, so the global data size should be designed with this fact in mind. Also, you should be aware that all project objects may not be loaded yet when `ProjectInit` is called, as it is called very early in the startup cycle. You cannot assume that all objects within the project are available when your custom `ProjectInit` class is executing.

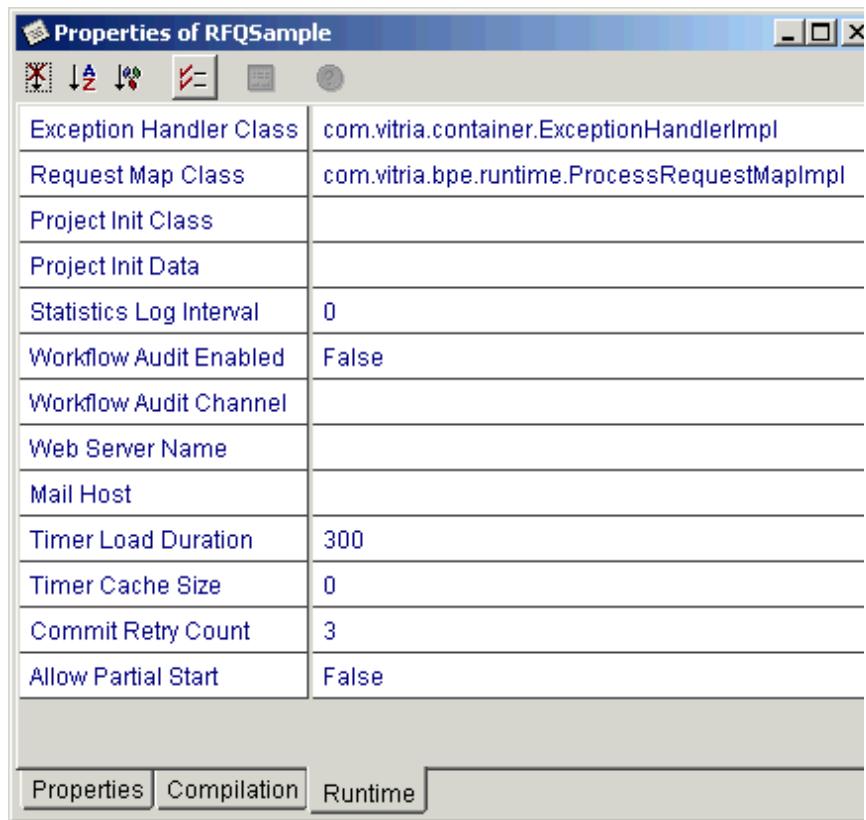


Figure 14-11 Project Init Class and Project Init Data Properties

WHAT TO IMPLEMENT IN YOUR PROJECT INIT CLASS

You must implement the following in your ProjectInit class:

- The method `initialize`, which takes a string input and uses it in the execution of the method at project startup:

```
void initialize(java.lang.String initStr) throws
java.lang.Exception
```

- The method `cleanup`, which performs any cleanup on items you created or initialized during the execution of `initialize`.

```
void cleanup()
```

- A public default constructor

The `initialize` and `cleanup` methods are invoked automatically by BusinessWare runtime at project startup and shutdown, respectively.

The input string to `initialize` comes from the Project Init Data property setting in the project properties. So, if you specify a class in the Project Init Class property, you can optionally supply a string in Project Init Data.

The input string can be anything you want, as long as the `initialize` method is expecting it. It could be a simple configuration value, it could be the full path name of a configuration file that `initialize` is going to process, or it could be something else. In the OrderProcess Sample located under the BusinessWare subdirectory `installdir\sample\modeling\OrderProcessSample`, the string is the name of a file containing region data that is loaded to be made available as global data to a process component in the project.

In the implementation of the interface, you can use a constructor and various (private) helper methods to process the input data or to carry out other initialization tasks.

However, if you want to make global data available to components in the project, you must implement public methods to provide access to the global data.

ACCESSING GLOBAL DATA IN YOUR PROCESS MODEL CODE

You can retrieve the global data by making calls in the condition code or action code in the process model. To do this, you use the `getProjectInit` method on the context and cast it to your implementation class. This returns your `ProjectInit` object, on which you then invoke the special method you implemented for retrieval.

In the following example, all of this is done in a single statement:

```
String Data =  
( (MyInit)ctx.getProjectInit() ).getMyData();
```

where `MyInit` is the implementation class, `ctx` is the context object automatically available in all action and condition code, and `getMyData` is the implemented method.

EXAMPLE IMPLEMENTATION

The following minimal sample shows mainly the required elements of an implementation of ProjectInit.

For a full sample showing a realistic scenario, see the file RegionInit.java located in the com\vitria\samples\orderprocesssample\init within the OrderProcessSample project directory. You must first import the sample before the files are visible. The sample is located at *installdir\samples\modeling\OrderProcessSample*.

```
package init;

import com.vitria.container.client.ProjectInit;

public class MyInit implements ProjectInit {

    private String myData_;

    // you must have a public default constructor
    public MyInit() {
    }

    public void initialize(String initStr) {
        myData_ = initStr;
    }

    public void cleanup() {
        myData_ = null;
    }

    public String getMyData() {
        return myData_;
    }
}
```

In the sample, the package name is *init* by convention. You can use any name you want but it is good practice to identify this as initialization code. The import statement and class declarations are always required, as shown. In a real implementation any number of Java and BusinessWare classes would be imported as well depending on the requirements.

SAMPLE ACTION CODE

This section describes a simple example of an entry action for a state that has three transitions leading into it. Each transition is triggered by a different type of event, and the entry action applies different processing and generates different log messages, depending on which event enters the state. The name of the state is OrderProcess. The events that can trigger the three transitions are dayTraderOrder, retailOrder, and institutionOrder. The BPO associated with this model is named myTrade.

During process component construction in the BME, the BME code generator automatically generates certain code that is useful for handling an event when the code generator finds an event transition or a state that receives an event transition. This generated code extracts the event data for you.

Notice that the data for every parameter in the event is automatically copied to a local variable within the action or transition code, so you need simply use that variable as needed in your custom code. For example, if a transition is triggered by an event `order.OrderEvents.newOrderItem` that has the string parameters, `orderId`, `customerId`, `totalItems`, `itemType`, `price`, and `quantity` the following code is generated for you automatically:

```
public void
process_com_vitria_samples_orderprocesssample_types_order
_OrderEvents_newOrderItem_At_State_Entry_3(ProcessModelCo
nText ctx) throws Exception {
    String orderId = (String)_eventParams[0];
    String customerId = (String)_eventParams[1];
    int totalItems = ((Integer)_eventParams[2]).intValue();
    String itemType = (String)_eventParams[3];
    float price = ((Float)_eventParams[4]).floatValue();
    int quantity = ((Integer)_eventParams[5]).intValue();
```

So, if you need to do something with the event data, it is already available for you to use. Notice that the formal parameter names used in the event definition are used to receive the parameter data. Notice also the indexing: the parameters are retrieved in the order of definition.

For illustration purposes, some actions were added using the Action Builder, and some actions were added using the Action Editor. Color-coding is used here to indicate how the code was created:

- Code created in the Action Builder is shown in *red **italics***.
- Code created in the Action Editor is shown in **blue bold**.

- Code that was automatically generated by the BME as the state and transitions were added to the process model is shown in black.

If you were to look at this code in the Source Code Editor, all the code created in the Action Builder or automatically generated by the BME would be on a light blue background, indicating it cannot be changed in the code editor. All the code created in the Action Editor would be on a white background, indicating it can be changed in the Source Code Editor.

```

/*
    Entry Action for State "OrderProcess"
*/
public void entryAction_S5(ProcessModelContext ctx) throws Exception {
    actionSetup(ctx);
    _thisState = "OrderProcess";
    if (_eventBody.checkEventDescription("stockTradeEvents.OrderEvents.institutionOrder")) {
        process_stockTradeEvents_OrderEvents_institutionOrder_At_State_Entry_5(ctx);
    } else if (_eventBody.checkEventDescription("stockTradeEvents.OrderEvents.retailOrder")) {
        process_stockTradeEvents_OrderEvents_retailOrder_At_State_Entry_5(ctx);
    } else if (_eventBody.checkEventDescription("stockTradeEvents.OrderEvents.dayTraderOrder")) {
        process_stockTradeEvents_OrderEvents_dayTraderOrder_At_State_Entry_5(ctx);
    }
    // Log Order ID
    CommonMessages.logGenericTrace("Order: " + getBPO().oid() + " is processed");

    //TODO: write your code here
}

//ACTION_END

/*
    Entry method called by state "OrderProcess"'s entry action
*/
public void process_stockTradeEvents_OrderEvents_dayTraderOrder_At_State_Entry_5(ProcessModelContext ctx) throws Exception {
    String ssn = (String)_eventParams[0];
    int accountNumber = ((Integer)_eventParams[1]).intValue();
    String symbol = (String)_eventParams[2];
    String tradeAction = (String)_eventParams[3];
    short quantity = ((Short)_eventParams[4]).shortValue();
    int price = ((Integer)_eventParams[5]).intValue();
    //TODO: write your code here
    if (myTrade.canBeDayTraded(symbol, ssn, accountNumber)) {
        myTrade.processDayTrader(ssn, accountNumber, symbol, tradeAction,
            quantity, price);
    } else {
}

```

```

        CommonMessages.logGenericError(symbol + " cannot be traded by " + ssn);
    }
} //ACTION_END

/*
   Entry method called by state "OrderProcess"'s entry action
*/
public void process_stockTradeEvents_OrderEvents_retailOrder_At_State_Entry_5(ProcessModelContext ctx) throws Exception {
    int accountNumber = ((Integer)_eventParams[0]).intValue();
    String symbol = (String)_eventParams[1];
    short tradeAction = ((Short)_eventParams[2]).shortValue();
    short tradeType = ((Short)_eventParams[3]).shortValue();
    short quantity = ((Short)_eventParams[4]).shortValue();
    int price = ((Integer)_eventParams[5]).intValue();
    // Log Retail account Info
    CommonMessages.logGenericTrace("Account " + accountNumber + " trade " + symbol);
    //TODO: write your code here
    myTrade.processSmallTrade(symbol, action, quantity, price, tradeType);
} //ACTION_END

/*
   Entry method called by state "OrderProcess"'s entry action
*/
public void process_stockTradeEvents_OrderEvents_institutionOrder_At_State_Entry_5(ProcessModelContext ctx) throws Exception {
    String institution = (String)_eventParams[0];
    String symbol = (String)_eventParams[1];
    float lowRange = ((Float)_eventParams[2]).floatValue();
    float upRange = ((Float)_eventParams[3]).floatValue();
    short tradeAction = ((Short)_eventParams[4]).shortValue();
    long quantity = ((Long)_eventParams[5]).longValue();
    // Log Institution Trade Info
    CommonMessages.logGenericTrace("Institution '" + institution +
        " trade " + symbol);
    //TODO: write your code here
    myTrade.processLargeTrade(symbol, tradeAction, quantity, lowRange, upRange);
} //ACTION_END

```

Notice that there are several different methods that are generated for a single entry action. They map to the different action code you can write for each triggering event to the state. Recall, when visualizing this in the Builder Tool, there is a separate choice box listing: All Transitions and each individual event separately. Choosing one of the events allowed you to customize action code for it alone.

The code above shows how this maps to the actual generated source code in the Source Code Editor. The first method, `entryAction_S5`, is invoked from the runtime. Notice that it checks each incoming event and appropriately invokes a matching method. The code that appears after the if statements is the All Transitions code. Each subsequent method is named for the event which triggers it. For example,

```
process_StockTradeEvents_OrderEvents_dayTrade  
Order_At_State_Entry_5
```

Exit actions have the same behavior. However, instead of mapping to the transition events into the state, they map to the transition events out of the state.

REQUESTRESPONSELIB VS. RESPONSELISTENERLIB

Both `com.vitria.container.syncinvocation.RequestResponseLib` and `com.vitria.connectors.requestResponseListenerLib` are used to hold a current event while waiting to receive a response from another process.

REQUESTRESPONSELIB

`RequestResponseLib` uses a channel or queue to send an event to the back-end system and listen to another channel or queue to receive the response. This type of architecture requires a unique ID to be used in identifying the waiting process. For example, the front-end system will send an event with a unique ID to the back-end system's incoming channel or queue. The back-end system will process that event and send a return event via its outgoing channel or queue with the same unique ID to the front-end system. Upon receipt the front-end system will use the unique ID to identify which waiting thread to unblock so that it can process the response. This also does not lend itself to clustering since when clustered the response may be consumed by a server in the cluster that is not the server waiting for the response.

RESPONSELISTENERLIB

`ResponseListenerLib` will publish an event containing the reference IOR of a CORBA object to a channel or queue for the back-end system to process. The back-end system can then directly call the front-end using the reference and send the response through this call-back. This type of architecture lends itself well to clustering.

Process model templates are prebuilt models that can be incorporated in any project. This chapter describes the process model templates supplied with BusinessWare and tells you how to create your own templates. It includes the following topics:

- Router Templates
- SimpleTransformer Template
- SimpleXQueryTransformer Template
- Creating a Process Model Template

ROUTER TEMPLATES

Two router templates are provided with BusinessWare:

- *RouteByPortName* routes events based on the component type attribute of the incoming event.
- *RouteByPortType* routes events based on the event interface.

Both routers are fairly simple process models consisting of a start state, one action state, and a terminator state. The entry action of the action state contains the logic whereby routing decisions are made.

[Figure 15-1](#) shows how a router might be used to push events to one of three process models.

PROCESS MODEL TEMPLATES

Router Templates

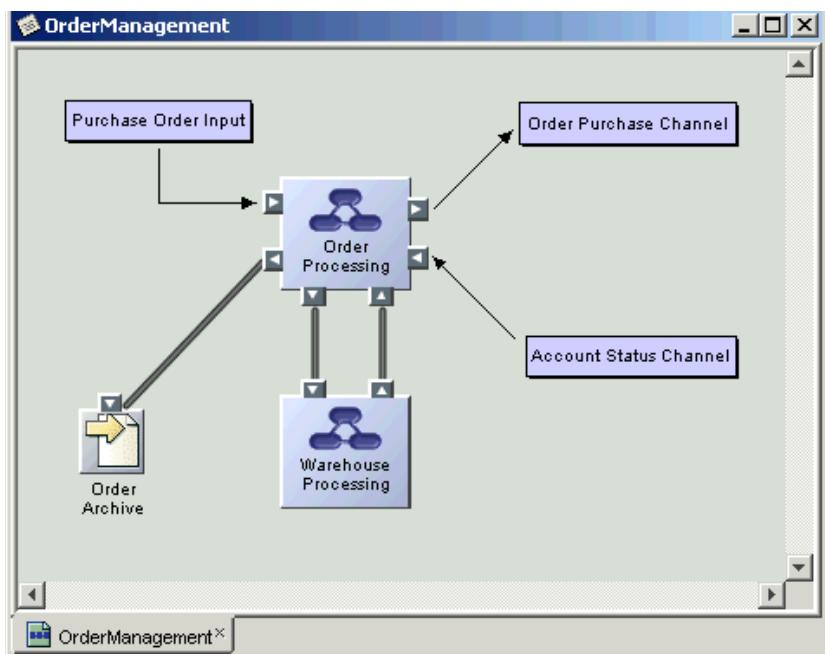


Figure 15-1 Integration Model with Router Component

The Order Router component in Figure 15-1 would be linked to one of the routers such as the RouteByPortName router. This router is shown in Figure 15-2.

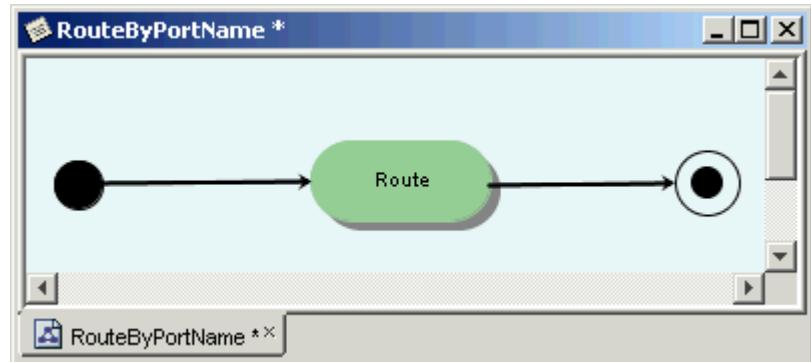


Figure 15-2 RouteByPortName Process Model

ROUTEBYPORTNAME ROUTER

By default, this router distributes events by matching the component type of the incoming event with the name of an output port. Each event is routed to at most one port. If there is no match, then the event will not be routed to any output port and an exception will be thrown.

For example, assume your router has three output ports named `toQuotes`, `toHardware`, and `toService`. If an incoming event has a component type of `toHardware`, then this event is pushed to the output port named `toHardware`.

The component type of the event is determined by the `getComponentType()` method of the `RequestMap` class. The default implementation of this class returns the `BPID.type` attribute of the event. If your events are not designed with the `BPID` structure, then you may need to create a custom `RequestMap` class and an implementation of the `getComponentType()` method that determines the component type based on other event parameters. See the section “[Request Map Class: Dispatching Events and Assigning BPOs](#)” on page 13-24 for information.

If you would like to use a different mechanism to compute the named output port for routing, change the implementation of the following method in the `RouteByPortName` model action code:

```
public String getName(ProcessModelContext ctx)
```

ROUTEBYPORTTYPE ROUTER

By default, this router pushes an incoming event to all output ports that have the same interface type as the event. Untyped ports accept all interfaces.

For example, assume your router has four output ports: one port has an interface type of `OrderEvents`, one has an interface type of `OrderStatusEvents`, and two are untyped. When an `OrderEvents` event is received, it is routed to three of the output ports: the port with the interface type of `OrderEvents` and the two untyped ports.

If the event’s interface does not match that of any output port (and there are no untyped ports), the event will not be routed to any output port and an exception will be thrown.

You will not usually need to change this default behavior. However, if necessary, you can customize the router by changing the implementation of the following method in the `RouteByPortType` model action code:

```
public String [] getNames(ProcessModelContext ctx)
```

ADDING A ROUTER MODEL

To use one of the prebuilt routers, all you have to do is add it to your project and link it to a process component in an integration model. Remember, you can link one router to multiple process components:

1. In the integration model, add a process component that you will link to the router.
2. Add and configure ports on the process component.
 - To use the RouteByPortName router, the port names must be the same as the component types for the events you want to handle.
 - To use the RouteByPortType router, the port types must be the same as the interfaces of the events you want to handle.
3. Use either of these methods to create the router process model:
 - Double-click a new process component, and when asked to link a process model, select **New...** to bring up the New Wizard. Select the desired router template from the New Wizard, by expanding the **Models > Routers** node.
 - Right-click the project directory in the Explorer, and select **New > All Templates...** and in the New Wizard, select **Models > Routers**. Choose the desired router.
 - Select **File > New...** and in the New Wizard, select **Models > Routers**. Select the desired router template.
4. Link the process component to your new router if you did not use the double-click method in [step 3](#) to create it.

Note: The routers are designed so that it is not necessary to replicate the component's ports on the router model. To avoid automatic port synchronization, the Auto Synchronize Ports property for this model is set to false.

SIMPLETRANSFORMER TEMPLATE

The SimpleTransformer template provides a quick shortcut for creating a process component that is used to perform a single transformation and push the output event to downstream components. This functions similarly to the RouteByPortType router, discussed above, but allows for an event transformation before the routing. Technically, it creates a process model with a nested transformer model, as shown in [Figure 15-3](#).

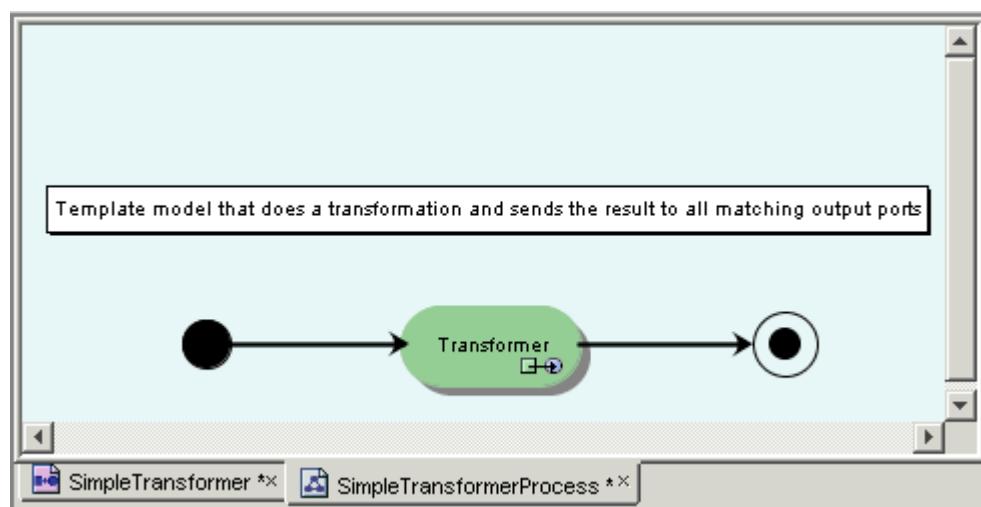


Figure 15-3 Process Model Containing a Transformer State

ADDING A SIMPLETRANSFORMER TEMPLATE

To use the SimpleTransformer template, simply add it to your project, then link a process component in an integration model to the new Simple Transformer template. Remember, you can link one transformer template to multiple process components.

1. In the integration model, add a process component that you will link to the template.
2. Configure ports on the process component. Port types must be the same as the interfaces of the events you want to handle.
3. Use any of these methods to create the SimpleTransformer process model and transformer model:
 - Right-click the project directory in the Explorer, and select **New > All Templates...**. In the New Wizard, select **Models > SimpleTransformer**. Click **Next**.
 - Select **File > New...** and in the New Wizard, select **Models > SimpleTransformer**. Click **Next**.
 - Double-click a new process component, and when asked to link a process model, select **New...** to bring up the New Wizard. Expand the **Models** node and select the SimpleTransformer template from the New Wizard. Click **Next**.

4. Enter a name for your transformer model. Click **Next**. The process model that nests the transformer model will be named <Name>Process. As shown in [Figure 15-3](#), the default name, *SimpleTransformer*, was used when creating the models. The process model name is *SimpleTransformerProcess*.
5. In the Select Types/Events pane, select a transformer to extend or leave the field blank to automatically create a new base transformer. Select input and output events as you normally would when creating a transformer model. Click **Finish**.

For more information about transformer models see the *BusinessWare Transformer Guide*.

6. Link the process component to your new simple transformer process model if you did not use the double-click method described in [step 3](#) to create it.

Note: The SimpleTransformer template is designed so that it is not necessary to replicate the component's ports on the process model. To avoid automatic port synchronization, the Auto Synchronize Ports property for this model is set to false.

SIMPLEXQUERYTRANSFORMER TEMPLATE

The SimpleXQueryTransformer template creates a process model with an Advanced XQuery transformation state that is linked to an Advanced XQuery Transformer. The Advanced XQuery Transformer is a data transformation tool that allows you to graphically map and convert XML data from one format to another. For more information on the Advanced XQuery Transformer, see the *BusinessWare Advanced XQuery Transformer Guide* and the *BusinessWare Advanced XQuery Transformer Help*.

ADDING A SIMPLEXQUERYTRANSFORMER TEMPLATE

To use the SimpleXQueryTransformer template, simply add it to your project, then link a process component in an integration model to the new SimpleXQueryTransformer template. Remember, you can link one transformer template to multiple process components.

1. In the integration model, add a process component that you will link to the template.
2. Configure ports on the process component. Port types must be the same as the interfaces of the events you want to handle.
3. Use any of these methods to create the SimpleXQueryTransformer process model and SimpleXQueryTransformer model:

- Right-click the project directory in the Explorer, and select **New > All Templates....** In the New Wizard, select **Models > SimpleXQueryTransformer**. Click **Next**.
 - Select **File > New...** and in the New Wizard, select **Models > SimpleXQueryTransformer**. Click **Next**.
 - Double-click a new process component, and when asked to link a process model, select **New...** to bring up the New Wizard. Expand the Models node and select the **SimpleXQueryTransformer** template from the New Wizard. Click **Next**.
4. Enter a name for your transformer model. Click **Next**. The transformer model is created with the default name, *SimpleXQueryTransformer* and the process model name is *SimpleXQueryTransformerProcess*.
 5. Begin configuring your SimpleXQueryTransformer using the Map Builder.

CREATING A PROCESS MODEL TEMPLATE

If you create a process model that defines a commonly used process or subsystem, you may want to save that model as a template so that you can use it in other projects. The procedure is simple.

1. Right-click on the model name in the Explorer, and select **Save As Template....**
2. In the Save as Template dialog box, expand the Models node, and select the folder in which you want to place your model (for example, *Routers*).
The template name will be your model name, so give it a meaningful name (for example, *MyOrderRouter*).
3. Open the second project and select **File > New...**
4. In the New Wizard, select **Models** and then select your process model.

You may need to adjust some of the property settings on the model to suit the new project. Also, if your model contains nested states, you either have to bring in the submodels as well or change the nesting specifications.

PROCESS MODEL TEMPLATES

Creating a Process Model Template

This chapter describes workflow and the different concepts that are part of a workflow solution. It covers information on building a workflow solution in BusinessWare.

Topics include:

- [Workflow Concepts](#)
- [Constructing a Workflow Model](#)
- [Setting Up the Task Manager Project](#)
- [Workflow Performance Tuning](#)

WORKFLOW CONCEPTS

Workflow is the automatic management of business objects across human-based systems. That is, the workflow system *interacts* with the human-based system. This interaction has a singular purpose—to have the human-based system perform some function for the business object.

BusinessWare workflow is based on the Workflow Management Coalition (WfMC) standards that have become the most widely supported standards in the workflow industry. WfMC is a nonprofit organization with the aim of advancing workflow technology through the development of common terminology and standards. For more information about the WfMC standards, see their Web site at <http://www.wfmc.org>.

The WfMC defines workflow as the “automation of a business process, in whole or part, during which documents, information, or tasks are passed from one *participant* to another for action, according to a set of procedural rules.”

ACTIVITIES

An *Activity* is a unit of work performed by a human-based system. BusinessWare provides:

- Objects to support activity definition

- Specialized runtime services to assign and manage activities

An activity typically generates one or more work items that constitute the *tasks* or *subtasks* to be undertaken by the *participant* within this activity. These work items are then presented to the user via a *Workflow UI*. The participant can view his or her *Task List*, select and open work items in it, and complete them as required.

Example: Suppose an organization automates the process of ordering a custom-built computer. One step requires an employee to manually verify that the packaged computer contains the requested components. The activity is the manual inspection of the packaged computer. When the computer is packed, the automation system assigns this activity to a team of inspectors. Because only one inspector needs to inspect the package, the automation system continues with the process after exactly one inspector successfully responds to the assigned activity.

BusinessWare supports activities with a process model state called an *activity state*. An activity state incorporates information such as:

- What activity must be completed
- When the activity must be completed
- Who must complete the activity
- The priority at which the activity must be completed

The activity state sends this information to a Task Manager project, which creates task objects and assigns them to workflow participants.

SUBTASK WORKFLOW

To perform a task, a performer might need to send one or more messages to other project members to request more information, get approval, or request that some action be taken before the task can be completed. These messages are tracked using subtasks.

During design time, you can designate who can be assigned subtasks by setting the Subtask Performer property in the activity state properties Expert tab. You can set the following subtask performer assignments:

- None
- Any Role in Project
- Any User
- Activity Performer

The performer is notified when the subtask is completed.

WORKFLOW PARTICIPANTS

The term *workflow participant* refers to a human resource who performs the work represented by a workflow activity. Complex business organizations typically organize their employees into groups and assign individual employees to work within and across these groups. A workflow solution includes the ability to manage individuals using roles and relationships, for example, by assigning access and tasks to people who are qualified to do the work.

BusinessWare workflow supports the following workflow participants:

- **Performer**—Any individual who is assigned to complete a task or a subtask during the execution of a business process.
- **Supervisor**—A supervisor is someone designated to supervise the completion of an activity. The activity specifies a role as its supervisor, and any member of that role acts as the activity's supervisor. A supervisor is usually required to handle exceptions. There are two types of workflow supervisors: activity supervisor and model supervisor. While activity supervisors oversee a particular activity within the process model, model supervisors oversee the entire process.
- **Manager**—A manager is anyone who has a manager relationship to one or more subordinates (these relationships are defined in the directory server, not in the BME). For example, assume user A and user B in the directory server have the following specified relationship: user A has a manager relationship with user B; that is, user A is user B's manager. Therefore, user A can log in and have access to user B's tasks.

The same is true if B is an organizational group (instead of an individual user). User A has access to any tasks that any member of B can see.

Managers have access to the activity (for reading activity properties) and also to the BPO (for reading BPO properties), but not for terminating the BPO.

Note: BW Manager is the only relationship supported in this release. To manage roles and relationships, see the Application Administration UI and the *Application Administration Help*.

Performers and supervisors refer to *roles* defined in the application project. When you model a workflow solution, you identify the *roles* of the people who perform the activities.

The BusinessWare Modeling Environment (BME) uses the following terms to help describe and organize your workflow participants:

- **Role**—identifies users by their responsibilities in the workflow.

- **Relationship**—forms a named and directional binding that ties one physical organizational principal (an organizational user or group in the directory server) to another. For example, a Reviewers Manager has a BW Manager relationship to a group called Reviewers, as shown in [Figure 16-1](#).

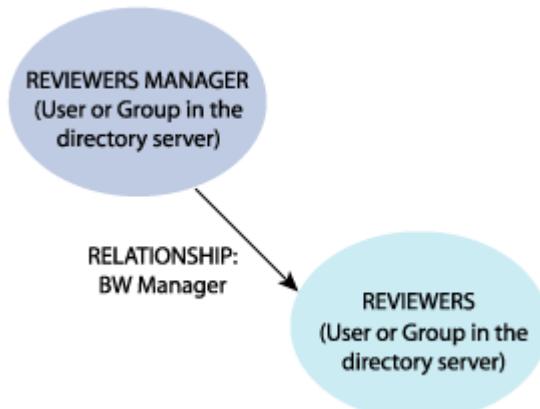


Figure 16-1 Relationships Between Users and/or Groups

To manage roles defined in your application project using the Application Administration UI, your project must depend on vtAFCommon. Access the Application Administration UI using the following URL:

`http://localhost:8080/af/main.jsp?Project=/Projects/<your project name>/<version>`

The Application Administration UI is available on the same Web server where the BusinessWare Administration tool runs. Consequently, <localhost> in the URL should refer to the host where the Web server is running. Port 8080 is the default port number used by the Web server.

For more information on using the Application Administration UI, see *Application Administration Help*.

For more information on managing performers, see “[Setting Up the Task Manager Project](#)” on page 16-26 and the *BusinessWare Administration Guide*.

BUSINESSWARE WORKFLOW UI

BusinessWare provides a Web-based tool, the *BusinessWare Workflow UI*, to enable performers and supervisors to manage their workflow tasks. The BusinessWare Workflow UI includes:

- **Performer Pages**—A series of Web pages the performer can use to view and manage his or her tasks or subtasks. A performer can view assigned tasks or subtasks, indicate when an assigned task or subtask is complete, and relay any data associated with completing the task or subtask back to the automation system for further processing. Tasks and subtasks can be sorted by due date, priority, and a number of other attributes.
- **Supervisor Pages**—A series of Web pages in the Workflow UI that supervisors can use to monitor and edit tasks. Using the appropriate supervisor pages, supervisors can:
 - Access the same basic functions as performers
 - Change the state of tasks on behalf of performers. For example, a supervisor can reassign a task if the assigned performer is out of the office.
 - View a list of the tasks related to his or her responsible activities
- **Manager Pages**—A series of Web pages in the Workflow UI that managers can use to monitor and edit tasks for their subordinates. Using the manager pages, managers can:
 - View a list of all of their subordinates' tasks
 - View a list of all of their subordinates' tasks organized by activity
 - Edit one of their subordinates' tasks

To access the Workflow UI, use the following URL:

`http://<hostname>:<port>/af/main.jsp?Project=/Projects/vtWorkflowUI/initial`

where `<hostname>` in the URL refers to the host where the Workflow UI is running and `<port>` is the port used by the Web server. For more information, see the *Application Framework User Guide*.

CONSTRUCTING A WORKFLOW MODEL

This section assumes that you are familiar with the BusinessWare Modeling Environment (BME). In the following sections, basic workflow modeling using process model activity states is described.

- [Process Modeling with Activity States](#)
- [Activity State Properties](#)

- Reference Data Property Settings
- Performer and Supervisor Properties
- Transitions Out of Workflow Activity States

When you install BusinessWare, a sample that demonstrates workflow and activity states is installed in *installdir\samples\modeling\ActivitySample*. You can review *ActivitySample.htm*, located in the same folder, for instructions on how to use the sample.

PROCESS MODELING WITH ACTIVITY STATES

Workflow solutions are modeled using *activity states* in process models. Activity states are a special modeling concept that allows human involvement in business processes. An activity state has information about what activities should be performed and by whom and how and when the activity should be performed. Activity states may have multiple outgoing transitions. For example, the most common outgoing transition is triggered by `bpe.ControllableBPO.completeActivity`, which indicates the completion of an activity.

Example: Consider an activity in which a document must be inspected and approved within three days. The business operation requires that a team of six inspectors inspect the document, and at least three of six inspectors approve the document. In addition, the document is not managed by any type of automated system—it simply resides at a specific URL. A workflow solution can assign the activity to a team of inspectors and pass the URL along with the assignment as part of its detail data. The workflow solution can manage the activity by aggregating approval votes, moving on only if three of six inspectors approve the document. If the activity is not completed within three days, the workflow solution sends an alert to the activity supervisor.

ACTIVITY STATE PROPERTIES

The properties of an activity state display in two tabs of the Properties window. Figure 16-2 shows the **Properties** and Figure 16-3 shows the **Expert** tab.



Figure 16-2 Activity State Properties

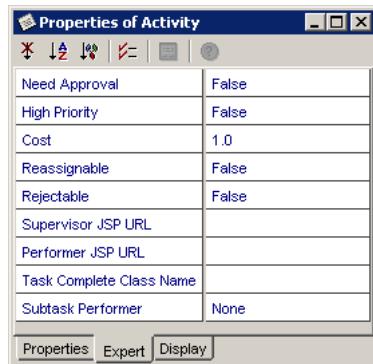


Figure 16-3 Activity State Properties, Expert Tab

WORKFLOW

Constructing a Workflow Model

The activity state properties in [Table 16-1](#) are listed in the order that they appear on the **Properties** and **Expert** tabs.

Table 16-1 Activity State Properties

Name	Description	Default Values
Name	Name of the state.	The default name for an activity state is <i>Activity</i> . A numeral is added to the name if the same model has multiple activities (<i>Activity 2</i> , <i>Activity 3</i> , and so on.).
ID	Generated ID of the state. This is a read-only property.	1, 2, 3, and so on.
Timer	Used to set a deadline on the activity. The timeout value indicates when the activity must be completed. This value can be relative (amount of time after entering the state) or absolute (a specified date and time). See “ Checking Timeout ” on page 16-21 for more information.	None
Entry Action	Action code executed <i>after</i> the action code of the incoming transition and <i>before</i> entering the state. Entry action code must be valid Java statements and can refer to BPO attributes or local data.	None
Exit Action	Action code executed <i>after</i> an event or operation triggers a transition out of the state and <i>before</i> the action code of the transition is executed. Exit action code must be valid Java statements and can refer to BPO attributes or local data.	None
Reference Data Properties		
Business Process Object (BPO) Reference Data	Specifies which BPO attribute data will be displayable, filterable, and editable in the performer’s Web pages. For more information, see “ BPO Reference Data ” on page 16-12.	None
Activity Reference Data	Specifies which local variables on this state will be displayable, filterable, and editable in the performer’s Web pages. For more information, see “ Activity Reference Data ” on page 16-14.	None
Performer and Supervisor Properties		
Performer	Name of the role to perform the task. See “ Performer ” on page 16-22 for more information.	None
Performer Assignment Policy	Specifies one of: Any, All, Round Robin, Percentage, Exclude Last, or Least Workload. For more information, see “ Performer Assignment Policy ” on page 16-23.	None

Table 16-1 Activity State Properties (Continued)

Name	Description	Default Values
Performer Assignment Percentage	Specifies the percentage of users in a role who must perform an activity when Percentage is chosen as the Performer Assignment Policy.	0.0
Notify Performer	<p>Specifies that email be sent to the performer when the task has arrived in the performer's inbox.</p> <p>If set to true, notification is sent in the following cases:</p> <ul style="list-style-type: none"> when a task arrives in the performer's inbox. when a task is reassigned. when a task that was suspended with a timeout is resumed as a result of the timeout. when a task is released. 	False
Supervisor	<p>The supervisor role for the activity.</p> <p>For more information, see “Supervisor” on page 16-24.</p>	None
Additional Properties		
Description	<p>Describes the state. This is used in generated source code comments. The description also appears in the performer's Workflow UI.</p> <p>You can incorporate BPO attributes in the description using the <code>setDescription()</code> method. For more information, see the following reference in the <i>BusinessWare Programming Reference</i>: <code>com.vitria.bpe.activity.ActivityStatelib</code>.</p>	Activity State
Label	<p>User-friendly name of the state that is displayed in the BusinessWare Workflow UI.</p> <p>Note: Activity state display labels MUST be unique across all projects served by a single Task Manager.</p>	None
Expert Properties		
Need Approval	<p>This is a boolean value that indicates whether approval or voting is necessary. If the flag is checked the performer must provide a float number indicating the approval rate.</p> <p>For more information, see “Need Approval Property” on page 16-21.</p>	False
High Priority	Determines whether the priority is High (if true) or Low (if false). To set an activity priority to Normal, use <code>ActivityStateLib</code> . The priority does not change the way the task manager handles the activity or its tasks.	False
Cost	An arbitrary unit that represents the amount of resources that are consumed to complete a given task. It is a measurement of workload and is particularly used by the Least Workload assignment.	1.0

Table 16-1 Activity State Properties (Continued)

Name	Description	Default Values
Reassignable	Specifies whether the activity performer can reassign a task of the activity. Supervisors and managers can reassign a task regardless of how this property is set. For more information, see “ Reassignable ” on page 16-24.	False
Rejectable	Specifies whether the performer can reject the task. Supervisors and managers can always reject a task regardless of how this attribute is set.	False
Supervisor JSP URL	Location of the customized supervisor JSP page for the Web interface. This property is deprecated in this release.	None
Performer JSP URL	Location of the customized performer JSP page for the Web interface. This property is deprecated in this release.	None
Task Complete Class Name	User-defined class implementation that provides methods that are called when a task is completed. For more information, see “ Task Complete Class Name ” on page 16-25.	None
Subtask Performer	Specifies the performers allowed to create subtasks.	None

Dynamically Modifying Properties at Runtime

Some properties of an activity state can be dynamically modified at runtime. For example, you may wish to change performer assignments from one role to another, depending on certain business conditions. For more information, see the following package in the *BusinessWare Programming Reference*:
`com.vitria.bpe.activity.ActivityStatelib`.

The activity state library methods can only be used either in an activity state entry action or in a transition action where the target state is an activity state.

Setting Labels for Activity States

As you specify activity state properties, you can set a **Label** that will be visible to your workflow performers.

For example, assume you are working through the Activity Sample included with BusinessWare. If you open the `ContractModel` in the BME and select the `PrepareInvoice` state, you see the Label property set to `Prepare Invoice`, as shown in [Figure 16-4](#).

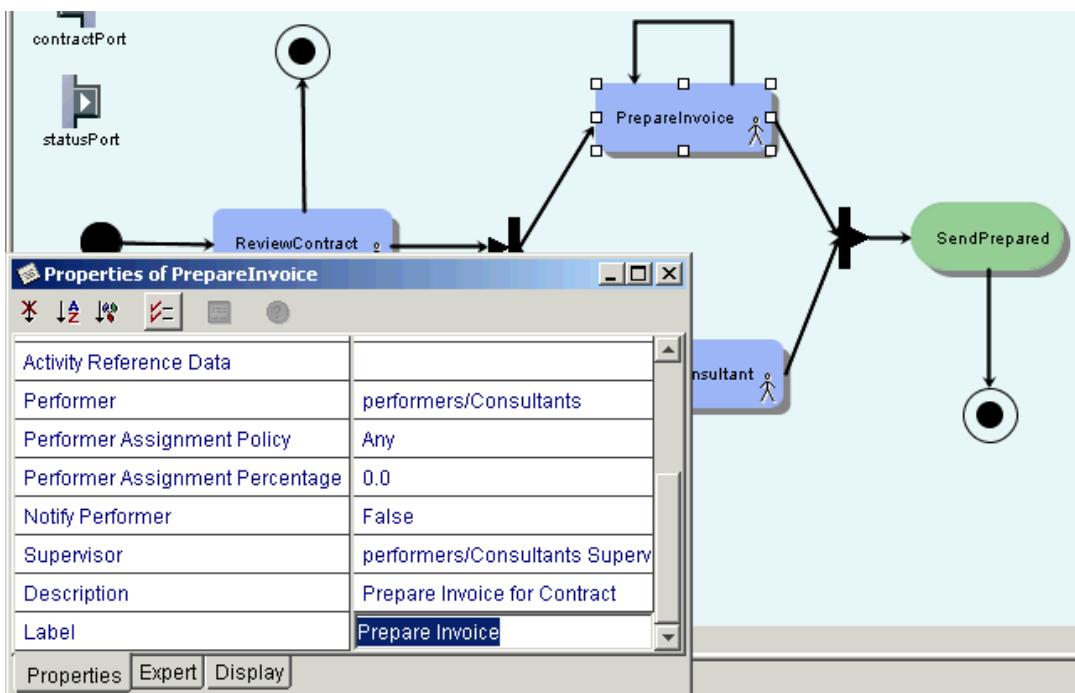


Figure 16-4 Activity State Label in the Contract Model

The Label property for the AssignConsultant state has also been set to Assign Consultant. The label changes make the names of these activities more user-friendly for your workflow performers.

Activity state labels are displayed and used by workflow participants for filtering in the BusinessWare Workflow UI. For best results, you should select a globally unique label across all your process models and application projects. Although changing the labels over time does not cause harm (such as data corruption), you should avoid doing so, as it may be potentially confusing.

REFERENCE DATA PROPERTY SETTINGS

Reference data refers to a set of runtime attributes that are critical for performers to execute a specific task. They can be viewed, filtered, and modified by task performers. The model designer configures what reference data can be viewed, filtered, and modified by performers. BusinessWare supports two types of reference data:

- **BPO Reference Data**

- [Activity Reference Data](#)

BPO Reference Data

When you create the activity state, you can specify which BPO attributes are available to be displayed in the performer interface. For each displayable attribute, you may choose if the attribute's value can be modified by the performer.

The BPO reference data includes the following attributes:

- Name
- Type
- Visible (whether the data is conveyed as the details of the task in the Task Details page of the BusinessWare Workflow UI)
- Editable (whether the data can be edited by the performer in the Task Details page of the BusinessWare Workflow UI)
- Filterable (whether the data is enabled for viewing, filtering, and sorting in the Activity-Based Task List)
- Label (displayed to the user in the BusinessWare Workflow UI)

Note: Reference data labels are displayed to and used by workflow participants for filtering in the BusinessWare Workflow UI. Although changing the labels over time does not cause harm (such as data corruption), you should avoid doing so, as it may be potentially confusing.

Example: In the contract approval process, the BPO (that is, the contract) includes attributes such as Customer ID and Contract Request. You may want to display the customer ID in the Workflow UI and allow a reviewer to filter and sort by the customer ID. On the other hand, you may want the reviewer to correct the contract request in the Task Details page, if necessary.

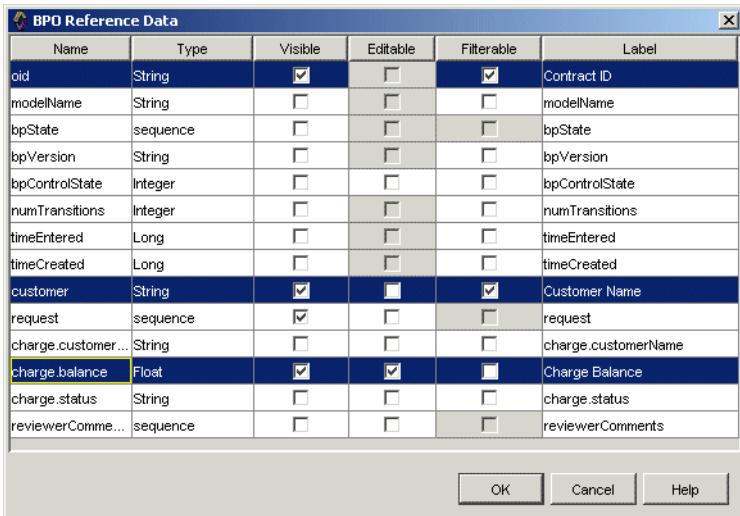
When making BPO reference data visible and editable, you allow performers to set values of those attributes. These attribute values are automatically merged into the corresponding BPO when

`com.vitria.bpe.activity.RefXMLDocLib.mergeBPO()` is called near the end of the transition action code that handles the `bpe.ControllableBPO.completeActivity` event.

When making BPO reference data filterable, you allow performers to search for tasks with reference data matching a specific criteria. For example, in the contract approval process, you may want the consultants to be able to filter the contracts against Contract ID and Customer Name and to provide input for Charge Balance.

To specify BPO reference data as filterable:

1. In the BME, open the ContractModel in the Activity Sample (this sample is used here as an example).
2. Select the PrepareInvoice state. In the Properties Window, select the **BPO Reference Data** property. The **BPO Reference Data** table displays.
3. For this example, select the check boxes in the table according to [Figure 16-5](#).



The screenshot shows a dialog box titled "BPO Reference Data". It contains a table with columns: Name, Type, Visible, Editable, Filterable, and Label. The rows represent different BPO reference data fields. The "Filterable" column has checkboxes. The "Label" column shows the visible column name. The "charge.balance" row is highlighted with a yellow background.

Name	Type	Visible	Editable	Filterable	Label
oid	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Contract ID
modelName	String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	modelName
bpState	sequence	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	bpState
bpVersion	String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	bpVersion
bpControlState	Integer	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	bpControlState
numTransitions	Integer	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	numTransitions
timeEntered	Long	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	timeEntered
timeCreated	Long	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	timeCreated
customer	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Customer Name
request	sequence	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	request
charge.customer...	String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	charge.customerName
charge.balance	Float	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Charge Balance
charge.status	String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	charge.status
reviewerComme...	sequence	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	reviewerComments

At the bottom right of the dialog box are three buttons: OK, Cancel, and Help.

Figure 16-5 BPO Reference Data Filter Settings

- **oid**—the **oid** reference data has been set to be **Filterable** and the Label changed to Contract ID. This label is visible as a column name in the task list.
- **customer**—the customer reference data has been set to be **Filterable** and the Label changed to Customer Name, which is visible as a column name in the task list.
- **charge.balance**—the **charge.balance** reference data is not specified to be **Filterable**, but the Label has been changed to Charge Balance. When you go to the task details page of any task associated with the **PrepareInvoice** state, you will see this label.

You can set similar options on the activity reference data.

Note: It's the best practice and strong recommendation that you minimize the number of filterable reference data. Only that reference data that must be seen, filtered, or sorted by the workflow users should be set as "filterable". A large number of filterable reference data often results in lengthy query execution and poor UI responsiveness in displaying the task list.

Activity Reference Data

The activity reference data is specific to the activity and is added to the detail data that is visible, filterable, or modifiable by the performer when the activity is created at runtime. This data communicates activity-specific information, but modified values will not be merged into the BPO. Examples of this include values that can be used for computation on activity completion, such as approval ratings. Only primitive types are supported.

You should always initialize all activity reference data, especially numeric attributes.

Example: You might create an activity within a stock purchase process model to handle special orders. You might then want to create activity reference data specific to the activity, such as an order limit. You can specify that the attribute is editable by the performer and can be used in transition conditions.

For more information about transition conditions, access *BME Help*. For activity reference data samples refer to the Activity Sample (in `installdir\samples\modeling\ActivitySample`).

Supported activity reference data types include:

- String
- Boolean
- Integer
- Long
- Float
- Double
- Short
- Byte
- Character
- Document ID

CREATING ROLE RESOURCES

This section describes how to create role resources and refer to them at design time. After creating the roles, you can manage them using the Application Administration UI. Roles are also used for access control for modeling objects such as projects, Integration Servers, ports and so on. For more information, see the *BusinessWare Security Guide*.

For more details on creating role resources, see the *Application Administration Help*.

Note: To use the Application Administration UI to administer roles in your workflow applications, the workflow project in the BusinessWare Modeling Environment (BME) must depend on the vtAFCommon project.

To create role resources in the BME:

1. Select **File > New....**
2. In the **New Wizard**, expand **Resources**.
3. Select **Role**.
4. Click **Next**.
5. Enter a name.
6. Click **Finish**.

WORKFLOW

Creating Role Resources

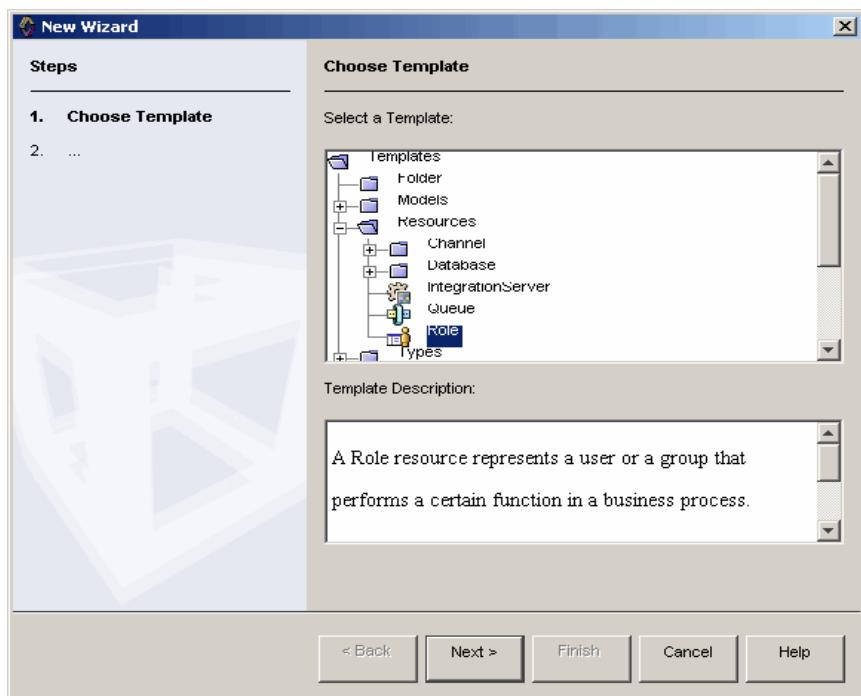


Figure 16-6 Creating a Role Resource

You can also set the **Role Description** property. This property, while not shown to the workflow performers, is seen by the project developers and the administrators using the Application Administration UI.

After you have created and named the role, you must select it when you configure the **Performer** and **Supervisor** properties of the activity state in the process model.

Mapping Roles

During project deployment, roles must be mapped to corresponding physical entities. The Vitria-supplied authorization pluggable modules map roles into groups in the directory server. Those groups are referred to as “role groups” to distinguish these groups from organizational groups.

Default role mapping maps roles based on settings configured in the security plug-in specified for authorization. The security plug-in is configured during BusinessWare installation. See the *BusinessWare Installation Guide* for more information.

You can explicitly map a role to a particular group in the directory server. In this case, you should not use an organizational group, but instead, use a group created specifically to contain the members of the role. Set the **Principal** property by browsing to select the group in the directory server. See [Figure 16-7](#).

IMPORTANT: Non-default role mapping is not recommended.

IMPORTANT: The Properties window only displays the prefix for the Distinguished Name (DN) value of the Principal property. The value of this property is determined by the VTPARAMS file of the person deploying the project. If a BusinessWare developer deploys a project to a .jar file, and an administrator with a different VTPARAMS file deploys the .jar file to the directory server, it is the value defined in the administrator's VTPARAMS file that determines the full DN for the Principal property.

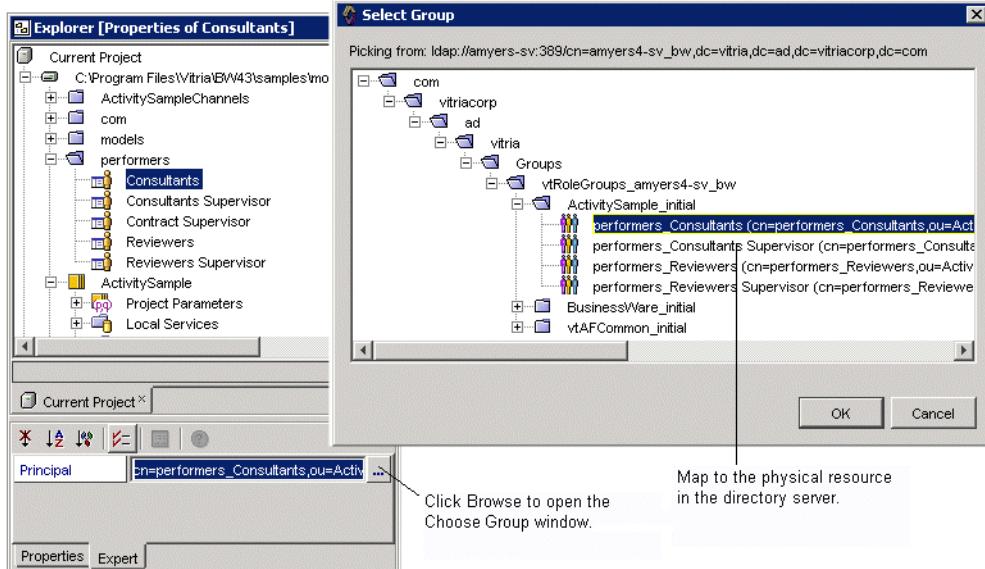


Figure 16-7 Mapping Roles

TRANSITIONS OUT OF WORKFLOW ACTIVITY STATES

An activity state's outgoing transitions are typically triggered upon activity completion by the `bpe.ControllableBPO.completeActivity()` method, which contains the following parameters:

- **id**—the ID of the BPO.
- **jndiComponentName**—the name of the process model component containing the activity state.
- **activity**—the name of the activity state.
- **properties**—contains the original reference data.
- **tasks2**—contains updated reference data inputs from the performers that can be examined during the transition out of the activity state.
- **last**—points to the position in the array where the copy of reference data inputs made by the last performer are located.

The `tasks2` event parameter contains an array of reference data inputs, one from each of the performers. You can iterate through the array to look for inputs from any performer and use `RefXMLLoaderLib.getValue()` to retrieve the value of a particular reference data attribute. The sample code in [Table 16-2](#) shows how to retrieve comments made by the reviewers that explain why the contract was disapproved.

The code samples in [Table 16-2](#) are examples of transitions out of workflow activity states. The code samples are taken from the Contract Model of the Activity Sample (in `installdir\samples\modeling\ActivitySample`).

Table 16-2 Code Examples from the Contract Model

Description	Code Sample
Transition from 'ReviewContract' to 'Fork'	<pre>// set the reviewer comments in the contract String[] reviewerComments = new String[tasks2.length]; for (int i=0; i<tasks2.length; i++) { reviewerComments[i] = RefXMLLoader.getValue(ctx, tasks2, i, "comments"); } contract.reviewerComments(reviewerComments); if (DefaultLogger.traceLevelProcessModel_ >= DiagLogger.NORMAL) { CommonMessages.logGenericTrace("**** Exiting ReviewContract Activity with status Approved."); } // publish the status event getStatusPort().status(getBPO().oid(), getBPO().customer(), getBPO().getTypeByRequest(), "Approved", getBPO().getConsultant(), getBPO().reviewerComments()); // NOTE - the activity state is merging task completion data with BPO here. // DO NOT EDIT the following code! com.vitria.bpe.activity.RefXMLLoader.mergeBPO(tasks2, last, (com.vitria.bpe.process.StorableBPO)contract); // DO NOT EDIT the above code!</pre>

Checking Reference Data Input

If an activity is performed by more than one performer, multiple copies of reference data are available upon completion of that activity, one from each performer. By default, the copy provided by the performer who triggers completion of the activity, referred to as “last” performer, is selected and merged into the BPO. This copy is indicated by the `last` event parameter specified in the incoming event.

You can change this behavior by modifying the value of the `last` parameter to point to the desirable copy of the reference data before `com.vitria.bpe.activity.RefXMLLoaderLib.mergeBPO()`. The `last` parameter is always set to zero if the activity was performed by only one performer.

If you want to use character data (IDL types `char`, `wchar`, or sequences thereof) as part of your BPO reference data for an activity state, the data should be initialized to some non-zero value. This applies to attributes of all structs, unions and `DataObjects`, except attributes of your BPO.

As previously described, BPO reference data provided by the performers are not merged into the BPO until `RefXMLLoaderLib.mergeBPO()` is executed in the transition action. To trigger a transition handling the `bpe.ControllableBPO.completeActivity` event out of the activity state based the performer's inputs, you can do either of the following:

- Compose the transition condition that gets and checks the performer's input via `RefXMLLoaderLib.getValue()`.
- Make the outgoing transition to an action state, during which the BPO reference data has been merged into the BPO. Use the action state as the decision node with other outgoing transitions whose condition simply checks the BPO attributes.

Checking for Approval

When you set the **Need Approval** property in the activity state property dialog box to `True` (for example, in the Activity Sample, Review Contract state), you automatically create an activity reference data item of type float called `_approveRate` in the transitions leaving activity states.

Refer to “[Need Approval Property](#)” on page 16-21 for further information about this property.

To check the approval rate, compose a transition condition to determine if an activity is approved or not by checking the value of `_approveRate`. The built-in variable, `_approveRate`, is the average of all the approval values entered by the activity performers. The model designer communicates the scale of the approval rating the activity performers should enter. For example, in the Activity Sample, a scale between 0 (total disapproval) and 1 (total approval) is adopted. Therefore, a contract is considered approved if `_approveRate` is greater than or equal to 0.5.

Need Approval Property

The **Need Approval** property of an activity provides prebuilt logic for handling the common task of approving an activity.

If an activity's Performer Assignment Policy is set to All, each member of the group enters a value for `_approveRate`. When all the tasks are completed, the value of `_approveRate` for the activity is the average of all the values entered by the performers. This value can be used by the transition out of the activity.

Example: If you wanted to take a majority vote on a proposal, you could have each performer in the group enter 0 (disapprove) or 1 (approve) for `_approveRate`. You could then set up transition conditions for the activity so that an `_approveRate` (the average of all the performer entries) greater than or equal to 0.5 would cause a transition to an approved state and a value for `_approveRate` less than 0.5 would lead to an unapproved state. If you wanted unanimous approval, you could make a transition condition of `_approveRate` equal to 1.

In an activity's Performer Assignment Policy is set to Percentage, `_approveRate` for the activity is the average of all the values entered by the number of performers specified by the percentage value. For example, if a group has 10 members and the percentage value is 50%, then the values entered by 5 members would be averaged.

In an activity is assigned to a single performer, for example if the Performer is Last Performer, or if the Performer Assignment Policy is set to Any, Round Robin, or Least Workload, `_approveRate` for the activity is equal to the value entered by the performer (since there is only one value in the average). See the Activity Sample for more information about the **Need Approval** property.

Checking Timeout

An activity defined in a business process should be completed within a time limit. You are able to define a time limit for an activity state using the **Timer** property.

You can also define alternative actions should the defined time limit be exceeded. The alternative actions can be a special transition out of this activity state to some other state, or a trigger of an alert to the supervisor if the timeout is not handled. You can specify relative time or absolute time.

You have a choice whether to handle the time-out event in the process model. If no outgoing transitions handle the time-out event, then the BPO remains at the same activity state and a notification is automatically sent to its supervisors. Supervisors can start, reassign, or complete tasks on behalf of the assigned performer. Process supervisors can also terminate the entire process, if necessary.

Alternatively, you can handle the time-out event by adding an outgoing transition from the activity state. Upon time-out, the BPO terminates the activity and the task and moves to the destination state.

PERFORMER AND SUPERVISOR PROPERTIES

This section contains additional information on some performer and supervisor activity state properties that are listed in [Table 16-1 on page 16-8](#).

PERFORMER

The **Performer** property allows you to specify the role of the individual or group performing the tasks. This can also be set to Last Performer of the previous activity.

- **Role**—a role determines which user or users are responsible for performing the activity. The number of tasks created depends on the Performer Assignment Policy and the number of members in the role.
- **Last Performer**—if a user completed a task that then completed the associated activity, the user is referred to as the last performer of the activity. If the subsequent activity is assigned to “Last Performer,” that user receives a task from that activity. This is not valid for the first activity state in the model because there will be no “last performer.”
- **Subordinate**—if the role is “BW Workflow Manager” (from the BusinessWare/initial project), use this field to set the subordinate that determines the manager to perform the role. You must use the complete DN of the subordinate. You can use the action builder or action code to construct the name from data stored in your process model. For more information on the action builder, see [“Using the Action Builder” on page 14-6](#).
- **Manager**—if the role is “BW Workflow Subordinate” (from the BusinessWare/initial project), use this field to set the manager that determines the subordinate to perform the role. You must use the complete DN of the manager. You can use the action builder or action code to construct the name from data stored in your process model. For more information on the action builder, see [“Using the Action Builder” on page 14-6](#).

[Figure 16-8](#) shows the Performer dialog accessed by clicking the browse button next to Performer in the Activity State properties window.

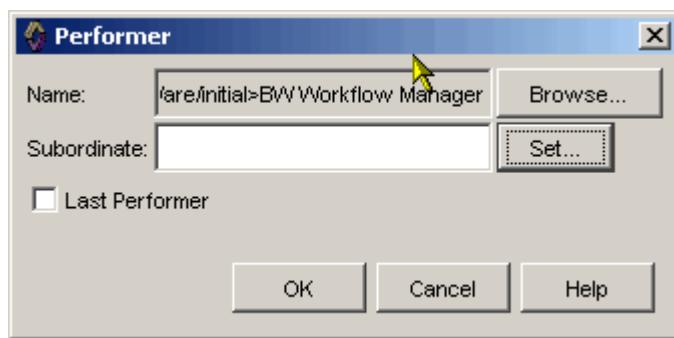


Figure 16-8 Activity State Performer Property

PERFORMER ASSIGNMENT POLICY

When a process model transitions into an activity state, the resulting activity generates one or more tasks to be completed by task performers.

You specify how tasks are assigned within the role using the **Performer Assignment Policy** property. Choices include:

- **Any**—tasks assigned to Any are performed by one of the users in the role. Once a user in the role has started an Any task, the task no longer appears in the Workflow UI of the other users who have the role.
- **All**—tasks assigned to All must be performed by all the users in this role.
- **Round Robin**—tasks assigned to Round-Robin are performed in a round-robin style by all users in the role.
- **Percentage**—tasks assigned to Percentage are performed by a percentage of users in the role. The percentage is specified in the separate property “Performer Assignment Percentage”.
- **Exclude Last**—tasks assigned to Exclude Last are performed by all users assigned the role except the performer of the previous activity. This is not valid for the first activity state in the model, because there will be no “last performer.”
- **Least Workload**—tasks assigned to Least Workload are performed by the user in the role with the least workload. Least workload is determined by comparing the sum of the cost factors for started and unstarted tasks for each individual performer. The cost for an activity is specified in the activity state.

SUPERVISOR

The **Supervisor** property specifies the role resource who has overall responsibility for the activity and who has the privileges to intervene in the associated tasks when necessary. For example, the activity supervisor can reassign a task to a different performer if the original performer is absent.

Note: Managers of performers also have access to their subordinates' tasks.

If the role is “BW Workflow Manager” (from the “BusinessWare/initial” project), use the Subordinate field to set the subordinate that determines the manager to perform the role. You must use the complete DN of the subordinate. You can use the action builder or action code to construct the name from data stored in your process model. For more information on the action builder, see [“Using the Action Builder” on page 14-6](#).

If the role is “BW Workflow Subordinate” (from the BusinessWare/initial project), use the Manager field to set the manager that determines the subordinate to perform the role. You must use the complete DN of the manager. You can use the action builder or action code to construct the name from data stored in your process model. For more information on the action builder, see [“Using the Action Builder” on page 14-6](#).

In contrast to activity supervisors, model supervisors are those who have overall responsibility for the entire process. Model supervisors can be specified in the **Model Supervisor** property of the process model. For more information, see [Chapter 10, “Process Models: Basic Concepts.”](#)

When nesting a process model containing activity states, the model supervisor of the top-level model will supervise the entire model including the nested ones; the model supervisor of the nested process model is ignored. Activity supervisors of the activity states in the nested process model will still supervise the activities as specified.

EXPERT PROPERTY SETTINGS

This section contains additional information on some Expert activity state properties that are listed in [Table 16-1 on page 16-8](#).

Re assignable

Any activity state may be marked as re assignable to enable reassignment by performers. Reassignment may be to any other role member, or to any user serviced by the same task manager as the original performer.

Supervisors and managers can always reassign a task regardless of how this attribute is set.

Task Complete Class Name

This class implements interface `com.vitria.workflow.TaskComplete` and must be included in the Admin Web Server classpath (see “[Runtime Architecture](#)” on page 24-1). It is recommended that you copy this class to `installdir\java`.

Using this interface, you can implement additional logic that, for example, validates the reference data inputs. For more information, please refer to package `com.vitria.workflow.TaskComplete` in the *BusinessWare Programming Reference*.

Dynamically Setting Performers and Supervisors

In setting your properties for performers and supervisors, please note:

- Use the `com.vitria.bpe.activity.ActivityStatelib.setPerformer` method to set the performer role and assignment policy.
- Use the `com.vitria.bpe.activity.ActivityStatelib.setSupervisor` method to set the supervisor role.

Note: Supervisors that are dynamically set will not have access to the BPO after the activity they supervised (via the override) has exited. This means that the override supervisor will not be allowed to see the BPO state, which design-time supervisors can see.

For more information about these properties, see “[Activity State Properties](#)” on page 16-7.

SETTING UP THE TASK MANAGER PROJECT

When a process model transitions into an activity state, the activity state automatically calls a *Task Manager Project*. Task Manager projects are preconfigured to create specialized types of business process objects (BPOs), called *activity* and *task*, in response to this type of event. For each activity state encountered, an activity is created in the Task Manager project. For each performer who must complete this activity, a task is created and assigned. A task is used to record progress made by the individual performer. A task is complete only when its assigned performer interacts with the Workflow UI to indicate the completion of the task. An activity is *complete* based on the rule that satisfies activity completion. This may mean that not all the tasks need to be complete.

Example: Consider an activity in which a document must be inspected. If the business operation requires that a team of five inspectors perform the activity, the work of each inspector is considered to be a unique task. The Task Manager creates one activity—an *inspect document activity*—and five tasks, one for each inspector. The activity is not complete until each of the five inspectors perform their individually assigned task. If the business operation requires only half of the inspector team to perform the activity, then only three out of five tasks are required to complete the entire activity.

When an activity completes, the Task Manager project calls back to the original project to allow continued processing of the application BPO, from the originating activity state. The Task Manager also may include data associated with the completion of the task back to the application BPO for further processing. For more information see “[Reference Data Property Settings](#)” on page 16-11.

Note: Your directory server must be running before starting the BusinessWare Server or any project. The Task Manager relies on user and group information stored in the directory server to manage activities and tasks.

You can easily scale up the Task Manager project by clustering the Integration Server defined inside the Task Manager project. A clustered Task Manager project typically results in more satisfactory system throughput and lower maintenance cost compared to managing multiple task managers.

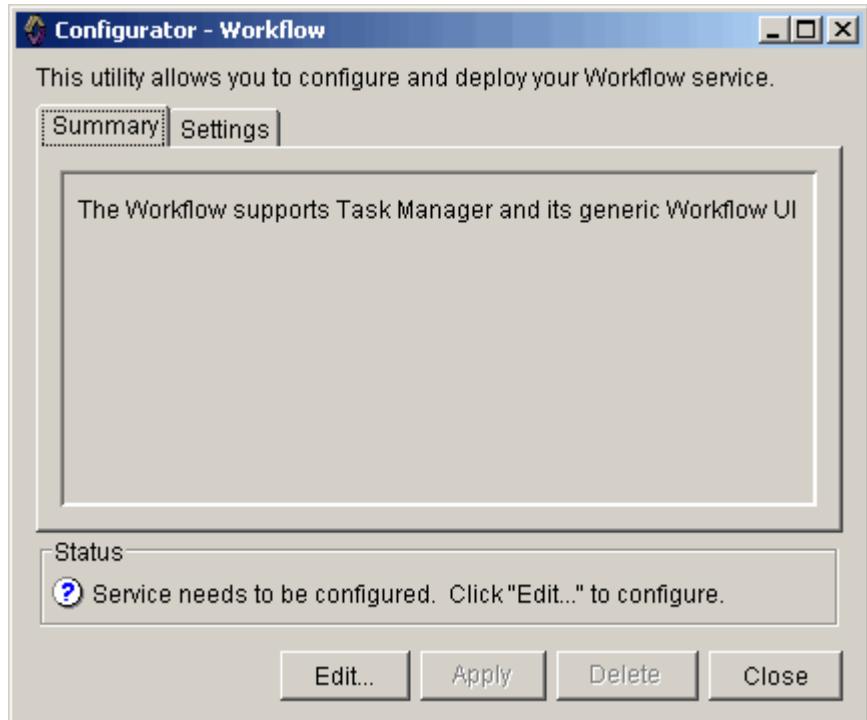
TASK MANAGER CONFIGURATION AND DEPLOYMENT

Use the Configure Solution Wizard to configure and deploy the Task Manager project.

Note: You can use the Configuration Solution Wizard or BME to configure the Task Manager only on a Windows or Solaris installation of BusinessWare.

To Configure the Task Manager:

1. Close any open instances of the BME.
1. Open a dos command window and type afconfig workflow.
2. Press **Enter**. The Configurator opens.

