

Oracle® Communications Order and Service Management Modeling Guide



Release 8.0
G37998-01
October 2025

ORACLE®

Copyright © 2015, 2025, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

About This Content

Part I Modeling OSM Solutions Overview

1 OSM Solution Modeling Overview

About the OSM Solution Modeling Process	1
About Determining the OSM Functionality to Implement	4
Solution Modeling Considerations	6
General Solution Data Modeling Principles	6
Performance Considerations	7
Planning OSM COM Solution Requirements	7
Modeling COM Order and Order Recognition Requirements	7
COM Data Modeling Considerations	8
Modeling COM Orchestration Order Items and Binding Conceptual Model Parameters	9
Modeling COM Orchestration Order Item Decomposition	10
Modeling COM Orchestration Fulfillment Patterns and Fulfillment Modes	12
Modeling COM Order Transformation Manager	15
Modeling COM Orchestration Dependencies	18
Modeling COM Processes and Tasks	19
Modeling COM Fallout Scenarios	20
Modeling COM Fulfillment States	21
Modeling COM Processing States	23
Modeling Change Order Management for COM	24
Cartridge Management Considerations for COM	24
Planning OSM SOM Solution Requirements	24
Modeling SOM Order and Order Recognition Requirements	25
SOM Data Modeling Considerations	25
Modeling SOM Orchestration Order Items and Bindings Conceptual Model Parameters	26
Modeling SOM Orchestration Order Item Decomposition	26
Modeling SOM Orchestration Fulfillment Patterns and Fulfillment Modes	27

Modeling SOM Orchestration Dependencies	28
Modeling SOM Processes and Tasks	28
Modeling SOM Fallout Scenarios	29
Modeling SOM Fulfillment States	29
Modeling SOM Processing States	30
Modeling Change Order Management for SOM	30
Cartridge Management Considerations for SOM	31
Planning OSM TOM Solution Requirements	31
Modeling TOM Order and Order Recognition Requirements	31
TOM Data Modeling Considerations	32
Modeling TOM Orchestration Order Items and Bindings Conceptual Model Parameters	33
Modeling TOM Orchestration Order Item Decomposition	33
Modeling TOM Orchestration Fulfillment Patterns and Fulfillment Modes	34
Modeling TOM Orchestration Dependencies	35
Modeling TOM Processes and Tasks	36
Modeling TOM Fallout Scenarios	36
Modeling TOM Fulfillment States	37
Modeling TOM Processing States	37
Modeling Change Order Management for TOM	37
Cartridge Management Considerations for TOM	38
About the OSM SDK	38

Part II Implementing an OSM Solution

2 Modeling Orders and Permissions

Modeling OSM Orders	1
About OSM Orders Without Orchestration	3
About OSM Orders With Orchestration	3
Modeling Roles and Setting Permissions	4
About Order Types	6
About Order Updates	7
Using a Job Control Order to Manage Multiple Orders	8
About Job Control Order Operations	11
About Job Control Order Permissions	11
About Job Control Order System Configuration Files	13
Viewing Orders in OSM Web Clients	13
Specifying Which Data to Display in the OSM Web Clients	13
Modeling Query Tasks for OSM Clients	13

3 Modeling Order Life-Cycle Policies

Modeling Order Life-Cycle Policy States and Transitions	1
About Modeling Transition Conditions	1
About Modeling Transition Grace Periods	2
About Modeling Transition Permissions	3
OSM Order States and Transactions	3
About Order State Categories	8
Common Order State Transitions	8
Optional, Mandatory, and Prohibited Transactions	10
About the Aborted Order State	12
About the Amending Order State	13
About the Cancelled Order State	15
About the Cancelling Order State	17
About the Completed Order State	18
About the Failed Order State	19
About the In Progress Order State	21
About the Not Started Order State	23
About the Suspended Order State	24
About the Waiting Order State	26
About the Waiting for Revision Order State	28
About Deleting Orders	29

4 Modeling Order Recognition

About Sending Orders to OSM and Order Recognition	1
Modeling Order Recognition Rules	2
Validating Incoming Order Data	3
Transforming Order Data	3
Modeling the Order Data Rule to Populate the Creation Task	3
Modeling Order Priority	4
Configuring JMS Message Priority on JMS Queue	5
Creating a JMS Destination Key	5
Configuring Destination Key for a JMS resource	5
Creating and Configuring JMS Destination Key in OSM Cloud Native	6
Modeling the Order Reference Number	6
Modeling a Catch-All Recognition Rule	6
Common Order Recognition Errors	6

5 Modeling Orchestration Plans

Orchestration Plan Overview	1
-----------------------------	---

Modeling an Orchestration Plan	3
About Component Names and Component IDs	6
About Order Items	6
About Creating Order Items from Customer Order Line Item Node-Sets	10
About Associated Order Items	11
Modeling Order Item Hierarchies	13
About Using a Distributed Order Template	14
About Mapping Order Items to Fulfillment Patterns	15
About Modeling Product Specifications	16
Modeling Fulfillment Modes	17
About the Decomposition of Order Items to Function Order Components	18
About Assigning Order Items to Fulfillment Pattern Function Components	18
About the Function Components Stage	19
About Order Component Control Data	19
About Fulfillment Pattern Conditions for Including Order Items	20
Summary of Order Item to Function Components Decomposition	20
About the Decomposition of Function to Target System Components	20
About Decomposition Rules from Function Components to Target Systems	20
About Decomposition Rule Conditions for Choosing a Target System	21
About the Target Systems Stage	22
Summary of Configuring Target System Components Decomposition	22
About the Decomposition of Target System to Granularity Components	23
About Decomposition Rules from Target System to Granularity Components	23
About Customized Component IDs for Separating Bundled Components	23
About the Granularity Components Stage	24
Summary of Configuring Granularity Components Decomposition	24
About Dependencies	24
About Intra-Order Dependencies	26
Modeling an Order Item Dependency	26
About Order Item Dependency Wait Conditions	27
About Order Item Dependency Wait Conditions Based on Data Changes	28
Modeling a Fulfillment Pattern Dependency	29
Modeling an Order Item Property Correlation Dependency	30
About Inferred Dependencies	30
About Modeling Orchestration Dependencies	31
About Processing Order Items Sequentially	32
About Inter-Order Dependencies	32
About Modeling Orchestration Dependencies	34
Using Task States to Manage Orchestration Dependencies	35

6 Modeling the Order Transformation Manager

Understanding the Order Transformation Manager	1
Order Transformation Manager in Runtime	1
The Order Transformation Manager and the Conceptual Model	1
OSM Entities Used in the Order Transformation Manager	2
Calling the Order Transformation Manager	4
Using the Distributed Order Template with the Order Transformation Manager	4
Modeling OTM With Calculate Service Order	5
Calculate Service Order Design Patterns	5
About the Calculate Service Order Provider Function	5
About Calculate Service Order Relationship Types	6
About the Calculate Service Order Transformation Sequence	6
User-Created Entities for Calculate Service Order	7
Modeling OTM Without Calculate Service Order	7

7 Modeling Processes and Tasks

Overview of Processes and Tasks	1
Modeling Processes	1
About Process Flows	1
Adding Process Activities	3
Configuring Subprocesses	4
Understanding Parallel Process Flows	5
About Amendments and Multi-Instance Subprocesses	5
About Order Rules in Processes and Notifications	5
Modeling Order Rules in Notifications	6
Using the System Date in Delays	7
Process and Task Design and Data Considerations for Compensation	7
Order Perspectives and Data Elements in Compensation	8
Effects of Process Loops on Compensation	8
Modeling Tasks Entities Common to All Task Types	10
Modeling Task States	10
Modeling Task Permissions and Execution Modes	10
About Normal and Fallout Execution Modes and Task States	11
Modeling Task Status Transitions	14
Specifying the Expected Task Duration	14
Specifying the Task Priority	15
About Extending Tasks	15
About Task Types	15
Modeling Automated Tasks	15
About Automation Plug-in and Automated Tasks	16

Completing an Automation Task That Handles Concurrent Status Updates	16
Modeling Manual Tasks	17
Deploying a Custom Task Algorithm using the OSM Cartridge Management Tool	18
Using a Custom Task Algorithm in OSM Cloud Native	21
Modeling Transformation Tasks	21
Modeling Activation Tasks	21
About Service Action Request Mapping	22
About Service Action Response Mapping	22
About Activation Tasks and Amendment Processing	22
About State and Status Transition Mapping for Activation Tasks	23
About Automation Plug-ins	23
Specifying Which Data to Provide to Automation Plug-ins	24
Modeling Query Tasks for Order Automation Plug-ins	24
About Automation Message Correlation	26
Example: Modeling a Basic Automator Plug-in for an Automated Task	27

8 Modeling OSM Data

Data Modeling Overview	1
Modeling Order Data	2
About the Data Dictionary	2
About the Order Template	3
Identifying Data Requirements for Order Payload	3
Adding the Input Message to an Order Recognition Rule	4
Adding the Input Message to the Order Template	5
Modeling Valid Data Keys	7
Modeling Data for Tasks	8
Determine Task Data for Manual and Automated Tasks	8
Determine Task Data for Data Returned from Fulfillment Applications	10
Generating Multiple Task Instances from a Multi-Instance Field	10
Modeling Data for Orchestration	11
About Order Item Control Data	12
About Order Template Data	13
About Order Item Specification Data	14
About ControlData for Order Component Data	14
Modeling Data for Fulfillment States	15
About ControlData for External Fulfillment States	16
About ControlData for Order Fulfillment State	16
About ControlData for Order Item Fulfillment State	16
Fulfillment States and Point of No Return	17
Fulfillment State and Point of No Return Initial Values	17
Sample XQuery for Changing Default Data Locations	17

Modeling Data for Processing States	19
About ControlData for Order Component Order Item Processing States	19
About ControlData for Order Item Processing States	19
Modeling Orders With Data Fields Above 1000 Characters	20
Using XML Types for Data Fields Above 1000 Characters	20
Using Order Remarks for Data Fields Above 1000 Characters	21
Using Attachments for Data Fields Above 1000 Characters	21
Using Data Providers to Retrieve Data	22
About Data Providers and Adapters	22
Data Provider Interface Tab	23
Accessing Data through Data Providers	24
Augmenting or Overriding Data	24
Objectel	25
Order	26
Adding a New Order Data Provider	26
Property File	27
SOAP	27
XML Attachment	30
XML File	31
XML Validation	31
JDBC	31
Web Service	32
Adding a New Web Service Data Provider	32
Sample soap.request XQuery	33
Accessing Data	33
Custom Data Providers	34
Handling Parameters	34

9 Modeling Behaviors

Modeling Behaviors Overview	1
About Behavior Evaluation	3
Evaluating Behavior Levels	4
Evaluating Design Studio Final and Override Options	4
Evaluating Behavior Type Precedence and Sequence	5
About Setting Conditions in Behaviors	9
Using the Calculation Behavior	10
Calculation Behavior XPath Examples	10
Calculation Behavior Overview	10
Using the Constraint Behavior to Validate Data	11
Displaying Constraint Behavior Error Messages	11
Evaluating Constraint Behaviors	12

Using Task Statuses to Control Process Transitions	12
Task Statuses and Constraint Behavior Violation Severity Levels	13
Constraint Behavior Overview	13
Using the Data Instance Behavior to Retrieve and Store Data	14
Evaluating Data Instance Behaviors	14
Data Instance Behavior XML, XPath, and XQuery Examples	14
Data Instance Behavior Overview	15
Using the Event Behavior to Re-evaluate Data	15
Event Behavior Overview	16
Using the Information Behavior to Display Data and Online Help	17
Information Behavior XPath Examples	17
Information Behavior Overview	18
Using the Lookup Behavior to Display Data Selection Lists	18
Lookup Behavior XPath Example	19
Lookup Behavior Overview	19
Using the Read-Only Behavior	19
Read-Only Behavior Overview	19
Using the Relevant Behavior to Specify if Data Should Be Displayed in the Web Client	20
Relevant Behavior Overview	21
Using the Style Behavior to Specify How to Display Data in the Task Web Client	22
About Style Behavior Layouts	24
About Style Behavior Password Fields	28
Style Behavior Overview	29

10 Modeling a TMF Solution (Cloud Native Only)

About Specifications	1
About Cancelling or Revising an Inflight Order	1
Modeling PONR	7
Change Order Support	8
Order Fulfillment Modes	8
Upstream Listener	9
About TMF Order Events For the External Event Listener	9
About Fallout Exception Management	9

11 Implementing a TMF Solution (Cloud Native Only)

Accessing the Specifications	1
About Extending the Specifications	1
Considerations When Extending the Main Resource	2
About Versioning the Specifications	2
About the "ANY" Schema Type	3

About anyOf, allOf, and oneOf	6
About TMF Cartridges and Non-TMF Cartridges	7
About Importing the Hosted Order Specification	8
About Fulfillment Modes	9
About TMF Order Lifecycle Policy	9
About Data Dictionary	10
About the Order Template	11
About the Master Order Template	11
About the Order Item Specification Order Template	11
About the Significance of CDT	15
About TMF Orders and Permissions	16
Permissions for Internal Gateway Role	16
About Order Recognition	16
About Updating the TMF Order Item with Downstream Data	19
Updates to Order Item Characteristics	19
Updates to General Order Item Data	23
Updates to External Fulfillment State	23
About TMF Order State	23
About TMF Order Item State	25
About Fulfillment State and Processing State	29

12 Modeling External REST Interactions using System Interaction (Cloud Native Only)

About Importing the OpenAPI Document into Design Studio	1
TMF APIs for BSS/OSS System Interactions	1
Importing a System Interaction	1
Updating a System Interaction Specification	2
System Interaction and OSM Order Components	2
Determining the Order Component	3
About Array of Arrays Support in System Interactions	5
About the OSM Gateway Functions	7
Support for JSON Patch and Merge-Patch Content Types in System Interaction	7
Considerations for OSM Cloud Native to OSM Cloud Native Integration using System Interaction	10
Developing Automation Plugins	10
Known Issues and Workarounds	10

Part III Modeling Run-time Order Management

13 Modeling Changes to Orders

About Amendment Processing and Compensation	1
About Revising or Canceling Orders by Using the Task Web Client	7
About Order Keys	7
About Submitting Multiple Revisions of an Order	8
About Compensation States	9
About Revising In-flight Revision Orders	9
About Insignificant Revision	10
About Terminating Compensation	11
Disabling Processing of Revisions on In-flight Revision Orders	11
Example: Revising an In-flight Revision Order	11
About Controlling When Amendment Processing Starts	13
About Compensation	14
About Order-Level and Task-Level Compensation Analysis	14
About Order Data Position and Order Data Keys	17
About Data Significance	18
About Task Execution Modes	21
Modeling Compensation for Tasks	23
Determining Task Compensation Strategy	23
About Compensating In Progress Tasks	26
About Task Compensation Strategy XQuery Expressions	27
About Managing Compensation in the Task Web Client	29
Modeling Compensation for Rules in Processes	29
Modeling Compensation for Task Automation Plug-Ins	29
Compensation Examples	30
Example 1: Compensation During Provisioning	30
Example 2: Compensation During Billing	30
Example 3: Amend Do Compensation	31
Examples of Changes to Orchestration Plans	32
Modeling a Point of No Return	35
Fulfillment Pattern Point of No Return	35
Life-Cycle Policy Point of No Return	35
About Modeling Order Change Management	36
Troubleshooting Order Change Management Modeling	37
About Order Change Management at the Orchestration Layer	37
About Compensation and Orchestration	38
About Point of No Return	39

14 Modeling Fallout

Overview of Fallout	1
Understanding Fallout Across OSM Roles	2
Understanding Fallout Sources	4
Managing Business Related Fallout Sources	4
Managing Fallout from Failures in Network or System Resources	5
Managing Fallout During Order Creation	6
Managing Fallout in the OSM Web Clients	7
Modeling Fallout in Tasks	8
About Failed Tasks and Execution Modes	8
About Alternate Task Fallout Management Methods	9
Modeling Task Notifications for Fallout	9
About Modeling Fallout Exceptions	9
Managing Fallout Exceptions in the Task Web Client	11
Simplified Fallout Exception Automation Framework (Cloud Native Only)	12
Modeling Fallout in Orders	14
Modeling the Failed Order State	14
Modeling Order Notifications for Fallout	15
About Terminating an Order	16
Managing Fallout in the OSM Order Management Web Client	17

15 Modeling Fulfillment States and Processing States

About Fulfillment States, and Processing States	1
Modeling Fulfillment States	1
Defining Fulfillment States	3
Modeling External Fulfillment States	3
Modeling Fulfillment State Maps	4
Modeling Fulfillment State Composition Rule Sets	5
Modeling Processing States	8
Order Component Order Item Processing States	9
Order Item Processing States	10

16 Modeling Jeopardy and Notifications

Best Practices for Using Notifications for Status Updates	1
Status Update Strategies	1
Strategies for Using Notifications	1
Modeling Notifications	2
Using Task States and Statuses to Trigger Event Notifications	2
About Notification Priority	2

About Sending Notifications in Email	2
About Configuring Entities to Support Notifications	3
About Jeopardy Notifications	3
About Modeling Jeopardy Notifications	3
About Jeopardy Notification Triggering	4
About Jeopardy Notification Conditions	5
Specifying Jeopardy Notification Conditions in the Order Jeopardy Editor	5
Specifying Jeopardy Notification Conditions in the Order Editor	6
Specifying Jeopardy Notification Conditions for a Task	6
About Event Notifications	6
About Using Task Transitions to Trigger Event Notifications	7
About Using Task States and Rules to Trigger Event Notifications	8
About Using Task States to Trigger Automated Event Notifications	9
About Using Order Milestones to Trigger Event Notifications	10
About Using Order Data Changes to Trigger Notifications	12
About Enabling Order Life-Cycle Events	13
Summary of Notification Functionality	13

17 Modeling Milestone Events

About Milestones and Model-driven Milestones	1
Usage of Milestone Events	2
Modeling Model-driven Milestones	3

18 Modeling Order Scheduling

About Order Item Requested Delivery Date and Order Components	1
How OSM Decomposes and Processes Order Items in Order Components	2
About Grouping Order Items in Order Components by Date Range	3
Modeling Order Component Minimum Processing Duration	3
About Minimum Processing Duration Inheritance in Fulfillment Patterns	5
About Minimum Processing Duration Expressions	6
Calculating the Earliest Order Component Start Date (Order Start Date)	7
About Calculated Order Component Start Dates	7
Modeling Order Component Dependencies and Requested Delivery Dates	9
Modeling Order Items Processed by Multiple Dependent Order Components	9
Revisions of Future-Dated Orders	9
Examples of Calculating the Expected Start Date	10
Example 1: Calculating Start Dates for Order Components with No Dependencies	10
Example 2: Calculating Start Dates for Order Components with Dependencies	11

19 Managing OSM Solution Cartridges

Solution Management Overview	1
About OSM Cartridge Scope	2
Scope of OSM Entities Without Namespaces	3
Design Studio Entities	3
XML Catalogs and Resource Files	3
Scope of OSM Entities with Namespaces	3
Standalone Cartridge Scope	4
XML Catalog Files in Standalone Cartridges	5
Avoiding Namespace Collisions for Design Studio Entities	5
Avoiding Namespace Collisions for Resource and XML Catalog Files	6
Composite Cartridge Scope	8
Special Cases for Scope	10
Order Recognition Rules	10
Fulfillment Patterns	10
Managing Cartridge Versions	12
Making Changes to Existing Cartridge Versions	13
Handling Multiple Cartridge Versions	14
Migrating Orders to a New Version of a Cartridge	15
Designation of the Default Cartridge Among Cartridge Versions	15
Handling Revision Orders When Multiple Cartridge Versions Are Deployed	16
Working with Cartridges in OSM Cloud Native	16
Building and Packaging a Cartridge	17
About Generating OSM Cartridges and Deployment Options	17
About Cartridge Types	18
About Design Studio Editors for OSM Cartridges	18
Organizing Design Studio and Naming Conventions	21
Cartridge Packaging Design	22
Modifying the Build	23
About XML Catalogs	23
Using XML Catalogs in OSM	24
Resource Packaging Considerations for Using XML Catalogs	25
Defining rewriteURI Entries in XML Catalogs	26
Specifying XML Catalogs for OSM	28
Enabling and Disabling XML Catalog Support	29
Examples of Using XML Catalogs	29
Using XML Catalogs to Support Cartridge Versioning	30
Using XML Catalogs to Load Resources from a Development File System (Traditional OSM Only)	30

Using XML Catalogs to Insulate Run-Time Environments from Development	31
Cartridge Deployment	32
Cleaning and Rebuilding Cartridges Prior to Deployment	32
Optimizing Cartridge Deployment	32
Deploying Multiple Cartridges	32
Deploying Cartridges with Dependencies	32
Deploying Cartridges to the OSM Database Using XMLIE	33
Building and Deploying Composite Cartridges	36
Setting Cartridge Dependencies	36
Post-Deployment Effect on Numeric Data	36
Post-Deployment Changes to Cartridge	36
Metadata Errors	36

Part V Working with Capabilities Cartridges

20 Working with Capabilities Cartridges (Cloud Native Only)

About Capabilities Cartridges	1
About Capabilities Cartridges Restrictions	1
User Workflow in Design Studio	1
Capabilities Cartridge Lifecycle	2
Creating and Delivering the CPAR	3
Updating and Re-Delivering the Capabilities Cartridge	3
Retiring a Capabilities Cartridge	4
Capabilities Cartridge Content	5
Design Studio Workspace	5
Capabilities Cartridge Conventions	5
Fulfillment Model	5
Order Recognition Rule	6
XML Catalogs	6
Automation Concurrency Map	6
Relationship Types	6
Provider Function	7
Description Fields	7
Fulfillment Pattern Property on the Order Item Specification	7
Transformed Order Item Properties	7
Order Component Organization	8
Configurability Limitations	8
General Restrictions	8
Mapping Rules	8
Transformation Sequence	9

Decomposition and Routing Rules	10
Cartridge Versioning	11
DataTypes for Product and Service Attributes	11
Transformed Order Item Properties	12
Capabilities Cartridge Test Scaffolding	12
Fulfillment Model	12
OSM Enrichment Data	12
Cartridge Guidelines for Test Data	13
OSM Test Content	13
Conceptual Model Test Content	14
Reusable Conceptual Model Content	15
Capabilities Cartridge Build	16
Using the Capabilities Cartridge Wizard	16
About the CPAR File	16
Capabilities Manifest Details	17

A Behaviors Quick Reference

OSM Behavior Type Overview	A-1
Common Behavior Elements	A-3
Annotation Element	A-3
Description Element	A-3
Instance Element	A-3
Adapter Element [externalInstanceType]	A-3
Parameter Element [externalInstanceType]	A-3
Cache Element	A-4
Expression Element	A-4
Declaring Behaviors in OSM XML Model	A-4
Data Dictionary Level	A-4
Master Order Template Level	A-4
View Level	A-4
Data Provider Overview	A-4
Programmatic Behavior Implementation Overview	A-5

B XQuery Examples

General XQuery Information	B-1
About Creating XQuery Expressions with Design Studio	B-1
OSM XQuery Functions	B-2
Referencing Items from a Distributed Order Template in XQuery Expressions	B-3
Order Recognition Rule XQuery Expressions	B-4
About Recognition Rule XQuery Expressions	B-4

About Validation Rule XQuery Expressions	B-5
About Order Priority XQuery Expressions	B-6
About Order Reference XQuery Expressions	B-7
About Order Data Rule XQuery Expressions	B-7
Decomposition XQuery Expressions	B-9
About Orchestration Sequence XQuery Expressions	B-9
About Order Sequence Order Item Selector XQuery Expressions	B-9
About Order Sequence Fulfillment Mode XQuery Expressions	B-10
About Order Item Specification XQuery Expressions	B-10
About Order Item Specification Order Item Property XQuery Expressions	B-10
About XQuery Expressions for Mapping Product Specifications and Fulfillment Patterns	B-12
About Order Item Specification Order Item Hierarchy XQuery Expressions	B-14
About Order Item Specification Condition XQuery Expressions	B-16
About Fulfillment Pattern Order Component XQuery Expressions	B-17
About Fulfillment Pattern Order Component Condition XQuery Expressions	B-17
About Associating Order Items Using Property Correlations XQuery Expressions	B-17
About Decomposition Rule Condition XQuery Expressions	B-20
About Component Specification Custom Component ID XQuery Expressions	B-21
Custom Order Component IDs Based on Hierarchy	B-22
Custom Component IDs Based on Requested Delivery Date and Duration	B-25
Custom Component IDs by Duration and Minimum Separation Duration	B-26
Combining Order Item Hierarchy with Duration-Based Groupings	B-27
About Component Specification Duration XQuery Expressions	B-28
About Fulfillment Pattern Duration XQuery Expressions	B-28
About Fulfillment Pattern Component Duration XQuery Expressions	B-28
Dependency XQuery Expressions	B-29
About Order Item Dependency Property Correlation XQuery Expressions	B-29
About Wait Delay Duration XQuery Expressions	B-30
About Wait Delay Date and Time XQuery Expressions	B-31
About Order Data Change Wait Condition XQuery Expressions	B-33
About Order Item Inter-Order Dependency XQuery Expressions	B-34
Order Transformation Manager XQuery Expressions	B-36
About Transformation Sequence XQuery Expressions	B-36
About Order Item Context XQuery Expressions	B-36
About Related Order Item Selector XQuery Expressions	B-36
About Stage Condition XQuery Expressions	B-37
About Mapping Rule XQuery Expressions	B-37
About Mapping Condition XQuery Expressions	B-38
About Action Mapping XQuery Expressions	B-38
About Entity-to-Entity Advanced Mapping XQuery Expressions	B-38
About Entity-to-Data-Element Advanced Mapping XQuery Expressions	B-39

About Data-Element-to-Data-Element Advanced Mapping XQuery Expressions	B-39
About Reverse Mapping XQuery Expressions	B-40
About Multi-Instance XQuery Expressions	B-40
About Order Item Parameter Binding XQuery Expressions	B-41
About Transformed Order Item Fulfillment State XQuery Expressions	B-41

About This Content

This guide provides modeling information about Oracle Communications Order and Service Management (OSM).