**Figure 16-9 Configure Solution Wizard**

3. Click **Edit....**
4. In the Configure Workflow Step 1 window, click **Next>**.
5. In the Select BW Server window, select a BusinessWare Server and click **Next>**.
6. In the Configure Workflow Options window, select **Task Manager** to configure and deploy the Task Manager project. You should also select **Workflow UI** to deploy the vtWorkflowUI project. Deploying the vtWorkflowUI project enables the Tasks tab in the Application Administration UI. Click **Next>**.

WORKFLOW

Setting Up the Task Manager Project

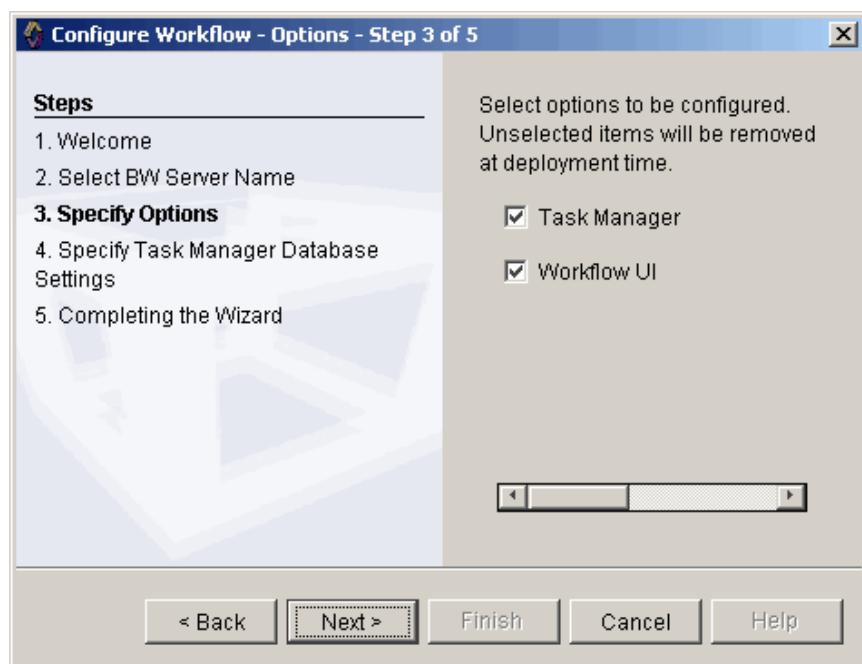


Figure 16-10 Configure Workflow Step 3

7. In the Task Manager Database Settings window, enter the following information:
 - a. Database Type
 - b. Server Name
 - c. Database Name
 - d. Port Number
 - e. User Name
 - f. User Password

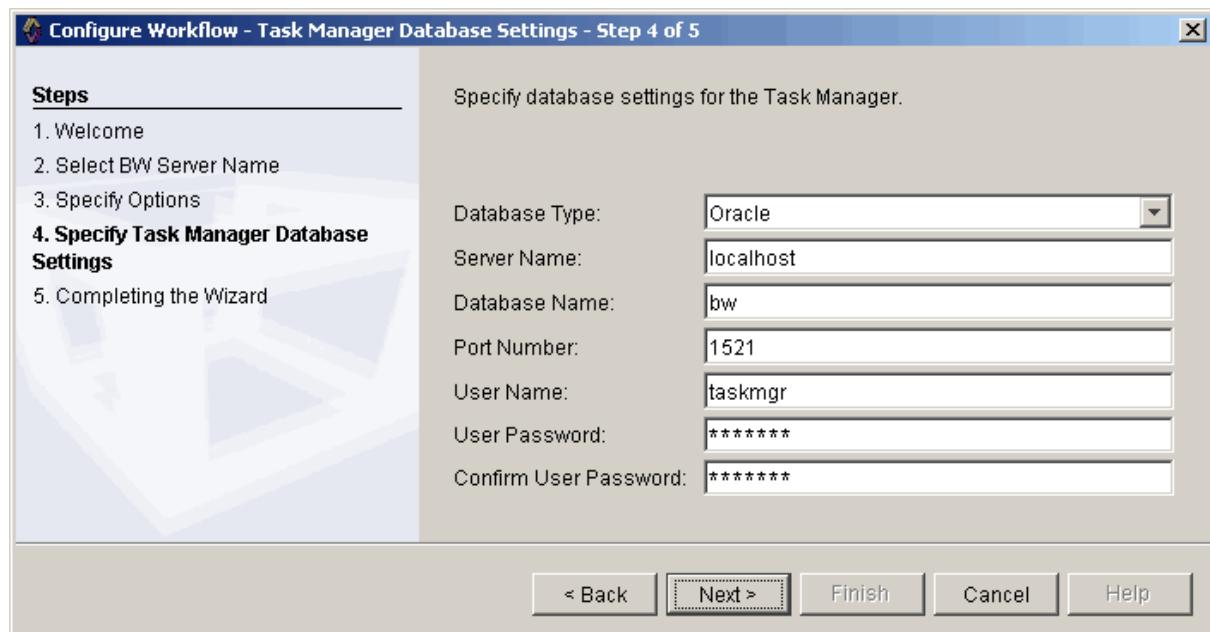


Figure 16-11 Configure Workflow Step 4

8. Click **Next>** to complete the configuration.
9. In the Configurator window, click **Apply** to start the Deploy Solution Wizard.

WORKFLOW

Setting Up the Task Manager Project

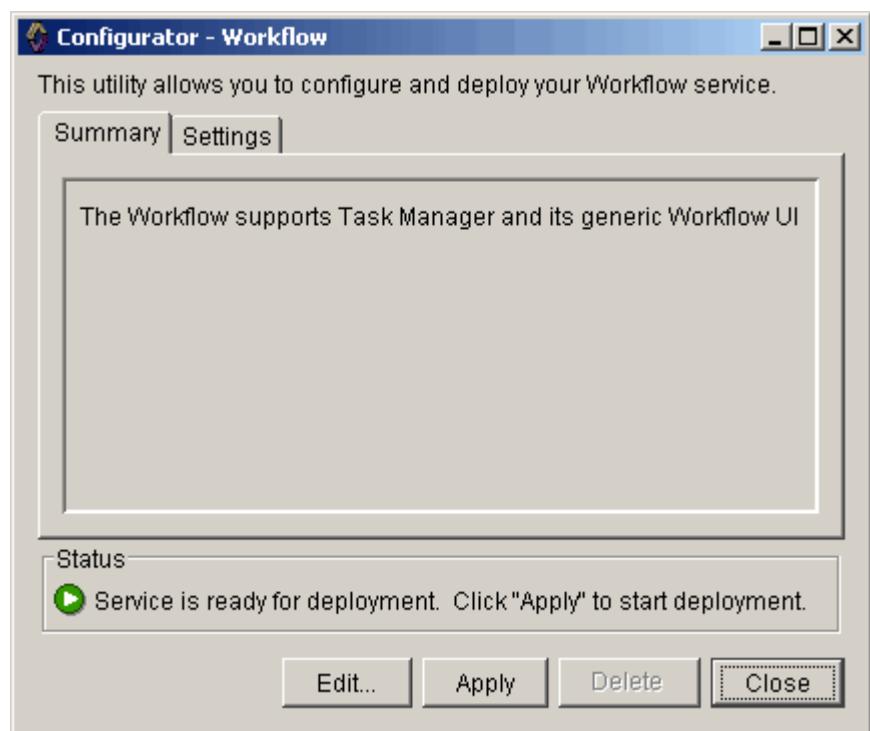


Figure 16-12 Configurator

To Deploy the Task Manager project:

1. In the Deploy Solution Wizard, click **Next>**.

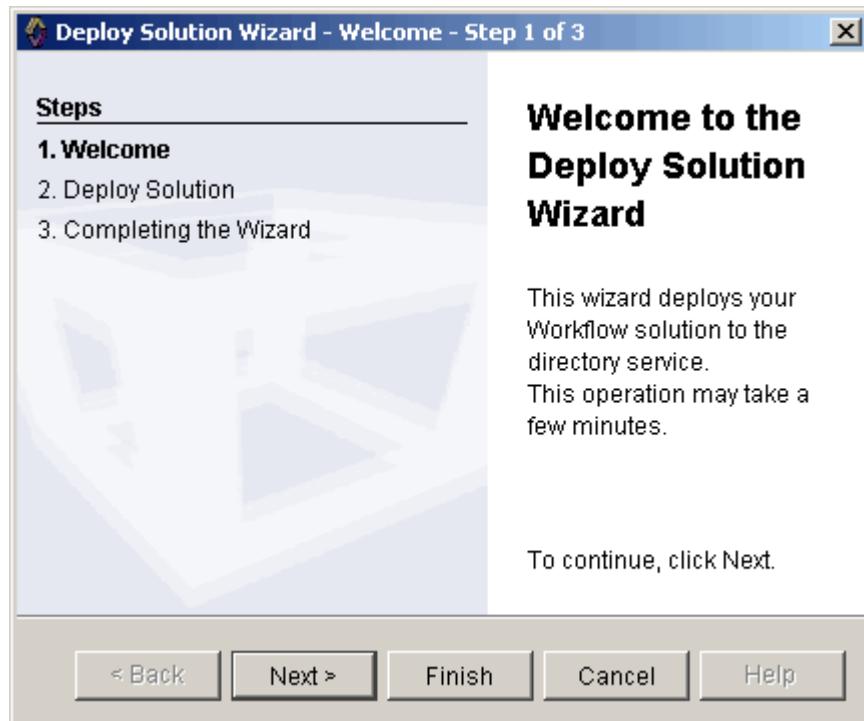


Figure 16-13 Deploy Solution Step 1

2. Confirm your configuration settings and click **Next>**.
3. Click **Finish** to complete the Task Manager project deployment.
4. Click **Close** to exit the Configurator.

ADVANCED TASK MANAGER CONFIGURATION

If you want to change the default settings, partition, or server configuration of the Task Manager properties after configuring the Task Manager as described above, follow the instructions here.

IMPORTANT: Do not modify any of the specialized events, BPOs, or control process models used by the Task Manager. Modification of these items may cause workflow failures.

WORKFLOW

Setting Up the Task Manager Project

The key steps required to use the Task Manager project include:

- Setting Database Project Parameters
- Setting the mailhost and Web server project properties
- Deploying the Task Manager project
- Running the Task Manager project

To set database project parameters:

1. Expand the Task Manager project.
2. Expand the Database Parameters.
3. Configure the following properties:
 - Database Behavior
 - Server Name
 - Port Number
 - Database Name
 - User Name
 - User Password

Note: Whether you use the default Task Manager project or use the advanced Task Manager configuration, you should configure the Task Manager project for database persistence. The Configurator prompts you for the database property values during configuration of the default Task Manager project.

For more information on creating and configuring project parameters, see the BME Help.

To set the Mail Host and Web Server Name project properties:

1. Click on the project object in the Explorer.
2. In the Properties Window, click the Runtime tab and select and set the values for the **Mail Host** and **Web Server Name** properties. Both these properties are used to facilitate notifications.

The **Mail Host** property must be set on your Task Manager project and on your model's project. To test that the property is set correctly, execute the following command to verify that it can be properly resolved and that no errors occur.

```
%nslookup <mailhost property value>
```

The **Web Server Name** property is used by the URLs in the notification messages. To use the default Task Manager as the Web server, set this property to: <http://localhost:8091>.

To deploy the Task Manager Project:

1. Specify a database to persist activity and task data. For instructions see “[To set database project parameters](#):” on page 16-32.
2. Open the Task Manager’s deployment configuration (*TaskManagerConfig*) from the **Explorer** window.
3. The default deployment assumes that your BusinessWare Server is named **bserv1**. If you are using a different name, rename the root node from **bserv1** to the name of your BusinessWare Server.
4. Save the deployment configuration using **File > Save**.
5. Make sure your directory server and your BusinessWare Server are running. Right-click the Partitioned Items node, and select **Deploy**. This will deploy the project in your directory server. Click **OK**.
6. Close the deployment configuration.

To run the Task Manager project:

1. Start the Task Manager project from either the Web Admin UI or the BME.

WORKFLOW PERFORMANCE TUNING

This section describes how to tune your workflow configuration for maximum performance.

CONFIGURING A SINGLE TASK MANAGER FOR MAXIMUM PERFORMANCE

This section describes how tasks are managed by BusinessWare and how to configure settings to enable full utilization of the CPU with a single task manager.

The following terms and settings are discussed in this section:

- **Web server thread pool**—the maximum number of threads that will be dedicated to handling Web requests.
- **Task fetcher thread pool**—a thread pool that is dedicated exclusively to getting Task Lists from different task managers.
- **Integration Server thread pool**—the maximum number of threads on the Integration Server.

- **Database connection pool**—each resource and target connector has its own connection pool. If the integration server uses multiple resources or target connectors, all these connection pools use the size specified in the Connection Manager of the integration server, that is, maximum connections.

In the case of the Task Manager project, the database connection pool serves the BusinessWare Workflow UI and Task Manager Integration Server for retrieving Task Lists and the three process model components that process events and task operations.

- **Concurrency of process model components**—maximum concurrency of the source queue connectors in the integration model, for example, Activity Operation, Activity, and Task queues.

Tip: Reduce your disk I/O and make sure that you have enough runtime resources, such as caches, threads, and database connections.

In addition, other parameters have an effect on system performance, such as heap size and batch commit size of source queues.

The following describes how tasks are handled in BusinessWare and how various factors influence performance:

1. When the BPO enters an activity state, the workflow infrastructure dispatches a request to a Task Manager to manage an activity by publishing an event to the Activity Operation queue in the Task Manager project.

Note: In this discussion, a port refers to a port in the integration model, not a socket port.

2. The Integration Server gets a thread from the Integration Server thread pool to handle that request, which turns the request to an event in the Activity Operation queue. Another thread that executes the Activity Control component processes the event and instantiates an activity instance.
3. The max concurrency determines the number of threads associated with the Activity Control component. The activity instance also publishes one or multiple Task Creation events to the Task queue.
4. The Task Control component processes those events and instantiates task instances.
5. Again, the max concurrency determines the number of threads allocated for the Task Control component to handle Task Creation events. Each of those threads performs queries via a database connection from the connection pool. The max concurrency settings of those task queues determine performance of activity and task creation.

Web server threads are used to service incoming the BusinessWare Workflow UI requests. A single Web server thread is dedicated to servicing all requests on a connection. Typically, a login sequence that leads to the inbox page establishes a few connections with the server, and the requests are sent to the Web server on these established connections. HTTP 1.1 connections, by default, are kept open awaiting further requests for a configurable (via the connection time out property of the Web server) period of time. Web server threads servicing requests on an HTTP connection are tied with servicing requests on that connection until the connection is closed explicitly by the client or the connection times out.

- **If the request is to retrieve the Task List**—the Web server thread handling the request spawns off one or multiple task fetcher threads depending on the number of Task Managers that need to be in contact. The task fetcher threads are also allocated separately in the task fetcher pool. Each task fetcher thread gets a database connection from the connection pool, runs the query, iterates through the result set, creates the tasks, merges them into the user inbox and places them in the global cache.
- **If the request is to perform a task operation**—the Web server thread sends a request to the Task Manager project, whose Integration Server gets a thread from its thread pool to handle the request which typically triggers a task transition. The thread gets a database connection from the connection pool to load necessary objects as well as persist the changed states.

The task fetcher and the Integration Server share the same thread pool, which allocates threads as needed. The number of threads created at runtime may be less than the maximum number of threads configured for the pool. If all threads in the thread pool are in use, the thread pool queues scheduling requests until one becomes available. There is no size limit on the queue.

Upon timeout, the users receive a message reporting the condition and asking them to retry the previous operation. If this situation occurs often, you should consider increasing the pool size and other hardware resources to see whether or not the condition improves.

In addition, upon completion of a task, an event is published to the Activity queue to notify its associated activity instance. The maximum concurrency of the connector affects how quickly task completion is reflected on the activity instances and activity completion to the original BPOs.

Finally, a task operation may involve multiple calls to Task Manager or the Integration Server that executes the application project.

CREATING MULTIPLE TASK MANAGERS

You should consider creating multiple Task Manager projects only if you want different Task Managers to physically manage tasks that involve certain activities or different groups of people. For example, you might consider designating a dedicated Task Manager to manage all the activities and tasks that are assigned to the finance group. As described in the previous section, you can boost the performance of the Task Manager by clustering TMServer.

Note: The fully qualified Task Manager project name must be restricted to 50 characters.

To create multiple Task Managers:

1. Start the BME and select **Project Manager** from the **Project** menu.
2. In the **Project Manager** window, click **Import**.
3. Select *installdir/projects/TaskManager.jar* as the import file name.

This automatically assigns the project name and populates the *Mountpoint Mappings* table. You must give a new name to the project (for example, *NewTM*) and change the directory of the new Task Manager project.

4. Click **OK**. This expands the project into the new directory structure and creates a new project in the BME. The new Task Manager project is now the currently open project.
5. Rename the *TaskManagerQueues* directory to something distinct.

The convention for naming the Task Manager Queues directory is *<project-name>Queues*, so using the example project name above gives you the name, *TMServerNewQueues*.

Note: Queues are deployed to a shared location in the namespace, so deploying multiple Task Manager projects requires you to change the name of the *TaskManagerQueues* directory to guarantee uniqueness.

6. Rename the Integration Server from *TMServer* to something distinct and specific to the new Task Manager project, for example, *TMServerNew*.

Note: Integration Servers are deployed to a shared location in the namespace, so deploying multiple Task Manager projects will require that you change the name of the Integration Server to guarantee uniqueness.

Note: The Integration Server in the Task Manager that runs all integration components must be named with the prefix *TMServer* where case is not sensitive.

7. Save the project.

If the Web server is enabled in the Integration Server above, for example, TMServerNew, you must copy the BusinessWare Task List files for the new server:

- 1 Copy the directory *installdir/web/servers/<your BusinessWare Server name>/TMServer/webapps/ROOT/awi* to *installdir/web/servers/<your BusinessWare Server name>/TMServerNew/webapps/ROOT/awi*
- 1 Copy the file *installdir/web/servers/<your BusinessWare Server name>/TMServer/webapps/ROOT/WEB-INF/web.xml* to *installdir/web/servers/<your BusinessWare Server name>/TMServerNew/webapps/ROOT/WEB-INF/web.xml*.

For more information, see the *BusinessWare Administration Guide*.

8. Change the server output log file name to something suitable for the project. Log files cannot be shared, so each server must use a separate file.

WORKFLOW

Workflow Performance Tuning

PART V: PROCESS ANALYSIS AND MONITORING

This part describes how to use process queries to monitor and correct the quality of service (QoS) delivered by your BusinessWare applications. It discusses the language used to write queries for monitoring QoS, process query language. It also describes how to display business process data by creating views of the BusinessWare processes.

Chapters include:

- [Process Query Models](#)
- [Process Query Language](#)
- [Process Views](#)

PART #: TITLE HERE GOES HERE

This chapter explains how process query models are used to analyze your business processes. When used in conjunction with process views, you can view data retrieved by process queries using Business Cockpit. See *Business Cockpit Guide* for more information. This chapter often refers to the concept of Business Process Objects (BPO) and assumes you have an understanding of this concept. For more information on BPOs see [Chapter 11, “Process Models: Defining and Using Business Objects.”](#)

Examples have been provided in the chapter from the ClaimsProcessing Sample and RequestForQuote (RFQ) Sample, which are BusinessWare samples that you can use to better understand queries. See the samples documentation shipped with BusinessWare at

`installdir\samples\modeling\ClaimsProcessingSample` and
`installdir\samples\modeling\RFQSample`, where `installdir` is the BusinessWare installation directory.

Topics include:

- [Introduction to Process Query Models](#)
- [Process Query Components](#)
- [Creating Process Query Models](#)
- [Setting Process Query Properties](#)
- [Defining Queries](#)
- [Deploying Process Query Models and Components](#)
- [Runtime Behavior of Process Query Models](#)
- [Query Performance at Runtime](#)
- [Handling Complex BPO Data](#)
- [Parameterized \(Drilldown\) Queries](#)

INTRODUCTION TO PROCESS QUERY MODELS

A process query model contains a list of queries defined in Process Query Language (PQL) against a BPO used in a process model. (At design time, a process model is associated with a BPO. During runtime BPOs are instantiated to store process data.)

Process query models are used by Business Cockpit and the BusinessWare runtime to get information for your business process by running the queries against the BPO to retrieve data.

You can create two types of process query models—snapshot and real-time—depending on your business requirement. The type of process query model that you choose to create will depend on the type of process data that you would like to generate, and how often you would need this data to be generated. See “[Runtime Behavior of Process Query Models](#)” on page 17-20 for more information. Also, see “[Query Types](#)” on page 17-7 for information on the types of queries the process query models support.

SNAPSHOT PROCESS QUERY MODELS

A snapshot process query model is used by Business Cockpit to generate a snapshot of the process data to render process views. Cockpit runs the queries in a snapshot process query model against a database to retrieve the BPO data persisted by a process component.

Snapshot process query models are ideal for reporting purposes, where query results need not be generated constantly. For business processes that need not be monitored constantly, it is simple and more efficient to use snapshot process query models.

See [Chapter 19, “Process Views”](#) for more information on snapshot process views. Also, see the *Business Cockpit Guide* for Cockpit related information.

REAL-TIME PROCESS QUERY MODELS

A real-time process query model is linked to a *process query component* in an integration model and used by the BusinessWare runtime to continuously monitor BPOs in a process model. The runtime continuously computes query results for queries in the real-time process query model and outputs the query results, as described later in this chapter. For more information on integration models and process query components [Chapter 6, “Integration Model Basics.”](#)

Real-time queries are key to building service-level management policies into your business processes. Service-level management is crucial for maintaining Quality of Service (QoS) in any business process. Service-level management encompasses creation, deployment, monitoring, and enforcement of Service Level Agreements (SLAs) based on a service level policy.

A service-level policy consists of a set of SLAs with a specification of how each SLA is to be enforced (that is, the associated actions that you will take when the SLA is violated). For example, if the time to fulfill an order exceeds 24 hours, your policy may state that the order should be shipped free of charge.

SLAs are contracts between a service provider and a service user that specify what QoS must be provided and with what level of quality. SLAs originated in the telecommunication industry, which uses them to quantify when the QoS falls below an expected level.

Real-time process query models are ideal for business processes that need to be monitored constantly.

PROCESS QUERY COMPONENTS

Process query components are used in integration models and link to real-time process query models. The link between a process query component and a process query model is defined by the Process Query Model Link property. This property is set at the component level and can be changed if you decide to use a different model.

In the Integration Model Editor, a process query component is wired to a process component (linked to process model with an underlying BPO) to continuously run queries against the BPO in the process model.

When an Integration Server is replicated as part of a cluster for load balancing, the process query components associated with that Integration Server are replicated also. For more information on clustered Integration Servers, see [Chapter 21, “Load Balancing.”](#)

POR TS

By default, the process query component contains one input port and output port.

- **Input port**—has only one interface, and it cannot be changed:
`processanalyzer.BPOTransition.`

Note: To link a process query component to a process component, you must wire the process query component's input port to a special output port called the *query port*  in the process component. See the *BME Help* for more details on how to wire a process query component to a process component.

- **Output port**—is untyped and has no interface specified. Although a process query component has only one output port by default, you can add more if required. Each output port will receive the same set of events in the same order. It is common for the output port to be wired to a Channel Target. This enables Business Cockpit to “listen” for new events and update its views dynamically.

CREATING PROCESS QUERY MODELS

Using the BME you can create a process query model and define queries to monitor a BPO. See “[Defining Queries](#)” on page 17-7 for more information on how to build queries.

PREREQUISITES

Before you create a process query model, you must ensure that a process model exists with a BPO defined. See [Chapter 10, “Process Models: Basic Concepts”](#) for more information on process models.

PROCEDURE

The procedure below briefly outlines the steps to be followed to create a process query model. For more detailed information refer to the *BME Help*.

1. Create a new folder to contain the process query model. (Right-click the root directory in the Explorer and select **New Folder** from the shortcut menu.)

or

Select an existing folder in the Explorer.

When creating a new folder, make sure you follow the naming conventions specified below:

- With the exception of the underscore “_”, do not use spaces or other punctuation.

- Specify folder names in English because folder names are used to generate module names during types generation, and only folder names specified in English are recognized and converted. See “[Generating Types](#)” on [page 17-15](#) for more information.
2. Create a process query model using any one of the following techniques:
 - Select **File > New...**. In the **New Wizard**, expand the **Models** node and select **ProcessQuery**.
 - Right-click a folder in the Explorer and select **New > All Templates... > Models > ProcessQuery**.
 - Double-click the process query component in the integration model to automatically link the new process query model to the process query component. Use this method only if you are creating a real-time process query model.
 3. When prompted, provide a name for the process query model and click **Finish**.
 4. Set model properties. Specify the BPO Type and Query Type properties in the Properties window. See “[Setting Process Query Properties](#)” on [page 17-6](#) for more information.

The selected BPO Type specifies the BPO that the process query model will query, for example, `claimObjects.Claim`. When you select a BPO type, all the attributes that belong to this BPO will be available to help you build queries against this BPO.

The selected Query Type determines the type of process query model—snapshot or real-time. The default type is snapshot. See “[Query Types](#)” on [page 17-7](#) for information on the types of queries supported by snapshot and real-time process query models.

5. For real-time process query models, there are additional steps that need to be performed as described in [step a](#) through [step f](#).
 - a. Select the process query component from the Integration tab of the Palette and place it in the Integration Model Editor.
 - b. Link the process query component to the process query model by double-clicking the component and selecting the process query model from the **Select Process Query Model** dialog box that appears. (If you created the process query model by double-clicking the process query component, the link was automatically set, and you do not have to perform [step a](#) through [step c](#).)

Note: Click **New** to create a new model.
 - c. Click **OK**.
 - d. Add a query port  to the process component, assuming a process component is already placed in the Integration Model Editor.

- e. Wire the process query component's input port to the process component's query port.
- f. Wire the process query component's output port to a target that will respond to output events. Typically the target will be a channel that will output into a process query component or be viewed by Business Cockpit.
6. Save the process query model and integration model by selecting **File > Save All**.
7. Use the Query Builder or Query Editor to define your queries. See “[Using the Query Builder](#)” on page 17-9 or “[Using the Query Editor](#)” on page 17-14 for more information.

SETTING PROCESS QUERY PROPERTIES

Properties can be set at model level, component level, and query level.

Note: In all property windows, you can display properties arranged in a logical, task-oriented order or in alphabetical order by clicking one of the sort buttons shown below. Throughout this chapter, the task-oriented order is used.

- Click the first button  for task-oriented order.
- Click the second button  for alphabetical order.

To view or set the component properties, select the component in the Integration Model Editor.

Table 17-1 Properties of the Process Query Component

Property	Description
Name	Name of the process query component.
Process Query Model Link	Link to the process query model associated with this component. Each process query component must link to one real-time process query model.

To view or set model properties, select the process query model in the Explorer.

Table 17-2 Properties of the Process Query Model

Property	Description
Name	Name of the process query model.
BPO Type	The interface type of the BPO being queried.
Query Type	Type of process query model—snapshot or real-time.
Description	Description of the process query model.

To view or set query properties, select the query in the Query Editor.

Table 17-3 Properties of a Query

Property	Description
Name	Name of the query. The query name must be a valid Java interface name and must not start with a leading “_”.
Parameters	Parameter name, which is the parent query alias name ending with _param. Make sure the name is different from the binding variable name used in the query and the aliases used in the SELECT clause. See “ Parameterized (Drilldown) Queries ” on page 17-27 for more information.

DEFINING QUERIES

You can define queries for your process query model in two ways:

- **Query Builder**—provides a graphical user interface to build queries. You simply select from prebuilt query clause tables and set parameters to define the information you like to see.
- **Query Editor**—provides a text box where you can manually type in the query strings.

QUERY TYPES

Before building queries, it will be useful to know *what* types of queries you can build using the Process Query Model Editor, and also, *which* query types are supported by the process query models.

Query types can be classified as follows:

- **Simple filter query**—monitors individual BPOs that meets certain conditions. A simple filter query does not use an aggregate function and is not time-based.
- **Aggregation query**—monitors a group of BPOs using a GROUP BY clause. The aggregation is performed based on the condition specified in the query.

Note: For aggregation queries in clustered Integration Servers, an internal database table is used to share global aggregation information for these queries among the clustered integration servers. For this reason, the Persistent Store property of the Integration Server must be set to RDBMS.

- **Time-based query**—monitors a BPO based on a time setting. The query uses a variable called CURRENT_TIME, which is a special PQL keyword. There are three types of Time-based queries:
 - **Time-based filter**—uses CURRENT_TIME variable in the WHERE clause.
 - **Time-based aggregation**—uses CURRENT_TIME variable within an aggregate function.
 - **Time-based filtering with aggregation**—uses the CURRENT_TIME variable within the WHERE clause and an aggregate function in the SELECT clause

The type of query that you define in a process query model depends on the *type* of process query model. Snapshot process query models support all query types. Real-time process query models support simple filters, aggregation, time-based filters, and time-based aggregation but do not support parameterized queries.

For information on query syntax and grammar, see [Chapter 18, “Process Query Language.”](#)

USING THE QUERY BUILDER

With the Query Builder, you construct your query (simple filter, aggregation, or time-based) simply by selecting BPO attributes, aggregate functions, and operators provided in the prebuilt clause tables.

If the query includes complex expressions, you can use the Expression Builder to construct these expressions. See “[Building Expressions](#)” on page 17-13 for more information.

If the query uses sequences or keywords such as `exists` or `forall`, you must use the Query Editor to create them. See “[Using the Query Editor](#)” on page 17-14 for more information.

The screenshot shows the Query Builder interface. On the left, a list of available clauses is displayed in a scrollable window. The clause "ShowLargeClaims" is currently selected and highlighted with a blue background. Below this list are buttons for "Add", "Remove", and navigation arrows. At the bottom of this pane is a "SnapshotQueryModel *x" button. To the right of this list is a large configuration panel divided into several sections:

- Query Builder**: A table titled "Selected Items" showing mappings between BPO attributes and aliases. The columns are "Aggregation", "Selected Items", and "Alias". The data includes:

Aggregation	Selected Items	Alias
None	(o.oid)	oid
None	(o.procedureDate)	procedure...
None	(o.doctor)	doctor
None	(o.patient.lastName)	lastName
None	(o.patient.firstName)	firstName
None	(o.claimAmount)	claimAmount
None	(o.claimDocument)	claimDocu...
- Where**: A table for defining conditions. It has three columns: "Left Operand", "Operator", and "Right Operand". One row is visible: "o.claimAmount" >= "5000".
- Having**: A table for defining group-level conditions. It has three columns: "Left Operand", "Operator", and "Right Operand". This section is currently empty.
- Action Buttons**: A row of buttons including "Add", "Remove", "Group", "Ungroup", and navigation arrows.
- Match Options**: Two radio buttons: "Match All Expressions" (selected) and "Match At Least One Expression".

Figure 17-1 Query Builder

To create a new query:

1. In the **Queries** panel, click **Add**. An empty query with a default name is created.
 - Use or buttons to reorder the queries. Reordering queries in a process query model does not impact the way the queries will be run.
 - Use the **Remove** button to delete queries.
2. Rename the query (right-click the query and select **Rename** from the shortcut menu).
3. Select the **Query Builder** tab.
4. To build the SELECT clause:
 - a. In the right pane of the Process Query Model Editor, click the first table with the selection items, and then click **Add** in the same pane. An empty row is added.
 - b. Select an aggregate function from the **Aggregation** column list to aggregate a group of BPOs. The following functions are available:
 - AVG—average of values in a column
 - COUNT—a count of all the BPO records grouped together (if this function is used with the (*) parameter)
 - SUM—total of values in a column
 - NONE—simply filters a BPO attribute (e.g., show all claims above a certain dollar value) without aggregating BPOs

For a description of the various aggregate functions, see [Chapter 18, “Process Query Language.”](#)

- c. From the **Selected Items** list, select any one of the following:
 - A BPO attribute (e.g., o.patient) to be analyzed
 - The “*” parameter to use with the COUNT aggregate function
 - **Custom** to build a complex expression. See [“Building Expressions” on page 17-13](#) for more information.

The **Selected Items** list displays all attributes that belong to the selected BPO (including DO and structs) and attributes from *nested* DOs and structs. For example, if `Claim` is the BPO selected that contains `Patient` as the DO with `firstName` and `lastName` attributes, the **Selected Items** list will display `o.patient`, `o.patient.firstName`, and `o.patient.lastName` to help you build a query against `Claim`.

Note: If you have selected an aggregate function (other than NONE) in the **Aggregation** column, the Query Builder will automatically add a GROUP BY clause with the selected attribute.

- d. Each BPO attribute has a predefined alias name that is displayed in the **Alias** column when you select a BPO attribute. The alias describes the structure of the data produced by a query. For example, if you select `o.bpState` as the BPO attribute, `bpState` is automatically displayed as the alias.

Note: You can change the alias name if needed, but make sure you do not enter a reserved word or keyword, such as COUNT, SUM, etc. when you modify the alias.

For more information on BPO attributes and aliases, see “[Source and Target Schemas](#)” on page 18-3.

5. To build a WHERE clause for your query:

- a. Click the **Where** clause table, and then click **Add** in the same pane. An empty row is added.
- b. From the **Left** and **Right Operand** columns, select a BPO attribute or *parameter name* to include in your WHERE clause. For example, `o.ClaimAmount` or `company_param`.

A parameter name is an alias name that you provide for a query when you parameterize a query. All parameter names end with “_param”, such as, `company_param`, which is automatically assigned by the system. See “[Parameterized \(Drilldown\) Queries](#)” on page 17-27 for more information.

You may also enter values to include in your query. For example, if you select the BPO attribute `o.ClaimAmount` in the **Left Operand**, and enter a value, such as 7000, in the **Right Operand** column with “=” selected as the operator, the WHERE clause will read as `o.claimAmount = 7000`.

- c. Select an operator from the **Operator** list to specify the type of action to be performed in the BPO filter process. The default operator is “=”.

6. To build a HAVING clause for your query:

- a. Click the **Having** clause table, and then click **Add** in the same pane. A new row is added.
- b. Select a variable or enter a value in the **Left** and **Right Operand** columns. (The Operand lists are populated with the alias names of the BPO attributes specified in the SELECT clause.)
- c. Select an operator from the list to specify the type of action to be taken.

7. In the **Where** or **Having** tables, group two or more expressions that you may want treated as a unit. Grouping maintains the AND/OR relationship within the group.
 - a. Click the **Where** or **Having** table and select the expressions to group. Click **Group**. The expressions are combined and a single expression is displayed.
 - b. In the resulting expression, select AND/OR as the operator from the **Operator** list. The default operator is “=”.

When you view a joined query in the Query Editor text box, the AND operator will be shown as “`&&`”, for example: `WHERE ((o.doctor = doctor_param) && (o.patient.insurance.company = company_param))` and the OR operator will be shown as “`||`”, for example: `WHERE ((o.doctor = doctor_param) || (o.patient.insurance.company = company_param))`.

Alternatively, you can group all the expressions listed in these tables without using the **Group** button by selecting the **Match All Expressions** (for AND) or **Match At Least One Expression** (for OR).

8. When you finish building a query, save the query by selecting **File > Save**.
9. Validate the process query model using any one of the following techniques:
 - By right-clicking the process query model in the Explorer and selecting **Validate** from the shortcut menu.
 - By right-clicking in the Query Builder window and selecting **Validate** from the shortcut menu.

Validation does the following:

- Checks the syntax of all queries in the process query model
- Displays the validation results in the Output window

Note: Validation checks only the query syntax regardless of the database used for persistence. These PQL queries are converted to SQL queries to be run against the database only at the time of deployment. See [“Deploying Process Query Models and Components” on page 17-16](#) for more information.

10. Continue adding queries as needed.

Building Expressions

You use the Expression Builder to construct complex query conditions for your SELECT clause by selecting operands (BPO attributes) and operators to be used with the operands. For example, `bpo.attribute1+bpo.attribute2` and `SUM(bpo.cost*bpo.num)` are complex expressions that you can build using the Expression Builder's **Select Expressions** dialog box. These expressions are then placed in the **Selected Items** list in the SELECT clause table of the Query Builder.

To build expressions using the Expression Builder:

1. In the Process Query Model Editor, click the **Query Builder** tab.
2. From the **Selected Items** list in the **SELECT** table, click **Custom**.

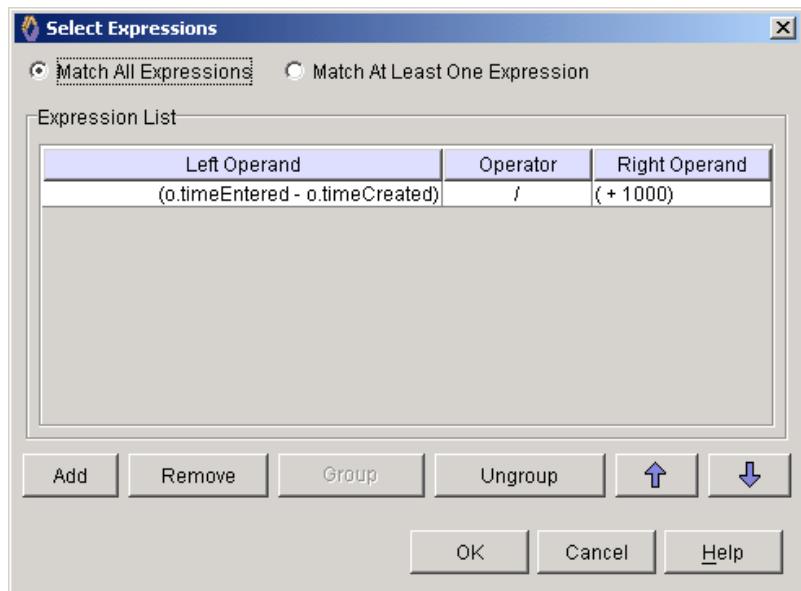


Figure 17-2 Select Expressions

3. In the **Select Expressions** dialog box (Figure 17-2) that appears, click **Add**. A row is added.
4. From the **Left** and **Right Operand** columns, select the BPO attributes to be used in the expression.
5. Select the operator to be used with the operands from the **Operator** list.
6. Create more expressions as needed by repeating step 3 through step 5.

7. Select **Match All Expressions** or **Match At Least One Expression** to join the expressions in the **Expression List** panel.
 - **Match All Expressions**—if selected, joins all the expressions with an AND.
 - **Match At Least One Expression**—if selected, joins all expressions with an OR.

Note: To use both AND and OR joins in your expressions, you must group expressions by clicking the **Group** button, which will be enabled once you select the expressions in the list.

USING THE QUERY EDITOR

The Query Editor allows you to write your own query strings (including sequence queries) without using the Query Builder or to refine queries that you constructed using the Query Builder.

The Query Editor allows you to manipulate the text of the query so you can easily copy and paste between queries and across models. This can be convenient if building large, complex expressions. It also enables you to use sequences and keywords not available in the Query Editor, such as `exists` and `forall`.

To refine queries generated using the Query Builder, you must select the **Enable Editing Text** checkbox in the Query Editor. The Query Editor will display the query string for you to make changes. You are allowed to toggle between the Query Editor and Query Builder while defining your queries. However, please be aware that any changes made to the query string using the Query Editor will be lost if you switch back to the Query Builder by clearing the **Enable Editing Text** checkbox.

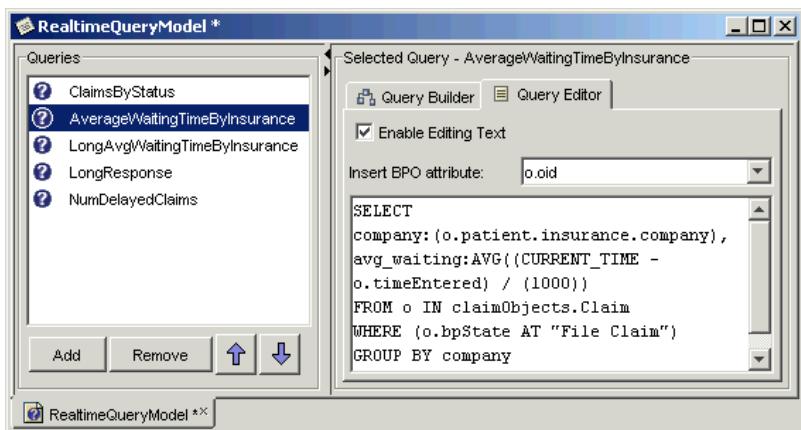


Figure 17-3 Query Editor

To write a query using the Query Editor:

1. In the **Queries** panel, click **Add** and create a new query.
2. Click the **Query Editor** tab.
3. Select the **Enable Editing Text** checkbox.
4. The **Insert BPO attribute** list box will be enabled. From the list, select the BPO attribute to add to your query string.
5. Begin writing your query string in the text editor box.
6. When finished, save the query by selecting **File > Save**.
7. Validate the query (right-click in the Query Editor and select **Validate** from the shortcut menu). Validation results are displayed in the Output window.

GENERATING TYPES

When you build your project, additional `DefinedType` objects and Java classes are generated automatically for real-time process query models. They are used at runtime by process query components to store, transport, and manipulate data.

Note: Additional classes are not generated for snapshot process query models because there is no process query component that requires them.

When you build a project containing process queries, the additional classes are created in the *GeneratedTypes* directory (Figure 17-4). The top node of the *DefinedType* structure is the module , which is created using the name of the folder containing the process query model. For example, in Figure 17-4 the module name is *ClaimsProcessingViews_RealtimeQueryModels*. The module contains interfaces and events corresponding to the query in the process query model.

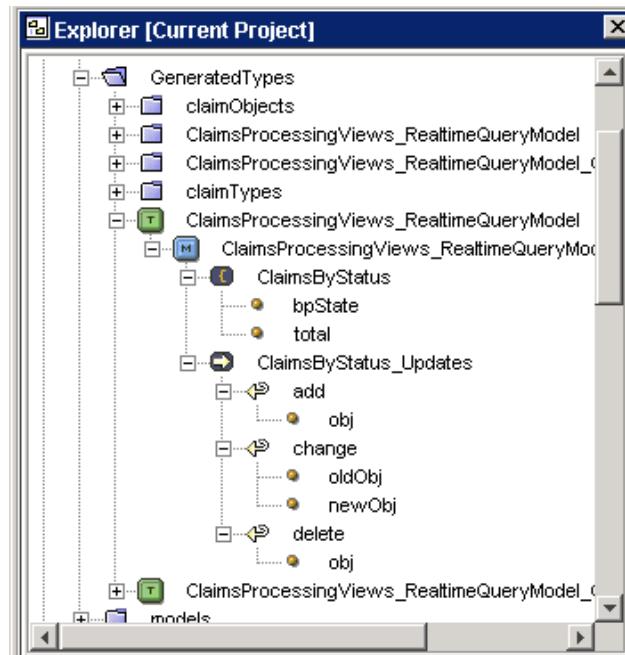


Figure 17-4 Explorer Showing Generated Types

DEPLOYING PROCESS QUERY MODELS AND COMPONENTS

Process query models are deployed when you deploy your BusinessWare project, and SQL queries are generated for use at runtime from the PQL queries defined in the model.

A process query component is partitioned automatically based on the process component to which it is wired. See [Chapter 22, “Deploying Projects”](#) for more information on partitioning.

IMPORTANT: Process query models can be shared by multiple process query components only if the process query components exist in different Integration Servers. That is, process query components partitioned to the same Integration Server cannot share the same process query model.

SQL queries generated from snapshot process query models are used by process views to retrieve BPO data from the database used for persistence. In the case of a real-time process query model, SQL queries are used by the process query component to retrieve *initial* BPO data from the database.

CUSTOMIZING SQL QUERIES

You can customize SQL queries generated from real-time process query models. For example, if the data representation of your BPO has been changed from the default representation, the correct SQL queries may not be generated, in which case you must modify the SQL queries. (SQL queries generated from snapshot process query models are not customizable).

To customize SQL queries:

1. Create a deployment configuration. See [Chapter 22, “Deploying Projects”](#) for more information.

In the Deployment Editor, all the components in the integration model are displayed under the **Items to Partition** node. The **Queries** folder under the process query component node will initially appear empty because SQL queries have not yet been generated.

Note: A process query component is always partitioned together with the process component to which it is wired. The process query component by itself is not partitionable.

2. Partition the components (right-click the **Items to Partition** node and select **Auto-Partition**). The components will be partitioned and listed in the **Partitioned Items** panel.

PROCESS QUERY MODELS

Deploying Process Query Models and Components

3. Expand the **Queries** folder in the **Items to Partition** panel, and you will now find SQL queries generated and listed (Figure 17-5).

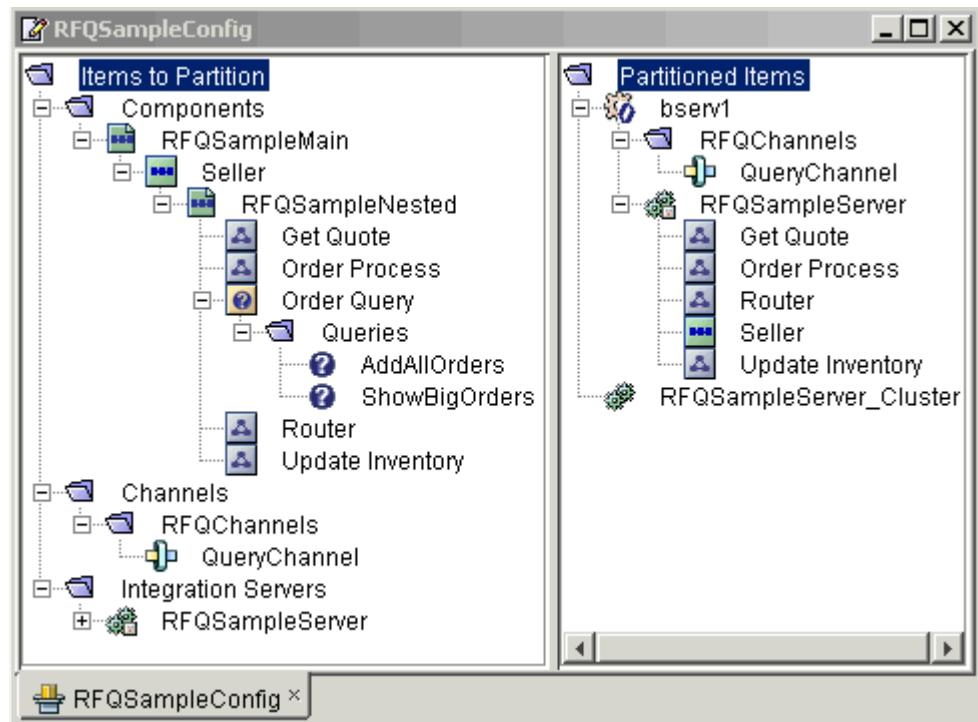


Figure 17-5 Queries Folder Showing Generated SQL Queries

SQL queries will not be generated in the following cases:

- If the process query model contains only simple filter queries.
- If the process component to which the process query component is connected is not partitioned.
- The process component is partitioned, but the project has not been built.
- If the Integration Server to which the process component is partitioned does not have its persistence set to RDBMS. (Typically in production environments, Integration Servers are configured to use database for persistence.)

4. Select the query to customize, and click the **Customizer** button  in the Properties window.

Note: The **Customizer** button is available only if the query you have selected is an aggregation or a time-based query.

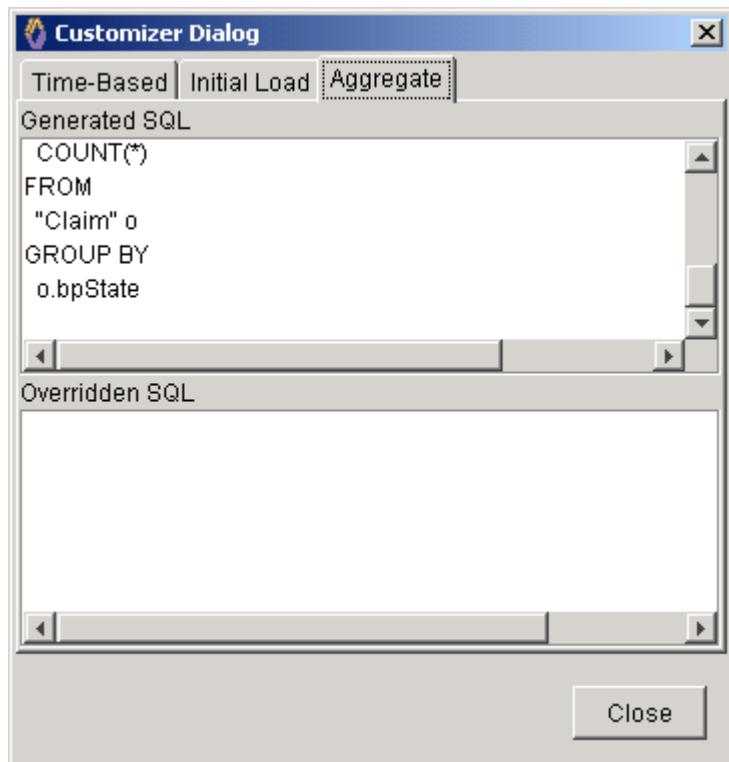


Figure 17-6 Customizer Dialog Box

5. In the **Customizer** dialog box, enter a SQL query to override the generated SQL query, and click **Close**.

Tip: A good way to submit an overridden query is to copy, paste, and modify the generated SQL.

IMPORTANT: This is one of the more advanced features of the process query model, and if you have difficulties modifying generated queries, contact technical support for assistance.

Retaining Query Customizations

Given below are some instances of how refreshing a deployment configuration impacts customized queries.

- If nothing changes with respect to the queries, refresh will retain customizations.
- If the actual content (PQL) of the query changes before you refresh, the customizations will be retained. However, after you refresh, the generated SQL will change because the PQL has changed, and the customizations may be invalid.
- If the query name changes and you refresh, the customizations are lost.

PURGING THE AGGREGATION RESULTS TABLE

After redeploying a load-balanced project that contains aggregation queries, you must purge the process query component using vtadmin or Web Admin.

DEPLOYING CLUSTERS CONTAINING QUERIES

To partition process query components to clustered Integration Servers, the Integration Server's Persistent Store property must be set to RDBMS. If the Integration Server is not configured correctly, you will see the following error:

Integration Server '*Server Name*' is clustered and has Persistent Store property set to Cache. Process Components connected to Process Query Components cannot be partitioned to it.

IMPORTANT: After deploying a project with a cluster that contains an aggregation query, you must purge the database containing the aggregation results table.

For more information on aggregation queries, see “[Query Types](#)” on page 17-7.

For more information on creating clusters, see “[Load Balancing](#)” on page 21-1.

RUNTIME BEHAVIOR OF PROCESS QUERY MODELS

This section describes the runtime behavior of snapshot and real-time process query models.

RUNNING THE SNAPSHOT PROCESS QUERY MODEL

At runtime, Business Cockpit runs the SQL queries generated at deployment time against the database set to persist BPO data and retrieves the information. See the *Business Cockpit Guide* for more information.

RUNNING THE REAL-TIME PROCESS QUERY MODEL

Runtime data from real-time process query models can be:

- Analyzed to correct the QoS that SLAs deliver in the case of violation.
- Visualized via process views for your business needs. For example, as reports for business analysis.

During runtime, the process component *sends* events (generated by BPOs transitioning from one state to another) to the process query component to which it is associated. On receiving the events, the process query component carries out the following:

- Uses the queries defined in the associated process query model to analyze the events from the process component.
- Returns a query result of all the BPOs that are in violation of the QoS constraints.

The process query component generates *update events* based on changes in the query result. The process query component pushes the update events to its output port. From the output port these update events are sent to the target for processing. See “[Processing Update Events](#)” on page 17-23 for more information.

PROCESS QUERY MODELS

Runtime Behavior of Process Query Models

Figure 17-7 illustrates the runtime process.

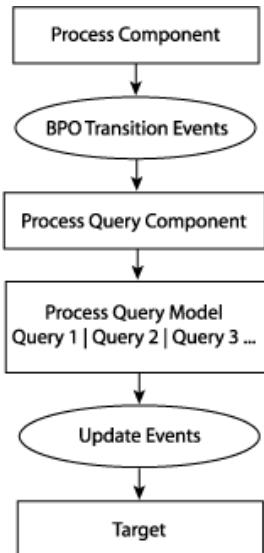


Figure 17-7 Runtime Process

CREATING UPDATE EVENTS

A process query component produces the following three types of update events:

- **add**—when a new record is added to the query result. This event contains the data needed to add a new record.
- **change**—when an existing record in the query result is changed. This event contains a copy of the record before the change, plus a copy of the record after the change.
- **delete**—when a record is deleted from the query result. This event contains a copy of the record to be deleted.

The `add` and `delete` events have a single parameter called `obj`. The `change` event has two parameters—`oldObj` and `newObj`. All these three parameters have the same data type, which is an IDL struct. Given below is an example of an IDL struct.

Example: Generated IDL Struct

```
module ClaimsProcessingViews_RealtimeQueryModel {
    interface ClaimsByStatus_Updates;
    struct ClaimsByStatus {
```

```

        bpe::StatesGroup bpState;
        long total;
    };
    interface ClaimsByStatus_Updates {
        event void add(in ClaimsByStatus obj);
        event void change(in ClaimsByStatus oldObj,
                          in ClaimsByStatus newObj);
        event void delete(in ClaimsByStatus obj);
    };
}

```

PROCESSING UPDATE EVENTS

It is the responsibility of the target to use the update events it receives to enforce SLAs and correct SLA violations. The target can be any BusinessWare object, such as a process component (to represent another process model), or a target connector, such as a channel target connector (to publish events for a subscriber application, such as Business Cockpit). The target receives update events from the process query component, interprets them, and takes whatever actions agreed upon to remedy the associated violation.

IMPORTANT: For performance reasons, the process query model does not update its queries in the same transaction as the process component that it serves. Consequently, if the Integration Server goes down after committing the transaction in the process model but *before* the process query model has computed the query updates, the query updates are lost. From the perspective of the target, it is possible (although not likely) that the notification from a query monitoring a specific condition may be lost.

Using a Downstream Process Component

A common way to enforce your service-level constraints is to use a process component downstream of the process query component.

Example:

Suppose that a QoS metric that must be monitored requires that all orders must be processed in less than an hour. Suppose also that in the case that the constraint is violated, one of a number of support engineers must be notified to take corrective action.

The following time-based query is designed to monitor this specific service constraint. It selects information from all orders that wait in the `ProcessOrder` state for more than an hour (3,600,000 milliseconds.)

```
select
    oid:o.oid
    customer:0.order.customer
    quantity:o.order.quantity
from
    o in rfqObjects.OrderBPO
where
    (o.bpState at "ProcessOrder") and
    (current_time - o.timeEntered > 3600000)
```

You can imagine building a downstream process component that connects to the output port of the process query component that includes this query. When an update event for this query is received, a special workflow activity is created for the appropriate support engineer and an email is sent (using an Email Target Connector) for further notification.

In many cases, for performance reasons it may be advisable to partition the process component that takes corrective action onto a different Integration Server than the one used for the source process component.

For more information about creating and working with process models, see [Chapter 10, “Process Models: Basic Concepts.”](#)

Using a Subscriber Application

A *subscriber application* is a custom program that receives the update events from a channel. In terms of a process query model solution, you must connect the process query component to a Channel Target connector that publishes to a channel to which the custom program should subscribe. The subscriber should implement the `_Updates` interface that is generated for the query. For an example of a custom subscriber, see the `installdir\samples\protocols\SimpleOrderSample`.

VISUALIZING UPDATE EVENTS

Process views linked to real-time process query models can subscribe to a channel to get up-to-date BPO information via update events generated by a process query component. This information can be viewed using Business Cockpit. See the *Business Cockpit Guide* for more information.

QUERY PERFORMANCE AT RUNTIME

Filter queries have a very low overhead in terms of their runtime behavior. Each BPO transition is examined, and if the filter condition is met, an update event is generated.

Alternatively, aggregation queries impose a higher cost on the runtime system. After each BPO transition it is necessary to search the current query result to see if a BPO transition group exists. Therefore, it is required to keep the current query result in memory. Usually, the number of elements of the query result for an aggregation query is much less than the number of BPOs and it is not a problem to store the result in memory. However, you can also construct queries for which the query result is very large. An example of this, is a query that groups based on the `oid` of the BPO.

In addition to the impact of the size of the query results stored in memory, you may notice some startup lag if there are many aggregation queries in your project. This is because query results have to be initialized from database at startup because they are stored in memory.

Time-based queries are implemented in a relatively complex fashion, a full description of which is beyond the scope of this chapter. However, in general the memory burden for a time-based query is relatively light. Time-based queries require database queries to be run periodically, which can incur some performance overhead. Database queries are run periodically based on the time duration specified in the queries at design time. For example, if a query selects BPOs that remain in a state for an hour, the database query is likely to be run periodically every six minutes (approximately 1/10th the time interval).

If you have a shorter time duration in your query, database queries will be run at more frequent intervals. It is therefore not advisable to have a very short time duration and a lot of BPOs that expire because this may have an adverse performance impact.

HANDLING COMPLEX BPO DATA

This section describes how you can handle complex BPO data in your queries.

FILTERING AND AGGREGATING SEQUENCE ATTRIBUTES

You can create a sequence query to filter or aggregate a BPO attribute that is a sequence. A sequence attribute is accessed through a separate binding variable defined in the FROM clause, which cannot be created using the Query Builder. You must use the Query Editor when working with sequence attributes. For examples and more information, see “[Filtering and Aggregating Sequence Attributes](#)” on page 18-13 in Chapter 18, “Process Query Language.”

CLOB DATA TYPE LIMITATIONS

A query can include a BPO attribute set to Character Large Object (CLOB) data type to store large data such as XML. For example, `o.claimDocument` in the ClaimsProcessing Sample is a BPO attribute with the data type specified as String `<10000>` mapped to CLOB.

A known limitation of using BPO attributes of CLOB type in your query is that you cannot aggregate or filter them. Therefore, aggregation queries that use aggregate functions such as COUNT, SUM, or AVG cannot be performed on BPO attributes set to CLOB data type. However, you can build simple filter queries to monitor BPO attributes of this type.

UNION DATA TYPE LIMITATIONS

Process queries currently do not support union data type. However, you can query attributes related to BPO interfaces that contain union data type of other unsupported data, without directly or indirectly referencing these data types.

LIMITATIONS OF MULTIPLE BPOs SHARING DATA OBJECTS

In case of an aggregation query, process query components maintain incremental query results based on data modified by the process component to which they are connected. If BPOs in a different process component are sharing data objects, the updates made by that process component will not be incorporated, thereby producing incorrect query results.

PARAMETERIZED (DRILLDOWN) QUERIES

Parameterized queries have named parameters that allow actual values to be substituted into the query at run time. Parameterized queries should be defined only in snapshot process query models and are used by drilldown views to provide the Business Cockpit user with more detailed information. See [Chapter 19, “Process Views”](#) for more details on drilldown views. Also, see the *Business Cockpit Guide* for runtime information.

To add parameters to a query:

1. Select a query and select **Properties**.
2. Click on **Parameters** to bring up the **Edit Parameters** dialog box. Enter the required values.

The parameter properties are described in [Table 17-4](#).

Table 17-4 Parameterized Query Properties

Parameter	Description
Name	Parameter name, which is the parent query alias name ending with _param. Make sure the name is different from the binding variable name used in the query and the aliases used in the SELECT clause.
Type	Parameter type, such string, long, integer, etc. Only primitive types are supported.

Example: This is the TotalAmountByDoctor query from the ClaimsProcessing Sample (considered as the parent query in our example) that groups and lists by doctor and totals the claim amount for each doctor.

```
SELECT doctor:(o.doctor), claimAmount:SUM(o.claimAmount),
FROM o IN claimObjects.Claim
GROUP BY doctor
```

Example: This is the TotalAmountByDoctor_DD query from the ClaimsProcessing Sample, which is a parameterized query that further drills down the parent query shown above to show the claims status using the alias name `doctor` in the parent query as the parameter (`doctor_param`).

```
SELECT doctor:(o.doctor), claimAmount:(o.claimAmount),
firstName:(o.patient.firstName),
lastName:(o.patient.lastName),
procedureDate:(o.procedureDate)
```

PROCESS QUERY MODELS

Parameterized (Drilldown) Queries

```
FROM o IN claimObjects.Claim  
WHERE (o.doctor = doctor_param)
```

Queries in a process query model are built using a subset of the Object Query Language (OQL), called the *Process Query Language* (PQL). This chapter describes how queries can be built using PQL and provides useful examples from the ClaimsProcessing Sample wherever required.

Note: The ClaimsProcessing Sample is a BusinessWare sample, which you can use to better understand queries. See the documentation on ClaimsProcessing Sample, which is shipped with BusinessWare at `installdir\samples\modeling\ClaimsProcessingSample`, where `installdir` is the BusinessWare installation directory.

Topics include:

- [An Introduction to Queries](#)
- [Examples of BPO and Query](#)
- [Source and Target Schemas](#)
- [Query Components](#)
- [Query Types](#)
- [Query Grammar](#)

AN INTRODUCTION TO QUERIES

A query is a formalized instruction to a database to either return a set of records (as query results) or perform a specified action on a set of records as specified in the query. A query result contains zero or more records, called a *collection*. Data in each collection record varies according to the query definition. Every record in a collection must satisfy the query condition. Queries defined in a process query model are run against a BPO in a process model. See [Chapter 17, “Process Query Models”](#) for more information.

EXAMPLES OF BPO AND QUERY

This section provides examples to help you understand the basic structure of a BPO and query before you read advanced information related to the different query types and their syntax.

Example: A BPO showing the IDL struct and definition of the BPO

```
module claimObjects {  
  
    typedef unsigned long long TimeStamp;  
    typedef string<10000> ClaimDocument;  
  
    interface Claim: bpe::BusinessProcessObject {  
        attribute Patient patient;  
        attribute string doctor;  
        attribute claimTypes::ProcedureList procedures;  
        attribute TimeStamp procedureDate;  
        attribute float claimAmount;  
        attribute float coveredAmount;  
        attribute ClaimStatus status;  
        attribute ClaimDocument claimDocument;  
    } ;
```

Figure 18-1 A BPO Example

Example: Query

```
SELECT doctor:(o.doctor), claimAmount:SUM(o.claimAmount)  
FROM o IN claimObjects.Claim  
GROUP BY doctor  
having claimAmount > 10000
```

Figure 18-2 A Query Example

In the example in [Figure 18-2](#), the query returns a collection that contains the doctors and claim amount of all the doctors that have a combined balance of more than \$10,000. The query has the following effect (even though it does not perform its actions as described here):

- It groups all BPOs by doctor.
- It adds the values of all the `claimAmount` fields for all BPOs in the same group (that is, having the same name). This results in one `[doctor, claimAmount]` pair record for each name.

For each record, if the value of `claimAmount` is greater than 10,000, the record becomes part of the query's collection.

SOURCE AND TARGET SCHEMAS

Source schemas and *target schemas* are important concepts tied to queries. The raw material for a query comes from data that gets its structure from the source schema, and the data produced by a query gets its structure from the target schema.

With respect to a query, the source schema consists of all the BPO attributes being analyzed and all the data objects (DOs) to which the BPO points. (For example, if one of the attributes of the BPO is a DO, then all the attributes of the data object are part of the source schema.)

The following attributes belong to every BPO. Because of this, they are available as part of every source schema:

- `timeCreated`—returns the time the current BPO was created.
- `timeEntered`—returns the time the current BPO entered its current state.
- `numTransitions`—returns the number of transitions the current BPO has taken within its process model.
- `bpState`—returns a sequence of structs that represent the BPO's current state.
- `oid`—returns the object identifier (`oid`) field of the current BPO.

The *aliases* named in the SELECT clause of a query make up the attributes of the target schema. (See the section “[Aliases](#)” on page 18-4 for details.) The target schema describes the structure of the data produced by each query. For example, the target schema of the query in the following code contains two attributes: `doctor` and `claimAmount`.

```
SELECT doctor:(o.doctor), claimAmount:SUM(o.claimAmount)
FROM o IN claimObjects.Claim
GROUP BY doctor
having claimAmount > 10000
```

QUERY COMPONENTS

This section explains the components that make up a query. See “[Query Grammar](#)” on page 18-15 for details on the syntax of a query, including the meaning of various token names (printed in italic).

ALIASES

The data items declared in the *selectDeclarationList*—that is—the text that follows the keyword SELECT—take the form “*alias:expr*” (for example, “*doctor:o.doctor*”). The name before the colon is called an *alias* because it is a substitute name given to the expression after the colon. In the example earlier in this paragraph, *doctor* is an alias for the expression *o.doctor*. It becomes one of the attributes of the query’s target schema.

SELECT CLAUSE

The SELECT clause specifies the columns to be returned by a query. It lists the data items that are to be included in the target schema, that is, it determines what data goes into each record of an update event. The data items declared in the SELECT clause take the form “*alias:expr*”.

The expression that follows each alias can be either:

- An attribute of the source schema
- A computed expression using any of the permitted attributes
- An aggregate function (sum, avg, or count) using the permitted attributes

See “[Query Grammar](#)” on page 18-15 for details.

FROM CLAUSE

The FROM clause specifies the BPO from which to retrieve BPO data. The FROM clause is always required in a query.

You can use a sequence binding variable in a FROM clause to filter and/or aggregate BPO attributes that are sequences. See “[Filtering and Aggregating Sequence Attributes](#)” on page 18-13 for more information.

WHERE CLAUSE

The WHERE clause filters out the specified BPO from the query result. Both simple filter queries and aggregation queries can contain a WHERE clause. However, a WHERE clause is optional. If the logical expression in the WHERE clause evaluates to false, the BPO being evaluated is not considered by the query. The logical expression that follows the keyword WHERE can contain only variables that are represented in the source schema.

Quantifiers

The existential quantifier “exists” and the universal quantifier “forall” are used in the WHERE clause to put a condition on an attribute of a sequence type. The conditional expression with an existential quantifier is True if and only if *at least* one element in the collection satisfies the condition. The conditional expression with a universal quantifier is True if and only if *all* the elements in the collection satisfy the condition.

Example: The following is the ClaimsWithShoulderProcedures query from the ClaimsProcessing Sample, which is an example of an existential quantifier.

```
SELECT oid:(o.oid), procedureDate:(o.procedureDate),
       doctor:(o.doctor), lastName:(o.patient.lastName),
       firstName:(o.patient.firstName),
       claimAmount:(o.claimAmount),
       claimDocument:(o.claimDocument)
FROM o IN claimObjects.Claim
WHERE exists p in o.procedures: p = "5590" OR p = "6099"
```

Example: The following is the ClaimsWithShoulderOnlyProcedures query from the ClaimsProcessing Sample, which is an example of an universal quantifier.

```
SELECT oid:(o.oid),
       procedureDate:(o.procedureDate), doctor:(o.doctor),
       lastName:(o.patient.lastName),
       firstName:(o.patient.firstName),
       claimAmount:(o.claimAmount),
       claimDocument:(o.claimDocument)
FROM o IN claimObjects.Claim
WHERE forall p in o.procedures: p = "5590" OR p = "6099"
```

GROUP BY CLAUSE

The GROUP BY clause combines a set of BPOs for the purpose of aggregation. The *identifierList* that follows the GROUP BY keyword can contain one or more alias names separated by commas. This list specifies which data items (identified by their alias names) are to be used to group BPOs for the purpose of aggregation.

Note: A GROUP BY clause is added automatically when you define an aggregation query using the Query Builder. See [Chapter 17, “Process Query Models”](#) for more information.

The presence of a GROUP BY clause imposes some constraints on the data items specified in the SELECT clause. Each *alias:expr* pair must meet one of the following conditions:

- The *alias* must be listed in the GROUP BY clause.
- The *expr* must be an aggregate function (sum, avg, or count).

HAVING CLAUSE

Just as a WHERE clause is used to filter what otherwise would be the output of a basic query, the HAVING clause is used to filter what otherwise would be the output of an aggregation query. The HAVING clause displays records grouped by the GROUP BY clause that satisfy the condition of the HAVING clause. If the logical expression following the HAVING keyword evaluates to *false*, a data record that would otherwise be included in the collection is excluded.

Note: All variables listed in the HAVING clause’s logical expression must be aliases.

NOTES ON SELECTED OPERATORS AND ATTRIBUTES

This section provides a description of selected operators and their attributes.

at Operator and bpState

You can use the `at` operator and the `bpState` attribute to create queries that depend on knowing the current state of the BPO. The `bpState` attribute returns a sequence of structs that includes the name of the state that the BPO is currently in. The `at` operator returns a value of `true` if the string on the right side is contained anywhere within the sequence of structs returned by the `bpState` attribute.

You can combine these two constructs to test whether the current BPO is in a given state or not. For example, the clause:

```
where not(o.bpState at "Finished")
```

is true when the current BPO is in any state except the one named “Finished”.

IMPORTANT: The at operator is valid only in conjunction with the bpState attribute.

in Operator and Membership in a Collection

In a WHERE clause, you can use the `in` operator to determine whether the value on the left side of the `in` keyword is a member of the value on the right side of the keyword. (The value on the right must be a sequence.)

For example, if the variable `weekdaysList` contains the strings “Monday” through “Friday”, then the following WHERE clause is true when “Tuesday” is one of the strings included in `weekdaysList`:

```
where ("Tuesday" in weekdaysList)
```

like Operator and Wildcard String Matching

The `like` operator is a general-purpose string-matching operator that allows the use of wildcards in the string to be matched. The wildcard characters:

- `%`, which matches an arbitrary sequence of zero or more characters
- `_` (underscore), which matches any single character

For example, the clause, `where a.custName like "Scott%"`, matches any customer name that begins with “Scott”.

The clause, `where a.custName like "%Scott%"`, matches any customer name that contains “Scott” anywhere in the string.

The clause, `where a.custName like "%Scott"`, matches any customer name that ends in “Scott”.

Finally, the clause, `where a.custName like "_____ %"`, matches any customer name the first word of which has five letters. (This string contains the following characters: five underscore characters, a space character, and the percent-sign character.)

Using the `oid` Attribute

When BusinessWare creates a BPO, it places a user-defined string into a field called `oid`. If you know what kinds of strings the `oid` field contains, you can use the `like` operator to create queries based on the value of each BPO's `oid` field. For example, a query might contain the following `where` clause:

```
where o.oid like "%Premium Customer%"
```

This clause would cause the selection of only those BPOs that have the string "Premium Customer" somewhere in the BPO's `oid` field.

QUERY TYPES

This section describes (with examples) the different query types supported by process query models.

SIMPLE FILTER QUERY

A simple filter query monitors individual BPOs. [Figure 18-3](#) shows the syntax of a simple filter query.

```
select selectDeclarationList
      from variableDeclarationList
      [whereClause]
```

Figure 18-3 Simple Filter Query Syntax

Square brackets indicate text that is optional. Text in *italics* represents a token—replacement text specific to a given situation—and is not to be typed into the query.

The `SELECT` clause specifies the target schema (the values that will be in each record of the query's collection).

The `FROM` clause specifies the source schema. It specifies which BPO is the source of the data. It cannot refer to a DO type.

Although the `WHERE` clause is optional, it is this clause that adds a filtering function to a simple filter query. The data that satisfies the condition in the `WHERE` clause is included in the query's collection. All the attributes referenced in this clause must come from the source schema.

[Figure 18-4](#) shows the ShowLargeClaims query from the ClaimsProcessing Sample that filters by claim amount.

```
SELECT oid:(o.oid), procedureDate:(o.procedureDate),
       doctor:(o.doctor), lastName:(o.patient.lastName),
       firstName:(o.patient.firstName),
       claimAmount:(o.claimAmount)
FROM o IN claimObjects.Claim
WHERE (o.claimAmount >= 5000)
```

Figure 18-4 A Simple Filter Query Example

As shown in [Figure 18-4](#), a simple filter query must contain a SELECT clause and a FROM clause; a WHERE clause is optional.

AGGREGATION QUERY

An aggregation query is performed not on individual BPOs but on a meaningful group of BPOs. Each record in the collection created by an aggregation query is based on data gathered since the query began running. The aggregation is performed based on the condition specified in the query.

[Figure 18-6](#) shows the syntax of an aggregation query.

```
select selectDeclarationList
      from variableDeclarationList
      [whereClause]
      [group by identifierList]
      [having expr]
```

Figure 18-5 Aggregation Filter Query Syntax

Three functions, called *aggregate functions*, are used in the SELECT clause to create an aggregation query:

- sum—returns the sum of a given expression for all the BPOs that have been grouped together.
- avg—returns the average value of a given expression for all the BPOs that have been grouped together.
- count—returns the number of BPOs that have been grouped together.

For a query to be an aggregation query, at least one of the expressions in the SELECT clause must use an aggregate function.

The GROUP BY clause, which is an optional clause, causes records to be grouped by a given data field. Wherever an *aggregate function* (for example, `sum(. . .)`) is present in the SELECT clause, all the records that have the same value for the “group by” data field are “collapsed” into one data record according to the value function that is being invoked.

The HAVING clause, which is optional, is typically used when the GROUP BY clause is present. It filters the records created by the GROUP BY clause. Only those records that match the specified logical expression become part of the query’s collection.

[Figure 18-6](#) is the NumClaimsByDoctor query in the ClaimsProcessing Sample that shows an aggregation query totalling the number of claims by doctor.

```
SELECT doctor:(o.doctor), total:COUNT( * )
FROM o IN claimObjects.Claim
GROUP BY doctor
```

Figure 18-6 Aggregation Query Example

The query in [Figure 18-6](#) aggregates the total number of claims and groups them by doctor.

This query is an aggregation query because of the alias named `total`, which is computed for all the qualifying BPOs of the same `doctor`.

Restrictions on Aggregation Queries

Regardless of whether an aggregation query has a WHERE clause, there are four possible forms for an aggregation query based on the presence or absence of GROUP BY or HAVING clause. The best way to explain the restrictions on an aggregation query is to describe them for each of the four possible cases.

- **A SELECT ... FROM query**—In this case, every expression in the SELECT clause must contain an aggregate function. An aggregation query of this form always has exactly one record in its collection. This record contains the values of each of the aggregate functions in the SELECT clause.
- **A SELECT ... FROM ... having query**—An aggregation query of this form is exactly like a SELECT ... FROM query, except that its collection will have zero records if the single record that would otherwise be present causes the HAVING clause to evaluate to `false`.

- **SELECT ... FROM ... GROUP BY query**—The restriction for this form of an aggregation query is that if an expression in the SELECT clause does not include an aggregate function, the alias for that expression must be listed in the GROUP BY clause.
- **A SELECT ... FROM ... GROUP BY... HAVING query**—An aggregation query of this form is exactly like a SELECT... FROM ... GROUP BY query, except that records are excluded from the collection if they cause the HAVING clause to evaluate to `false`.

TIME-BASED QUERY

Time-based queries use a special variable called `CURRENT_TIME` that returns the current time and special BPO attributes called `o.timeCreated` and `o.timeEntered`. Time-based queries are different from the other query types because they do not need a change in the BPO state to update events.

There are three types of time-based queries:

- Time-based *filter*
- Time-based *aggregation*
- Time-based *filtering with aggregation*

Note: The `CURRENT_TIME` variable and the BPO attributes—`timeCreated`, and `timeEntered`—are expressed in milliseconds. For example, a duration of seven seconds is expressed as 7,000 milliseconds.

Time-Based Filtering

A time-based filter query is one that uses the variable `CURRENT_TIME` in the WHERE clause.

Example: Figure 18-7 is the LongResponse query from the ClaimsProcessing Sample. The query is interested in BPOs that are currently in the “File Claim” state and have been waiting for a claim to be processed for more than five minutes.

```
select oid:(o.oid), timeCreated:(o.timeCreated),
       timeEntered:(o.timeEntered),
       claimDocument:(o.claimDocument),
FROM o IN claimObjects.Claim
where (((CURRENT_TIME - o.timeEntered)>=(5*1000) &&
(o.bpState AT "File Claim"))
```

Figure 18-7 A Query Showing Time-Based Filtering

Note: Because the WHERE clause of this query includes the variable CURRENT_TIME, the value of this query might change between true and false based on the passage of time alone.

Time-Based Aggregation

A time-based aggregation query is one that uses the CURRENT_TIME variable inside an aggregate function within the SELECT clause.

Example: Figure 18-8 is the LongAvgWaitingTimeByInsurance in the ClaimsProcessingSample. This query computes the average waiting time for the Claim objects that are currently in the “File Claim” state.

```
SELECT company: (o.patient.insurance.company),
       avg_waiting: AVG((CURRENT_TIME - o.timeEntered) / (1000))
  FROM o IN claimObjects.Claim
 WHERE (o.bpState AT "File Claim")
 GROUP BY company
 HAVING (avg_waiting > 5*60)
```

Figure 18-8 A Query Showing Time-Based Aggregation

Note: Because the aggregate function in the SELECT clause includes the variable CURRENT_TIME, the HAVING clause of this query will change between true and false based on the passage of time alone.

Time-Based Filtering with Aggregation

A time-based filter query with aggregation is one that uses the CURRENT_TIME variable within the WHERE clause and an aggregate function in the SELECT clause.

Example: The following code is the NumDelayedClaims from the ClaimsProcessing Sample.

```
SELECT numDelayedClaim: count(*)
  FROM o IN claimObjects.Claim
 WHERE (((CURRENT_TIME - o.timeEntered) >= (5*60*1000)) &&
       (o.bpState AT "File Claim"))
```

Limitations

Time-based queries do not work when running the Integration Server in cache-only mode. You must set the Integration Server to RDBMS data persistence.

For time-based, queries are used to trigger the computation of queries when the query condition is first violated. However, if the Integration Server is down at the time the timer is supposed to fire, the events to be generated by the timer will not be published when the Integration Server is restarted later.

FILTERING AND AGGREGATING SEQUENCE ATTRIBUTES

You can create a query to filter or aggregate a BPO attribute that is a sequence. A sequence attribute is identified by a separate binding variable called the *sequence binding variable* defined in the FROM clause to access nested sequences. See “[“FROM Clause” on page 18-4](#) for more information.

Example: A sequence query

```
Select p_name: o.product_name, s_name:  
    supplierQuote.supplier_name  
From o in quoteObjects.ProductsQuote, supplierQuote in  
    o.supplierQuotes
```

where `supplierQuote in o.supplierQuote` is a sequence binding variable that associates the variable with a sequence attribute in the BPO. A sequence binding variable has the following semantics:

- It represents a member in the sequence because it represents an individual BPO from the given BPO collection.
- The sequence binding type is the sequence element type.
- Multiple sequence binding variables result in a join across these sequences within an individual BPO.

When using a sequence binding variable, keep the following points in mind:

- The BPO must be used first in the FROM clause.
- The sequence binding variable must be dependent on the BPO binding variable.
- Only the “`*`” parameter must be used with the `count` aggregate function.

Examples

Some examples are given below to give you an idea of how to define a query to filter or aggregate BPO attributes in sequence.

Example: BPO Definition

```

module quoteObjects {
    struct Product {
        long product_id;
        string product_name;
    };
    struct SupplierQuote {
        string supplier_name;
        Addresses addresses
        float price;
        boolean available;
    };
    typedef sequence<SupplierQuote> SupplierQuotes;
    interface ProductQuote: bpe::BusinessProcessObject {
        attribute Product      product;
        attribute SupplierQuotes  supplierQuotes;
        attribute long          currentReplies;
    };
}

```

Example: Select fields from sequences of structs

```

Select
    p_name: o.product.product_name,
    s_name: supplierQuote.supplier_name
From o in quoteObjects.ProductQuote, supplierQuote in
    o.supplierQuotes

```

Example: Aggregate on sequences of structs across BPOs

```

Select
    p_name:      o.product.product_name,
    avg_price:  avg(supplierQuote.price)
From o in quoteObjects.ProductQuote, supplierQuote in
    o.supplierQuotes
Group by p_name

```

Example: Aggregate on sequences of structs across BPOs using the count function.

```

Select
    p_name: o.product.product_name,
    replies: count(*)
From o in quoteObjects.ProductQuote, supplierQuote in
    o.supplierQuotes
Where

```

```
    supplierQuote.available  
Group by p_name
```

QUERY GRAMMAR

The tables below define the subset of OQL that BusinessWare uses for SLA queries. Here are some conventions that you should keep in mind:

- The description of the query language grammar given below uses the Backus-Naur Form syntax, which describes the syntax for tokens that combine with each other to create a valid top-level production. In this case, the top-level production is the query token. For details on how to read this notation, see the tutorial at <http://braid.rad.jhu.edu/til/AboutBNF.html>
- Text in italic represents a token and is not to be typed into a query. Tokens combine to create other tokens, with a fully formed query resulting from a query token. Text in bold, fixed-width font is part of the query. Text in square brackets is optional.
- When a single or double non-alphabetic character string is meant to be part of a token, it is enclosed in double quotation marks. In the query itself, only strings and single-character constants are surrounded by double quotation marks. For example, the definition **sum** " (" *expr* ")" means the string "sum" followed by a left parenthesis, followed by whatever text is a valid *expr*, followed by a right parenthesis. The quotation marks exist only to alert you of the non-alphabetic character string.
- The keywords in the tables (represented with bold, fixed-width text) are shown using lowercase characters. However, these keywords are case-insensitive, and you can use either lowercase or uppercase letters.

PROCESS QUERY LANGUAGE
Query Grammar

Note: This rule does not apply to the attributes mentioned earlier in this chapter.

Table 18-1 Query Grammar

query Defined	<pre>query ::= select selectDeclarationList from variableDeclarationList [whereClause] select selectDeclarationList from variableDeclarationList [whereClause] [group by identifierList] [having expr] of</pre>
Explanation	<p>This is the top-level production that represents a query. All queries must conform to one of the two forms listed in the definition above. The first form represents a simple filter query, while the second form represents an aggregation query.</p> <p>In an aggregation query, for each "alias : expr" in the <i>selectDeclarationList</i>, one of the following must be true: either the alias name appears in the GROUP BY clause, or <i>expr</i> is an <i>aggregateFunction</i>. This means that if the GROUP BY clause is absent, every <i>expr</i> must be an <i>aggregateFunction</i>.</p>
Examples	<ol style="list-style-type: none"> 1. The following example is a simple filter query that returns the BPO ID, procedure date, doctor, patient's last name, patient's first name, and claim amounts for all claims greater than or equal to \$7000: <pre>select oid:(o.oid), procedureDate:(o.procedureDate), doctor:(o.doctor), lastName:(o.patient.lastName), firstName:(o.patient.firstName), claimAmount:(o.claimAmount) FROM o IN claimObjects.Claim WHERE (o.claimAmount >=7000) 0</pre> 2. The following example is an aggregation query that returns the doctor and total of all claims grouped by doctor. <pre>select doctor:(o.doctor), total:COUNT (*) FROM o in ClaimObjects.Claim GROUP BY doctor</pre>

Table 18-2 SELECT Clause

selectDeclarationList Defined	<pre>selectDeclarationList ::= alias ":" expr selectDeclarationList "," alias ":" expr</pre>
Explanation	<p><i>selectDeclarationList</i> represents the list of one or more data items to be returned in each query result. Each data item has an alias name (<i>alias</i>) and value (<i>expression</i>).</p>
Examples	<pre>oid:(o.oid), procedureDate:(o.procedureDate), doctor:(o.doctor), lastName:(o.patient.lastName), firstName:(o.patient.firstName), claimAmount:(o.claimAmount)</pre>

Table 18-3 Variable Declaration

variableDeclaration Defined	<i>variableDeclaration</i> ::= <i>identifier</i> in <i>pathExpression</i>
Explanation	This token represents the named variable (<i>identifier</i>) that represents an individual BPO from the BPO data type being queried (<i>pathExpression</i>).
Examples	o in ClaimObjects.Claim

Table 18-4 Identifier

identifier Defined	See “Explanation”
Explanation	This token represents an arbitrary name (like a variable name) for the data item you wish to refer to elsewhere.
Example	o (as in select ... from o in ClaimObjects)

Table 18-5 Alias List

identifierList Defined	<i>identifierList</i> ::= <i>identifier</i> “ ” <i>identifier</i> “,” <i>identifierList</i>
Explanation	This token occurs after the GROUP BY keyword. It lists one or more aliases by which intermediate results will be grouped.
Example	doctor in (select doctor:(o.doctor), claimAmount:SUM(o.claimAmount))

Table 18-6 Path Expression

pathExpression Defined	<i>pathExpression</i> ::= <i>identifier</i> <i>pathIdentifier</i> “.” <i>identifier</i>
Explanation	This token refers to a BPO data type or data within that data type. The dot operator (“.”) is used to separate the module name from the BPO name or to extract an element from within a BPO.
Examples	o in ClaimObjects.Claim o.patient.firstName

PROCESS QUERY LANGUAGE
Query Grammar

Table 18-7 Where Clause

whereClause Defined	<i>whereClause</i> ::= where <i>expr</i>
Explanation	This token expresses a condition that limits the results the query returns. The <i>expr</i> token must evaluate to either true or false.
Example	where o.claimAmount > 5000 WHERE (o.doctor = doctor_param)

Table 18-8 Aggregate Function

aggregateFunction Defined	<i>aggregateFunction</i> ::= sum "(" <i>expr</i> ")" avg "(" <i>expr</i> ")" count "(" <i>expr</i> ")" count (*)
Explanation	This token returns a numeric value based on a series of BPO records that have been grouped together. When the argument of the function is <i>expr</i> , the function works on the named expression. When the function is “count (*)”, this function returns the number of BPO records that have been grouped together.
Example	The following query returns, for each doctor, the total of all the claims amounts. SELECT doctor:(o.doctor), claimAmount:SUM(o.claimAmount) FROM o IN claimObjects.Claim GROUP BY doctor

Table 18-9 Quantifiers

quantifier Defined	<i>quantifierClause</i> ::= (forall exists) <i>identifier</i> in <i>pathExpression</i> : <i>expr</i>
Explanation	This token expresses a condition that limits the results the query returns. The <i>pathExpression</i> should be a collection. An existential quantifier “exists” clause is true if and only if <i>at least</i> one element in the collection specified by <i>pathExpression</i> satisfies the condition specified by <i>expr</i> . An universal quantifier “forall” clause is true if and only if <i>all</i> the elements in the collection satisfies the condition specified by <i>expr</i> .
Example	<pre>SELECT oid:(o.oid), procedureDate:(o.procedureDate), doctor:(o.doctor), lastName:(o.patient.lastName), firstName:(o.patient.firstName), claimAmount:(o.claimAmount), claimDocument:(o.claimDocument) FROM o IN claimObjects.Claim WHERE exists p in o.procedures: p = "5590" OR p = "6099" SELECT oid:(o.oid), procedureDate:(o.procedureDate), doctor:(o.doctor), lastName:(o.patient.lastName), firstName:(o.patient.firstName), claimAmount:(o.claimAmount), claimDocument:(o.claimDocument) FROM o IN claimObjects.Claim WHERE forall p in o.procedures: p = "5590" OR p = "6099"</pre>

Table 18-10 Literal

literal Defined	<i>literal</i> ::= <i>integerConstant</i> <i>characterConstant</i> <i>floatConstant</i> <i>doubleConstant</i> <i>stringConstant</i> current_time true false
Explanation	This token represents some kind of unchanging value. A <i>characterConstant</i> is a single character, surrounded by double quotes. A <i>stringConstant</i> is a series of characters, surrounded by double quotes. The last three possibilities, current_time , true , and false , are not tokens but are items to be entered, as-is, into the query. They represent the current time, a Boolean true value, and a Boolean false value.
Examples	12345 (integer), 'q' (character), 123.45 (float), 123.45F (float), 123.45D (double), "navy van" (string), current_time, true, false

Table 18-11 Expr

expr Defined	$\begin{aligned} expr ::= & \text{literal} \mid \\ & \text{pathExpression} \mid \\ & \text{valueFunction} \mid \\ & "("\text{expr}") \mid \\ & "+"\text{expr} \mid \\ & "-"\text{expr} \mid \\ & expr "+"expr \mid \\ & expr "-"expr \mid \\ & expr "*"expr \mid \\ & expr "/"expr \mid \\ & expr "%"expr \mid \\ & "!"\text{expr} \mid \\ & \text{not}\text{expr} \mid \\ & expr \text{and}\text{expr} \mid \\ & expr \text{or}\text{expr} \mid \\ & expr "="\text{expr} \mid \\ & expr "!="\text{expr} \mid \\ & expr "<"\text{expr} \mid \\ & expr ">"\text{expr} \mid \\ & expr "<="\text{expr} \mid \\ & expr ">="\text{expr} \mid \\ & expr \text{like}\text{expr} \mid \\ & expr \text{at}\text{expr} \mid \\ & expr \text{in}\text{expr} \end{aligned}$
Explanation	<p>This token defines how literals, path expressions, and value functions can be combined. Most of these combinations are obvious, but several require some explanation. The "%" operator computes remainders; for example, $7 \% 3 = 1$. In the case of logical operators (for example, and and not), the argument or arguments to these operators must be logical, not numeric or string, values. The like, at, and in operators are explained in “Notes on Selected Operators and Attributes” on page 18-6.</p>
Examples	<pre>(a.balance * 1.17) !(a.bpState at "Wait_for_signal") totalBal - (a.balance / 2.0)</pre>

This chapter provides information on creating *process views* in the BME. Process views define information to be displayed in Business Cockpit after deploying and running a BusinessWare project. Business Cockpit is a web application that runs inside a web-enabled Integration Server. For more information on Business Cockpit, see the *Business Cockpit Guide*.

Topics Include:

- [Overview](#)
- [Creating Process Views](#)
- [Configuring Process View Properties](#)
- [Configuring Display Properties](#)
- [Rendering Views](#)
- [Using Single Sign-On](#)
- [Validating Views](#)
- [Deploying Views](#)
- [Log Files](#)

OVERVIEW

Process views display data generated by process models using queries defined in process query models. For information on how the views are generated, see “[Rendering Views](#)” on page 19-21.

There are two types of process views:

- **Snapshot**—Snapshot process views display data retrieved by queries in a snapshot process query model. Cockpit runs the queries against a database to retrieve BPO data persisted by a process component. Snapshot views must be created with process components that use a database.

Snapshot views are best for reporting process data when the queries are run on demand rather than continuously.

- **Real-time**—Real-time process views display data sent to a channel by a process query component. Real-time process query models are used by process query components to monitor process models as BPOs transition from one state to another. Cockpit subscribes to the channel to which the process query component outputs the query results.

Real-time views are best for monitoring process data when queries are computed continuously by *Process Analyzer*.

The *Claims Processing Sample* provides examples of snapshot and real-time process query models and their associated views.

For more information on process queries, see “[Process Query Models](#)” on page 17-1.

CREATING PROCESS VIEWS

To create a process view, you must do the following:

- Create a process query within a process query model.
- Select or create a folder to contain the process view.
- Configure the following view properties:
 - Process Query Model Link
 - Query
 - Integration Model Link
 - Component
- Add and configure attribute properties.
- Configure presentation properties.
- Save the process view
- Validate the process view

CREATING A PROCESS QUERY

To create a process view, first you must create a process query inside a process query model. The query you create determines the data that populates the tables and charts in Cockpit. For more information on process queries, see [Chapter 17, “Process Query Models.”](#)

SELECTING A FOLDER

You can select an existing folder from the Explorer or create a new folder to contain the process views. Folder names are visible to the end user. With the exception of the underscore “_”, folder names cannot contain spaces, other punctuation, or internationalized characters. The process query name, and the folder that contains the process query model must follow these same guidelines. The Cockpit user will see all folders that contain process views from all projects in the directory server. Therefore, folders should be uniquely named, easily identifiable, and meaningful to the Cockpit user.

IMPORTANT: Even though you can create views at the root level and in nested folders, only the views directly inside a top-level folder are displayed by Cockpit.

Organizing Views

When selecting or creating folders for process views, you should keep in mind security issues for those views. For example, if you have a number of views that provide process data for hospital billing and you want only hospital administrators to have access to those views from Cockpit, then you should group those views together in a folder named “billing.” When the views are deployed to a directory server, the directory server administrator can easily set the appropriate access control on that folder.

For more information see:

- **Access control**—*BusinessWare Administration Guide*
- **Deployment**—[Chapter 22, “Deploying Projects”](#)
- **Process view location**—“[Directory Server Namespace](#)” on page 19-30
- **Creating folders**—*BME Help*

CREATING A NEW VIEW

To create a new process view:

1. Select a process view folder from the Explorer.
2. From the **File** menu, click **New....** You can also right-click on a process view folder in the Explorer and click **New > ProcessView**.
3. In the **New Wizard**, select **ProcessView** and click **Next**.
4. In the **Name** box, enter a name for the new process view.

5. Click **Finish**.

Note: View names must be unique inside of a folder.

DELETING VIEWS

To delete a process view:

1. Right-click on the process view and click **Delete**.
2. In the **Confirm Object Deletion** dialog, click **Yes**.

IMPORTANT: If you delete a folder containing process views, all views in the folder are deleted also.

RENAMING VIEWS

After you have created a view and named it, you can rename the view at anytime.

To rename a process view:

1. Right-click on a view.
2. Click **Rename**.
3. In the New Name text box, enter a new name and click **OK**.

IMPORTANT: If you rename a view referenced as a drilldown view, you must reconfigure the Drilldown View property in the parent view. If you do not reconfigure the Drilldown View property, the parent process view will have a broken link. See “[Drilldown View](#)” on page 19-7 for more information.

COPYING VIEWS

You can copy and paste views to the current folder or to a different folder.

To copy a process view:

1. Right-click on a view.
2. Click **Copy**.
3. Right-click on a folder to contain the copied view.
4. Click **Paste > Copy**.

CONFIGURING PROCESS VIEW PROPERTIES

Process view properties appear in the Properties tab of the Explorer when you select a process view. You can also view process view properties by right-clicking on a process view and selecting Properties. For all process views, the following properties must be configured:

- Process Query Model Link
- Query
- Integration Model Link
- Component

Configuring the following properties is optional:

- Drilldown View
- Resource File
- Top Level Visible
- Description
- View Permission

PROCESS QUERY MODEL LINK

The Process Query Model Link property specifies the process query model that contains the process query that the process view uses. Process query models can be real-time or snapshot models. The type of model you select determines the type of component you must select. The component determines the data source. For more information on process query models, see “[Process Query Models](#)” on page 17-1.

For example, in [Figure 19-1](#) the process view Procedures by Patient in the *Claims Processing Sample* is a snapshot process view that displays data from a query in the process query model SnapshotQueryModel. Queries in SnapshotQueryModel retrieve data persisted by the process component Claims Process.

To configure this property, type the name of the process query model link in the text box, or click the browse button. If you click the browse button, select a process query model and click **OK**.

PROCESS VIEWS

Configuring Process View Properties

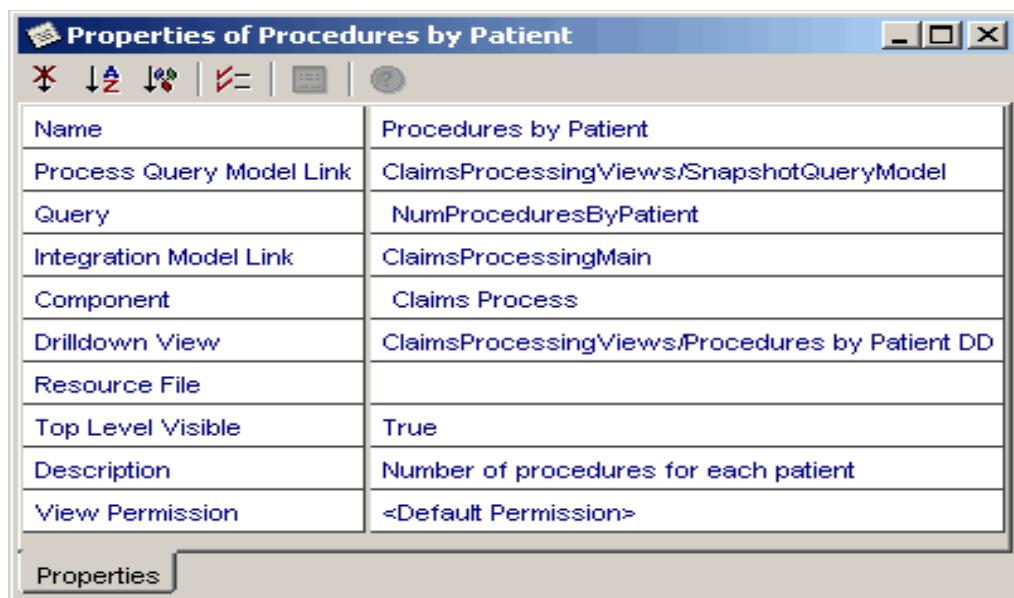


Figure 19-1 Procedures by Patient Properties

QUERY

This is the process query that the new process view uses to gather BPO data. To configure this property, type the name of the query in the text box, or click the browse button. If you click the browse button, all process queries inside the process query model selected in Process Query Model Link are displayed. Select a process query and click **OK**.

INTEGRATION MODEL LINK

Select the integration model that contains the component that generates the data for the view. The component can be a process component or a process query component. See "[Overview](#)" on page 19-1 for more information on process components and process query components. To configure this property, click the browse button to see all integration models in the project. Select an integration model and click **OK**. The browse button will not work if the Process Query Model Link property is not configured with a valid value.

COMPONENT

Select a process component for a snapshot process query model or a process query component for a real-time process query model. To configure this property, type the name of the component in the text box, or click the browse button. If you click the browse button, all components in the project are displayed. Select a component and click **OK**.

The type of component you select depends on the [Process Query Model Link](#) you specify. See “[Overview](#)” on page 19-1 for more information.

DRILLDOWN VIEW

Drilldown functionality is possible from enhanced views for both snapshot and real-time process views. A drilldown view is a parameterized process view that provides more detailed information than the parent process view. You must first create the process view that will act as a drilldown view and then link to it.

Drilldown views display data defined by parameterized process queries that are run against a database. When a Cockpit user clicks on a data point in a process view and the drilldown view is displayed, the required parameter data is supplied based on the data point where the user clicks. Any view referencing a drilldown view must provide all parameters for that drilldown view. For example, if a drilldown view is linked to a process query with a parameter called `doctor_name_param`, the parent view must have an attribute called `doctor_name`. When a Cockpit user clicks on a bar graph labeled Dr. Smith in a process view, the parameter value “Dr. Smith” is supplied and the parameterized drilldown view is displayed.

You can also create drilldown views across different BPO types. For example, if you have two process query models, `TaskProcessQueryModel` for tasks and `SONQueryModel` for service order numbers, a process view that is based on a process query in `TaskProcessQueryModel` can drilldown to another process view that is based on a different process query in `SONQueryModel`.

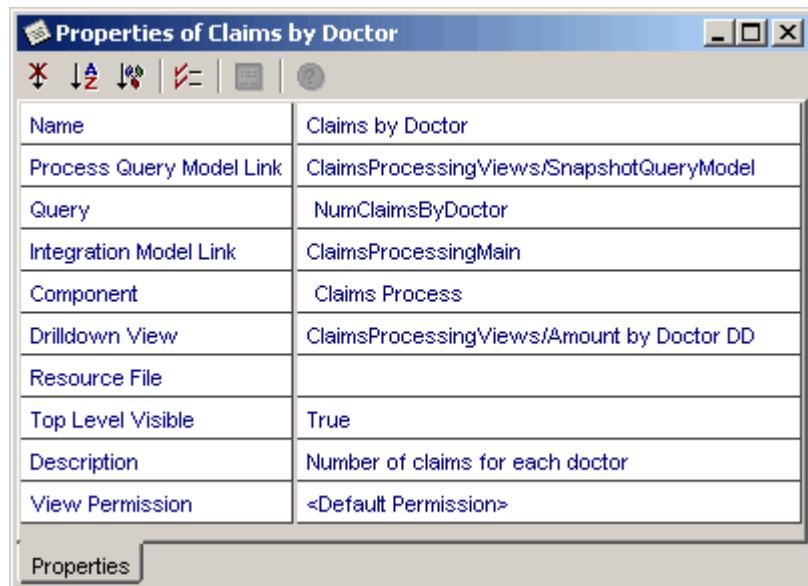
Note: Multiple process views can link to the same drilldown view. You can only have one level of drilldown for a process view. You cannot drilldown from a drilldown view. For an example of how multiple views can share a drilldown view, observe how the Claims by Doctor and Amount by Doctors views both drill down to the Amount by Doctor DD view in the Claims Processing Sample.

To link to a drilldown view, type the name of the process view or click the browse button. If you click the browse button, all process views in the project are displayed. Select a drilldown process view and click **OK**.

PROCESS VIEWS

Configuring Process View Properties

Figure 19-2 shows the Properties Window for a process view with a drilldown link to a view called Amount by Doctor DD. Note that the four required properties, Process Query Model Link, Query, Integration Model, and Component have been configured.



The screenshot shows the 'Properties of Claims by Doctor' dialog box. It contains a grid of properties and their values:

Name	Claims by Doctor
Process Query Model Link	ClaimsProcessingViews/SnapshotQueryModel
Query	NumClaimsByDoctor
Integration Model Link	ClaimsProcessingMain
Component	Claims Process
Drilldown View	ClaimsProcessingViews/Amount by Doctor DD
Resource File	
Top Level Visible	True
Description	Number of claims for each doctor
View Permission	<Default Permission>

Figure 19-2 Process View Properties and Drilldown View Link

RESOURCE FILE

The resource file is a Java resource bundle used to map an attribute value to a customized display value. For example, an attribute named “state” has the value “CA.” To make this value more readable, you can remap “CA” to “California.” To do this, you must add the following line to a resource file:

```
state.CA=California
```

To create a resource file:

1. Create a text file.
2. Rename this file with the extension “.properties”. For example:
`ProcedureCode.properties`
3. Place this file in the project directory.
4. Refresh the project directory in the Explorer. The resource file is listed in the project directory.

5. To edit the file, select it and right-click. Select **Open**.

Note: Properties files must use an ISO 8859-1 encoding or contain Unicode escape sequences to represent Unicode characters. Unicode escape sequences are specified as \uxxxxxx, where xxxxxx represents the hexadecimal value of the specific character.

Java provides a utility called native2ascii, which takes a file written in a given character encoding and converts non-ASCII characters to the appropriate Unicode escape sequences. The native2ascii utility also takes an encoding option with which you can specify the character encoding of the file to be converted. If this option is not specified, it uses the default character encoding of the platform on which the program is running. The command-line option -reverse allows you to convert a file containing Unicode escape sequences to a native character encoding. You must specify an input file and you can also specify an optional output file where the converted file will be written. If the output file is not specified, standard output is used:

```
C:\> native2ascii -encoding SJIS custom.ja
customizations_ja.properties
```

TOP LEVEL VISIBLE

This property specifies if a process view is visible to the Cockpit user in the drop-down list of the Homepage. For example, you can configure a view to be inaccessible to the Cockpit-user.

Note: Parameterized views (process views that are based on a parameterized query) are never accessible by the Cockpit user in the Homepage. Cockpit uses the parameterized view for drill-down views.

DESCRIPTION

The description is displayed in the Cockpit Select Views list to help a Cockpit user determine which views in a folder to add to the My Views list. [Figure 19-3](#) shows an example of the Cockpit Views Available list with five views in the ClaimsProcessingViews folder and the view descriptions.

PROCESS VIEWS

Configuring Display Properties

Select Order	View Path	View Description
<input type="checkbox"/>	1 ▾ ClaimsProcessingViews/Processing Time Long	Claims that take a long time to process
<input type="checkbox"/>	2 ▾ ClaimsProcessingViews/Claims by Status (real-time)	Number of claims that are in different processing state using real-time query
<input type="checkbox"/>	3 ▾ ClaimsProcessingViews/Processing Time by Insurance	Average processing time for each insurance company
<input type="checkbox"/>	4 ▾ ClaimsProcessingViews/Claims by Status	Number of claims that are in different processing state
<input type="checkbox"/>	5 ▾ ClaimsProcessingViews/Amount by Doctor	Total value of claims for each doctor
<input type="checkbox"/>	6 ▾ ClaimsProcessingViews/Procedures by Patient	Number of procedures for each patient
<input type="checkbox"/>	7 ▾ ClaimsProcessingViews/Claims by Doctor Insurance	Number of claims for each doctor separated by insurance company

Figure 19-3 Views Available Table with View Descriptions

VIEW PERMISSION

Setting the View Permission property specifies the list of roles allowed to view the Process View. Leaving this property unchecked specifies that no authorization checks need to be completed. The default value matches the value of the project object's default permission. For more information on security, see the *BusinessWare Security Guide*.

CONFIGURING DISPLAY PROPERTIES

Display properties specify how data is displayed in the process views. For charts that support style sheets (all charts except table and time series support style sheets), it is recommended that you use style sheet properties to control the chart presentation. See “[Using Style Sheets](#)” on page 19-22 for more information on style sheets.

Display properties are divided into three categories:

- [General Attribute Properties](#)
- [Expert Attribute Properties](#)
- [Chart Properties](#)

When a Data Object (DO) or Struct is a BPO field, BusinessWare expands all fields in the DO or Struct as BPO fields. For example, if an “Order” BPO has a “Customer” DO and the “Customer” DO has a field labeled “Name,” “Customer.Name” appears as one of the attributes that you can select when you create a process view. The order of the attributes in the attributes list determines the order of the attributes displayed in the table chart type.

IMPORTANT: Persisting BPOs and/or DOs as serialized BLOBS will prevent you from developing process queries and views against those objects. Although Query Editor and View Editor allow you to define process queries and views against a serialized BPO or DO, Business Cockpit will mark those process views as invalid views at run-time. When the serialized DO is turned on, those DOs can only be queried by non-time-based filter queries. Other queries (such as aggregation or time-based queries) cannot query on those DOs.

GENERAL ATTRIBUTE PROPERTIES

The following attribute properties apply to all views:

- [Name](#)
- [Label](#)
- [Type](#)
- [Color](#)

PROCESS VIEWS

Configuring Display Properties

Figure 19-4 shows the general attribute properties in the editor window for a process view titled “Amount by Doctor.”

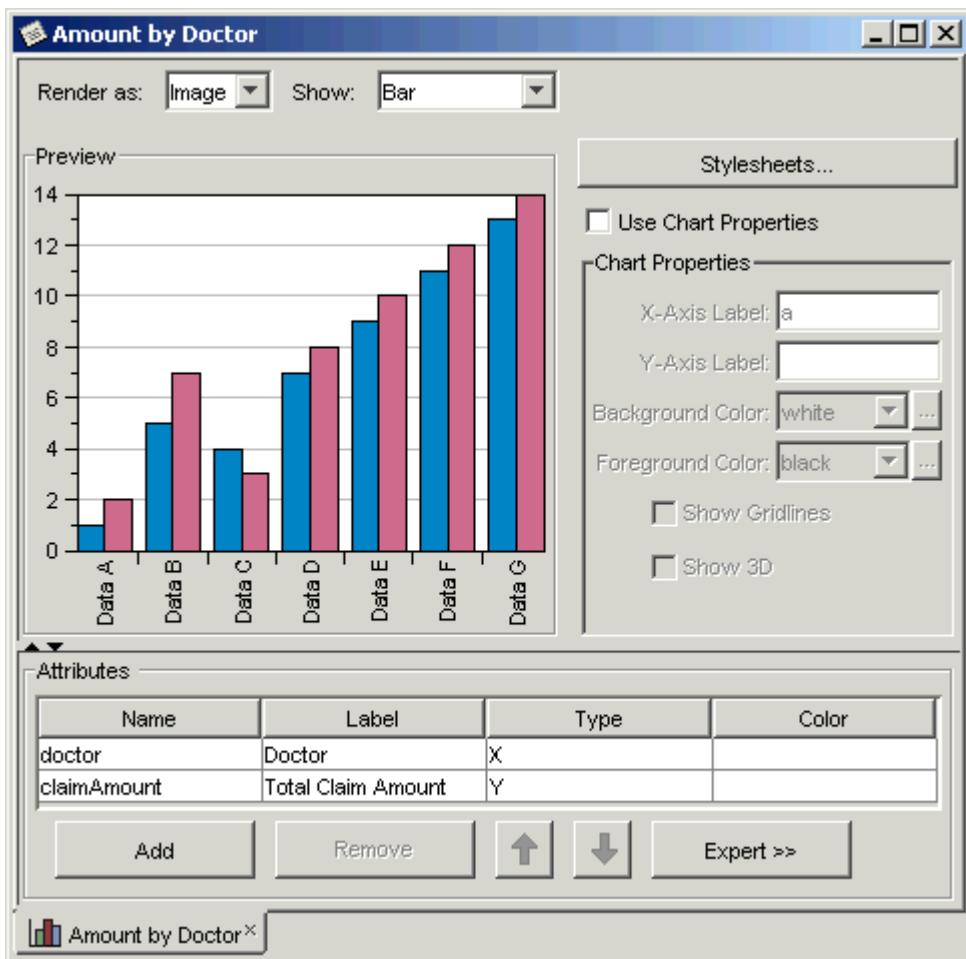


Figure 19-4 Chart Properties and General Attribute Properties

Name

The attribute name is the alias for an element in a query's select clause. After you add or change a query in a query model or create a new query model, you must save the query model before you can create a view using that query. When the query model has been saved, a list of available attributes appears in the drop-down list in the Name column. Select the attribute name for which you want to create a process view.

Label

The label is a user-friendly name for an attribute. It is displayed as a column heading in a table chart. The default label is empty, but Cockpit displays the attribute name when the label is empty. For example, in [Figure 19-4](#), if the Label property was empty, Cockpit would display “claimAmount” as the label.

Type

Attributes that belong to the “group-by” clause are automatically selected for the X axis. For these attributes, you can also select Aggregated X to create stacking charts. All other attributes of numeric data type are selected for the Y axis. For data that can only be displayed in a table, the attributes are of type “none.”

Color

Graph colors can be configured for attributes that represent a data series. For all charts except tables, colors are only applied to data series of type y attributes. The following preset colors are available: white, light gray, gray, dark gray, black, red, pink, orange, yellow, green, magenta, cyan, and blue. In addition, you can select custom RGB colors by clicking the browse button in the color column.

EXPERT ATTRIBUTE PROPERTIES

Click on **Expert** at the bottom of the process view editor to access additional attribute properties. The following properties can be configured:

- [Date/Time Format](#)
- [Numeric Format](#)
- [Resource Prefix](#)
- [Model State Filters](#)
- [XSLT Renderer](#)

[Figure 19-5](#) shows the general and expert presentation properties for two attributes named “doctor” and “claimAmount.”

Attributes								
Name	Label	Type	Color	Date ...	Numeric ...	Resource ...	Model ...	XSLT ...
doctor	Doctor	X						
claimAmount	Total Claim ...	Y			\$49.99			
Add	Remove			Expert <<				

Figure 19-5 General and Expert Attribute Properties

Date/Time Format

Date and time formats can be specified using standard JAVA SimpleDateFormat patterns. Supported formats are listed in [Table 19-1](#).

Table 19-1 Date and Time Formats

Format	Sample Output
time,short	3:30 PM
time,medium	3:30:12 PM
time,long	3:30:12 PM PDT
date,short	01/01/02
date,medium	Jan 01, 2002
date,long	January 01, 2002
date,full	Tuesday, January 01, 2002

You can enter date and time information using custom formats, also. [Table 19-2](#) provides two examples of custom formats.

Table 19-2 Custom Date/Time Formats

Format	Sample Output
date,MM/DD/YYYY 'at' H:mm:ss z	07/24/2002 at 16:15:18 PDT
date,M/D/YY h:mma	7/24/02 4:15PM

Numeric Format

Numeric formats can be specified using standard JAVA MessageFormat patterns. Supported formats are listed in [Table 19-3](#).

Table 19-3 Numeric Value Formats

Format	Sample Value	Sample Output
50 (integer)	34.95243	35
50% (percent)	0.95	95%
\$49.99 (currency)	1.95	\$1.95

You can customize numeric formats, also. For example, [Table 19-4](#) lists possible formats and output for the following input: Double(1234.567).

Table 19-4 Custom Numeric Value Formats

Format	Sample Output
number,currency	\$1,234.57
number,integer	1,234
number,percent	123,456%
number,#.00	1234.57
number,###,###.0000	1,234.5670

Resource Prefix

The resource prefix is prepended to the runtime value to find the key of the mapping rule in the resource file. For example, if there is an attribute for the customer's state, and the resource prefix is set to "state" then the value "MA" will be displayed as "Massachusetts" by looking up "state.MA" in the resource file. See "["Resource File"](#) on page 19-8" for more information on the resource file.

Model State Filters

For a bpstate attribute, this property specifies the top-level states in the process model that are displayed in the process view. By default, all states are displayed.

Figure 19-6 shows the Model State Filters window. Select the states to show from the available states list and order them in the selected states list by clicking the up and down arrows. Click **OK** when you are finished. Cockpit displays data in charts based on the order of the bpstates specified in this window.

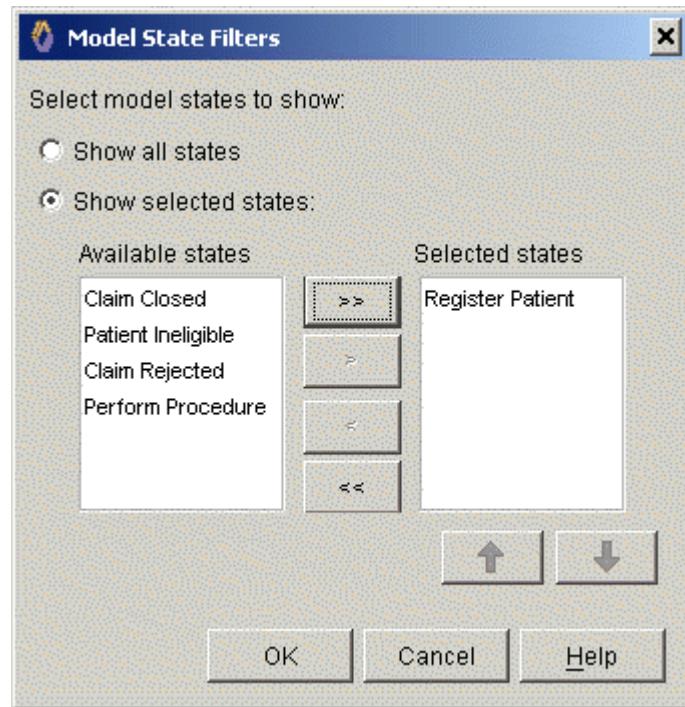


Figure 19-6 Select Model State Filters

XSLT Renderer

This property specifies the XSL file that is used to transform the data that is in XML format. The XSL file must be in the same project that contains the process views that use it. [Figure 19-7](#) shows the location in the Claims Processing project of the XSL file used in the Show Large Claim view.

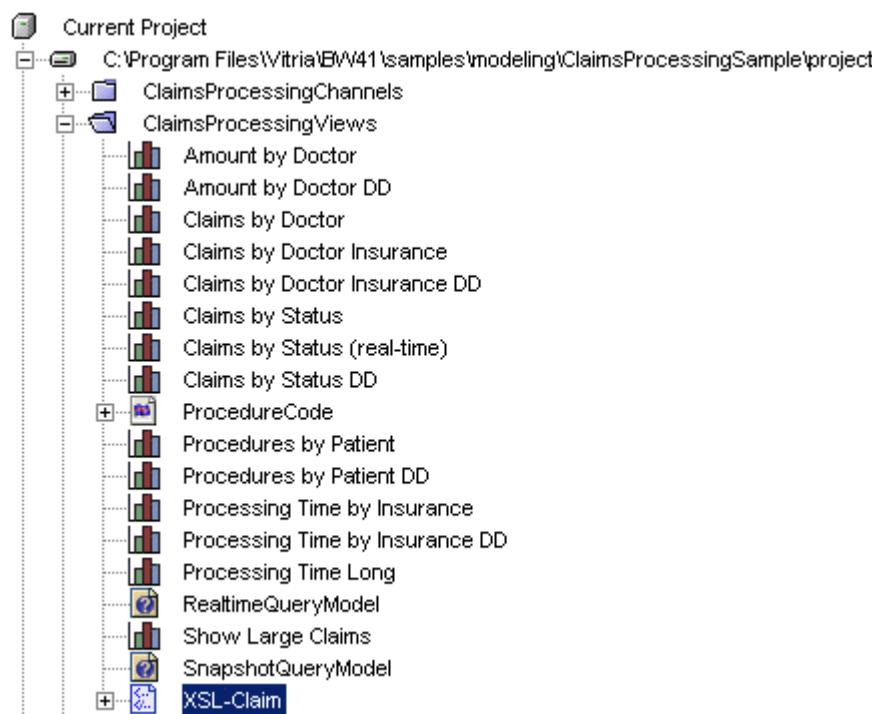


Figure 19-7 XSL File Location in the Claims Processing Sample

To add an XSL file to the project, you can mount the file directory in the project, you can copy the file into a project folder, or you can link to the XSL file by entering a URL. For example:

`http://mymachine/xsl/mytransformation.xsl`

To select a style sheet:

1. Click the browse button in the XSLT Renderer box for a specific attribute.
2. Navigate to the XSL file.

PROCESS VIEWS

Configuring Display Properties

3. Select the file and click **OK**.

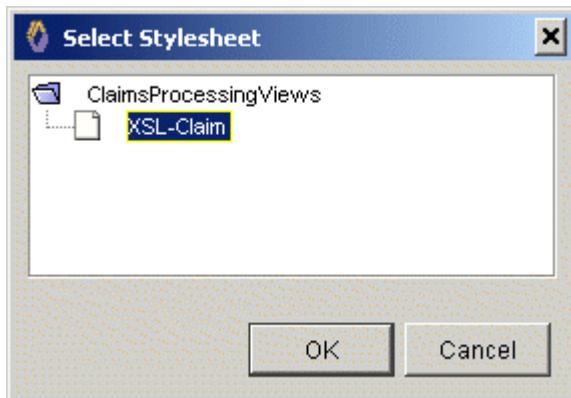


Figure 19-8 Select Style Sheet for XSLT Renderer

CHART PROPERTIES

In addition to the general and expert attribute properties, there is a set of chart properties specific to chart views. These properties are ignored when a table view is specified. See [Figure 19-4](#) for an example of the chart properties for a process view titled “Amount by Doctor.”

[Table 19-5](#) lists the process view chart properties.

Table 19-5 Process View Chart Properties

Property	Description	Default
Renderer Type (Render as)	Specify Image for the Cockpit server to generate the chart image or HTML table needed by a Web browser to display the process views. Specify Applet for the Cockpit Server to generate an applet tag which prompts a Web browser to load an applet.	Image

Table 19-5 Process View Chart Properties (Continued)

Property	Description	Default
Chart Type (Show)	<p>Specifies the default chart type that appears in the Cockpit homepage. Available chart types include:</p> <ul style="list-style-type: none"> • Bar Chart • Stacking Bar Chart • Plot Chart • Stacking Plot Chart • Area Chart • Stacking Area Chart • Bubble Chart • Hi Lo • Dial Chart • Pie Chart • Radar Chart • Scatter Chart • Stair Chart • Table <p>Note: You should select Table if the view is a drilldown view or if any of the attributes are non-numeric.</p> <p>The following chart types require two or more data series:</p> <ul style="list-style-type: none"> • All stacking charts • Bubble • Hi Lo 	Bar Chart
Use Chart Properties	Check this box to use the chart options below in addition to the stylesheet properties.	False
X-Axis Label	Specifies the label applied to the attribute selected for the X axis. See “ Type ” on page 19-13 for more information about specifying X axis attributes.	No Default
Y-Axis Label	Specifies the label applied to the attribute selected for the Y axis. See “ Type ” on page 19-13 for more information about specifying Y axis attributes.	No Default
Background Color	Specifies the color that appears behind the data points in the chart.	White

Table 19-5 Process View Chart Properties (Continued)

Property	Description	Default
Foreground Color	Specifies the color of the data points in the chart.	Black
Show Gridlines	Specifies if gridlines should be applied to the chart.	False
Show 3D	Specifies if data points should be displayed with a three dimensional aspect applied.	False

CLOB SUPPORT

Business Cockpit provides support for the Character Large Object (CLOB) data type when the Table chart type is selected. See the *BusinessWare Installation Guide* for a list of supported databases.

Table cells containing CLOB data provide a “Data” hyperlink. Clicking this link displays the data in a new popup window. If an XSLT file is specified, it is used to transform the data before it is displayed.

CLOB is only supported in snapshot views. Your query must include an OID in order to see CLOB data. For an example of a view that demonstrates CLOB support, see the Show Large Claims view in the *Claims Processing Sample*.

IMPORTANT: If the process query contains a CLOB attribute, the order of the attributes in your process view must match the order of the attributes in the process query’s select clause. See “[Expert Attribute Properties](#)” on [page 19-13](#) for more information.

DISPLAYING AN XML FILE

XML data can be displayed by clicking on an XML attribute in a table view. The XML attribute can contain an XML document or a link to an XML document. If the [XSLT Renderer](#) property has been configured, an XSL file is used to format the XML file before it is displayed. For more information on the XSLT Renderer, see “[XSLT Renderer](#)” on [page 19-17](#).

For an example of a view that displays XML content transformed by an XSL file, see the Show Large Claims view in the *Claims Processing Sample*.

RENDERING VIEWS

Process views are generated in two steps:

1. Create and transform the dataset using a process view handler that acquires and transforms process data determined by the process query specified in “Query” on page 19-6.
2. The renderer presents the data in a process view in either a chart image or a table. Business Cockpit supports server-side rendering and client-side rendering.
 - **Server-side rendering**—the Cockpit server generates the chart image or HTML table needed by a Web browser to display the process views.
 - **Client-side rendering**—the Cockpit Server generates an applet tag which prompts a Web browser to load an applet. The applet may update the display at a specified interval by retrieving data in an XML format from the Cockpit server. The applet uses a charting engine to display the data in the specified chart format. Client-side rendering is best when real-time data must be displayed.

When configuring chart properties, you should first specify:

- Renderer Type (Render as)—image for server-side rendering or applet for client-side rendering.
- Show—the default chart type. See [Table 19-5](#) for a list of chart types.

As described in “[Chart Properties](#)” on page 19-18, data from process views is rendered into specific chart types. Cockpit users only see data in charts that you specify during process view design.

ADDING CUSTOM RENDERERS

By default, Business Cockpit provides a renderer that uses ILOG JVViews for server-side rendering. Business Cockpit provides an applet that uses ILOG JVViews for client-side rendering. It is possible to create a custom renderer to support chart types not currently available by default. Available chart types are determined by the specified renderer.

A custom renderer and its supported chart types must be defined in the following file:

installdir/web/common/webapps/web-inf/cockpit-renderer-config.xml

Renderer attributes include:

- Name

- Label
- Classname

To add a custom renderer:

1. Create a new Java class that implements the Renderer interface.
2. Add a new renderer entry to cockpit-renderer-config.xml.
3. Add the new renderer under the list of chart types that the renderer supports.

USING STYLE SHEETS

Renderers use style sheets to configure chart and data display properties. A style sheet can be applied to one or more views. ILOG JVViews uses a cascading style sheet (CSS). You can modify a style sheet to customize chart properties for your views. It is important that you edit your style sheet according to the format required by the specified renderer. Style sheets are deployed with the BusinessWare project.

Business Cockpit includes the following style sheets for the default chart types:

- area.css
- bar.css
- base.css
- bubble.css
- dial.css
- hilo.css
- pie.css
- plot.css
- radar.css
- scatter.css
- stair.css
- stacking-area.css
- stacking-barr.css
- stacking-plot.css

The style sheets are installed in the following location:

installdir/web/common/webapps/cockpit/classes/chart_template

The chart style sheets all extend a base style sheet named base.css. The base style sheet defines properties common to all of the chart style sheets. You can override the properties in the base style sheet by defining them in one of the chart style sheets.

Common attributes that you can modify are listed in the style sheets. You can specify properties for a whole series or for individual data points.

The following code examples are style sheets used by ILOG JVViews. The first section of each example shows how to style the chart component, while the second section shows how to style the chart data.

```
//-----
-----  
-- Default Bar Chart stylesheet.  
//-----  
-----  
// import the common base properties for all chart types  
@import "base.css";  
//===== Component Styles ======  
chart {  
    type          : CARTESIAN;           // CARTESIAN, PIE,  
    POLAR, RADAR  
    renderingType : BAR;                // BAR,  
    STACKED_BAR, SUPERIMPOSED_BAR  
}  
chartRenderer {  
    //widthPercent : 40;  
}  
===== Data Styles =====  
//point[y>7] {color1: red;}
```

The following code is from the base stylesheet.

```
//-----  
-----  
-- Base chart stylesheet that contains common properties  
for all chart types.  
//-----  
-----  
//===== Component Styles ======  
chart {  
    legendVisible   : true;  
    legendPosition  : EAST;  
    opaque          : true;  
    foreground      : black;
```

PROCESS VIEWS

Rendering Views

```
background      : white;
font           : 'dialog,plain,10';
// interactors   : "InfoView,Zoom,Pan,Pick>EditPoint";
scalingFont    : true;
antiAliasing   : true;
antiAliasingText: true;
//border         : @#chartBorder;
highlighting   : true;
//optimizedRepaint: false;
//xValuesSorted : true;
}
chartRenderer {
    widthPercent   : 70;
}
chartArea {
    //plotStyle     : @#plotStyle;
    //border        : @#emptyBorder;
}
#xScale {
    category       : true;
    labelRotation  : 90;
}
#yScale {
    tickLayout     : TICK_OUTSIDE;
    titleRotation  : 270;
}
#yScale axis {
    dataMin        : 0;
}
#xGrid {
    visible        : false;
}
#yGrid {
    visible        : false;
}
#chartBorder {
    class          : 'javax.swing.border.LineBorder(lineColor,
thickness)';
    lineColor      : black;
    thickness      : 2;
}
chartLegend {
    border         : @#legendBorder;
    //background    : white;
    opaque         : true
}
#legendBorder {
```

```

        class           :
'javax.swing.border.CompoundBorder(outsideBorder,
insideBorder)';
    outsideBorder   : @#outBorder;
    insideBorder    : @#inBorder;
}
#outBorder {
    //class           :
'ilog.views.chart.IlvLegendSeparator(thickness)';
    thickness       : 0;
}
#inBorder {
    class           :
'javax.swing.border.EmptyBorder(borderInsets)';
    borderInsets    : 4,0,4,0;
}
#emptyBorder {
    class           :
'javax.swing.border.EmptyBorder(borderInsets)';
    borderInsets: 10,10,10,10;
}
#plotStyle {
    class           : 'ilog.views.chart.IlvStyle(strokePaint,
fillPaint)';
    strokePaint    : black;
    fillPaint      : 'white/wheat/white';
}
//===== Data Styles =====
// Set gradients for each series.
series[index=0] {
    color1: "#668fcd";
}
series[index=1] {
    color1: "#839a6e";
}
series[index=2] {
    color1: "#2fb9a9";
}
series[index=3] {
    color1: "#cc6601";
}
series[index=4] {
    color1: "#f9c97d";
}
series[index=5] {
    color1: "#00b600";
}
series[index=6] {
    color1: "#81007f";
}

```

```
        }
        series[index=7] {
            color1: "#755edd";
        }
        series[index=8] {
            color1: "#f67400";
        }
        series[index=9] {
            color1: "#e90078";
        }
        /*
        series[index=0] {
            color1: "#98bdff-#476aa9";
            color2: "#476aa9";
        }
        series[index=1] {
            color1: "#78ff78-#249224";
            color2: "#249224";
        }
        series[index=2] {
            color1: "#ff9d9d-#924242";
            color2: "#924242";
        }
        series[index=3] {
            color1: "#4dfcff-#009395";
            color2: "#009395";
        }
        */
        // Special gradient for highlighted series.
        series:highlighted {
            color1: #ffceff-#9e729e;
            color2: #dda0dd;
        }
```

Note: How `color1` and `color2` are applied depends on the type of renderer that displays the data. For a filled renderer (such as bar), `color1` specifies the fill color and `color2` specifies the border color. For a renderer that is not filled (such as plot), `color1` is the stroke color and `color2` is not used.

[Table 19-6](#) lists the CSS elements defined in the base style sheet to reference the different parts of a chart:

Table 19-6 CSS Elements

Element Type	Description
chart	chart component
chartArea	chart area component
chartLegend	chart legend
chart3Dview	chart 3-D view
chartRenderer	chart renderers
chartScale	chart scales
chartGrid	chart grids

Elements are used to modify the Bean properties of the corresponding target class. For example, the following code shows you how to specify chart and chart area colors by defining element type properties.

```

chart {
    foreground      : black;
    background      : yellow;
    opaque          : true;
}
chartArea
    plotBackground   : oldlace;
}

```

[Table 19-7](#) lists properties that are used to customize the rendering of data series and data points.

Table 19-7 Properties for Data Series and Data Points

Name	Type	Default Value
color1	java.awt.Paint	null
color2	java.awt.Paint	null
endCap	int (enumerated)	java.awt.BasicStroke.CAP_BUTT
lineJoin	int (enumerated)	java.awt.BasicStroke.JOIN_BEVEL
lineStyle	float[]	null

Table 19-7 Properties for Data Series and Data Points (Continued)

Name	Type	Default Value
lineWidth	float	1
miterlimit	float	10
stroke	java.awt.Stroke	null
annotation	IlvDataAnnotation	null
visible	boolean	true

Editing a Style Sheet

To edit a style sheet:

1. Open the style file in a text editor.
2. Edit the chart display properties.

For example, change the series color in the sample style sheet above to:

```
background : 250,250,250;
```

3. Edit the data display properties.

For example, change the series color properties for series one in the sample style sheet above to:

```
series[name="Series1"] {
    color1: "#993300-#993333"; (for gradient color)
    color2: "#0033ff";
```

To change the data point color to red if the data value is greater than 100:

```
point [y>100] {color1:red;}
```

USING SINGLE SIGN-ON

Vitria Web applications that are configured to participate in single sign-on require that a user login only once when using multiple applications. For example, if a user wants to access Business Cockpit and BusinessWare Task List, it is only necessary to login to the first application accessed. After providing a valid user name and password to login to BusinessWare Task List, the user can go directly to the Business Cockpit Homepage without first having to re-login to Business Cockpit.

When users log out of a Vitria Web application, the credential cookie created during login is removed, requiring users to enter a valid user name and password the next time they attempt to login. Logging out of one application logs you out of all applications. If a session times out in an application, but the single sign-on session has not timed out, the next time you access the application that timed out, you will not have to login again.

ENABLING SINGLE SIGN-ON

By default, Business Cockpit is not configured for single sign-on.

To enable single sign-on for Business Cockpit:

1. Open the following file:

installdir/data/install/cockpit-bservlet.xml

2. Search for and uncomment the following code:

```
<SSO configFile="$VITRIA/data/install/sso-
config.xml"/>
```

3. Save the file.

VALIDATING VIEWS

Although projects are automatically validated during deployment, validating your process views before deployment is a good way to check that they contain no errors prior to formal deploying and debugging.

Save any changes to the process views before you validate them.

To save your project:

1. From the **File** menu, select **Save** or **Save all**.

To validate your process views:

1. Right-click on a process view in the Explorer.
2. Click **Validate**.

Any errors encountered during validation appear in the Output window.

Note: If you are not using RDBMS persistence, your project deploys, but your snapshot process views will not work at runtime.

DEPLOYING VIEWS

Process views are deployed with the BusinessWare project in which they were created.

All Business Cockpit servers can serve all process views from all projects that are deployed to a common directory server. Business Cockpit serves all process views even if the project is not currently running.

DIRECTORY SERVER NAMESPACE

[Figure 19-9](#) shows an example of process views deployed in a directory server.

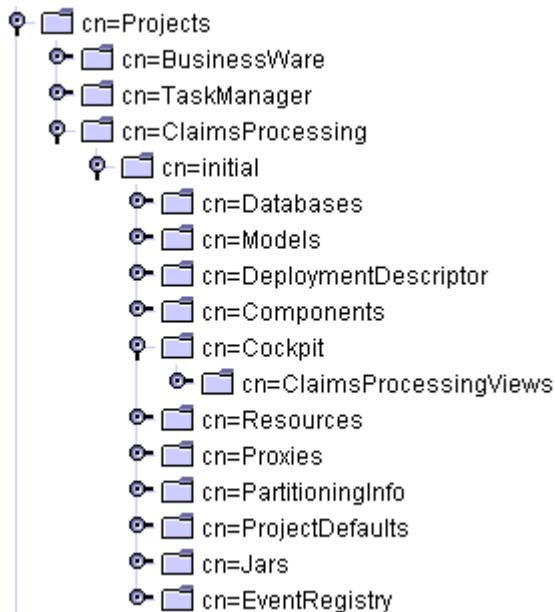


Figure 19-9 Directory Server Namespace

IMPORTANT: After a project is redeployed, the Cockpit administrator should reload all projects in Cockpit. See *Business Cockpit Guide* for more information.

SECURITY

Directory server administrators can set access controls on individual views. See the *BusinessWare Security Guide* for more information on security.

LOG FILES

The Cockpit trace level is defined under the Integration Server's trace level properties. The Cockpit log messages are in the Integration Server log. The Integration Server log file is usually located in the following directory:

installDir\logs

PROCESS VIEWS

Log Files

PART VI: DEPLOYMENT

This part describes how to configure an Integration Server, how to partition and deploy projects for use in a production environment, and how BusinessWare supports custom project installations.

Chapters include:

- [Integration Servers](#)
- [Deploying Projects](#)
- [Load Balancing](#)
- [Project Packaging](#)

PART #: TITLE HERE GOES HERE

An Integration Server provides the runtime execution context for projects. It also provides multiple platform services required at runtime, such as transaction management, connection management, logging, persistence, and Web server and servlet engine. This chapter discusses the configuration for the Integration Server and its associated platform services.

Topics include:

- [Integration Server Resource](#)
- [Configuring Integration Servers](#)

For more information on Integration Servers at runtime, see [Chapter 24, “BusinessWare Runtime Architecture.”](#)

INTEGRATION SERVER RESOURCE

An Integration Server is a resource created as part of a project.

To create an Integration Server resource:

1. From the main menu, select **File > New...**. The New Wizard appears.
2. Expand the **Resources** folder and select **Integration Server**.

For more information, see the *BME Help*.

Note: BusinessWare automatically creates an Integration Server for you if you deploy a project that does not contain at least one Integration Server in the current project or in any dependent project. For more information, see [“Deployment Options” on page 22-6.](#)

Integration Servers are integral to the deployment process because they are the entities to which you partition components (that is, assign components to run). For more information on deployment, see [Chapter 22, “Deploying Projects.”](#)

Integration Servers house the components and connectors in your project, and they are responsible for running the platform services as described in this chapter. Hence, there are several configuration properties you should understand. These properties are described in the remainder of this chapter.

IMPORTANT: Do not confuse the platform services associated with Integration Servers with application services. For information on the application services, see [Chapter 28, “Application Services.”](#)

CONFIGURING INTEGRATION SERVERS

An Integration Server defines several areas where you must configure properties. When you select the Integration Server in the Explorer, the Properties window displays two tabs that list categories of properties:

- Properties—contains the main properties applicable to the Integration Server itself. It also contains properties that will show or hide the existence of other objects underneath it (that is, other objects that contain properties for the platform services running inside the server).
- Trace Levels—contains the categories of trace levels you can set for objects running inside the Integration Server. See [Appendix B, “Error, Message, and Tracing Framework”](#) for more information on trace levels.

In addition, there are five types of child objects that may appear under the Integration Server when you expand it in the Explorer window. These objects let you configure various platform services provided by the Integration Server:

- **Logger**—service that manages message logging. You can have multiple Logger nodes for a single Integration Server.
- **Transaction Manager**—service that manages transactions.
- **Connection Manager**—service that manages connections to external systems.
- **Service Trace Levels**—the trace levels for debugging local services. See [“Setting Trace Levels” on page 20-4](#) for more information on trace levels and [“Application Services” on page 28-1](#) for information on local services.
- **RDBMS**—service that manages database persistence for application and recovery data.
- **Web Server**—service that manages Web server access. To use BusinessWare’s Web services capabilities, you can use the Web server provided with BusinessWare or specify your own external Web server.

The properties in the Properties tab for an Integration Server are described in [Table 20-1](#).

Table 20-1 Properties Tab for Integration Servers

Property	Description	Default Value
Name	Name of the Integration Server.	
Command	Command executable to start the process.	java
Java Options	Standard java command-line options, except –classpath (users should use Extra Classpath property to set classpath).	-Xms16m -Xmx64m
Extra Classpath	Part of classpath that will be prepended to the current user \$CLASSPATH. Also supports entries in the form: \$VITRIA/myclasses.	None
Port	Port on which the server listens for requests. Use 0 to make BusinessWare automatically pick an available port.	0
Max Threads	Specifies the number of threads allocated to the process in the default thread pool.	20
Enable Debug and Animation	Enables debugging and animation.	False
Debugging and Animation Port	Specifies the debugging and animation port.	8999
Persistent Store	Type of persistence: <ul style="list-style-type: none">• Cache• RDBMS Note: Database persistence is the only supported configuration in production. Cache mode is not supported in production.	Cache
Enable Web Server	Type of Web server: <ul style="list-style-type: none">• None• Internal• External Note: If BusinessWare automatically creates an Integration Server for you, it will also enable a Web server by setting this property to Internal.	None
Directory Server Connection Pool	A numeric value specifying the maximum number of connections allowed in the connection pool for the directory server.	20

INTEGRATION SERVERS

Configuring Integration Servers

Table 20-1 Properties Tab for Integration Servers (Continued)

Property	Description	Default Value
Directory Server Connection Timeout	The time (in seconds) that an inactive connection to the directory server is valid.	600
Admin Permission	Specifies the list of roles allowed to administer the Integration Server, such as changing attributes post-deployment, starting/stopping a server, etc. Leaving this property unchecked specifies that no authorization checks need to be completed.	The default value matches the value of the project object's default permission.

IMPORTANT: The first time an Integration Server configured with a Port value of 0 is started, it chooses a freely available port. This port number is persisted at runtime in the directory server. Each time the Integration Server is started, the same port number is used. You can only change the port number by re-deploying the Integration Server.

SETTING TRACE LEVELS

The properties in the Trace Levels tab enable you to specify the level of logging you want for each type of object or service in your Integration Server, as shown in [Table 20-2](#). For each property, an integer parameter determines the level of detail written to the log. The values range from 0 to 5, where:

- 0 - NONE = No logging
- 1 - ERROR = Log major, serious errors
- 2 - WARNING= Log warnings
- 3 - NORMAL = Log major, normal occurrences
- 4 - VERBOSE = Log all occurrences
- 5 - TRIVIA = Log full details for occurrences

Table 20-2 Trace Levels for an Integration Server

Property	Description	Default
Global	Logs overall environmental details for the Integration Server, including Bservlet and User Manager details.	3-NORMAL
Container	Logs details for the container running within the selected Integration Server.	3-NORMAL
Connector	Logs details associated with a specific connector.	3-NORMAL

Table 20-2 Trace Levels for an Integration Server (Continued)

Property	Description	Default
Process Model	Check this log level when creating log messages in action code and workflow related traces. See the OrderProcessSample models for examples.	3-NORMAL
Process Query	Logs process query details.	3-NORMAL
Transformer	Logs BusinessWare Transformer details.	3-NORMAL
Cockpit	Logs process view details.	3-NORMAL
Connection Manager	Logs Connection Manager details.	3-NORMAL
Transaction Manager	Logs Transaction Manager details.	3-NORMAL
RStore	Logs RStore details.	3-NORMAL
Orb	Logs BusinessWare CORBA ORB details.	3-NORMAL
Thread	Logs details for all threads associated with the selected Integration Server.	3-NORMAL
Transport	Logs network operation details.	3-NORMAL
Type System	Logs details associated with metadata management at runtime.	3-NORMAL
Web Server	Logs details associated with your specified Web server.	3-NORMAL
Security	If the Security Service is configured, logs associated details.	3-NORMAL

See [Appendix B, “Error, Message, and Tracing Framework”](#) for more information on trace levels.

CONNECTION MANAGER

The connection manager coordinates the connections made within an Integration Server, such as connections between connectors and their external applications. You can configure the connection manager by specifying the properties shown in [Table 20-3](#).