Audience

This guide is intended for:

- Business domain experts who make decisions about the order fulfillment process.
- Order management personnel who need to know how OSM works and how orders are processed.
- Developers who extend OSM to interface with external systems.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

Conventions

The following text conventions are used in this document.

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I

Modeling OSM Solutions Overview

Part I contains the following chapter providing an overview of modeling Oracle Communications Order and Service Management (OSM) solutions:

- [OSM Solution Modeling Overview](#)

1

OSM Solution Modeling Overview

This chapter provides an overview of an Oracle Communications Order and Service Management (OSM) solution.

Before reading this chapter, read *OSM Concepts* to learn about general OSM concepts.

Note

In this guide, “traditional OSM” refers to the traditional way of installing and maintaining an OSM environment and “OSM cloud native” refers to OSM deployed in a cloud native environment.

About the OSM Solution Modeling Process

An OSM solution is part of a larger operations support system (OSS) and business support system (BSS) solution. The OSM solution brings together the elements relating to order processing within an overall OSS and BSS solution. To understand how the OSM solution fits into this OSS and BSS solution, you must do the following:

1. Scope the solution and perform an initial analysis: This stage is where you decide the nature of the business change required for the OSS and BSS solution at a high level. Generally, an OSS and BSS solution falls under the following scope categories:
 - Solutions that involve adding or changing product offerings with no effect on the underlying service or IT infrastructure. For example, the marketing department wants to create a new offering category with new discounts and incentives.
 - Solutions that involve adding of or changing both product offerings and the underlying service and IT infrastructure. For example, a company may expand their product offerings from broadband Internet and email to include a mobile offering. This change required adding new product categories, offering and bundling possibilities, new underlying services, and new IT infrastructure requirements.
 - Solutions that involve adding or changing the network fulfillment infrastructure. For example, adding new network technology, the upgrade of existing network technology, the expansion of the company into new geographical locations, and so on.
 - Solutions that involve additions of or changes to the BSS and OSS service fulfillment IT infrastructure. For example, the addition of new service fulfillment systems, such as billing, activation, work force management, or partner gateway systems.
2. Plan, analyze, and design the solution: You plan, analyze, and design a solution primarily by creating an Oracle Communications Service Catalog and Design - Design Studio conceptual model (see *Design Studio Concepts* for more information). You can use conceptual model entities to capture the impact of the changes specified in the initial solution scope. Such entities may include:
 - Products: Here you capture any changes to simple products, bundles of products, and offerings, including the data required at this level.

- Customer-facing services (CFSs): A CFS represents the service that the customers want. Here you capture the impact of any product and resource changes. You need to determine whether an existing CFS require changes as a result of product-level change or resource-level change.
 - Resource-facing services (RFSs) and resources: An RFS represents the technology options available to implement a service. Here you capture the technology options available to fulfill a CFS and the parameters required. For example, a broadband CFS may have the DOCSIS, GPON, or DSL RFS options available, each of which specifies one or more resources that represent specific instances and versions of the RFS technology category.
 - Actions: You can model the specific actions available for each CFS and RFS. The actions represent subsets of CFS and RFS entity data. For example, an add action may require that all parameters of a CFS be populated, but a change action may require only a subset of the parameters.
 - Location: You can designate which locations support what resources and services.
 - Fulfillment patterns: You indicate which conceptual model fulfillment patterns are involved in processing products, CFSs, RFSs, and resources. For example, products relating to broadband Internet may require a different fulfillment pattern than products relating to mobile service or IP TV.
3. Implement the solution: You model OSM application entities and data in Design Studio to realize the conceptual model entities you created in the planning, designing, and analyzing phase. Keep in mind that the OSM solution is closely interrelated with other OSS and BSS solutions, such as billing, activation, service resource management (SRM), workforce management (WFM), and partner gateway (PGW) solutions.

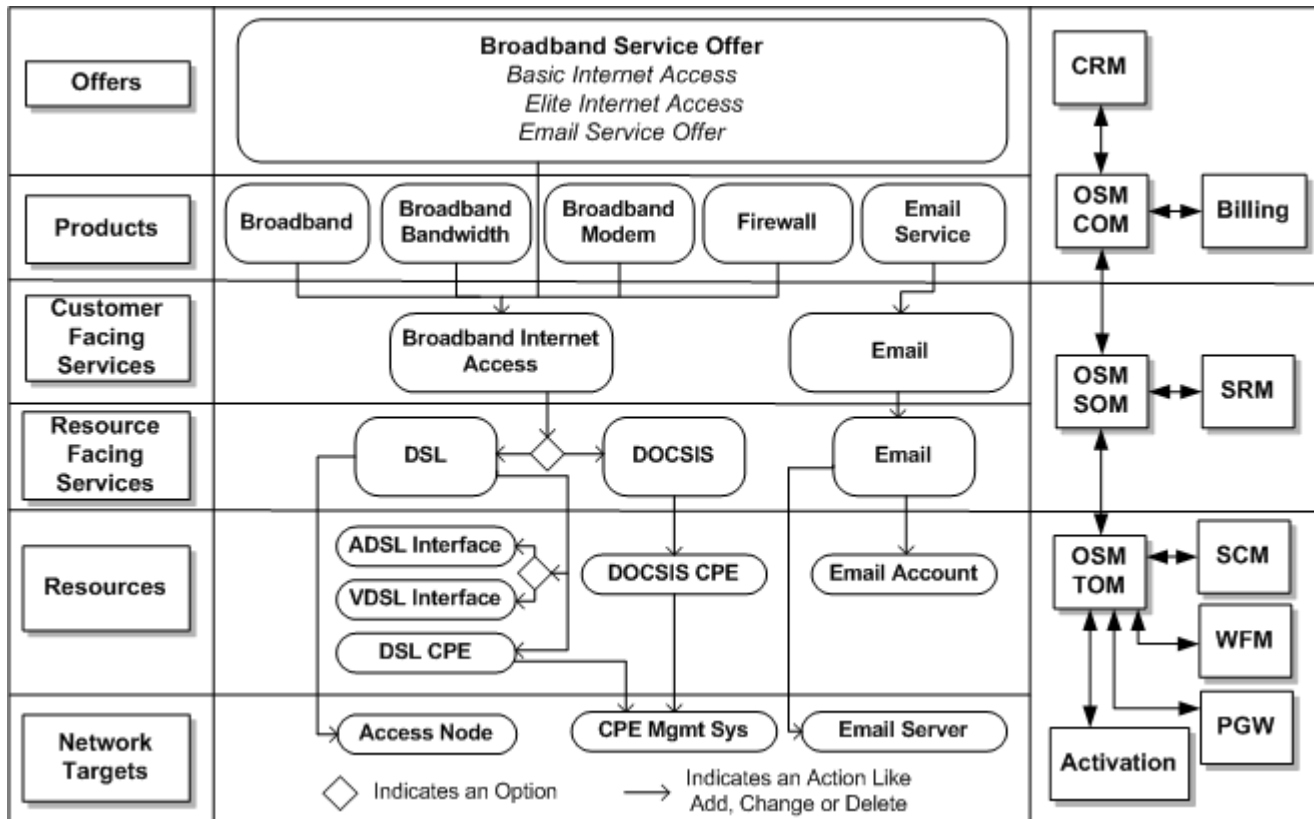
[Figure 1-1](#) represents a conceptual model that defines all offers, products, CFSs, RFSs, resources, network targets, and actions that a fictional communications service provider (CSP) requires to fulfill a sample broadband Internet and email service. The CFS and RFS entities unify the business and marketing concerns represented by the products and offers with the IT infrastructure concerns represented by the resources and network targets. The CFS and RFS entities also decouple the changes that occur in products, offers, from the changes that occur in resources. For example, for business and marketing, products and offers are changed frequently. Likewise for IT infrastructure, technology, vendors, and vendor versions are changed frequently. But the underlying services being offered and the underlying technology types do not change often.

[Figure 1-1](#) also shows the OSM roles and fulfillment systems involved in fulfilling orders containing the data defined in the conceptual model for this fictional CSP.

Note

[Figure 1-1](#) shows each application as a separate system, however these applications can also be co-resident.

Figure 1-1 Sample Conceptual Model



You can use this sample conceptual model as a basis for modeling data and functions generated by orders. The following shows the OSM roles that run the functions that fulfill the sample conceptual model entities:

- **Central order management (COM) role**

OSM in the COM role manages **sales orders** sent from a customer relationship management (CRM) system. The sales orders contains offer and product information. Functions at this level include:

1. Synchronizing customer account information between the CRM system and the billing system. Customer account information can be name, address, account details, order number, billing profile, and so on.
2. Updating service subscription details in the billing system so that the billing system can begin to collect service usage information.
3. Transforming the products and offers into CFSs and sending them to OSM in the SOM role as a **service order**.
4. Billing for usage by updating service subscription details in the billing system after the provisioning function has completed, and then notifying the CRM system that the sales order is complete.

- **Service order management (SOM) role**

OSM in the SOM role manages service orders sent from the provisioning function of OSM in the COM role. Functions at this level include:

1. Sending CFS information to an SRM system so that the SRM system can design a service instance based on the RFS specification, allocate resources to the service instance, and specify what needs to be configured on the resources to support the features, qualities, and policies of the service.
 2. Requesting resource actions from the SRM system, which are the actions that need to be performed by OSM in the TOM role and by the fulfillment systems communicating with the TOM role.
 3. Sending the **technical order** containing the resource actions to OSM in the TOM role. The technical order outlines the work that must be performed to enable the service design in the network. Some actions impact the WFM system, some the activation system, and so on.
 4. Completing the service order when OSM in the TOM role completes the technical order, and updating the OSM instance in the COM role.
- **Technical order management (TOM) role**

OSM in the TOM role manages technical orders sent from OSM in the SOM role. OSM in the TOM role decomposes each resource in the technical order into the appropriate functions and target system process. Functions at this level include:

 1. Sending actions to a supply-chain management (SCM) system for selecting, packing, and shipping physical goods to the destination selected by the customer.
 2. Sending actions to a partner gateway (PGW) used to manage relationships with third-party suppliers or partners that provide services or infrastructure involved in fulfilling the order. For example, the last mile of a telecommunication network involved in service delivery is often owned by a third-party telecommunications company.
 3. Sending actions to an activation system involved in configuring and activating network resources.
 4. Sending actions to a workforce management (WFM) system to dispatch a technician to perform work in the field.
 5. Completing the technical order when the fulfillment systems involved with OSM TOM complete their tasks and updating the OSM instance in the SOM role.

You must also analyze data and function requirements for other order processing scenarios, such as managing order fallout, managing order changes, tracking fulfillment states and processing states as orders are processed, managing notifications to upstream systems, and so on.

After you have completed this analysis and design stage, you can model entities in OSM Design Studio projects. You can then generate cartridges from those projects that you can deploy to OSM servers for development test environments and finally to production environments.

About Determining the OSM Functionality to Implement

After you have analyzed the information contained in the conceptual model, you must determine the following functionality to implement in the OSM solution you are planning:

- What kinds of orders you need to model for OSM roles (COM, SOM, and TOM) and what kinds of order life-cycle policies the orders need.

See "[Modeling Orders and Permissions](#)" and "[Modeling Order Life-Cycle Policies](#)" for more information.

- What kinds of order recognition rules each OSM role requires to capture incoming customer, service, technical, or revision order types.
See "[Modeling Order Recognition](#)" for more information.
- What kinds of order items each OSM role needs to fulfill based on the conceptual model entities and actions.
See "[Modeling Orchestration Plans](#)" for more information.
- What kinds of fulfillment modes, fulfillment patterns, order decomposition and dependencies you require based on the OSS and BSS solution requirements and order fulfillment flows.
See "[Modeling Orchestration Plans](#)" for more information.
- What kinds of order item and order component scheduling you need when fulfilling your orders.
See "[Modeling Order Scheduling](#)" for more information.
- What kinds of tasks and processes you need to implement for each order component function, what systems to target, and what order or order item granularity is required when sending messages to the target systems. For example, do you configure automated tasks to send all the order items that are decomposed to the function that triggers the process, or do you generate separate functions that trigger separate processes for each bundle of order items contained in the order?
See "[Modeling Processes and Tasks](#)" for more information.
- What kinds of manual tasks you need to implement in the OSM Task web client and what kinds of behaviors the tasks should exhibit. The goal of any OSM solution is to automate tasks as much as possible; however, sometimes manual tasks are necessary. For example, when initially creating a solution, you might want to model all automated tasks as manual tasks first, and then convert them to automated tasks after you have a better understanding of what the tasks must do.
See "[Modeling Processes and Tasks](#)" and "[Modeling Behaviors](#)" for more information.
- What kinds of fulfillment states and processing states you need to configure for the customer, service, technical orders, and order component order items. In addition, you must determine what messages from external systems trigger fulfillment state and processing state changes.
See "[Modeling Fulfillment States and Processing States](#)" for more information.
- Whether you need to use the conceptual model Calculate Service Order provider function with the order transformation manager.
See "[Modeling the Order Transformation Manager](#)" for more information.
- What kinds of change order management scenarios you expect for COM, SOM, and TOM orders.
See "[Modeling Changes to Orders](#)" for more information.
- What kinds of notifications you need to set up that would be specific to the order component functions and process tasks of each OSM role.
See "[Modeling Jeopardy and Notifications](#)" for more information.
- What kinds of fallout scenarios to anticipate and how to recover from them.
See "[Modeling Fallout](#)" for more information.

The following sections provide details about the different ways you can implement these OSM functions in general and in COM, SOM, and TOM contexts that are part of an overall BSS and OSS solution:

- [Solution Modeling Considerations](#)
- [Planning OSM COM Solution Requirements](#)
- [Planning OSM SOM Solution Requirements](#)
- [Planning OSM TOM Solution Requirements](#)

You implement these functions differently in each OSM role.

Solution Modeling Considerations

It is important to plan your solution implementation before modeling your solution. The following sections provide some general guidelines for solution modeling.

General Solution Data Modeling Principles

Modeling an OSM solution involves creating orders that contain the data involved in fulfilling actions such as add, change, delete, modify, move, and so on, on a product, service, or resource. In general, when you begin to model an OSM solution, you must understand the following data modeling principles:

- You must identify where the data you define comes from. For example, most data is defined in the CRM system in response to a request from a customer, but other data may be generated by downstream fulfillment systems that OSM interacts with.
- You must identify which system is the primary owner of each data structure or element. This principle is especially important in change order management and fallout management scenarios, where OSM must update data to modify or correct the fulfillment of an order.

For example, if the SRM system that interacts with OSM SOM provides faulty network resource data that generates an error in the activation system that OSM TOM interacts with, then the SRM system must correct the faulty network resource data. Although it may be possible to correct the problem directly in the OSM TOM task that communicates with the activation system, this does not resolve the root problem, which originated in the SRM system. Allowing OSM TOM to correct the problem also causes the network resource data to be inconsistent between OSM TOM and the activation system, and between OSM SOM and the SRM system.

- You must understand how the data is propagated throughout OSM systems and service fulfillment systems. Although OSM should not add, change, or modify data owned by a fulfillment system, OSM does sort, route, format, and send the data so that other fulfillment systems can consume the data in the format they require.

The solution that the OSM solution is a part of may contain an integration layer that determines a canonical format for data and provides standard interfaces to which OSM must conform. For example, Oracle Application Integration Architecture (Oracle AIA) integrates Oracle applications, such as OSM, Siebel Customer Relationship Management (Siebel CRM), and Oracle Communications Billing and Revenue Management (BRM), and also provides a standard format for message exchanges.

When an OSM solution is not part of a solution with an integration layer OSM must conform to the data requirements and interfaces of each external fulfillment system.

Use these principles to clearly understand how order data is kept in OSM systems, and how data is communicated at the interactions between OSM systems and other fulfillment systems for every fulfillment action performed by the OSM solution.

See "[Modeling OSM Data](#)" for more information about modeling data in OSM solutions.

Performance Considerations

When modeling a solution, you should always keep solution performance in mind. The following factors have an impact on performance:

- Number of tasks in a process.
- Number of concurrently executing automation plug-in instances.
- Number, size, and depth of data elements in the order.
- Order view (query task) size and complexity.
- Number of order line items and complexity in an incoming order.
- Degree of XSLT and XQuery transformation.
- Complexity of the generated orchestration plan.
- Average number of revision orders per base order.

If you have not taken these factors into consideration when modeling your solution, and the results of system performance testing are unsatisfactory, you may have to change the solution modeling. For information about other system performance considerations, see *OSM System Administrator's Guide* and *OSM Installation Guide*.

Planning OSM COM Solution Requirements

This section describes OSM modeling entities and functions involved in the OSM COM role. It includes examples intended to guide you in understanding how you can use these entities and functions in your OSM COM solution. This information can help you plan your implementation efforts by exemplifying at a high level the full scope of work involved in modeling a typical COM solution. Follow the links in each section for specific details about the functions described.

You typically model COM solution changes based on adding and changing new products, bundles, and offers that reflect purely business and marketing concerns or that also reflect changes to the SOM and TOM solution, such as when a company introduces a new technology domain.

Modeling COM Order and Order Recognition Requirements

You need to determine what kinds of orders to model in OSM, what order life-cycle states and state transitions the orders have, what user roles (workgroups) have permissions to perform tasks in fulfilling the orders, and what data should be visible to each user role (workgroup).

For example, you model customer orders for new orders, whether an in-progress order can be revised, and fallout orders that are triggered when customer orders or revision orders fail. You enable various order life-cycle states for each order, such as Not Started, In Progress, Canceling, Amending, and so on, and what transitions are possible from state to state. You must determine what personnel or systems have permissions to perform order state transitions and other functions and tasks involved in fulfilling orders.

You must also specify whether you want the order to use an orchestration or not. Oracle recommends using orchestration for most solutions. Use non-orchestration processes only when the order management requirements are simple, well understood, and relatively static.

You can model a single target order type that can process any type of incoming sales order, or you can model multiple order types based on product domain groupings. For example, you can create one order type for broadband, another for mobile, a third for cable, and so on. Using multiple order types, however, makes it difficult to bundle services and is also costly to maintain. Oracle recommends using one standard order type that accepts all incoming orders, and using other order types for only very specific uses, such as a fallout management order type that can extract information about failed orders.

If you create multiple order types, you also need to create corresponding order recognition rules that match incoming orders to the target order. Consider the following approaches when modeling order recognition rules:

- If you have different order source systems, each having its own order format, you can create multiple order recognition rules that point to the same target order. The order recognition rules transform the incoming order data into the target order data format.
- If you have multiple target orders based on domain groupings, you must create a separate order recognition rule for each target order type.
- If you have one OSM instance operating in more than one role (for example, if the same system is operating in both the SOM and the TOM role) you need to configure an order recognition rule that points to a corresponding target order for each role.
- If you have one order source system and one OSM instance operating in only one role, then you need only one order recognition rule that points to one target order.

You must determine what corresponding order recognition rules you need to model in OSM to recognize, validate, prioritize, and transform order data from sales orders into a matching OSM target order. You must map incoming order data to the data defined in the creation task of the target order.

See the following sections for more information:

- [Modeling Orders and Permissions](#)
- [Modeling Order Life-Cycle Policies](#)
- [Modeling Order Recognition](#)

COM Data Modeling Considerations

Answer the following questions to determine what corresponding order data you need to model on the sales order and the target order in OSM COM:

- What data is required to complete an order?
- What data do the tasks require? For example, if service provisioning requires the customer's location, then the order needs to include the customer's location.
- What data does the customer account require?
- What data is not required on the customer order, but is required by the service order that is derived from the customer order?
- What data do the tasks require when the order is created?
- Does the incoming order include all of the data needed for the order? If not, you can use data providers in your tasks to get the data from some other source.
- Which data on the order contain order item information that represents the offers, bundles, and products on the order?
- What order item parameters represent the name of the order item?