Table 20-3 Connection Manager Properties

Property	Description	Default
Max Connections	Maximum number of connections for each Connection Factory. A Connection Factory is a connection pool used for a specific resource (such as Oracle database target connectors). A Connection Manager is a container of connection factories. The Max Connections value can be increased up to the Max Threads value of the Integration Server.	20
Min Connections	Number of connections that should be maintained (kept alive).	0
Connection Request Timeout	Time to wait (in seconds) before freeing a connection.	5
Unused Connection Timeout	Time to wait (in seconds) before closing an unused connection.	180

You should increase the Max Connections value if you expect to have multiple timers firing simultaneously in process models running on the Integration Server. Each timer uses a connection, and this may cause the number of actual connections to exceed the specified limit. In this case, `ResourceAllocationExceptions` occur.

SERVICE TRACE LEVELS

Select the Service Trace Levels node under the Integration Server to set values for the following four trace level properties:

- **Logging Service**—if the Logging Service is configured, logs associated details.
- **Document Store Service**—if the Document Store Service is configured, logs associated details.
- **Registry Service**—if the Registry Service is configured, logs associated details.
- **Security Service**—if the Security Service is configured, logs associated details.

For a description of the possible property values, see “[Setting Trace Levels](#)” on page 20-4.

TRANSACTION MANAGER

Each BusinessWare Integration Server has a transaction manager that automatically coordinates with other transaction managers. When you deploy a project, you can adjust a transaction manager’s behavior by setting its properties.

Each transaction manager uses the following properties to control transaction timeouts. You can find these properties by clicking on the transaction manager node.

Table 20-4 Transaction Manager Properties

Property	Description	Default
Duration Timeout	<p>Time (in seconds) for the transaction manager to wait for the application to call <code>commit</code> before delivering an abort outcome to resource managers.</p> <p>The timer starts when the application begins the transaction.</p> <p>This property limits the time resources are locked if the application has a problem.</p> <p>The default of 0 means infinite timeout.</p>	0
Inactivity Timeout	<p>Time (in seconds) for a Sub-Coordinator to wait for another call from a Coordinator that has begun a transaction.</p> <p>The timer begins when the Sub-Coordinator returns from a call by the Coordinator. After this time, the Sub-Coordinator delivers an abort outcome to its resource managers.</p> <p>This property limits the time resources are locked if there is a problem with the network or the Coordinator.</p> <p>The default of 0 means infinite timeout.</p>	0
Retry Count	Controls the number of times that the system will attempt to commit or rollback a specific resource before halting the Integration server. It is an ‘outband’ operation. Specify zero (0) to retry infinitely. See “ Outband Exceptions ” on page 26-4 for more information.	30
Retry Interval	Specifies time (in seconds) between retries.	10

For instructions on examining and setting these properties, access the *BME Help*.

DATABASES

You can configure each Integration Server to use one of two types of persistent storage:

- Cache
- RDBMS

By default, an Integration Server is configured to use Cache persistence. Cache mode should be used only as a convenience for initial development because data is stored in memory only. You must use RDBMS persistence in a production system to maintain both application and recovery data.

IMPORTANT: Database persistence is the only supported configuration in production. Cache mode is not supported in production.

To configure your Integration Server to use a database for persistence:

1. Click on the **Integration Server** object in the Explorer.
2. In the Properties window, select **RDBMS** in the drop-down box for the **Persistent Store** property.
3. Expand the Integration Server object in the Explorer and select the **RDBMS** object.
4. Configure the RDBMS object properties.

For more information on configuring Integration Servers, see the *BME Help*.

[Table 20-5](#) shows the persistent store properties associated with RDBMS.

Table 20-5 RDBMS Persistent Store Properties

Property	Description	Default
Database Resource	The full path name of the Database Resource that should be used. It can be defined in the current project or a dependent project.	None
Truncate Strings	Enables BusinessWare to truncate string values (VARCHAR) if the strings are longer than the database's supported column size.	ON

Table 20-5 RDBMS Persistent Store Properties (Continued)

Property	Description	Default
Serialize Blob	Controls how object data is stored in the database. If this property is set to OFF, objects will be mapped using the normal object-relational mappings. If it is set to ON, objects will be mapped as key/blob pair, where all data is stored as a blob. Note: Only default schemas are blob supported.	OFF
Prefix	Specifies the prefix of internal and generated table names. Note: RDBMS-specific key words and special characters cannot be used since the value is dependent on the RDBMS type.	None
Special Char	Specifies the separator in generated table names and field names. Note: RDBMS-specific key words and special characters cannot be used since the value is dependent on the RDBMS type.	\$
Class Separator	Specifies the separator in non-generated table names. This character replaces the '.' character.	#

Access the *BME Help* for information about the properties for the different types of database resources.

XA Configuration for Oracle

If Oracle is used for Integration Server persistence, the Oracle user specified in the database resource must have database administrator privileges or the following set of permissions:

```

DBA_2PC_PENDING SELECT
DBA_PENDING_TRANSACTIONS DELETE
DBA_PENDING_TRANSACTIONS INSERT
DBA_PENDING_TRANSACTIONS SELECT
DBA_PENDING_TRANSACTIONS UPDATE
DBA_PENDING_TRANSACTIONS REFERENCES
DBA_PENDING_TRANSACTIONS ON COMMIT REFRESH
DBA_PENDING_TRANSACTIONS QUERY REWRITE
DBA_PENDING_TRANSACTIONS DEBUG
DBA_PENDING_TRANSACTIONS FLASHBACK
JAVA_XA EXECUTE
V$PENDING_XATRANS$ SELECT

```

```
V$XATRANS$    DELETE
V$XATRANS$    INSERT
V$XATRANS$    UPDATE
V$XATRANS$    REFERENCES
V$XATRANS$    ON COMMIT REFRESH
V$XATRANS$    QUERY REWRITE
V$XATRANS$    DEBUG
V$XATRANS$    FLASHBACK
V$XATRANS$    SELECT
```

LOGGERS

Loggers control how system messages are handled. Clicking on the Logger node lets you configure properties such as log file location, size, encoding, and so on. By default, the log entries for each Integration Server in your project are written to a single text file. However, you can add as many additional text file loggers, binary file loggers, and channel loggers as you wish. You do so in the Integration Server ([Figure 20-1](#)).

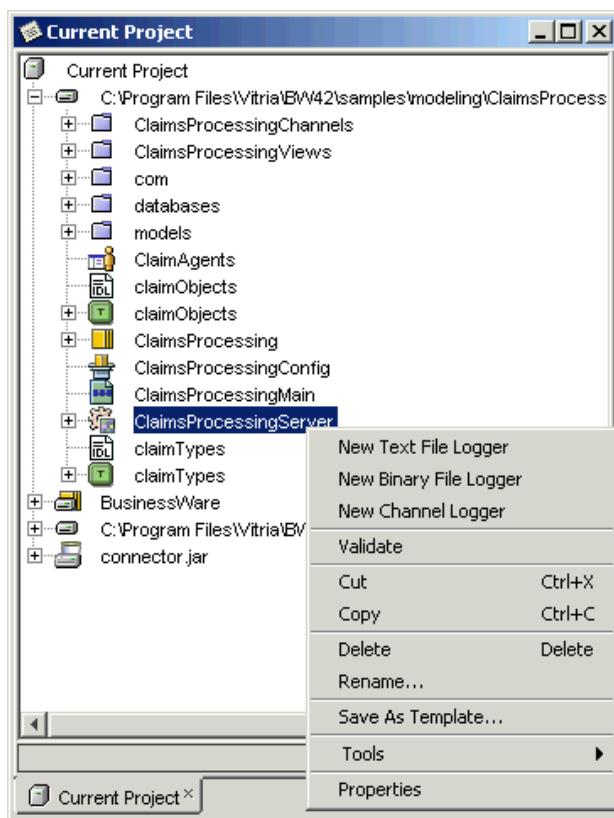


Figure 20-1 Adding a Logger for an Integration Server

To add a logger:

1. In the Explorer, right-click the Integration Server.
2. Choose the type of logger you want to create.
3. Set properties for the new logger.

To specify the level of detail you want in the logs for a given server, you set the trace levels for that server. Set the trace levels for your Integration Servers within the project, as described below.

To set trace levels:

1. In the Explorer, right-click the Integration Server resource.
2. In the **Trace Levels** tab of the Properties window, set the level for each service.

Note: You can have BusinessWare generate custom log messages specific to your application by adding diagnostic messaging methods in the Action Builder or in your Java code. See [Chapter 14, “Process Model Code Construction”](#) for more information on logging actions.

Logger Properties

The tables below describe the properties associated with each type of logger. This information is also available in the *BME Help*.

[Table 20-6](#) describes the properties for the Text File Logger.

Table 20-6 Text File Logger Properties

Property	Description	Default
File Name	Specifies the fully qualified name of the file to which log messages are printed.	\$VITRIA/logs/ <i>IntegrationServerName.log</i>
File Size	Sets the maximum size of a log file (in bytes). Log data is backed up as specified by the Number of Backups property. As log data grows, old backups are purged.	500000
Number of Backups	Sets the number of log backup files to create before purging old data. Files are named with extensions .0, .1, .2, etc.	1
Flush After Message	Specifies whether to write to the log file after a message is received to ensure that the message is written. If true, the message is written immediately to the log file.	True
Source Prefix	Specifies whether to include the module that sent the message in the log.	False
Message ID Prefix	Specifies whether to include the message resource name in the log.	False
Thread ID Prefix	Specifies whether to include the current thread ID in the log.	True
Time Prefix	Specifies whether to log the long version of the time at which the message occurs. The long timestamp displays the month, date, time of day, day of week, and year.	False
Short Time Prefix	Specifies whether to log the short version of the time at which the message occurs. The short timestamp displays the month, date, and time of day.	True
Category Prefix	Specifies whether to include the message category in the log output.	False
Encoding	Specifies the output encoding for localization.	UTF-8

Table 20-7 describes the properties for the Binary File Logger.

Table 20-7 Binary File Logger Properties

Property	Description	Default
File Name	Specifies the fully qualified name of the file to which log messages are printed.	\$VITRIA/logs/ <i>IntegrationServerName</i> .log
File Size	Sets the maximum size of a log file (in bytes). Log data is backed up as specified by the Number of Backups property. As log data grows, old backups are purged.	500000
Number of Backups	Sets the number of log backup files to create before purging old data. Files are named with extensions .0, .1, .2, etc.	1
Flush After Message	Specifies whether to write to the log file after a message is received to ensure that the message is written. If true, the message is written immediately to the log file.	True

Table 20-8 describes the properties for the Channel Logger.

Table 20-8 Channel Logger Properties

Property	Description	Default
Channel Name	Specifies the channel resources to which logging information is sent.	
Event Batch Size	Specifies the number of events to send before the server confirms that a subscriber channel has received the events.	10

You can also access the *BME Help* for information on creating additional loggers and configuring their properties.

WEB SERVER

BusinessWare includes a Web server/servlet engine out-of-the-box so you can immediately take advantage of the available Web-based functionality such as Web services, HTTP connectivity, Business Cockpit, and the Web Administration tool.

Clicking on the Integration Server node lets you configure the type of Web server to use by setting the **Enable Web Server** property to the required value—internal, external, or none.

INTEGRATION SERVERS

Configuring Integration Servers

To view or set the internal Web Server properties, select the node underneath the Integration Server in the Explorer window.

Table 20-9 Internal Web Server Properties

Property	Description	Default
HTTP Compression Type	The type of compression to enable on the web server. <ul style="list-style-type: none">• Off—Do not use compression• On—use compression if client supports it• Force—always use compression.	On
Protocol	Protocol supported by the Web server: <ul style="list-style-type: none">• HTTP• HTTPS• BOTH	HTTP
HTTP Port	Specifies the port on which HTTP requests will be accepted.	8081
HTTPS Port	Specifies the port on which HTTPS requests will be accepted.	8444
Max Threads	Specifies the number of threads allocated to process incoming requests to the Web server. This value must be greater than five.	20
Min Threads	Number of request processing threads that will be created when the Web server is started.	0
Max Keep Alive Requests	The maximum number of HTTP requests which can be pipelined until the connection is closed by the server. Setting this attribute to 1 will disable HTTP/1.0 keep-alive, as well as HTTP/1.1 keep-alive and pipelining. Setting this to -1 will allow an unlimited amount of pipelined or keep-alive HTTP requests.	-1
Connection Acceptance Count	Specifies the maximum number of connection requests that are queued for the Web server. If a connection request arrives and the limit has been reached already, the connection is refused.	10
Connection Timeout	Specifies the amount of time an inactive request connection will remain open.	5 seconds
Ciphers	The allowed ciphers in the secure connection while running in HTTPS mode.	
Event Listener Class	Specifies the listener class for Web server events that can be used to implement a URL stream handler for specific protocols.	

To view or set the external Web server properties, select the node underneath the Integration Server in the Explorer window.

Table 20-10 External Web Server Properties

Property	Description	Default
Redirection Protocol	Protocols supported for redirection from external Web servers. Currently, the protocol must be AJP.	AJP (read-only property)
Redirection Port	Port on which the servlet will listen for requests.	8010
Max Threads	The number of threads allocated to process incoming requests to the Web server.	20
Min Threads	The number of request processing threads that will be created when the servlet engine is started.	5
Connection Acceptance Count	Specifies the maximum number of connection requests that are queued for the Web server. If a connection request arrives and the limit has been reached already, the connection is refused.	10
Connection Timeout	Specifies the amount of time an inactive request connection will remain open.	5 seconds
Event Listener Class	Specifies the listener class for Web server events that can be used to implement a URL stream handler for specific protocols.	

Web Applications Directory Structure

All HTTP connectors in a project that are partitioned to an Integration Server are deployed to a Web application named after the project on the Integration Server. When the project is started on the Integration Server, the project's Web applications are written to disk on the Integration Server's home directory.

For example, [Figure 20-2](#) illustrates a scenario in which you have two projects (Project1 and Project2) with HTTP connectors (HCx) partitioned across two Integration Servers.

INTEGRATION SERVERS

Configuring Integration Servers

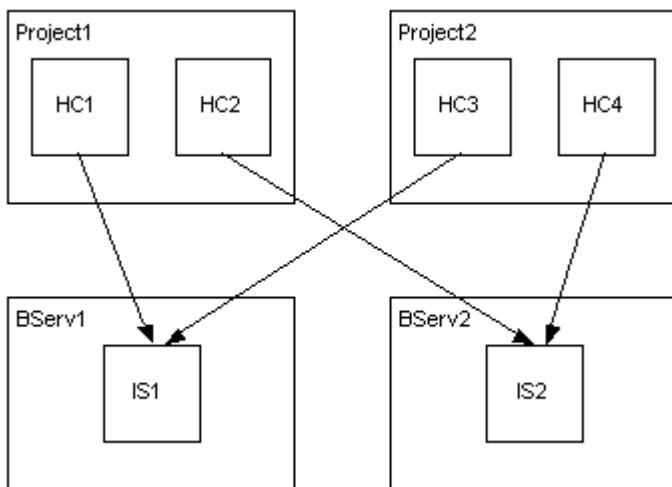


Figure 20-2 Sample Web Server Partitioning

Table 20-11 describes the directory structure for this scenario.

Table 20-11 Sample Directory Structure for Partitioned Web Servers

Directory Path	Description
%Vitria%\web\servers\bserv1	Home directory for BusinessWare Server – bserv1
\IS1	Home directory for Integration Server 1
\webapps	Home directory for Web applications loaded by the Integration Server
\project1name-version	Entries for HTTP Connector 1
\project2name-version	Entries for HTTP Connector 3
%Vitria%\web\servers\bserv2	Home directory for BusinessWare Server – bserv2
\IS2	Home directory for Integration Server 2
\webapps	Home directory for Web applications loaded by the Integration Server
\project1name-version	Entries for HTTP Connector 2
\project2name-version	Entries for HTTP Connector 4

For more information on the BusinessWare directory structure, see the *BusinessWare Administration Guide*. For information on partitioning projects, see [Chapter 22, “Deploying Projects.”](#)

This chapter describes how to implement load balancing using the tools provided in the BME.

Topics include:

- [Introduction to Load Balancing](#)
- [Designing Load Balanced Projects](#)
- [Configuring a Project for Load Balancing](#)
- [Debugging and Animating Clusters](#)
- [Administering Clusters](#)
- [Error Handling](#)

INTRODUCTION TO LOAD BALANCING

Load balancing is the distribution of a stream of requests to multiple servers for processing. Load balancing has two main benefits:

- **Scalability**—throughput of the system is increased by adding more servers
- **Fault tolerance**—in the event of a server crash, requests are dynamically reassigned to other servers

In BusinessWare, load balancing is accomplished using clusters. A *cluster* is a collection of Integration Servers distributed across one or more s.

A cluster contains:

- **Master node**—an Integration Server from which a cluster is created. The master node runs both the clusterable and non-clusterable items in a project.
A master node is indicated by  in the Deployment Editor window.
- **Slave nodes**—Integration servers that are members of a cluster that run only the clusterable components partitioned to the cluster. Slave nodes inherit most attribute values from the master node. A cluster can have zero or more slave nodes.

Not all components of a BusinessWare integration model can be clustered. Specifically, most source connectors are non-clusterable because running them on multiple nodes would cause the same source data to be processed by multiple servers. For example, if a File Source connector were run on two nodes of a cluster, placing a file in the source directory would produce two identical events, one from each source connector.

Most components of a BusinessWare integration model are clusterable, including process components, all proxies, all target connectors, and the following source connectors:

- Queue Connector
- JMS Connector (when connected to a queue)
- HTTP Connector (using an HTTP redirector)

If you want to implement load balancing in a project with non-clusterable components, see “[Asynchronous Load Balancing](#)” on page 21-6.

If you want to load balance HTTP requests, use an HTTP redirector to distribute the requests to the internal Web servers of the Integration Servers in the cluster.

If your project uses BPOs or DOs, then any clustered Integration Servers must use database persistence. A process component can be stateful because all servers use the same database for persistence. As a result, different requests to the same BPO may be handled by different servers.

Note: To avoid database contention, timer events are delivered only to the master Integration Server in a cluster.

DESIGNING LOAD BALANCED PROJECTS

When you design your models, you should consider whether the project will implement synchronous or asynchronous load balancing. The following sections describe how load balancing is implemented in BusinessWare and compare synchronous versus asynchronous load balancing.

SYNCHRONOUS LOAD BALANCING

If events enter a BusinessWare model through synchronous calls on a Simple Input proxy, the events can be distributed to replicas of the proxy running on one or more Integration Servers in a cluster.

When a servant (input proxy or port) that is running on a clustered Integration Server is resolved, the object returned is a Java dynamic proxy containing a stub for each instance of the servant on each server in the cluster. [Figure 21-1](#) shows a simple integration model and [Figure 21-2](#) shows the runtime view of this model.

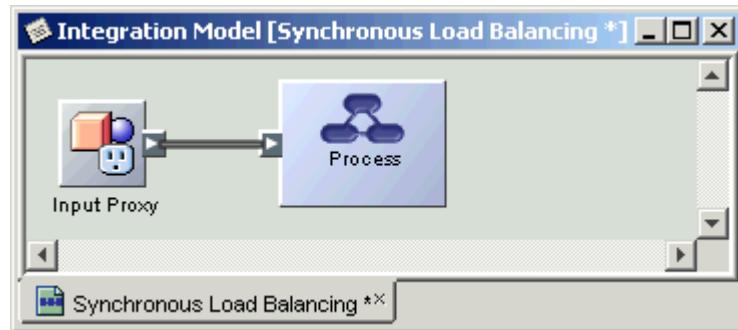


Figure 21-1 Synchronous Load Balancing (Design-Time View)

Invoking an operation on the dynamic proxy causes one of the servers to be selected ([Figure 21-2](#)) and the operation is then forwarded to the stub for that server.

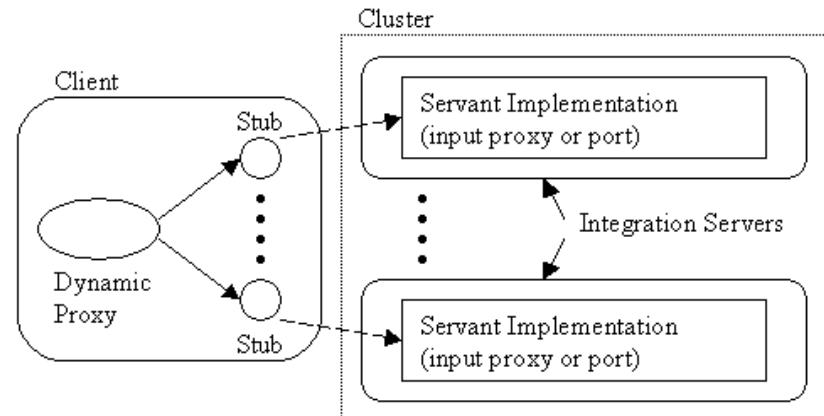


Figure 21-2 Synchronous Load Balancing (Runtime View)

Synchronous Call Routing

When a synchronous CORBA or RMI call is made on a clustered input port or proxy, the request is sent to one instance of the port or proxy as determined by the following criteria:

- If there is an instance of the port or proxy running locally on the same Integration Server, this instance is used (Local Affinity).
- All invocations on a clustered port or proxy made within a given transaction are sent to the same instance of the port or proxy (Transaction Affinity).
- If there is no local instance and no prior invocations on the port or proxy in the current transaction, then the cluster's call router class is invoked to select the instance.

Using Custom Routing Policies

Applications control the distribution of synchronous requests to running servers by providing a class implementing the `ClusterCallRouter` interface, shown below. Three predefined classes are included with BusinessWare:

- A class implementing Round-robin routing
- A class implementing Random routing
- A class that distributes requests to servers weighted by their average response time

To specify a custom routing policy:

1. Create a Java class that implements the `ClusterCallRouter` interface:

```
package com.vitria.container.client;
public interface ClusterCallRouter {
    /**
     * This is called before getServer and whenever set of
     * servers changes.
     */
    void setServerList(String [] servers,
                      String clusterName);
    /**
     * Returns server name to use, possibly based on
     * servant's JNDI name,
     * method called, and actual parameters.
     */
    String getServer(String jndiName, Method method,
                    Object[] params);
    /**
     * Called before invocation on server starts.
    */
```

```

        */
void beforeCallOnServer(String jndiName, Method method,
    String serverName);
/**
 * Called after invocation on server returns.
 */
void afterCallOnServer(String jndiName, Method method,
    String serverName);
}

```

In the New Cluster window, enter the fully qualified class name for the routing policy in the Routing policy text field.

On startup and whenever the set of running servers changes, the BusinessWare runtime invokes `setServerList()` to tell the router the names of the running Integration Servers. On each invocation where transaction and local affinity rules do not apply, the `getServer()` method of the cluster's call router is invoked to select the server for this call, based on the JNDI name of the servant (the port or proxy binding path), the method name, and the actual parameters passed in the call.

In addition, the cluster call router receives callbacks from the runtime before and after any call on a servant in the cluster. The predefined response-time weighted call router uses these callbacks to measure the response time of the remote server.

No synchronization is required for call routers: all method calls are serialized by the container runtime. The following example is a cluster call router that routes "placeOrder" events whose first argument is an integer to server 1 if the argument is even, or to server 2 if the argument is odd.

```

public class ParityCallRouter implements
    ClusterCallRouter {
String [] servers_;
public ParityCallRouter() { }
public void setServerList(String [] servers,
    String clusterName) {
    servers_ = servers;
}
/**
 * Return first server if first argument is even, second
 * if
 * odd.
*/
public String getServer(String port, Method method,
    Object[] params) {
    if (servers_ == null || servers_.length == 0)
        return null;
    if (servers_.length == 1 || params.length == 0 ||

```

LOAD BALANCING

Designing Load Balanced Projects

```
        !(params[0] instanceof Integer))
        return servers_[0];
    int arg = ((Integer)params[0]).intValue();
    return servers_[arg % 2];
}

public void beforeCallOnServer(String jndiName,
    Method method, String serverName) { }

public void afterCallOnServer(String jndiName,
    Method method, String serverName) { }
}
```

ASYNCHRONOUS LOAD BALANCING

This section describes how to use asynchronous load balancing with queues to distribute event processing from non-clustered sources (such as the FTP connector) across servers.

As mentioned above, not all project items can be clustered. Most source connectors cannot be shared across servers in a cluster. Each cluster contains an Integration Server that acts as the master node. Non-clusterable components run only on the cluster's master node.

You can use queues managed by BusinessWare or external queues, managed by a JMS server, to distribute asynchronous requests to the Integration Servers in a cluster. BusinessWare queues provide better performance and are more integrated into the modeling environment, but a JMS server that uses a database for persistence can simplify administration (for example, by making consistent backups).

Unlike most source connectors, queue source connectors are clusterable. If data is entering BusinessWare through a non-clusterable source connector (such as the FTP connector), then the best way to cluster the BusinessWare processing is to insert a Queue TargetSource connector just after the non-clusterable source connector.

The following example describes how to implement load balancing on a project that contains non-clusterable components.

Consider a project that has an FTP source connector that sends events to a process component as shown in [Figure 21-3](#).

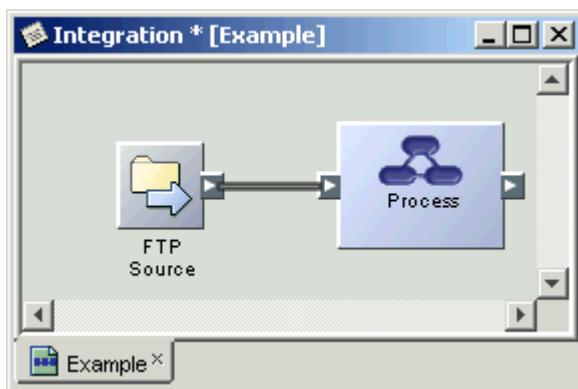


Figure 21-3 Original Integration Model with Non-Clusterable Source

Because the FTP connector cannot be clustered, the best way to implement load balancing on this project is to place a queue between the FTP source connector and the process component and use asynchronous load balancing to distribute the requests. Because connectors cannot be wired directly together, a simple pass-through model must be inserted between the connector and the queue as shown in [Figure 21-4](#).

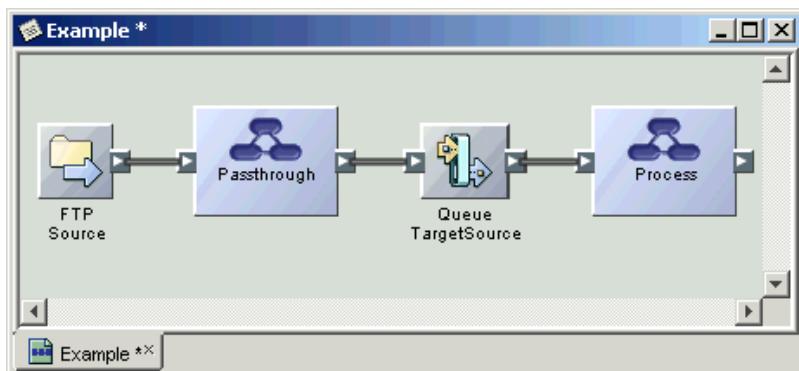


Figure 21-4 Modified Integration Model (Design-Time View)

Without a Queue Target-Source connector, the FTP Source connector would run only on the master Integration Server and all orders would be processed by the components on that server. With the Queue Target-Source connector, the FTP connector and passthrough model run only on the master, but the queue source and main process component run on all servers. As a result, orders are processed concurrently on multiple servers as shown in [Figure 21-5](#).

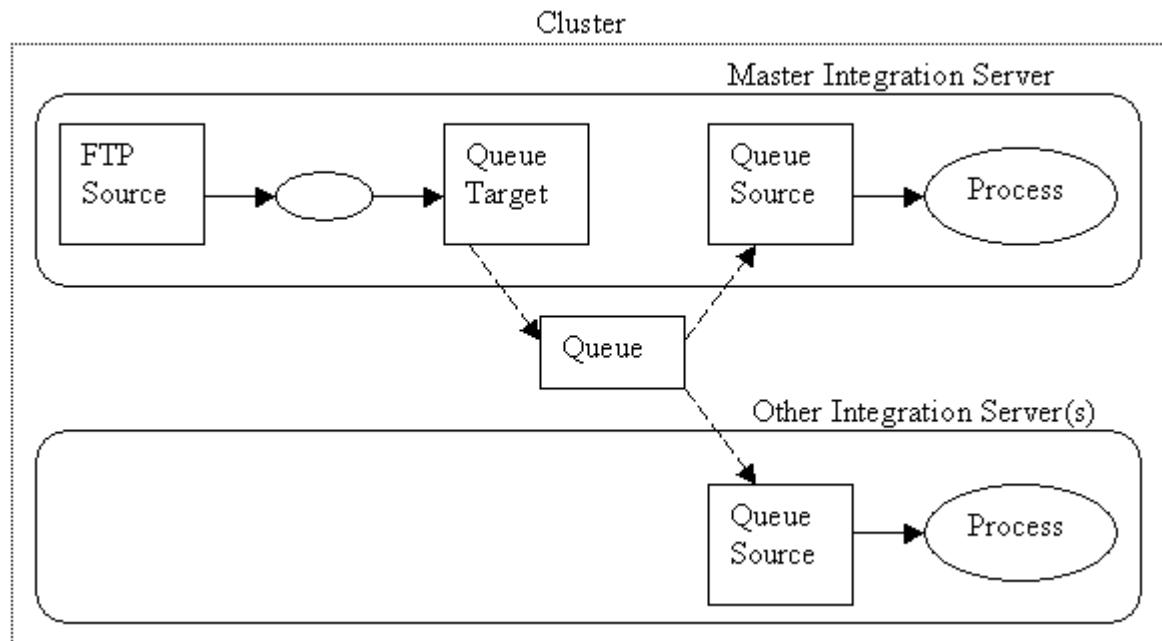


Figure 21-5 Modified Model (Run-Time View)

In Figure 21-5, processing of an event occurs in two separate transactions: one to move the event from the source connector to the queue, and the second to retrieve the event from the queue and push it through the process model.

ASYNCHRONOUS VS. SYNCHRONOUS LOAD BALANCING

The following sections describe the advantages and disadvantages of asynchronous and synchronous load balancing.

Asynchronous Load Balancing

The advantages of asynchronous load balancing using queues are:

- **Performance**—Asynchronous load balancing enables batching of requests, which generally improves performance. Large batches of events can be retrieved from the queue and pushed through the model in a single XA transaction, greatly reducing transaction overhead and increasing throughput.
- **Automatic distribution of events by demand**—Because servers pull events down from a queue as needed, events are automatically sent to each server at the rate that server can process them.

The disadvantages of asynchronous load balancing are:

- **Less fault tolerant**—Because only one instance of a non-clusterable source connector is running, the master server is a single point of failure, as is the server holding the queue.
- **No routing**—Synchronous requests can be routed to specific servers based on information in the event, but for asynchronous requests, the application has no control over which queue subscriber (such as an Integration Server) receives a given event.
- **Ordering not preserved**—Because the application cannot control the distribution of requests to different servers, it has no control over the order in which events are processed. For example, if event 1 represents the creation of a bank account and event 2 represents a deposit into that account, these events may be processed out of order (resulting in an error) if the events are delivered to different servers, even if event 1 precedes event 2 in the queue. The events in a queue must be independent.

Synchronous Load Balancing

The key advantages of synchronous load balancing are:

- **Fault Tolerance**—Because all servers are identical, no Integration Server is a single point of failure. However, the directory server and database servers used by the Integration Server must be available for continued operation.
- **Control over routing of requests**—Each cluster has a call router class. You can select a predefined router class implementing a simple policy (for example, round-robin), or write your own class that routes each synchronous call to a specific server based on the method and actual parameters of the invocation. For more information on routers, see “[Synchronous Call Routing](#)” on page 21-4.
- **No changes to models**—No queues or connectors need to be added to the integration model to support synchronous load balancing. Clusters are created at deployment time.

The key disadvantages of synchronous load balancing are:

- **Lower performance**—During synchronous invocation, each call either starts or resumes a transaction causing overall performance and throughput to be lower than with asynchronous load balancing, which can batch several events into each transaction. You can minimize the performance impact by triggering as much transactional work as possible with each invocation. For example, a large EDI document comprising many items can arrive in a single CORBA/RMI call.

LOAD BALANCING

Configuring a Project for Load Balancing

- **Routing can be complex**— If servers run at different speeds, simple routing policies like round-robin do not distribute requests optimally (for example, if server 1 is twice as fast as server 2, it should be sent twice as many requests to keep it busy). You can use the Response-time weighted routing policy to distribute requests based on the average response time from each server (for example, if server 1 takes half the time, per invocation, on average, than server 2, then it will be sent twice as many requests).

CONFIGURING A PROJECT FOR LOAD BALANCING

In general, configuring a project for load balancing involves the following steps:

1. Open a project that does not contain any clusters.
2. Open a deployment configuration and create a new cluster for the Integration Server that you want to cluster. See “[Creating Clusters](#)” on page 21-11 for more information.
3. Build and deploy the project. The components originally partitioned to the Integration Server are now deployed to run in a cluster.

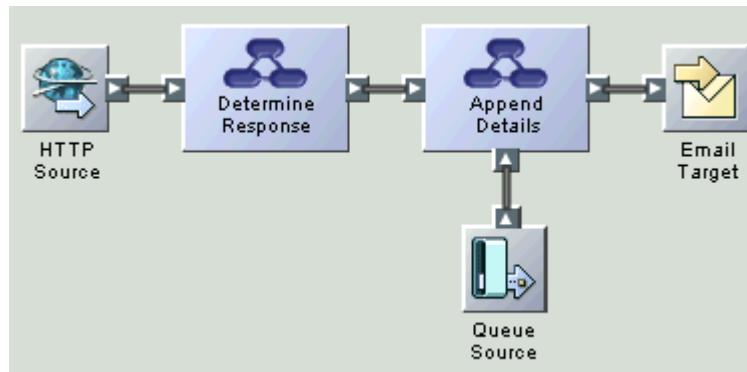


Figure 21-6 Integration Model in Load Balanced Project

For example, a project containing the integration model in [Figure 21-6](#) could be deployed to run on a cluster with one slave node. Two concurrently executing instances (one in the master node and one in the slave node) of the Determine Response and Append Details process models would receive events from the HTTP Source and Queue Source connectors.

CREATING CLUSTERS

After you have created a deployment configuration that contains the Integration Server for which you want to implement load balancing, you can create a new cluster.

To create a new cluster:

1. Open a deployment configuration.
2. In the Editor, right-click on an Integration Server.
3. Click on **New Cluster....** The New Cluster window opens.

Note: If you select an unpartitioned Integration Server, the New Cluster window opens and you must select a BusinessWare Server to which the Integration Server will be partitioned.
4. Enter the number of slave nodes to be added to the cluster. Slave nodes are automatically named by appending *_Slave_X* to the name of the master node. X represents the current number of slave nodes following the default naming convention.
5. Select a routing policy from the drop-down list. The following policies are available:
 - **Round robin**—events are handled in a fixed, cyclic order
 - **Random**—events are handled in no specific order
 - **Weighted by response time**—if server 1 responds twice as fast as server 2, server 1 will be sent twice as many requests
6. Enter the refresh interval in seconds. The default interval is 120 seconds.
7. Click **OK**. A new cluster is created with the specified number of slave nodes and the Integration Server icon changes to  indicating that the server is now clustered.

Figure 21-7 shows an example of a cluster and a clustered Integration Server in a deployment configuration.

LOAD BALANCING

Configuring a Project for Load Balancing

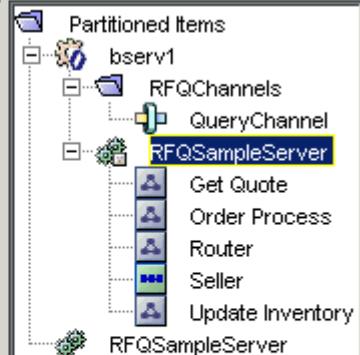
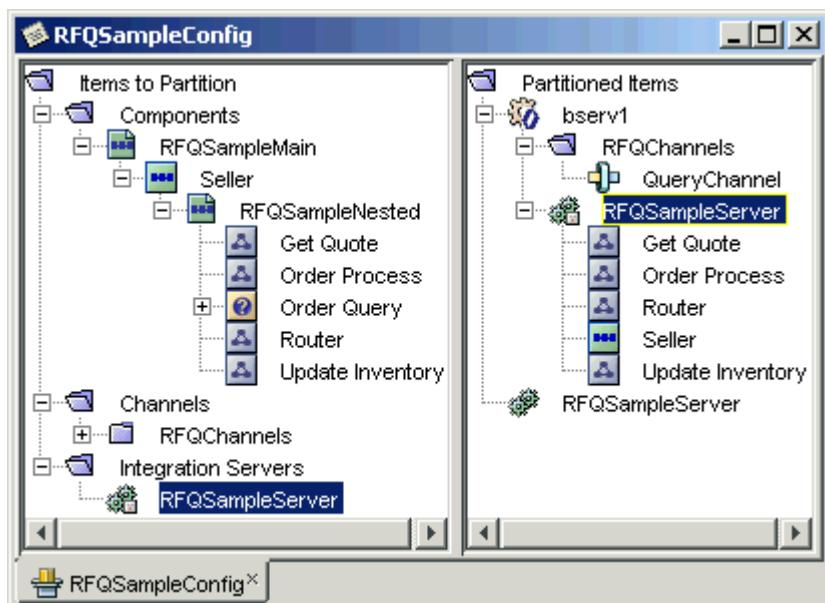


Figure 21-7 Cluster in Deployment Configuration

Tip: If you plan to implement load balancing in the future, but only need one server at present, you can create a cluster with zero slave nodes. The advantage of this configuration is that you can create slave nodes in the future without having to stop a running project.

IMPORTANT: You can create clusters only for Integration Servers defined in the current project. Integration Servers defined in dependent projects cannot act as a master node for the current project. However, you can create a cluster for an Integration Server in a dependent project and use it in the current project.

CONFIGURING A NEW CLUSTER

[Table 21-1](#) lists the cluster properties and their default values. These properties are configured when you create a new cluster or when you edit an existing cluster.

Table 21-1 Cluster Properties

Property	Description	Default
Name (Read Only)	Specifies the name of the cluster.	<i>Integration Server Name</i>
Master Node	Specifies the Integration Server that runs clustered and non clustered components.	Currently selected Integration Server
Number of Slave Nodes	Specifies the number of slave nodes.	1
Routing Policy	<p>Determines how transactions are routed to the nodes. Choices are:</p> <ul style="list-style-type: none"> • Round robin • Random • Response-time weighted <p>Note: You can also reference a custom Java class. See “Using Custom Routing Policies” on page 21-4.</p>	Round robin
Refresh Interval	Specifies the time interval for detecting changes to the cluster.	120 seconds

IMPORTANT: If the Persistent Store property of the master node is set to RDBMS, it is recommended that the Transaction Isolation property of the RDBMS resource be set to *SERIALIZABLE*. Under some conditions, you may know that events for the same BPO will never be concurrently delivered to different Integration Server instances in the cluster. For such conditions only, it is safe to use a weaker isolation level such as *READ_COMMITTED*.

CREATING CLUSTERS WITH WEB SERVICE INPUT PROXIES

A Web service generates a WSDL file after deployment and sends it to the client. The client uses the service location URL to connect to the service. When clustering projects containing a Web Service Input Proxy, you must do the following to ensure that the client connects to the service:

- Before handing the WSDL file to the client, the service location URL address must be modified so that the address points to the location of the Web server specified in the master node.
- An HTTP redirector must handle the redirection of the incoming calls. The redirector should be configured so that the service location address specified in the WSDL file in the previous step maps to other ports present in the clustered project. See the *BusinessWare Administration Guide*.

EDITING CLUSTERS

It is possible to edit an existing Cluster.

To edit a cluster:

1. Open a deployment configuration.
2. In the Editor, right-click on a master node and click **Edit Cluster**. The Edit Cluster window opens.

The Edit Cluster window displays cluster base properties and additional properties based on the Enable Web Server property value of the master node.

When editing Clusters, you can do the following:

- Modify the following cluster properties:
 - Routing policy
 - Refresh interval
- Add or remove slave nodes

Additional slave nodes are created with property values inherited from the master node. Slave node names are automatically sequentially numbered. For example:

- RFQSampleServer_Slave_1
- RFQSampleServer_Slave_2
- Modify the following slave node properties:
 - Integration Server (slave node name)

- BusinessWare Server (It is possible to partition slave nodes of a common master node on different BusinessWare Servers.)
- Port
 - Note:** When specifying port numbers make sure that they do not collide with another Integration Server's port number. If they collide and the other server is already running then the project will not start completely.
- Additional properties determined by the Enable Web Server property value of the master node

The next time you deploy the project, the changes are made in the directory server.

UPDATING A DEPLOYED CLUSTER

You can update different attributes of a cluster that is already deployed without affecting the rest of the project using the Update Deployed Cluster command. This is very useful when you want to change the number of slave nodes without stopping a running project.

To update a deployed cluster:

1. Edit the cluster attributes as described in “[Editing Clusters](#)” on page 21-14.
2. Open a deployment configuration and Right-click on a master node or cluster object in the Editor.
3. Click **Update Deployed Cluster**.

After updating the cluster, a window opens providing information on the status of the update.

If master node properties were changed before the update, any slave node that is updated as a result of the update will inherit the new property values from the master node. This could cause one slave node to be out of sync with other slave nodes.

Note: Slave nodes that are currently running are not updated. To update slave nodes that are running, you must stop the slave nodes and follow the procedure above for updating a deployed Cluster. It is not possible to update the master node using the **Update Deployed Cluster...** command. For information on administering running projects, see the *BusinessWare Administration Guide* and the *BusinessWare Administration Help*.

DELETING CLUSTERS

Deleting a cluster returns the master node to an unclustered Integration Server. All partitioned components remained partitioned on the Integration Server.

To delete a cluster:

1. Open a deployment configuration and Right-click on a cluster in the Editor.
2. Click **Delete**.
3. Refresh the deployment configuration.

Note: Deleting a parent object also deletes all objects partitioned under the parent object. Special attention is needed when unpartitioning master nodes. Clusters associated with master nodes will be lost if the master node is unpartitioned.

DEBUGGING AND ANIMATING CLUSTERS

Debugging and animating load balanced projects is handled in the same manner as a non-load balanced project. Debugging and animation are not supported for slave nodes.

For more information on debugging a project, see [Chapter 27, “Debugging and Animation”](#).

INJECTING EVENTS

When injecting events into a port on a clustered Integration Server, the specific node to which the event is sent is determined by the cluster's call router. For example, if the cluster's routing policy is set to round robin, you can inject events into a port and the event is sent to the nodes in the cluster based on the round robin policy. It is only possible to set breakpoints on the master node and to inspect events on the master node. See [“Using Custom Routing Policies” on page 21-4](#) for more information on call routers.

VIEWING LOGS

Log file names are automatically generated and based on the master node's log file name and the slave node name. The log file location is the same as the location specified for the master node's log file.

To view the log file for an Integration Server that belongs to a cluster:

1. Select **View > Logs**. The Configure Log View dialog opens.
2. Select **Deployment** in the Choose log from drop-down list.
3. Click **Browse...**.
4. Navigate to the log file and click **OK**.
5. In the Configure Log View dialog, click **OK**.

ADMINISTERING CLUSTERS

Slave nodes are deployed similarly to Integration Servers. Like any Integration Server, slave nodes can be administered using Web Admin and the vtadmin command-line tool. For more information on Web Admin and vtadmin, see the *BusinessWare Administration Help* and the *BusinessWare Administration Guide*.

The following sections provide information on administering clusters.

SHARING CLUSTERS

Clusters created in one project can be used to run components defined in a dependent project. If you have configured a cluster in a library project, you can add the library project to the main project.

To use a cluster from a library project:

1. Add the library project to your project. See “[Using Project Modules](#)” on page 3-16 for more information on project dependencies.
2. Refresh the deployment configuration containing the cluster. This displays the master nodes from the dependent project under the desired BusinessWare Server.
3. Partition the project components to the master node.
4. Deploy the project.
5. You can inspect the shared cluster attributes by opening the deployment configuration in the dependent project.

ERROR HANDLING

Synchronous CORBA or RMI calls on ports or proxies may throw exceptions. The runtime environment never retries a synchronous call automatically; it is the responsibility of the caller to retry the operation if it is safe to do so.

LOAD BALANCING

Error Handling

If a server crashes, is shut down, or becomes unreachable, calls to this server will fail. The client-side clustering runtime environment detects the specific exceptions indicating these conditions and will not route any more calls to the server hosting that port or proxy instance until a periodic check on the status of the server indicates that it is running again.

Developers and business analysts use the BME to create complete BusinessWare solutions. These solutions require the services of the BusinessWare Runtime Environment. Moving an application from the BusinessWare Modeling Environment to the runtime environment and ensuring that it will perform as expected is the process of *deployment*. Key BusinessWare runtime concepts are summarized in [Chapter 24, “BusinessWare Runtime Architecture.”](#)

Topics include:

- [Introduction to Deployment](#)
- [Deployment Process](#)
- [Partitioning Components and Resources](#)
- [Unpartitioning](#)
- [Deploying a Partitioned Project](#)
- [Process Query Deployment](#)
- [Executable Project Deployment](#)
- [Role Deployment](#)
- [Undeploying a Project](#)

INTRODUCTION TO DEPLOYMENT

Deployment is the process by which a BusinessWare project is partitioned and packaged to be able to run in the BusinessWare Runtime Environment. When you deploy a project, all of the information required to run the project is stored in the directory server; then when the project is started, the runtime environment reads all of the necessary information from the directory server.

You can deploy a project directly to a directory server, to a JAR file (called a project module), or to both. Deploying to a JAR file allows you to set up the deployment without being connected to the network. With a deployed project module, you can automate deployment for a later time. It is also necessary when publishing to the design repository or for creating a reusable, parameterized project. For more information on project packaging, see [Chapter 23, “Project Packaging”](#). For more information on the Design Repository, see [Chapter 4, “Design Repository.”](#)

Note: Channels and Integration Servers are global resources that share a common namespace on a directory server and should be named uniquely.

The Deployment Editor, shown in [Figure 22-1](#), is a workspace within the BME that enables you to deploy a BusinessWare project as a fully functioning BusinessWare application.

Note: Dependent projects must be deployed separately. They can be deployed using the BME or the vtadmin command-line tool.

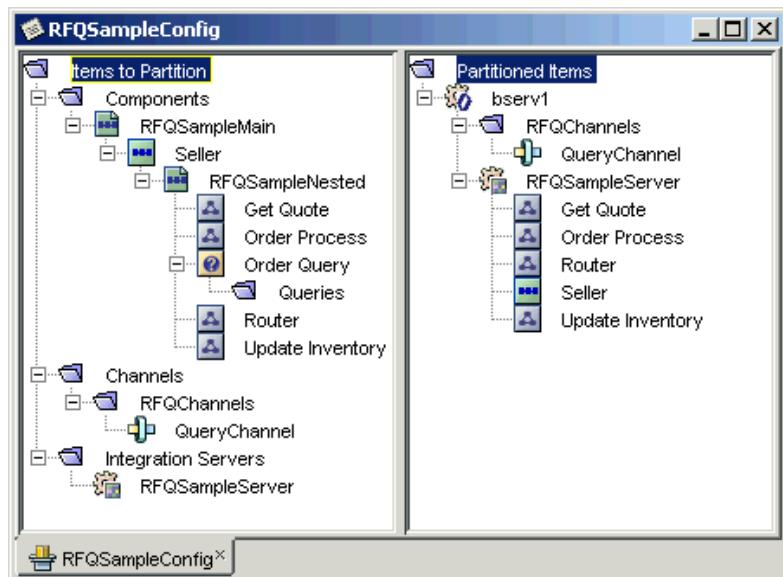


Figure 22-1 BME Deployment Editor

Partitioning involves specifying the servers on which BusinessWare components will run and placing the components on those servers.

The Deployment Editor is used to assign components and resources to specific physical servers and host machines. Partitioning can be as simple as automatically allowing the system to place everything in a single server environment, or can be complex, requiring a collective knowledge of:

- Directory services
- BusinessWare Server installation
- Hostnames
- Network topology
- Other details of the production environment

DIRECTORY SERVER AND DEPLOYMENT

A directory server plays a key role in the BusinessWare Runtime Environment. The directory server contains all deployed projects. During the installation process, the BusinessWare schema and several namespace objects are installed into the directory server.

Schema and Namespace in the Directory Server

[Figure 22-2](#) shows a directory server as viewed through a simple client browser tool. This directory server has already been configured with the necessary BusinessWare objects, specifically, the BusinessWare directory server schema (`bwschema.xml`) and the BusinessWare namespace (`bwnamespace.xml`).

Note: You cannot view the schema using only a browser tool. You must use the administration tool of the directory server itself.

As shown in [Figure 22-2](#), the BusinessWare root (`bw_root` in the `VTPARAMS` file) in this directory server is named `BusWareRoot`. Under this root entry, the namespace consists of folder objects for projects, servers, and types.

See the *BusinessWare Administration Guide* for more information on directory servers.

DEPLOYING PROJECTS

Introduction to Deployment

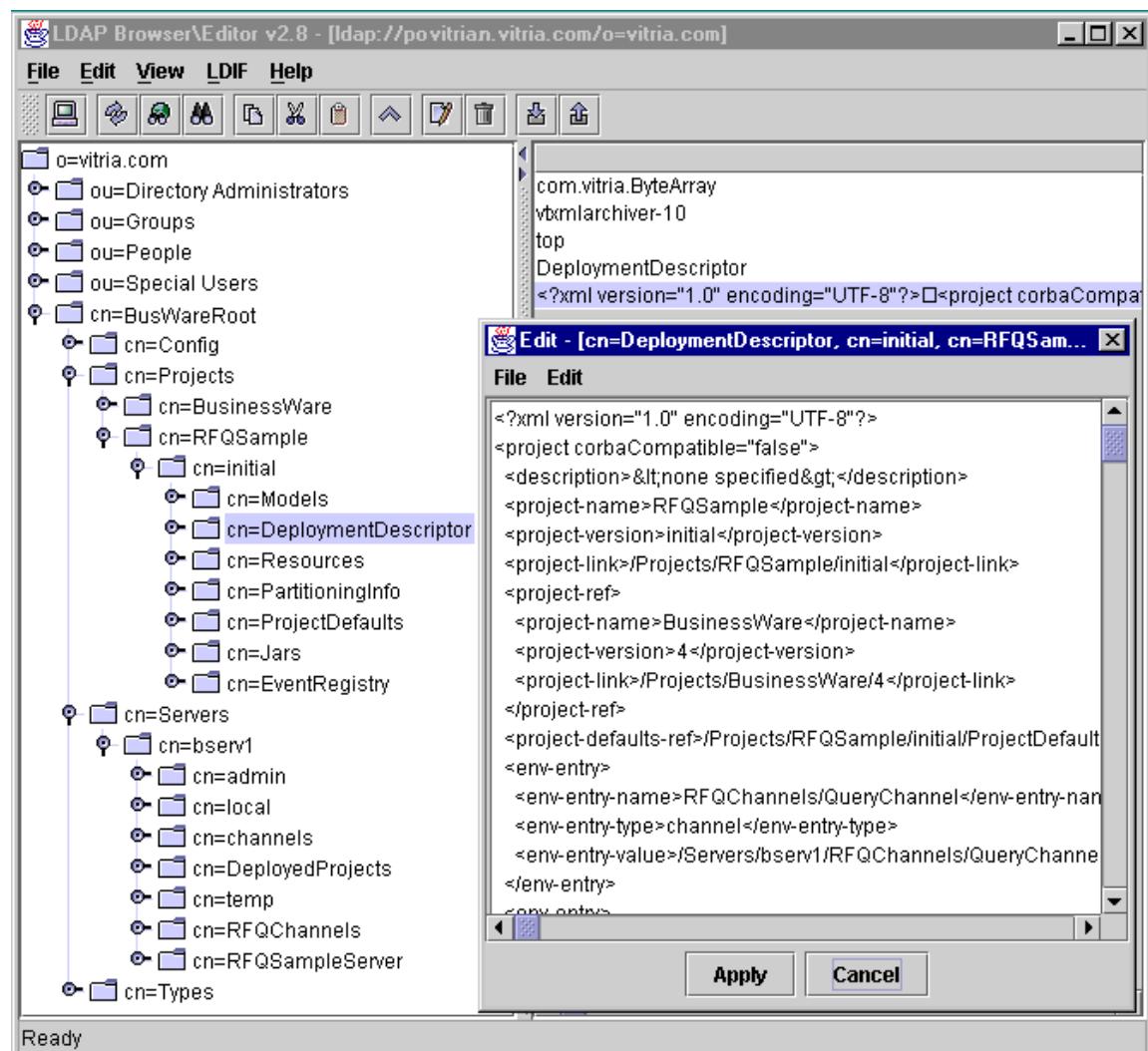


Figure 22-2 Deployed Projects and Servers in the Directory Server

DEPLOYMENT PROCESS

This section discusses how to partition and deploy a BusinessWare project.

Deployment is the process by which a BusinessWare project is partitioned and packaged to be able to run in a distributed environment. Deployment includes the following tasks:

- **Partitioning**—specifying the servers on which BusinessWare components run.
- **Deploying**—installing BusinessWare executable files onto the distributed systems that will run the BusinessWare solution.

BEFORE YOU BEGIN

During installation, the BusinessWare installer prompts you for data necessary to deploy projects in the specified development environment. The data is typically written to a file (`installdir\VTPARAMS`). If your production environment differs, you may need to reset some of the following entries:

- Name and location of the directory server that contains the BusinessWare namespace.
- Tip:** If you installed all BusinessWare software on a development machine, the VTPARAMS file (`installdir\VTPARAMS`) should include the name of the directory server and the BusinessWare root (`bw_root`) location.
- Name and password of an account in the `<BusinessWare/initial>` BusinessWare System Administrators group in the directory server directory.
 - Hostnames of the BusinessWare Servers.

Tip: If you installed all BusinessWare software on a single development machine, the hostname typically is the name of the machine.

IMPORTANT: You also must ensure access permissions have been set up properly in the directory server. See the *BusinessWare Security Guide* for more information.

In production environments, your Integration Server will be configured to use a database for persistence. A database resource will need to be configured with the database server, user, and password information before you deploy your project. You should gather this information before proceeding with deployment.

The BusinessWare schema, namespace, and BusinessWare project must already be installed in the directory server. For information about installing these items, see the *BusinessWare Administration Guide*.

Prior to deployment, make sure you have configured all of the necessary objects for your project as described in previous chapters. Deployment will package these objects for runtime use.

DEPLOYMENT OPTIONS

The deployment process begins after the modeling components and resources have been created and the project has been built with no errors. You need at least one Integration Server if you have any components in your root integration model. See [Chapter 6, “Integration Model Basics”](#) for a discussion of building projects.

Deploying a project requires a deployment configuration to contain customized partitioning information about the project.

There are three ways to deploy a project:

- Auto-Deploy—When you auto-deploy a project (select **Project > Start** with the **Auto-Deploy** option box checked), BusinessWare creates a deployment configuration (and Integration Server, if needed), partitions the project components, and automatically starts running the project.

Tip: You can also deploy a project by selecting **Project > Deploy**. In this case, BusinessWare automatically creates a deployment configuration (and Integration Server, if needed) and partitions the project components. It does not automatically start running the project.

- Auto-Partition—if you already have a deployment configuration that you created manually, you can automatically partition servers, components, and resources from the Deployment Editor (by right-clicking **Items to Partition** and selecting **Auto-Partition**). After using the auto-partition option, you must manually deploy the project.
- Manually Deploy—Create a deployment configuration, manually partition the project components, and then deploy the project to a directory server and/or file.

Note: If you deploy only to a file, you also need to deploy the file to the directory server using the vtadmin command before running the project.

Also, make sure the **Enable Debugging and Animation** option for the Integration Server is set to False if you are deploying to a file to give to someone else.

Note: You can also use the Scriptable BME, a command-line tool, to automate the build and deployment process during development. For more information, see the *BusinessWare Administration Guide*.

The sections that follow outline each phase of the deployment process.

CREATING A DEPLOYMENT CONFIGURATION

A given project can have several different deployment configurations, each of which prepares it for a different physical configuration. A deployment configuration contains the customized partitioning information for your project. You must create a deployment configuration before you can auto-partition or manually deploy a project.

Note: Typically, moving a project to a different physical location requires changes in property values and the physical BusinessWare Server names. It is likely that partitioning choices across Integration Servers will remain the same. Therefore, you should consider using the project parameterization features instead of creating additional deployment configurations. For more information on project parameters, see [Chapter 23, “Project Packaging.”](#)

To create a deployment configuration:

1. In the Explorer window, right-click on the project root and select **New**, or go to the **File** menu and select **New**.
2. Select **DeploymentConfiguration**.
3. Name the new deployment configuration.
4. Click **Finish**.

To partition your project for deployment, you use the Deployment Editor. You can access the Deployment Editor by double-clicking on the deployment configuration or right-clicking and selecting **Open**.

The Deployment Editor displays all partitionable items in the project and Integration Server resources available in dependent projects. You can perform the actions listed in [Table 22-1](#) from the Deployment Editor.

Table 22-1 Deployment Actions

Action	Description
Add BusinessWare Server	Adds a BusinessWare Server as a top-level node in the Partitioned Items pane, enabling you to partition BusinessWare components onto the server.
Auto-Partition	Automatically partitions the current configuration.

Table 22-1 Deployment Actions (Continued)

Action	Description
Refresh	Synchronizes the deployment configuration with the current state of the design objects in the project.
Validate	Compiles and validates the current configuration and displays errors in the Output window. Note: This command functions properly only if you have previously issued the Project > Build command.
Deploy	Deploys the current configuration by placing runtime objects into the directory server and/or file, and optionally the design repository.
Undeploy	Undeploys the current configuration by removing runtime objects from the directory server.

These actions are discussed in the following sections and in the *BME Help*.

PARTITIONING COMPONENTS AND RESOURCES

Partitioning involves specifying the servers on which BusinessWare components will run and placing the components on those servers. You must partition all components that are partitionable. The Deployment Editor also enables you to automatically partition all components.

Note: Components in an integration model can be partitioned anywhere irrespective of the nested integration model or parent integration model to which they belong.

In the Deployment Editor, the **Items to Partition** pane contains specific project components that can be or have been partitioned, including:

- **Integration Servers**—partitioned under an installed BusinessWare Server.
- **Components**—partitioned under a partitioned Integration Server.
- **Channels**—partitioned under an installed BusinessWare Server.
- **Queues**—partitioned under an installed BusinessWare Server.

Note: Although they appear in the **Items to Partition** pane, process query components are not partitionable. For more information, see “[Process Query Deployment](#)” on page 22-17.

SPECIFYING THE DEFAULT BUSINESSWARE SERVER

The default BusinessWare Server is used when you auto-partition your project. It is initially set to bserv1. You can change the default BusinessWare Server to be used during deployment using the Options tool. For example, if you have multiple BusinessWare Servers on multiple machines, you may want to change the default BusinessWare Server.

To set the default BusinessWare Server:

1. Select **Tools > Options**.
2. Click on **BusinessWare Options**.
3. Enter the name of the server in the **Default BusinessWare Server** field.

Note: You can also configure the design repository from here. For more information, see [Chapter 4, “Design Repository.”](#)

ADDING A BUSINESSWARE SERVER

The BME provides a default BusinessWare Server to which components are partitioned when you auto-partition a project. See [“Specifying the Default BusinessWare Server” on page 22-9](#) for information on setting the default BusinessWare Server. You also may add more BusinessWare Servers and partition components to those servers instead of the default BusinessWare Server.

To partition project components on a BusinessWare Server other than the default BusinessWare Server:

1. In the Deployment Editor, right-click on **Partitioned Items**.
2. Select **Add BusinessWare Server**.
3. Click on the **Servers** node.
4. Select a BusinessWare Server from the list of servers.
5. Click **OK**.

Note: The Choose BusinessWare Server dialog box lists BusinessWare Servers that exist in the directory server specified in your VTPARAMS file.

Tip: You can also right-click on an existing BusinessWare Server node to rename it (to specify a different server) or delete it. If a BusinessWare Server has an Integration Server from a dependent project, then you can't rename it.

AUTO-PARTITION OR MANUALLY PARTITION

You can partition your project in one of two ways:

- Auto-Partition
- Manually Partition

To auto-partition:

1. In the Deployment Editor, right-click **Partitioned Items**.
2. Select **Auto-Partition** from the shortcut menu. BusinessWare automatically partitions your project.

If there is no Integration server in the current project, but the dependent project has one or more servers then the components will be partitioned to the first shared server.

To manually partition:

1. In the Deployment Editor:
 - a. Select a partitionable project component.
 - b. Right-click and select **Copy**.
 - c. Right-click on a valid position in the right pane of the Deployment window and select **Paste**. If the component cannot be partitioned, the paste command is disabled. If an item is already partitioned, nothing happens when you select paste.
2. Repeat this process until all partitionable project elements have been partitioned.

[Figure 22-3](#) shows the Deployment Editor.

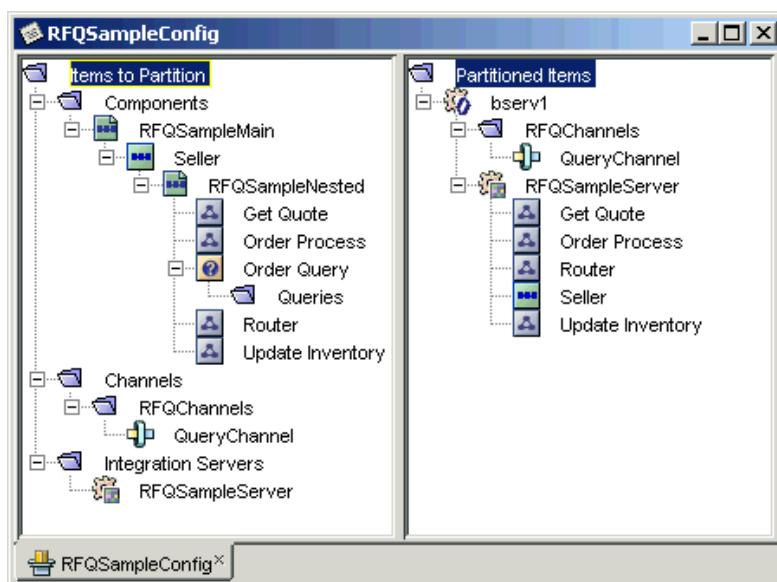


Figure 22-3 Partitioning a Project

Note: Load balancing requires specific partitioning steps as described in Chapter 21, “Load Balancing.”

UNPARTITIONING

To unpartition an object:

1. Right-click on an object in the Partitioned Items tree.
2. Select **Delete**.
3. In the Confirm Object Deletion window, click **Yes**.

or

1. Select an object in the Partitioned Items tree.
2. Click **Delete**.
3. In the Confirm Object Deletion window, click **Yes**.

Note: Deleting a parent object also deletes all objects partitioned under the parent object.

Special attention is needed when unpartitioning the following items:

- Process components with associated process queries—changes to customized queries will be lost if the process component is unpartitioned
- Master nodes—clusters associated with master nodes will be lost if the master node is unpartitioned

DEPLOYING A PARTITIONED PROJECT

Deployment serves two purposes:

- Create an executable project.
- Create a library project.

To create an executable project, you must install the project on the directory server so that the BusinessWare runtime can use it. This operation can be done either by deploying directly to the directory server or by deploying to a file (referred to as a “project JAR file” or “project module”) that can later be deployed into the directory server using `vtadmin`. See “[Deployment Using `vtadmin`](#)” on [page 22-16](#) for more information.

Creating a project on which other projects may depend requires that you:

- Create a project JAR file by selecting **Project > Deploy** and checking the **File** option in the **Destination** area. The file is saved to the location indicated in the File: text box.
- Select **Tools > Install Project Module** to install the project JAR file. This adds your project to the list of projects that may be depended on.
- Deploy the project into the directory server *before* you try to run any projects that depend on it.

For more information on project modules, see [Chapter 3, “Projects.”](#)

Note: All deployment-related activity occurs in `/tmp` on Solaris. The `/tmp` directory is a separate file system that may be mounted on swap. For large applications, you may need to configure additional free area in `/tmp` (see your Solaris documentation). Otherwise, you may encounter exceptions during the deployment phase if there is inadequate space in `/tmp`.

DEPLOYMENT USING THE BME

The BME provides several methods for you to deploy a project. For example:

- From the Main menu, select **Project > Start**.
- From the Main menu, select **Project > Deploy**.

- From the Explorer, right-click on the deployment configuration and select **Deploy** from the shortcut menu.
- From the Deployment Editor, right-click on **Items to Partition** or **Partitioned Items** and select **Deploy** from the shortcut menu.

Using the Project > Start Method

The first method listed above, **Project > Start**, is the easiest way to deploy your project because BusinessWare does everything for you. With this approach, BusinessWare automatically creates a deployment configuration, partitions the project components, and starts the project.

Note: If the project already includes a deployment configuration or Integration Server, auto-deploy uses the existing resources rather than create new ones.

Before you choose this method, however, you must set your default Start settings to include auto-deployment.

To configure auto-deployment:

1. From the Main menu, select **Project > Start Settings**.
2. In the Start Settings dialog box, make sure the **Auto Deploy** option is selected, as shown in [Figure 22-4](#).

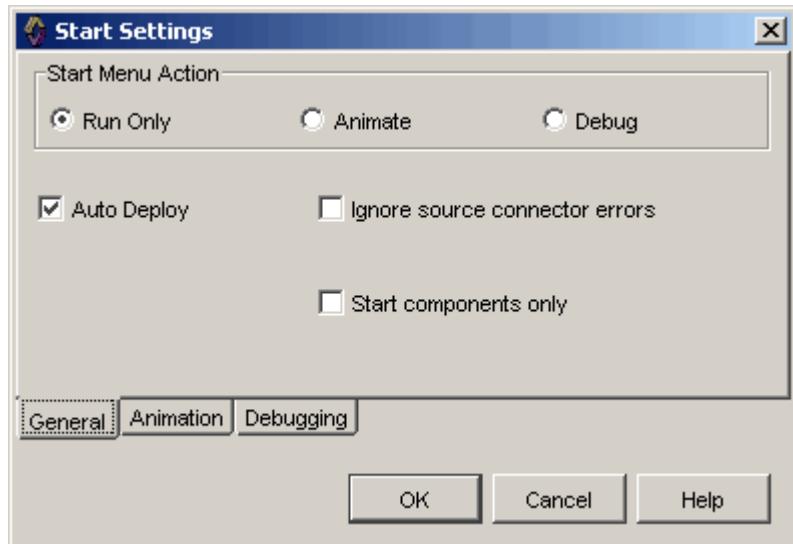


Figure 22-4 Start Settings Dialog Box

3. Click **OK** to save your settings.

Deploying a Partitioned Project

Now when you select **Project > Start**, BusinessWare handles all of the necessary deployment tasks without requiring user intervention.

For more information on the options available from the Start Settings dialog box, see [Chapter 27, “Debugging and Animation.”](#)

Using the Project Deploy Method

If you choose one of the project deploy menu items (such as **Project > Deploy**) rather than **Project > Start** to deploy, you are prompted to validate information about the project deployment, as shown in [Figure 22-5](#).

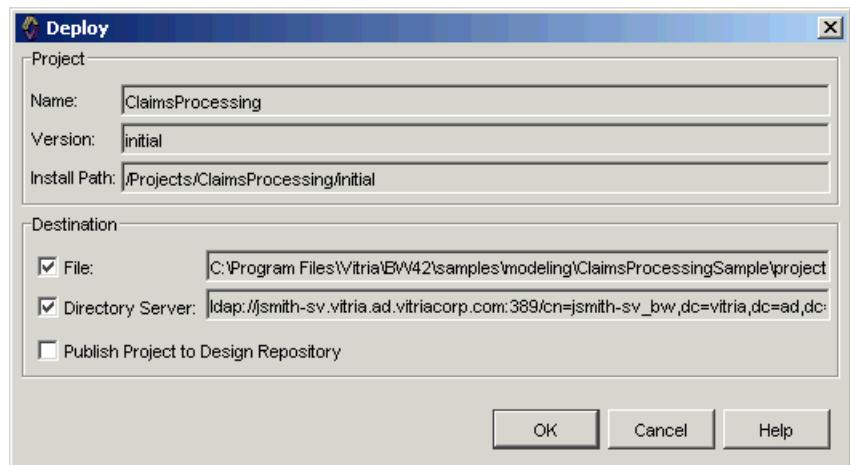


Figure 22-5 Deploy Dialog Box

Note: The directory server information is retrieved from the `bw_root` entry in the VTPARAMS file on disk. If you want to deploy to a different BusinessWare installation, you need to change your VTPARAMS file and restart the BME.

The Deploy dialog box also provides the option to specify whether the project should be published to the design repository. This option is only enabled if you have configured the design repository during installation or through the Options tool (select **Tools > Options**, then select **BusinessWare Options**). For more information, see the [Chapter 4, “Design Repository.”](#)

Note: Publishing to the design repository requires that you deploy to a file.

Refer to the Output window to determine if the deployment/undeployment actions were successful. Messages are provided to confirm validation, deployment, and undeployment.

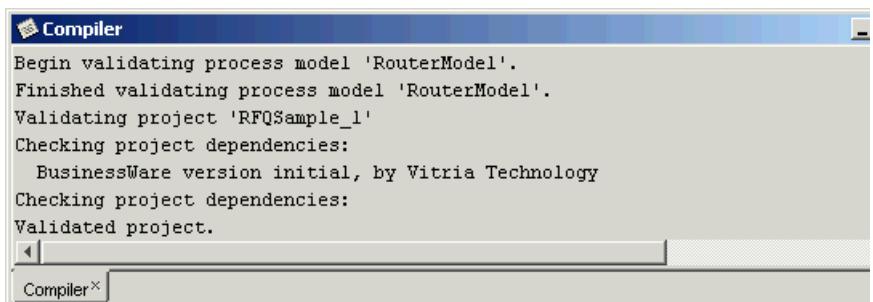


Figure 22-6 Deployment Messages in the Output Window

Stages of Deployment

When you invoke the Deploy command, the BME performs four stages. Each stage must succeed before you can proceed to the next stage.