- If the order items are hierarchical, what data elements specify the parent and child relationship between each order item?
- If the order items must be delivered at different times and dates, what data element contains the requested delivery date?
- What data element specifies the action that must be performed on the order item during the fulfillment process (such as add, change, and so on)?
- What data element specifies the overall action of the order, such as deliver, cancel or technical service qualification (TSQ)?
- What data is required on the sales order for the billing system?

See the following sections for more information:

- [Modeling Order Data](#)
- [Modeling Orders and Permissions](#)
- [Modeling Behaviors](#)

Modeling COM Orchestration Order Items and Binding Conceptual Model Parameters

You must determine what order data to model as order items (see "[COM Data Modeling Considerations](#)") and what product specification the order items belong. You typically map order items to product specification by defining an order item property that OSM then uses to map the order item to specific fulfillment patterns, and so on.

You can also use Design Studio order item parameter bindings to map conceptual model entities their parameters to OSM order item specifications. This enables you to have strongly typed parameters that you do not need to model within order item specifications. You map conceptual model entities and their parameters to OSM order item specifications mainly to validate incoming order items and their parameters and/or transform the order items and their data from one type to another (for example, from products to CFSs).

In the OSM COM context, you map conceptual model products and data parameters to the COM order item specification for the incoming sales order and accompanying products, bundles, and offers. For example, [Figure 1-1](#) defines the broadband service offer and several child product offerings such as broadband, bandwidth, firewall, email, and so on. OSM validates all order items against the corresponding conceptual model entities and their parameters. If you have configured the order transformation manager, OSM also transforms the products into CFSs (see "[Modeling COM Order Transformation Manager](#)" for more information).

If you do not need order item parameter bindings for validation or transformation, you must model parameters in the order item specification. You can designate an order item property to contain these parameters (typically name-value pairs) and designate the structure as XML Type in the Order editor Order Template tab, Properties tab, Order Data subtab for the selected data element. For example, order items at the COM level relating to billing information, such as promotional offers or recurrent charging information, may not need validation because the sales catalog is separately synchronized between the CRM and billing systems. In this scenario, OSM is only required send the order items on the sales order directly to the billing system. However, all other order items that must go to OSM in the SOM role would typically require both validation and transformation using the conceptual model.

Validating data against the conceptual model is important because this ensures the data is consistent across the entire OSS and BSS solution (OSM in COM, SOM, and TOM roles, the activation system, the SRM system, and so on).

See "[Modeling Orchestration Plans](#)" for more information.

Modeling COM Orchestration Order Item Decomposition

You need to decide what orchestration stages OSM should evaluate when decomposing order items into order components. These order components typically designate functions, systems, and granularity options.

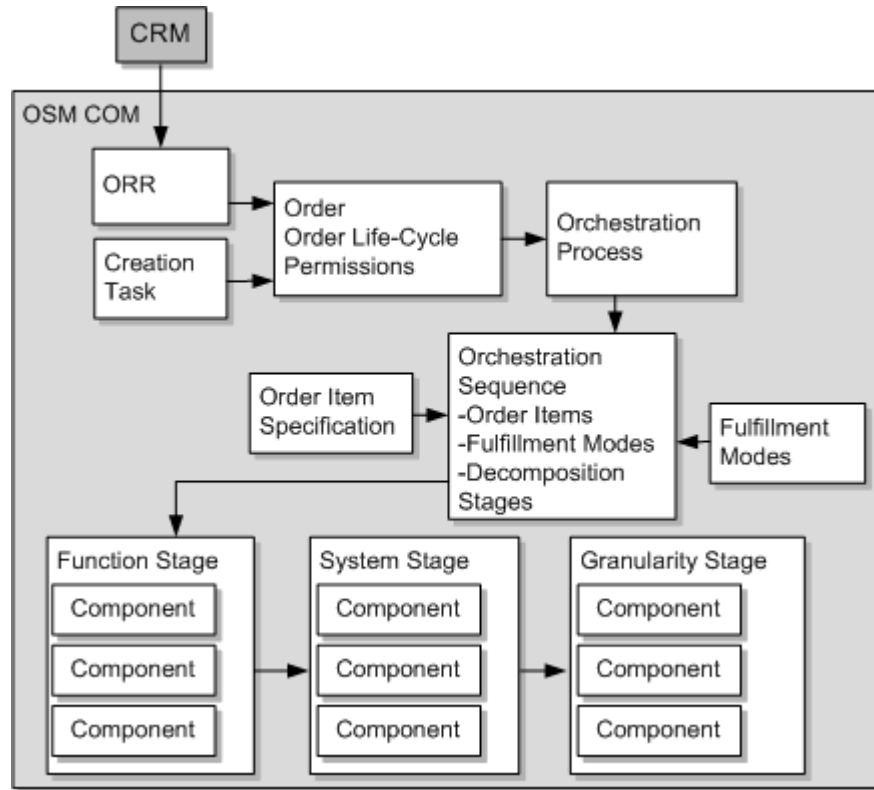
Based on the previous sections, you should already have some knowledge of what the order you are creating contains. Use that knowledge to answer the following questions:

- What systems does OSM communicate with?
- What order items does each system require?
- What are the business functions that each system must perform on the order items?
- How should OSM deliver data for the functions to process? Does the function operate on the whole order, or does the function require a separate interaction per order item bundle or per order item?

Based on the answers you provide to these questions, you can begin to model order item decomposition stages.

[Figure 1-2](#) shows a sample order capture and orchestration process. The process captures orders with an order recognition rule that maps the order and order data to a target order and creation task data elements. This order then begins an orchestration process that triggers an orchestration sequence. The sequence specifies which order node contains the order items, which order parameter defines the fulfillment mode for the order, and the stages in which the order items can be evaluated. The orchestration process determines which order components the order items should decompose to. In [Figure 1-2](#), OSM sequentially evaluates:

- A stage that decomposes order items into order components that define functions.
- A stage that decomposes order items into order components that define systems.
- A stage that decomposes order items into order components that define granularity.

Figure 1-2 Sample Order Capture and Orchestration Process

You must determine what high-level tasks are involved in fulfilling customer orders at the OSM COM level. You can define orchestration order components that correspond to the functions performed by these tasks that you can then add to the function stage (see [Figure 1-2](#)). See "[About the OSM Solution Modeling Process](#)" for a sample list of OSM COM related tasks that can be modeled as function order components.

You must determine what kinds of BSS fulfillment systems you have at the OSM COM level. You can define orchestration order components that correspond to these systems. You can add these system order components to the system stage. As illustrated in [Figure 1-1](#), the systems that normally interact with OSM COM include

- One or more billing systems: The billing systems manage the initial and recurring charges applied to the order.
- One or more OSM SOM systems: The OSM SOM system at the OSS level interacts with inventory systems. The inventory system designs and assigns services with their corresponding network resources.

OSM COM may need to communicate with different systems based on the location where the service is requested for.

You must also determine what kind of order granularity you need when fulfilling each function order components. Does the orchestration plan need to generate a separate function for each bundle destined for a particular system? Or, can the whole order be sent as one function to one system? You can define orchestration order components that correspond to the different levels of granularity you want to define. You can then add the order component to the granularity stage (see [Figure 1-2](#)).

For example, a customer might request an offer that includes the following order items:

```
BroadBand Service (Offer)
  BroadBand (Bundle)
    Promotion (Product)
    Bandwidth (Product)
    Router (Product)
    Firewall (Product)
  Email (Product)
```

The broadband service offer is parent to the broadband bundle order item and an email product order item. The broadband bundle is itself parent to promotion, bandwidth, router, and firewall product order items. It may be that you want the billing related functions to run separately for each child order item of the broadband service offer (the broadband bundle order item with all its children order items and the email product order item). Or you may want each billing function to run separately for each product order item in the order (email, promotion, router, bandwidth, and firewall). Finally, you could also send the entire offer with all bundles and products contained within it.

For each of these options, you need to create an order component that OSM can use to decompose the order items to. The order components can represent whole order granularity, bundle granularity, or product order item granularity, and so on. At this point, you are only defining the order components that OSM can use to decompose order items to. You configure the actual decomposition behaviors and conditions with other Design Studio orchestration entities such as fulfillment patterns and orchestration decomposition rules.

See "[Modeling Orchestration Plans](#)" for more information.

Modeling COM Orchestration Fulfillment Patterns and Fulfillment Modes

You must determine what fulfillment patterns each order item should decompose to and what action you want to specify on the overall order. You model the actions (such as deliver, technical service qualification, or cancel) as fulfillment modes. Each fulfillment pattern can have more than one fulfillment mode. When an order item decomposes to a fulfillment pattern, the fulfillment pattern generates a different orchestration plan based on the action defined on the order. The action corresponds to the fulfillment mode associated to the fulfillment pattern.

OSM fulfillment patterns define (among other things) the first stage of order item decomposition. In fulfillment patterns, you can specify order item decomposition conditions for whether a specific order item decomposes to an order component. Typically fulfillment patterns contain all order components that specify functions.

In a conceptual model project, you map conceptual model product specifications which define offers, bundles, and products to conceptual model fulfillment patterns. OSM fulfillment patterns realize these conceptual model fulfillment patterns. When you build an OSM cartridge, OSM generates a sample XML file. The sample XML file contains the product-to-fulfillment pattern mappings. You can reference those mappings using an order item property from the OSM order item specification. The order item property defines XQuery logic that determines how each order item decomposes to a fulfillment pattern.

For example, an order might contain the following order items:

- Five decompose to a broadband fulfillment pattern.
- Four decompose to a VoIP fulfillment pattern.
- One decomposes to an Email fulfillment pattern.

At run-time, OSM evaluates the function decomposition stage first which contains the function order component. This evaluation determines whether order items decompose to each function order component based on the conditions (if any exist) specified in the fulfillment pattern.

OSM next evaluates how order items decompose to the system and granularity order components in the system and granularity stages using on orchestration decomposition rules. These decomposition rules define how order items decompose from function order components to system order component, and from function order components to granularity order components. All order decomposition rules relating to system order components in the system decomposition stage are evaluated. Then those order decomposition rules relating to the granularity order components in the granularity stage are evaluated.

After OSM evaluates each stage for each order item in the fulfillment patterns they are associated with, OSM generates run-time order components. These run time order components are the sum of the order components that each order item decomposes to. For example, order item A can be decomposed to functionA-systemA-granularityA. This sequence of order components constitutes a single run-time order component.

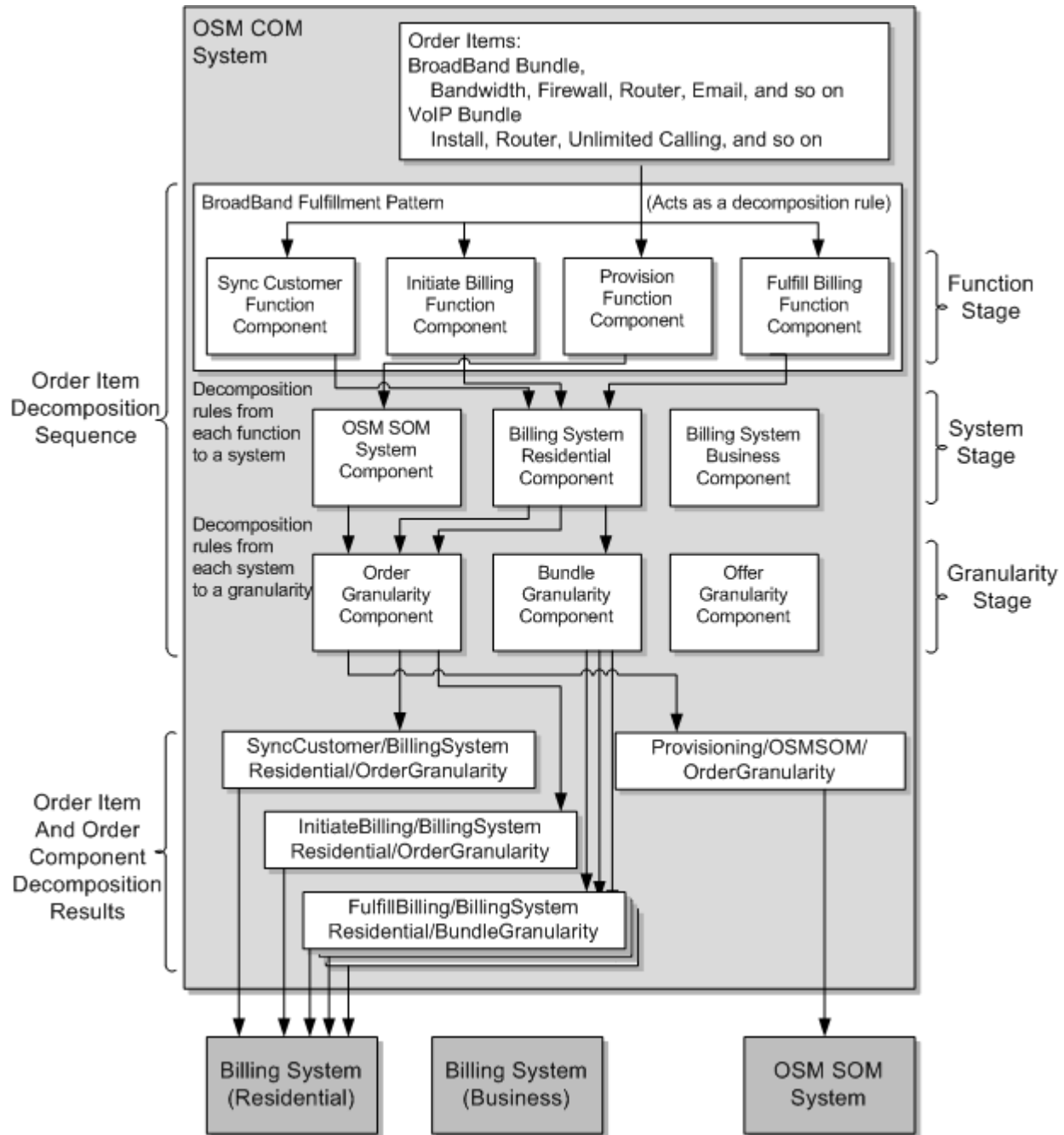
However, consider a scenario where the order items in an order decompose to more than one fulfillment pattern. If fulfillment pattern A generates the same run-time order component as fulfillment pattern B, then OSM generates only one run-time order component. This run-time order component processes the order items from both fulfillment pattern A and B.

some decomposition sequences of the three different fulfillment patterns end up being identical, then only one run-time order component is created that runs a process for all order items.

[Figure 1-3](#) shows an order item decomposition run-time sequence with the BroadBand order item bundle.

The first stage of order item decomposition uses the broadband fulfillment pattern which defines decomposition rules for each function order component. The second stage uses decomposition rules from each function to each system order component. The third stage uses decomposition rules from each system to each granularity order component. OSM generates the resulting run-time order components based on this sequence. Each unique decomposition flow generates a new executable order component.

Figure 1-3 Sample Run-Time Order Item Decomposition Sequence



Those generated in [Figure 1-3](#) are:

- SyncCustomer/BillingSystemResidential/OrderGranularity
- InitiateBilling/BillingSystemResidential/OrderGranularity
- Provisioning/OSMSOMSystem/OrderGranularity
- FulfillBilling/BillingSystemResidential/BundleGranularity

In this scenario, the decomposition rules to the billing system business component and offer granularity component rejects all order items. The rejection is based on a condition that

specifies that only order items from business customers can be included. All order billing related components are directed to the billing system residential as opposed to the billing system business order component.

In addition, the VoIP bundle order items and all its child order items decompose to another fulfillment pattern not represented in [Figure 1-3](#). It is important to note that an orchestration plan that contains multiple fulfillment patterns may generate identical run-time order components if some of the functions, systems, and granularity component decomposition flows are the same. For example, the broadband internet and VoIP fulfillment patterns may each specify the same SyncCustomer and InitiateBilling functions with the same systems and granularity components. In such a case, OSM generates only one run-time component to which the order items from each fulfillment pattern decomposes to. However, it may be that separate provisioning and billing components are required for each fulfillment pattern. For example, you may want a separate provisioning component for the broadband order items. When the provisioning component completes, the billing component runs. The billing system then begins charging for broadband service immediately. The VoIP related order items might decompose to separate provisioning and billing order components that only start when the provisioning component for broadband completes. This decomposition pattern may be appropriate based on the fact that the VoIP related order items functionally depend on the broadband order items and also take much longer to fulfill than the broadband order items. In this scenario, the CSP does not need to wait for the VoIP order items to fulfill before beginning to charge for the broadband service.

Similar decomposition scenarios may be important when considering the date when the customer wants a particular service fulfilled. For example, a customer may request an IPTV and VoIP bundle that are normally fulfilled within the same provisioning and billing functions. But because the customer requests a start date for the IPTV service that is much later than the one specified for the VoIP service, then these two order item bundles must decompose to separate provisioning and billing order components.

See the following sections for more information:

- [Modeling Orchestration Plans](#)
- [Modeling Order Scheduling](#)

Modeling COM Order Transformation Manager

You should determine whether you want to use the order transformation manager (OTM) with the calculate service order (CSO) conceptual model provider function. OTM with CSO transforms products, bundles, and offer order items and actions into CFS order items and actions. The run-time order component creates and sends a service order to OSM SOM or some other provisioning system. The OTM functionality provides a way to decouple the commercial layer from the service layer.

[Figure 1-4](#) shows a sample run-time order transformation with the design-time conceptual model entity associations. This example has the following run-time flow that incorporates design time data from conceptual model entities:

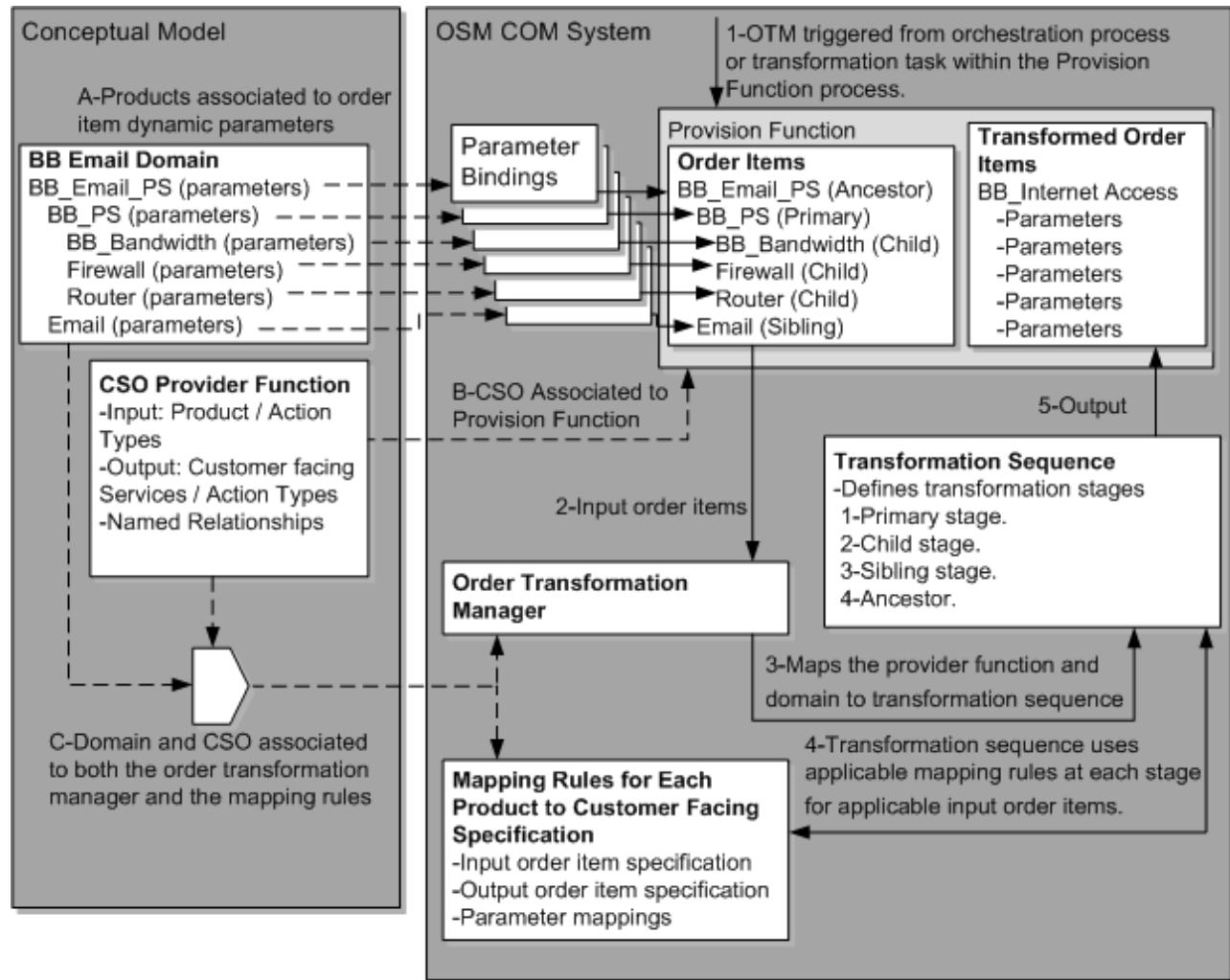
1. OTM triggers either when the orchestration process begins to design an orchestration plan or from within the process associated with the provision function in a transformation task. OTM can process data elements for product entities in the conceptual model. OTM can process data elements only if order parameter bindings are created. The order parameter bindings are between product data in the conceptual model and an order item property in the order item specification. The order item property must be designated as a dynamic parameter that defines a data structure.

OTM can only process conceptual model product entity data elements on the order if order parameter bindings (**conceptual model association A**) have been created between

conceptual model product data and an order item specification order item property designated as a data structure definition dynamic parameter.

2. OSM sends the input product order items to OTM. In order for OSM to do this, the conceptual model CSO provider function must be associated with the provision function order component at design-time (**conceptual model association B**).
3. OTM sends the order data to a transformation sequence. In order for OTM to map to the appropriate transformation sequence, OTM must also be associated with the conceptual model CSO provider function and a conceptual model domain (**conceptual model association C**). The domain is a repository of all the conceptual model products possible for the order item. In this case, the domain is the BB Email Domain for broadband and Email products. Other domains could be VoIP, mobile, cable and so on.
4. The transformation sequence goes through a series of stages. The stages use mapping rules to map the input order item data to output order item data. The stages also define whether the data is primary (which creates a new order item) or auxiliary (which augments the data on a new order item). The mapping rules must also be associated with the same CSO provider function and domain as OTM (**conceptual model association C**). For example:
 - **Primary Stage:** Transforms product order items and actions into transformed order items and actions. At the primary stage, product order items create new transformed customer service-facing order items. In this example, the primary product specification is BB_PS which maps to the BB Internet Access customer-facing service specification.
 - **Child Stage:** Transforms all child order items and actions and any of their child order items and actions, and so on, into data that augments the order items created in the primary stage. In this example, the child order items of BB_PS are BB_Bandwidth, Firewall, and Router.
 - **Sibling Stage:** Transforms all sibling order items and actions into data that augments the order items created in the primary stage. In this example, a sibling order item of BB_PS is Email.
 - **Ancestor Stage:** Transforms all ancestor order items and actions and any of their ancestor order items and actions, and so on, into data that augments the order items created in the primary stage. In this example, the ancestor order item of BB_PS is BB_Email_PS.

Figure 1-4 Sample Run-Time Order Transformation with Conceptual Model Associations



5. The transformation sequence includes the resulting order items into the provision function order component. The provisioning function order component runs a process that creates and sends a service order to OSM SOM. The service order includes the transformed service order items. The product order items are not required in the service order because all necessary data contained in the product order items are now consolidated into the service order item.

When the input sales order lines are transformed to CFS:

- If the input sales order lines are not mapped to CFS as defined in modeling, CFS pay load will not be generated.
- If the input sales order line is missing any of the required parameters for mapping, the missing mandatory parameter will be reported as error and no CFS pay load will be generated.

In both the cases, the provisioning function process fails and SOM order is not created. A revision order should be submitted to correct the sales order line data, so that OTM maps properly, and then CFS payload is generated and the SOM order is created.

See the following sections for more information:

- [Modeling the Order Transformation Manager](#)
- [Modeling OTM With Calculate Service Order](#)

- [Modeling OTM Without Calculate Service Order](#)

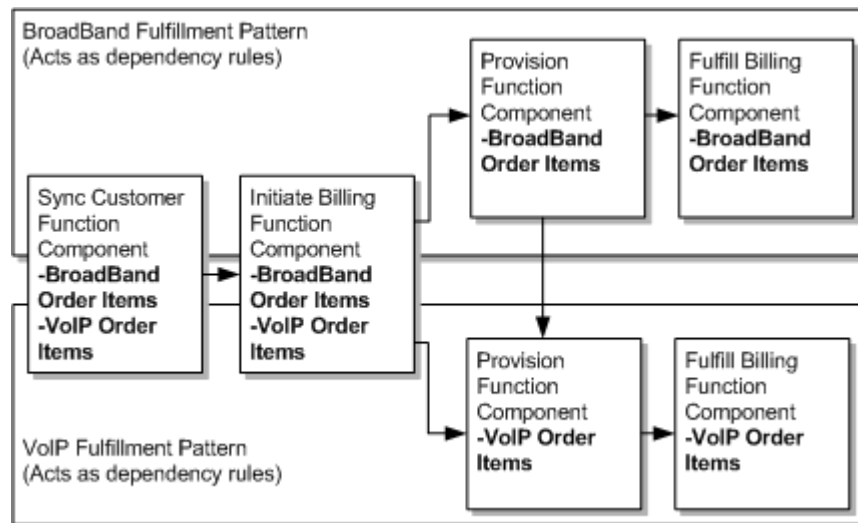
Modeling COM Orchestration Dependencies

You need to determine what dependencies exist between executable order components. These dependencies are called orchestration dependencies. You typically define all orchestration dependencies using fulfillment patterns for function order components. However, you can also specify dependencies between other order components using orchestration dependency rules. In addition, you can specify when order components can start based on dates provided by customers for when they want a service to begin.

For example, the tasks specified in "[About the OSM Solution Modeling Process](#)" for OSM COM may need to be fulfilled in the following order (see [Figure 1-5](#)):

1. The Sync Customer and Initiate Billing function that are shared between both fulfillment patterns occur sequentially. Order items from both broadband and VoIP fulfillment patterns decomposed into identical order components. All order items in the Sync Customer function must complete before the Initiate Billing Function can continue.
2. The Initiate Billing function must complete before the Provision Function can start processing the broadband order items and the second Provision Function can start processing the VoIP order items. In addition, the second provision function must wait until the first provision function completes because VoIP is functionally dependent on broadband and because the VoIP service takes longer to fulfill than the broadband service.
3. The Fulfill Billing function for the broadband order items starts after the Provision function for the broadband order items completes.
4. The Fulfill Billing function for the VoIP order items starts after the Provision function for the VoIP order items completes.

Figure 1-5 Example Dependency Between Fulfillment Pattern Order Items



Having separate Provision functions means that OSM COM sends multiple service orders to OSM SOM. Other factors can also impact order component creation and dependencies such as the requested delivery date for each service.

See the following sections for more information:

- [Modeling Orchestration Plans](#)
- [Modeling Order Scheduling](#)

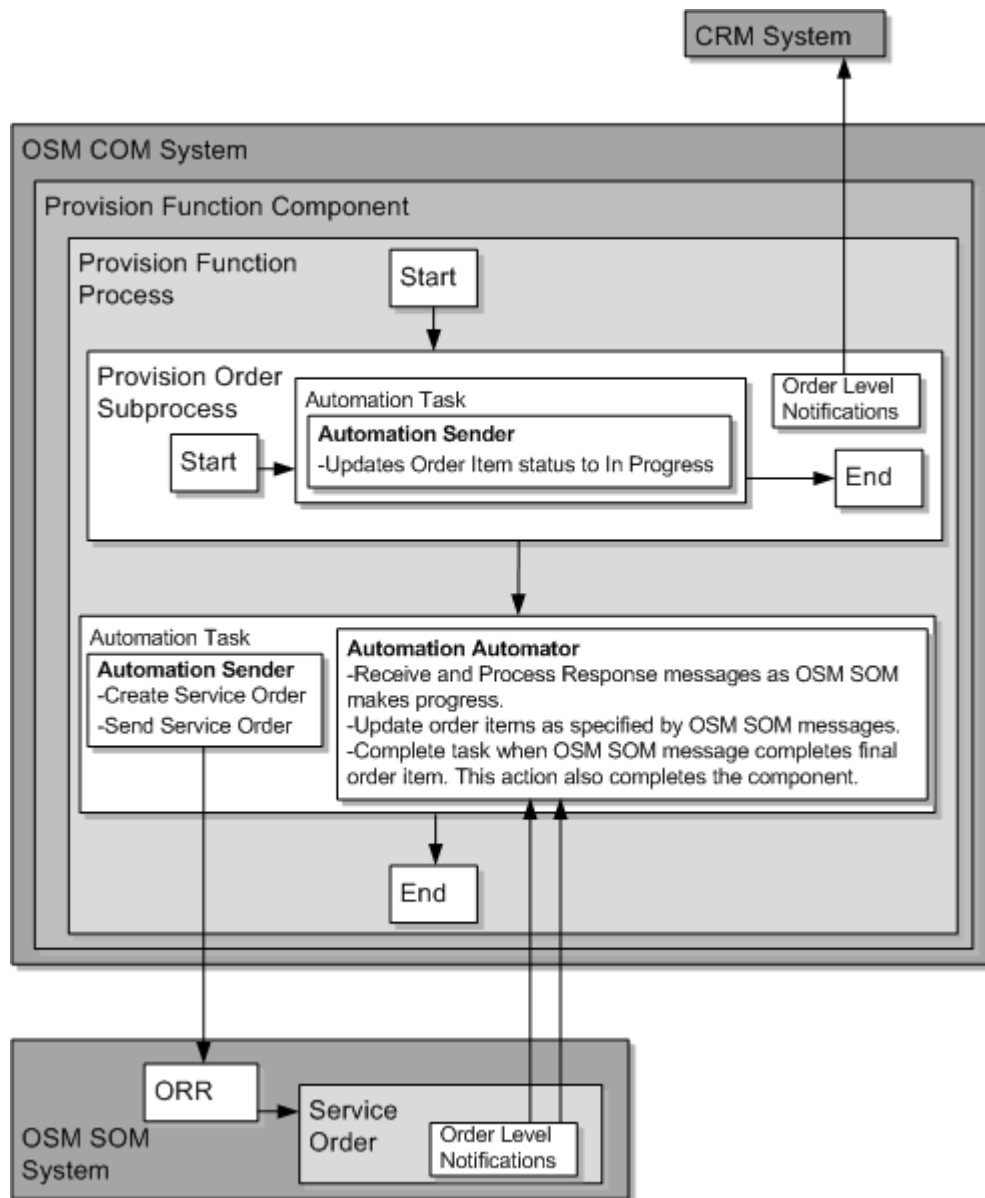
Modeling COM Processes and Tasks

What process flow of sequential or parallel manual and automated tasks are required when interacting with target systems for each functional component.

For example, each function may have a few automated tasks that perform different functions, such as sending messages to an external system, receiving back and processing a response from external systems, or manipulating data received from previously completed tasks. There may also be manual tasks where an operator is required to input data directly into the Order Management Task web client. The manual tasks may also make use of behaviors that effect how the data is organized, displayed, or retrieved in the OSM Task web client and in the Order Management web client.

[Figure 1-6](#) shows an OSM process that includes a subprocess with an automated task and automation plug-in sender. An order level notification updates the status of order items then notifies the CRM of the status change. The automated task then transitions to another task that sends a service order to the OSM SOM system. The server order includes all the order items required to design and assign the products and services that the customer has requested.

Figure 1-6 Example Task Processing



See the following sections for more information:

- [Modeling Processes and Tasks](#)
- [Modeling Behaviors](#)
- [Modeling Data for Tasks](#)

Modeling COM Fallout Scenarios

You must determine:

- What fallout management scenarios can be anticipated.
- How fallout scenarios can be detected.
- How relevant parties or systems can be notified of problems.

- What recovery processes can be implemented.

For example, you should anticipate fallout around network connectivity from time to time. External systems may fail to return responses or fail to accept messages. In such cases you can configure automated tasks to transition to fallout an execution mode. Failed messages generated by automation plug-ins can revert to an error queue. You can also configure jeopardy notifications when messages are taking too long to return. The jeopardy notification can generate warning messages to fallout personnel so they can manual investigate the problem on the external fulfillment system or in OSM.

See the following sections for more information:

- [Modeling Fallout](#)
- [Modeling Behaviors](#)
- [Modeling Processes and Tasks](#)
- [Modeling Jeopardy and Notifications](#)

Modeling COM Fulfillment States

You must determine what kind of order and order item fulfillment states you need to configure. Fulfillment states track the overall status of an order and each order item based on status messages received from external systems. You also need to consider what notification messages you want to send to interested parties or systems as the order progresses.

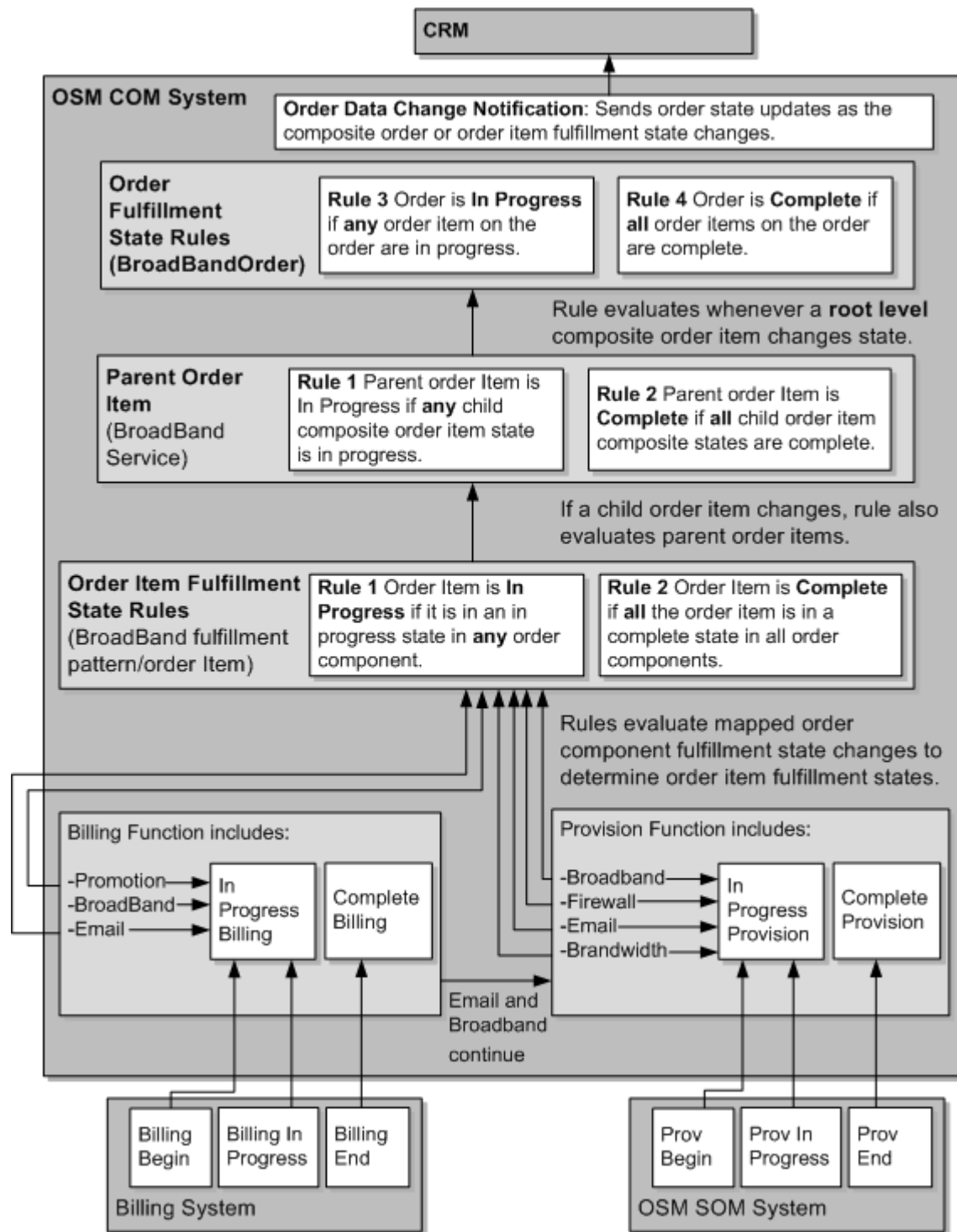
For example, [Figure 1-7](#) shows external fulfillment state change messages from the billing system and the OSM SOM system returning to the OSM COM system.

The hierarchical structure of the order defined in the order item specification is as follows:

```
BroadBandOrder (target order)
  BroadBand Service (Bundle)
    Promotion (Product)
    BroadBand (Product)
    Bandwidth (Product)
    Firewall (Product)
    Email (Product)
```

1. The billing function is a generic order component, but could represent any of the billing functions listed in "[About the OSM Solution Modeling Process](#)" for OSM COM. After the billing function sends a message to the billing system, the billing system sends three status messages back to OSM COM as the billing system processes the **Promotion**, **Broadband**, and **Email** order item. Two of the external fulfillment state messages, **billing begin** and **billing in progress**, map to the OSM COM **in progress billing** fulfillment state. The billing system sends these messages to confirm that it has received the message from OSM with the order items and then to confirm that it has begun to process the messages. When the billing system finished processing all order items, it sends the **billing end** external fulfillment state that maps to the **complete billing** fulfillment state.
2. The provision function has a similar exchange of messages with the OSM SOM system for the **broadband**, **bandwidth**, **firewall**, and **email** order items.

Figure 1-7 Example Fulfillment States



- When the order items first begin processing in the billing function component, **Rule 1** evaluates to true when the billing system sends the **billing begin** and **billing in progress** external fulfillment state messages that map to the **In Progress Billing** fulfillment state for the billing order component. This causes the order items included in the billing function component to have a composite fulfillment state of in progress.
- When the billing system completes and sends the **billing end** message which maps to the **Complete Billing** fulfillment state for the function component, the **Broadband** and **Email**

order items continue processing in the provision function order component while the **Promotion** order component does not. **Promotion** did not decompose to the provision function and is therefore fully complete. **Rule 1** still evaluates to true for **BroadBand** and **Email**, but **Rule 1** evaluates to false for **Promotion**. **Rule 2** evaluates to true for **Promotion**, causing its order item composite fulfillment state to move to **Complete**.

5. When **Rule 1** and **Rule 2** make any change to the state in an order item, they also evaluate the parent order item, which is the **BroadBand Service** bundle order item. Although **Promotion** is now complete, the **BroadBand Service** remains in the **In Progress** state until the other child order items complete.
6. When the remaining order items complete in the provision function, **Rule 1** evaluates to false and **Rule 2** evaluates to true for the order component. **Rule 2** then changes the composite order item fulfillment states for all remaining order items to complete. Because of this change, the parent **BroadBand Service** order item also changes to complete because all its child order items are now complete.
7. Because of this change in **BroadBand Service**, which is a root level order item, the order fulfillment state **Rule 3** evaluates to false because there are no longer any in progress order items at the root level. **Rule 4** evaluates to true because all order items at the root level are now complete.
8. An order data change notification triggers whenever an order or order item composite fulfillment state changes that sends messages to the CRM to report the changes.

[Figure 1-7](#) shows only in progress and complete billing states at both the order item and the order level. Many other external order fulfillment states, order item fulfillment states, and order states are possible, such as failure states, cancelation states, and so on. Each of these would also have a corresponding order item fulfillment state composition rule and order fulfillment state composition rule.

See the following sections for more information:

- [Modeling Fulfillment States and Processing States](#)
- [Modeling Jeopardy and Notifications](#)

Modeling COM Processing States

Processing states are similar to fulfillment states in that you can configure automated tasks to trigger order component order item processing states based on messages received from external fulfillment systems. The main difference is that the order component order item processing states come from a predefined list that OSM provides. OSM automatically aggregates order component order item processing states across all order components processing the order item into a single order item processing state that is visible in the Order Management web client. OSM also propagates child order item processing states to parent order items. The only work necessary to model processing states is to map incoming status message from external fulfillment systems to order component order item processing states. You can even map the same message to both order component order item processing states and to external fulfillment states.

For example, the following shows how the Email product is successfully processed in two order components, but the third order components returns a failure order component order item processing state, causing the Email product to display the Partially Failed order item processing state, which in turn caused the parent and grandparent order items (Brilliant Broadband and Broadband Service) to also display the Partially Failed order item processing state.

```
Brilliant Broadband (Offer) ----- Partially Failed
    BroadBand Service (Bundle) -- Partially Failed
```

```

Promotion (Product) ----- Completed
BroadBand (Product)----- Completed
Bandwidth (Product)----- Completed
Firewall (Product) ----- Completed
Email (Product) ----- Partially Failed  ---- OC0I1 - Completed
                                           ---- OC0I2 - Completed
                                           ---- OC0I3 - Failed

```

The Order Management web client tracks Normal, Warning, and Failure counts of order item processing states.

See the following sections for more information:

- [Modeling Fulfillment States and Processing States](#)
- [Modeling Jeopardy and Notifications](#)

Modeling Change Order Management for COM

You need to determine if you want to enable change order management, and if you do, what change order management scenarios you want to support.

Do you need a point of no return where the order cannot be changed? For example, you can configure a point of no return that is tied to the provisioning function that generates a service order to be enforced whenever that order component receives a fulfillment state update of Complete (assuming you have configured the order component with such a fulfillment state update). You can also tie a point of no return directly to the any order life-cycle policy transition to the Amending state, such as the Submit Amendment transition from the In Progress state to the Amending state.

If a revision order arrives, what behavior do you want each tasks to exhibit? Do you want the task to undo, redo, or undo then redo? For example, do you want to configure the automated task responsible for sending the service order to OSM in the SOM role to trigger an automation plug-in that sends a revision order to OSM SOM that undoes the previously sent service order whether it is complete or still in progress? Or do you want the task to redo the previously sent service order as a revision order and allow SOM to perform change order management functions?

See the following sections for more information:

- [Modeling Changes to Orders](#)
- [Modeling Processes and Tasks](#)

Cartridge Management Considerations for COM

What kinds of cartridge management scenarios you want to plan for in advance, such as the impact of upgrading cartridge functionality, how such upgrades impact run-time orders, how best to structure cartridges to minimize the impact of such changes, and so on.

See "[Managing OSM Solution Cartridges](#)" for more information.

Planning OSM SOM Solution Requirements

This section describes OSM modeling entities and functions involved in the OSM SOM role. It includes examples intended to guide you in understanding how you can use these entities and functions in your OSM SOM solution. This information can help you plan your implementation efforts by exemplifying at a high level the full scope of work involved in modeling a typical COM solution. Follow the links in each section for specific details about the functions described.

OSM in the SOM role receives CFSs from OSM in the COM role within a service order. OSM SOM then sends this CFS information to an SRM system that designs the service then assigns it to resources. The SRM system uses the RFS and resource information defined in the conceptual model to perform this design and assign task. The SRM also calculates the actions required to fulfill the services. OSM SOM requests these actions from the SRM system then sends the actions to OSM in the TOM role.

Modeling SOM Order and Order Recognition Requirements

You must determine what kind of orders you need to model in OSM SOM, what order life-cycle states and state transitions the orders have, and who has permissions to do various tasks in fulfilling the order.

At this point, it is important to understand that the SOM order is a child of the COM parent order and must report back to the COM order component that generated the service order. In the example provided in "[Modeling COM Orchestration Fulfillment Patterns and Fulfillment Modes](#)", the order component that generated the service order to SOM is a task that is part of a process triggered by the Provisioning/OSMSOMSystem/OrderGranularity run-time order component. Any notification from OSM SOM relating to the service order would return back to this task.

Depending on how you have modeled your COM solution, OSM SOM may receive more than one service order. For example, you may want to configure OSM COM to send a separate service order to fulfill the broadband internet CFS. When that service order completes, OSM COM may send a second service order with a VoIP CFS.

You may want to create separate target orders for each CFS or one generic target order that receives all service orders. In the SOM context, OSM does not generally need separate target orders because most of the work is accomplished within the SRM system that OSM SOM communicates with and any dependencies between CFSs are enforced in OSM COM.

Likewise, if you have only one target order at the SOM level, then you need only one order recognition rule that maps incoming service orders to this target order.

See the following sections for more information:

- [Modeling Orders and Permissions](#)
- [Modeling Order Life-Cycle Policies](#)
- [Modeling Order Recognition](#)

SOM Data Modeling Considerations

Answer the following questions to determine what corresponding order data you need to model on the OSM order to capture the incoming order data:

- What data is required to complete an order?
- What data do the tasks require? For example, what data is required on each interaction with the SRM system? What format to OSM TOM require for the technical order that OSM SOM sends?
- What data is not required on the service order, but is required by the technical order that is derived from the customer order?
- Does the incoming order include all of the data needed for the order? If not, you can use data providers in your tasks to get the data from some other source. Typically SOM is only responsible for passing on CFSs created in COM to the SRM system. However, it is possible that other data may be required that does not come from the SRM system.

- Which nodes on the service order contain order item information that represents the CFSs on the order?
- What order item parameters represent the name of the CFS order item?
- If the order items are hierarchical and what data elements specify the parent child relationship between each order item? Typically service orders do not require a hierarchy.
- If the order items must be delivered at different times and dates, what data element contains the requested delivery date?
- What data element specifies the action that must be performed on the CFS order item during the fulfillment process? For example, add, change, delete, move, and so on.
- What data element specifies the overall purpose of the order, such as deliver, cancel or technical service qualification (TSQ)?