1. **Predeployment**—the project is compiled and validated. If you see any messages indicating that “predeployment” failed, then you must fix all errors before anything is written to the directory server or to a JAR file. Check the Output window for any errors.
2. **Undeploy the project from the directory server** (if it has already been deployed)—if the project has already been deployed, it must first be undeployed before it can be redeployed. The BME prompts you to choose which channels, queues, and Integration Servers to remove. If you cancel, nothing is removed and your project is not redeployed. If you click **OK**, your project is undeployed and you proceed to the deployment stage (assuming no fatal errors were encountered). See “[Undeploying a Project](#)” on page 22-19 for more information.
3. **Deploy your project to the directory server** (and/or file)—this operation writes the project to the directory server, file, and design repository (as specified in the dialog box). It also creates an informational file with the same base name as the Deployment Configuration object and a .info extension. This file contains:
 - Servlet Information—provides information about the servlets (including the servlet class, servlet name, and servlet parameters) deployed on Web servers associated with Integration Servers. This file provides all of the information needed to invoke these servlets from client Web pages.
 - Port Information—specifies which input ports were created, the type of port, and its path in the directory server. This provides information that is used by projects or external applications invoking components in the project.

- Parameterization Information—specifies the current set of project parameters defined in the project. For more information, see [Chapter 23, “Project Packaging.”](#)

Note: You can only see the parameters in the `.info` file after you deploy the project to a file. The file is not visible in the Explorer. It is located in the same location as the project object. You can inspect it using any text editor.

If this step is successful, then a message displays in the output window stating that deployment succeeded (possibly with warnings). You can also see a similar message in the status bar. Otherwise, you see an error message.

IMPORTANT: The namespace for IDL or DefinedTypes is shared between all BusinessWare projects. If you deploy one project with a particular interface (for example, `mymodule.MyInterface`) and then try to deploy another project to the same directory server using the identically named interface, the system warns you that the interface is already deployed. To use the interface with the second project, you must undeploy the first project and redeploy the second project. If the data is meant to be shared among projects, you should define a separate library project containing the shared types and have both projects that need this data establish a dependency on it.

4. **Undeploy your project** (if an error occurs)—if an error occurred during deployment ([step 3](#)), your project is undeployed (to avoid leaving a project partially deployed and inconsistent). Any channels, queues, and Integration Servers that were created (before the failure) are removed. Channels, queues, and Integration Servers that were *not* created as a result of this deployment attempt are *not* removed.

Tip: If you believe something might be going wrong and you would like to see more output, look at the deployment log file, `installdir\logs\deployment.log`.

Note: When a project is initially deployed, a change log is created. Any subsequent changes made to the project using `vtadmin` or `webadmin` are recorded in the change log. For more information, see the *BusinessWare Administration Guide*.

DEPLOYMENT USING `vtadmin`

During the BME deployment, you can deploy to a JAR file. Later, you can use the `vtadmin` command-line tool to deploy the contents of the JAR file to the directory server for the runtime to use.

There are a number of command-line options that you can append to the end of the command-line arguments. For a list of options, you can see the `vtadmin` help message by executing `vtadmin` without any arguments. For more information about `vtadmin`, see the *BusinessWare Administration Guide*.

Deploying using `vtadmin` has the same phases as deploying using the BME (except for predeployment):

1. Undeploy the project (if it has already been deployed).
2. Deploy your project to the directory server.
3. If an error occurs, undeploy your project.

As with the BME, `vtadmin` provides you with either error messages or a success message (and possibly warning messages). The messages indicate whether your deployment was successful.

Note: The directory server information is retrieved from the `bw_root` entry in the VTPARAMS file on disk. If you want to deploy to a different BusinessWare installation, you need to change your VTPARAMS file before running the `vtadmin` command.

IMPORTANT: BusinessWare treats all files with the extension .cfg as deployment configuration files. If there is a file that was created externally that has a .cfg extension, BusinessWare treats these files as invalid configuration files. When this happens, the deployment configuration object in the Explorer appears with an “x” overlaying the object. If you attempt to open the deployment configuration object, an error message appears.

This also occurs when BusinessWare cannot reconstruct successfully a deployment configuration from a saved BusinessWare project.

All post-deployment changes to a project are logged in the Change log. For more information, see the *BusinessWare Administration Guide*.

PROCESS QUERY DEPLOYMENT

Although they appear in the **Items to Partition** pane, process query components are not partitionable.

When you create a deployment configuration, all of the components in the integration models are added to the **Items to Partition** pane, including process query components. Each process query component is connected to a process component via a query port (for an example, see the nested integration model in the RFQSample project). A process query component itself is not partitionable. Instead, the partitioning is inferred from the process component to which it is wired. However, process query components do have an additional configuration capability at deployment time.

Note that the queries themselves only appear in the **Items to Partition** pane after you partition the associated process component.

For more information about process queries in general or about customizing process queries during deployment, see [Chapter 17, “Process Query Models.”](#)

EXECUTABLE PROJECT DEPLOYMENT

When deploying an executable project, an internal queue called “ResponseQueue” is created. ResponseQueue is used to dispatch responses from service provider projects to the current project. One service queue is created for each Integration Server in the current project. For more information on service queues, see [“Remote Services” on page 28-12.](#)

The Task Manager is implemented as a Remote Service and communicates with application projects using the service queue. It contains the event data signaling the completion of activities. If you are using activity states in a project, you need to be aware of the existence of this queue.

All BusinessWare Servers must be running when an executable project is deployed to a directory server. If one or more of the BusinessWare Servers to which the current project is partitioned is not running, an error message indicates that the queue cannot be created.

When undeploying an executable project, the service queues appear in the undeploy dialog with other user-created queues. Before undeploying a service queue, you should determine if the events in the queue (responses from the service provider) are important. As stated above, if you are using activity states in your project, you may have important data on your service queue that should not be deleted.

ROLE DEPLOYMENT

When a project containing a Role resource is deployed, the corresponding role group is validated. Validation of the role group causes the role group to be created in the directory server if it does not already exist. The user deploying the role must have the correct deployment permissions. If the user does not have the correct permissions, the Output window displays a warning and a warning is logged in deployment.log. These roles must be created manually using the directory server tool for the project to work correctly. The deployment.log contains the list of roles that were not created.

Note: Even though deploying a project with a Role resource creates the role group, undeploying the project does not remove the role group from the directory server. In addition, removing a Role resource from a project and redeploying the project to the directory server does not remove the role group. You must manually remove unused roles from the directory server.

UNDEPLOYING A PROJECT

Undeployment is the process of removing a project's items from the directory server. When you select **Undeploy** from the menu, you are prompted to specify the shared resources you want to remove from the directory server, as shown in Figure 22-7.

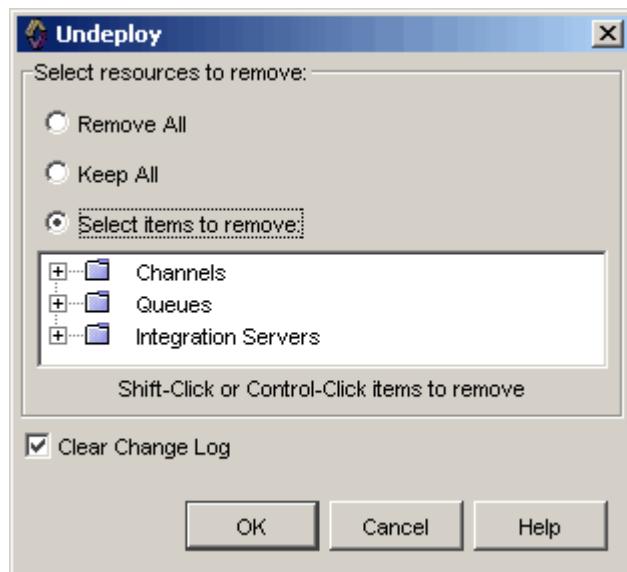


Figure 22-7 Undeploy Dialog Box

Note: You can choose to remove all resources, keep all resources, or remove only specific resources. To remove a group of resources from the directory server, select the resource group (Channels, Queues, or Integration Servers) and then click **OK**. To remove a particular resource from within a group, double-click on the resource group, select the resource you want to remove, and then click **OK**. To clear the Change Log, select the Clear Change Log box. For more information on the Change Log, see the *BusinessWare Administration Guide*.

IMPORTANT: If the project is started, undeployment first stops the project and then removes it. If the project stop operation fails, then the project cannot be removed. If project stop fails, you should try to manually (using vtadmin or the Web Administration tool) stop each Integration Server on which the project is running, and then undeploy the project.

CHANNELS AND QUEUES

You should remove channels and queues to clean up resources created by the project except when:

- The channel or queue is shared by other projects. However, in this case, all projects sharing the same channel or queue should share it by using a common project dependency so that only one project is responsible for creating the channel or queue.
- There are events on the channel or queue that should be preserved during a redeploy of the project.
- The channel or queue has been configured using the Web Administration tool. (If you configure something from the Web Administration tool and then undeploy it, all of the customizations made using the tool will be lost.) For information on backing up and restoring changes made to projects with the Web Administration tool, see the *BusinessWare Administration Guide*.

Note: Undeployment will remove channels and queues that the project tried to create, but it will not remove channels and queues from other projects on which your project depends.

INTEGRATION SERVERS

You can choose whether to remove Integration Servers that the project tried to create. As with channels, you should generally remove Integration Servers to clean up resources created by the project except when:

- The Integration Server is shared by other projects. All projects sharing the same Integration Server should share it by using a common project dependency so that only one project is responsible for creating the Integration Server.
- The Integration Server has been configured using the Web Administration tool. (If you configure something from the Web Administration tool and then undeploy it, all of the customizations made using the tool will be lost.) For information on backing up and restoring changes made to projects with the Web Administration tool, see the *BusinessWare Administration Guide*.

Note: The Undeploy dialog indicates when an Integration server is the master node of a cluster. If you decide not to undeploy the master node, the cluster and slave nodes are undeployed as a part of the project undeploy operation, but the Integration Server that was acting as the master node remains in the directory server.

IMPORTANT: Even if you specify that you want to remove an Integration Server, BusinessWare will *not* remove it if the server is still running. Typically, this happens when there are other projects still running on the server. In this case, BusinessWare posts a warning message specifying which project is still running on the server.

DEPLOYING PROJECTS

Undeploying a Project

This chapter describes how BusinessWare supports custom project installations. Topics include:

- [Introduction to Project Packaging](#)
- [Specifying Project Parameters](#)
- [Distributing a Project](#)
- [Performing a Custom Project Installation](#)

INTRODUCTION TO PROJECT PACKAGING

Project packaging is an advanced deployment feature that enables environment-specific installation of BusinessWare projects. It involves two primary tasks:

- Specifying project parameters
- Performing a custom project installation

These tasks are performed by you, the developer or modeler of the project, and the users who receive and deploy the project. During development, you can specify parameters as placeholders for specific property values that can later be defined by the project's end users. This type of project is called a *parameterized project*.

All projects are capable of using parameters. A parameterized project lets users specify the properties unique to their environment so they can perform a custom project installation. This process of custom configuration and installation is called *project installation*.

Project packaging, or parameterization, is most useful when you want to model an environment in which some details are unknown at design time. For example, it is common to interact with a database at some point during business processing. You can model a database, but you may not know the password to the specific database that will eventually be available in the runtime environment. In this case, you can use a project parameter to defer this type of decision. By creating a project parameter for the password and mapping this parameter to a database resource's User Password property, the password is isolated along with other parameters, thus enabling the project to be customized for deployment at specific environments without the need to remodel or change anything in the design-time environment (that is, the BME).

Parameters can be used for any property. They are commonly used to specify:

- Passwords
- Channel names
- Log levels
- File names
- Port numbers
- Role mappings
- Time values
- BusinessWare Server name
- Batch sizes

PROJECT PARAMETER SETTINGS

To access or specify project parameters, expand the Project object in the Explorer. The project parameter settings are contained in the Project Parameters node (see Figure 23-1).

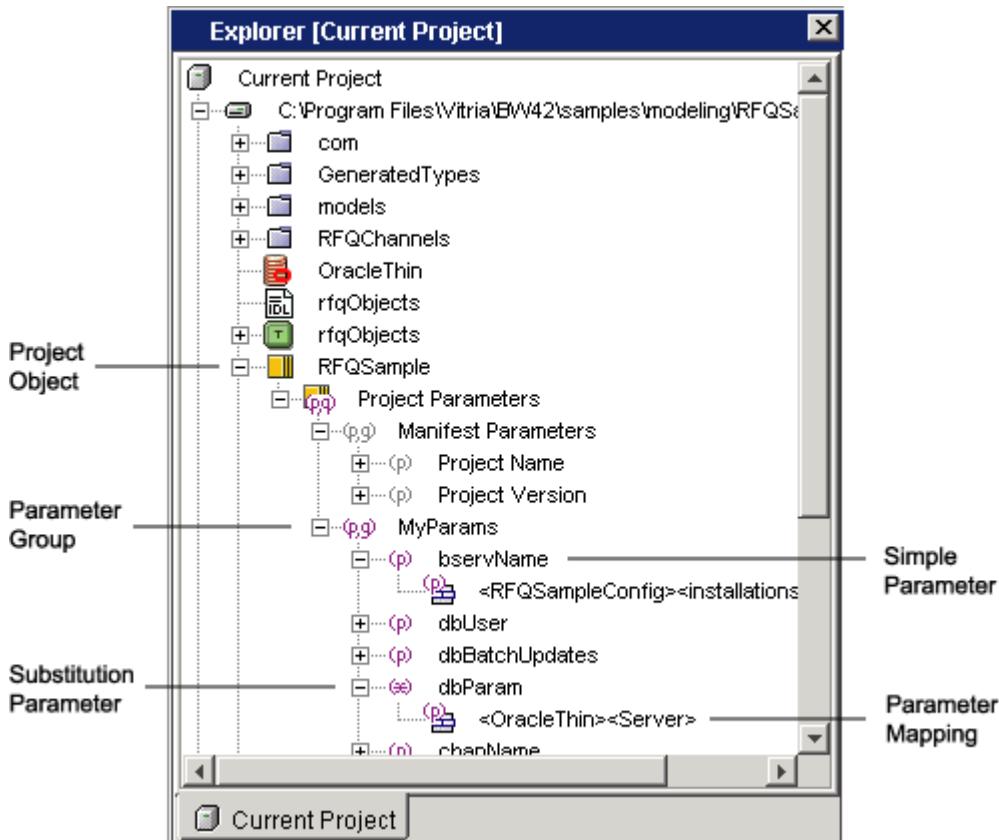


Figure 23-1 Project Parameters Node

The Project Parameters node (and all of its children) is stored along with the project and is exported/imported along with other project elements.

The Project Parameters node can contain three basic levels of information—parameter groups, project parameters, and the actual parameter mappings. As the names imply, a parameter group is a collection of project parameters. A parameter is used as a placeholder for an installation-specific value, which is identified by the parameter mapping.

Predefined parameters are stored in the Manifest Parameters group. All projects have parameters for Project Name and Project Version that are used to identify the project in the directory server. You cannot directly modify these parameters. (See [Chapter 3, “Projects”](#) for information on modifying a project’s name and version.) Manifest parameters enable you to make abstract references to the project name and version in the context of other parameters and properties. They take concrete values at the time of deployment.

You can create your own parameters within your own user-defined parameter groups. User-defined parameters consist of:

- **Simple parameters**—basic user-defined parameters that are defined by a single property value.
- **Substitution parameters**—user-defined String parameters that hold the combined value of multiple other parameters. The value of a substitution parameter can be a mix of constant strings and parameter specifications.

Each parameter can be mapped to one or more properties. During deployment, the value of the parameter is used to define the value of mapped properties.

PROPERTIES

As parameters are selected, the BME’s property window is updated to display the relevant properties for individual parameters. [Table 23-1](#) describes these properties.

Table 23-1 Parameter-Specific Properties

Property	Description
Name	The name by which the parameter may be referred to in substitution parameters and the .info file.
Value	The property’s value. When the Value property of a simple parameter is not mapped, it defaults to a string. After it is mapped, it takes the type of the first property to which it is mapped.

SPECIFYING PROJECT PARAMETERS

During project development, you can:

- Create parameter groups
- Create simple parameters and substitution parameters inside a group

- Map the parameters to properties in the project
- Set parameter values

These actions are described in the remainder of this section and in the *BME Help*.

Through parameterization, you can enable other users to define (set/fetch) parameter values during project installation. This action is described later in this chapter and in the *BME Help*.

CREATING PARAMETER GROUPS

By creating parameter groups, you can organize or categorize related parameters. You must create at least one parameter group before you can add parameters.

To add a parameter group:

1. From the BusinessWare Explorer, right-click on Project Parameters and select **New Parameter Group**. The New Parameter Group window appears.
2. Enter a name for the parameter group, then click **OK**. BusinessWare adds the group to the end of the group list in the Project Parameters node.

When you right-click on a parameter group, the shortcut menu displays the following options:

Table 23-2 Parameter Group Shortcut Menu

Menu Option	Description
New Parameter	Create a simple parameter inside the selected group.
New Substitution Parameter	Create a substitution parameter inside the selected group.
Move Up	Move the parameter group up to reorganize the groups within the Project Parameters node.
Move Down	Move the parameter group down to reorganize the groups within the Project Parameters node.
Delete	Remove the parameter group from the project.
Rename	Rename the parameter group.
Properties	View and edit the properties of the parameter group. For parameter groups, Name is the only property.

CREATING SIMPLE PARAMETERS

You can set up simple parameters at any time prior to distribution. For example:

- Before commencing project development—if you know what properties will require customization, you can specify parameters prior to creating project elements. Then, when you need a parameter during modeling, you can map the element's property to the parameter you created earlier.
- During project development—if you decide that the value for a specific field cannot be known during development, you can add a simple parameter in the context of your current task.

You can add parameters to any parameter group you created. You cannot add parameters to the manifest group, which contains predefined parameters.

To add a simple parameter:

1. From the Project Parameters node in the BusinessWare Explorer, right-click on the parameter group for which you want to add a parameter, then click **New Parameter**. The New Parameter window appears.
2. Enter the name of the parameter you want to add, then click **OK**. BusinessWare adds the parameter to the parameter group node.

When you right-click on a parameter, the shortcut menu displays the following options:

Table 23-3 Simple Parameter Shortcut Menu

Menu Option	Description
New Mapping	Select the property to which the parameter should apply.
Move Up	Move the parameter up to reorganize the parameters within the current parameter group. You cannot move parameters between groups (that is, move a parameter from one group to another).
Move Down	Move the parameter down to reorganize the parameters within the current parameter group. You cannot move parameters between groups (that is, move a parameter from one group to another).
Delete	Remove the parameter from the project.
Rename	Rename the parameter.
Properties	View and edit the values of the selected parameter.

MAPPING PARAMETERS

When you map a parameter, you specify the property to which the parameter should apply. You can map a parameter to multiple properties, but the properties must be of the same type. For example, you cannot map a parameter first to a property of type Boolean and then to a property of type String. In addition, you can map parameters only to properties in the same project.

To apply a parameter to a specific property in the current project:

1. From the Project Parameters node in the Explorer, right-click on the parameter for which you want to apply a mapping, then click **New Mapping**. The New Mapping window appears.

Note: To set the parameter type, BusinessWare uses the type of the first mapping in the current list of mappings associated with the parameter. That is, for the initial mapping of a parameter, you can select any property of any type, but any subsequent mappings must match that type.

2. From the list of project elements, navigate to and select the property you want to associate with the parameter, then click **OK**.

When you right-click on a parameter mapping, the shortcut menu displays the following options:

Table 23-4 Parameter Mapping Shortcut Menu

Menu Option	Description
Delete	Remove the mapping from the parameter.
Properties	View the mapping value.

Parameterizing Integration Server Properties

You can parameterize the properties of sub-objects of Integration Servers, such as loggers and RDBMS persistence.

To parameterize a logger in an Integration Server:

1. In the Explorer, select the Integration Server and create a Text File Logger, Binary File Logger, or Channel Logger (or make note of an existing logger).
2. Create a simple parameter.
3. Create a mapping.
 - a. In the New Mapping dialog box, expand the Integration Server's Diagnostic Loggers and expand the logger from step 1.
 - b. Select the property you want to parameterize (for example, File Name).

Note: You cannot parameterize objects that are created dynamically as a result of applying parameter values. In other words, you can only parameterize objects that are available at design time when the parameter is being created. That is why the logger must exist before you can parameterize it.

To parameterize RDBMS persistence for an Integration Server:

1. Set the Integration Server's Persistent Store property to RDBMS, which creates an RDBMS object under the Integration Server.
2. Select the RDBMS object under the Integration Server and set its Database Resource property to a new or existing database resource of the type available in the deployed environment (for example, Oracle or MS SQL Server).
3. Create a simple parameter.
4. Create a mapping.
 - a. In the New Mapping dialog box, expand the database resource you set for the persistent store.
 - b. Select the property you want to parameterize (for example, Server, User Name, or User Password).
5. Repeat step 3 and step 4 to parameterize additional parameters related to the database resource.

Also, you can parameterize which database resource to use, as described below. In this case, you need to create multiple preconfigured database resources in the project that correspond to the databases available in the deployed environment.

To parameterize which database resource to use:

1. Set the Integration Server's Persistent Store property to RDBMS, which creates an RDBMS object under the Integration Server.
2. Select the RDBMS object under the Integration Server and set its Database Resource property to a new or existing database resource of the type available in the deployed environment (for example, Oracle or MS SQL Server).
3. Create a simple parameter.
4. Create a mapping.
 - a. In the New Mapping dialog box, expand the Integration Server and then expand its RDBMS properties.
 - b. Select the **Database Resource** property.

In order to provide flexibility for the Integration Server's persistence, you can optionally:

1. Create another simple parameter.
2. Create a mapping and, in the New Mapping dialog box, expand the Integration Server.
3. Select the Integration Server's **Persistent Store** property.

The optional steps above enable a user who is installing the project to set either the database settings or ignore them by setting the Integration Server to use cache persistence instead. If the Integration Server's persistence were set to cache (rather than RDBMS), then it would be impossible to parameterize the Database Resource property because the RDBMS object would not exist.

Parameterizing Nested Process Model Properties

You can use a combination of project parameters and dynamic nesting of nested states to change the process model associated with a process component. You can associate the process model at runtime as opposed to during deployment by doing the following:

1. Map the process component to a process model that is wired as a Start state, Nested State, and a Terminator state.
2. Create a project parameter that is not mapped to a property.
3. Specify the process model using dynamic nesting code. Right click on the nested state and select **Tools>Dynamic Nesting Code**. In the Dynamic nesting code use the Project Parameter value to set the model name. For more information on getting Project Parameter values at runtime see "[Getting Project Parameters at Runtime](#)" on page 23-11.
4. During deployment, specify the project parameter value created in step 2.

SETTING PARAMETER VALUES

To set a parameter's value, select the parameter from the Project Parameters node and modify the Properties window.

CREATING SUBSTITUTION PARAMETERS

As with simple parameters, you can set up substitution parameters at any time prior to distribution. A substitution parameter is a parameter based on the values of other parameters. That is, you can create individual parameters to represent individual properties and create another parameter to hold the combined value of those properties.

The value for a substitution parameter is a String. The String may contain the names of other parameters delimited with angle brackets; for example:

```
installdir\logs\<GroupName/ParamName>\foo.log
```

where *GroupName* is the name of the parameter group and *ParamName* is the name of the parameter. For example, *installdir\logs\<Manifest Parameters/Project Name>.log* would be a log file for the predefined parameter Project Name.

The actual value of a substitution parameter is computed during deployment. A substitution parameter may be applied to any String property.

Note: You can also use environment variables in substitution parameters (for example, \$VITRIA), although they are not evaluated until runtime.

To create a substitution parameter:

1. From the Project Parameters node in the BusinessWare Explorer, right-click on the parameter group for which you want to specify the substitution, then select **New Substitution Parameter**. The New Substitution Parameter window appears.
2. Enter the name of the parameter you want to use as a substitution, then click **OK**. BusinessWare adds the substitution parameter to the parameter group node.

You can perform the same actions from a substitution parameter as you can from a simple parameter. For a description of the options available on the shortcut menu, see [Table 23-3](#).

To select the parameters whose values you want to include in the substitution parameter:

1. In the Explorer, select the substitution parameter.
2. From the Properties window, launch the Property Editor for the Value property.
3. Select a parameter that you want to use for the substitution parameter, then click **Insert**.

4. After you have inserted all of the parameters you want to use for the substitution parameter, click **OK**.

Tip: In addition to selecting parameters, you can also type text to include in the substitution parameter's value.

GETTING PROJECT PARAMETERS AT RUNTIME

It is possible to get a project parameter value within the Java source code by calling the following method:

```
String getParameter(String parameterName) throws  
ParameterNotFoundException
```

This method is found in the following interface:

```
com.vitria.container.client.ProjectExecutionContext
```

For example,

```
String pValue = ctx.getParameter(parameterName);
```

If the parameter is not found, a `ParameterNotFoundException` exception is thrown.

DISTRIBUTING A PROJECT

BusinessWare validates all projects before deployment. For parameterized projects, validation ensures that the parameters you specified include valid default values.

When a project is ready for distribution, you must deploy the project to a JAR file (for more information, see [Chapter 22, “Deploying Projects”](#)). Along with other project content, this file contains the project parameter mappings that were created. The current default values are listed in the `ProjectName.info` file that is generated along with the project JAR file.

Note: Along with the project JAR file, you may want to supply users with a customized `.info` file that includes additional comments and/or customized parameter values. (Note that the `.info` file is overwritten during every deployment, so the file must be renamed if it is to be saved.) Specifically, you might want to make the following changes to the `.info` file:

- Add or change comments to provide additional instructions for the user, such as what the parameter values should look like.

Performing a Custom Project Installation

- Remove the port and servlet information, which is not relevant for the user's project installation.
- Remove parameters that users will not need to edit, such as Substitution Parameters (whose values are set automatically).
Note: Substitution parameters are automatically commented out in the .info file.
- Also note that the vtadmin command-line tool allows "discovery" of customizable project parameters, in case you decide not to provide a customized .info file with the JAR file.

Users of the project can then deploy the project with different values for the parameters defined. They must modify the .info file to contain the values appropriate for their environment, and use the vtadmin command-line tool to deploy the JAR file with the modified .info file. For more information, see "["Performing a Custom Project Installation" on page 23-12.](#)

Note: Parameters are defined on a project-by-project basis. They cannot span projects and you cannot apply parameters from dependent projects. Parameters are read-only outside of their parent project.

PERFORMING A CUSTOM PROJECT INSTALLATION

To perform a custom project installation, users must complete the following actions:

1. Get the project JAR file from the developer or modeler.
2. Obtain a .info file for the project you are installing. If you did not receive a .info file with your JAR file, you can retrieve it from the JAR file. For more information, see "["Generating a Project Parameter XML File" on page 23-13.](#)

Note: To install a project, you need to specify project parameter values using an XML file named *project.info*, which contains the project parameters for you to set. The developer may have supplied a customized .info file for the project along with the project JAR file. The customized .info file can contain comments or other changes that make it easier for you to specify the parameter values for the project.

3. Specify the values for the project parameters by editing the .info file from [step 2](#). For more information on the contents of the XML, see "["Project Parameter XML Contents" on page 23-13.](#)

4. Invoke the vtadmin command-line tool, passing it the project JAR file and the edited XML file. This installs the project into the BME and deploys the project to the runtime environment. For instructions, see “[Installing a Custom Project](#)” on page 23-13.

GENERATING A PROJECT PARAMETER XML FILE

To read the project and generate XML output that contains the current (default) parameter values, run the following command:

```
vtadmin getinfo project ProjectJarFile
```

The results are written to the command-line output, which you can then copy to a file for editing. For information on the contents of the XML, see “[Project Parameter XML Contents](#).”

Note: The parameters are only visible from the `vtadmin getinfo` call after the project has been deployed successfully to a JAR file.

Project Parameter XML Contents

The XML content contains four sections:

- Parameters—describes all the project parameters and their values. All of the parameters can be modified, including the project name and version. Any parameters that should not be changed can be removed or commented out in the XML.
- Mountpoints—describes the project’s mountpoints. The `exportPath` specifies the path of the mountpoint in the developer’s environment. The `importPath` specifies what the mountpoint path should be in the user’s environment. Users should verify that the `importPath` is correct for any mountpoints that have type “local” or “JAR”. If the `importPath` contains a variable (for example, `$VITRIA`), then users can safely leave it as is.
- Ports—describes the project’s ports. This section is for information purposes only. It is ignored during project installation and can be omitted from the file.
- Servlets—describes the project’s servlets. This section is for information purposes only. It is ignored during project installation and can be omitted from the file.

INSTALLING A CUSTOM PROJECT

After you have modified the parameter values and mountpoints to suit your environment, you need to apply the new values to the project.

Performing a Custom Project Installation

To use the edited XML file, enter the following command:

```
vtadmin [-v] deploy project ProjectJarFile useparams= XMLFile  
[-overwrite]
```

The `-overwrite` parameter must be used if the project already exists in the directory server or the BME. The `-v` option provides status information (for example, whether the project is being imported, built, or deployed). Other options, such as `-removeall` and `-keepall`, are also available. For more information, see the *BusinessWare Administration Guide*.

IMPORTANT: When using `vtadmin` with the `useparams=` flag, neither the JAR file nor the `.info` file (which contains the new parameter values) should be in the directory where the project existed.

If the project exists in the BME and you use the `-overwrite` parameter, the following sequence of events occur when you execute the above command:

1. The project is deleted from the BME, along with the project's filesystems and all of its data. This operation is not reversible, so data should be backed up as needed.
2. The project is imported into the BME using the mountpoints specified in the `.info` file.
3. Project parameter values from the `.info` file are applied.
4. The project is built.
5. If the project already exists in the directory server, it is undeployed.
6. The project is deployed to the directory server.

PART VII: RUNTIME

This part describes the BusinessWare runtime architecture and services. It explains what transactions are and how BusinessWare supports transaction processing. In addition, it describes how BusinessWare components respond to runtime exceptions and how you can modify or extend the default response. It also describes the BME testing and debugging tools and outlines the debugging procedure.

Chapters include:

- [BusinessWare Runtime Architecture](#)
- [Transaction Management](#)
- [Exception Handling](#)
- [Debugging and Animation](#)

PART #: TITLE HERE GOES HERE

This chapter describes how projects developed with the BME are represented as runtime entities, and the runtime services (platform services) that BusinessWare provides for projects.

Topics include:

- [Runtime Architecture](#)
- [Project Startup](#)
- [Activation](#)
- [Runtime Services](#)

RUNTIME ARCHITECTURE

BusinessWare runs and supports your projects as described in this section.

PROCESSES AND HOSTS

If you could see a project deployed for testing and running on your computer, you would see seven processes categorized as illustrated in [Figure 24-1](#).

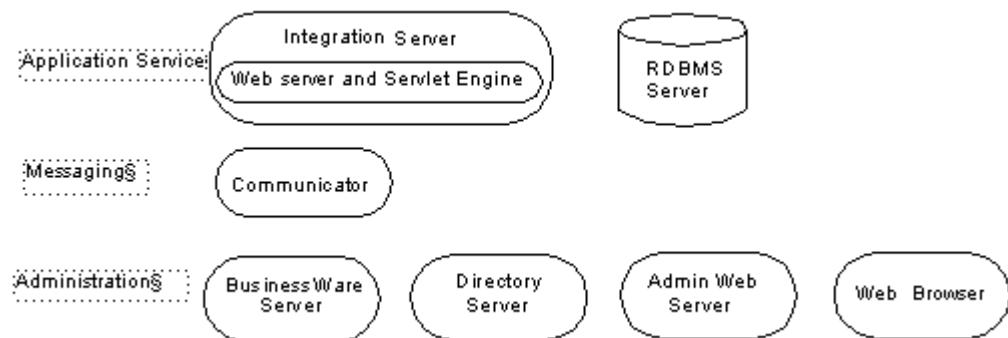


Figure 24-1 Simple Project Test Deployment

These process groups perform the following functions:

- Application service processes:
 - **Integration Server**—starts projects on demand and provides execution environments for projects (described in “[Containers](#)” on page 24-4). The Integration Server hosts the following:
 - **Web server and Servlet Engine**—handles HTTP requests to serve web content defined as HTML pages and serves as an execution environment for servlets and Java Specified Pages (JSP) that make up a web application. (You can use the bundled Web server as a servlet engine alone, if needed.) See “[Web Server and Servlet Engine](#)” on page 24-9 for more information.
 - **Third-party RDBMS server**—manages the database that stores application data and system recovery data.
- Messaging process:
 - **Communicator**—manages channels and queues allowing asynchronous communication via publish/subscribe paradigm. Channel and queue data is stored on disk in a special file.
- Administration processes:
 - **BusinessWare Server (bserv)**—starts the Communicator process and starts Integration Server processes on demand. The BusinessWare Server is often configured to start automatically when its host is started.
 - **Third-party directory server**—manages the directory server repository, which holds project data, such as Java classes and objects, and system configuration data such as a list of Integration Servers.
 - **Admin Web Server**—runs the servlet that executes Web Administration commands, such as starting projects.
 - **Web Browser**—displays the BusinessWare Web Administration user interface.

Production environments often have multiple hosts. To take advantage of the parallelism offered by multi-host environments, the processes shown in [Figure 24-1](#) can be applied to different hosts, subject to these rules:

- There can be numerous BusinessWare hosts in a network. The Web Administration tool provides a single point of administration for all BusinessWare hosts.
- A BusinessWare host can have one or more BusinessWare installations (BusinessWare Server/Communicator pair), although there is typically only one BusinessWare installation per host.
- A BusinessWare host can have numerous Integration Servers.

For example, [Figure 24-2](#) shows how the processing load can be shared by six hosts. In this example, BusinessWare has been installed on three hosts, and the directory server, Web, and RDBMS Servers run on different hosts.

Note: Placing RDBMS servers on the same host as the Integration Servers can yield performance benefits.

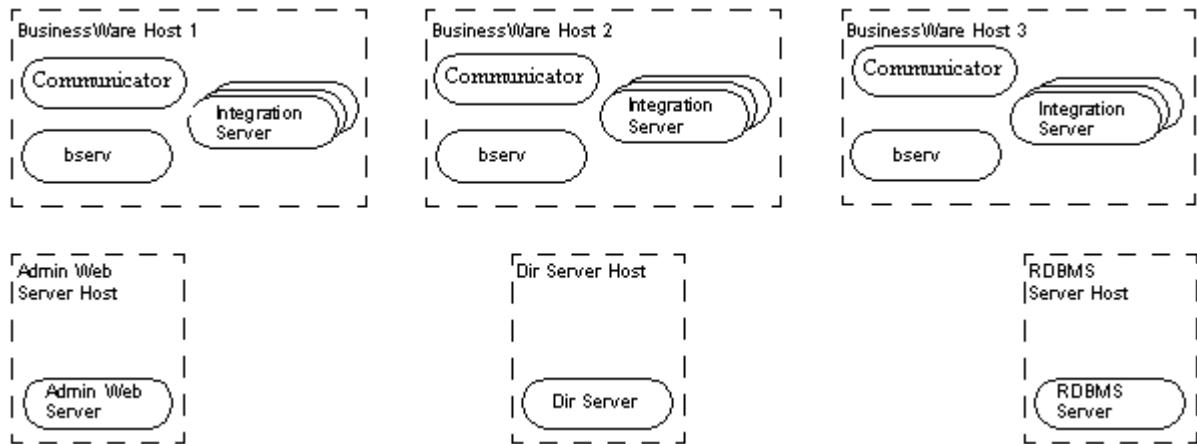


Figure 24-2 Example Multiple Host BusinessWare Configuration

CONTAINERS

A container provides deployment and runtime services for project components (see “[Runtime Services](#)” on page 24-8) and insulates projects from each other.

Projects are loaded and run in containers, as shown in [Figure 24-3](#).

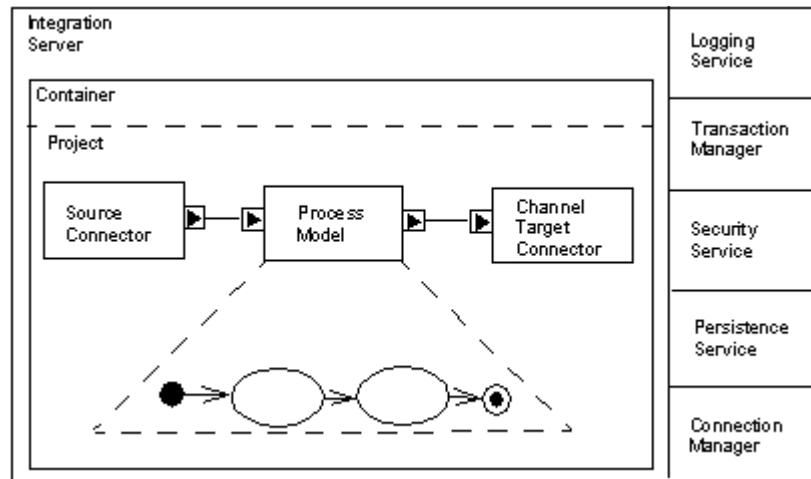


Figure 24-3 Project Running in a Container

Containers exist and run within an Integration Server. This allows the containers to provide additional flexibility for deploying projects. An Integration Server can host multiple containers, but containers cannot be shared among projects. Each project or version of a project is run in its own container. If you have a large project, or one whose components differ in their demand for computing resources, you can improve performance by partitioning the project among multiple Integration Servers running on different hosts as shown in [Figure 24-4](#). Again, the part of the project partitioned on an Integration Server will use its own container.

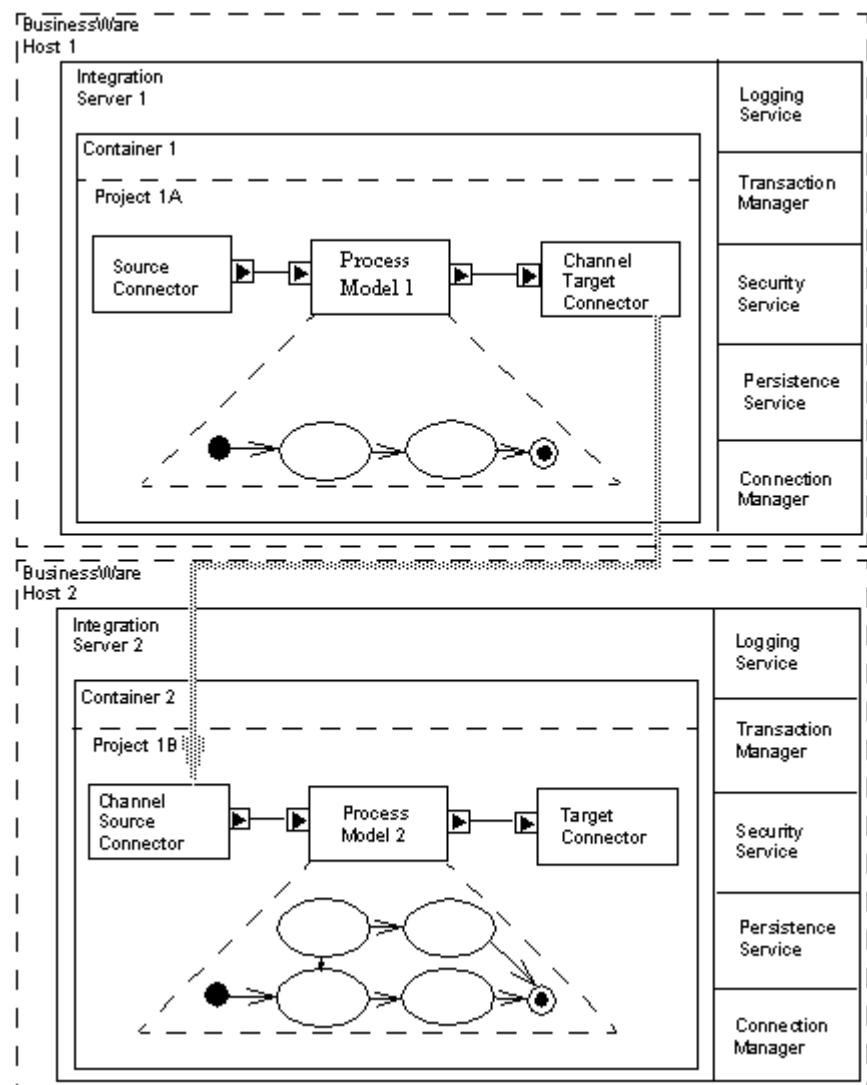


Figure 24-4 Partitioning a Project between Two Integration Servers

In Figure 24-4, Process Model 1 and Process Model 2 communicate through a channel. However, you can partition projects as you like. Process models that communicate directly with each other through synchronous invocations, for example, can be located in the same container or in different containers in different Integration Servers. Partitioning decisions are typically independent from design decisions.

Because containers are isolated, you can run multiple projects in the same Integration Server (see [Figure 24-5](#)). In such cases, resources may conflict as threads are allocated from a common thread pool.

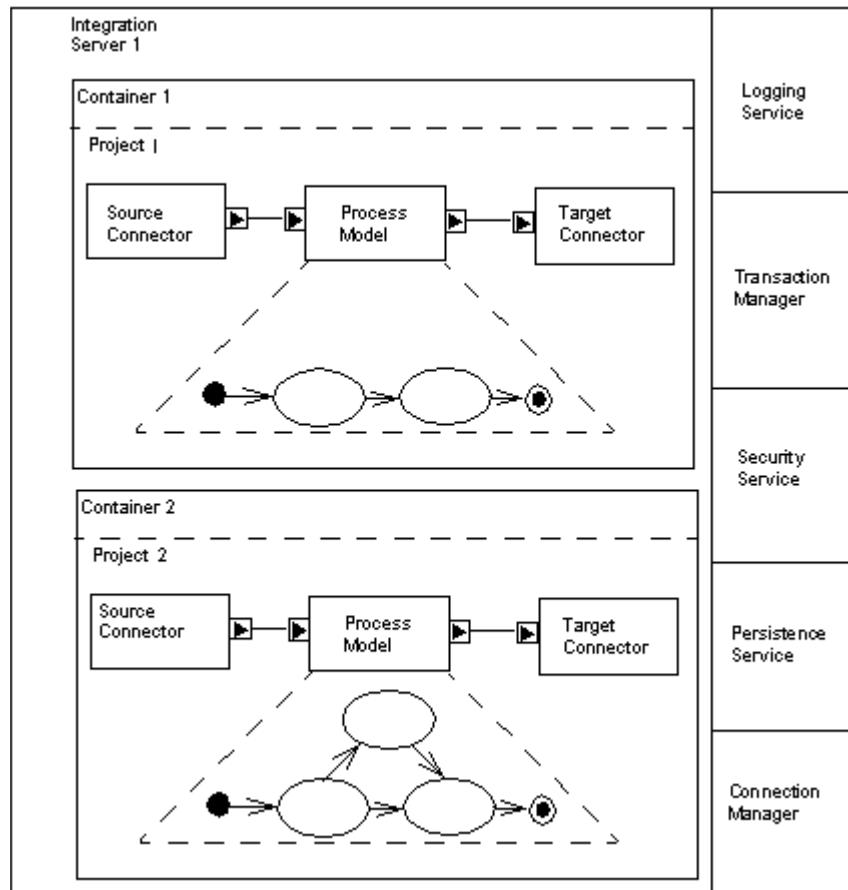


Figure 24-5 Running Two Project Versions in Two Containers

COMPONENTS AND PORTS

Components defined in integration models are used at runtime to provide a context in which to execute the linked models. See [Chapter 6, “Integration Model Basics.”](#) To remotely invoke the input ports of components, the container ensures the ports are registered in the directory server on startup.

Ports that use Local Java as their interface type can only be invoked locally. You should use a local (non-RMI) Java interface on an output port of a process component only if that port is connected to an input port of a process component that is partitioned in the same Integration Server as the calling component.

You cannot use local Java interfaces for publishing to channel or queue target connectors. Interfaces must be RMI interfaces because the message is serialized.

PROJECT STARTUP

The following is the sequence of events that occurs when a project starts:

1. When an administrator, using the Web Administration tool or the vtadmin command-line tool, starts a project, the Web Administration tool or vtadmin command-line tool looks up the project in the directory server and contacts the BusinessWare services that control the Integration Servers used by the project.
2. Each BusinessWare Server starts the project's Integration Server, if it is not already running, and directs the Integration Server to start the project.
3. Each Integration Server creates a container for the project.
4. The container loads project configuration data from the directory server, prepares project components, and starts project source connectors. As source connectors emit events, the container invokes the destination components and activates models as needed.

ACTIVATION

A component can be invoked by:

- A source connector
- A remote synchronous client
- Another component

When a request arrives, the container determines which model to use to handle the request and which model to activate. Activation includes loading and caching model classes for runtime use. If the model is a stateful process model, it will also determine which business process object will be created or loaded from the database.

RUNTIME SERVICES

BusinessWare Integration Servers and containers provide runtime services (platform services) used by project components. These runtime services are controlled by property settings. See [Chapter 20, “Integration Servers”](#) for more information on Integration Server properties.

PERSISTENCE

The Persistence service manages the runtime state of various components in your system. This runtime state is used for both recovery and application data management.

It is common to consider the persistence when developing stateful process components. At design time, business objects are developed to maintain the state of a business process; these objects must be persisted at runtime. The Integration Server provides an object-relational mapping service that automatically loads and stores business objects at runtime. See [Chapter 11, “Process Models: Defining and Using Business Objects.”](#)

In addition to business process object data, recovery information for components and source connectors is also persisted to a database. This includes information such as timers, transaction IDs, channel positions, file delimiters, and so on.

TRANSACTION MANAGEMENT

Handling an invocation may require updates to one or more persistent resources. Correctness requires that these be all-or-nothing updates—every resource gets updated or none of them gets updated—regardless of network failures, server crashes, or resource failures.

An Integration Server manages transactions within itself and between itself and other Integration Servers, observing the Java Transaction API JTA, JTS, and OTS standards. At project design time, you can control whether a component participates in the caller’s transaction context or runs in an entirely separate transaction context.

For more information on transactions, see [Chapter 25, “Transaction Management.”](#)

CONNECTION MANAGEMENT

Target connectors use connections to communicate with their external resources or applications. These connections are generally used intermittently and require considerable overhead to set up. To minimize setup overhead, an Integration Server maintains a pool of connections that its target connectors share.

The Integration Server connection manager also works with the transaction manager and connectors to ensure that updates to connected external resources and applications are performed transactionally. The Integration Server can be configured with a connection manager service. The connection manager implements the JCA (Java Connector Architecture) standard.

For more information on connectors see [Chapter 6, “Integration Model Basics”](#) and [Chapter 7, “Channels and Queues.”](#)

WEB SERVER AND SERVLET ENGINE

The Integration Server is bundled with an internal Web server and servlet engine to host BusinessWare web applications, such as the following, to handle HTTP and HTTPS requests:

- Workflow Applications
- HTTP Connectors
- Cockpit
- Web services

You can configure the Integration Server to use the bundled Web server in three modes.

- **Internal mode**—the bundled Web server is used *both* as a Web server and servlet engine in which the Integration Server supports HTTP and HTTPS protocols.
- **External mode**—the bundled Web server is used *only* as a servlet engine, and third-party Web servers, such as Apache and IIS, are used instead to forward requests to the servlet engine in the Integration Server with the help of redirectors. See the *BusinessWare Administration Guide* for more information on redirectors.
- **None**—Web server is not enabled. This is the default mode.

When a project is started with the Web server enabled either in internal or external mode, the web applications are written to disk on the Integration Server’s home directory. See [Chapter 20, “Integration Servers”](#) and the *BusinessWare Administration Guide* for more information.

SECURITY

Administrators establish user and group accounts in the directory server repository using the directory server's native administration tool, and grant them permissions to use BusinessWare runtime entities such as projects, channels, and Integration Servers.

For components that are invoked by remote synchronous clients, the Integration Server:

1. Authenticates and checks the authorization of all such invocations.
2. Extracts the credential from the incoming call. The BusinessWare runtime can carry credentials in an invocation from the client to the server. If no credentials exist, configured guest credentials are used.

Note: Under certain conditions, a credential may not be present in an incoming call. For example, a synchronous invocation from an EJB running on a foreign application server into a process model may not carry a credential. Similarly, Web services or HTTP invocations do not carry a credential. In such cases, the Integration Server grants the caller the BusinessWare *guest* credential. (The BusinessWare guest credential is set up, if necessary, during security configuration using the Web Administration Console.)

3. Verifies whether the caller has the permission to make the invocation. Access controls on components are specified through directory server-based Access Control Instructions (ACI) on the input port bound to the directory server under the project.

For Web applications, such as P2P applications where email notifications are sent outside the firewall, the Integration Server:

- Extracts the credential from an incoming call, (for example, via HTTP basic authentication) and maps it to a user credential in the directory server. All tasks performed within the Integration Server will use this credential. If no credentials exist, the configured guest credentials are used.

Note: If no guest credentials have been configured, an authentication failure exception will be thrown.

IMPORTANT: For more information on security, see the *BusinessWare Security Guide*.

LOGGING

When it starts running, an Integration Server creates the loggers configured for it. All projects running in an Integration Server share the same log. See [Chapter 20, “Integration Servers”](#) for information on configuring loggers and log trace levels.

This chapter describes transaction processing in BusinessWare, introduces transactions in general, and describes BusinessWare’s support for transactions in modeling. This chapter briefly describes transaction configuration and semantics of the BusinessWare runtime. (See [Chapter 24, “BusinessWare Runtime Architecture”](#) for more information.)

Topics include:

- [Transactions Overview](#)
- [BusinessWare and Transaction Standards](#)
- [Transactions and Modeling](#)
- [Transactions and Integration Servers](#)
- [Transaction Propagation Using OTS](#)

This chapter does not describe the effect of exceptions on BusinessWare transaction outcomes. Refer to [Chapter 26, “Exception Handling”](#) for information on that subject.

TRANSACTIONS OVERVIEW

Transactions provide a way to ensure data integrity in the face of multiple users, multiple servers, and the possibility of system crashes. Because businesses depend on accurate, consistent data, transactions are critical to business systems, such as those built with BusinessWare. For example, in [Figure 25-1](#), a transaction automatically protects all steps of this simple order entry system, ensuring that no order, customer, or financial data is lost, duplicated, or partially updated, even if a server running one of the components crashes or a network failure occurs.

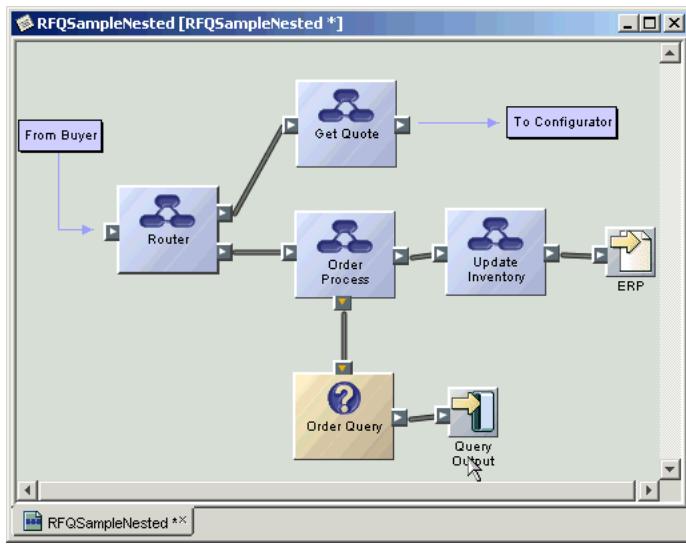


Figure 25-1 Simple BusinessWare Integration Model

A *transaction* delimits an “all or none” group of related updates; enclosing these updates in a transaction guarantees that either all of them will be performed or none of them will be performed.

Example: The classic transaction example is the two updates required to transfer funds from bank account A to bank account B. Executing these actions in a transaction ensures that either:

- A is debited and B is credited
- Neither A nor B is changed

The first case occurs when all resources needed to complete the operations are functioning properly. The second case occurs when there is a problem with one resource, for example, the database holding account B is not responding. In that case, by not debiting A, the accounts are still correct, and the transaction can be retried later.

Even if a crash occurs after A has been updated but before B has been updated, a transaction ensures that when the system comes back up, either A will be reset to its old value or B will be updated to its new value before any software or person sees the out-of-balance condition.

TRANSACTION PROPERTIES

Transactions have the following properties, which are known by the acronym ACID:

- **Atomicity**—either a transaction completes all the actions within it, or none of them. If it completes all of the actions, the transaction is said to have *committed*; if it performs none of the actions, leaving the data in the state it was in prior to running the transaction, the transaction is said to have *rolled back*.
- **Consistency**—after the transaction completes, the data will be in either the state that results from performing the transaction (if the transaction committed) or the state it was in prior to the transaction (if the transaction rolled back). In either case, the data will not be corrupt or in a state where only some of the updates within the transaction have occurred.
- **Isolation**—transactions do not interfere with each other; even if transactions run concurrently, they are not aware of each other. Isolation ensures that one transaction never sees data in an inconsistent state that would result from another running transaction having only partially updated the data.
- **Durability**—the actions performed by a committed transaction are not lost, even in the face of system shutdown or failure.

Example: Consider a system that holds customer information: first name, last name, street address, city, ZIP (postal) code, area code, phone number, etc. When customers call to update their accounts, they may want to update multiple fields, such as street address, city, zip code, area code and phone number. A transaction would ensure that:

- All of those fields are updated, or none of them are. The situation would not occur in which only the street address and area code were updated (atomicity and consistency).
- While the customer's record is being updated, another customer service representative cannot look at the record. This keeps the transactions unaware of each other and prevents the other representative from seeing partially-updated data (isolation and consistency).
- After the transaction commits, the customer will not receive the unpleasant surprise of having new correspondence go to the old address (durability).

Note: Transactions guarantee these properties even if the system crashes.

TRANSACTION TYPES

Transactions can be divided into two types: single-server and multiple-server.

Single-Server

In a *single-server* transaction (see [Figure 25-2](#)), the data that must be updated as a unit resides in a resource managed by a single server. A transaction *resource* is an entity whose state can be changed in a persistent way. In BusinessWare, writable entities associated with connectors are transaction resources. Updating two bank accounts that reside in one database is an example of a single-server transaction.

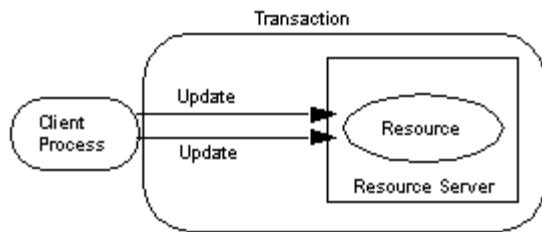


Figure 25-2 Single-Server Transaction

Multiple-Server

In a *multiple-server* transaction (see [Figure 25-3](#)), the data that must be updated as a unit resides in multiple resources managed by multiple resource managers. Updating two bank accounts that reside in different databases is an example of a multiple-server transaction. Multiple-server transactions are more complex to implement because independent servers must be coordinated to ensure that all of them or none of them complete their updates.

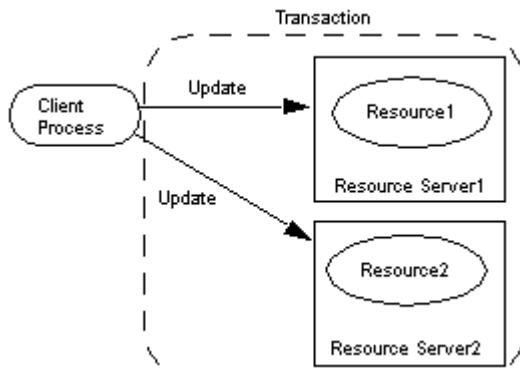


Figure 25-3 Multiple-Server Transaction

TRANSACTION PROTOCOLS

Transactions are implemented by system-level software; applications call that software to delimit the updates that are to be protected by a transaction. The system software implements a transaction *protocol*. The most widely used transaction protocol is called *two-phase commit*; it supports both single and multiple-server transactions. A related protocol, called one-phase commit, is an optimization that supports only single-server transactions. The two-phase commit protocol can be optimized if there is only one resource in that transaction. In this optimization, `prepare()` is unnecessary, and the resource is directly committed. These protocols are described in "[“Two-Phase Commit” on page 25-7](#)" and "[“One-Phase Commit” on page 25-9](#)".

Resource and Transaction Managers

The implementation of a transaction protocol is divided between resource managers and a transaction manager (see [Figure 25-4](#)).

- **Resource Manager**—administers a resource server, responding to update requests from applications and to protocol requests from a transaction manager.

- **Transaction Manager**—provides an application interface to transactions, directs the operations of resource managers, tracks the progress of transactions, and recovers incomplete transactions after a crash. An application that wants transaction protection interacts with a transaction manager and the resource manager of each resource it uses in the transaction.

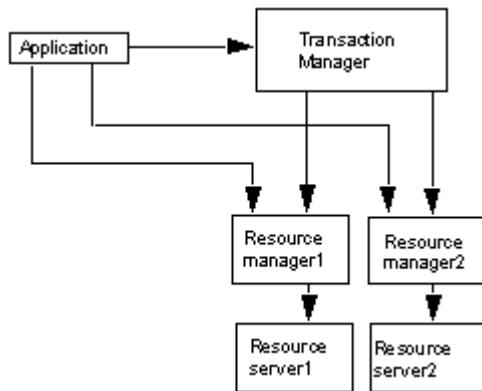


Figure 25-4 Resource and Transaction Managers

To an application programmer, a transaction looks something like the following pseudocode.

Example:

```
begin transaction;
    open Resource1();
    update Resource1();
    open Resource2();
    update Resource2();
    if x < y then
        open Resource3();
        update Resource3();
        commit;
    else
        abort;
    if transaction_status = failed then
        // Respond, typically by retrying several times
```

In this pseudocode, `begin transaction` indicates that succeeding statements through `commit` or `abort` are protected by a new transaction. The pseudocode opens and updates resources; however, these updates are not durable (they are, in effect, tentative) until the pseudocode calls `commit`.

If the pseudocode encounters no problem with its own execution, it calls `commit` to direct the transaction manager to apply the tentative updates. Conversely, if the application discovers a problem, it calls `abort`, which directs the transaction manager to abandon the tentative updates. In this pseudocode, the `transaction_status` indicates whether the transaction manager and the resource managers successfully executed the `commit`.

Two-Phase Commit

The two-phase commit protocol supports single and multiple-server transactions. To support *distributed* multiple-server transactions, in which resource managers run on different servers, each server must have a transaction manager that can play two roles, *Coordinator* and *Subcoordinator*. When playing the Coordinator role, the transaction manager on the application's host directs the operation of transaction managers on other hosts (see [Figure 25-5](#)). The remote transaction managers play the Subcoordinator role, operating on their resource managers as directed by the Coordinator.

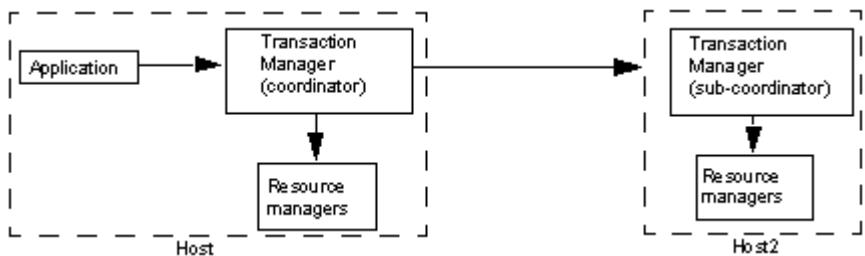


Figure 25-5 Transaction Coordinator and Subcoordinator Relationship

When an application begins a two-phase transaction, the transaction manager on its server becomes the Coordinator for that transaction. When the application obtains (for example, opens) its first resource, the Coordinator creates a new transaction record in memory and notifies the relevant resource manager that a transaction has begun. As the application obtains additional resources, the Coordinator updates its transaction record and informs the relevant resource managers. As the application updates resources, the resource managers do not actually make the updates but record data that will enable them to make the updates later.

- **Precommit**—if the application commits the transaction, the Coordinator begins the *prepare* phase (sometimes called the *precommit* phase). The Coordinator logs its transaction record in a database that survives crashes. Following a crash, the Coordinator uses the persistent transaction log to complete transactions that were in progress when the crash occurred. The Coordinator asks each resource manager to confirm whether it can complete the updates it has been accumulating. Each resource manager can confirm or refute by sending their reply as Yes or No to the Coordinator. Before confirming, a resource manager locks its resource so that no other transaction can change the state of the resource until the current transaction completes.
Note: Not all resource managers may lock their resources, but they achieve the same effect of preventing conflicting resource state changes.
- **Outcome**—after asking all resource managers to prepare, the Coordinator examines their replies and initiates the second phase called the *outcome*. There are two kinds of outcomes: commit and abort. If all resource managers replied Yes in the prepare phase, the outcome is *commit*. The Coordinator instructs each resource manager in turn to actually make the changes it promised when it replied with a Yes in the prepare phase. Each resource manager uses the data it has accumulated to make changes, discards the accumulated data, and unlocks its resource. When the last resource manager finishes, the transaction Coordinator updates its transaction record to show that the transaction is committed and returns success to the application.
- **Abort**—if one or more resource managers reply No in the prepare phase, the outcome is *abort* (also called rollback). The Coordinator instructs each resource manager to abort the transaction. Each resource manager discards the update data it had been accumulating for the transaction and unlocks its resource. When all resource managers have aborted, the Coordinator updates its transaction record and returns failure to the application.

Summary

To summarize the two-phase protocol:

- Resource managers durably update their resources until told to commit.
- The prepare phase begins when the application commits the transaction.
- The Coordinator uses the prepare phase to determine if all resource managers can commit; resource managers prevent conflicting updates until the transaction commits or aborts.
- If all resource managers can commit, the Coordinator directs them to commit and returns success to the application.
- If one or more resource managers cannot commit, the Coordinator directs all of them to abort and returns failure to the application.

In addition to orchestrating the execution of transactions, a transaction Coordinator recovers incomplete transactions after a crash. When a Coordinator restarts after a failure, it reads its persistent transaction log and recovers as follows:

- If the outcome (commit or abort) of the prepare phase was recorded before the crash, the Coordinator re-delivers the outcome to the resource managers that have not already received it.
- If the outcome was not recorded, the Coordinator delivers an abort to the resource managers.

One-Phase Commit

Not all resource managers support the two-phase commit protocol. Some support the simpler one-phase commit protocol, which has no prepare phase. The one-phase commit protocol supports single-server transactions. A one-phase resource manager also can participate in a restricted variant of the two-phase protocol.

A *single* one-phase resource manager can participate in a two-phase transaction if the Coordinator sequences its operations as follows:

- Prepares the two-phase resource managers.
- Commits the one-phase resource manager if all the two-phase resource managers reply with a Yes; otherwise aborts all the resource managers.
- Commits the two-phase resource managers if the one-phase resource manager successfully commits; otherwise, aborts the two-phase resource managers.

This sequence of operations ensures that the transaction retains four important properties of an enterprise-level transaction: Atomicity, Consistency, Isolation, Durability. It does so only because the number of one-phase resource managers is limited to one. (If a transaction included two one-phase resource managers A and B, and A committed but B did not, the Coordinator cannot abort A because it had already committed.)

If a crash occurs and a Coordinator encounters a transaction record that includes a one-phase resource manager, it proceeds as follows:

- If the outcome of the prepare phase was recorded, the Coordinator delivers it to the two-phase resource managers that had not previously received it.
- If the outcome was not recorded, and the one-phase resource manager had committed, the transaction Coordinator delivers a commit outcome to the two-phase resource managers.

BUSINESSWARE AND TRANSACTION STANDARDS

To allow multiple-server transactions to span independently developed systems, industry consortia have created transaction processing standards. Systems that conform to the same standards can interoperate transactionally. BusinessWare's implementation of transactions complies with the following standards:

- **Java Transaction Application (JTA)**—defines the application programming interface (API) for beginning and ending a transaction and incorporates the XAResource standard, which defines the operations a resource manager must implement to participate in two-phase commit transactions.
- **Java Transaction Service (JTS)**—defines how transaction managers on different servers interoperate to implement distributed multiple-server transactions; incorporates the OTS standard.
- **Object Transaction Service (OTS)**—defines the wire-level protocol for transaction manager interoperation.

Foreign clients and servers that implement these standards can interoperate with BusinessWare, extending the scope of transactions across independent systems.

TRANSACTIONS AND MODELING

BusinessWare objects have built-in support for two-phase transactions, including distributed transactions and a single one-phase resource manager in a two-phase transaction. Some components work with transactions in a predefined way; others have default behaviors that you can adjust with properties you set.

For example, BusinessWare process components can fully participate in BusinessWare’s two-phase commit transactions. From a modeler’s perspective, connectors access resources and their transaction capabilities vary according to the capabilities of the application or resource manager to which they are connected.

Connector Transactionality

Each connector has a read-only property called *Transactionality* that describes its transaction capabilities as follows:

- **Two-phase**—The connector can participate in BusinessWare transactions without restriction. Examples: all channel connectors, all queue connectors, File Source, Line Source, and Email Source Connectors.
- **One-phase**—The connector can be the single one-phase connector in a transaction (see “[One-Phase Commit](#)” on page 25-9). Examples: connectors for some third-party applications.
- **Local**—The connector does not participate in the global transaction. The connector’s work is committed in a separate local transaction that commits after the event is received by the connector. An event is received, the local transaction commits, then the global transaction may either commit or roll back. Example: File Target connector, Channel Target connector, Queue Target connector.
- **Dynamic**—The RDBMS Connector (alone) displays this value. It means that the connector’s transactionality depends on the database to which it connects at runtime.
- **Nontransactional**—The connector does not guarantee data integrity; in particular, retried transactions can cause the associated resource to be read or written multiple times. Examples: Email Target, and HTTP Target Connectors.
- **Unknown**—The connector cannot display its transactionality; this is mainly true of ported BusinessWare 3.x connectors. Consult the connector’s documentation to learn if it has transactional limitations.

TRANSACTION SCOPES

A transaction's *scope* defines the component actions that are protected within the transaction.

Note: A transaction's scope is a runtime phenomenon distinct from textual extent of a transaction, which is the source code statements between a begin transaction and a commit or abort. The scope defines how much of the runtime call depth is included in a transaction.

For example, refer to Figure 25-6.

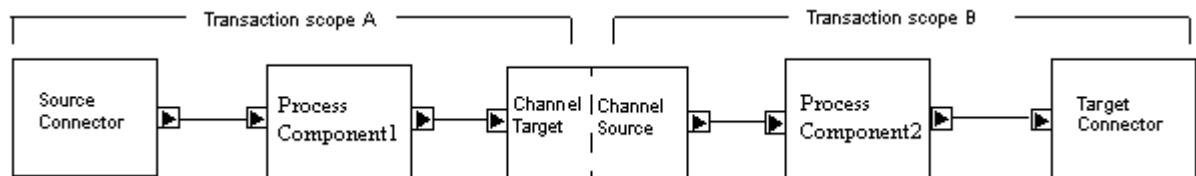


Figure 25-6 Sample Integration Model and Transaction Scopes

Transaction A protects the actions of the source connector, the process component, and the publish action of the Channel Target. (The BME displays a Channel TargetSource as a single icon, but the icon represents two connectors that share a channel.) When the source connector successfully commits, you are ensured that all events it has emitted and all subsequent actions by the process component and Channel Target, have committed.

Example: Suppose that for each event it receives from the source connector, the process component sends an event to the Channel Target. Transaction scope A ensures that when the source connector successfully commits:

- The source connector's position in its data stream has been updated.
- The process component's business process objects, data objects, timers, and other auxiliary data, if any, have been updated.
- The events generated by the process component have been stored in the channel associated with the Channel Target Connector.

Notice, however, that a successful commit by the source connector in transaction scope A does not mean that events have been processed by the target connector, whose actions are in transaction scope B. The Channel Source Connector begins the transactions that have scope B. When the Channel Source successfully commits, you are ensured that all events emitted in the transaction by the Channel Source have been successfully processed by the process component and the target connector.

BusinessWare transaction scopes do not always correspond to the scope of a “business transaction.” For example, suppose that, from a business point of view, all actions in [Figure 25-6](#) should be protected by a single transaction. That is, a complete “business transaction” consists of all the processing from connector source through connector target. However, as [Figure 25-6](#) shows, these actions will be protected by a pair of BusinessWare transactions. If both BusinessWare transactions successfully commit, there is no discrepancy; the two BusinessWare transactions produce the same result as a single “business transaction.” However, it is possible that a transaction created by the source connector (scope A) commits but the following transaction (scope B) created by the Channel Source fails. In that case, the BusinessWare transactions have been completed ACIDly but the “business transaction” has not been completed ACIDly; only those actions in transaction scope A have been completed ACIDly.

Definition

BusinessWare determines transaction scopes as follows:

- A source connector always starts a transaction scope (that is, it begins and commits or aborts a transaction). Multiple threads in source connectors begin distinct transaction scopes; transactions do not span threads.
- A model that is invoked by a source connector participates in the source connector’s transaction scope. See [“Transaction Control Property” on page 25-17](#) for more information.
- A model that is invoked by another model or a foreign client obeys the setting of its invoked input port’s Transaction Control property (see [“Transaction Control Property” on page 25-17](#)). You can set this property on an input port of a component to force it (and its linked model) to participate in the transaction scope of its invoker (the default) or to start a new transaction scope.
- A target connector participates in the transaction propagated by the upstream component that calls the target connector.
- Channel Target Connectors have a property called Transactionality that allows you to choose whether the Channel Target participates in a transaction and publishes transactionally to its channel. This trade-off can be advantageous for non-critical events, such as those containing status messages.
 - Transactional publishing (Transactionality = Two-phase) is the default. It makes the Channel Target’s publish operation a part of the transaction scope that started upstream as illustrated in [Figure 25-6](#). If the transaction is successfully committed, the events published by the target connector are guaranteed to have reached the channel.

- Publishing non-transactionally (Transactionality = Local) improves performance by excluding the Channel Target from the transaction scope that started upstream. For example, in [Figure 25-6](#), if the Channel Target's Transactionality was set to Local, transaction scope A would end with the process component.

Specifying Channel Target transactionality as Local improves performance at the cost of not guaranteeing that events reach the Channel Target's channel.

BATCH COMMIT LIMITS

Source connectors have a Batch Commit Number property that places an upper limit on the number of events they emit in a single transaction. For example, suppose a File Source Connector's Batch Commit Number is 10, and it has 12 files to convert into events. The File Source Connector begins a transaction, emits 10 events, and commits; it then starts a second transaction, emits two events, and commits.