See the following sections for more information:

- [Modeling Order Data](#)
- [Modeling Orders and Permissions](#)
- [Modeling Behaviors](#)

Modeling SOM Orchestration Order Items and Bindings Conceptual Model Parameters

You must determine what nodes in the incoming order you want to designate as order items containing CFSs. What data in the order items you want to use for specific orchestration functions, such as the actions you want OSM to perform on those order items, the requested delivery date when the order item actions need to occur, what CFS the order item represents that OSM then uses to map the order item to specific fulfillment patterns, and so on.

You can also use order item parameter bindings to bind conceptual model CFSs and the parameters defined for them to a OSM SOM order item specifications. This allows you to have strongly typed parameters that you don't need to model within order item specifications. For example, [Figure 1-1](#) defines the Broadband Internet Access CFS and the Email CFS. You can use order item parameter bindings to map these conceptual model product entities to order item specifications by configuring an order item for the order item recognition and an order item parameter as a dynamic parameter where the parameters are stored.

Order item parameter bindings in OSM SOM are important for validating the incoming CFSs generated from OSM COM, however OTM is not required in the OSM SOM role because the SRM system is typically responsible for transforming CFSs into RFSS, resources, and actions. OSM SOM sends the CFSs to the SRM system and receives back the actions on the resources. OSM SOM does not need order item parameter bindings on these actions because OSM TOM is responsible for validating these actions.

See "[Modeling Orchestration Plans](#)" for more information.

Modeling SOM Orchestration Order Item Decomposition

You need to decide what orchestration stages you want OSM to evaluate when decomposing CFS order items into order components. These order components typically designate functions, systems, and granularity.

Based on the previous sections, you should already have some knowledge of what the service order you are creating contains. Using this knowledge, you can provide answers to the following questions:

- What are the systems that OSM must communicate with? For example, OSM SOM typically communicates with an SRM system and OSM in the TOM role.
- What order items do each system require? For example, the SRM system requires CFS information and OSM in the TOM role requires technical actions.
- What are the business functions that each system must perform on these order items? For example, sending the CFSs to the SRM system, requesting the actions from the SRM system, and building the technical order that contains the actions for the OSM TOM system.
- How should OSM deliver this data for the functions to process? Does the function operate on the whole order, or does the function require a separate interaction per order item bundle on the order, or per order item?

Based on the answers you provide to these questions, can you begin to model order item decomposition stages.

[Figure 1-1](#) defines the Broadband Internet Access CFS and the Email CFS. You can decompose these order items into function, system, and granularity order components in the same way you do in the COM (see "[Modeling COM Orchestration Order Item Decomposition](#)").

You must determine what high level tasks are involved in fulfilling customer orders at the OSM SOM level. You can define orchestration order components that correspond to the functions performed by these tasks that you can then add to the function stage (see [Figure 1-2](#)). See "[About the OSM Solution Modeling Process](#)" for a sample list of OSM SOM related tasks that can be modeled as function order components.

You must determine what kinds of OSS fulfillment systems you have at the OSM SOM level. You can define orchestration order components that correspond to these systems that you can then add to the system stage. As illustrated in [Figure 1-1](#), the systems that normally interact with OSM SOM include one or more SRM systems that interact with inventory systems to design and assign services with their corresponding network resources.

You must also determine what kind of order granularity you need when fulfilling each function order components. At the SOM level, OSM typically passes on every CFS order items to the SRM system because CFSs are typically not hierarchically ordered. However, this all depends on how you model the overall solution.

For each of these options, you need to create an order component that OSM can use to decompose the order items to, such as whole order granularity, bundle granularity, or product order item granularity, and so on. At this point, you are only defining the order components that OSM can use to decompose order items to during order decomposition. You configure the actual decomposition behaviors and conditions with other Design Studio orchestration entities such as fulfillment patterns and orchestration decomposition rules.

See "[Modeling Orchestration Plans](#)" for more information.

Modeling SOM Orchestration Fulfillment Patterns and Fulfillment Modes

You must determine what fulfillment patterns each order item should decompose to and what action you want to specify on the overall order that you can model as fulfillment modes for the order, such as deliver, technical service qualification, or cancel.

In a conceptual model project, you map conceptual model CFSs to conceptual model fulfillment patterns. These conceptual model fulfillment patterns are realized by OSM SOM fulfillment patterns. When you build an OSM cartridge, OSM generates a sample XML file that contains all these CFS to fulfillment pattern mappings that you can reference using an OSM SOM order item specification order item property that defines XQuery logic that determines how each order item decomposes to what fulfillment pattern during run-time.

For example, OSM SOM may have a Broadband Internet Access CFS order item and the Email CFS order item that maps to a conceptual model fulfillment pattern that is realized by an OSM SOM fulfillment pattern that specifies a function to send these CFSs to the SRM system so that the SRM system can perform the design and assign functionality. You can also create another function that requests the actions that must be included in a technical order, and a third function that creates, sends the technical order to OSM TOM, and a fourth function that completes the service order.

At run-time, OSM evaluates the function, system, and granularity stages in a similar way to OSM in the COM role (see "[Modeling COM Orchestration Fulfillment Patterns and Fulfillment Modes](#)"). The function, systems, and granularity stages might generate the following run-time order components:

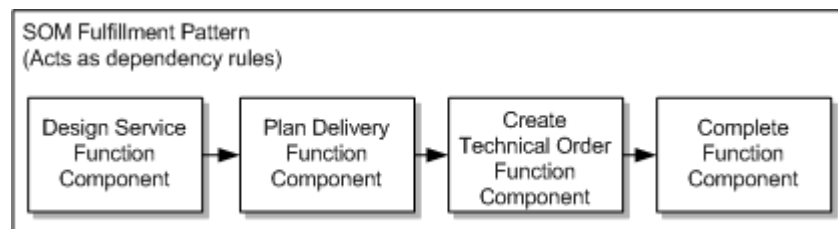
- DesigningServiceFunction/SRMsystem/OrderGranularity
- PlanDeliveryFunction/SRMsystem/OrderGranularity
- CreateTechnicalOrderFunction/OSMTOMsystem/OrderGranularity
- CompleteFunction/SRMsystem/OrderGranularity

Modeling SOM Orchestration Dependencies

You must determine what dependencies exist between executable order components. These dependencies are called orchestration dependencies. You typically define all orchestration dependencies using fulfillment patterns for function order components, but you can also specify dependencies between system order components using orchestration dependency rules. In addition, you can specify when order components can start based on dates provided by customers for when they want a service to begin.

For example, the tasks specified in "[About the OSM Solution Modeling Process](#)" for OSM SOM may require that each function operate in the sequence specified in [Figure 1-8](#).

Figure 1-8 Example Dependency Between Fulfillment Pattern Order Items



Because there is typically only one fulfillment pattern at in OSM SOM for fulfilling service orders, each component can run one after the other.

See the following sections for more information:

- [Modeling Orchestration Plans](#)
- [Modeling Order Scheduling](#)

Modeling SOM Processes and Tasks

You must determine what process flow of sequential or parallel manual and automated tasks are required when interacting with target systems for each functional component.

Tasks in OSM SOM typically involve sending the CFS order items required by the SRM system and receiving back the action from the SRM system. For example, the design service function would start a process that triggers an automated tasks with an automation plug-in sender instance that send builds and sends a message containing the CFS information to the SRM system in the format required by the SRM system API. When the SRM system completes, it sends a response back to OSM SOM that OSM SOM correlates back to the automated task to an automation plug-in automator that is waiting for a response message. The automation plug-in automator reviews the response message and determines that the SRM system completed its tasks successfully and then transitions the automated task to the completed state which also completes the DesigningServiceFunction/SRMsystem/OrderGranularity order component.

Each run-time order components with associated processes and tasks would perform similar exchanges.

See the following sections for more information:

- [Modeling Processes and Tasks](#)
- [Modeling Behaviors](#)
- [Modeling Data for Tasks](#)

Modeling SOM Fallout Scenarios

What fallout management scenarios can be anticipated, determine how they can be detected, how relevant parties or systems can be notified of the problem, and what recovery processes can be implemented.

For example, in addition to possible communication issues, you may anticipate the possibility that fallout may occur because OSM COM sends faulty or incomplete CFS information to OSM SOM, or the SRM system has somehow provided incorrect data to OSM SOM that may cause a fallout to occur in OSM TOM. You must carefully analyze when such issues can occur and develop fallout strategies to recover from such fallouts scenarios. In some cases, manual intervention may be required while in other cases, you may be able to model automatic fallout recovery capabilities.

See the following sections for more information:

- [Modeling Fallout](#)
- [Modeling Behaviors](#)
- [Modeling Processes and Tasks](#)
- [Modeling Jeopardy and Notifications](#)

Modeling SOM Fulfillment States

What kind of order and order item fulfillment states you need to configure to track the overall status of an order and each order item based on status messages received from external systems. You also need to consider what notification messages you want to send to interested parties or systems as the order progresses.

For example, you can map messages from the SRM system and the OSM TOM system returning as these systems process messages sent by various order components to external fulfillment state in the OSM SOM system. These external fulfillment states can represent the result of various interactions between OSM SOM and these systems on each CFS order item that OSM SOM then aggregates into order item and order-level fulfillment states based on order and order item fulfillment state composition rule sets.

See the following sections for more information:

- [Modeling Fulfillment States and Processing States](#)
- [Modeling Jeopardy and Notifications](#)

Modeling SOM Processing States

You can track the state of order items by mapping responses from the SRM system and OSM TOM to order item processing states. You must decide what messages correspond to which predefine order component order item processing state that OSM provides. OSM then aggregates these order component order item processing states for each order item into an overall order item processing state. You may decide to use warning and failure order item processing states to trigger jeopardy notifications from OSM SOM to OSM COM or from OSM SOM to a fallout personnel. In many cases, you can also use the same messages from external systems to trigger external fulfillment state changes.

See the following sections for more information:

- [Modeling Fulfillment States and Processing States](#)
- [Modeling Jeopardy and Notifications](#)

Modeling Change Order Management for SOM

You need to determine if you want to enable change order management, and if you do, what change order management scenarios you want OSM SOM to support.

You must consider change order management based on end-to-end scenarios that span OSM COM, SOM, and TOM. For example, if you enable OSM COM to send a revision order through to OSM SOM from the OSM COM provision function then you must decide what compensation OSM SOM must undertake to implement the changes in the revision order. For example, you might consider some of the following question:

- Is there a point of no return where you do not want OSM SOM to accept any new revision orders from the OSM COM provision function? For example, you may want to configure a point of no return that is associated with when the SRM system completes its design and assign functions based on an external fulfillment state change. This would effectively mean that OSM TOM should not receive revision orders from OSM SOM stemming from changes coming from OSM COM service orders. Or you may decide that there should not be any point of no return configured in OSM SOM.
- It may be that the SRM system that you are communicating with does not have the capability of accepting revisions to CFSs sent by the original interaction between OSM SOM and the SRM system, but can only accept cancelation requests. In which case OSM SOM must configure the automation task to completely undo the original request then redo it with using the new CFS information.
- It may be that you want to configure OSM TOM to accept revision orders, in which case, you can configure OSM SOM to redo the task that sends the technical order to OSM TOM such that it sends a versioned revision order.

See the following sections for more information:

- [Modeling Changes to Orders](#)
- [Modeling Processes and Tasks](#)

Cartridge Management Considerations for SOM

What kinds of cartridge management scenarios you want to plan for in advance, such as the impact of upgrading cartridge functionality, how such upgrades impact run-time orders, how best to structure cartridges to minimize the impact of such changes, and so on.

See "[Managing OSM Solution Cartridges](#)" for more information.

Planning OSM TOM Solution Requirements

This section describes OSM modeling entities and functions involved in the OSM TOM role. It includes examples intended to guide you in understanding how they can use these entities and functions in your OSM TOM solution. This information can help you plan your implementation efforts by understanding the full scope of a typical TOM solution. Follow the links in each section for specific details about the functions described.

The bottom up approach is where you begin to analyze a conceptual model from the perspective of the network resources and infrastructure in place to fulfill orders.

Modeling TOM Order and Order Recognition Requirements

You must determine what kind of orders you need to model in OSM TOM, what order life-cycle states and state transitions the orders have, and who has permissions to do various tasks in fulfilling the order.

At this point, it is important to understand that the TOM order is a child of the SOM parent order and must report back to the SOM order component that generated the technical order. In the example provided in "[Modeling COM Orchestration Fulfillment Patterns and Fulfillment Modes](#)", the order component that generated the service order to TOM is a task that is part of a process triggered by the CreateTechnicalOrderFunction/OSMTOMsystem/OrderGranularity run-time order component. Any notification from OSM TOM relating to the service order would return back to this task.

Depending on how you have modeled your SOM solution, OSM TOM may receive more than one technical order. For example, you may want to configure OSM COM to send a separate service orders to fulfill the broadband internet CFS and another that fulfills a VoIP CFS. OSM SOM would process these orders separately. OSM SOM sends the service order to the SRM system to generate technical actions that OSM SOM sends to OSM TOM as a technical order. Therefore, OSM TOM would receive two separate technical orders to fulfill resource actions on the original sales order sent to OSM COM.

Like OSM COM, you may want to create separate target orders for each technical order based on the different domains they interact with (broadband, VoIP, Mobile, and so on), or one generic target order that receives all technical orders.

If you create individual order types, you also need to create corresponding order recognition rules that match incoming orders to the target order. You can consider the following approaches when modeling order recognition rules for OSM TOM:

- You would typically not have multiple OSM SOM instance interacting with the same OSM TOM instance using different message format, but usually each OSM SOM instance would have its own OSM TOM instance. This means it is unlikely that you would need multiple order recognition rules pointing to the same OSM TOM instance target order.
- If you have multiple order target orders based on domain groupings, you must create a separate order recognition rule for each target order type.

- If you have one OSM instance operating in more than one role, for example, if the same system is operating in both the SOM and the TOM role, you need to configure an order recognition rule that points to a corresponding target order for each role.
- If you have one OSM SOM instance and one OSM TOM instance, then you typically need only one order recognition rule that points to one target order.

You need to determine what corresponding order recognition rules you need to model in OSM to recognize, validate, prioritize, and transform order data from sales orders into a matching OSM target order. You must map incoming order data to the data defined in the creation task of the target order.

See the following sections for more information:

- [Modeling Orders and Permissions](#)
- [Modeling Order Life-Cycle Policies](#)
- [Modeling Order Recognition](#)

TOM Data Modeling Considerations

Answer the following questions to determine what corresponding order data you need to model on the OSM order to capture the incoming order data:

- What data is required to complete an order?
- What data do the tasks require? For example, what data is required on each interaction with the Activation, the PWG, the WFM, and the SCM systems? What message format do these interactions require?
- Which data is not required on the technical order, but is required by the different fulfillment systems that OSM TOM interacts with?
- Does the incoming order include all of the data needed for the order? If not, you can use data providers in your tasks to get the data from some other source.
- Which nodes on the service order contain order item information that represents the actions on the order?
- What order item parameters represent the name of the action order item?
- If the order items are hierarchical what data elements specify the parent child relationship between each order item? For example, you may want to specify a hierarchy between an overall parent action with related child order items, such as CreateDSL_CPE with children order items that decompose to a shipping component, another to a workforce management component, and a third for the activation component.
- If the order items must be delivered at different times and dates, what data element contains the requested delivery date?
- What data element specifies the action that must be performed on the order item during the fulfillment process? For example, add, change, delete, move, and so on.
- What data element specifies the overall purpose of the order, such as deliver, cancel or technical service qualification (TSQ)?

See the following sections for more information:

- [Modeling Order Data](#)
- [Modeling Orders and Permissions](#)
- [Modeling Behaviors](#)

Modeling TOM Orchestration Order Items and Bindings Conceptual Model Parameters

You must determine what nodes in the incoming order you want to designate as order items containing actions on resources and RFSs. What data in the order items you want to use for specific orchestration functions, such as the actions you want OSM to perform on those order items, the requested delivery date when the order item actions need to occur, what CFS the order item represents that OSM then uses to map the order item to specific fulfillment patterns, and so on.

You can also use order item parameter bindings to bind conceptual model resources and RFSs with their corresponding conceptual model actions and the parameters to a OSM TOM order item specifications. This allows you to have strongly typed parameters that you don't need to model within order item specifications. For example, [Figure 1-1](#) defines the DSL resource-facing service that can optionally be fulfilled using the ADSL or VDSL interface, that also requires a DSL customer premise equipment (CPE). You can use order item parameter bindings to map these conceptual model resources, RFS entities, the actions associated with them and their data to order item specifications by configuring an order item property for the order item recognition and an order item property as a dynamic parameter where the parameters are stored.

Order item parameter bindings in OSM TOM are important for validating the incoming resource and RFS data generated from the SRM system and sent to OSM TOM from OSM SOM. Transformation is not typically required in OSM TOM because the SRM system that produced the technical actions should have already used the correct format.

See "[Modeling Orchestration Plans](#)" for more information.

Modeling TOM Orchestration Order Item Decomposition

You need to decide what orchestration stages you want OSM to evaluate when decomposing resources and RFS order items into order components. These order components typically designate functions, systems, and granularity.

Based on the previous sections, you should already have some knowledge of what the service order you are creating contains. Using this knowledge, you can provide answers to the following questions:

- What are the systems that OSM must communicate with? For example, OSM TOM typically communicates with shipping, activation, WFM, and SCM systems.
- What order items do each system require? For example, the activation system may require the DSL RFS, the DSL CPE resource, and the Email account resource, the PGW may require the local loop resource, and the WFM and SCM systems may require the DSL CPE.
- What are the business functions that each system must perform on these order items? For example, the SCM must ship the DSL CPE, the WFM system must dispatch personnel to install the CPE, the PGW must configure the local loop, and the activation system must activate the DSL access node, the DSL CPE, and the Email account.
- How should OSM deliver this data for the functions to process? Does the function operate on the whole order, or does the function require a separate interaction per order item parent child order item combination on the order, or per order item?

Based on the answers you provide to these questions, can you begin to model order item decomposition stages.

[Figure 1-1](#) defines the multiple resources and RFS entities. You can decompose these order items into function, system, and granularity order components in the same way you do in the COM and SOM (see "[Modeling COM Orchestration Order Item Decomposition](#)").

You must determine what high-level tasks are involved in fulfilling customer orders at the OSM TOM level. You can define orchestration order components that correspond to the functions performed by these tasks that you can then add to the function stage (see [Figure 1-2](#)). See "[About the OSM Solution Modeling Process](#)" for a sample list of OSM TOM related tasks that can be modeled as function order components.

You must determine what kinds of OSS fulfillment systems you have at the OSM TOM level. You can define orchestration order components that correspond to these systems that you can then add to the system stage. As illustrated in [Figure 1-1](#), the systems that normally interact with OSM SOM include activation, PGW, shipping, WFM, and SCM systems.

You must also determine what kind of order granularity you need when fulfilling each function order components. For example, at the TOM level, you might configure OSM to pass the whole order to the activation and completion function but requires order item specific granularity for the shipping, WFM, and SCM related functions.

For each of these options, you need to create an order component that OSM can use to decompose the order items to, such as whole order granularity or product order item granularity, and so on. At this point, you are only defining the order components that OSM can use to decompose order items to during order decomposition. You configure the actual decomposition behaviors and conditions with other Design Studio orchestration entities such as fulfillment patterns and orchestration decomposition rules.

See "[Modeling Orchestration Plans](#)" for more information.

Modeling TOM Orchestration Fulfillment Patterns and Fulfillment Modes

You must determine what fulfillment patterns each order item should decompose to and what action you want to specify on the overall order that you can model as fulfillment modes for the order, such as deliver, technical service qualification, or cancel.

In a conceptual model project, you map conceptual model resource and RFSs to conceptual model fulfillment patterns. These resources also specify the conceptual model actions that fulfill them. For example, the AAA_Account may be associated with the ActivateAAA_Account, AlterAAA_Account, and DeactivateAAA_Account actions. In the conceptual model, you must also specify what realizes these actions. In this case these actions would be realized by an activation system, such as Oracle Communications ASAP, and more specifically, by ASAP service actions. Other resources would be realized by other systems and action types in a similar way. However, the information important to OSM in term of order item decomposition, are the resources and RFSs that contain these actions.

When you build an OSM cartridge, OSM generates a sample XML file that contains all these resource and RFS to fulfillment pattern mappings that you can reference using an OSM TOM order item specification order item property that defines XQuery logic that determines how each order item decomposes to what fulfillment pattern during run-time.

For example, a technical order to OSM TOM may have an Email_Account resource order item that maps to a conceptual model fulfillment pattern that is realized by an OSM TOM fulfillment pattern. The OSM TOM fulfillment pattern would in turn decompose the Email_Account order item into the Activation functions because there is a condition on the activation order component within the fulfillment pattern that only allows order items to decompose to that order component that contain an order item property with a value of Activation. All other functions order components would also have conditions such that only order items destined for the systems the function order components are associated with can decompose to them.

At run-time, OSM evaluates the function, system, and granularity stages in a similar way to OSM in the COM role (see "[Modeling COM Orchestration Fulfillment Patterns and Fulfillment Modes](#)"). The function, systems, and granularity stages might generate the following run-time order components with associated resource and RFS order items and the actions they contain:

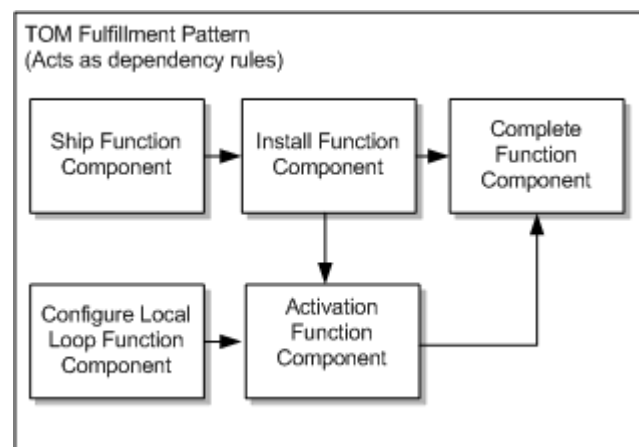
- ShipFunction/SCMsystem/OrderItemGranularity
- InstallFunction/WFMsystem/OrderItemGranularity
- ConfigureLocalLoopFunction/PGWsystem/OrderItemGranularity
- ActivationFunction/Activationsystem/OrderGranularity
- CompleteTechnicalFunction/SRMsystem/OrderGranularity

Modeling TOM Orchestration Dependencies

You must determine what dependencies exist between executable order components. These dependencies are called orchestration dependencies. You typically define all orchestration dependencies using fulfillment patterns for function order components, but you can also specify dependencies between system order components using orchestration dependency rules. In addition, you can specify when order components can start based on dates provided by customers for when they want a service to begin.

For example, the tasks specified in "[About the OSM Solution Modeling Process](#)" for OSM TOM may require that each function operate in the sequence specified in [Figure 1-9](#). The ship function and the install function must complete before the activation function can start. In addition, the activation function is dependent on the configure local loop function that communicates with the PGW. The complete function has dependencies to the install and activation function and does not start until those functions have completed.

Figure 1-9 Example OSM TOM Dependencies



These dependencies make sense when you understand what each function is doing. For example, the activation function cannot activate the service until the DSL CPE has been shipped to the customer and the CPE has been configured properly. In addition, the activation function requires that the third-party company that owns the local loop configure this resource for the CSP's customer. It is only after these dependencies have been met that the activation function can configure the network resources that deliver the service to the customer.

See the following sections for more information:

- [Modeling Orchestration Plans](#)
- [Modeling Order Scheduling](#)

Modeling TOM Processes and Tasks

You must determine what process flow of sequential or parallel manual and automated tasks are required when interacting with target systems for each functional component.

Tasks in OSM TOM typically involve sending resource and RFS order item actions to the SCM, WFM, PGW, and activation systems and receiving back the responses from these systems. You must carefully analyze the API requirements for the interfaces to each system and model the data on the tasks to meet these requirements for each interaction with these systems. You also model automated tasks with automation plug-in senders that build and send messages to these systems containing the actions each external system is to fulfill. When the systems complete their work, they send responses back to OSM TOM that OSM TOM correlates back to the automated task to an automation plug-in automator that is waiting for a response message. You must develop so that the automation plug-in automator can review the response message and determine that the system completed its tasks successfully (or whether some problem occurred) and then transitions the automated task to the completed state (or a fallout execution mode) which also completes the order component to the task belonged to.

OSM also provides a specialized automated task for communicating service requests to Oracle Communications ASAP product or the Oracle Communications IP Service Activator product. You can use this task to define the relationship between OSM task data and ASAP and IP Service Activator service actions.

See the following sections for more information:

- [Modeling Processes and Tasks](#)
- [Modeling Behaviors](#)
- [Modeling Data for Tasks](#)

Modeling TOM Fallout Scenarios

What fallout management scenarios can be anticipated, determine how they can be detected, how relevant parties or systems can be notified of the problem, and what recovery processes can be implemented.

For example, fallout scenarios may occur within the external fulfillment systems the OSM TOM communicates for a variety of reasons. For example, there may be an outage in one of the network elements that the activation system is working with, or a package sent from the SCM containing the router may have been lost or broken during delivery. Typically, many of these problems can be resolved directly in the external system, however, you may want the tasks communicate with these external systems to trigger jeopardy notifications informing upstream systems of the delay so that the upstream systems can communicate the delay back to the customer who requested the service. You must carefully analyze as many of these fallout scenarios as you can and develop fallout strategies to recover from such scenarios. In some cases, manual intervention may be required while in other cases, you may be able to model automatic fallout recovery capabilities.

See the following sections for more information:

- [Modeling Fallout](#)
- [Modeling Behaviors](#)

- [Modeling Processes and Tasks](#)
- [Modeling Jeopardy and Notifications](#)

Modeling TOM Fulfillment States

What kind of order and order item fulfillment states you need to configure to track the overall status of an order and each order item based on status messages received from external systems. You also need to consider what notification messages you want to send to interested parties or systems as the order progresses.

For example, you can map messages from the SCM, WFM, PGW, and activation systems returning as these systems process messages sent by various order components to external fulfillment state in the OSM TOM system. These external fulfillment states can represent the result of various interactions between OSM TOM and these systems on the actions contained on each resource and RFS order item that OSM TOM then aggregates into order item and order-level fulfillment states based on order and order item fulfillment state composition rule sets.

See the following sections for more information:

- [Modeling Fulfillment States and Processing States](#)
- [Modeling Jeopardy and Notifications](#)

Modeling TOM Processing States

You can track the state of order items by mapping responses from the SCM, WFM, PGW, and activation systems to resource and RFS order item processing states. You must decide what messages correspond to which predefine order component order item processing state that OSM provides. OSM then aggregates these order component order item processing states for each order item into an overall order item processing state. You may decide to use warning and failure order item processing states to trigger jeopardy notifications from OSM TOM to OSM SOM or from OSM TOM to fallout personnel. In many cases, you can also use the same messages from external systems to trigger external fulfillment state changes.

See the following sections for more information:

- [Modeling Fulfillment States and Processing States](#)
- [Modeling Jeopardy and Notifications](#)

Modeling Change Order Management for TOM

You need to determine if you want to enable change order management, and if you do, what change order management scenarios you want OSM TOM to support.

You must consider change order management based on end-to-end scenarios that span OSM COM, SOM, and TOM. For example, if you enable OSM SOM to send a revision order through to OSM TOM from the OSM SOM Create Technical Order function then you must decide what compensation OSM TOM must undertake to implement the changes in the revision order. For example, you might consider some of the following question:

- Is there a point of no return where you do not want OSM TOM to accept any new revision orders from the OSM SOM Create Technical Order function? For example, you may want to configure a point of no return that is associated with when the activation system completed the activation functions based on an external fulfillment state change. This would mean that OSM TOM does not accept revision orders from OSM SOM stemming from changes coming from OSM COM service orders. Or you may decide that OSM TOM

should never accept revision orders, in which case you could disable this functionality on the target order specification.

- For each automated task that communicates with a different external fulfillment system, you must determine how the task should behave based on the changes on the technical order.

See the following sections for more information:

- [Modeling Changes to Orders](#)
- [Modeling Processes and Tasks](#)

Cartridge Management Considerations for TOM

What kinds of cartridge management scenarios you want to plan for in advance, such as the impact of upgrading cartridge functionality, how such upgrades impact run-time orders, how best to structure cartridges to minimize the impact of such changes, and so on.

See "[Managing OSM Solution Cartridges](#)" for more information.

About the OSM SDK

A number of directories within the SDK are referenced in procedures throughout this guide. SDK is available as a separately downloadable .Zip file which is common for both OSM cloud native and OSM traditional.

Part II

Implementing an OSM Solution

Part II contains the following chapters providing information about implementing an Oracle Communications Order and Service Management (OSM) solution:

- [Modeling Orders and Permissions](#)
- [Modeling Order Life-Cycle Policies](#)
- [Modeling Order Recognition](#)
- [Modeling Orchestration Plans](#)
- [Modeling the Order Transformation Manager](#)
- [Modeling Processes and Tasks](#)
- [Modeling OSM Data](#)
- [Modeling Behaviors](#)

2

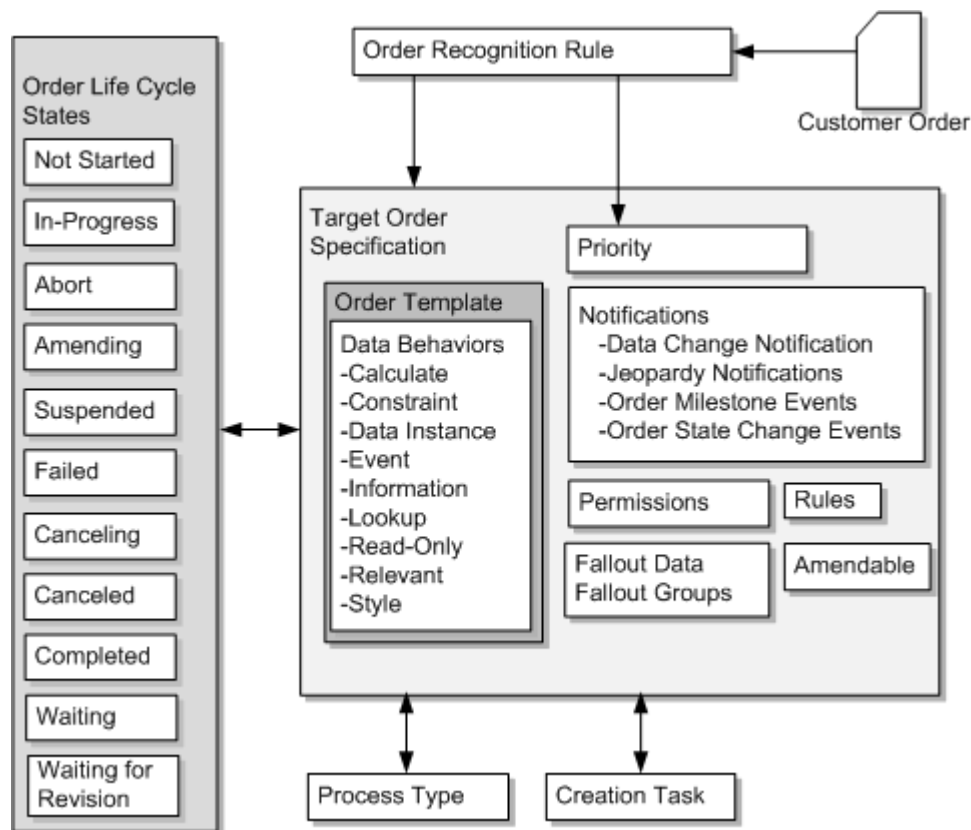
Modeling Orders and Permissions

This chapter describes how to model orders and permissions in an Oracle Communications Order and Service Management (OSM) solution.

Modeling OSM Orders

The order specification is the cornerstone model entity in Oracle Communications Service Catalog and Design - Design Studio; most other specifications among cartridges are tied, directly or indirectly, to the order specification to control order execution. [Figure 2-1](#) shows the entities that relate to an order specification and the content you can configure in order specifications.

Figure 2-1 Order Specification Configuration and Related Entities



In the order specifications you can define:

- The **order template**, which specifies the elements and structures of the order data that OSM receives from incoming orders and from other fulfillment systems.

- **Order priority** in conjunction with the priority defined on an order as detected by the order recognition rule. See "[Modeling Order Recognition](#)" for more information about order priority.
- Various order-level **notifications**. See "[Modeling Jeopardy and Notifications](#)" for more information about notifications you can configure in the order specification.
- Whether the order is **amendable**. See "[Modeling Changes to Orders](#)" for more information.
- **Fallout data** and **fallout groups** that define data and groupings of data that can potentially trigger a fallout exception on tasks that are associated with the fallout data or data groups. Fallout exceptions trigger amendment processing. See "[About Modeling Fallout Exceptions](#)" for more information.
- **Permissions** that associating roles to query tasks. Query tasks define what data can be displayed to a user associated with a specific role (whether a human user or a user account associated to automated tasks). See "[Modeling Roles and Setting Permissions](#)" for more information.
- **Rules** that to determine when notifications should run, when various process flow decisions or actions should occur, within decomposition rule to determine when order items should decompose to an order component, and so on. You can use rules in various OSM entities.

In the order specification, you must do the following:

- You must designate **creation task** data that defines the internal OSM order data, elements and structures that OSM generates as part of order processing (such as control data that OSM uses to generate orchestration plans), and any elements and structures generated by external fulfillment systems in response to messages from OSM. A creation task is a manual task that is not part of a process flow where you define data elements and structures in the Task Data tab.
- You must designate an order life-cycle policy that the order uses to determine valid states and state transitions for the order. Order states define sequential states through which an order passes and the transactions it undergoes from the time it is received in OSM until the time it is completed. For example, an order can be in progress, not started, suspended, and so on. You can enable multiple states in the order life-cycle policy and define what transitions can occur between states. For example, you can configure an order to be able to transition from in progress to cancelled. For more information about order life-cycle policies, see "[Modeling Order Life-Cycle Policies](#)".
- You must define whether OSM triggers an orchestration process or a standard process. An orchestration process causes OSM to generate an orchestration plan. An orchestration plan orchestrates order items into order components that trigger a series of standard processes. Most OSM orders require orchestration.

You can use the following OSM Web Service operations to submit orders:

- **CreateOrderBySpec** In this operation, you must specify the cartridge and order type so that OSM understands which Order entity to use to process the order. Also, the incoming order payload has to be in XML format as defined in the cartridge.
- **CreateOrder** This operation accepts arbitrary payload in XML for the incoming order. You specify an order recognition rule to recognize the payload, and transform it to the format as defined in the cartridge. There is no need to specify the cartridge or order type in the operation.

The target order specification runs if the CreateOrder request includes the **StartOrder=true** parameter and value in the order header.

See *OSM Developer's Guide* for more information about `CreateOrderBySpec` and `CreateOrder`.

With either operation, when the order is created in OSM, it is tied to the order type in the cartridge. OSM relies on that cartridge and any the dependent cartridges to determine how to display and process the order. This means, as long as the order resides in the OSM server and is not purged, the cartridges must remain in the run-time OSM environment.

You can use Java Message Service (JMS) or HTTP or HTTPS to send orders to OSM. Use JMS on production systems, because it provides quality-of-service guarantees not available from HTTP or HTTPS. Use HTTP or HTTPS on development and test systems (see *OSM Installation Guide* for more information).

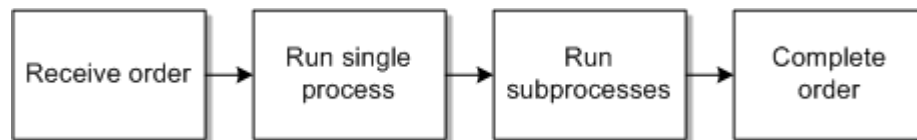
About OSM Orders Without Orchestration

For orders that do not require an orchestration plan for fulfillment (called **process-based orders**), the OSM runs a single process and any subprocesses defined within the process, which includes tasks such as `Activate_DSLAM`. When a process-based order is submitted to OSM, the following occurs:

1. OSM starts the process.
2. The process can start subprocesses that run sequentially or in parallel.
3. After the last task has completed, the order transitions to the Completed state.

[Figure 2-2](#) shows the process flow for a process-based order.

Figure 2-2 Process-Based Order

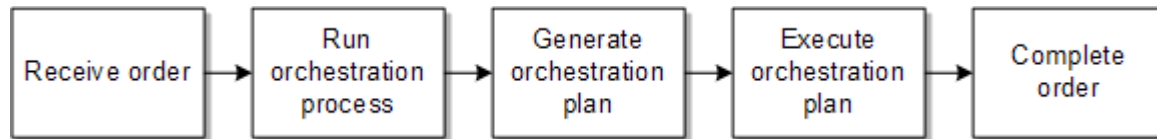


About OSM Orders With Orchestration

For orders that require an orchestration plan for fulfillment, (called **orchestration-based orders**), OSM runs an Orchestration process. When a orchestration-based order is submitted to OSM for processing, the following occurs:

1. OSM starts the orchestration process.
2. OSM generates the orchestration plan which includes run time order components that run processes with tasks.
3. After the last task has within each order component completes, the order component completes.
4. After the last order component completes, the order transitions to the Completed state.

[Figure 2-3](#) shows the process flow for an orchestration-based order.

Figure 2-3 Orchestration-Based Order

Modeling Roles and Setting Permissions

You use roles to control what operations can be performed and what data can be viewed by OSM user that you associated with the roles. You create roles then apply the roles to OSM entities in Design Studio. For example, roles are used in the following OSM entities:

- **Order specifications:** You define what order data users with specific roles can view in the OSM web clients by defining this data in query tasks and assigning the query tasks to roles within orders specifications. The OSM web client uses the query tasks to determine what data to display to users. The role applied to a query task determines the data that users associated to the role can retrieve. For more information about query tasks, see "[Modeling Query Tasks for OSM Clients](#)". You also define filters that specify whether orders with specific values can be displayed to users with the defined roles, and flexible header that define custom searchable data fields.

[Figure 2-4](#) shows roles defined in an order specification in Design Studio. In this example, members of BillingUpdateRole are allowed to view orders for customers in the 408 and 510 area codes.

Figure 2-4 Roles Defined in an Order Specification

Roles	Role Settings			
DefaultRole	Details Filters Query Tasks			
ProvisionRole				
ProvisioningUpdateRole				
SummaryRole				
BillingUpdateRole				
	Condition	Filter Node	Operator	Value
		/CustomerDetails/areaCode	=	408
	Or	/CustomerDetails/areaCode	=	510

- **Order life-cycle policy:** You define what transactions can be performed by users associated with the roles assigned to each transition. For example, you may want to a standard role to handle normal order processing from the Not Started state through to Completed state. You may also want to assign a role for fallout management operations or amendment processing work. For more information, see "[About Modeling Transition Permissions](#)".
- **Tasks:** You define what tasks can be performed by users associated with the roles assigned to each task. For example, you may want a role that can run normal processing tasks, another for tasks during amendment processing, and another for tasks during fallout management. You define what data is available for each role associated to these tasks functions using query tasks. For more information about query tasks, see "[Modeling Query Tasks for OSM Clients](#)".
- **Order, task, and process notification:** You define what notifications are sent to which group of users by assigning roles to specific notification instances in the Order editor, a Task editor, or a process activity or flow.

- Order components: You define what data users with specific roles can view by applying those roles to query tasks and assigning the query tasks to components. OSM web clients uses the query tasks to specify which what data to display to users. The role applied to a query task determines the data that task will retrieve. For example, you may define a ProvisioningRole for a provisioning order component that allows OSM client users to view certain data.

Figure 2-5 shows roles used in an order component. In this example, members of ProvisioningRole can perform queries based on ProvisioningFunctionTask and view the data in both the summary and detail views in the Order Management web client.

Figure 2-5 Roles Used in an Order Component Specification

Roles		Query Tasks		
OrderDisplay				
ProvisionRole				
automation				
		Name	Summary	Detail
		ProvisioningFunctionTask	☒	☒

- Order item specification: You can associate roles with corresponding query tasks from the Order Item Specification Permissions tab. The method of applying roles in an order item specification is identical to the method of applying roles in an order component specification. For more information about query tasks, see "[Modeling Query Tasks for OSM Clients](#)".

In addition to associating roles with OSM entities, you can also configure permissions for various actions on the roles. Figure 2-6 shows a role defined in Design Studio. In this example, users assigned to this role can generate online reports, search for orders, and access the Task web client Worklist display.

Figure 2-6 Role Defined in Design Studio

Role : OrderDisplay

Display Name

OrderDisplay

Permissions

☐ Create Versioned Orders
☐ Exception Processing
☒ Online Reports
☐ Order Priority Modification
☐ Reference Number Modification
☒ Search View
☐ Task Assignment
☒ Worklist Viewer

Role

[Table 2-1](#) shows the actions that can be assigned to roles in Design Studio.

Table 2-1 Functions Assigned to Roles

Function	Description
Create Versioned Orders	Enables users to create orders for different versions of cartridges. If not granted this permission, users can create orders only for the default version of the cartridge.
Exception Processing	Enables users to alter the flow of a process by applying exception statuses at any time throughout the process.
Online Reports	Enables users to view summarized reports on all orders and tasks on the system.
Order Priority Modification	Enables users to modify the priority of a task in an order.
Reference Number Modification	Enables users to modify the reference number of an order.
Search View	Enables users to access the order Query function. See " Specifying Which Data to Display in the OSM Web Clients " for more information.
Task Assignment	Enables users to assign tasks to others.
Worklist Viewer	Enables users to display the worklist in the Task web client.

Because roles are defined globally in OSM, roles specified in one cartridge can be applied to any other cartridge, and roles used in one order can also be used in any other order. If you want to further restrict certain operations in an order, you must do so in the Design Studio entities that the roles are associated with, such as the life-cycle policy transaction or the task execution modes.

You associate roles with OSM user accounts using the OSM Order Management web client. Roles are called workgroups in the OSM Order Management web client. Each user account can belong to as many workgroups as are available on the OSM server. For more information, see *OSM Order Management Web Client User's Guide*.

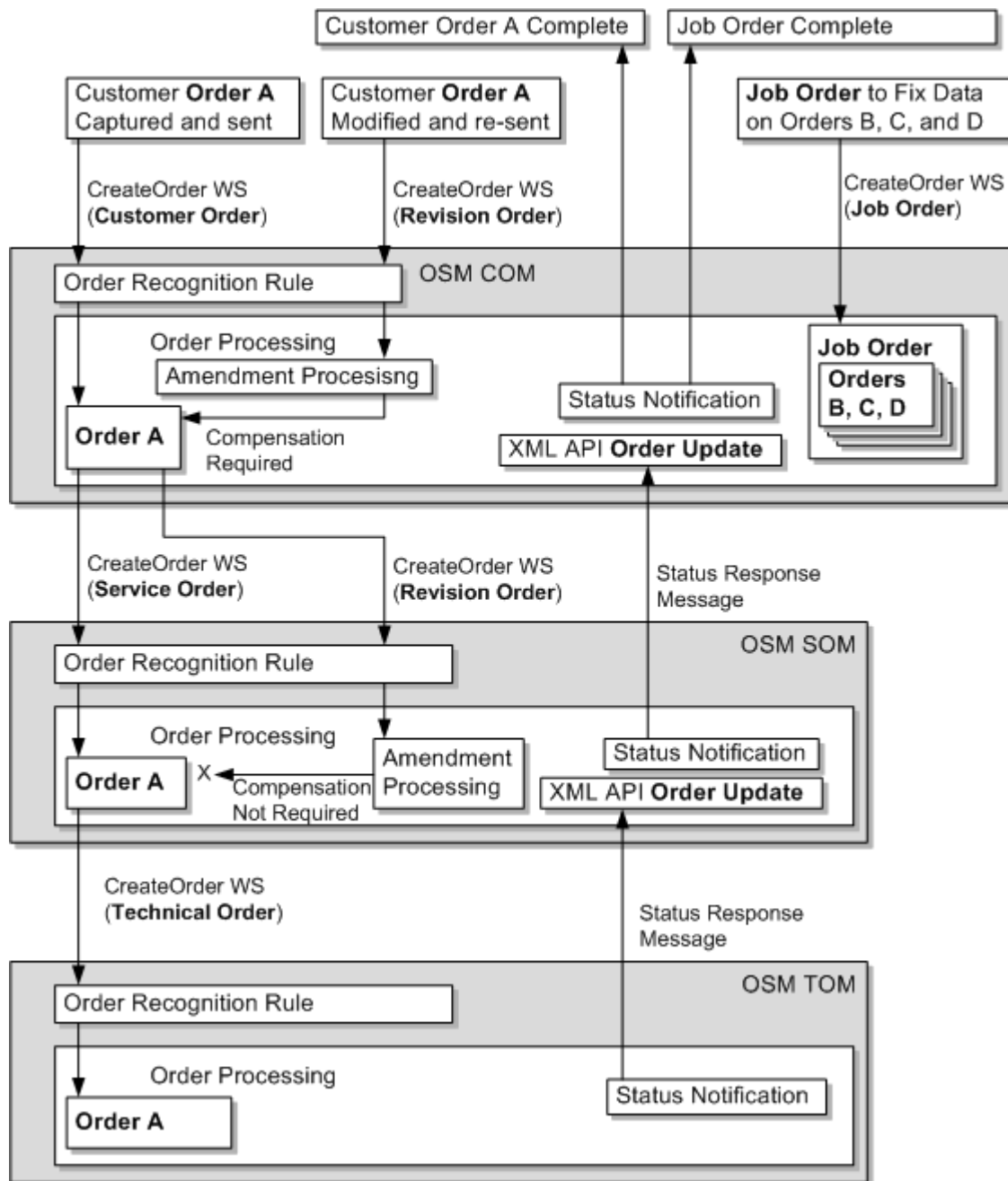
About Order Types

[Figure 2-7](#) shows OSM orders in different order processing scenarios and how OSM receives them. These scenarios include:

- **Customer orders, service orders, and technical orders** that are sent to OSM systems running in the central order management (COM), service order management (SOM), and technical order management (TOM) roles. For more information, see "[About Determining the OSM Functionality to Implement](#)".
- **Revision orders** sent to change an in-progress order. For more information, see "[Modeling Changes to Orders](#)".
- **Order update** performed either manually through the Task web client or through an automation task automator plug-in that sends an UpdateOrder request. For more information, see "[About Order Updates](#)".
- **Job orders** performed either manually through the Order Management web client or through the OSM CreateOrder Web Service operation. For more information, see "[Using a Job Control Order to Manage Multiple Orders](#)".