Example: Suppose a Channel Source Connector whose Batch Commit Number is 10 has a single event; it begins a transaction, emits the one event, and commits. (In other words, the implied lower limit of a transaction batch is 1.) For instructions on setting Batch Commit Number, access the *BME Help*.

DESIGNING WITH ONE-PHASE RESOURCES

When you design an integration model, you must ensure that, at runtime, *at most one one-phase resource is referenced in a transaction*.

IMPORTANT: Breaking this rule will produce a runtime exception. [Figure 25-7](#) shows a design that may or may not break the rule. If, in processing a single event, Process Component invokes both one-phase connectors, the design is illegal.

If Process Component invokes One-phase Target Connector1 when it processes an event of type A, and One-phase Target Connector2 when it processes an event of type B, the design is legal—so long as A and B events never occur in the same transaction. One way to guarantee that two events never occur in the same transaction is to set the source connector's Batch Commit Number to 1.

However, larger Batch Commit Numbers produce better performance, so that is not always an acceptable solution. You can use a larger Batch Commit Number only if you *know* that an A event and a B event cannot possibly occur in the same transaction. (You may also want to prominently document this subtle yet critical design decision.)

Note: This example with events of type A and B is only an example: The rule for including multiple one-phase resources in the scope of a transaction is that no more than one of them can be referenced in any particular transaction.

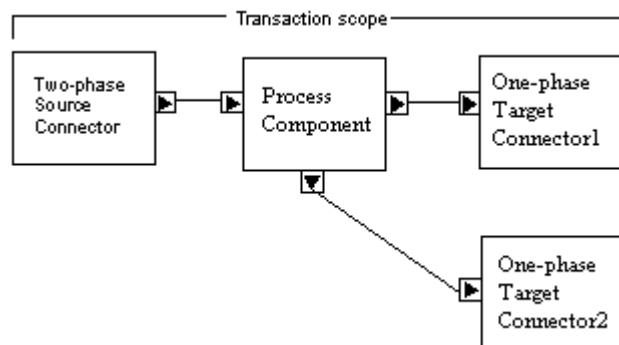


Figure 25-7 Two One-Phase Connectors in One Transaction Scope (Potentially Illegal)

As one alternative to the design in [Figure 25-7](#), you can use a Channel Target Connector to terminate the scope of the transaction started by the source connector. [Figure 25-8](#) shows this option. Process Component2, which does nothing but pass the event through, is necessary because connectors cannot communicate directly. You can construct a pass-through process component quickly by creating it with the RouteByPortType template built into the BME.

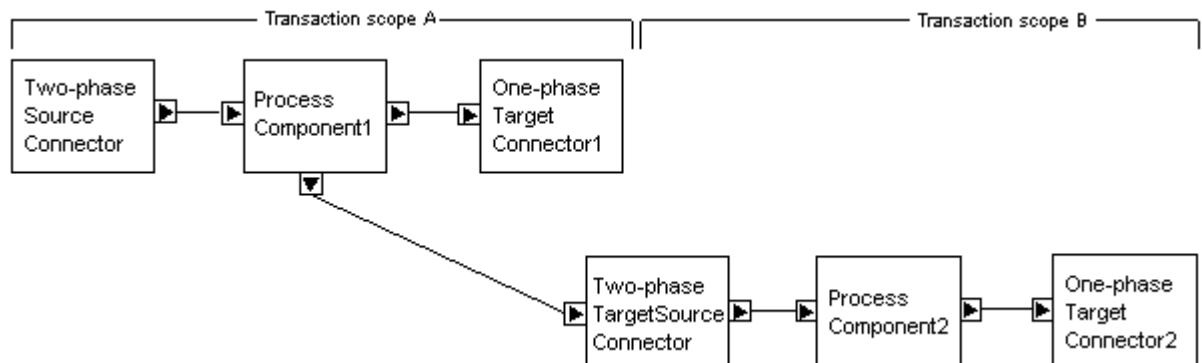


Figure 25-8 Two One-Phase Connectors in Two Transaction Scopes

Another alternative is to break Process Component into two process components, each of which writes to a single one-phase connector.

Note: The database used by process components to store business process objects is a two-phase resource; therefore, any number of process components can participate in a transaction.

TRANSACTIONS AND INTEGRATION SERVERS

Each BusinessWare Integration Server has a transaction manager that transparently coordinates and sub-coordinates with other transaction managers. You can adjust a transaction manager's behavior for deployment when you configure a project by setting its properties. Component input ports also have a transaction-related property that you can set.

TRANSACTION MANAGER PROPERTIES

Each transaction manager uses the following properties to control transaction timeouts.

To set Transaction Manager properties:

1. Expand the Integration Server node in the Explorer.
2. Right-click the **Transaction Manager** node, and select **Properties** from the shortcut menu.
3. Set the properties.
 - **Duration Timeout**—Time (in seconds) for the transaction manager to wait for the application to call `commit` before delivering an abort outcome to resource managers. The timer starts when the application begins the transaction. This property limits the time the resources are locked if the application has a problem. For slow-running processes, you must increase the timeout to exceed your transaction duration; otherwise, your transactions will abort. The default value is 0, infinite timeout.
 - **Inactivity Timeout**—Time (in seconds) for a Subcoordinator to wait for another call from a Coordinator that has begun a transaction. The timer begins when the Subcoordinator returns from a call by the Coordinator. After this time, the Subcoordinator delivers an abort outcome to its resource managers. This property limits the time the resources are locked if there is a problem with the network or the Coordinator. The default is 0, infinite timeout.

For instructions on examining and setting these properties, access the *BME Help*.

TRANSACTION CONTROL PROPERTY

You can use a component input port's Transaction Control property to specify whether the component should start a new transaction scope or participate in the scope passed by its invoker. (However, components ignore Transaction Control when they are directly invoked by a source connector, in which case they always participate in the transaction scope begun by the connector.)

Double-click an input port to set the Transaction Control property.

The Transaction Control values (which are the same as those defined in the EJB specification) are:

- **Required**—the component participates in the transaction scope of the invoker if there is one. If the invocation is not part of a transaction scope, the component begins one and commits it before returning. This is the default value.
- **Requires New**—the component suspends the incoming transaction (if there is one) and begins a new transaction. The component commits the new transaction before returning to its invoker.
- **Mandatory**—the component throws an exception if the invoker does not pass a transaction. If the invoker does pass a transaction, the component participates in it.

TRANSACTION PROPAGATION USING OTS

BusinessWare supports the Object Transaction Service (OTS), which allows transactions to span multiple servers. Since RMI objects within BusinessWare are managed by the SUN ORB, the extent of this support depends on the version of the JDK under which BusinessWare is run:

- **JDK 1.4 or higher**—added support in the SUN ORB will propagate transactions for RMI calls (i.e., the port type is Java-defined), both between BusinessWare Integration Servers and between BusinessWare Servers and third-party application servers that support OTS.

The three use cases supported are shown in [Figure 25-9](#) through [Figure 25-11](#).

- Integration Server to Integration Server
- BusinessWare Integration Server to application server
- Application server to BusinessWare Integration Server

INTEGRATION SERVER TO INTEGRATION SERVER

As shown in [Figure 25-9](#), a component A invokes a component B that has been partitioned on a separate Integration Server or is running in a separate project and invoked through proxies.

The transaction initiated by Integration Server A will be propagated to Integration Server B for CORBA calls and, on JDK 1.4, RMI calls as well. If the transaction is not propagated (for RMI calls on JDK 1.3), a separate transaction is initiated when the call enters Integration Server B (assuming the Transaction Control is Requires or Requires New). As a result, a rollback in component B will not cause a rollback in component A.

To propagate a transaction, the Transaction Control property must be set correctly for the process component's input port. Use either "Required" or "Mandatory".

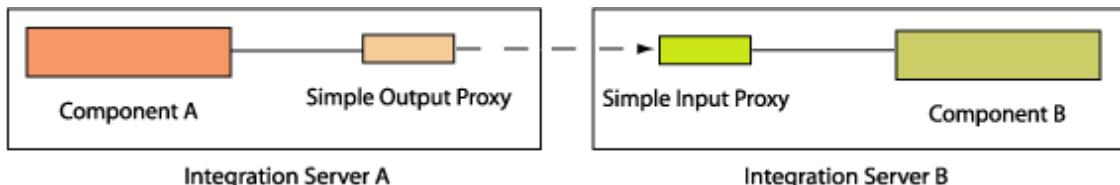


Figure 25-9 Integration Server-to-Integration Server Use Case

BUSINESSWARE INTEGRATION SERVER TO APPLICATION SERVER

As shown in [Figure 25-10](#), a component A invokes an EJB running on a third-party application server. All such calls are RMI. The transaction initiated by the Integration Server will be propagated to the application server if BusinessWare is running under JDK 1.4 and the application server supports OTS (such as, WebLogic 7.x). In this case, the application server's transaction manager acts as a subcoordinator in the two-phase commit protocol initiated by BusinessWare.

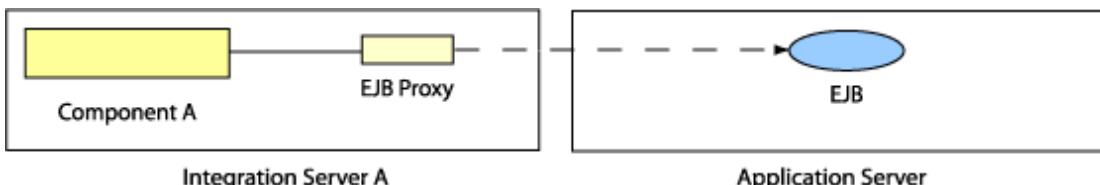


Figure 25-10 BusinessWare Integration Server-to-Application Server Use Case

APPLICATION SERVER TO BUSINESSWARE INTEGRATION SERVER

As shown in [Figure 25-11](#), an EJB running on a third-party application server invokes a BusinessWare component through a Simple Input Proxy via RMI. The transaction initiated by the application server will be propagated to BusinessWare if BusinessWare is running under JDK 1.4 and if the Application server supports OTS (such as, WebLogic 7.x). In this case, the BusinessWare Integration Server's transaction manager acts as a subcoordinator in the two-phase commit protocol initiated by the third-party application server.

To propagate a transaction, the Transaction Control property must be set correctly for the process component's input port. Use either “Required” or “Mandatory”.

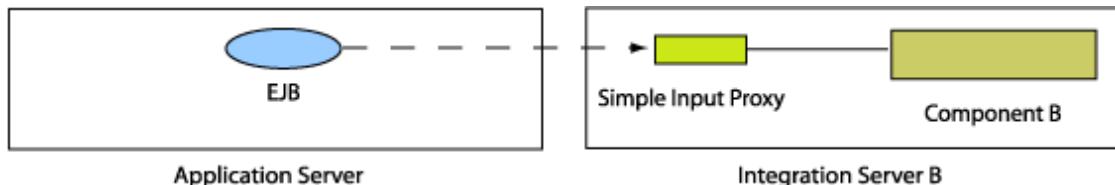


Figure 25-11 Application Server-to-BusinessWare Integration Server Use Case

TRANSACTION PROPAGATION WITH WEBLOGIC

To enable BusinessWare to propagate transactions into WebLogic and vice versa:

1. Make sure that BusinessWare is running JDK1.4.
2. Make sure you are running WebLogic 7.0 service pack 2 with patch CR093704_70sp2.jar, or WebLogic 7.0 service pack 3 or higher.
3. Configure the WebLogic server to use OTS 1.1 by setting the TxMechanism property in the server's config.xml file to “OTSv11”:

```
<IIOP Name="ServerName" TxMechanism="OTSv11"/>
```

where *ServerName* is the server you are running.

If WebLogic will not accept this value on startup, the patch has not been correctly applied.

You can patch the example server by adding the following line to the script that starts the server (before the call that starts the server) as follows:

```
set PRE_CLASSPATH=<patch_dir>\CR093704_70sp2.jar
```

where <*patch_dir*> is the directory containing the patch JAR file.

TRANSACTION MANAGEMENT

Transaction Propagation Using OTS

This chapter describes how BusinessWare components and connectors respond to runtime exceptions and what you can do to modify or extend the default response.

Topics include:

- [Exceptions Overview](#)
- [Exception Sources](#)
- [Common Actions for Exception Handling](#)
- [Exception Transitions in Process Models](#)
- [Exception Handler Overview](#)
- [Exception Handler Classes](#)
- [Exception Maps](#)
- [Additional Exception Handling Properties](#)
- [Using Port Invocation APIs](#)

Exceptions and transactions are closely related; for a description of transactions, see [Chapter 25, “Transaction Management.”](#)

EXCEPTIONS OVERVIEW

Exceptions are indications of problems in a business process that may need to be handled in a special way. Some exceptions have handlers, which are actions that are designed to respond to an exception when it occurs, giving the program a chance to recover from the abnormal condition.

KINDS OF EXCEPTIONS

There are two kinds of exceptions:

- Application
- System

Application Exceptions

Application exceptions are explicitly defined in interface definitions. That is, if you examine an interface definition, you can see the application exceptions that it can throw (if any). An application exception is an element of the client-server contract expressed by the interface; invokers must be prepared to receive and respond to application exceptions. An application exception is typically caused by erroneous parameters; for example, a `transferFunds()` method in a bank account interface may define an `InsufficientFunds` exception. A client that receives such an exception can respond by asking its users to enter a smaller amount or choose a different account, and then try again with the new parameter value.

Because application exceptions are specified within the interface definition, they are known as part of the contract established when wiring ports in the integration model. For more information on ports and interfaces, see [Chapter 6, “Integration Model Basics.”](#)

System Exceptions

System exceptions are not explicitly declared in interface definitions. Rather, system exceptions are implicitly defined by components that implement an interface. A system exception typically represents an environmental problem, such as a database that is not responding.

A client does not cause a system exception and cannot fix the underlying problem. All it can do is inform its user and/or retry the invocation; if the problem is transient, the invocation will eventually succeed.

EXCEPTION SOURCES

There are four "contexts" under which exceptions can be classified:

- **Component**—a component can encounter an exception in its own code or one rethrown by a component it has invoked.
- **Source**—a source connector can encounter an exception when connecting or polling its external resource or application.
- **Target**—a target connector can encounter an exception related to the external resource or application to which it is connected; for example, an application that does not respond to a connection attempt.
- **Outband**—an outband exception occurs outside the context of processing an event. That is, outside the component context or connector context.

COMPONENT EXCEPTIONS

A component can be invoked by several sources:

- A source connector
- An external client
- Another component

Usually, the invoked component processes the event and then returns it to the caller. However, the invoked component may detect a condition that can prevent it from completing its work.

Typically, components deal with application exceptions. For example, in a component representing account management at a bank, a missing account number would be an application error that may require special handling.

When a component encounters a condition it considers an error, it throws an exception. The BusinessWare runtime responds by invoking the component's exception handler class, which can be designed to "handle" the exception with specific actions.

In addition to handling exceptions that arise from within itself, a component can also respond to exceptions thrown from downstream components or target connectors. As in the previous case, the component's exception handler is invoked with the exception that originated downstream, and it is given the opportunity to "handle" it as well. Thus, exceptions can cause a chain reaction, giving each component in the chain an opportunity to handle it. It is up to the solution architect to decide where it is best to handle certain types of exceptions, and define exception handlers on those components.

FINE-GRAINED EXCEPTION HANDLING

Process components have a mechanism to provide more fine-grained exception handling through the process models themselves. Instead of using a coarse-grained exception handler mechanism to handle all exceptions that arise from within it, process models can define exception transitions. This way, exceptions can be managed on a per-state basis, by architecting a modeled path for the exceptions to follow. See "[Exception Transitions in Process Models](#)" on page 26-7 for more information.

External Applications Exceptions via Proxies

Proxies provide a visual representation of connections to external applications (for example, foreign clients or other projects). In other words, when you attach a proxy (representing an external application) to a BusinessWare component, you enable the component to invoke the external application via the proxy. Proxies do not handle exceptions thrown by the external application they represent; exceptions are handled by the component that invoked them.

SOURCE AND TARGET CONNECTOR EXCEPTIONS

Connectors enable BusinessWare components to interact with external systems. Therefore, connectors should also handle exceptions that occur when they communicate with these systems. For example, if an external database system is offline for a period of time, a source connector may try to reconnect for a specific duration or send a notification to someone to manually restart the system. A target connector may try to reconnect a certain number of times, and if not successful, it may abort transactions when requests are received to connect to this system.

Source connectors are at the beginning of the calling chain where requests to connect to BusinessWare originate; therefore they should be able to proactively handle situations when the system fails to function properly.

Target connectors are at the end of the calling chain. If they cannot handle a particular exception, the exception will be thrown back to the component, which invoked it, for handling. Typically, the connector does not throw the original external system exception, instead, it throws a connector-specific exception to the upstream BusinessWare component. This behavior is similar to the way the connector provides specialized events for components to invoke.

Note: When possible, target connector exceptions should be handled in the process component.

Note: See the appropriate connector documentation for more information on exception definitions and error handling requirements.

OUTBAND EXCEPTIONS

Outband Exceptions occur outside the context of a component or connector. For example, they can happen when the transaction manager is preparing to commit.

For example if we set the batch size, the Transaction Manager will commit after processing the number of events specified in the batch size. The commit happens outside the processing of an event. If the Transaction Manager failed or if the database connection is lost so that we can not persist the transaction state, then an outband exception will occur.

Note: To catch outband exceptions, the Exception Map must be specified on the project object.

COMMON ACTIONS FOR EXCEPTION HANDLING

When an exception is encountered, and an exception handler is invoked, there are some actions that are commonly performed by the exception handler. The following list provides a description of these actions. For more information, see more information, see the *BusinessWare Programming Reference*.

- **Abort**—mark the transaction “abort”, to indicate to the transaction initiator that the exception condition has prevented the component from completing successfully, and therefore the transaction should be aborted and retried. (The `abort()` method is available on the context object.)
- **Delayable Abort**—This adds an additional capability to the abort by allowing the user to define a delay time to wait before the abort occurs. (The `abort(long milli)` method is available on the context object.)

Note: Use caution when implementing this action as inappropriate use can cause deadlock or timeout.

- **Skip**—mark the transaction “skip”, which is only meaningful for transactions initiated by source connectors. Skip is like abort but additionally directs the source connector to omit the current event when it retries the transaction. (The `skip()` method is available on the context object.)

Note: Unlike “abort,” which only needs to mark the transaction, “skip” operates directly on the source connector. The call will propagate through a chain of components within a single Integration Server. However, the skip operation does not propagate across servers. The component that marks the skip must exist in the same process as the source connector. To cause a skip to occur in a different process, the remote component that would have marked the skip, would instead have to be designed to throw a specific exception to cause a more local (to the source connector) component exception handler to interpret and invoke the skip. While partitioning decisions are usually reserved until deployment time, this is an important design point that can influence partitioning later.

- **Delayable Skip**—This adds an additional capability to the skip by allowing the user to define a delay time to wait before the skip occurs. (The `skip(long milli)` method is available on the context object.)
Note: Use caution when implementing this action as inappropriate use can cause deadlock or timeout.
- **Throw Exception**—as a rule, the method should end by rethrowing the original exception or throwing a different exception, so the exception handler of the next component upstream can also respond. This action is performed for handling component and target connector exceptions alone.
- **Restart**—if the exception requires that the source flows be restarted, they can be restarted from the last successful commit. (The `restartCurrentProjectSourceFlows()` method is available on all contexts (`SourceFlowContext`, `ComponentContext` and `TargetConnectorContext`) and can be called anywhere as long as the method is available to the context. This is an action for the source connector exception handler.
- **Stop**—if the exception is severe, stop the project. (The `stopCurrentProject()` method is available on all contexts (`SourceFlowContext`, `ComponentContext` and `TargetConnectorContext`) and can be called anywhere as long as the method is available to the context. This action is performed by source connector exception handler.
- **Log and Notify**—produce an external notification of the exception, such as logging a message, publishing an event (non-transactionally), or sending an email. It is useful to augment one of the actions above with a logging message to help in diagnosing exceptions.

As a rule, your method should end by rethrowing the original exception or by throwing a different exception, so the exception handler of the next component upstream can respond.

The following sections describe how to handle exceptions using some of these methods in exception transitions, exception handler classes, and exception maps.

EXCEPTION TRANSITIONS IN PROCESS MODELS

Process models provide finer-grained exception handling through the use of *exception transitions*. (For a description of process models, states, and transitions, see [Chapter 12, “Process Models: Ports, States, and Transitions”](#)). Whereas normal transitions are triggered by events (and rendered in black), exception transitions are triggered by exceptions (and are rendered in pink). If, for example, an action state that has an outgoing exception transition encounters an exception, the transition to the next state follows the exception transition. If a state does not have an outgoing transition that is designed to handle a specific exception, the exception will then be handled in a more coarse-grained manner via the hosting component’s exception handler. For more information, see [“Exception Handler Overview” on page 26-10](#).

A state can have one or more incoming exception transitions. Such a state is referred to as a *successor state*. (However, there is no actual state of that name in the BME.) Any kind of state can be a successor state.

A successor state can analyze and respond to exceptions. Alternatively, if the exception handling logic is sufficiently complex to warrant centralization and isolation, a successor state can delegate exception handling to another process model as illustrated in [Figure 26-1](#).

In this example, the successor state transmits exception-related data to another process model by calling a port.

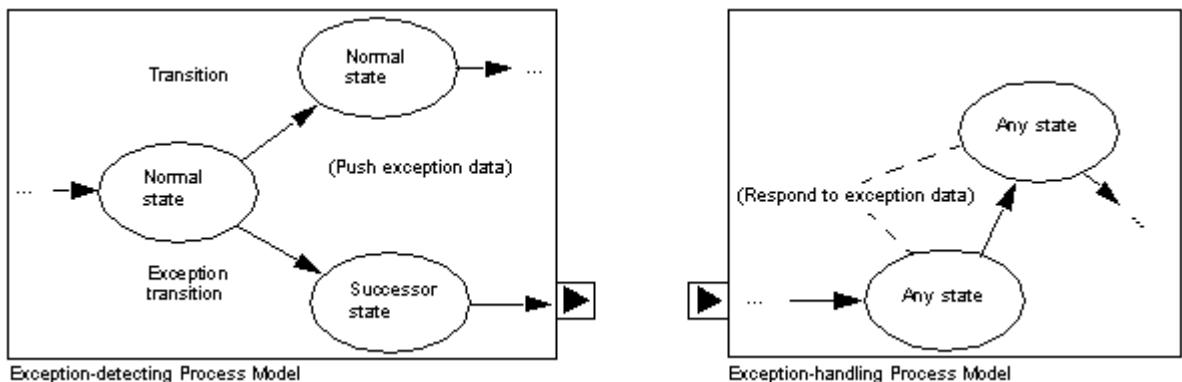


Figure 26-1 Responding to Process Model Exceptions in Another Process Model

The exception handling process model responds to the incoming exception event, analyzes and responds to the exception data, and returns. As a result of handling the exception, the process model usually passes the exception-related data to a target (a channel, or another process model) for appropriate action.

IMPORTANT: An exception should be handled in the same transaction. Therefore, do not interpose any asynchronous processing component (such as a channel or queue) between the exception-detecting and exception-handling process models. Asynchronous invocations are handled in their own new transactions.

TYPES OF EXCEPTIONS

Similar to events, exceptions can be defined based on the communication protocol used, for example, IDL, Java, etc. Exceptions that are defined in Java can include exceptions you have defined in your project or existing Java exceptions defined in JDK. For more information on communication protocols and type definition, see [Chapter 6, “Integration Model Basics.”](#)

IMPORTANT: The Java wrapper event, `bpevents.javaExceptionEvent`, is now deprecated; it will not be supported in future versions of BusinessWare. However, it continues to be supported in this version for backward compatibility for migrated projects. See *BusinessWare Migration Guide* for more information.

VISUALIZING ERROR PATHS

As described in [“Exception Transitions in Process Models” on page 26-7](#), exception transitions in process models provide a fine-grained exception handling mechanism that allows for users to incorporate special transitions that are visually distinct. These special types of transitions give the user the ability to visualize only the error paths that have been modeled.

Similar to normal transitions, where the `defaultEvent` trigger means “all other events,” the `defaultEvent` trigger on exception transitions means “all other exceptions.” For an introduction to the use of the `defaultEvent` refer to [“Default Transitions and Model Actions” on page 12-16](#). The type of transition chosen determines the runtime behavior. [Figure 26-2](#) shows the use of the `defaultEvent` on an exception transition to mean “all other exceptions besides ex1.”

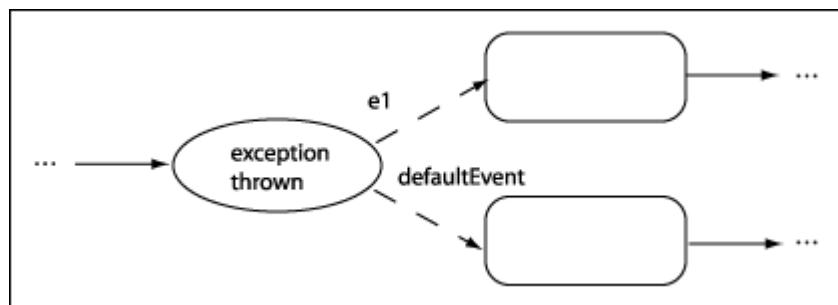


Figure 26-2 Default Exception on an Exception Transition

In addition, you can model entire error paths with the use of exception transitions. That is, exception transitions can be defined on a set of chained action states to fully define error paths in the model as shown in [Figure 26-3](#).

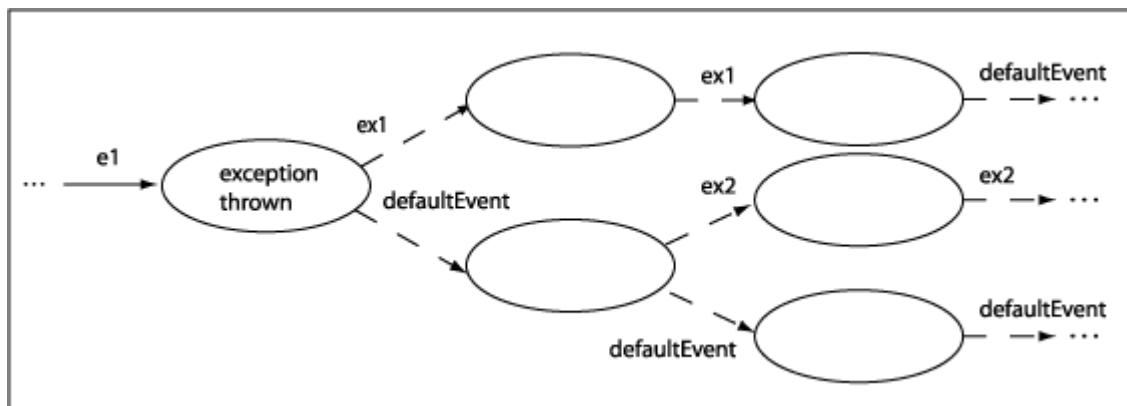


Figure 26-3 Exception Transitions in Chained Action States

Also, as discussed earlier, you can define separate error components with specific error process models using exception transitions as shown in [Figure 26-4](#). Note that in this case exceptions can originate from start states.

EXCEPTION HANDLING

Exception Handler Overview

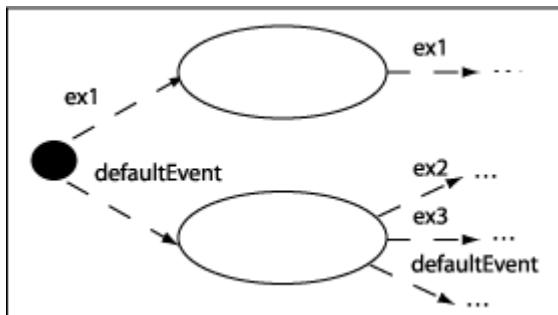


Figure 26-4 Exceptions Originating from Start State

EXCEPTION HANDLER OVERVIEW

Exception handlers provide a systematic and customizable way to respond to exceptions within the following four contexts discussed in “[Exception Sources](#)” on page 26-2:

- Component
- Source connector
- Target connector
- Outband

At the root of the exception handling mechanism is a base class definition, the `com.vitria.container.ExceptionHandlerImpl` class. See “[ExceptionHandlerImpl Class](#)” on page 26-11 for more information.

There are two ways to define exception handlers:

- Write an Exception Handler class that subclasses `ExceptionHandlerImpl`
- Create an Exception Map that allows you to define exception handling logic in a tabular form

Once an exception handler is created, it can be defined via the Exception Handler property on a component, source connector, target connector, and on the project itself. The project exception handler is used to manage outband exceptions, but is also used as a default if an exception handler is not explicitly configured for a component or connector.

DEFAULT PROJECT EXCEPTION HANDLER

The **Exception Handler** property in the project properties names the default project exception handler class. The BusinessWare *default project exception handler* is com.vitria.container.ExceptionHandlerImpl.

To edit project properties:

1. Select the project object in the Explorer.
2. Select the **Runtime** tab in the Properties Window.
3. Select the **Exception Handler** property.

Note: You can substitute your own project exception handler for the one supplied with BusinessWare. By supplying your own project exception handler, you change the exception handling behavior of all components in the project whose exception handler class is not explicitly specified by the component or connector.

EXCEPTION HANDLER CLASSES

ExceptionHandlerImpl CLASS

The com.vitria.container.ExceptionHandlerImpl class is the base java class for all exception handlers. It is used to handle component, source connector, target connector, and outband exceptions. This class implements the ExceptionHandler interface, which has the following four methods:

- **handleComponentException()**—handles exceptions arising in model code or thrown from another model or downstream target connector. The base class implementation of this method just rethrows the exception to the upstream component or source connector.

Note: When an exception is thrown to a foreign client, the effect on the current transaction depends on whether the exception is an application or system exception.

IMPORTANT: If the exception is thrown as the result of invoking a target connector, the exception may be wrapped within com.vitria.connectors.framework.common.EventInvocationException. The originating exception can be retrieved by calling `getTargetException()` on the EventInvocationException.

- **handleOutbandException**—handles exceptions generated outside of the event invocation (for example, during a two-phase commit). The Transaction Manager continues to retry the commit according to the values of the Transaction Manager's Retry and Retry Interval properties. If the exception continues after the Retry count is reached, the server is intentionally shutdown immediately. Additional actions can be performed by subclassing and implementing this method.
- **handleTargetConnectorException()**—handles exceptions related to a target connector external resource or application. This method usually rethrows the exception to the upstream component.
- **handleSourceConnectorException()**—handles exceptions related to a source connector's external resource or application.

Note: A source connector can also encounter an exception rethrown by a component it invokes. In that case, the source connector aborts the current transaction and retries it, subject to the project's Commit Retry Count property. If the value of this property has been exceeded, the source connector stops the project.

DESIGNING A CUSTOM EXCEPTION HANDLER CLASS

You can provide your own exception handler class by writing a subclass of com.vitria.container.ExceptionHandlerImpl. Put your exception handler class in one of your project directories, so it will be versioned, deployed, and built with your project. You can make your exception handler class the project exception handler or you can make it the exception handler for one or more components or connectors. You can have as many custom exception handlers in a project as you want, but only one of them can be the default project exception handler:

- To make a component use your exception handler class, set the component's Exception Handler property to the class you have written. If you do not change a component's Exception Handler Class, it uses the default project exception handler.

- To make all components that do not have an Exception Handler property value use your project exception handler, set the Exception Handler property of the Project object to the name of your exception handler class.

You cannot specify an Exception Handler property value for the process query component. The process query component rethrows exceptions.

The steps outlined below show how to develop an exception handler that will be used for a component, and implement the `handleSourceComponentException()` method.

To write a custom exception handler class:

- If you don't already have one, create a project subdirectory for your files. This example uses a package named "exception."
- Create a .java file in the new subdirectory for the exception handler class you are implementing.
- In the .java file, insert the package statement, using the name of the subdirectory you just created ("exception"). Then insert the three import statements shown below, replacing `myCustomTypes` with the name of the file where you defined the custom exceptions you are handling:

```
package exception;

import com.vitria.container.client.*;
import com.vitria.container.ExceptionHandlerImpl;
import myCustomTypes.*;
```

- All custom exception handlers must extend the class `ExceptionHandlerImpl`, so insert the class extension statement as shown below, replacing `MyCustomExceptionHandlerImpl` with the name you want your handler to have. Make sure it matches the name of the .java file.

```
package exception;

import com.vitria.container.client.*;
import com.vitria.container.ExceptionHandlerImpl;
import myCustomTypes.*;

public class MyCustomExceptionHandlerImpl extends
    ExceptionHandlerImpl {
```

- Add the constructor as shown below, using the `super` keyword to specify the use of the base class constructor.

```
public MyCustomExceptionHandlerImpl() {
    super();
}
```

6. Insert your new definition of the method

`handleComponentException()`. The sample below aborts if one custom exception is encountered and skips if a different exception is encountered. Notice that the exception is rethrown before exit. You must do this so that the exception will flow upstream to the invoking components or client so that they can be notified of the problem; however, the exception handler has typically handled the specific exceptions by signaling abort or skip.

Note: This is common for component or target connector handlers. In the case of source connectors, they are the request originators and therefore, exceptions will not flow upstream. If implementing the `handleSourceConnectorException()`, do not rethrow the exception.

```
package exception;
import com.vitria.container.client.*;
import com.vitria.container.ExceptionHandlerImpl;
import myCustomTypes.*;

public class MyCustomExceptionHandlerImpl extends
    ExceptionHandlerImpl {

    public MyCustomExceptionHandlerImpl() {
        super();
    }

    public void handleComponentException(Exception ex,
        ComponentContext ctx) throws Exception {
        if (ex instanceof MyCustomException_1) {
            ctx.abort();
        }
        else if (ex instanceof MyCustomException_2) {
            ctx.skip();
        }
        //rethrown by convention for re-handling upstream
        throw ex;
    }
}
```

7. Package the .java file in a mounted directory in your project. The full path to the file must be directly underneath the mountpoint. In the above example, the exception directory would have to exist underneath the project mountpoint.
8. In the BME, enter the *fully qualified* Java class name for the Exception Handler property as shown in [Figure 26-5](#).

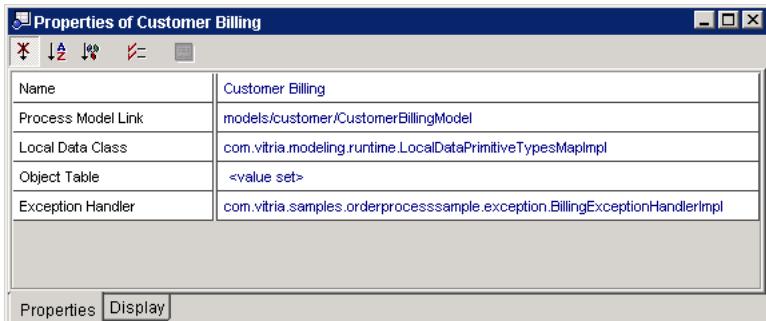


Figure 26-5 Specifying Exception Handler Class Property

PORTEXCEPTIONHANDLERIMPL CLASS

The `com.vtria.modeling.runtime.Por tExceptionHandlerImpl` class is a special class that pushes events to an output port of the component on which it is defined. This class can be used only on components and overrides the `handleComponentException()` method. The default implementation of the class expects the component's port to be untyped and port name to be "error"; however, the port name property is configurable. This class can be applied to any component in the BME that supports these port settings by specifying the class name for the component's Exception Handler property.

You can override the default implementation of this class by using the methods described below:

- **`getOutputPortName()`**—returns the string value containing the name of the output port to which the exception will be pushed. The default implementation sets the port name to "error". You can override the default port name by changing the method to return the port name that you specify.
- **`throwOriginalException()`**—returns a Boolean value that determines whether the original exception should be thrown back to the caller after it has been pushed through the output port. The default implementation of the class has the value set to True. To override this method, you can change the Boolean value to False, if the exception handling model to be invoked should control throwing of exceptions.

See *BusinessWare Programming Reference* for more information.

EXCEPTION MAPS

An exception map defines exception handling logic in a tabular form. It is an alternative to manually written Java code and presents the most commonly used elements of an exception handler class in an easy to configure form.

Exception maps specify an ordered list of exception mappings. Each exception mapping specifies a triggering exception, optional conditional guidelines, and a description of the desired action to handle the exception.

The advantage of creating an exception map is that you do not have to create a custom Java exception handler class to handle exceptions. The exception map auto-generates a Java class based on the exception mappings that you define in the BME. Like other model types, the exception map can be defined in the project in which it is used or within another project module, to which you can add a dependency.

Before creating an exception map, you should be familiar with the information provided previously in this chapter. You should also know how to use the Action Builder and the Condition Builder. For more information, see [Chapter 14, “Process Model Code Construction.”](#)

CREATING AN EXCEPTION MAP

During design time, you can create an exception map that contains a list of exceptions and their associated processing rules. This exception handling policy is implemented during runtime.

To create an exception map:

Use one of the following methods:

- In the Explorer, right-click on a directory and select **New > All Templates....**. In the New Wizard, expand **Models**, select **ExceptionMap** and click **Next**. When prompted, provide a name for the exception map. Click **Finish**.
- Select **File > New...** to display the New Wizard. Expand **Models**, select **ExceptionMap** and click **Next**. When prompted, provide a name for the exception map. Click **Finish**.

Note: No spaces or special characters are allowed in the name because it will be used as a valid Java class name.

After creating the exception map, you must configure its properties and add the exception mappings.

Configuring Exception Map Properties

[Table 26-1](#) lists the exception map properties.

Table 26-1 Exception Map Properties

Property	Description	Default Value
Name	Name of the ExceptionMap.	ExceptionMap(_2,3,4...)
Parent	Specifies the ExceptionMap or exception handler class that this ExceptionMap extends. Exceptions can be processed at more than one location. For example, ExceptionMapLocal may handle an exception but not “consume” it. In this case, the exception is passed to the parent for more processing.	com.vitria.container.map.ExceptionHandlerImpl
Imports	List of packages to import. Used for generated source code. The specified import statements are issued within the generated exception map code. You use these imports to enable your action or condition code to access functionality that exists in other Java packages.	No Default
Description	Description of the exception map.	No Default

DEFINING EXCEPTION MAPPINGS

When specifying an exception mapping, you must provide information on which exceptions are intercepted, any conditions that may apply, the specific processing to perform, and what to do when processing is complete. The following sections describe how to define an exception mapping.

Note: The order of the elements in the table determines the order that they will be examined and processed at runtime.

Exceptions

The value defined in the Exception field of the exception map determines the exceptions that BusinessWare will handle. BusinessWare provides lists of exceptions that it recognizes from the following categories:

- **Project**—exceptions that are present in the current project or in any dependent projects. Add exceptions to the list by selecting an exception in the Explorer and selecting **Use As Type** from the shortcuts menu.
- **Registered**—inband and outband exceptions provided by BusinessWare and the installed connectors.

Inband exceptions occur while an event is processed as it moves through a source connector, target connector, or component. Contextual information regarding event, BPO, components and connectors is available during processing. [Table 26-2](#) lists the inband exceptions.

Outband exceptions occur when a batch of events is committed and also during startup processing. No contextual information is available during processing. [Table 26-3](#) lists the outband exceptions.

The list of available exceptions is determined, in part, by the installed connectors.

- **Custom**—user-defined exceptions. Specifying a custom value for an exception is useful when:
 - The exception that you want to handle is not a member of the Project or Registered categories. This can occur if the exception comes from an older version of a connector or it is not explicitly used as a type in the project.
 - You want to perform matching based on a pattern match.

[Table 26-2](#) lists some examples of inbound exception producers and how the exceptions are typically processed. This list is representative of the inbound exceptions, but does not include exceptions installed with BusinessWare connectors.

Table 26-2 Inband Exceptions

Exception Producer	Exceptions	Context	Default Processing	Comments
Source Connector	Poll, Notification errors	Source Connector	Log, Retry	Indicates connector/application connectivity errors.

Table 26-2 Inband Exceptions

Container	EventMismatch, Authorization, Authentication, Transaction (if incoming transaction suspended) Component Activation	Component	Log, StopProject	Indicates severe errors concerning security failures or application design errors such as wrong event types or incorrect RequestMap on the port.
Container	Load BPO	Component		Indicates if BPO load fails.
Process Model Runtime	BadRuleException	Component	Log, StopProject	Indicates semantic errors in the process model design.
Process Model	Application Exceptions	Component	Log, Rethrow	By application design.
Transformation				
Target Connector	Resource Exception	Target Connector	Log, Rethrow	Indicates connector/application connectivity errors.

[Table 26-3](#) lists some examples of outband exception producers and how the exceptions are typically processed. This list is representative of the outband exceptions, but does not include exceptions installed with BusinessWare connectors.

Table 26-3 Outband Exceptions

Exception Producer	Exception Types	Context	Default Processing	Comments
Source Connector	Prepare, commit, rollback failures	Outband	Retry the operation as specified by the Transaction Manager's Retry Count and Retry interval properties. Perform hard system exit if unable to recover in order to ensure transaction integrity.	Applicable only to legacy source connectors – connectors built on the native framework handle the exceptions using internal logic.

EXCEPTION HANDLING

Exception Maps

Table 26-3 Outband Exceptions

Container	BPO, DO flush to database	Outband	Retry the operation as specified by the Transaction Manager's Retry Count and Retry interval properties. Perform hard system exit if unable to recover in order to ensure transaction integrity.	
Target Connectors	Connector creation error	Outband		Some target connectors perform initialization work during creation. For example, the RDBMS Target Connector creates its target table.

To define an exception mapping:

1. In the Editor, click **Add** to create a new mapping.
2. Click the browse button in the Exception field.
3. In the Exception Editor, select **Project**, **Registered**, or **Custom** based on the exception you are mapping.

For Project and Registered exceptions, click **Browse...** to select an exception from the Exception Picker.

For Custom exceptions, enter the fully qualified Java class name.

4. Click **OK**. The exception is listed in the Exception field.

To copy an exception mapping:

1. In the Editor, select an exception mapping.
2. Click **Copy**. The exception mapping is copied to the next line in the exception map.

To remove an exception mapping:

1. In the Editor, select an exception mapping.
2. Click **Remove**. The exception mapping is removed from the exception map.

Context

At runtime, processing attempts to match an exception based on the context in which the exception originally occurs. Contexts correspond to the methods invoked on the exception handler class and include the following:

- **All**—indicates that the match should succeed regardless of the context in which the originating exception occurred.
- **Component**—exceptions that occur within a component.
- **Outband**—exceptions that occur outside a component or a connector (for example, during transaction commit or project startup).
- **Source**—exceptions that occur between the source connector and the system to which it is connected.
- **Target**—exceptions that occur between the target connector and the system to which it is connected.

Match

At runtime, attempts to match an exception based on one of the following algorithms:

- **Exact**—requires a precise matching of what is specified to what is being tested.
- **Instance Of**—allows specifying a base class to match all descendent classes.
- **Pattern Match**—interprets the value entered in Exception and can be used to match based on the string name of the actual exception (`getClassName().getName()`), as opposed to its class hierarchy. (For example, the pattern '*' matches any number of characters and '?' matches exactly one character). Pattern matching is useful when capturing all exceptions from a specific package or custom, user-defined exceptions.

Condition

You can use the Condition Builder to specify additional conditional expressions in order to identify instance-specific conditions at runtime.

Note: When adding custom code in the User Condition box of the Condition Editor, you must include “`condition &=`” when specifying custom conditions.

Unwrap

Selecting Unwrap enables the matching algorithm to traverse the exception's `getCause()` chain while performing the matching.

It is common that one exception is wrapped or contained within another exception. For example, if the File Connector cannot access a directory, it throws a `FileConnectorException` to the BusinessWare container. The container creates a `ContainerException` and initializes the cause as a `FileConnectorException`. In order for the exception map to match on the `FileConnectorException`, it must be unwrapped from the `ContainerException`. Selecting Unwrap performs this operation automatically.

Transaction

Transaction indicates the effect that a matched exception should have on the current transaction or event. Options include:

- **Abort**—aborts the current transaction. During an abort, data is not lost and the transaction is retried.
- **None**—no action on the current transaction.
- **Skip**—aborts the current transaction and marks the current event as “skipped.” The event is not sent during the subsequent retry.

Action

Use the Action Builder to specify additional action code to identify instance-specific responses at runtime. For more information on using the Action Builder, see [“Using the Action Builder” on page 14-6](#).

Post Process

Post Process indicates what the runtime processing should be after an exception has been matched and its associated action is invoked. There are three possibilities:

- **Continue**—processing and evaluation proceed to the next exception mapping specified by the exception map (if any) or to the parent if this exception mapping is the last in the table.
- **Done**—processing is considered finished at this point and the parent is not notified.
- **Rethrow**—causes the originating exception to be rethrown.

[Figure 26-6](#) shows three exception mappings in the `BillingExceptionMap` in the `OrderProcess` sample. The first two exceptions are project exceptions and the third exception is a custom exception.

Exception	Context	Match	Condition	Unwrap	Transaction	Action	Post Process
...CustomerAlreadyExistsException	Component	Exact		<input checked="" type="checkbox"/>	Abort		Continue
....CustomerDoesNotExistException	Component	Exact		<input checked="" type="checkbox"/>	Skip		Continue
*	Component	Pattern Match		<input type="checkbox"/>	None		Rethrow

Buttons at the bottom: Add, Copy, Remove, Up, Down.

Figure 26-6 OrderProcess Sample Billing ExceptionMap

USING EXCEPTION MAPS

You can associate an exception map at any location where an Exception Handler is configurable:

- Process component
- Source connector
- Target connector
- Project

In a component properties window, clicking the browse button in the Exception Handler property opens the Select Exception Map dialog. Use this dialog to select an exception map from within the current project or any of the dependent projects. [Figure 26-7](#) shows the Select Exception Map dialog.

EXCEPTION HANDLING

Additional Exception Handling Properties

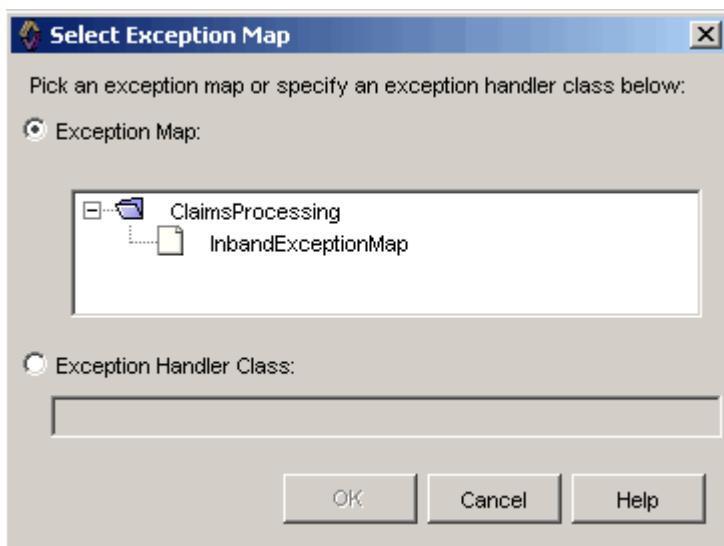


Figure 26-7 Select Exception Map Dialog

When exceptions are processed during runtime, information concerning exception map processing is automatically logged in the Integration Server's runtime log. For more detailed information concerning how the elements of the exception map are internally processed, set the Integration Server's Container trace level to Verbose. This is useful when diagnosing unexpected processing in an exception map.

ADDITIONAL EXCEPTION HANDLING PROPERTIES

Exception handling is also influenced by the following Project properties:

- **Commit Retry Count**—number of times to retry before shutting down the project, if a commit failure occurs. Enter zero (0) to retry infinitely.
- **Commit Retry Interval**—The number of seconds to pause between retries, default is 10.
- **Ignore Source Connector Errors**—ignore source connector errors on start and allow project to be started partially.

Exception handling is influenced also by the following Transaction Manager properties:

- **Retry Count**—Controls the number of times that the system will attempt to commit or rollback a specific resource before halting the Integration server. Enter zero (0) to retry infinitely.

- **Retry Interval**—Specifies time (in seconds) between retries.

USING PORT INVOCATION APIs

There are cases where it may be necessary to invoke connectors or other components to process exception handling logic. For example, to send email, publish to a channel, or apply more rules-oriented modeling logic to handle exceptions, you will need to invoke ports from the Exception Handler class (Figure 26-8). A set of context APIs have been provided to assist you.

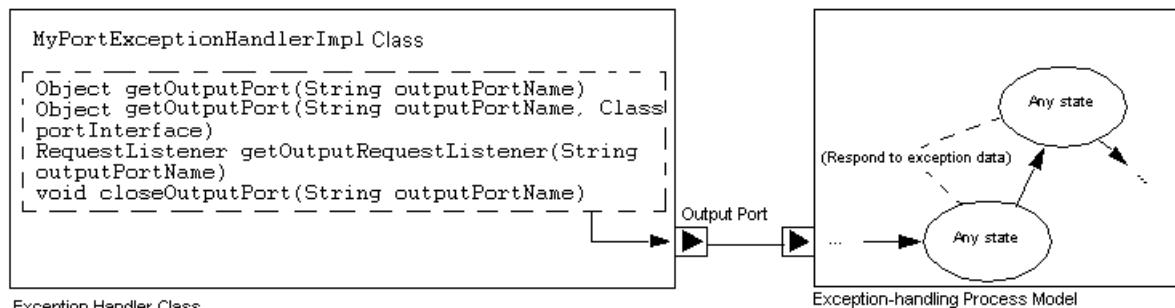


Figure 26-8 Invoking Ports from Exception Handler Class

RETRIEVING OUTPUT PORTS FROM COMPONENT EXCEPTION HANDLER CLASSES

When invoking methods on component output ports, the name of the port to invoke is specific to the component itself and does not change if the component to which it is wired is modified in any way.

Typically, you can use

`com.vitria.modeling.runtime.PortExceptionHandlerImpl` class defined in “[PortExceptionHandlerImpl Class](#)” on page 26-15, to invoke methods on component ports. However, you may need to define your own interfaces and methods to pass on data in the appropriate format. Methods in the `com.vitria.container.client.ComponentNamingContext` interface can be used for this purpose, as described below.

- `Object getOutputPort(String outputPortName)`—resolves ports that are typed with a known interface and returns an object that can be casted to the type specified by the port. The output port name used in the method should be the same as the name specified for the port’s name property. The connection must be closed when it is no longer needed.

- `Object getOutputPort(String outputPortName, Class portInterface)`—resolves ports that may support multiple interfaces and returns an object that can be cast to the provided interface. The output port name used in the method should be the same as the name specified for the port's name property. The connection must be closed when it is no longer needed.
- `RequestListener getOutputRequestListener(String outputPortName)`—returns a `RequestListener` instance for an untyped output port. The connection must be closed when it is no longer needed.
- `void closeOutputPort(String outputPortName)`—closes a connection (to a target connector) that has been opened to resolve an output port.

RETRIEVING INPUT PORTS FROM CONNECTOR EXCEPTION HANDLER CLASSES

Wires always connect output ports to input ports. Because you cannot define additional output ports on connectors, you may have to invoke methods on input ports of components that are not connected or wired.

The following methods in
`com.vitria.container.client.ComponentNamingContext`
interface can be used for port invocation:

- `Object getInputPort(String componentName, String inputPortName)`—resolves input ports that are typed with a known interface and returns an object that can be cast to the type specified by the port. The input port name used in the method should be the same as the name specified for the port's name property. The connection must be closed when it is no longer needed.
- `Object getInputPort(String componentName, String inputPortName, Class portInterface)`—resolves input ports that may support multiple interfaces and returns an object that can be cast to the provided interface. The input port name used in the method should be the same as the name specified for the port's name property. The connection must be closed when it is no longer needed.
- `Object getInputPortRequestListener(String componentName, String inputPortName)`—returns a `RequestListener` instance for an untyped input port. The connection must be closed when it is no longer needed.

For further discussion and information, see *BusinessWare Programming Reference*.

RETRIEVING OUTPUT PORTS FROM PROCESS MODEL ACTION CODE

Use the generated output port method in the process model, such as `getMyOutPort()`, appropriate for the current transaction. Invoking these methods, establishes and maintains port connection through the transaction and closes the connection when the transaction is complete. For more information on invoking ports from action code, see [Chapter 14, “Process Model Code Construction.”](#)

EXCEPTION HANDLING
Using Port Invocation APIs

BusinessWare provides several powerful tools to help you test and debug your project. This chapter introduces those tools, outlines the debugging procedure, and then describes each tool in greater detail. It also describes how you can animate your project for demonstration purposes.

Topics include:

- [Overview of the Debugging Tools](#)
- [Debugging Procedure](#)
- [Debugging Configuration](#)
- [Event Injector](#)
- [Event Inspector](#)
- [BPO Inspector](#)
- [Channel/Queue Inspector](#)
- [Log Viewer](#)
- [Animation](#)

OVERVIEW OF THE DEBUGGING TOOLS

[Table 27-1](#) describes the purpose each tool serves and the prerequisites for using it. Although commonly used during debugging, the Channel\Queue Inspector, Log Viewer, and Event Injector also can be used outside of a debugging session.

Table 27-1 BusinessWare Debugging Tools

Tool	What Is It Used For?	When Can It Be Used?
Debugger	Sets breakpoints in models, Java code, components, and target connector input ports. When the execution pauses at a breakpoint, you can inspect events, BPO, and Java variables.	All components to be debugged must run in one Integration Server. Debugging must be enabled on the Integration Server. Project must be deployed.
Event Inspector	Inspect event parameter values at a breakpoint.	Debugger must be paused at a breakpoint.
BPO Inspector	Inspect BPO attribute values at a breakpoint.	Debugger must be paused at a breakpoint in a process model.
Event Injector	Generate events and inject them into component ports in the integration model.	Project must be running.
Channel/Queue Inspector	Inspect queues and event parameters values on a channel or queue.	BusinessWare Server must be running. Project must be deployed.
Log Viewer	View logs in text files, binary files, or on channels for diagnostic information about the BusinessWare Server, Integration Servers, or project.	Text-based log files can be viewed at any time. For channel-based logs, the BusinessWare Server must be running.

DEBUGGING PROCEDURE

Debugging allows execution to be halted at breakpoints so that you can view event and BPO data as well as Java variables.

DEBUGGING YOUR PROJECT USING AUTO DEPLOY

The procedure for debugging a project using the **Auto Deploy** option is summarized below. Where appropriate, cross-references direct you to additional information.

Note: If your project is configured for load balancing using clustered Integration Servers, only the master node can be debugged. See [Chapter 21, “Load Balancing”](#) for more information on load balancing.

1. Select **Project > Start Settings**.
2. In the **Start Settings** screen ([Figure 27-1](#)), click the **General** tab. (This tab is selected by default when the screen opens for the first time.)

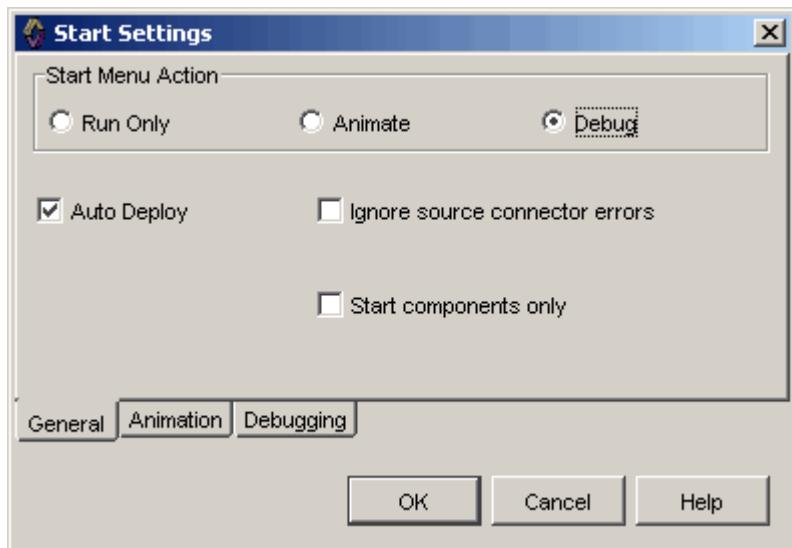


Figure 27-1 Start Settings Dialog Box

3. Click **Debug**. (The **Run Only** option is selected by default.)
4. Select the **Auto Deploy** checkbox to accomplish the following tasks automatically:
 - Set the Integration Server's Enable Debug and Animation property to True.
 - Deploy the project before you start to debug your project.

Note: If the project has been deployed before, the **Undeploy** dialog box appears before the project can be deployed again, prompting you to confirm if the system can overwrite the existing configuration. To overwrite the configuration, select the appropriate options and click **OK**. If you do not want to overwrite the configuration, click **Cancel**, and the operation will be aborted and the project will not be deployed. See [Chapter 22, “Deploying Projects”](#) for more information.

5. Click **OK**.
6. Set breakpoints in your models or code:
 - **Models**—select **Debug > Show Breakpoints in Models** to display indicators showing where breakpoints can be set. Click on an indicator to set the breakpoint.
 - **Code**—in the Source Code Editor, right-click where you want to insert the breakpoint and select **Add/Remove Breakpoint**. Or select **Debug > Add Breakpoint**.
- See “[Working with Breakpoints](#)” on page 27-11.
7. Display the Debugger window to inspect your project while it is being debugged by selecting **View > Debugger**. See “[Debugging Configuration](#)” on page 27-7 for more information.
8. From the **Project** menu, select **Start (Deploy + Debug)**. (The command name changes depending on whether the **Auto Deploy** checkbox has been selected.)
9. Send events into your system using any of these techniques:
 - Use the Event Injector to inject events into a component’s input port. See “[Event Injector](#)” on page 27-19.
 - Have a source connector get external data and send events.
 - Write a Java client application that sends events.
10. When an event reaches a breakpoint and the execution pauses, inspect the data:
 - Use the Event Inspector to inspect the event parameters. See “[Event Inspector](#)” on page 27-25.
 - Use the BPO Inspector to view all the attribute values of the BPO. See “[BPO Inspector](#)” on page 27-26.
11. Resume execution using any of these Debug menu items:
 - Continue
 - Step In
 - Step Out
 - Step Over

See “Stepping Through Your Program” on page 27-5 for more information.

12. To stop the debugging session, select **Project > Stop**. (You have the option to stop the project as well.)
13. Make any necessary changes to your models and code, redeploy, and start another debugging session.

Tip: You can also use toolbar buttons to perform the same functions as the menu commands. To display the debugging toolbar, right-click in the toolbar area and select **Debug**.

Stepping Through Your Program

You can resume execution of your program when paused at a breakpoint by using the following commands in the **Debug** menu.

- **Continue (Ctrl + F5)**—resumes execution after pausing at a breakpoint.
- **Step In (F7)**—takes debugging to the next lower level. It can be done from either a port or nested state entry action in process models. If the flow is stopped at a port, **Step In** will stop at the first breakpoint that it encounters in an integration model or a process model. If the flow is stopped at a nested state entry action, Step In will stop at the first breakpoint it encounters in the nested model being executed.

When debugging Java code, Step In starts execution of a Java method being called by the current method.

Sometimes when you use Step In to resume execution, you may see a **Question** alert dialog box, as shown in [Figure 27-2](#), that prompts you to specify the action to be performed when stepping into a method whose source is not available.

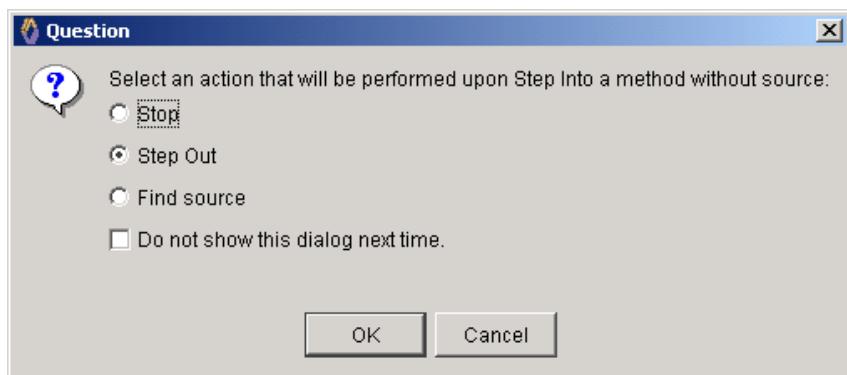


Figure 27-2 Question Alert Dialog Box

1. Select any one of the following actions:
 - **Stop**—to stop debugging
 - **Step Out**—to step out of the method
 - **Find source**—to find the method's source
2. If you wish not to have this dialog box displayed, select the **Do not show this dialog next time** checkbox.
3. Click **OK**.

Step Out (Ctrl + F7)

Returns debugging to the model executed previously (that is, the parent model from which Step In was done) and stops at the first breakpoint (regardless of whether the breakpoint is enabled) in the parent model unless another breakpoint is encountered in the current model. In the source code, execution returns to the calling method.

Step Over (F8)

Executes the next line of source code when debugging in the Source Code Editor. When debugging an integration model or a process model, Step Over from the current breakpoint will stop at the next breakpoint encountered in the flow of events.

DEBUGGING YOUR PROJECT MANUALLY

If you do not select the **Auto Deploy** check box in the **Start Settings** screen, and if the project has not yet been deployed, you must exit the **Start Settings** screen and do the following manually before you can debug your project:

1. Create an Integration Server (by selecting **File > New... > Resources > IntegrationServer**), if an Integration Server has not been created earlier.
2. Enable debugging on the server, and set the debug port number to a number that is not used by other services. The default is 8999.
3. Select **Project > Start Settings > Debug**. Click **OK**.
4. Select **Project > Start (Debug)**.

Tip: To get detailed information in the logs, adjust the trace levels for the server. See “[Configuring Loggers](#)” on page 27-33 for more information.

DEBUGGING CONFIGURATION

A BusinessWare project can have more than one deployment configuration defined. One reason to have multiple configurations is to distinguish a test environment configuration from a production environment configuration. The debugging configuration determines which components will be debugged based on the Integration Server selected.

Vitria recommends that the debugging configuration have all the project components assigned to one Integration Server because only the components running in the selected Integration Server are available during debugging and animation.

If a project has multiple Integration Servers in its deployment configuration, the first Integration Server with debugging/animation enabled, is selected. If you wish to select another Integration Server, you can do so by customizing debugging and animation settings after you deploy your project. See “[Customizing Debug Settings](#)” on page 27-7 and “[Customizing Animation](#)” on page 27-36 for more information.

Note: If your project is configured for load balancing using clustered Integration Servers, only the master node can be debugged. See [Chapter 21, “Load Balancing”](#) for more information on load balancing.

A project must be deployed before it can be debugged. With the BME, it is possible to deploy and debug together, without having to perform these functions separately. See “[Debugging Procedure](#)” on page 27-2 for more information on how you can set up your project to automatically deploy and debug.

It is also helpful to launch the Log Viewer and display the Integration Server log so that you can diagnose any startup problems. You also might want to display the Channel/Queue Inspector and position it at the end of the channel so that you can inspect new events as they are published.

CUSTOMIZING DEBUG SETTINGS

The debug settings take their values from the last deployed configuration in the project. Therefore to customize your debug settings, you must ensure your project is deployed. If the project is not deployed or if the deployment configuration was removed from the project, **Debugging Configuration** in the **Start Settings—Debugging Tab** screen will be displayed as <NONE>.

To customize debug settings:

1. Select **Project > Start Settings**.
2. In the **Start Settings** dialog box, click the **Debugging** tab (Figure 27-3).

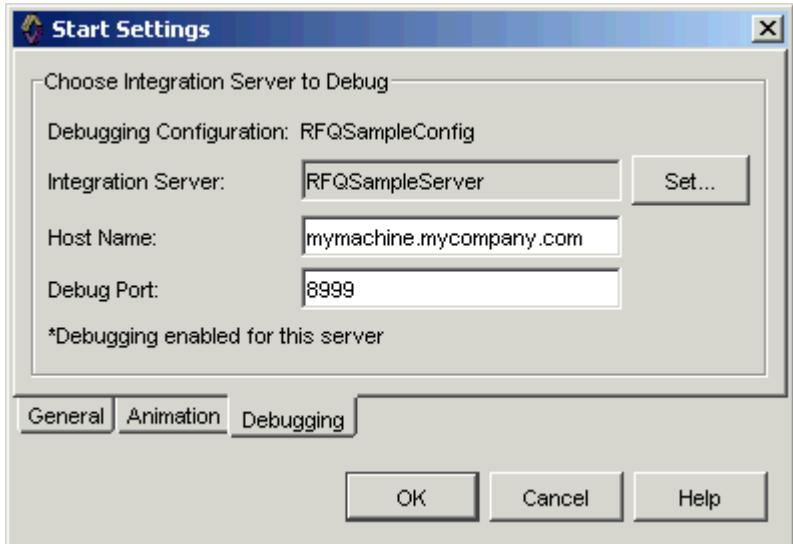


Figure 27-3 Start Settings—Debugging Tab Screen

3. The default Integration Server is displayed. Select **Set** to choose a different Integration Server (Figure 27-4).



Figure 27-4 Server Selection Box

4. In the **Set Integration Server to Debug** screen, select the Integration Server and click **OK**. You are returned to the **Start Settings** screen.
5. The **Host Name** is automatically displayed. You may change it, if needed.

6. The debug port number is automatically displayed from the Integration Server properties.

To change the port settings, you must change the Integration Server's Debugging and Animation Port property and redeploy the project. See [Chapter 6, "Integration Model Basics"](#) for more information on Integration Server properties.

Note: If the Integration Server is not enabled for debugging and animation, and if the project is not deployed, the **Debug Port** field will be empty.

7. Click **OK** to set the debugging configuration with the new settings.

DEBUGGER WINDOW

The BusinessWare Debugger lets you step through every aspect of your business solution, including all integration models, process models, and Java code. The Debugger window ([Figure 27-5](#)):

- Includes facilities for viewing breakpoints.
- Lets you set watches and track the values of variables as the code is executing.
- Lets you inspect and monitor threads, call stacks, variables, and classes while the project is running.

DEBUGGING AND ANIMATION

Debugger Window

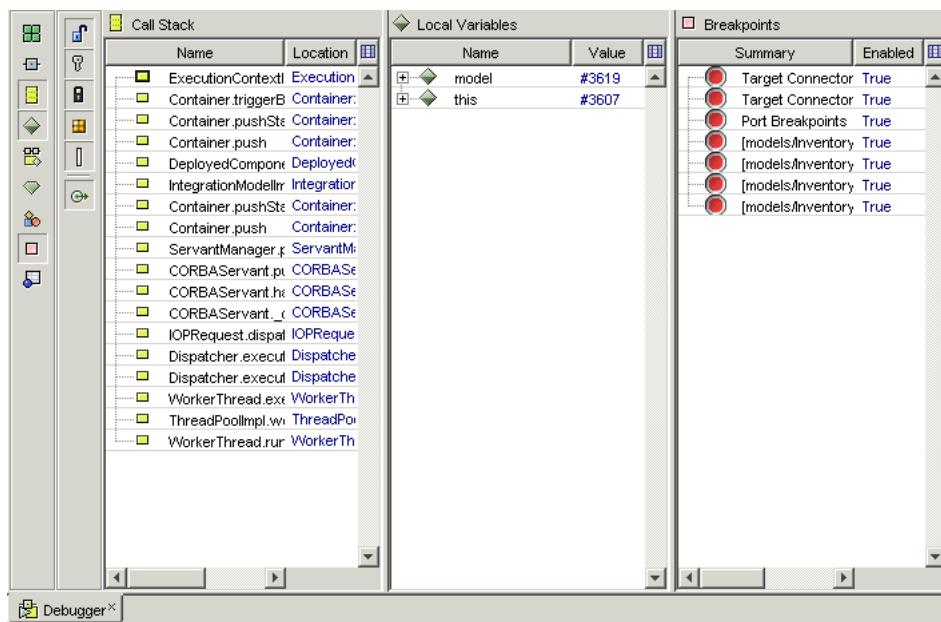


Figure 27-5 Debugger Window

The Debugger window consists of the views listed in [Table 27-2](#).

Table 27-2 Debugger Window Toolbar

Button	View	Description
	Sessions	Lists the current debugging session being run. Note: BME supports only one debugging session at a time.
	Threads	Lists all the running threads in the current debugging process and indicates which thread is currently stopped. Clicking on a stopped thread reveals the Execution Stack Trace and local variables for the function code where the thread is stopped.
	Call Stack	Lists the execution stack trace for a thread paused at a breakpoint.
	Local Variables	Lists all values of variables local to a method or arguments in a method call.
	All in One	Combines sessions, threads, call stacks, and local variables in one view.
	Watches	Lists all watches set on variables in the project. Whenever a variable comes into scope during execution, its value is posted in the watch properties. You can add or delete watches in this tab.
	Classes	Lists all the mounted classes.

Table 27-2 Debugger Window Toolbar (Continued)

Button	View	Description
	Breakpoints	Displays all the breakpoints set in the project. You can use this tab to quickly enable, disable, or delete multiple breakpoints. To see where a breakpoint is located in the model, right-click the breakpoint and select Go to Source .
	Properties	Displays properties for the currently selected item in the Debugger window.

Debugger window also includes a filter toolbar. [Table 27-3](#) lists the filter toolbar buttons.

Table 27-3 Debugger Window Filter Toolbar

Button	Name	Description
	Public	Displays public members.
	Protected	Displays protected members.
	Private	Displays private members.
	Package Private	Displays package members.
	Static	Displays static members.
	Inherited	Defines groups of inherited members. By default, inherited members are grouped by superclass.

WORKING WITH BREAKPOINTS

A breakpoint specifies a location where project execution will pause. Breakpoints are useful to monitor the inner workings of your project at certain predefined points. You can set breakpoints in any model or code in your project.

Note: Breakpoints set in code are useful only for debugging; they do not work for animation.

Setting Breakpoints in Models

In process models, breakpoints can be set to pause before execution of:

- Entry actions
- Exit actions
- Transition actions

Note: You cannot set breakpoints on a transformer state in a process model because transformer states have no entry or exit actions.

In integration models, breakpoints can be set on input ports of components and target connectors.

When you select **Debug > Show Breakpoints in Models**, grey indicators appear at all the *possible* locations for breakpoints.

Click on a breakpoint to set it. If the indicator turns red, the breakpoint is enabled. During debugging, a white arrow is posted on the breakpoint where the execution is currently paused. [Table 27-4](#) shows the color-coding used on breakpoint indicators.

Table 27-4 Breakpoint Icons

Indicator	Description
	Potential breakpoint location
	Breakpoint set
	Debugging stopped at current breakpoint

Note: If your project is running but stopped at a breakpoint, the information on the status line will indicate that the project is running.

Setting Breakpoints in the Code

You can set breakpoints on any line in the source code or in a method for any custom Java class you created.

Use any of the following techniques to set breakpoints in the code:

- Right-click a line of code and select **Toggle Breakpoint**.
- Right-click in the **Breakpoints** view of the Debugger and select **New Breakpoint....**

The **Add Breakpoint** menu item displays a dialog box, shown in [Figure 27-6](#), in which you:

- Select the type of breakpoint to set.
- Specify breakpoint settings depending on the breakpoint type selected.
- Specify actions to be performed when execution encounters the breakpoint.

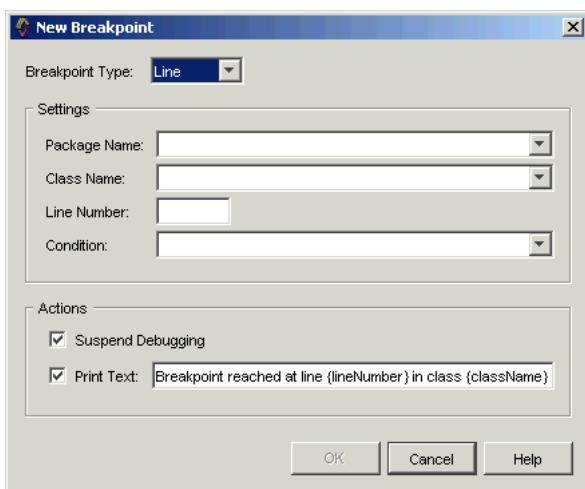


Figure 27-6 Add Breakpoints Dialog Box

The breakpoints you add in the code are not represented visually in the model diagrams.

Selecting Breakpoint Types

You can set a breakpoint in any of the following:

- Line
- Method
- Variable
- Exception
- Class
- Thread

Specifying Breakpoint Settings

Specify breakpoint settings based on the breakpoint type selected.

Line

1. Enter the name of the class to set the breakpoint.
2. Specify the line number in the code.
3. Add a condition (a boolean expression using any in-scope variables) to the breakpoint.

Class

- 1.** Enter the class name in the **Class Filter** field.
- 2.** Select the **Exclusion filter** checkbox to specify if the class name is exclusive of the debug process.
- 3.** Select the type of action to trigger the breakpoint:
 - Class prepare—when the class is loaded into the virtual machine.
 - Class unload—when the class is unloaded from the virtual machine.
 - Class prepare or unload—when the class is either loaded or unloaded.

Variable

- 1.** Enter the name of the class that contains the variable.
- 2.** Enter the field name to set the breakpoint.
- 3.** Specify when execution must be stopped.
 - Variable Access—to stop your program when it accesses a variable in the specified class and field.
 - Variable Modification—to stop your program when the value of the variable changes.
- 4.** Add a condition so that the breakpoint will occur when the condition evaluates to True.

Exception

- 1.** Enter the full name of the exception class, for e.g.,
`java.lang.InternalError`.
- 2.** Specify when execution must be stopped by selecting any one of the following options:
 - Exception caught
 - Exception uncaught
 - Exception caught or uncaught
- 3.** Add a condition to the breakpoint.

Method

- 1.** Enter the name of the class that contains the method to set the breakpoint.
- 2.** Specify the method to set the breakpoint or select the **All methods for given classes** checkbox to select all the methods in the specified class.
- 3.** Add a condition to the breakpoint.

Thread

1. Specify where the breakpoint must be triggered by selecting any one of the following options:
 - Thread start
 - Thread death
 - Thread start or death

Specifying Actions

- Write a text message to be printed in the Output window when debugging reaches a breakpoint.
- Select **Suspend** debugging checkbox to have execution pause at the breakpoint.

Enabling and Disabling Breakpoints

To enable or disable a breakpoint in a model:

1. If breakpoints are shown in a model, click on a breakpoint to enable/disable it. The breakpoint will appear red if enabled and grey if disabled.

To enable or disable a breakpoint in a code:

1. In the Debugger window, click the **Breakpoints** button.
2. In the **Breakpoints** column, select a breakpoint and in the **Enabled** column select True or False from the drop-down list.
3. Select **Properties** from the shortcut menu, and change the Breakpoint Enabled property to True or False.

To enable or disable all breakpoints:

1. In the Debugger window, click the **Breakpoints** button.
2. Right-click the **Breakpoints** column, and select **Enable All** or **Disable All** from the shortcut menu.

Removing Breakpoints

You can remove breakpoints using any of these methods:

- Click an enabled breakpoint (red) in the model.
- Click a highlighted line in the code and select **Toggle Breakpoint**.
- Right-click a breakpoint in the **Breakpoints** column and select **Delete**.
- Right-click in the **Breakpoints** column in the Debugger window and select **Delete All**.

WORKING WITH WATCHES

A watch is used to monitor the value of a local variable or class instance as debugging progresses. Whenever the variable is in scope (for example, visible to the execution engine), its value is obtained and posted in the **Watches** tab of the Debugger window. If you suspect that a variable is being set to an invalid or unexpected value, a watch can help you determine exactly when and why that behavior is occurring.

To add a watch in the Debugger window:

1. Click the Watches button. In the **Watches** column, right-click and select **New Watch....**
2. Specify the variable you want to watch.

To add a watch in the source code:

1. In the Source Code Editor, right-click on a variable and select **New Watch....**
2. Specify the variable you want to watch.

To see the value of a watched variable:

1. In the **Watches** column of the Debugger window, right-click the watch and select **Properties** from the shortcut menu. The properties, including current value, are displayed in the properties pane.

To delete a watch:

1. Click on a watch in the **Watches** column of the Debugger window. Delete the watch using any one of the following techniques:
 - Right-click a watch and select **Delete** from the shortcut menu.
 - Right-click in the Watches column and select **Delete All** from the shortcut menu.
 - Select the watch and press the **Delete** key.

INSPECTING THREADS

Using the **Threads** pane in the Debugger window, you can quickly check the local variables and the stack trace for the thread where the execution is paused.

To inspect threads:

1. Click the **Threads** button.

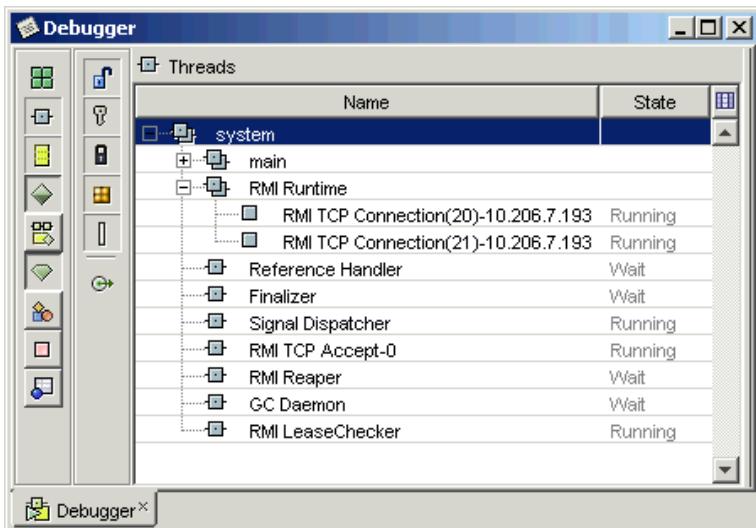


Figure 27-7 Debugger Window-Threads Pane

2. Expand the **Threads** node to locate the current thread.

The thread that caused the stoppage is indicated by a dark orange icon of a gear; the other threads have yellow icons.

3. Click on the current thread.

Two nodes appear: CallStack and Locals.

4. To check variables:

- a. Expand the **Locals** node. The Debugger displays the names and current values of all local variables for the function where the thread is stopped.
- b. Expand a variable node to see its structure.

5. To view the stack trace, expand the **CallStack** node.

The Debugger lists all the method calls made up until execution stopped. The method that was executing when the program stopped is at the top of the stack.

You can also use the shortcut menu in the **Threads** tab to perform any of these tasks:

- Suspend a thread
- Resume a thread
- Switch to the selected thread

INSPECTING CALLSTACKS

To view the stack trace:

1. Click the **CallStack** tab in the Debugger window.
2. Expand the **CallStack** node.

The Debugger lists all the method calls made until execution stopped. The method that was executing when the program stopped is at the top of the stack.

INSPECTING VARIABLES

To check variables:

1. Click the **Variables** tab in the Debugger window.
2. Expand the **Variables** node. The Debugger displays the names and current values of all local variables for the function where the thread is stopped.
3. Expand a variable node to see its structure.

To create a fixed watch:

1. To take a snapshot of the variable data, right-click the variable in the Debugger window.
2. Select **Create Fixed Watch**.

Note: Fixed watches are not updated; you must clean up by deleting them.

INSPECTING CLASSES

To inspect classes:

1. Click the **Classes** button in the Debugger window.
A list of classes will be displayed.

INSPECTING SESSIONS

To inspect sessions:

1. Click the **Sessions** button in the Debugger window.
The status of the current debugging session is displayed.

EVENT INJECTOR

The Event Injector is used to interactively create test events. You can create a single event or collections of events. The events that you create can be injected into a running project or they can be saved to a file for use at a later time.

The Event Injector simplifies testing because you do not have to rely on external applications for input.

You can inject events directly from an Extensible Markup Language (XML) file.

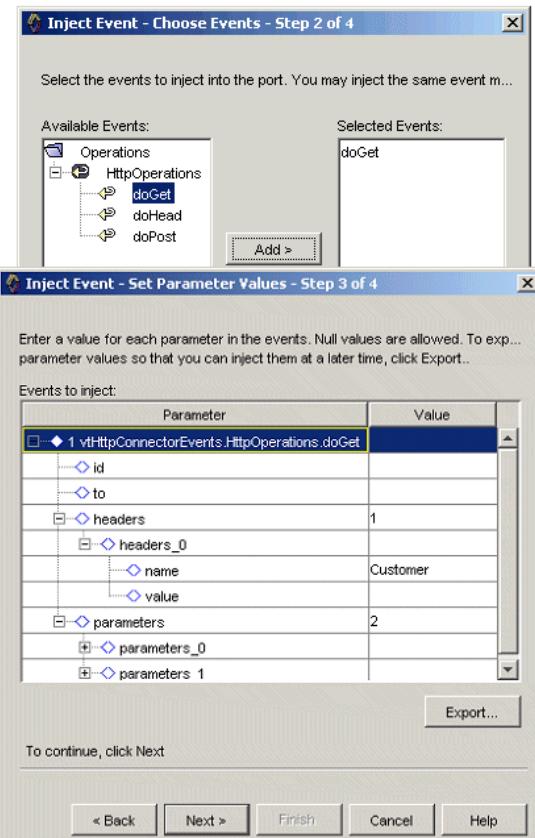


Figure 27-8 Creating Events in the Event Injector

Although the Event Injector is very useful during debugging, it can be used at other times. The only requirements are that the BusinessWare Server and the project be running.

USING THE EVENT INJECTOR

You can inject events into the input ports of the following integration model components:

- Integration component
- Process component

Note: The Transaction Control property for the input port must be set to Required or RequiresNew, both of which allow the runtime to start a new transaction. If Transaction Control is set to Mandatory, the runtime will throw an exception since it expects the invoking client to pass a transaction context. You set a port's Transaction Control property by right-clicking the input port on an integration or a process component and selecting **Properties**. See [Chapter 25, “Transaction Management”](#) for more information.

Events created using the Event Injector can be saved and reused, including IDL/DTD events and Java events. The saved files retain the event parameter values and can be used to recreate events for another debugging session without having to re-define the values. The saved data can be edited using the Event Injector.

To inject events:

1. To access the Event Injector, you must select the input port on an integration component or process component. Use either of these techniques:
 - Right-click the input port, and select **Tools > Inject Event....**
 - Select the input port; then select **Debug > Inject Event...** from the main menu.

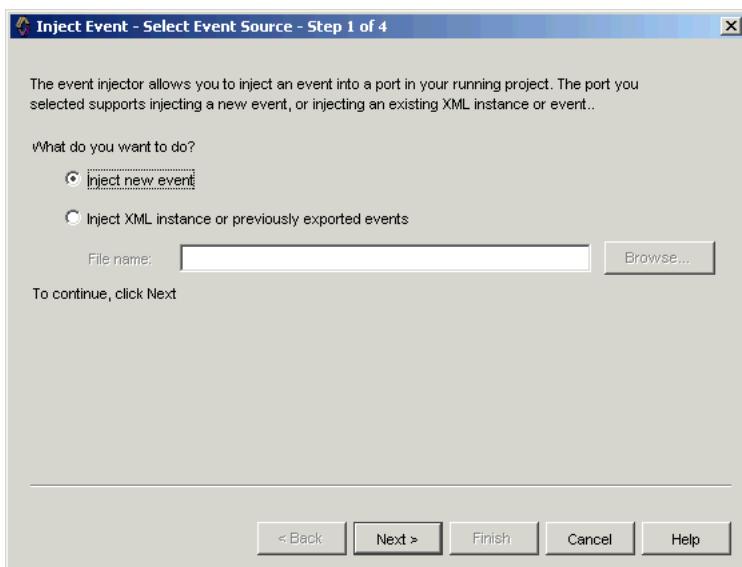


Figure 27-9 Event Injector

2. Specify the source of the events by selecting one of these options:
 - **Inject New Event**—displays panels in which you select the types of events you want to generate and then enter values for their parameters. You can save these events in a file for reuse. See the *BME Help* for more information on how to use these panels.
 - **Inject XML instance or previously exported events**—lets you use a file containing events that you created in the Event Injector and exported. Automatically detects if it is an exported event.
 - Lets you select an XML file from an outside source (that is, the file was not generated in the Event Injector). The corresponding DTD file must have been copied into the project and converted to type before deployment. See “[Using an Externally Created XML File](#)” on [page 27-24](#).
 - If the port type into which you are injecting events specifies XMLEvents, and the XML is determined to be a previously exported event, the exported event is injected or a File can be specified which is used as payload data for an xmlEnvelopeEvent.
3. Select the XML file or the events to inject.

The following types of events can be injected using the Event Injector:

- Java events

- XML events
 - IDL events
- 4.** Set parameter values.

You can enter values for all event parameters except for the parameter types listed below; the Event Injector assigns a null value to these parameters.

- Object references in IDL/XML events
- Java interface in Java events
- Java class (without a default constructor in Java events)

If the port is untyped, you can select from all the events and operations associated with the current model, the current project, or any dependent project module (such as the BusinessWare project).

If the port is typed, the Injector will display events corresponding to the port's type, and you can select an event from this list. If the port type is set to XMLEvents, you can select an XML document to be the payload for the event to be injected as shown in [Figure 27-10](#).

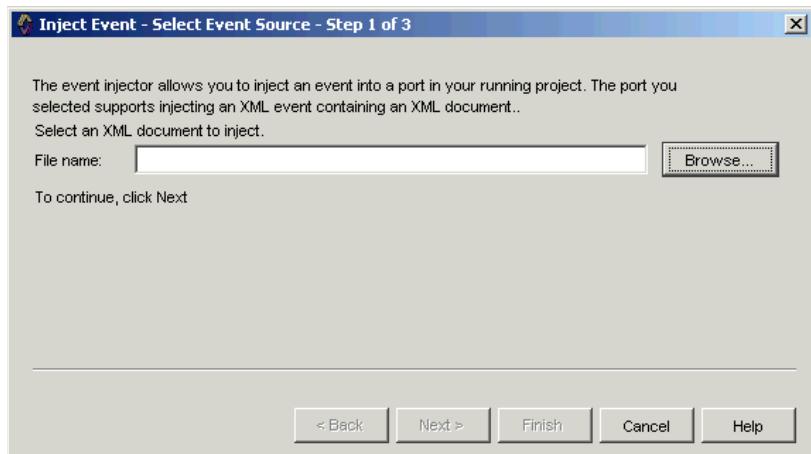


Figure 27-10 Selecting an XML Document

After specifying the XML document to be injected, click **Next** to set the parameter values. Browse the XML document by clicking the browse button next to **XML Envelope Payload...** in the Set Parameters Value screen. The XML Explorer ([Figure 27-15](#)) opens and you can browse the parameters by expanding the XML document.

When injecting sequences or arrays, specify the number of elements in the sequence or the array. The default value is 0. Expand the sequence or array parameter to enter values for each parameter in the sequence or array.

For example, if you select a CDRArrayEvent to inject, the default value of the array is 0. If you change the value to 1, you can expand the CDRArray and enter values for the eventSpec and eventdata parameters.

Events will be injected in the order in which they are added to the target list.

To see results, you can examine event parameters when the process pauses at breakpoints, using the Event Inspector (see “[Event Inspector](#)” on page 27-25). The result of any operation you inject is displayed in the Output window so you can view the resulting parameter values. To resume the process, select **Debug > Continue**.

Note: The Event Injector may fail to produce the expected results if you change the models after the project is deployed. One common scenario is when you change the port type after deploying a project, and then inject events without redeploying the project. In this case, the results will *not* be based on the change made to the port type, instead, it will be based on the port type specified in the deployed version. Therefore, make sure to redeploy the project if you change the models before you inject events.

You can re-inject the *most recently injected* set of events into the *same* port using either of these methods:

- Right-click the input port and select **Tools > Inject Again....**
- Select the input port and then select **Debug > Inject Again...** from the main menu.

For more information on injecting events, see the *BME Help*.

USING AN EXTERNALLY CREATED XML FILE

An externally created XML file can be used in the Event Injector only if its associated DTD file is copied into the project and converted to the required type using **Tools > ConvertToType**.

Example: DTD defining an Order event.

```
<!ELEMENT order (name, address, item+)>

<!ELEMENT name (first, last, middle?)>
<!ATTLIST name salutation CDATA #IMPLIED>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT middle (#PCDATA)>

<!ELEMENT address (street1, street2?, city, zip, country?)>
<!ELEMENT street1 (#PCDATA)>
<!ELEMENT street2 (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT zip (#PCDATA)>

<!ELEMENT item ((partname | partnumber), price, quantity)>
<!ELEMENT partname (#PCDATA)>
<!ELEMENT partnumber (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT quantity (#PCDATA)>
```

After the project is built and deployed, order events can be injected through the Event Injector simply by selecting an XML file that conforms to the DTD.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE order SYSTEM "orderdef.dtd">

<order>
    <name salutation="Ms">
        <first>Jane</first>
        <last>Doe</last>
    </name>

    <address>
        <street1>1 Main Street</street1>
        <street2>Apartment 1</street2>
        <city>Anytown</city>
        <zip>12345</zip>
```

```
        <country>USA</country>
    </address>

    <item>
        <partname>Widget 4</partname>
        <price>0.99</price>
        <quantity>10</quantity>
    </item>

    <item>
        <partnumber>10107</partnumber>
        <price>34.50</price>
        <quantity>2</quantity>
    </item>

</order>
```

EVENT INSPECTOR

The Event Inspector displays the parameter values of an event or operation that is stopped at a breakpoint. To access the event inspector, select **View > Event Inspector**.

Note: This tool is available only while the debugger is paused at a breakpoint.

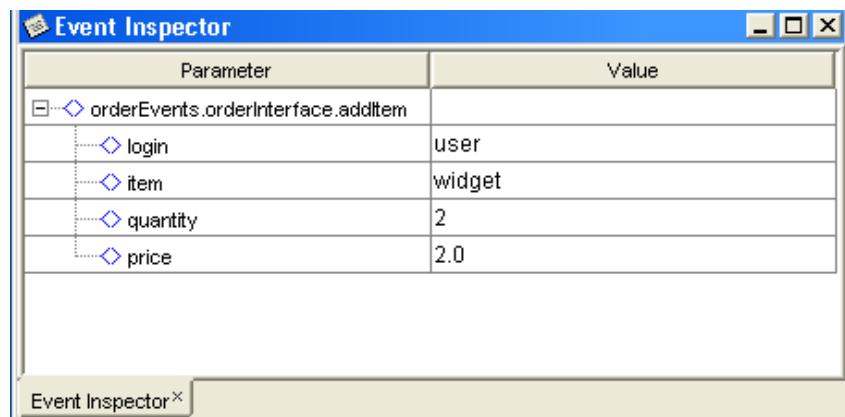
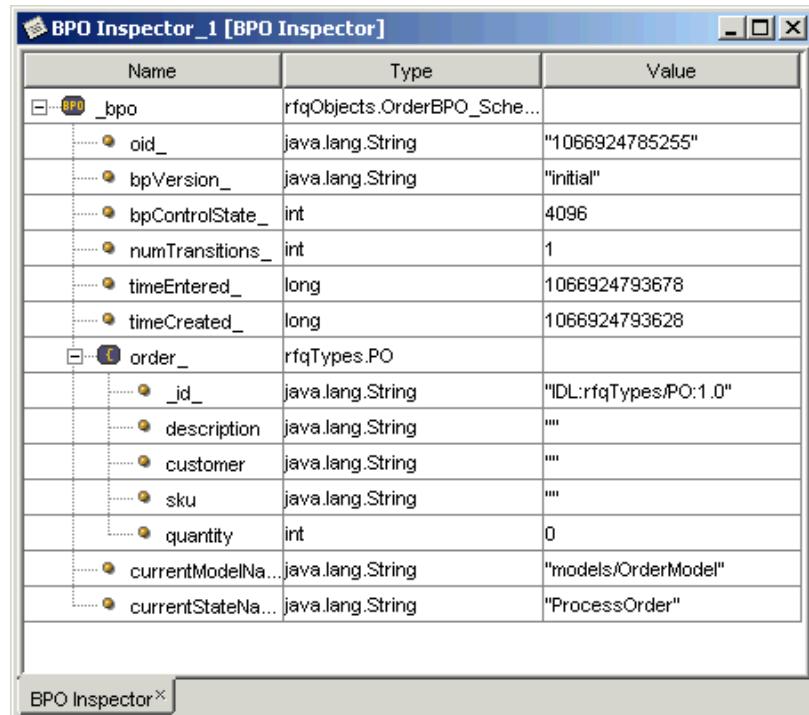


Figure 27-11 Event Inspector

BPO INSPECTOR

The BPO Inspector displays the attribute values for a business process object that is stopped at a breakpoint. To access the inspector, select **View > BPO Inspector**.

Note: This tool is available only when the debugger is paused at a breakpoint in a process model and a BPO is associated with the model.



The screenshot shows the 'BPO Inspector_1 [BPO Inspector]' window. It contains a table with three columns: Name, Type, and Value. The table shows attributes of a BPO object named '_bpo' and its nested 'order_' object.

Name	Type	Value
_bpo	rfqObjects.OrderBPO_Sche...	
oid_	java.lang.String	"1066924785255"
bpVersion_	java.lang.String	"initial"
bpControlState_	int	4096
numTransitions_	int	1
timeEntered_	long	1066924793678
timeCreated_	long	1066924793628
order_	rfqTypes.PO	
id	java.lang.String	"IDL:rfqTypes/PO:1.0"
description	java.lang.String	""
customer	java.lang.String	""
sku	java.lang.String	""
quantity	int	0
currentModelNa...	java.lang.String	"models/OrderModel"
currentStateNa...	java.lang.String	"ProcessOrder"

Figure 27-12 BPO Inspector

CHANNEL/QUEUE INSPECTOR

The Channel/Queue Inspector lists events in a channel and lets you view the event data.

Click on an event to display a hierarchical list of its parameters and their current values. For easier reading, double-click on the value to display it in a separate text box.

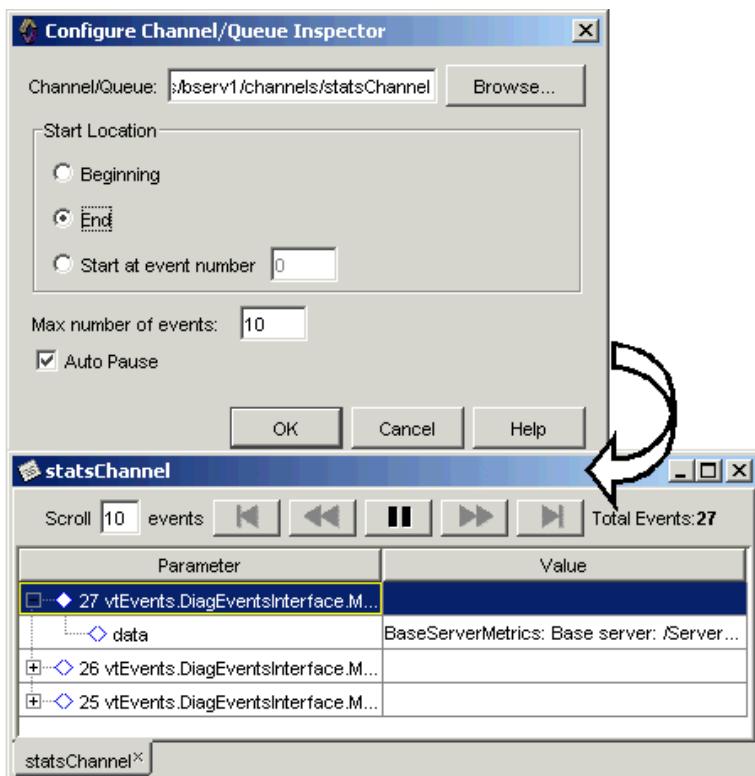


Figure 27-13 Channel/Queue Inspector

Note: Although the Channel/Queue Inspector is very useful during debugging, it can also be used at other times. The only requirements are that the BusinessWare Server be running and the project's channels exist.

The Channel Inspector displays XML attribute values in a single line text field. If an XML attribute value is present, a browse button appears in the line as shown in [Figure 27-14](#).

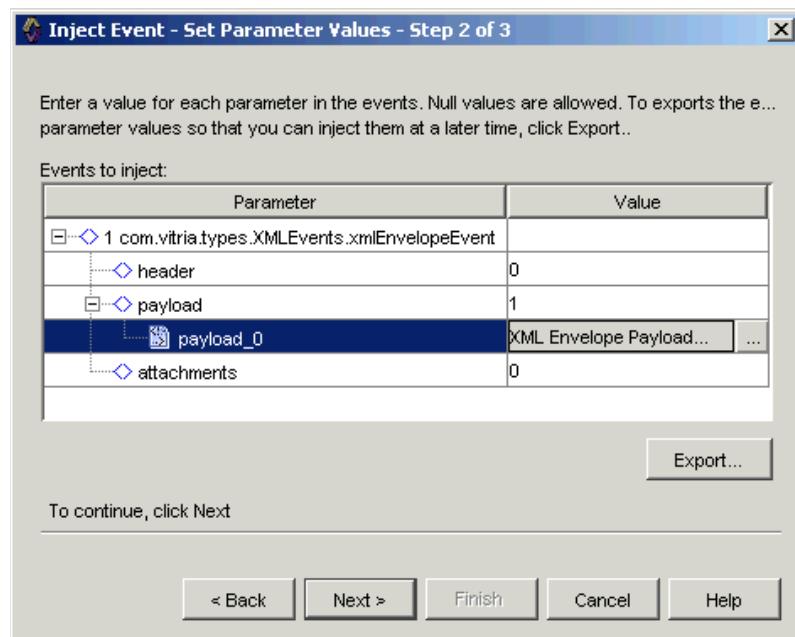


Figure 27-14 Event Injector

To browse the content, click the browse button to open a browser that displays the content in an XML tree. [Figure 27-15](#) shows an example of the XML Explorer.

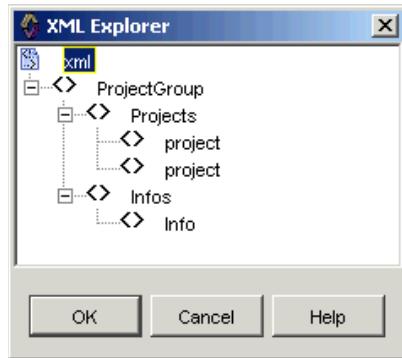


Figure 27-15 XML Explorer

USING THE CHANNEL/QUEUE INSPECTOR

You can start inspecting events at the beginning or end of the channel/queue or at a specific event.

1. Launch the Channel/Queue Inspector by either of these methods:

- Select **View > Channel/Queue Inspector**.
- Right-click on a channel connector component in the Integration Model Editor, and select **Tools > Channel/Queue Inspector**.
- Right-click on a channel in the Explorer, and select **Tools > Channel/Queue Inspector**.

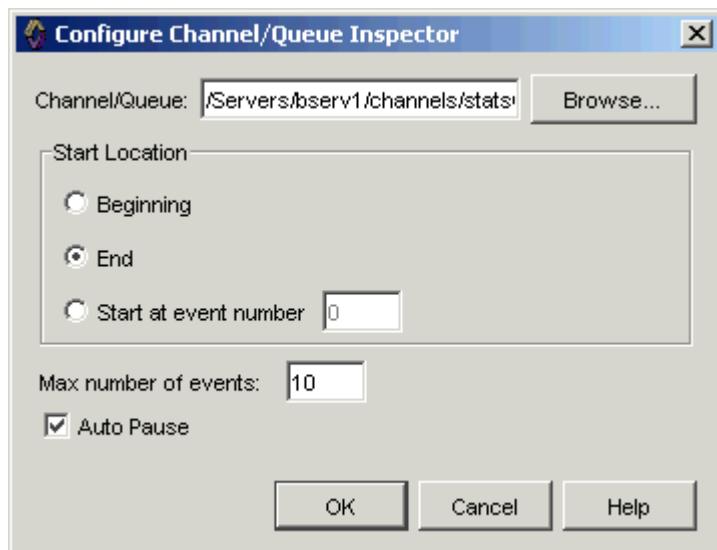


Figure 27-16 Channel/Queue Inspector

2. In the **Configure Channel/Queue Inspector** dialog box (Figure 27-16), specify:
 - a. Channel or queue you want to inspect. (Use the Browse button to select a channel or queue.)
Note: If the Channel/Queue Inspector is opened by right-clicking the channel connector component, the **Browse** button will be disabled because you have already selected the channel to inspect.
 - b. Channel or queue position where you want to begin inspecting.

- c. Maximum number of events to be displayed at one time.
 - d. If you want the inspector to pause automatically after displaying the maximum number of events by checking the **Auto Pause** checkbox.
3. Click **OK**.

Displaying Events

If you turned **Auto Pause** on, the channel/queue inspector displays the maximum number of events and then pauses. It does not display the next set of events until you click the **Start** button .

If you turned **Auto Pause** off, as each new event enters the channel, the channel/queue inspector adds the new event to the bottom of the list and removes the topmost event. You can click the **Pause** button  to stop the scrolling events.

While the channel/queue inspector is paused, you can display sets of events by clicking the buttons at the top of the inspector:

-  to jump back to your starting location on the channel
-  to scroll back the number of events specified in the Scroll field
-  to resume the flow of events (the Start and Pause buttons toggle)
-  to scroll ahead the number of events specified in the Scroll field
-  to jump to the end of the channel

IMPORTANT: For a queue, the Channel/Queue Inspector may not show any events if subscribers are pulling events when they arrive. If no subscribers are active, the events can be browsed on a queue.

LOG VIEWER

Depending on your configuration, diagnostic logs for your BusinessWare Server and Integration Server may be sent to a text file, a binary file, or a channel (in binary format). The Log Viewer:

- Displays logs from any of these sources (see [Figure 27-19](#)).
- Offers helpful sorting and filtering options for binary logs, which are automatically displayed in the local language.

Example: A developer in Japan will see his log messages in Japanese, but when he sends the logs to a colleague in the United States, they are displayed in English.

VIEWING LOGS

You can view log files at any time. To view log channels, the BusinessWare Server that contains the channel must be running.

To run Log Viewer from the BME:

1. Select **View > Logs**. The Configure Log View screen appears, as shown in Figure 27-17.

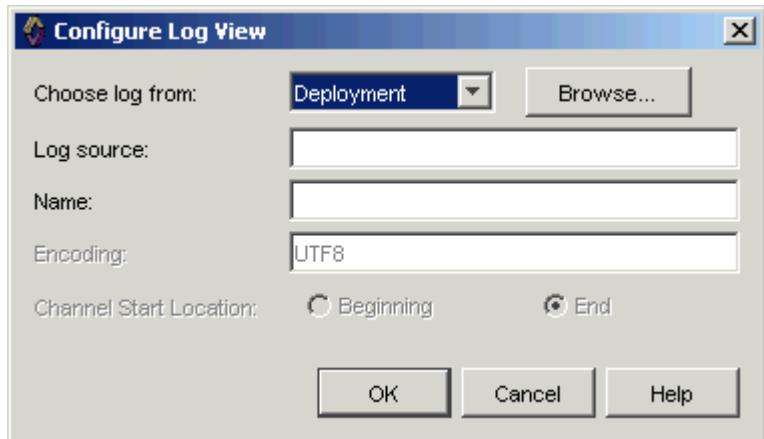


Figure 27-17 Configure Log View

2. Choose the log you want to view, by browsing **File System** or **Deployment**.

Deployment displays the deployment configurations in hierarchical fashion, with all the Integration Servers listed for each configuration and all the loggers listed for each server ([Figure 27-19](#)).

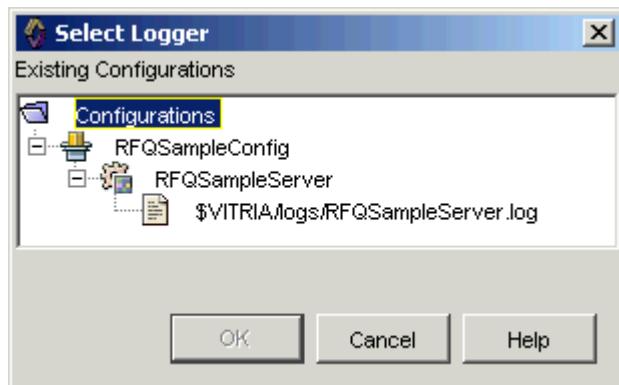


Figure 27-18 Select Channel Logger

3. Enter a **Name** to be displayed on the tab for this log.

You can display multiple logs in the viewer and switch from one to another using the tabs for each log.

4. If you have chosen a channel log source, specify whether you want to start viewing log entries at the beginning or end of the channel.

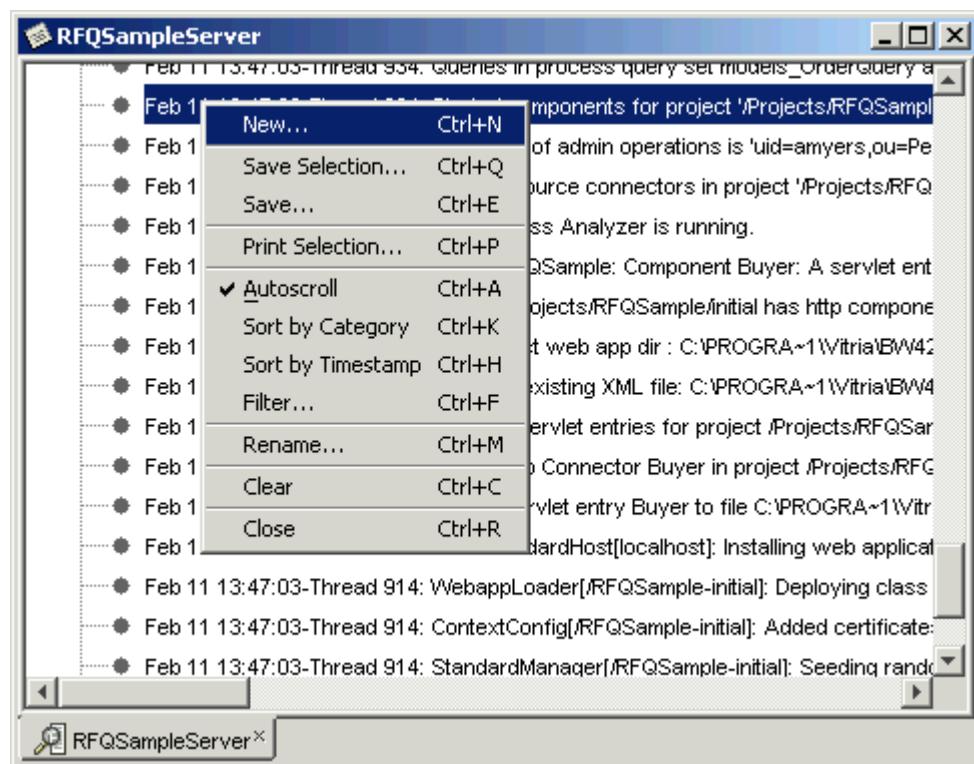


Figure 27-19 Log Viewer with Context Menu Displayed

Using the shortcut menu in the Log Viewer, you can:

- Display another log by selecting **New**
- Print a selection from the log
- Save a selection from the log to a file
- Sort logs messages by category or timestamp (binary logs only)
- Filter log messages by category, module, or timestamp (binary logs only)

CONFIGURING LOGGERS

By default, the log entries for each Integration Server in your project are written to a single text file. However, you can add as many additional text file loggers, binary file loggers, and channel loggers as you wish. See [Chapter 6, “Integration Model Basics”](#) for more information.

ANIMATION

Animation is a good way to visualize projects that are running. Animation adds interest and flair to your demonstration and helps your users understand the content better. It clearly shows you where you are, and where you have been. You can watch events flow through your project, and as they flow, the path of the flow, the current port, state, or transition is highlighted.

Note: If your project is configured for load balancing using clustered Integration Servers, only the master node supports animation. See [Chapter 21, “Load Balancing”](#) for more information on load balancing.

You can control animation by:

- Animating all models or choosing the models to animate.
- Setting the speed at which animation runs ranging from slow to normal.
- Enabling breakpoints in strategic places for animation to pause.

Keep these points in mind when you configure your project for animation:

- An Integration Server must exist with animation enabled by setting the Enable Debugging and Animation property to True.
- A deployment configuration must exist with components partitioned to the Integration Server.
- All the components you want to animate must be running in a single Integration Server.

Note: The BME offers an easy method by which you can configure your project to automatically enable animation on an Integration Server, deploy the project, and start animation, all in a single sequence. See [“Animating a Project Using Auto Deploy” on page 27-35](#) for more information.

Because animation accurately depicts server-side runtime behavior, visualization of concurrently executing functionality may not provide results that are easily interpreted. Specifically, animation demonstrations should avoid the following patterns:

- **Multithreaded projects**—You may have multiple source connectors in a project, but you must ensure that only one of them is processing events at any given time. Additionally, channel and queue source connectors must not use the fan-out capability, as this will result in multiple simultaneous threads processing the events.
- **Multiple fan-in**—a project with multiple sources connected to the same port will be unable to determine the actual source of the event.

ANIMATING A PROJECT USING AUTO DEPLOY

To animate a project using the Auto Deploy option:

1. Select **Project > Start Settings** ([Figure 27-20](#)), and then click the **General** tab (if the tab is not already selected by default).
2. In the **Start Settings** screen, click **Animate**.

By default, all the integration models and process models in your projects are selected for animation. However, you can change the default settings by choosing specific models for animation. See “[Customizing Animation](#)” on [page 27-36](#) for more information.

3. Select the **Auto Deploy** checkbox to accomplish the following tasks automatically:
 - Set the Integration Server’s Enable Debug and Animation property to True if set to False.
 - Deploy the project.

Note: If the project has been deployed before, the **Undeploy** dialog box appears before the project can be deployed again, prompting you to confirm if the system can overwrite the existing configuration. To overwrite the configuration, select the appropriate options and click **OK**. If you do not want to overwrite the configuration, click **Cancel**, and the operation will be aborted and the project will not be deployed. See [Chapter 22, “Deploying Projects”](#) for more information.

4. Set breakpoints in your models for animation. (Breakpoints set in code do not work for animation.)
5. From the **Project** menu, select **Start (Deploy + Animate)**. (The command name changes depending on whether the **Auto Deploy** checkbox has been selected.)
6. Send events into your system using any of these techniques:
 - Use the Event Injector to inject events into a component’s input port. See “[Event Injector](#)” on [page 27-19](#).
 - Have a source connector get external data and send events.
 - Write a Java client application that sends events.
7. When an event reaches a breakpoint and the execution pauses, inspect the data:
 - Use the Event Inspector to inspect the event parameters. See “[Event Inspector](#)” on [page 27-25](#).
 - Use the BPO Inspector to view all the attribute values of the BPO. See “[BPO Inspector](#)” on [page 27-26](#).

Note: You can enable breakpoints during animation.

8. Resume execution by selecting **Debug > Continue** or by pressing **Ctrl + F5**.
9. Let animation run its course.
10. To stop animation, select **Project > Stop**. (You are given the option to stop the project when you stop animation.)

SETTING YOUR PROJECT MANUALLY FOR ANIMATION

To set your project manually for animation:

1. If **Auto Deploy** is not selected in the **Start Settings** screen, and if the project is not deployed, complete the following steps manually:
 - a. Create an Integration Server, if one does not exist.
 - b. Enable debugging on the server, and set the debug port number to a number that is not used by other services. The default is 8999.
 - c. Partition all components to the Integration Server and select the server for animation.
 - d. To get detailed information in the logs, adjust the trace levels for the server.
- Note:** See “[Debugging Configuration](#)” on page 27-7 and “[Configuring Loggers](#)” on page 27-33 for more information.
2. Deploy the project.

CUSTOMIZING ANIMATION

To choose models for animation:

1. Select **Project > Start Settings**.
2. In the **Start Settings** screen, click the **Animation** tab.

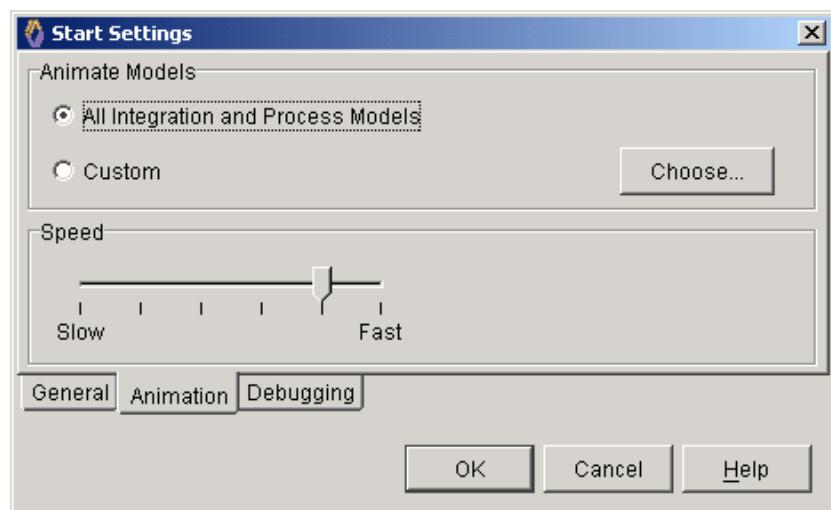


Figure 27-20 Start Settings (Animation) Dialog Box

3. Click **Custom**, and then click the **Choose** button.

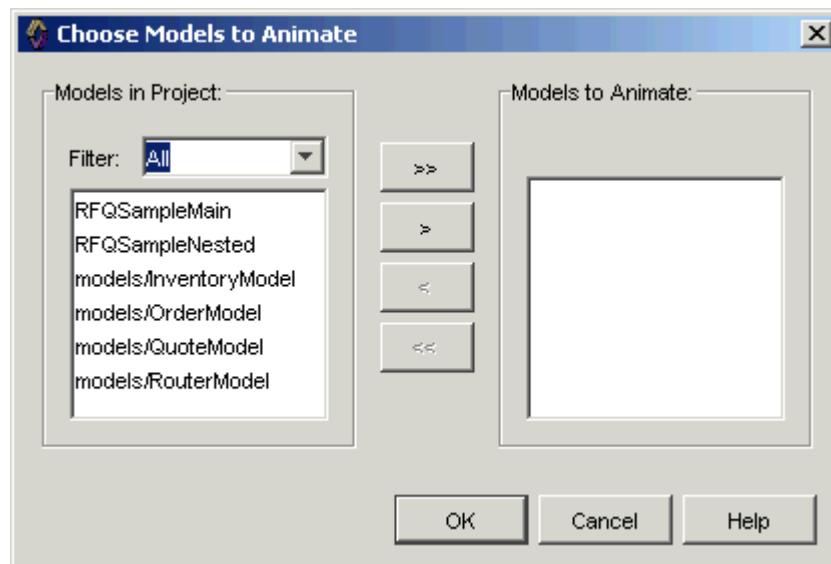


Figure 27-21 Choose Models to Animate

- a. To filter the type of model to be displayed, select **Integration Models** or **Process Models** from the **Filter** list. **All** models is the default.
 - Use button to select all the models or to select specific models.
 - Use or buttons to deselect (not include) the models.
- b. Click **OK**. You are returned to the **Start Settings** ([Figure 27-20](#)) screen.
4. Set the speed to specify the amount of time animation should take.
5. Click **OK**.

CLEARING ANIMATION

When animation is completed on the current event, the travelled paths continue to be highlighted. These paths are cleared when animation is stopped. You can also clear animation by selecting **Debug > Reset Animation**. This will not cause server connection to be lost, and you can continue animating the next event.

PAUSING ANIMATION

You can pause animation by setting and enabling breakpoints. This will allow you to pause at the specified location.

To continue animation, do one of the following:

1. Press **Ctrl + F5**.
- or
- Select **Continue** from the **Debug** menu.
- or
- Select the **Continue** toolbar button in the **Debug** toolbar, if displayed.

ANIMATING INTEGRATION MODELS

The following components in an integration model are animated:

- Ports and states to show that they have been visited.
- Wires connecting ports to show the flow of events.
- Integration model to show the events received and sent.

When you pause animation in an integration model, the flow will stop at the port where a breakpoint is set.

Note: Models cannot be edited during animation.

ANIMATING PROCESS MODELS

Animation provides you with visual cues as to what is happening within a process model by helping you to clearly visualize the flow of events and the entry/exit action process.

The following components in a process model are animated when executed:

- States
 - Start
 - End
 - Fork
 - Join
- Actions
 - Start
 - End
 - Transition
- State Actions
 - Entry
 - Exit
 - Action code

Note: Models cannot be edited during animation.

DEBUGGING AND ANIMATION

Animation

This chapter provides information on using the BusinessWare services. It includes the following topics:

- Local Service Manager
- Remote Services

For more information on developing solutions using these services, see the *Application Development Guide*.

LOCAL SERVICE MANAGER

The Service Manager groups the application services of the current BusinessWare project together and enables the definitions of the services to be controlled by other (dependent) projects. Generally, collaborative applications are designed to address problems that are domain specific; however, the characteristics of collaborative applications are such that they usually depend on certain common services. In BusinessWare, these services are collectively referred to as application services. BusinessWare provides the following services:

- **Document Store Service**—provides persistent storage and versioning of documents at various stages of processing.
- **Logging Service**—provides persistent storage of audit trail data for logging transport, protocol, and application specific information.
- **Registry Service**—provides a repository for storing information about trading partners and installed solutions.
- **Security Service**—provides persistent storage of private keys used for signing, encrypting, or decrypting messages and documents.

The following examples describe some of these services.

Example: The basic goal of a business-to-business application is to help businesses electronically interact with each other. To achieve this goal, the application needs to manage information about trading partners. This information includes how to communicate with partners and what business transactions are allowed between partners. This set of information is stored in a trading partner registry.

Example: The basic goal of a claims pre-adjudication application is to reduce errors by automating and amending submitted claims. The application must execute business rules that can be general or provider specific. The relationship between the payer and the provider is similar to the trading partner relationship found in a business-to-business application. As with the business-to-business application, the specifics of payer/provider contracts managed by the claims pre-adjudication application can be encoded and stored in the trading partner registry.

Example: The basic goal of a financial straight-through processing application may be to facilitate the flow of orders from order systems to trading systems and aid in the validation and repair of incomplete or invalid orders. The rules for validation and repair may be general based on some standard specification or it may be broker or equity specific. These broker specific rules may be encoded and stored in the trading partner registry.

The applications in the examples described above execute business transactions that can be logged for analysis and visibility purposes and in some cases, must be logged for legal (non-repudiation) reasons. In these cases, you should use the logging service.

The applications described above also run on servers that may or may not be monitored continuously. As a result, when exceptions occur, administrators may have to be proactively notified. In this case, you should implement workflow as described in [Chapter 16, “Workflow.”](#)

Partner registry, logging, and notification services are very common in the applications described above. BusinessWare supplies the services described in the following sections.

[Figure 28-1](#) shows the Local Service Manager and the four application service objects in the Explorer window.

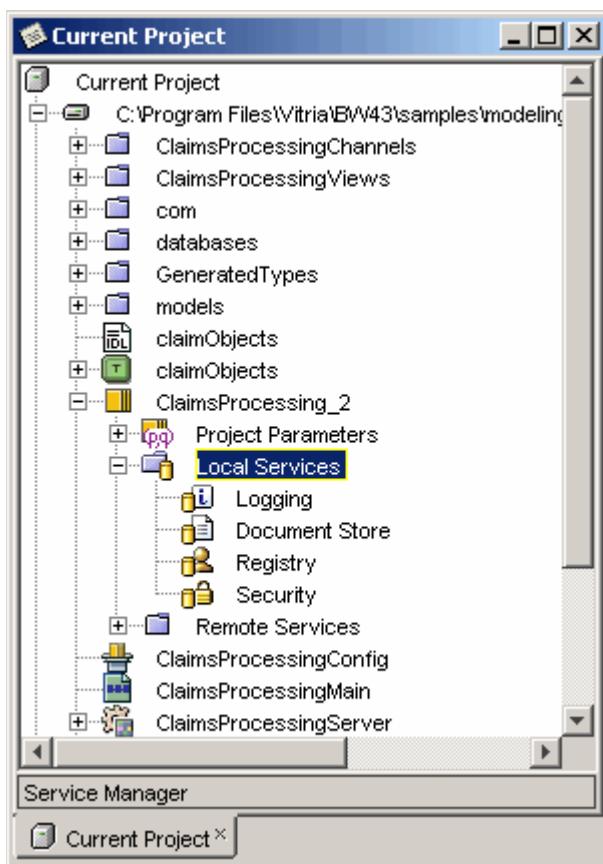


Figure 28-1 Local Service Manager

CONFIGURING THE SERVICE MANAGER

Services can be initialized in your project in two ways:

- Services can inherit Service Manager settings as configured in a dependent project.
- You can configure the services explicitly for each service.

IMPORTANT: If you configure your project to inherit service configurations, the individual service properties are ignored. To configure the individual services, the Services Project property value described below must be empty.

Inheriting Services

To inherit service manager settings and configure services exactly as configured in a dependent project:

1. Select the Services object in the Explorer.
2. In the Properties Window, set the **Services Project** property to the dependent project name and version as shown in [Figure 28-2](#).

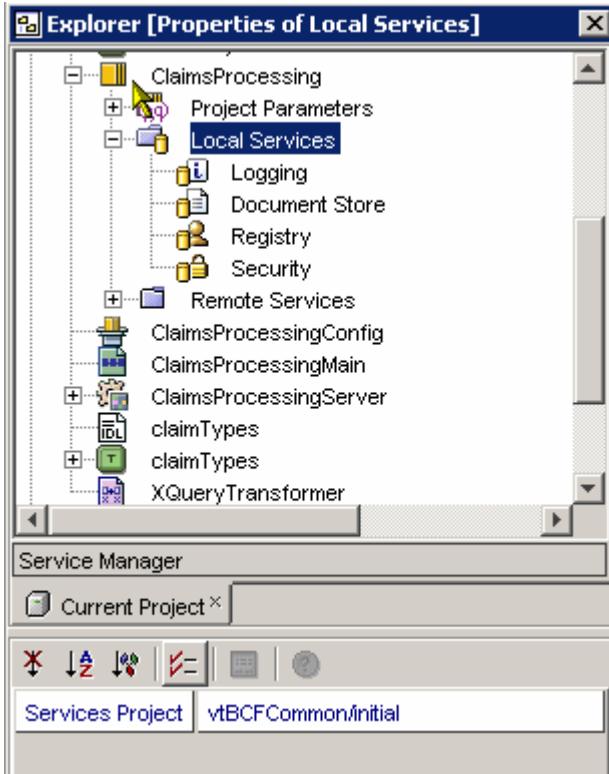


Figure 28-2 Setting the Services Project

Configuring Services

To configure services for a project, you must configure the properties for each service that you are using. Information for configuring the services is provided in the following sections.

DOCUMENT STORE SERVICE

Document store service provides persistent storage and versioning of documents at various stages of processing.

The document store service eliminates the need for multiple instances of a document, ensures document consistency, provides a central location for common metadata, and provides the ability to identify documents by reference when the actual document contents are not needed. [Figure 28-3](#) shows an example where two process models are using the Document Store Service to share documents. Process2 is able to retrieve document 1213 after the Process transaction commits.

Note: The Doc Store object in [Figure 28-3](#) is for illustrative purposes only. See below for information on configuring the Document Store Service.

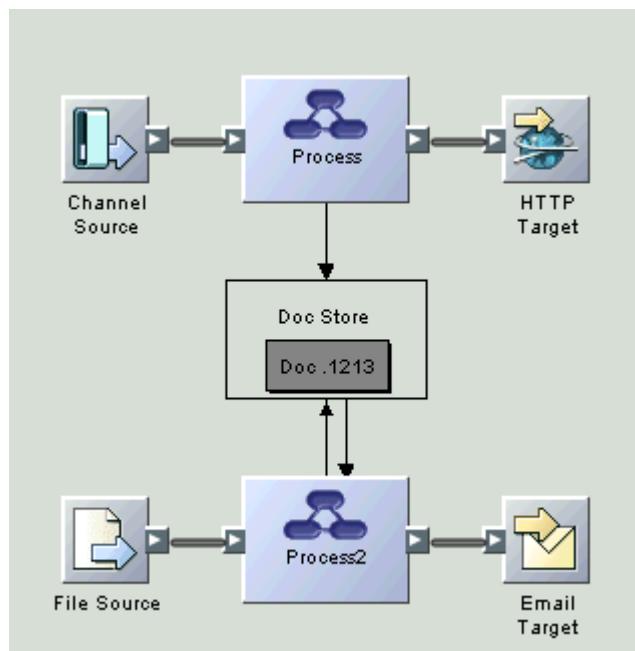


Figure 28-3 Conceptual Example of the Document Store Service

The document store service is initialized by the project. Process models and custom code can add and retrieve documents using the document store. There is a “transactionality” relationship between the process model and the document store; documents are added as part of a distributed transaction together with the process model state management; changes to the document are not available outside the transaction until the transaction commits, usually when the model gets to a restful state or terminates.

The process model holds the ID to the document in the document store. If the event in your model moves on to another service, it sends the doc ID to the that service, which then also has access to the document.

To configure the document store service:

1. In the Explorer, expand the Project and Services objects and select the Document Store object.
2. In the Properties Window, set the **Database Resource** property to the database resource that provides persistence support for document store as shown in Figure 28-4.

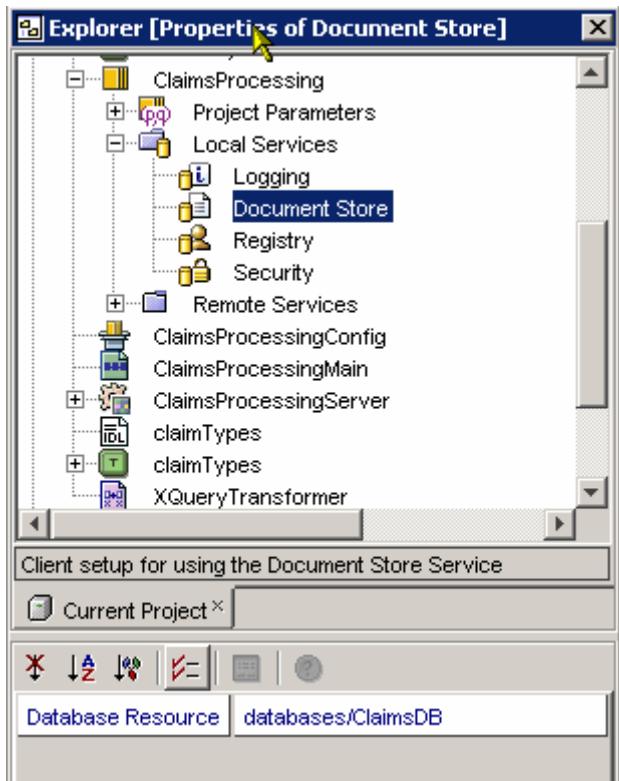


Figure 28-4 Setting the Database Resource

If the Document Store service is configured correctly, you can access the service within your model (for example, using action code or condition code) by using the following code:

```
com.vitria.af.docstore.DocStore docstore =  
com.vitria.af.docstore.DocStoreLib.getDocStore();
```

Null is returned if the service was not configured correctly.

LOGGING SERVICE

The logging service persists audit trail data to a database for logging transport, protocol, and application specific information. Information in the audit trail may refer to documents stored in the document store. It also provides correlation of documents as they are split or aggregated during the routing process.

The logging service allows you to look at any process flow with the capability of setting detailed logs and drilling into them for more details during runtime. You can specify what level of your process model is significant, and then track progress through those significant steps.

To configure the logging service:

1. In the Explorer, expand the Project and Services objects and select the **Logging** object.
2. In the Properties Window, set the **Log Map Class** property to the `com.vitria.af.aln.AuditLogger` implementation class. The default implementation provided with BusinessWare is `com.vitria.af.aln.DefaultRStoreLogger`.
3. Set the **Database Resource** property to the database resource that provides persistence support for logging data.
4. Set the **Schema Mapping** property. To configure the property, click the **browse** button in the property field and in the Schema Mapping window, create a mapping list of schema names to XML documents. The Schema Name corresponds to the log type name and the associated XML document contains the schema for the logging data.

Each Schema Map is bound to a logging type. The schema map describes the XML field to database column mapping. The binding informs the logging service that all logging requests that carry a specific logging type should be logged to the corresponding table that has the table schema described in the Schema Map.

The following column types are supported:

- int
- long
- float
- double
- string

- boolean
- BLOB
- CLOB
- Datetime

The following code is an example of a Schema Map:

```
<Map name="PurchaseOrder">
<LogDefaults type="STRING" size="200"></LogDefaults>
<Log table="PurchaseOrder" id="ID" logStatus="CAT"
      docid="DOCID">
    <Field name="Sender" type="STRING"
          size="200">/PO/Sender</Field>
    <Field name="Sender" type="STRING" <Field
          name="Receiver" type="STRING"
          size="50">/PO/Receiver</Field>
    <Field name="Sender" type="STRING" <Field
          name="ItemBoolean"
          type="Boolean">/PO/ItemBoolean</Field>
    <Field name="Sender" type="STRING" <Field
          name="ItemInt" type="Int">/PO/ItemInt</Field>
    <Field name="ItemFloat"
          type="Float">/PO/ItemFloat</Field>
    <Field name="ItemDouble"
          type="Int">/PO/ItemDouble</Field>
    <Field name="ItemClob"
          type="Clob">/PO/ItemClob</Field>
    <Field name="ItemBlob"
          type="Blob">/PO/ItemBlob</Field>
  </Log>
</Map>
```

The values of the type and size attributes of the LogDefaults elements are applied to the id, logStatus, and docid attributes of the Log element.

The id, logStatus, and docid fields are created for all logging tables. For the Field element, size and type attributes are optional; when they are not there, the type and size value specified in the LogDefaults element are applied to each Field.

The following schema can be used for all maps.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              elementFormDefault="qualified"
              attributeFormDefault="qualified" version="1_0">
```

```
<xsd:element name="Map" type="MapType" />

<xsd:complexType name="MapType">
  <xsd:sequence>
    <xsd:element name="LogDefaults" type="LogDefaultsType" />
    <xsd:element name="Log" type="LogType" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"
    use="required" />
</xsd:complexType>

<xsd:complexType name="LogDefaultsType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute name="type" type="typeValue.type"
        use="required" />
      <xsd:attribute name="size" type="xsd:string"
        use="required" />
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="LogType">
  <xsd:sequence>
    <xsd:element name="Field" type="FieldType"
      maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="table" type="xsd:string"
    use="required" />
  <xsd:attribute name="id" type="xsd:string" use="required" />
  <xsd:attribute name="logStatus" type="xsd:string"
    use="required" />
  <xsd:attribute name="docid" type="xsd:string"
    use="required" />
</xsd:complexType>

<xsd:complexType name="FieldType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute name="name" type="xsd:string"
        use="required" />
      <xsd:attribute name="type" type="typeValue.type" />
      <xsd:attribute name="size" type="xsd:string" />
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

```
<xsd:simpleType name="typeValue.type">
  <xsd:restriction base="xsd:NMTOKEN">
    <xsd:enumeration value="STRING" />
    <xsd:enumeration value="Boolean" />
    <xsd:enumeration value="Int" />
    <xsd:enumeration value="Float" />
    <xsd:enumeration value="Clob" />
    <xsd:enumeration value="Blob" />
    <xsd:enumeration value="Long" />
  </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

For development purposes, the information stored in the Logging Service and Document Store Service data repositories can be purged easily by purging the appropriate project using the service. The purging mechanism is supported in BusinessWare through the web admin UI or the vtadmin command-line tool.

Note: If you are using Crystal Reports with DB2, the table column size should not exceed 254.

If the Logging service is configured correctly, you can access the service within your model (for example, using action code or condition code) by using the following code:

```
com.vitria.af.aln.AuditLogger logger =
  com.vitria.af.aln.AuditLoggerLib.getLogger();
```

Null is returned if the service was not configured correctly.

ID CORRELATION

As a document flows through an application, it changes from one form to another. For example, when a business document is received from a partner in an HTTP message, the message can be uniquely identified using the Message-ID header. However, once the message has been processed and it is discovered to contain an EDI document, the identifying criteria now changes to the EDI ISA/GS/ST control numbers. As the document is further processed, it is determined to contain 100 individual claims. Each claim is identified by its CLM number. The ID Correlation service facilitates correlations between these identifiers and can be used later to find the original HTTP message using a claim ID. See the *BusinessWare Programming Reference* for more information on the ID Correlation service.

REGISTRY SERVICE

Registry service provides a repository for storing information about trading partners and installed solutions. Information stored in this repository is specified using the Application Administration UI (AppAdmin). Select the Partners tab in AppAdmin to complete the following tasks:

- Define new contexts—a context is the set of properties that are specific to a partner or division and a particular context or trading protocol.
- Set properties for your contexts—including properties at the following levels: partner, division, partner-level relationship, division-level relationship, and global.
- Import trading partner information
- Administer trading partner information

For more information on Application Administration and using AppAdmin, see the *Application Administration Help*.

To configure the registry service:

1. In the Explorer, expand the Project and Services objects and select the **Registry** object.
2. In the Properties Window, set the **Database Resource** property to the database resource that provides persistence support for the registry.

If the Registry service is configured correctly, you can access the service within your model (for example, using action code or condition code) by using the following code:

```
com.vitria.af.tpr.Repository repository =  
com.vitria.af.tpr.RepositoryLib.getRepository();
```

Null is returned if the service was not configured correctly.

SECURITY SERVICE

The Security Services provide support for digital signatures, digital envelopes (which includes encrypting, decrypting, and packaging data), and messages digests. The Security Services are designed to be used for both inbound and outbound messages between trading partners. The specific formatting, sequencing, and other details of the message, are specified by the trading protocols using the security services.

The Security Services consist of the following API:

- **Document-Level Security API**—provides the primary API to create and verify digital signatures, encrypt and decrypt documents, and create message digests. The document-level security API consists of a set of static methods in the `com.vitria.af.security.SecurityLib` class as well as a number of supporting classes in the security package.

When you use this API, you should provide string identifiers for a security credential and a set of Public Key Infrastructure (PKI) parameters.

- **Security Credential**—this identifier maps to either a certificate or a private key and a certificate.
- **PKI Parameters**—this identifier maps to a set of values for use with a specific operation: including a type, an algorithm name, and a set of name-value pairs representing operation-specific parameters. In addition to the default PKI parameters, there are also predefined PKI parameter sets for each of the supported trading protocol operations, which are defined and added to the database by each individual trading protocol.

For more information about security service functionality and trading protocol messages, see the *Business Collaboration Guide*.

To configure the security service:

1. In the Explorer, expand the Project and Services objects and select the **Security** object.
2. In the Properties Window, set the **Database Resource** property to the database resource that provides persistence support for security.

ACCESSING THE SECURITY SERVICE CIPHER

The persistently-stored private keys for the Security Services are protected by a security services cipher. This cipher is created at installation time and is stored in the directory server. Only BusinessWare Administrators are able to access the security services cipher in the directory server.

REMOTE SERVICES

A service project is any project that exposes a service end-point to a client project. A service end-point accepts XML documents by using `XMLEvents` or `XMLServices` interfaces. An end-point typed with `XMLEvents` receives messages asynchronously. An end-point typed with `XMLServices` receives messages synchronously. A queue is used to accept asynchronous calls.

A service project is identified when the Remote Services properties are configured in the BME. During deployment, these properties are used to set up a service registry in the directory service. The service registry identifies the service instance using a logical name.

Note: Remote Services are reserved for use by internal Vitria projects only.

The Remote Services object groups the service-related objects:

- The service project using the Service Provider Configuration
- The callback mechanism using the Service Requestor Configuration

Figure 28-5 shows the Remote Services object in the Explorer.

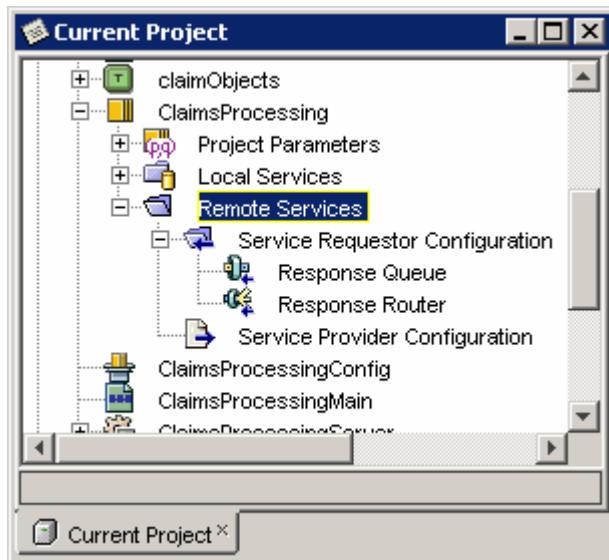


Figure 28-5 Remote Services

SERVICE REQUESTER CONFIGURATION

Use the Service Requester Configuration for service-related callbacks, typically from a service project. Callbacks from a service project are handled by a response router which routes an event to a specific target component.

The Service Requester Configuration includes the Response Queue and Response Router.

The Response Queue defines properties that configure the queue used by the Service Provider to send responses to asynchronous service requests.

Figure 28-6 shows the Response Queue properties.

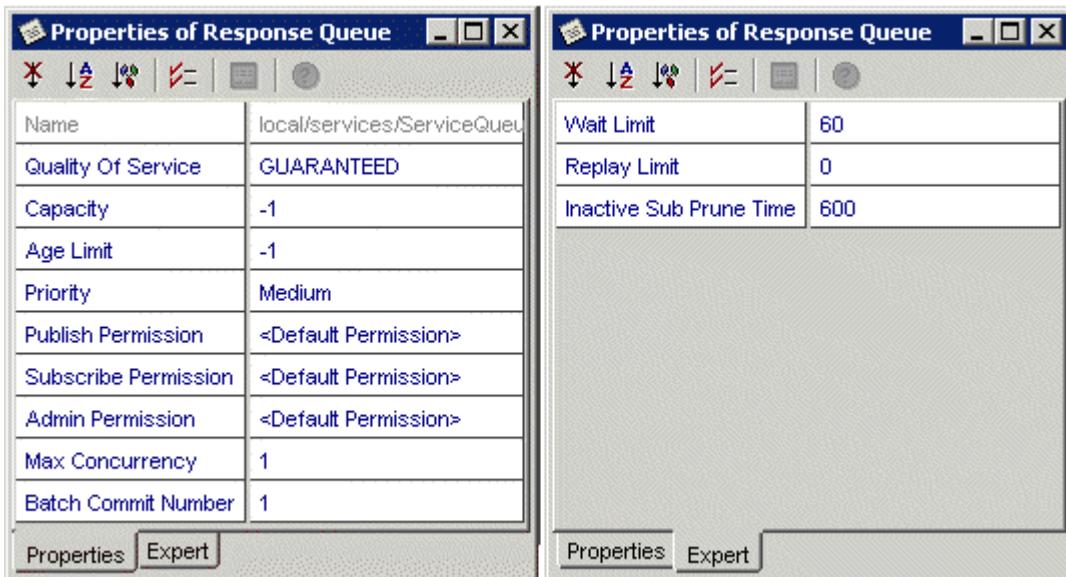


Figure 28-6 Response Queue Properties

The Response Router defines properties that instruct the Service Requester how to handle Service Response events. Figure 28-7 shows the Response Router properties.

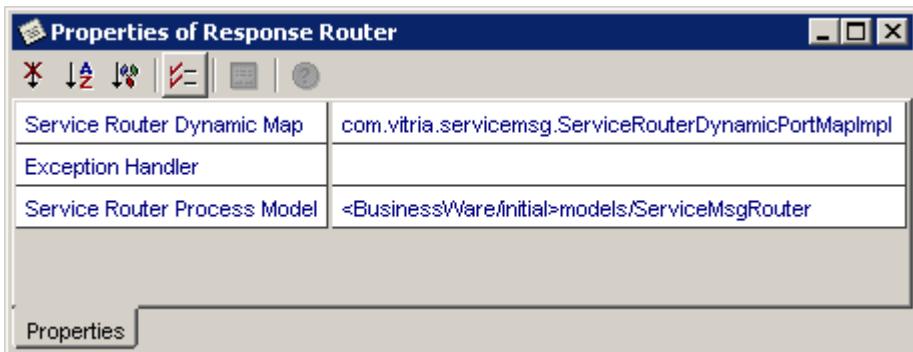


Figure 28-7 Response Router Properties

SERVICE PROVIDER CONFIGURATION

Completing the Service Provider Configuration specifies that the current project will be deployed as a service project. A service project can accept service-related synchronous and/or asynchronous requests. Service Name and Service Type

The Service Name and Service Type are combined to generate a logical service name. Typically the format is <service_type>/<service_name>. Any client to this service project uses this logical name to lookup the service and invoke the synchronous and/or asynchronous endpoint requests.

Synchronous and Asynchronous Endpoints

Define the source of synchronous requests by configuring the Service Provider Configuration Synchronous Endpoint property. Define the source of asynchronous requests by configuring the Service Provider Configuration Asynchronous Endpoint property.

An asynchronous endpoint represents a queue in the service project. The client makes asynchronous, service-related requests to this queue. The service project is responsible for attaching a source connector to the queue to retrieve these events.

APPLICATION SERVICES

Remote Services

PART VIII: APPENDICES

This part contains the following appendices:

- [Event Interoperability](#)
- [Error, Message, and Tracing Framework](#)

PART #: TITLE HERE GOES HERE

A

EVENT INTEROPERABILITY

C++ STANDALONE PUBLISHER OR SUBSCRIBER INTEROPERABILITY

Publishers and subscribers created outside the context of a BW 4.x project are considered *standalone* publishers and subscribers, and there are limitations on the types of data a BW 4.x project can exchange with standalone publishers and subscribers. Refer to the SimpleOrder Sample for examples of standalone publishers and subscribers.

As discussed in the section, “[Ports](#)” on page 6-15, the following types of data can be exchanged among 4.x model components:

- CORBA Interface Definition Language (IDL)
- J2EE Enterprise Java Beans (EJB)
- Java Remote Method Invocation (RMI)
- Java interfaces
- Web Services Description Language (WSDL)

The only type of data that a project can exchange with C++ standalone publishers and subscribers is IDL. Java data cannot be exchanged.

BUSINESSWARE VERSIONS 3.X MODELS

BW 3.x and BW 4.x models can publish events to each other using a channel in either the 3.x or the 4.x installation. When configuring a BW 4.x port to send or receive events to or from BW 3.x, you must use an IDL interface for this port, and import it into the 3.x and 4.x installations.

Beware that there is a small syntactic difference in IDL interface definitions for channel communication between BW 3.x and BW 4.x. In BW 3.x, the event keyword is required (please see BusinessWare 3.x documentation for the syntax of such definitions) whereas the keyword is optional in BW 4.x. Consequently, when you use the same IDL interface definitions in both BW 4.x and BW 3.x, you will need to keep the event keyword in the definitions.

To illustrate this difference, consider the following IDL interface fragment:

```
interface sample {
    event void placeOrder(in string orderDocument);
    event void cancelOrder(in long orderNum);
};
```

Note the use of the event keyword above. This syntax is required for BW 3.x. However, it is, optional for BW 4.x. The equivalent fragment for BW 4.x is:

```
interface sample {
    void placeOrder(in string orderDocument);
    void cancelOrder(in long orderNum);
};
```

B

ERROR, MESSAGE, AND TRACING FRAMEWORK

This appendix discusses the Error, Message and Tracing framework (EMT), and describes how to use it. This appendix contains the following topics:

- [Overview](#)
- [Resource Files](#)
- [Localization Utilities](#)
- [Using EMT APIs](#)
- [Example](#)
- [Connector Development](#)
- [Message Compatibility Requirements](#)

OVERVIEW

The Error, Message, and Tracing framework (EMT) provides a uniform mechanism for internationalizing and localizing diagnostic messages and errors, which allows applications to generate diagnostic messages and errors in a convenient and flexible manner. It incorporates exceptions into its architecture so that all trace and error information is handled uniformly. In addition, EMT handles simple locale-specific text strings, such as menu labels.

The EMT framework includes resource files, localization utilities that process the resource files, and APIs that applications use to create language-independent EMT messages. Some of the APIs are generated by the localization utilities. An EMT message is a first-class object that can be manipulated and interrogated programmatically, and can be rendered into human-readable text at any point. Both internal code and external applications can efficiently process or display these messages.

Using the EMT Framework includes creating resource files, adding to existing resource files, using the localization utilities, and incorporating the EMT APIs into source code.

RESOURCE FILES

Resource files are files that contain locale-specific text that can be translated into other languages. Resource files are necessary because internationalized source code must not contain any locale-specific text. Instead, the source code depends on resource files to hold all trace messages, user-interface labels, descriptions of exceptions, debugging output, and any other user-visible text.

Create resource files, or add entries to existing ones, as part of your normal coding process. They should hold text that would otherwise be part of your source code, such as log messages.

This section describes the formats for the following:

- A resource file
- A message within a resource file
- A message parameter

After you put text in a resource file, you can then use EMT framework APIs to access the content of these resource files in a language-neutral manner.

RESOURCE-FILE FORMAT

A resource file contains modules that contain sets of messages. The format of a Vitria resource file resembles that of a Microsoft resource file (.rc file). It has the following syntax:

```
// Comments begin with two slashes
ModuleName1 "Description of module 1"
BEGIN
    Category MessageName1 "Text of message 1"
    Category MessageName2 "Text of message 2"
    ...
END

ModuleName2 "Description of module 2"
BEGIN
    ...
END
```

The keyword JAVAONLY or the keyword CXXONLY can occur immediately before the BEGIN keyword that begins a module's messages. If one of these keywords appear, the resource compiler will assume that only Java applications or C++ applications will access messages within that module, and reduce the amount of code it generates for that module. Note that some C++ stubs must always be generated, even if the JAVAONLY keyword is used. The C++ stubs contain data needed to create resource bundles.

As the syntax description above shows, comments begin with two slashes and continue to the end of the line. Add comments that indicate who added the message and when, and most importantly, explain the meaning of the message parameters if there is any ambiguity. This information is essential when the messages are translated, to avoid forcing the translators to examine source code. The resource compiler does not enforce commenting of messages, but it is strongly recommended that you do put in comments.

For example, if a message expects an integer as the first parameter and a string as the second parameter, your associated comment might explain that the integer represents the number of events received by the component and the string represents the name of the C++ function that generated the message.

Example Resource-File Entries

The following sample shows an excerpt from a resource file:

```
// Messages from the Folder module
BFolder "Folder resources"
BEGIN
    // Added by Bob on 23 June 1999
    ERROR FileNotFoundError "Could not find file %s@0"
    // Added by Maria on 24 June 1999
    // The first parameter is the number of
    // kilobytes of memory in use.
    EXCEPTION OutOfMemory "Out of memory, using %d@0 KB."
END
```

Module and Message Identifiers

The EMT framework uses module and message names to generate numeric identifiers. It generates the identifiers by hashing the names; the resulting numeric values are position-independent, so adding messages or modules later will have no effect on existing messages. Users never refer to these numerical identifiers directly. Instead, they use symbolic constants that EMT generates.

The following examples show the symbolic module and message identifiers that EMT would generate from the resource-file excerpt in the previous section:

```
BFolderMessages::baseid           // C++ Module ID
BFolderMessages::FileNotFoundException // C++ Message ID

BFolderMessages.baseid           // Java Module ID
BFolderMessages.FileNotFound     // Java Message ID
```

MESSAGE FORMAT

The format of individual message entries in resource files follows the pattern of Microsoft ".rc" files but does not match it exactly. Each message in the resource file contains the following three parts, separated by whitespace:

- Message category
- Symbolic message name
- Message text-string

The following illustration points out each part of a message entry:

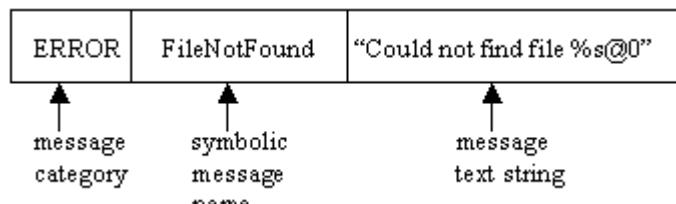


Figure 0-1 The Parts of a Message Entry

This following sections discuss each part individually.

Message Categories

The message categories are the following:

Table 0-1 Message Categories

Category	Meaning
EXCEPTION	Explanation associated with a thrown exception.
ERROR	Serious error that requires administrative attention, not associated with an exception.

Table 0-1 Message Categories (Continued)

Category	Meaning
WARNING	Warning that may eventually require administrative attention.
TRACE	Informational message; no administrative attention is needed.
STATISTIC	Statistics or performance data—used internally.
LABEL	Simple string stored in the resource file for later retrieval—Use this for user interface elements such as menu items, not for logging or exception messages.
BANNER	Banner message, such as copyright information—used internally.

Symbolic Message Names

EMT uses the symbolic name of a message when it generates the *symbolic identifiers* that users place in their code to indicate a specific message. **Never change the symbolic name of a message or module.** Translators should only modify the text within quotes.

Message Text-Strings

A message text-string is enclosed in quotes and is the only part of the resource file that is language-specific. Make sure that each of your messages is a complete sentence, written using correct grammar and spelling, with an initial capital and an ending period. You may omit the period if the sentence ends in a parameter. Make your message text-strings meaningful, because a human translator may have to translate the resource files into other languages.

The following example shows an entry in a resource file. The message text-string is in bold:

```
// An application has run out of virtual memory
ERROR OutOfMemory "Out of memory, used %ld@0 KB."
```

Tokens in a message text-string that begin with the percent character (%) are message parameters. These are described in more detail in the next section. LABEL messages cannot contain parameters.

Standard backslash escaping can be used within a message text-string. For example, the following message shows a legal message string that uses escaping:

"Quote (\\"), percent (%%), backslash(\\\")"

This message would be displayed as follows:

```
Quote (" ), percent (%), backslash(\)
```

MESSAGE PARAMETER FORMAT

Following the conventions used in Java's message localization, message text-strings have placeholders (also called parameter specifications) to mark the position at which parameters must be inserted and the type of the parameters. See the next sections for more information on the permissible *types of parameters* and their *modifiers*. The following sample message string contains parameter specifications:

```
"Component %s@0 at time %t@1 has received %ld@2 events."
```

The parameter-specification, %s@0, will be replaced by the first parameter, which should be a string. The parameter-specification, %t@1, will be replaced by the second parameter, which should be a date/time value. The parameter-specification, %ld@2, will be replaced by the third parameter, which should be a long integer. In languages where it is more natural to have verbs at the beginning of the sentence, a translator could rearrange the message something like this:

```
Received %ld@2 events at time %t@1 in component %s@0
```

Notice that although the positions of the parameter-specifications changed, the parameter-specifications themselves did not. When the text is translated into another language, the parameter-specifications can be reordered freely (as the example above shows) but the parameter-specifications themselves must never change and must never be removed.

Parameter Types

Generating the right text for some parameters (such as time codes) depends on locale. Therefore, it is important that you use the appropriate type for each parameter, and not think of each parameter merely as a string.

Here is the complete list of parameter types:

Table 0-2 List of Parameter Types

Code	Meaning
a	Embedded diagnostic message
b	Array of bytes
c	Character (default is narrow)
d	Signed decimal integer (default is 32-bit)

Table 0-2 List of Parameter Types (Continued)

Code	Meaning
f	Floating point (default is 64-bit)
m	host name (string)
n	4-byte IP address
o	Unsigned octal integer (default is 32-bit)
p	Pointer (default is 32-bit)
s	String (default is narrow)
t	time code (64-bits, Java time code convention)
u	Unsigned decimal integer (default is 32-bit)
x	Unsigned hexadecimal integer (default is 32-bit)
y	1-byte unsigned integer

Modifiers

Some types can have modifiers, which affect how they are printed. Modifiers precede the type but go after the percent (%).

Here is a list of modifiers:

Table 0-3 A List of Modifiers

Modifier	Meaning
l	(normal length) specifies a 32-bit integer (which is the default)
ll	(double length) specifies a 64-bit integer
h	(half length) specifies a 16-bit integer or a 32-bit float
w	Specifies a wide (16-bit/Unicode) character or string

In addition, field width and precision can also be specified. For example, "%10s" and "%5.2f". The meanings of these specifications are the same as the standard printf function in C. In general, any syntax that works with the printf function will work here.

LOCALIZATION UTILITIES

The message resource-compiler utility, `rescomp`, does two jobs:

- It generates a *resource bundle*, which contains message stubs and a runtime database of message text. The message text-strings are indexed by numeric message ID.
- It generates C++ header-files and Java class-files that define the symbolic constants for the messages.

This section describes the architecture of the localization utilities and their output, and the *stubs* that the resource-compiler generates.

ARCHITECTURE

The following picture describes the overall architecture:

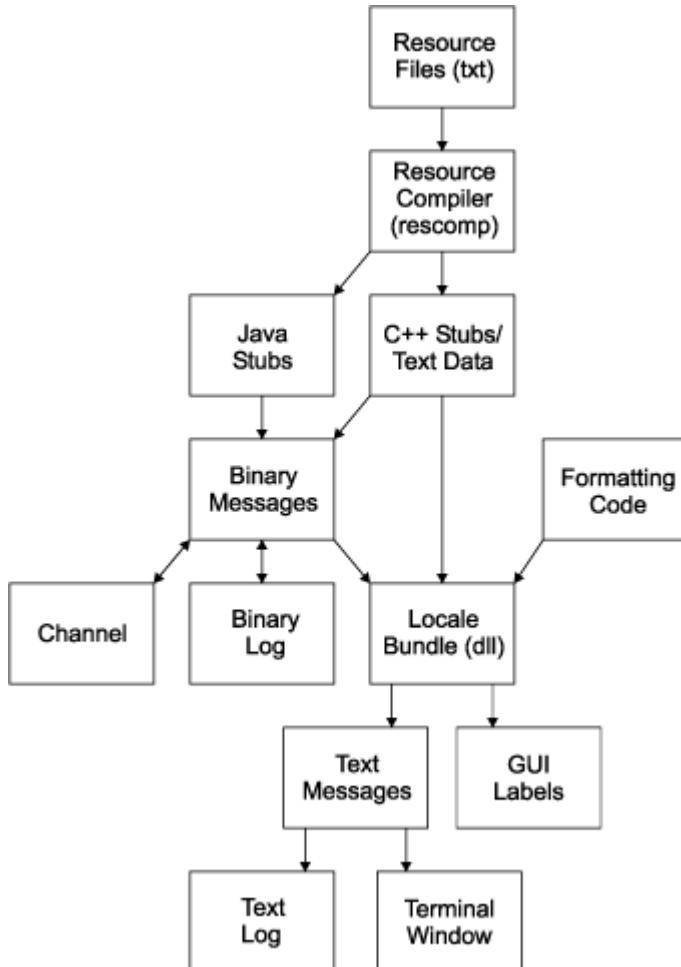


Figure 0-2 The Architecture of EMT

The resource compiler generates Java and C++ stubs from the resource files. You can use these stubs to generate diagnostic messages containing language-neutral, binary data. You can then write these binary messages to a file or send them on a channel as an event.

Eventually, an application converts a binary message into readable text. For example, user interface code may convert a message to text for labels and menu items. To accomplish the conversion, the EMT framework needs some generic formatting code as well as a resource bundle. You create this generic formatting code by compiling all the C++ stubs and linking them together into a shared library. (The C++ stubs contain the actual text in a special data structure.)

STUBS

This section discusses the Java stubs that the resource compiler generates. Generated C++ stubs follow the same general structure as the Java stubs.

In Java, a message with the category LABEL produces a constant declaration. A message with one of the other categories produces six stub methods and a constant declaration. For example, take the following TRACE message from the BusinessWareServer module:

```
TRACE ActivatorNotFound "Activator not found for %s@0.  
Service id is %s@1."
```

The BusinessWareServer module yields the Java class com.vitria.msg.BusinessWareServerMessages, and the example message produces stubs in that class with the following signatures:

```
public static final long ActivatorNotFound = 0x3156bad0048befcCL;  
public static Diagnostic createActivatorNotFound(  
    int srcModule, String a0, String a1);  
public static Diagnostic createActivatorNotFound(String a0, String a1);  
public static void logActivatorNotFound(  
    DiagLogger logger, int srcModule, String a0, String a1);  
public static void logActivatorNotFound(String a0, String a1);  
public static void logActivatorNotFound(int srcModule, String a0, String a1);  
public static void logActivatorNotFound(  
    DiagLogger logger, String a0, String a1);
```

In the example above, the constant `ActivatorNotFound` is the numeric identifier for the message. You can use the two `create` methods to create an instance of the message containing language-neutral, binary data. You can then pass the message to a logger or store it in a data structure. Eventually you must release this message with the method

`DiagnosticManager.defmgr.freeDiagnostic`. The four `log` methods allow you to create and log a message in a single operation. (These are the most commonly used methods.) In the forms where the logger is not explicitly given, the method uses the global default logger.

Some forms of the `create` and `log` methods take a `srcModule` argument. Every EMT message has two module identifiers associated with it. The first indicates the module in which the message was defined; the second indicates the part of the source code in which the message was generated. By default, both module identifiers are the same. The `srcModule` argument allows you to specify the module that generated the message. The ability to specify the source module allows different pieces of software to share the same pool of messages, without creating confusion about where the messages were generated.

External applications can use the source-module information to selectively filter messages that come from one part of the product. It is important, therefore, that the source-module information not be misleading. Sometimes it is appropriate for the source and generating module names to be the same, at other times the source-module must be specified separately to avoid confusion.

For example, calling the method

`BusinessWareServerMessages.logActivatorNotFound("x", "123")`, from within the BusinessWare Server correctly causes both module identifiers to be the constant `BusinessWareServerMessages.baseid`. On the other hand, calling `CommonMessages.logUnexpectedType` from within the BusinessWare Server, without specifying the `srcModule`, would cause the message to use `CommonMessages.baseid` as the identifier of the module that generated the message. This would be misleading. Using the method that takes the `srcModule` argument, giving it a value of `BusinessWareServerMessages.baseid`, would be more appropriate.

If the message category is `EXCEPTION`, the stubs will have an additional argument of type `Throwable`. `EXCEPTION` messages explain how and where the `Throwable` instance was generated.

USING EMT APIs

This section discusses how to incorporate the EMT APIs, primarily those generated by the `rescomp` utility, into your code. It covers messages of the following categories:

- TRACE
- EXCEPTION
- LABEL

It also covers sending EMT messages and errors to other processes.

TRACE MESSAGES

All components of BusinessWare support diagnostic tracing, which a user can “turn on” to see what the product is doing. An implementation is not complete until it has full diagnostic-tracing capability.

Tracing is critical for debugging problems. It allows an engineer to diagnose problems by looking at the server log file, instead of having to re-create the problem. Re-creation is often the most time-consuming part of debugging. Additionally, the user may not be accurate about what was happening when the problem occurred. The log file can never lie.

Trace Levels

The concept of trace levels in BusinessWare must be understood before using the EMT framework. BusinessWare supports six trace levels, represented by the following six constants from the `com.vitria.fc.diag.DiagLogger` class:

Table 0-4 Trace Levels

Constant	Significance
NONE	No tracing at all under any circumstances.
ERROR	Log serious errors and exceptions.
WARNING	Log warning and other questionable situations.
NORMAL	Log major occurrences, such as administrative changes and startup. This is not per-event logging. Under good conditions, this level should produce very little output and impose little or no performance cost.

Table 0-4 Trace Levels (Continued)

Constant	Significance
VERBOSE	Log every system occurrence. This is typically per-event or per-transition logging. This level implies some performance overhead.
TRIVIA	Log every scrap of information available. This level imposes high performance overhead and should never be used in a production system. Use it only for debugging.

Each software component or engine has an associated trace level. Application and server code can determine the global trace level by accessing the variable:

```
DefaultLogger.traceLevel_
```

Some underlying software layers have their own trace-level variables. At present, the layers with their own trace levels are:

- Transport
- ORB
- Flow
- C++ threads
- License
- logfile (.dat file)

To access the trace level of an ORB, use `com.vitria.fc.object.ORB.traceLev_`. To access the trace level of a flow, use `com.vitria.fc.flow.FlowLib.traceLev_`.

Using the APIs for TRACE Messages

Tracing code must not contain text. It should follow the following general form:

```
if (traceLevel >= constant)
    log...(arguments);
```

Where:

- `traceLevel` is the variable, such as `DefaultLogger.traceLevel_`, that indicates the current trace-level of the server or layer for which the trace message is being generated.
- `constant` is a constant, such as `DiagLogger.NORMAL`, that indicates the importance of the trace message.

- `log` is a stub, defined by the `rescomp` utility, for creating and logging an EMT message.

For example, assume the following line is in a resource file:

```
TRACE Req "Request has operation=%s@0 request id=%lu@1
key=&b@2 principal=%b@3"
```

The following example shows a call that would log that message, if indicated by ORB's current trace-level:

```
if (ORB.traceLev_ >= DiagLogger.VERBOSE)
    ORBMessages.logReq(opname, reqid, key, princ);
```

Note: No locale-specific text appears in the code. This is crucial for internationalization.

Guidelines for Adding Tracing Code

The following guidelines are useful when adding tracing code to an application:

- Trace messages should be placed at the entry and exit points of important methods. These messages should include a description of the method parameters.
- All server methods should have at least one trace message. The message should include a description of the client that made the call.
- Any important event, such as a cache hit or transaction, should be logged.
- Any `error` condition, such as a security violation or an allocation failure *must* be logged.

Choose trace levels carefully. The `NORMAL` level should not generate so much output that it causes a performance hit. At this level, you should not see more than a few messages per second overall under typical conditions. Users should use this level for production environments. The `VERBOSE` level should generate enough output that an engineer can do meaningful debugging, but the server should still be able to perform at close to normal speed. The `TRIVIA` level should generate voluminous output that allows an engineer to see absolutely everything that the server is doing. Performance may be severely degraded in this case.

EXCEPTION MESSAGES

Java uses exceptions to handle various types of errors. The EMT framework provides the EXCEPTION category of messages to enable exception data to be logged along with other tracing and error messages in a uniform fashion, and to augment Java exception handling.

Using the APIs for EXCEPTION Messages

To simply record an exception and an accompanying explanatory message in the log file, use the same general form as for a TRACE message, but make sure that the message is of category EXCEPTION. Remember that EXCEPTION messages cause the stub method to take a `Throwable` as an argument. Take, for example, the exception message:

```
EXCEPTION ImportFailed "Exception during importing
metadata."
```

The stubs it produces include one with the following signature:

```
public static void logImportFailed(Throwable t)
```

An example use of this stub, following the general form for logging TRACE messages, is:

```
...
} catch (ImportException ex) {
    if (ORB.traceLev_ >= DiagLogger.ERROR)
        ORBMessages.logImportFailed(ex);
    <attempt to recover>
}
```

When you use the stub in this way, the EMT framework will combine the message (in the example, “Exception during importing metadata.”) with information from the exception (such as its stack trace), to produce a complete and helpful description of what went wrong and where.

Another method, `DiagError.logWholeException`, also logs an exception. It is not recommended for the simple logging shown here because it logs the exception without additional explanatory text. **Always provide a message** describing what the program was doing when the exception occurred.

Using the APIs for Exception-Handling

Exception handling is more complex than tracing, and there are many different styles. This section describes two ways that you could use the EMT Framework to enhance the exception handling capabilities of Java. One style involves catching, and possibly re-throwing an EMT runtime-exception called `DiagError`. The other involves catching, and possibly rethrowing application exceptions (that is, exceptions other than `DiagError`) while using the EMT framework to improve the error-data available to the user. Deciding which style to use in a particular piece of code requires some knowledge of Java exceptions.

The `Exception` class and its subclasses are for errors that a reasonable application might want to catch. A `RuntimeException` is used less often than other `Exception` classes because a runtime exception generally indicates a coding error that the application cannot correct. For this reason, `RuntimeException` and its subclasses do not have to be included in the `throws` clause of the method signature. When a `RuntimeException` is included in a method signature, it is typically to make the developer using the method aware of the possibility of a particular exceptional condition occurring.

Because a `DiagError` is a runtime exception, convention suggests that they not be thrown for errors from which an application would reasonably be expected to recover. For this type of error, consider using the EMT framework while catching and throwing application exceptions.

Using EMT with Application Exceptions

Applications, including BusinessWare itself, generally have their own exception classes to indicate their application-specific errors. In addition, they commonly use exceptions that Java defines, such as `FileNotFoundException`. The EMT exception class, `DiagError`, does not completely replace these classes; instead, this section describes how to use them together.

While Java exceptions provide a stack trace and the name of the exception class, they do not provide a good mechanism for adding supplementary descriptions of what went wrong. (The detail string in the exception is useless because it violates the rules for internationalization. **Never use a detail string with a Java exception.**) To supplement Java exceptions, the EMT framework provides a mechanism called the *error stack* to handle supplementary descriptions. The error stack is different from the stack trace that is attached to Java exceptions. The error stack is a thread-specific data structure that holds useful information about the current thrown exception.

The EMT framework allows you to add information to the error stack, to print the topmost exception and all the information that has been added to the error stack, to clear the error stack, and to manipulate the error-stack directly.

To add a message to the error stack, call the `DiagError.addCurrent` method before throwing an exception. Use this method before throwing an exception for the first time, and after you catch an exception. For example, the following piece of code stores a diagnostic message on the error stack, then throws an exception:

```
DiagError.addCurrent(
    ORBMessages.createExceptionDuringSend());
throw new SocketException();
```

While the following piece of code appends a message to the error stack after catching an exception:

```
catch(SocketException ex) {
    DiagError.addCurrent(
        ORBMessages.createClientFailure(clientid));
    throw ex;
}
```

When you catch an exception of one type, and throw an exception of another type, add the original exception to the error stack, preferably with a message. Always provide the user with as much information as possible. For example:

```
catch(SocketException ex) {
    DiagError.addCurrent(
        ex,
        ORBMessages.createClientFailure(clientid));
    throw new AppException();
}
```

At the top of the software stack, print the top-most exception and all the information that has been added to the error stack, and clear the error stack by using either the `DiagError.logWholeException` method or a `log` method generated for a message of type `EXCEPTION`. For example:

```
catch(AppException ex) {
    ORBMessages.logImportFailed(ex);
    <attempt to recover>
}
```

If you want to discard a caught exception without printing it, you must clear the error stack to prevent confusing messages from appearing later. The `DiagError.resetCurrent` method clears the error stack. For example:

```
catch(Exception ex) {
    DiagError.resetCurrent();
}
```

Catching and Throwing `DiagError` Instances

An application can explicitly throw and catch exceptions of type `DiagError`. As noted before, this is a reasonable course of action for handling errors that an application probably cannot correct.

To throw a `DiagError`, create a message from the `EXCEPTION` category that describes the problem, wrap it in a `DiagError`, and throw it. For example:

```
throw new DiagError(
    XlateMessages.createInvalidRecord(tag));
```

To throw a `DiagError` in response to catching an exception of another type, you can pass the original exception into the `DiagError` constructor to provide additional information. You can also give an explanatory message to the constructor; this is recommended but not required. For example:

```
catch(SocketException ex) {
    throw new DiagError(
        ex,
        ORBMessages.createExceptionDuringSend());
}
```

LABEL MESSAGES

Sometimes it is necessary to store a text string in a resource file, then recover it in the code as a string. Examples include the text that appears on a button or as a menu item in the graphical user interface. Resource-file messages with the `LABEL` category provide such text. The following line shows a `LABEL` message from the `CommonUI` module of a resource file:

```
LABEL AddChannel_MENU "Add Channel"
```

This message will yield a constant declaration when the file is processed by the rescomp utility, as discussed above. Notice that the text message-string has no variables. This is true for all LABEL messages.

Using the APIs for LABEL Messages

To access the localized string associated with a LABEL message, use the formatMessage method of the com.vitria.diag.TextMessage class with the following signature:

```
static java.lang.String formatMessage(
    long messageCode
)
```

The method returns an internationalized string that can be used, for example, in a graphical user interface. The following example illustrates this:

```
new MenuItem(TextMessage.formatMessage(
    CommonUIMessages.AddChannel_MENU)) ;
```

Guidelines for Adding LABEL Messages

Use the LABEL category for legitimate purposes, such as text that must be provided to the user as part of a user interface. **Do not** abuse the LABEL category by using it to store large numbers of random strings in the resources files.

Create reasonable messages with correctly typed arguments. Remember that some human translator may have to translate the resource files into other languages.

SENDING MESSAGES AND ERRORS TO OTHER PROCESSES

Diagnostic messages and errors have a convenient representation as byte arrays.

- To turn a Diagnostic into a byte array, call the diagnostic's data method.
- To regenerate a Diagnostic from a byte array, call the DiagnosticManager.defmgr.createDiagnostic method with the array as the sole argument. Note that messages created in this way must be freed with the method DiagnosticManager.defmgr.freeDiagnostic.
- To turn a DiagError into a byte array, call the error's data method.
- To regenerate a DiagError from a byte array, call the form of the constructor that takes a byte array.

You can pass a byte array through an ORB by using the IDL type `sequence<octet>`.

EXAMPLE

This example shows how to:

- Define a message in a resource file
- Create a resource bundle and stubs
- Log an EMT message from a Java application

To use the Error, Message, and Tracing Framework:

1. Create a resource file called “`testres.txt`” that contains:

```
MyApplication "Resources for my application"
JAVAONLY BEGIN
    TRACE ProgramStart "Program started at time %t@0."
    EXCEPTION ProgramError "Error in main thread of
        application."
END
```

Note: Use of the `JAVAONLY` keyword. This tells the resource compiler that only Java applications will access these messages.

If you do not want to use a C++ compiler, skip to [step 4](#).

2. Generate C++ stubs in preparation for creating a resource bundle:

```
rescomp -h testres.txt testres.hxx
rescomp -c testres.txt testres.cxx
```

3. Compile the C++ stubs and build a resource bundle (we assume supported Windows platforms here):

```
cl /c testres.cxx
link /dll /out:testbundle_o3r.dll testres.obj vtlocale3.lib
```

The suffix of the resource bundle, for example `_o3r.dll`, must match the suffix of the other BusinessWare shared libraries, or the bundle loading code will not work.

Skip to [step 5](#).

4. If you do not want to use a C++ compiler, you can load the data in text form by calling `TextMessage.loadText (byte [] text)`. For example:

```
package com.vitria.diag

public abstract class TextMessage {

    public static void loadtext (byte []){
        XportNative.Obj ().loadText (text, text.length);
    }
}
```

Alternatively, you can load the resource file by calling `TextMessage.loadBundle (filename)`. For example:

```
package com.vitria.diag

public abstract class TextMessage {

    public static void loadbundle (textres.txt){
        XportNative.Obj ().loadBundle (textres.txt);
    }
}
```

Note: Both the resource file and the text file must be part of your project.

5. Generate Java stubs:

```
rescomp -p com.vitria.testpackage -j testres.txt
```

On Windows, this creates a file named
`\com\vitria\testpackage\MyApplicationMessages.java`

The Java package for that file is `com.vitria.testpackage`.

6. Write the application. For this example, it is a very simple one:

```
package com.vitria.testpackage;

public class testapp {

    public static void main(String[] args) {
        com.vitria.diag.TextMessage.loadBundle(
            "testbundle");
        MyApplicationMessages.logProgramStart(
            System.currentTimeMillis());
        try {
            <body>
```

```

        } catch(Throwable th) {
            MyApplicationMessages.logProgramError(th);
        }
    };
}

```

The first line loads the resource bundle. Note that `loadBundle` automatically appends the correct suffix to the bundle name. This provides platform independence. The second line logs the `ProgramStart` message using the current time as the single argument. If the body of the application throws an exception, it is caught and logged with the `ProgramError` message.

7. Compile and run the application.

CONNECTOR DEVELOPMENT

All connectors ship with their own resource bundle. By convention, resource bundles are loaded in the “Rep” class. For example:

```

static {
    String bundle = "vtSAPLocale30";
    try {
        TextMessages.loadBundle(bundle);
    } catch (Exception e) {
        ConnectorMessages.logLoadResourceDLL(bundle);
    }
}

```

Connectors do not use the global default logger or log level. Instead, each instance of a flow in a connection model has a logger and log level. Use the `getLogger` method, or the `logger_` protected instance variable, defined in the `BaseConnector` class, to get the logger. Then pass this logger as the first argument to stub-generated log methods. Use the protected instance variable `logLevel_` for the log level. For example:

```

if (logLevel_ >= DiagLogger.NORMAL) {
    ConnectorMessages.logStarting(
        logger_, Clarify2TTTargetConnectorMessages.baseid,
        name_);
}

```

The logger used by a flow in a connection model is always an instance of `com.vitria.jct.EventLogger`. This logger writes messages to an underlying logger (usually a file) and also sends the messages as events to the appropriate error model. Models can intercept these messages and send them to channels.

By convention, you must log a message whenever a connector starts or stops. Log all significant errors. Whenever there is an external API call, log the success or failure of that call (usually at the `NORMAL` level).

MESSAGE COMPATIBILITY REQUIREMENTS

An EMT message is like a documented API. All EMT message categories, except `BANNER` and `LABEL`, are subject to the compatibility and migration requirements of the software release.

Currently, there is no graceful mechanism to deprecate an EMT message.

Changing a message may or may not make the messages incompatible with its release. The following changes do not effect message compatibility:

- Changing the words in the message body. However, arguments may not be added or removed.
- Changing the order of the arguments in message text. However, argument indexes, like `@1`, may not be changed.
- Interchanging the categories `TRACE`, `WARNING`, and `ERROR`. However, other category changes break compatibility.

The following changes make the message incompatible with its release:

- Changing the module of a message.
- Changing the category of a message, other than interchanging `TRACE`, `WARNING`, and `ERROR`.
- Changing the name of a message.
- Changing the index number of an argument.
- Adding or deleting an argument.
- Removing a message.

ERROR, MESSAGE, AND TRACING FRAMEWORK

Message Compatibility Requirements

INDEX

Symbols

.info file. See information file.

.vtparams

- naming context 8-5
- proxies 6-35

_Schema

- class 11-16
- file 11-16

A

abort 26-5

aborting a transaction 25-8

access control 19-3

ACID 25-3

Action Builder

- accessing by double-clicking 14-5
- accessing from context menu 14-4
- described 12-7, 14-2
- using 14-6

action code

- action categories 14-22
- BPO actions 14-22
- ChannelConnectorLib 14-22
- connectors and custom 14-22
- logging actions 14-25
- model context actions 14-26
- port actions 14-27
- QueueConnectorLib 14-24

building 14-7

common actions 14-22

custom 14-33, 14-35

example 14-33

model variables 14-4, 14-28

- _eventBody 14-28
- _eventParams 14-28
- _fromState 14-28
- _thisModel 14-28
- _thisState 14-28
- _toState 14-28

typical sequence of actions 14-4

Action Editor

described 14-2

using 12-7, 14-10

action states

- defined 12-5

- special requirements 12-15

- using as decision nodes and decision trees 13-23

activity states

- activity reference data 16-14

- defined 12-5

- entry action 16-8

- exit action 16-8

- least workload assignments 16-23

- need approval 16-9, 16-21

- properties 16-7

- reference data 16-8

- reference data types 16-14

- setting labels 16-10

- timer 16-8

- transitions out of 16-18

add events 17-22

add parameters 17-27

adding a BusinessWare server 22-7, 22-9

admin permission 3-9

admin Web server 24-2

administrators group 22-5

aggregate functions

- avg 18-9

- count 18-9

- defined 18-9

- sum 18-9

aggregation query

- description 17-7

- restrictions 18-10

- syntax 18-9

aliases 18-3

animating projects 3-26, 27-34

annotation, adding to models 6-8, 10-13

application exceptions 26-2

application server

- asynchronous integration 8-7

- interoperability 8-1

- J2EE 8-4

INDEX

making requests to 8-3
receiving requests from 8-3
synchronous integration 8-3
application service
processes 24-2
arrays
injecting 27-23
asynchronous communication
destination considerations 7-7, 7-8
via channels 7-3
asynchronous integration 8-7
attribute properties
expert
date/time format 19-14
defined 19-13
model state filters 19-16
resource prefix 19-15
XSLT renderer 19-17
general
color 19-13
defined 19-11
label 19-13
name 19-12
type 19-13
attributes part of source schema 18-3
Auto Synchronize Ports property 12-2
auto-deploy
animation 27-35
debugging 27-6
overview 22-6
auto-partition 22-6, 22-7, 22-10

B

basic channels 7-10
batch commit limits 25-14
best practices 6-28
binary file logger properties 20-13
Binding Path Name 6-34
BLOB 11-11
BME
described 2-1
getting help 2-1
key windows 2-2
shortcut menus 2-3
starting the BME 2-1
boolean 5-5
bpid parameter 13-24
BPO
attribute initialization 11-14

attributes 18-3
complex definition 11-8
definition 11-3, 11-8
example 18-2
interface type 11-19
lifecycle 11-2
BPO inspector
described 27-26
enabled 27-26
BPOs 5-3
bpState 18-3
breakpoints
described 27-11
removing 27-15
setting in code 27-12
setting in models 27-11
bserv 24-2
Build 3-24
Build All 3-24
building expressions 17-13
bundled Web server 24-9
business objects
BPO 11-7
custom methods 11-10
dataobject attributes 11-10
definition 11-7
description 11-7
DO 11-7
local data 11-7
sequence attributes 11-11
BusinessWare administrators group 22-5
BusinessWare project module 1-11
BusinessWare projects 3-10
BusinessWare server 24-2
adding 22-7, 22-9
specifying the default 22-9
BusinessWare solution, lifecycle of 1-14
BusinessWare Transformer 2-12, 12-6
BW3x channel shortcut 7-10
bwnamespace.xml 22-3
bwschema.xml 22-3

C

chained action state, default event 12-19
change events 17-22
channel
about 7-8, 7-9
basic channels 7-10
BW3x channel shortcut 7-10

composite channels 7-11
 properties 7-13
 replica 7-12
 replica channels 7-12
 resources 7-10, 7-12, 7-13
 types 7-10
 undeploying 22-20
 channel connectors 7-15
 adding to an integration model 7-16
 characteristics 7-15
 channel logger properties 20-13
 Channel Source Connector 6-13, 7-15
 Channel Target Connector 6-13, 7-15, 25-13
 Channel Target/Source Connector 6-13
 Channel TargetSource Connector 7-15
 channel/queue inspector
 auto pause 27-30
 described 27-27
 display events 27-30
 requirements 27-27
 using 27-29
 char 5-5
 character large object 17-26
 checklist for creating BusinessWare solutions 1-15
 Claims Processing Sample 19-2
 CLASSPATH
 Java classes 8-4
 clause
 from 18-4
 group by 18-6
 having 18-6
 select 18-4
 where 18-5
 CLOB 11-11, 19-20
 data type 11-11, 17-26, 19-20
 limitations 17-26
 Clone View 2-10
 code
 automatically generated 14-2
 builders 14-3
 editors 14-3
 in process models 14-1
 source 14-1
 tools for writing 14-2
 using 14-3
 code builders
 accessing 14-4
 relationship between builders and editors 14-3
 code construction 14-4
 Action Builder 14-6
 action categories 14-22
 Action Editor 14-10
 common actions 14-22
 Condition Builder 14-13
 example of action code 14-33
 model variables 14-28
 relationship between builders and editors 14-3
 relationship of models and source code 14-1
 Source Code Editor 14-19
 tools 14-2
 typical sequence of actions 14-4
 Code Helper 14-8
 collection 18-1
 collision policy 3-15
`com.vitria.modeling.runtime.PortExceptionHandlerImpl` 26-15
 commit retry count 3-9
 commit retry interval 3-9
 common actions for handling exceptions
 log and notify 26-6
 restart 26-6
 skip 26-5
 stop 26-6
 throw exception 26-6
 communication
 asynchronous 7-2
 in models 1-7
 near real time 7-7
 communication styles
 overview 1-7
 communicator 24-2
 Compilation Properties 3-7
 Compile 3-24
 Compile All 3-24
 component
 adding 6-5
 process views 19-7
 components 6-9
 and ports 24-6
 described 6-9
 exceptions 26-2
 integration components 6-9
 introduction to 6-4
 linking 6-7
 linking to models 6-11
 properties 6-10
 wiring 6-7
 composite channels 7-11
 composite types
 sequence 5-5
 struct 5-5
 union 5-5

INDEX

concurrency 16-34
concurrent processing in models 13-22
Condition Builder
 accessing by double-clicking 14-5
 accessing from context menu 14-4
 described 14-2
 using 14-13
Condition Editor 14-3
Conditional Fork 12-6
configuring
 database persistence 20-8
 Integration Servers 20-2
 Web service input proxy 9-12
 Web Service Output Proxy 9-18
 Web Service State 9-28
connected systems exceptions 26-4
connection manager properties 20-6
Connectors 14-22
connectors
 adding 6-6
 channel
 adding 7-16
 defined 7-15
 properties 7-16
 Channel Source Connector 6-13
 Channel Target Connector 6-13
 Channel Target/Source Connector 6-13
 connecting with external systems 1-7
 described 6-12
 Email Source Connector 6-13
 Email Target Connector 6-13
 File Source Connector 6-12
 File Target Connector 6-12
 FTP Source Connector 6-12
 guidelines for configuring 6-14
 HTTP Source Connector 6-13
 HTTP Target Connector 6-13
 introduction to 6-4
 Line Source Connector 6-12
 linking to resources 6-14
 queue
 properties 7-22
 Queue Source Connector 6-13
 Queue Target Connector 6-13
 Queue TargetSource Connector 6-13
 queues 7-20
 source connectors 1-7, 6-12
 target connectors 1-7, 6-12
 transactionality 25-11
 types of 6-12
containers 24-4
context actions 14-26
copying process views 19-4
CORBA Interface Definition Language (IDL) A-1
Creating 16-36
creating
 deployment configurations 22-7
 integration models 6-4
 Integration Servers 20-1
 nested integration models 6-39
 process models 10-9
 process views 19-3
 projects 3-10
 solutions, a checklist 1-15
creating process query models 17-4
creating project parameters 23-6
credentials
 guest 24-10
 used 24-10
ctx object 11-13
custom
 in Expression Builder 17-13
 installation 23-1, 23-12
custom actions 14-22
custom method definitions
 BPO 11-10
 DO 11-10
 local data 11-10
customizer 17-19
CXXONLY B-3

D

data object 10-8, 11-4
data objects 5-4
data values
 large 11-11
database connection pool 16-34
database persistence 20-8
database properties 20-8
dataobject attribute 11-10
debugging
 breakpoints, removing 27-15
 breakpoints, setting in code 27-12
 breakpoints, setting in models 27-11
 debugger window 27-9
 description 27-1
 introduction to 1-13
 procedure 27-2
 projects 3-26
 tools 27-2

watches 27-16
 decision nodes and trees 13-23
 default permission 3-6
 default transitions 12-17
 defaultEvent 12-17, 12-19, 26-8
 definition 11-5
 delayable abort 26-5
 delayable skip 26-6
 delete events 17-22
 deleting process views 19-4
 dependent project 10-16
 deploying
 auto-deploy 22-6
 auto-partition 22-6
 manual 22-6
 overview 22-8
 process query components 22-17
 queries 22-17
 deploying projects 1-12, 3-25, 22-6
 deployment
 actions 22-7
 auto-deploy 22-6
 auto-partition 22-6
 configuration 22-7
 definition 1-12
 directory server 22-3
 manual 22-6
 options 22-6
 overview 22-1
 process 22-5
 process query 22-17
 process query models 17-16
 refresh 22-8
 stages 1-12
 deploy 22-15
 predeployment 22-15
 undeploy 22-15, 22-16
 using the BME 22-12
 using vtadmin 22-16
 validate 22-8
 description
 process views 19-9
 design repository 3-6, 10-16
 designing with one-phase resources 25-14
 directory server
 deployment 22-3
 namespace 22-3
 overview 24-2
 schema 22-3
 display options for
 integration models 6-7
 process models 10-12
 display properties
 process view 19-10
 style sheets 19-22
 distributing projects 23-11
 DO
 attribute initialization 11-14
 complex definition 11-8
 definition 11-4, 11-8
 description 11-4
 interface type 11-19
 Dock View Into 2-11
 document store service 28-5
 double 5-5
 drilldown queries 17-27
 drilldown view 19-7

E

ECA specifications 10-3, 12-11, 14-2
 edit parameters 17-27
 Editor 2-9
 EJB 5-6, 6-15, 6-21, A-1
 EJBs
 making requests on 8-5
 synchronous communication 8-3
 Email Source Connector 6-13
 Email Target Connector 6-13
 EMT
 APIs B-12
 overview B-1
 enable editing text checkbox 17-14
 Enable Label Selection option 6-8, 10-13
 enabling and disabling breakpoints 27-15
 entry actions 12-7, 14-2
 enum 5-5
 Error, Message, and Tracing framework
 See EMT
 event
 add 17-22
 change 17-22
 definition 10-4
 delete 17-22
 interoperability A-1
 output 17-22
 event injector
 described 27-19
 event inspector
 described 27-25
 enabled 27-25

INDEX

event interface 5-3
exception
 messages B-15
exception handler class 3-8
exception handlers 26-10
 default project 26-11
exception map
 creating 26-16
 exceptions 26-18
 overview 1-6
 properties 26-17
exception maps 26-16
 action 26-22
 condition 26-21
 context 26-21
 creating 26-16
 exceptions 26-18
 inband exceptions 26-18
 match 26-21
 outband exceptions 26-19
 post process 26-22
 properties 26-17
 transaction 26-22
 unwrap 26-21
 using 26-23
Exception Transitions 26-9
Exception transitions 10-4, 10-11, 12-11, 12-18, 12-20
exception transitions 12-5, 12-15, 12-17, 26-7
exceptionhandlerimpl class 26-11
exceptions
 application 26-2
 component 26-3
 defining exception interface 10-4
 external applications 26-4
 overview 26-1
 sources 26-2
 system 26-2
 via proxies 26-4
Exclude Files 3-7
exit actions 12-7, 14-2
Explorer 2-5
exporting
 projects 3-13
 types 5-10
 Web services 9-25
 from a Web service input proxy 9-27
 from an input port 9-27
 from the project 9-26
expression builder
 match all expressions 17-14
 match at least one expression 17-14

overview 17-13

F

File Source Connector 6-12
File Target Connector 6-12
float 5-5
folders
 process views 19-3
forks
 defined 12-6
 in process models 13-5
FTP Source Connector 6-12

G

General Properties 3-6
generating
 information file 23-13
 XML output 23-13
generating types 17-15
getInputPort(String componentName, String inputPortName) 26-26
getInputPort(String componentName, String inputPortName, Class portInterface) 26-26
getInputPortRequestListener(String componentName, String inputPortName) 26-26
getOutputPort(String outputPortName) 26-25
getOutputPort(String outputPortName, Class portInterface) 26-26
getOutputPortName() 26-15
GIOP 8-7
global data
 access in code 14-31
 in projects 14-29
 storage 14-29
global data storage 14-29
global options 10-13
group by
 described 18-10
grouping parameters 23-5

H

handleComponentException() 26-11
handleSourceConnectorException() 26-12
handleTargetConnectorException() 26-12

- hashtable 9-24
HTTP
 transport headers 9-24
 HTTP compression 9-13
 HTTP Source Connector 6-13
 HTTP Target Connector 6-13
 HTTPS 9-22
- I**
- id correlation 28-10
 IDL 5-6, 6-21, 11-7
 ignore 3-15
 ignore source connector errors 3-9
 IIOP 1-7
 importing
 projects 3-13
 inband exceptions 26-18
 information file
 contents 23-13
 generating 23-13
 inspecting and editing types 5-9
 installing a custom project 23-13
 integration
 asynchronous 8-7
 between BusinessWare and external application servers 8-3
 between BusinessWare and external servers 8-1
 synchronous 8-3
 integration component
 properties 6-10
 integration components
 described 6-9
 ports on component and model 6-38
 integration model
 process views 19-6
 root 6-1
 integration models
 annotating 6-8
 components
 types of 6-9
 connectors 6-12
 contents 6-3
 creating 6-4, 6-5
 customizing the display 6-7
 description 6-1
 designating the root model 6-37
 display options 6-7
 editing labels 6-8
 example 6-2
- integration components 6-9
 introduction to 1-3, 6-1
 layers 6-8
 linking components to models 6-11
 linking connectors to resources 6-14
 nested integration models 6-37
 port configurations 6-38
 port specifications 6-5
 printing 6-41, 10-18
 process components 6-9
 process query components 6-9
 reusing 6-40
 reusing in other projects 6-40
 saving as templates 6-40
 setting global options 6-8
- Integration Server**
 and deployment 20-1
 child objects
 binary file logger 20-13
 channel logger 20-13
 connection manager 20-6
 loggers 20-10
 overview 20-2
 RDBMS Persister 20-8
 text file logger 20-12
 transaction manager 20-7
 Web server 20-13
 configuring 20-2
 creating 20-1
 overview 24-2
 properties 20-3
 trace levels 20-4
- Integration Servers**
 undeploying 22-21
- interface**
 definitions 6-18
 IDL A-1
 matching 6-27
- interface definition**
 creating 6-23
- interfaces**
 Web services 9-9
 Service Endpoint 9-9
- Internet Inter-ORB Protocol** 8-5
- interoperability**
 event A-1
- interoperability with application servers** 8-1
- J**
- J2EE**

INDEX

application server
 receiving requests from 8-4
application servers
 compliant 8-1
Enterprise Java Beans (EJB) A-1
Java
 interfaces A-1
Java compilation settings 3-7
Java compiler
 debug setting 3-7
 deprecation setting 3-7
 encoding setting 3-7
 optimize setting 3-7
Java implementation
 BPO 11-15
 business object 11-15
Java methods 14-4
Java Remote Method Invocation (RMI) A-1
Java Source Code Editor
 accessing 14-4
 description 14-3
 usage 14-19
Java transaction application 25-10
Java transaction service 25-10
JAVAONLY B-3
JAX-RPC 9-1
JAXRPCException 9-24
JNDI 8-4
joins in process models 12-6, 13-7
JTA 25-10
JTS 25-10

L

labels
 editing in integration models 6-8
 editing in process models 10-13
 messages B-18
large data values 11-11
layers in
 integration models 6-8
 process models 10-12
Line Source Connector 6-12
linking
 components to models 6-11
 process component and process model 10-9
load balancing 21-1
 benefits 21-1
 clusterable components 21-2
 configuring projects for 21-10

master node 21-1
local data 5-4, 11-5
 actions 14-24
 complex definition 11-8
 definition 11-8
 description 11-5
 local data class 11-18
local data class 11-18
localization utilities B-8
log and notify 26-6
log map class 28-7
log viewer 27-30
logger 20-10
 binary file 20-13
 channel 20-13
 configuring 20-10, 27-33
 properties 20-12
 text file 20-12
logging
 actions 14-25
logging service 28-7
logging, runtime 24-11
logs, viewing 27-30
long 5-5
long double 5-5
long long 5-5

M

mail host 3-9
Main window 2-3
major types
 BPOs 5-3
 data objects 5-4
 event interface 5-3
 interface 5-3
 module 5-3
manager 16-22
manifest parameters 23-4
manual deployment 22-6
manual partition 22-10
mapping options 23-7
mapping parameters 23-7
master node 21-1
match all expressions
 expression builder 17-14
 query builder 17-12
match at least one expression
 expression builder 17-14
 query builder 17-12

Max javac Memory 3-7

messages

- categories B-4
- exception B-15
- format B-4
- identifiers B-3
- labels B-18
- parameter format B-6
- stubs B-10
- text strings B-5
- trace B-12

methods

- custom 11-10
- RequestMaps 13-26
- request map 13-26

model

- router 15-4
- saving 6-7

model context actions 14-26

model default action 12-16, 14-2

model variables

- defined 14-4
- overview 14-28

model, invoking 24-7

modeling overview 1-1

module 5-3

mountpoints 3-7

multiple-server 25-5

N

namespace 22-3, 22-6

nested integration models

- advantages 6-37
- creating 6-39
- integration components 6-9
- matching ports 6-38

nested process models

- creating 6-38, 13-14
- described 13-9
- example 13-11
- how events are processed 13-10
- ports 13-14
- terminator states 13-13
- terminology 13-10

nested states 12-5

numTransitions 18-3

O

object table 10-14, 11-18, 11-19

object transaction service 25-10, 25-17

objects, defining 10-8

octet 5-5

oid 11-2, 11-4, 18-3

one-phase commit 25-9

opening projects 3-11

operator

- at 18-6

- in 18-7

options for

- integration models 6-8

OTS 25-10, 25-17

outband exceptions 26-19

Output window 2-11

overwrite 3-15

Owner 3-6

P

packaging projects 23-1

Palette 2-3

parameter group options 23-5

parameterized project 23-1

parameters 17-7

- groups, creating 23-5

- manifest 23-4

- mapping 23-7

- predefined 23-4

- project 23-1

- Project Parameters node 23-3

- properties 23-4

- setting values 23-9

- settings 23-3

- simple 23-4

- simple, creating 23-6

- substitution 23-4

- creating 23-10

- specifying values 23-10

- system 23-4

- tasks 23-4

- user-defined 23-4

partition

- auto 22-10

- manual 22-10

partitionable components 22-8

partitioning 1-12, 22-2, 22-8

- auto 22-7

INDEX

- persistence 20-8
 - cache 20-8
 - DO 11-4, 11-13
 - local data 11-5
 - performance considerations for 11-12
 - RDBMS 20-8
 - runtime 24-8
- PKI parameters 28-12
- point-to-point communication 7-7
 - queues 7-7
- port configurations
 - integration models 6-38
- port invocation APIs 26-25
- port synchronization 10-7, 12-2
- ports
 - adding 6-6, 6-26
 - EJB requests 8-4
 - function 12-1
 - guidelines for configuring 12-2
 - importing interface definitions 5-5, 6-21
 - input 6-15, 8-4
 - input ports 12-1
 - interface for 6-21
 - introduction to 6-4
 - matching 6-28
 - modeling constraints 6-26
 - nested integration models, ports on 6-38
 - nested process models, ports on 13-14
 - output 6-15, 12-1
 - port synchronization 10-7, 12-2
 - port type 12-1
 - properties 6-16
 - query 6-16, 12-2
 - receiving requests from EJBs 8-4
 - removing 6-16, 6-26
 - synchronous invocation 8-1
- predefined parameters 23-4
- primitive types
 - boolean 5-5
 - char 5-5
 - double 5-5
 - enum 5-5
 - float 5-5
 - long 5-5
 - long double 5-5
 - long long 5-5
 - octet 5-5
 - short 5-5
 - string 5-5
 - unsigned long long 5-5
 - unsigned short 5-5
 - wchar 5-5
- wstring 5-5
- printing 10-18
 - fit to page 10-18
 - properties 10-18
- process components
 - described 6-9
 - linking to model 10-9
 - properties 6-10
 - relationship to process models 10-6
- Process Model Link property 10-7
- process model templates 15-1-15-7
 - creating templates 10-16, 15-7
 - incorporating in project 15-4
 - RouteByPortName model 15-3
 - RouteByPortType model 15-3
 - router 15-1-15-4
 - SimpleTransformer 15-4
- process models
 - analyzing business processes 10-18
 - annotating 10-13
 - concurrent processing in 13-22
 - creating 10-9
 - customizing the display 10-12
 - editing labels 10-13
 - example 10-2
 - forks 13-5
 - introduction to 1-4, 10-1
 - joins 13-7
 - layers in display 10-12
 - linking to component 10-9
 - model default action 12-16
 - nested models
 - creating 13-14
 - described 13-9
 - example 13-11
 - how events are processed 13-10
 - ports 13-14
 - terminator states 13-13
 - terminology 13-10
 - objects, defining 10-8
 - prebuilt models 15-1-15-7
 - printing 10-18
 - properties 10-14
 - relationship to process components 10-6
 - reusing in other projects 10-16, 15-7
 - setting global options 10-13
 - source code, and 14-1
 - statechart diagrams 10-2
 - stateful models 10-5
 - stateless models 10-5
 - templates 15-1-15-7
 - timers on states 13-1

workflow models 10-5
 process query component
 deploying 22-17
 described 6-9
 input port 17-3
 output port 17-4
 ports 17-3
 properties 6-10, 17-6
 process query model link 19-5
 process query models
 creating 17-4
 deploying 17-16
 introduction to 1-5
 properties 17-6
 runtime behavior 17-20
 process views
 access control 19-3
 attribute properties
 expert 19-13
 general 19-11, 19-12
 CLOB 19-20
 copying 19-4
 creating 19-2, 19-3
 deleting 19-4
 display properties 19-10
 folders 19-3
 organizing 19-3
 properties 19-5
 component 19-7
 description 19-9
 drilldown
 view 19-7
 integration model 19-6
 process query model link 19-5
 query 19-6
 resource file 19-8
 renaming 19-4
 project
 packaging 23-1
 parameterization 23-1
 project dependencies
 steps to add 3-16
 project directory 3-6
 project exception handler 26-11
 project global data 14-29
 project icon 3-6
 project information file. See information file
 project init class 3-8
 project init data 3-8
 project modules 1-11
 usage 3-16
 project parameter options 23-6
 Project Parameters node 23-3
 project parameters. See parameters
 Project properties 3-4
 project startup 24-7
 ProjectInit 14-29
 class 14-29
 data 14-29
 implementation 14-30
 sample implementation 14-32
 projects
 animating 3-26, 27-34
 BusinessWare 3-10
 creating 3-10
 custom installation 23-1, 23-12
 debugging 3-26
 deploying 3-25, 22-1, 22-6
 distributing 23-11
 exporting 3-13
 importing 3-13
 introduction 3-1
 introduction to 1-10, 3-1
 mountpoints 3-3
 objects 3-2
 opening 3-11
 packaging 23-1
 partitioning 22-8
 project modules 1-11
 sample projects 1-19
 saving 3-11
 settings 3-4
 sharing 3-11
 synchronization 3-13
 undeploying 22-19
 versioning 3-23
 working with 3-10
 prompt 3-15
 properties
 Binding Path Name 8-4
 channel 7-13, 7-16
 dynamic port 6-20
 process query component 17-6
 process query model 17-6
 process views 19-5
 query 17-7
 queue connectors 7-22
 transaction control 6-20
 type propagation 6-20
 Properties window 2-6
 protocols
 transaction 25-5

INDEX

proxies 6-14
 adding 6-31
 communicating externally 6-29
EJB Output Proxy 6-29
input proxies 6-29
introduction to 6-4
output proxies 6-29
properties 6-33
server integration 8-3
Simple Input 6-29
Simple Input Proxy 6-29
Simple Output 6-30
Simple Output Proxy 6-29
types 6-29
Web Service Input Proxy 6-29
Web service input proxy 9-3, 9-8
Web Service Output Proxy 6-29, 9-3, 9-13

proxy
 adding 6-6
 adding to integration model 6-32
 RMI Input 8-4
Public Key Infrastructure
 See PKI parameters
publishing non-transactionally 25-14
publishing port actions 14-27
publish-subscribe paradigm
 described 7-3

Q

quality of service
 guaranteed 7-6
 reliable 7-6
queries
 aggregate functions 18-9
 aliases 18-4
 deploying 22-17
 example 18-2
 filter 18-8
 grammar 18-15
 introduction 18-1
 process views 19-6
 select clause 18-4
 time-based aggregate query 18-12
 time-based filter query 18-11

query
 aggregation 17-7
 builder 17-7, 17-9
 editor 17-7, 17-14
 properties 17-6
 simple filter 17-7

SQL 17-17
time-based 17-8
types 17-7
query builder
 described 17-7
 group 17-12
 match all expressions 17-12
 match at least one expression 17-12
 selected items 17-10
 using 17-9
 validation 17-12
query editor
 described 17-7
 enable editing text checkbox 17-14
 sequence query 17-26
 using 17-14
query ports 12-2
queue connectors
 adding 7-21
 characteristics of 7-21
Queue Source Connector 6-13, 7-21
Queue Target Connector 6-13
Queue TargetSource Connector 6-13, 7-21
queues
 about 7-18
 point-to-point communication 7-7
 resources 7-19
 setting resource properties 7-19
 undeploying 22-20
QueueTarget Connector 7-20

R

RDBMS server 24-2
real-time 19-2
reference data
 BPO reference data 16-12
 filtering 16-13
refresh 22-8
registeration
 code builders
 custom actions 14-22
registry service 28-11
relationship 16-4
renaming
 process views 19-4
renaming projects 3-11
replica channels 7-12
request descriptor 13-26
 definition 13-26

request map 13-25
 request map class 3-8
RequestListener
`getOutputRequestListener(String outputPortName)` 26-26
RequestMap
 associating with a port 13-28
 methods 13-26
RequestMap Class 13-24
 custom 13-28
RequestMapImpl 13-25
 resource and transaction managers 25-5
 resource file 19-8
 format B-2
 sample B-3
 resource files
 overview B-2
 resource manager 25-5
 resource properties
 setting for queues 7-19
 resources
 channel 7-12, 7-13
 channels 7-10
 creating 7-12
 definition 1-9, 6-14
 file 19-8
 introduction do 1-9
 linking a connector to 6-15
 linking connectors to 6-14
 overview 1-9
 queues 7-19
 restart 26-6
 RMI 1-3, 1-7, 3-20, 5-5, 6-4, 6-15, 6-18, 6-21, 8-4, A-1
 roles 16-3, 16-22
 root entry 22-3
 Root Integration Model 3-6
 root integration model
 defined 1-3
 designating 6-37
 router
 incorporating in project 15-4
 RouteByPortName 15-4
 RouteByPortName model 15-3
 RouteByPortType 15-4
 RouteByPortType model 15-3
 templates for router models 15-1-15-4
 router model 15-4
 running
 real-time process query model 17-21
 snapshot process query model 17-21
 runtime
 administration processes 24-2
 application service processes 24-2
 architecture 24-1
 connection management 24-9
 logging 24-11
 messaging process 24-2
 performance 24-3
 persistence 24-8
 processes and hosts 24-1
 security 24-10
 services 24-8
 transaction 24-8
 Web server and servlet engine 24-9
Runtime Properties 3-8

S

sample files
 included with BusinessWare 1-19
 Web services 9-28
schema 22-3, 22-6
 directory server 22-3
schema map 28-8
schema mapping 28-7
scriptable BME 22-7
SDT-CommonTransformationLib 14-22
security
 considerations 8-7
security credential 28-12
security service 28-11
security, runtime 24-10
select clause 18-4
self-transitions 12-13
sequence 5-5
sequence attributes 11-11, 17-26
sequence table 11-12
sequences
 injecting 27-23
 of actions 14-4
 understanding behavior 11-14
server
 admin Web 24-2
 BusinessWare 24-2
 directory 24-2
 Integration Server 24-2
 RDBMS 24-2
 Web server 24-2
Service Endpoint interface 9-9
service manager 28-1
service URI 9-18

INDEX

- services
 - configurations 28-3
 - explicit 28-4
 - inherited 28-4
 - inheriting 28-4
 - logging 28-7
 - registry 28-11
 - security 28-11
- services project 28-4
- servlet engine 24-9
- setting parameter values 23-9
- setting process query properties 17-6
- setting trace levels 20-4
- settings
 - Java compilation 3-7
- sharing projects 3-11
- sharing types 5-10
- short 5-5
- simple filter query 17-7
 - described 18-8
 - syntax 18-8
- Simple Object Access Protocol (SOAP) 9-1
- simple parameters 23-4
- simple states 12-4
- SimpleOrderSample 8-6
 - RMI Simple Input Proxy 8-5
- single-server 25-4
- skip 26-5
- slave nodes 21-1
- snapshot 19-1
- SOAP 9-1
- Source 1.4 3-7
- source code 14-1
- Source Code Editor 2-10
 - accessing by double-clicking 14-5
 - accessing from context menu 14-4
 - described 12-7, 14-3
 - using 14-19
- source connectors 1-7, 6-12, 25-13
- source schema 18-3
- specifying the default BusinessWare server 22-9
- SQL queries 17-16
- SSL 9-22
- start components only 3-9
- start states
 - described 12-4
 - using multiple start states 13-22
- statechart diagrams 10-2
- stateful models 10-5
- stateless models 10-5
- states
 - action states 12-5
 - activity states 12-5
 - adding to model 12-10
 - defined 10-2, 12-3
 - entry actions 12-7, 14-2
 - exit actions 12-7, 14-2
 - nested states 12-5
 - properties 12-8
 - simple states 12-4
 - start states 12-4
 - terminator states 12-4
 - timers on states 13-1
 - transformer states 12-6
 - types of 12-3
 - Web Service State 9-4
 - Web services states 12-6
- statistics log interval 3-8
- stop 26-6
- string 5-5
- struct 5-5
 - struct attribute 11-9
- subordinate 16-22
- subscriber application 17-24
- substitution parameter options 23-10
- substitution parameters 23-4, 23-10
 - specifying values 23-10
- symbolic identifiers B-5
- synchronizing projects 3-13
- synchronous communication
 - EJBs 8-3
- synchronous integration 8-3
 - application servers 8-3
- syntax
 - aggregation query 18-9
 - simple filter query 18-8
- system exceptions 26-2
- system parameters 23-4

T

- target connectors 1-7, 6-12, 24-9
- target schema 18-3
- Task List 16-5
- Task Manager 16-26
 - creating multiples 16-36
 - deploying 16-33
 - projects 16-26
- template 10-17

RouteByPortName 15-3
 RouteByPortType 15-3
 router 15-1
 Template Chooser 6-5, 6-40, 7-19, 15-4, 17-5, 26-16
 terminator states
 defined 12-4
 role in nested models 13-13
 text file logger properties 20-12
 third-party
 directory server 24-2
 RDBMS server 24-2
 thread pool
 Integration server 16-33
 task fetcher 16-33
 Web server 16-33
 throw exception 26-6
 throwOriginalException() 26-15
 time-based aggregate query 18-12
 time-based aggregation 17-8
 time-based filter 17-8
 time-based filter query 18-11
 time-based queries
 described 18-11
 time-based query 17-8
 timeCreated 18-3
 timeEntered 18-3
 timeout transitions 12-13
 timer cache size 3-9
 timer load duration 3-9
 timers
 absolute timer 13-2
 described 13-1
 relative timer 13-2
 setting a timeout 13-2
 trace levels 20-4, B-12
 trace messages B-2, B-12
 tracing code B-14
 transaction
 abort 25-8
 and Integration Servers 25-16
 and modeling 25-11
 application 25-10
 control property 25-17
 manager 25-6
 overview 25-1
 participation 25-13
 precommit 25-8
 protocols 25-5
 resource 25-4
 scopes 25-12
 standards 25-10
 types 25-4
 transaction application 25-10
 transaction control property
 mandatory 25-17
 required 25-17
 requires new 25-17
 transaction manager 25-6
 transaction manager properties 20-7, 25-16
 duration timeout 25-16
 inactivity timeout 25-16
 transaction propagation 25-17
 transaction properties 25-3
 atomicity 25-3
 consistency 25-3
 durability 25-3
 isolation 25-3
 transaction protocol
 one-phase commit 25-5
 two-phase commit 25-5
 transaction service 25-10
 transaction types
 multiple-server 25-5
 single-server 25-4
 transactional publishing 25-13
 transactionality
 dynamic 25-11
 nontransactional 25-11
 one-phase 25-11
 two-phase 25-11
 unknown 25-11
 transformer models
 introduction do 1-4
 transformer states 12-6
 transitions
 adding to model 12-14
 default transitions 12-17
 ECA specifications 10-3, 14-2
 exception transitions 12-13
 how transitions are evaluated 12-18
 how transitions work 12-11
 normal transitions 12-13
 properties 12-13
 self-transitions 12-13
 timeout transitions 12-13
 two-phase commit 25-7
 types 1-16, 22-6
 and structured editor 1-16
 boolean 5-5
 BPOs 5-3
 channels 7-10

INDEX

char 5-5
data object 10-8
data objects 5-4
double 5-5
enum 5-5
event interface 5-3
exceptions 10-4
existing 1-16
exporting 5-10
float 5-5
IDL 1-16
inspecting and editing 5-9
introduction to 1-9
Java 1-16
long 5-5
long double 5-5
long long 5-5
module 5-3
octet 5-5
overview 5-1
sequence 5-5
sharing 5-10
short 5-5
string 5-5
struct 5-5
union 5-5
unsigned long long 5-5
unsigned short 5-5
wchar 5-5
WDSL files 1-16
wstring 5-5

U

undeploy 22-8, 22-19
channels 22-20
Integration Servers 22-21
queues 22-20
Unified Modeling Language 10-2
union 5-5
unpartitioning 22-11
unsigned long long 5-5
unsigned short 5-5
update event
 add 17-22
 change 17-22
 creating 17-22
 delete 17-22
 processing 17-23
 visualizing 17-24
URI 9-18

user-interface labels B-2
using
 query builder 17-9

V

validate 17-12, 22-8
Version 3-6
versioning projects 3-23
void `closeOutputPort(String outputPortName)` 26-26
vtadmin 22-16
vtadminreplay 7-18

W

watches, used in debugging 27-16
wchar 5-5
Web applications directory structure 20-15
Web browser 24-2
Web server
 and servlet engine 24-9
 external 20-15
 internal 20-14
 properties 20-13
web server name 3-9
Web server settings
 external mode 24-9
 internal mode 24-9
 none 24-9
Web service input proxy
 configuring 9-12
 description 9-8
 overview 9-3
 properties 9-10
Web Service Output Proxy
 configuring 9-18
 description 9-13
 overview 9-3
 properties 9-15
Web Service State 9-4
 configuring 9-28
 invoking a Web service 9-27
Web services
 exporting 9-25
 from a Web service input proxy 9-27
 from an input port 9-27
 from the project 9-26
 input proxy 9-8
 configuring 9-12, 9-18

- properties 9-10
- interfaces 9-9
 - Service Endpoint 9-9
- introduction 9-1
- invoking 9-27
- objects 9-3
- output proxy 9-13
 - properties 9-15
- proxies 9-3
- sample files 9-28
- state 9-4
- supported standards 9-2
- Web Services Description Language (WSDL) 1-7, 9-1, A-1
- Web services states 12-6
- WebLogic 25-19
- where clause
 - described 18-5
 - quantifiers 18-5
- wires
 - introduction to 6-4
- workflow
 - manager 16-3
- models 10-5, 16-5
- overview 16-1
- participants 16-3
- performer 16-3
- performer assignment percentage 16-9
- performer assignment policy 16-8, 16-23
- performer properties 16-8
- reassignable 16-10
- rejectable 16-10
- roles 16-3
- supervisor 16-3
- workflow audit channel 3-8
- workflow audit enabled 3-8
- Workflow Management Coalition 16-1
- WSDL 9-1
 - dependencies 9-25
- wstring 5-5

X

- XQuery Builder 14-3

INDEX