See "[Modeling OSM Data](#)" for more information about how orders and order items are structured.

Figure 2-7 Order Types and Order Processing



About Order Updates

You can update order data within customer, service, and technical orders that have already been created in OSM. OSM defines the following contexts where you can update order data:

- Order context:** This context defines an overall view of OSM data. Although you can update order data in this context, doing so may compromise the integrity of order data, especially if the data you update may be subject to amendment processing. If you know that the data you want to update in the order context should never trigger amendment processing, you can mark the data as not significant (see "[About Data Significance](#)" for more information).

- Task context: This context defines a task specific view of OSM data. You typically use tasks to communicate with external fulfillment systems or manual task users that make changes to the data define in the task. Updating order data within the task context ensures the data integrity, especially when the data you update may be subject to amendment processing. See "[About Order-Level and Task-Level Compensation Analysis](#)" for more information about how tasks ensure data integrity during amendment processing.

You can update order data on orders and tasks in following ways:

- XML API UpdateOrder
After receiving a JMS response message from an external system at an automation task automator plug-in, you can use the XML API UpdateOrder to update data or add any new data to the order. For example, you can use UpdateOrder to update any status notification data returned from an external system or another instance of OSM (see [Figure 2-7](#)). Oracle recommends that you run the XML API only from within the task context.
- OSM Java API methods
After receiving a JMS message from an external system at an automation task automator plug-in, you can use various OSM Java API methods to update data or add any new data to the order. Oracle recommends that you run these Java API methods from within the task context.
- Web Service UpdateOrder
You can use the OSM Web Service UpdateOrder operation to update order data outside of the Task web client and the automation framework. However, OSM Web Service operations can only access the overall order data context and do not have direct access to the task context. Use caution because doing so can compromise order integrity.
- Task web client
Personnel can update task data manually, by opening and editing an order using the Task web client order query. Changes to task data in the Task web client are within the task context.

See *OSM Developer's Guide* for more information about updating order data using the XML API UpdateOrder, the OSM Web Service UpdateOrder operation, and the OSM Java API methods. See *OSM Task Web Client User's Guide* for more information about using the Task web client to update order data.

Update orders can:

- Update a complete order. The existing order is updated (elements are added, changed, or deleted) to match the supplied order. Order-level order updates are typically sent in the context of order-level notifications, jeopardy notifications, or event automation automators. See "[Modeling Jeopardy and Notifications](#)" for more information about update order transactions used in the context of jeopardies, notifications, and events.
- Update nodes in an order. Elements can be added or changed. Deleting nodes are not performed using the update node functionality. The nodes are supplied in the format of the existing order and are typically sent as part of task-level automation automators.
- Add, delete, or change element data values that are typically sent as part of task-level automation automators.

Using a Job Control Order to Manage Multiple Orders

Job control orders enable you to efficiently apply changes to many orders at the same time. You can use job control orders to apply the same set of OSM Web Service operations or OSM Order Management web client actions to multiple orders. For example, you could model a job

control order using a CreateOrder OSM Web Service operation that selects multiple orders, suspends each of them, updates order data on all the orders, and then resumes each orders.

You can specify how many OSM Web Service operations or Order Management web client actions OSM can process at the same time. You can specify a failure threshold for job control order operations that, if crossed, causes the entire job control order to enter the suspended state. In addition, job control orders support a variety of counters that track job order progress.

To use job control orders:

1. Ensure that the **JobControl_Solution** portal archive (PAR) file has been deployed on the OSM server. This PAR file can be deployed either when OSM is first installed or manually. For manual deployment instructions follow the instructions in the readme file that is in the *OSM_home/ProductCartridges/install* directory. more information about deploying the job control order cartridge, see *OSM Installation Guide*. The PAR file packages the following OSM projects that you can see in the Design Studio Cartridge Management editor when you query an OSM server for cartridge information:
 - **BatchJobCommonResources**: Contains the job control order system configuration file that defines default job control order settings.
 - **JobControl**: contains the OSM Design Studio cartridge that enables the job control order functionality.
 - **JobControl_Solution**: contains the solution cartridge that packages the BatchJobCommonResources and JobControl cartridges.

Note

You can only view these cartridges when you query OSM for cartridge information. Oracle does not provide access to the actual Design Studio projects.

2. In the OSM WebLogic server, create a new user account or use an existing user account and associate it with the OMS_ws_api group.

Note

You can also associate the user account to the OMS_client group to give the user access to Order Management web client and Task web client.

3. In the Order Management web client, associate the user account with the JCO_UserRole or the job control order functionality in the JCO_SuperUserRole workgroups (roles). For more information, see ["About Job Control Order Permissions"](#).
4. Ensure that all orders to be managed by job control orders have roles associated with the default oms-automation OSM user account.
5. Model job control orders using the CreateOrder OSM web service operation and the following syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<ord:CreateOrder xmlns:ord="http://xmlns.oracle.com/communications/ordermanagement">
  <CreateJob xmlns="http://oracle.communications.ordermanagement.cartridge/job">
    <FailureThreshold>threshold</FailureThreshold>
    <ConcurrentOperationsAmongOrdersInJob>degree</ConcurrentOperationsAmongOrdersInJob>
    <Priority>priority</Priority>
    <RequestedDeliveryDate>requestedDeliveryDate</RequestedDeliveryDate>
    <Selection>
```

```

    byOrderCriteria
  or
  bySelectionCriteria
</Selection>
<Operations>
  operations
</Operations>
</CreateJob>
</ord:CreateOrder>

```

where:

- *threshold*: (Optional) The number of operations that must fail before the job control order automatically transitions to the Suspended state. Valid values are percentages from 1% to 99%, absolute values, such as 10, 15, and so on, or 0 which specifies no threshold. If specified, this value overrides the default value specified in the **batch_job_cfg.properties** file (see "[About Job Control Order System Configuration Files](#)").
- *degree*: (Optional) The number of executable components created for each operation in the job. If specified, this value overrides the default value specified in the **batch_job_cfg.properties** file (see "[About Job Control Order System Configuration Files](#)").
- *priority*: (Optional) The priority of the job control order (see "[Modeling Order Priority](#)").
- *requestedDeliveryDate*: The date and time when the job control order is to begin (for example, 2014-08-01T03:10:00Z). For more information about requested delivery dates, see "[Modeling Order Scheduling](#)".
- *byOrderCriteria*: The orders to which the job control order operations apply. For example:

```

<Orders>
  <OrderId>1</OrderId>
  <OrderId>2</OrderId>
  etc...
</Orders>

```

- *bySelectionCriteria*: The selection criteria that OSM uses to match corresponding orders to. For example:

```

<ord:SelectBy>
  <ord:OrderState>lifecyclestate</ord:OrderState>
  <ord:Cartridge>
    <ord:Name>cartridgename</ord:Name>
    <ord:Version>1.0.0.0</ord:Version>
  </ord:Cartridge>
</ord:SelectBy>

```

The job control order selection criteria is identical to the SelectBy option of the FindOrder OSM Web Service operation. See *OSM Developer's Guide* for more information. The number of orders returned using the selection criteria is limited by the **FindOrderMaxOrderThreshold** oms-config.xml parameter. The default value is 1000. For information about modifying the default FindOrderMaxOrderThreshold parameter, see *OSM System Administrator's Guide*.

- *operations*: One or more OSM Web Service operations listed in "[About Job Control Order Operations](#)" according to the permissions listed in "[About Job Control Order Permissions](#)". For example:

```

<ord:SuspendOrder>
  <ord:Reason>Job Test</ord:Reason>

```

```

</ord:SuspendOrder>
<ord:UpdateOrder>
  <ord:View>CreationView</ord:View>
  <ord:DataChange>
    <ord:Update Path="/account_information/amount_owing">444</ord:Update>
  </ord:DataChange>
</ord:UpdateOrder>

```

About Job Control Order Operations

You can run combinations of the following web service operations as a part of a job control order:

- **SuspendOrder:** Causes OSM to stop all processing on the orders specified in a job control order. The orders transition from the In Progress or Not Started state to the Suspended State.
- **ResumeOrder:** Causes OSM to resume processing all orders specified in a job control order that are in the Suspended state. The orders transition from the Suspended state to In Progress state.
- **CancelOrder:** Causes OSM to cancel all orders specified in the job control order. All applicable order components and tasks are undone. The orders transition to the Cancelling state while order components and tasks are running in the undo mode. After all order components and tasks complete, the order transitions from the Cancelling state to the Cancelled state.
- **AbortOrder:** Causes OSM to stop all orders specified in the job control order. The orders transition to the Aborted state.
- **FailOrder:** Causes OSM to fail all orders specified in the job control order. The orders transition to the Failed state.
- **ResolveFailure:** Causes OSM to revert all orders specified in the job control order to the previous order state before the orders failed.
- **RetryOrder:** Causes OSM to retry an order or a collection of order components for a given order. All failed tasks in the order or within the order components are retried and transitioned from the failed execution mode back to the normal execution modes such as do, redo and undo.
- **UpdateOrder:** Causes OSM to update order data on all orders specified in the job control order.

Operations run in the sequence they appear in the job control order. You must ensure that the sequence is logical. For example, you can suspend, update, and then resume an order, but you cannot resume, suspend, and update an order. You must also ensure that order life-cycle policies of the orders that the job control order interacts with supports the use of the operations you want to be available to a job control order.

For more information about the transitions associated with job control order and the roles that can run these transitions, see "[About Job Control Order Permissions](#)". For more information about OSM Web Service operation syntax, see *OSM Developer's Guide*.

About Job Control Order Permissions

The job control order solution cartridge contains the **JCO_UserRole** and the **JCO_SuperUserRole** workgroups (roles) with different permissions configured for each. You associate user accounts with workgroups using the Order Management web client. For more information about associating user accounts with workgroups, see *OSM Order Management Web client User's Guide*.

[Table 2-2](#) shows the permissions available to each role.

Table 2-2 Permissions for JCO_SuperUserRole and JCO_UserRole

Permission	JCO_SuperUserRole	JCO_UserRole	Description
Version Orders	Yes	No	Allows users to create orders for different versions of cartridges. If not granted this permission, users can create orders only for the default version of the cartridge.
Exception Processing	Yes	No	Allows users to alter the flow of a process by applying exception statuses at any time throughout the process.
Order Priority Modification	Yes	Yes	Allows users to modify the priority of a task in an order.
Reference Number Modification	Yes	Yes	Allows users to modify the reference number of an order.
Task Assignment	Yes	Yes	Allows users to assign tasks to others.
Modifications to configuration parameters in order data	Yes	Yes	Allows users to modify default configuration parameters for job control orders.
Modifications to other order data	Yes	No	Allows users to modify order data in operations.
Suspend State Transaction	Yes	Yes	Allows users to suspend or update a job control order in the In Progress state. Job control orders automatically enter the Suspended state when the job control order passes the jobFailedOperationsThreshold threshold. Users can also manually suspend a job control order by sending a SuspendOrder web service operation.
Resume State Transaction	Yes	Yes	Allows users to resume a suspended order. When a Suspended order is resumed, it returns to the state it was in prior to the Suspended state (for example, In Progress or Not Started).
Abort State Transaction	Yes	No	Allows users to stop an order. All transitions to the Aborted state occur after the grace period expires.
Cancel State Transaction	Yes	No	Allows users to Cancel a job control order. Canceling a job control order stops all further processing of the job control order. The cancel order does not reverse job operations that have already run.
Create Job Control Order	Yes	Yes	Allows users to create a job control order.
Transition to Complete State	Yes	Yes	The job control order enters the Completed state when all operations on all orders have completed, whether successfully or unsuccessfully.
Transition to Failed State	Yes	Yes	The job control order may transition to the Failed state. However, job control orders do count all failed operations.

About Job Control Order System Configuration Files

[Table 2-3](#) shows the job control order configuration file parameters that manage all job control orders. Parameter values specified directly in a job control order override the job control order configuration file parameters.

Table 2-3 Parameters for the `batch_job_cfg.properties` File

Parameters	Description	Default
Concurrent Operations among Orders in Job	The number of executable components created for each operation in the job.	1
Failure Threshold	The number of operations that must fail before the job control order automatically transitions to the Suspended state. Valid values are percentages from 1% to 99%, absolute values, such as 10, 15, and so on, or 0, which specifies no threshold.	0

Viewing Orders in OSM Web Clients

You can view orders in the following ways:

- You can display the orchestration plan, and the order components and order items included in it, in the Order Management web client. For more information, see *OSM Order Management Web Client User's Guide*.
- You can display current and historical information about tasks in the Task web client. For more information, see *OSM Task Web Client User's Guide*.

Specifying Which Data to Display in the OSM Web Clients

You can choose the data to display in the OSM web clients using the following methods:

- Use **task data** to specify which data to display in the Task web client for manual tasks.
- Use **behaviors** to specify how OSM displays the task data within a manual task; for example, to hide or show task data or to make data read only. See "[Modeling Behaviors](#)" for more information.
- Use **query tasks** to specify which data to display in the Order Management web client **Summary** tab and **Data** tab. Query tasks are manual tasks that specify which data to display in the Order Management web client when opening an order. A query task is associated with a role that gives permission to view the order data that the particular role is allowed to view. For example, some users may only need to view billing related order data, while others may only need to view provisioning data. Some users may need to view the entire order. See "[Modeling Query Tasks for OSM Clients](#)" for more information.

Modeling Query Tasks for OSM Clients

Order management personnel can display orders in the Task web client and in the Order Management web client. You can specify which data is displayed by assigning query tasks to an order. The data that is specified in the query task is the data that is displayed.

You can select any task as the query task. You can also create special tasks whose only function is to specify which data to display.

[Figure 2-8](#) shows the **Permissions** tab in the Design Studio Order editor. The upper screenshot shows the permissions for the provisioning role, with the provisioning function task as the query task. The lower screenshot shows the permissions for the billing role, with the billing function task as the query task.

Figure 2-8 Roles Assigned to Query Tasks

The figure consists of two screenshots of the Design Studio Order editor, specifically the **Permissions** tab. Both screenshots show the **Display Name** as `OsmCentralOMExampleOrder`.

Upper Screenshot (Provisioning Role):

- Roles:** A list of roles including `DefaultRole`, `ProvisionRole` (highlighted), `ProvisioningUpdateRole`, `SummaryRole`, and `BillingUpdateRole`.
- Role Settings:** The **Query Tasks** tab is selected. It shows a table with columns **Name**, **Summary**, **Detail**, and **Default**. The row for `ProvisioningFunctionTask` has ☐ for Summary, ☒ for Detail, and ☒ for Default.

Lower Screenshot (Billing Role):

- Roles:** A list of roles including `DefaultRole`, `ProvisionRole`, `ProvisioningUpdateRole`, `SummaryRole`, and `BillingUpdateRole` (highlighted).
- Role Settings:** The **Query Tasks** tab is selected. It shows a table with columns **Name**, **Summary**, **Detail**, and **Default**. The row for `BillingFunctionTask` has ☐ for Summary, ☒ for Detail, and ☒ for Default.

The Order Management web client uses the following types of views to display orders; a summary view in the **Summary** tab and a detailed view in the **Data** tab. When you model a query task, you can specify which of those views (either or both) to display the query task data in.

You can use multiple tasks as query tasks for an order. When you do so:

- For the summary view, all the data is displayed in the Order Management web client **Summary** tab.
- For the detailed view, the data from the query tasks appears as options in the Order Management web client **Data** tab **View** field; each option presents the OSM user with a different view, each containing a specific set of data.

You can use multiple query tasks in the Order Management web client when using an orchestration cartridge. For process-based cartridges, only the default query task is available in the Order Management web client. To display the query task in the Task web client, select the **Default** check box, as shown in [Figure 2-8](#).

In addition to defining the data that can be displayed, you can specify who can see it by using roles. Each role that is associated with an order can be assigned different query tasks. For example, if your order management personnel includes a role for billing specialists, you can create query tasks that show data specific to their activities.

3

Modeling Order Life-Cycle Policies

This chapter describes how to model order life-cycle policies in an Oracle Communications Order and Service Management (OSM) solution.

Modeling Order Life-Cycle Policy States and Transitions

Every order has an **order state**, which indicates the current condition of the order; for example, In Progress, Amending, or Completed. These OSM order states control the progress of the order. For example, an OSM user cannot work on tasks while the order is in the Suspended state, and an order in the Aborted state cannot be restarted.

Note

The order state represents the technical processing state of the order in the OSM system, not the state of the order as defined in a CRM system, or the fulfillment state defined in a fulfillment system. OSM order states are typically not equivalent to the states of the order in the CRM system or other order-source system, which might have different states for the customer order, as well as states for order line items on the order.

A typical order uses the following states:

1. An order is created in the Not Started state.
2. When processing begins on the order, the state transitions to the In Progress state.
3. When the order is complete, it transitions to the Completed state.

Changes from one order state to another order state are called **transitions**. Each order state has a set of allowable transitions. For example, when an order is completed, it transitions from the In Progress state to the Completed state.

Transitions are controlled by **transactions**. A transaction is an action taken by the OSM system. For example, the Suspend Order transaction performs the following actions:

- Stops all processing on the order
- Transitions the order to the Suspended state

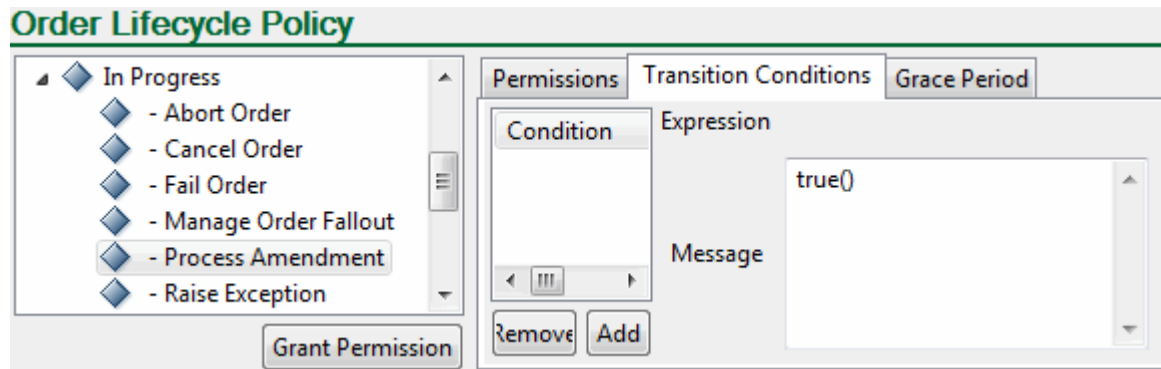
Most transactions perform transitions that change the state of the order. However, some transactions do not perform a transition to another state. For example, the Update Order transaction can make changes to an order without changing the order's state.

About Modeling Transition Conditions

Transition conditions are Boolean expressions that specify if a transition from one state to another is allowed. For example, for the Submit Amendment state, you can prevent the Process Amendment transition from occurring until a condition is true.

[Figure 3-1](#) shows the life-cycle policy for the Process Amendment transition. In this case, it returns true, so it is always allowed to transition.

Figure 3-1 Order Life-Cycle Policy for the Process Amendment Transition



A common scenario for configuring permissions is when you set the PONR for amendment processing. See *OSM Concepts* for more information.

When specifying conditions, the minimum set of required order states is Not Started, In Progress, and Completed. The life cycle must allow an order to transition to those states.

OSM uses more transactions than those shown in Oracle Communications Service Catalog and Design - Design Studio. Design Studio shows only the transactions that you can assign permissions on and set conditions for. For example, the Complete Task transaction can transition an order to the Completed state, but that transaction cannot be customized.

About Modeling Transition Grace Periods

The transition grace period specifies the amount of time that OSM should wait before transitioning the order. For example, if a Suspend Order transaction is run on an In Progress order, a grace period can allow the order processing to reach a definitive state for all currently executing tasks before transitioning to the Suspended state. In this case, OSM waits until all active tasks are in the Received, Completed, or user-defined Suspended task state. (An active task is a task that is in the Accepted state.) If the grace period expires before all tasks move out of the Accepted task state, OSM transitions the order state.

During the grace period, the target order state header in the Task web client displays the order state the order is transitioning to. The target order state is populated only when an order is in grace period.

You can specify a grace period for certain transactions, such as Suspend Order, Process Amendment, Cancel Order, and Fail Order. For other transactions, a grace period is unnecessary or not permitted, such as for Submit Amendment, Update Order, and Abort Order.

You can define the following grace period parameters:

- The length of time to wait (minimum and maximum, or indefinite)
- How often to generate a jeopardy event during the grace period

[Figure 3-2](#) shows how you can customize the order life cycle in Design Studio. In this figure, the Cancel Order exit transaction for the In Progress order state is selected. A grace period for transitioning to an order cancellation is set for a minimum of one day, and a maximum of five days. A jeopardy event is triggered every hour.

Figure 3-2 Order Life Cycle in Design Studio

Order Lifecycle Policy

Left Pane (States and Transitions):

- In Progress**
 - Abort Order
 - Cancel Order
 - Fail Order
 - Manage Order Fallout
 - Process Amendment
 - Raise Exception
 - Submit Amendment
 - Suspend Order
 - Update Order
- Not Started**
 - Abort Order
 - Delete Order
 - Fail Order
 - Suspend Order
 - Update Order
- Suspended**
 - Abort Order
 - Cancel Order

Right Pane (Configuration):

Permissions | **Transition Conditions** | Grace Period

Wait Duration

☐ Indefinitely

☒ Time frame

	Days	Hours	Minutes	seconds
Minimum	1	0	0	0
Maximum	5	0	0	0

Event Frequency

Every

Days	Hours	Minutes	seconds
0	1	0	0

About Modeling Transition Permissions

You can specify the roles that are allowed to perform a transaction. For example, while an order is in the In Progress state, your customer service role might need to perform the Update Order and Cancel Order transactions, whereas your fallout specialist role might perform only the Raise Exception transaction.

OSM Order States and Transactions

OSM includes a standard set of order states and transactions. You cannot add states or transactions, but you can control how the order transitions between states.

[Table 3-1](#) shows the OSM order states.

Table 3-1 Order States

Order State	Description
Aborted	The order has been permanently stopped. This is a final state. An order in the aborted state cannot transition to another order state. See " About the Aborted Order State " for more information.
Amending	The order is being amended. OSM identifies which tasks are affected by the amended data and compensates the order as necessary. See " About the Amending Order State " for more information.

Table 3-1 (Cont.) Order States

Order State	Description
Cancelled	The order has been canceled. All tasks have been undone back to the creation task. If an order includes an orchestration plan, the Cancelled state is the final state. The order cannot be resumed. If the order does not have an orchestration plan, the canceled order is returned to the creation task for the order. The order can be re-submitted to be run again. See "About the Cancelled Order State" for more information.
Cancelling	The order is being canceled. At least one task is running to perform amendment processing for the cancellation. While the order is in the Cancelling state, OSM undoes all completed tasks to return the order to the creation task. When OSM is finished, the order transitions to the Cancelled state See "About the Cancelling Order State" for more information.
Completed	The order has been fulfilled. There are no tasks running and processing is complete. A completed order can be canceled, updated, or deleted. See "About the Completed Order State" for more information.
Failed	The order has failed during processing. The Failed state is not a final state; the order can be resumed when the problem is fixed. See "About the Failed Order State" for more information.
In Progress	The order is actively running. Future-dated orders have an In Progress state while they wait for dependencies to be resolved. See "About the In Progress Order State" for more information.
Not Started	The order has been created but has not started. There are no tasks running. See "About the Not Started Order State" for more information.
Suspended	The order has been suspended and all processing on the order in OSM has been halted. No task can be updated or transitioned while the order is in this state. See "About the Suspended Order State" for more information.
Waiting	The order is not ready to start because it is future-dated or blocked by another order. See "About the Waiting Order State" for more information.
Waiting for Revision	The order is waiting for a revision. This state is common following compensation to an order for fallout, when the order is awaiting a revision from the order-source system to correct something that caused a failure in the originally submitted order. See "About the Waiting for Revision Order State" for more information.

[Table 3-2](#) shows transactions that are included in the order life-cycle policy and the operations they perform.

Table 3-2 Order State Transactions

Transaction	Description
Abort Order	Immediately and permanently stops order processing. Transitions the order state to Aborted. In the Order Management web client and the Task web client, Abort Order transactions are identified as "Terminate Order".
Cancel Order	Transitions the order to the Cancelling state. After OSM performs the necessary tasks to cancel the order, the order transitions to the Cancelled state. In Design Studio, you can specify a grace period to wait for all accepted tasks to complete before transitioning the order.

Table 3-2 (Cont.) Order State Transactions

Transaction	Description
Complete Task	Completes a task and allows the transition to the next task. Completing the last active task implicitly transitions the order to a Completed state. This transaction is not configurable in the life-cycle policy.
Copy Order	Copies an order; for example, when you create an order in the Task web client by copying an order. This transaction does not change the order state. It is not configurable.
Create Order	Creates an instance of an order. The transaction starts the order in either the Not Started state or the In Progress state. This transaction is not a configurable transaction in the OSM life-cycle policy. Permissions for creating an order are not set in the life-cycle policy. Instead you assign an order creation permission to roles and assign permissions on the orders.
Delete Order	Removes an order from the system. To delete orders, use the orderPurge command. See <i>OSM System Administrator's Guide</i> for more information. If the order does not have an orchestration plan, you can delete an order using the Task web client when the order is at the creation task.
Fail Order	Transitions the order to the Failed state. Processing on the order is stopped. In Design Studio, you can specify a grace period to wait for all accepted tasks to complete before transitioning the order.
Fallout Order	Compensates an existing order based on error data identified during provisioning. This transaction is not configurable in the life-cycle policy.
Manage Order Fallout	Transitions the order to the state it had before it failed. Processing on the order resumes. This transaction enables Task web client users to locate orders with errors that require manual intervention, analyze the order to determine the cause of the errors, and take the corrective action to correct errors allowing the order to continue to process normally.
Process Amendment	Transitions the order to the Amending state. This transaction is always preceded by the Submit Amendment transaction. See "About the Amending Order State" for more information. In Design Studio, you can specify a grace period to wait for all accepted tasks to complete before transitioning the order.
Raise Exception	Raises an exception. The system can be configured to initiate fallout compensation with this transaction. In this situation the order transitions to the Amending state while it compensates tasks. From the Amending state, it can transition to the Failed, In Progress, or Waiting for Revision states. For backward compatibility this transaction can also trigger a preconfigured exception process. Exception processes are incompatible with OSM's built-in compensation. An order for which an exception process is triggered cannot have compensation applied for revisions, cancellations, or fallout. In this case, the order remains in the In Progress state. In Design Studio, you can specify a grace period to wait for all accepted tasks to complete before transitioning the order.
Resume Order	Transitions the order to the In Progress state, typically from the Suspended state.
Submit Amendment	Submits an amendment but does not change the order state. This transaction is followed by the Process Amendment transaction, which changes the order state to Amending. See "About the Amending Order State" for more information.
Suspend Order	Transitions the order to the Suspended order state. Processing on the order halts. In Design Studio, you can specify a grace period to wait for all accepted tasks to complete before transitioning the order.

Table 3-2 (Cont.) Order State Transactions

Transaction	Description
Update Order	Allows changes to order data, remarks, and attachments outside the context of a task. The Update Orders can add new data elements, delete existing data elements, or change existing data element. Update Orders can be sent from locations such as: • The Task web client. • Automation plug-in XSLT or XQuery automators. • Web Services or XML API requests. In most situations, Update Order does not allow the state of the task to change; for example, when updating an order that is in the Aborted state. When an order is in the Not Started state or the Cancelled state, the Update Order transaction can start or resume the order by running the creation task. To use Update Order to start or resume an order, you need to use the startOrder flag in the Update Order transaction, in an automation plug-in, a web service operation, or through the Task web client. You cannot specify to start or resume an order by configuring the order life-cycle policy in Design Studio.

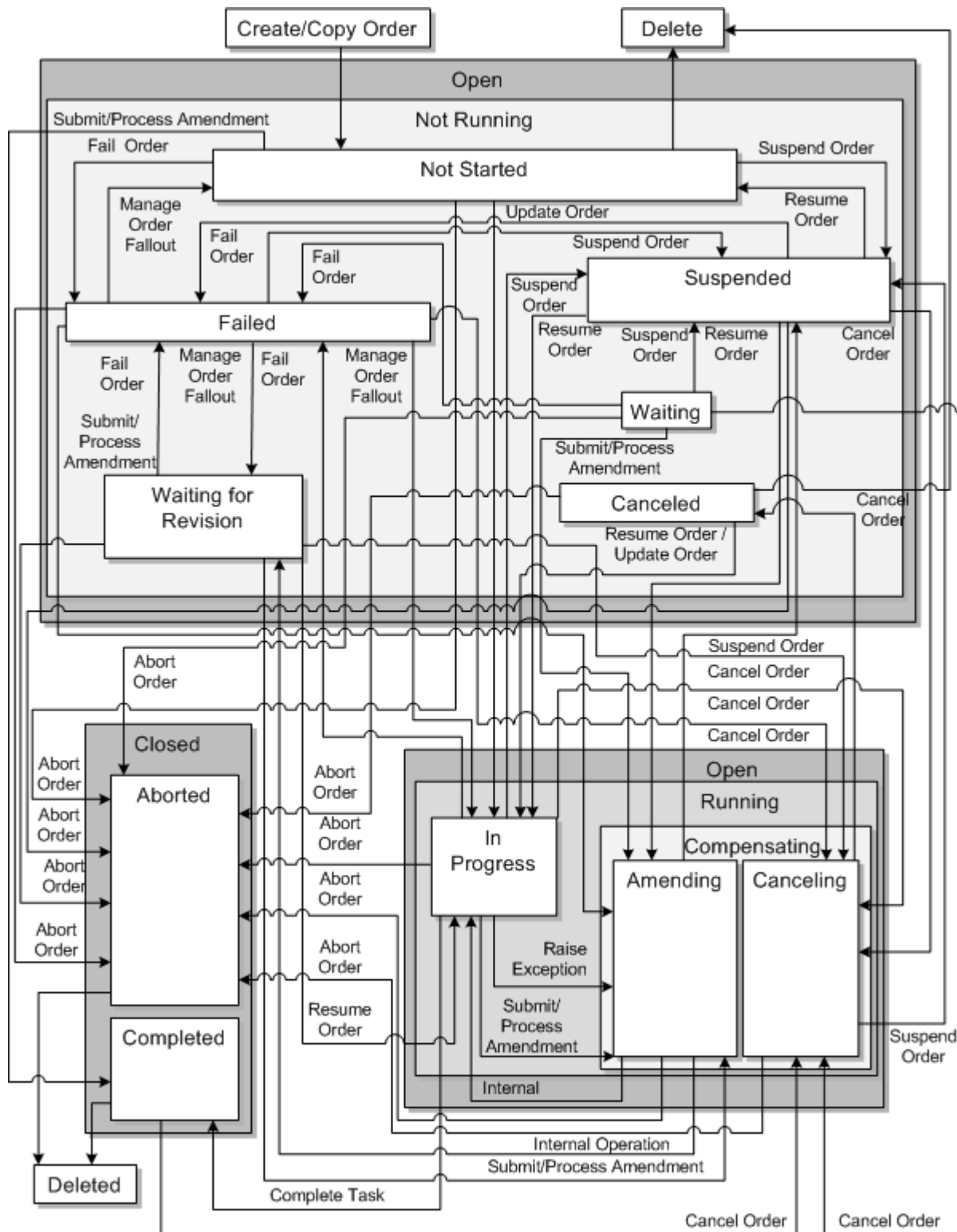
Figure 3-3 shows OSM order states and transactions.

- The transactions shown are those that perform transitions between order states. Some transactions, such as Update Order, do not always perform a transition.
- In this figure, a Resume Order transaction is shown from the Cancelled state to the In Progress state. This transaction is only possible for orders that do not have an orchestration plan. If the order has an orchestration plan, the Cancelled state is a final state and cannot be resumed.
- Some order state transitions are performed internally by OSM, not by running a transaction.
- The transition from Not Started to Completed occurs when an order is submitted for a revision to an in-flight order. In this case, all that the revision order must do is submit an amendment. When the revision order is processed, the Submit Amendment transaction places the revision order in the amendment queue. After doing so, the revision order itself requires no further processing because compensation happens to the base order, so the revision order is transitioned directly to the Completed state automatically by OSM, without going to the In Progress state.

Note

Because the transaction from Not Started to Completed for revision orders is required by OSM and is performed by the system, you cannot define permissions or conditions for it. Therefore, it is not shown as a transaction from the Not Started state in Design Studio.

Figure 3-3 OSM Order States and Transactions



About Order State Categories

Order states can be categorized by the overall condition of the order that they apply to; for example, if the order is open, closed, or running:

- **Open - Not Running**
 - Not Started
 - Suspended
 - Waiting
 - Waiting for Revision
 - Canceled
 - Failed
- **Open - Running**
 - In Progress
- **Open - Running - Compensating**
 - Amending
 - Cancelling
- **Closed**
 - Completed
 - Aborted

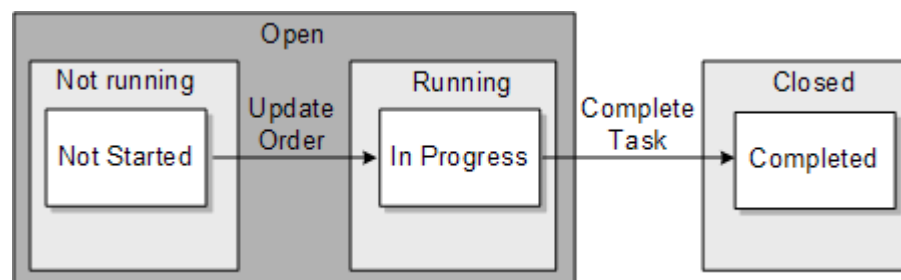
Common Order State Transitions

A typical order processing scenario uses the following order states:

1. The order is submitted to the Not Started state and transitions to the In Progress state. The order remains in the In Progress state while processing occurs.
2. When the last task has completed, the order transitions to the Completed state.

[Figure 3-4](#) shows the states, state categories, and transactions for a basic order processing flow.

Figure 3-4 Simple Order Processing Flow



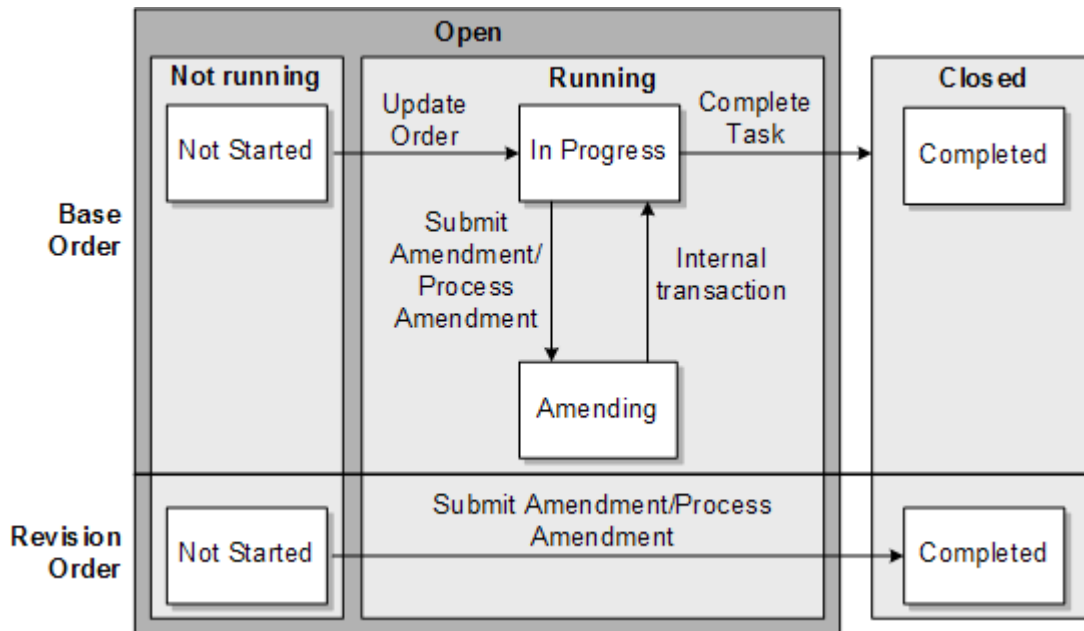
The process for revising an order uses the following order states:

1. The base order is submitted and transitions to the In Progress state.

2. The revision order is submitted and transitions to the In Progress state.
3. The base order transitions to the Amending state.
4. The revision order, after it has amended the base order, transitions to the Completed state.
5. After processing the amendment, the base order returns to the In Progress state.
6. When the last task has completed, the base order transitions to the Completed state.

Figure 3-5 shows the order states used for a revision order.

Figure 3-5 Order States Used When Processing a Revision Order



A follow-on order uses the following order states:

1. The base order is submitted and transitions to the In Progress state.
2. The follow-on order is submitted and transitions to the In Progress state, but it must wait until an order item in the base order completes before it can be processed.
3. The order item in the base order completes. The base order can continue processing, or it can complete and transition to the Completed state.
4. Since the order item in the base order has completed, the dependency has been met and the follow-on order begins processing.
5. When the last task in the follow-on order has completed, it transitions to the Completed state.

A future-dated order uses the following order states:

1. The order is submitted, but OSM determines that there is a future start date. The order transitions to the Not Started state.
2. When the order start date is reached, the order transitions to the In Progress order state.
3. When the last task has completed, the order transitions to the Completed order state.

Optional, Mandatory, and Prohibited Transactions

Transactions for each order state can be optional, mandatory, or prohibited. Optional transactions can either be allowed or prohibited based on conditions and permissions defined in the order life-cycle policy.

[Table 3-3](#) shows the order states and their transactions.

Table 3-3 OSM Order Transactions

Order State	Mandatory Transactions	Prohibited Transactions	Optional Transactions
Aborted	None	• Abort Order • Cancel Order • Complete Task • Fail Order • Manage Order Fallout • Process Amendment • Raise Exception • Resume Order • Submit Amendment • Suspend Order	• Delete Order • Update Order
Amending	None	• Cancel Order • Complete Task • Delete Order • Fail Order • Raise Exception • Resume Order • Update Order	• Abort Order • Manage Order Fallout • Submit Amendment • Suspend Order • Process Amendment
Canceled	None	• Complete Task • Fail Order • Manage Order Fallout • Process Amendment • Raise Exception • Resume Order • Submit Amendment • Suspend Order	• Abort Order • Delete Order • Update Order • Cancel Order
Canceling	None	• Cancel Order • Complete Task • Delete Order • Fail Order • Process Amendment • Raise Exception • Resume Order • Submit Amendment • Update Order	• Abort Order • Suspend Order • Manage Order Fallout

Table 3-3 (Cont.) OSM Order Transactions

Order State	Mandatory Transactions	Prohibited Transactions	Optional Transactions
Completed	None	• Abort Order • Complete Task • Fail Order • Process Amendment • Raise Exception • Resume Order • Submit Amendment • Suspend Order	• Delete Order • Update Order • Cancel Order
Failed	None	• Complete Task • Delete Order • Fail Order • Process Amendment • Raise Exception • Resume Order • Suspend Order	• Abort Order • Cancel Order • Manage Order Fallout • Submit Amendment • Update Order
In Progress	Complete Task	• Delete Order • Resume Order	• Abort Order • Cancel Order • Fail Order • Manage Order Fallout • Process Amendment • Raise Exception • Submit Amendment • Suspend Order • Update Order
Not Started	Complete Task	• Cancel Order • Manage Order Fallout • Process Amendment • Raise Exception • Resume Order • Submit Amendment	• Abort Order • Delete Order • Fail Order • Suspend Order • Update Order
Suspended	None	• Complete Task • Delete Order • Process Amendment • Raise Exception • Suspend Order	• Abort Order • Cancel Order • Fail Order • Manage Order Fallout • Resume Order • Submit Amendment • Update Order
Waiting	None	• Complete Task • Delete Order • Process Amendment • Raise Exception	• Abort Order • Cancel Order • Fail Order • Submit Amendment • Suspend Order • Update Order

Table 3-3 (Cont.) OSM Order Transactions

Order State	Mandatory Transactions	Prohibited Transactions	Optional Transactions
Waiting for Revision	None	• Complete Task • Delete Order • Process Amendment • Raise Exception • Suspend Order	• Abort Order • Cancel Order • Fail Order • Manage Order Fallout • Resume Order • Submit Amendment • Update Order

About the Aborted Order State

An order can be transitioned to the Aborted order state when an unrecoverable error or condition has stopped the processing for the order and the order cannot return to a valid processing state through a revision or fallout management activity within OSM. It can be considered a last resort to prevent any further execution of an order.

An order can be terminated manually from the Order Management web client or from the Task web client. (In the web clients, the command **Terminate Order** moves the order to the Aborted order state.) You can also transition to the Aborted order state programmatically by using the OSM Web Service API or by using an automated task.

The Aborted order state is a final state; the order has been permanently stopped. An order in the Aborted state cannot transition to another state.

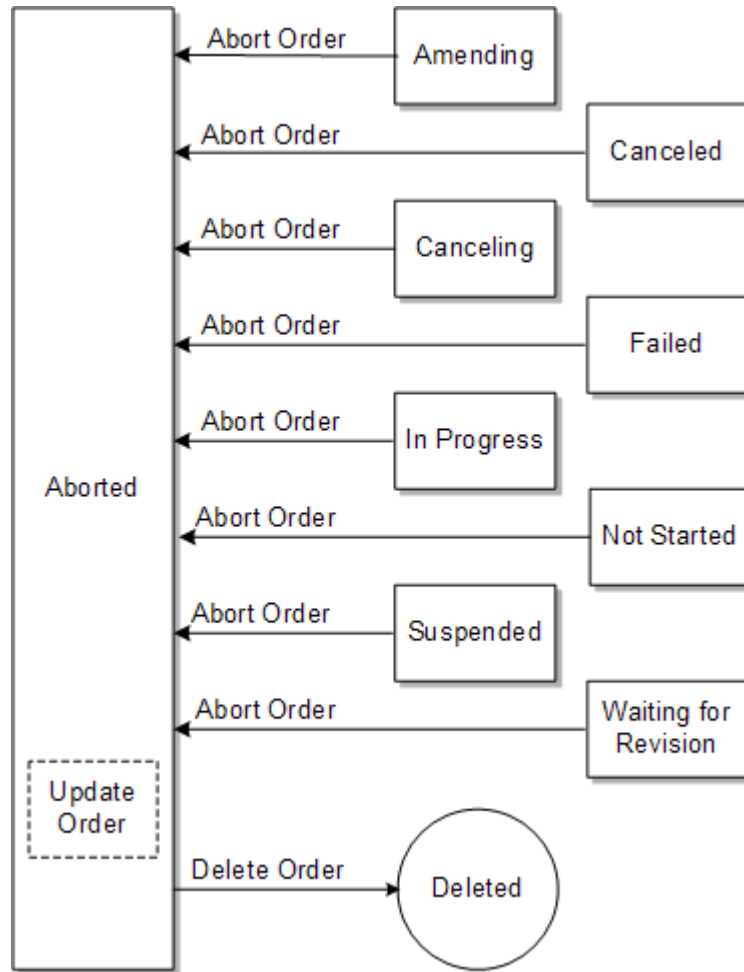
Terminated orders may require manual intervention in an OSM web client to compensate for tasks that have completed or that were in the process of completing. For example, you may be required to release port assignments, delete accounts in billing systems, and so forth.

The entrance transaction to the Aborted order state is Abort Order. This transaction can be run from all order states except the Completed order state.

The exit transaction from the Aborted state is Delete Order, which removes the order from the OSM system.

The Update Order transaction is used when the order is updated manually, outside of the order processing.

[Figure 3-6](#) shows the order states that can transition to or from the Aborted order state.

Figure 3-6 Order States that Can Transition to or from the Aborted Order State

About the Amending Order State

An order in the Amending state is undergoing compensation.

The transactions that cause an order to move to the Amending state are the Submit Amendment transaction (as a result of a revision order) and the Raise Exception transaction (as a result of fallout for which compensation is needed). The order can be amended from the following order states:

- In Progress
- Failed
- Suspended
- Waiting for Revision

To transition an order to the Amending state, OSM uses two transactions: Submit Amendment and Process Amendment. These transactions work together to make sure that the order is in a condition that can be amended and that the amendment is allowed.

Each revision to an order uses the Submit Amendment transaction to place the amendment in a queue. The Submit Amendment transaction does not change the order state. Instead, it makes sure that the order is ready to be amended and that there are no life-cycle rules that

prevent the order from being amended until a condition is met. For example, although an order in the Suspended state can receive amendments from the Submit Amendment transaction, the order must be resumed before it can process the amendments.

When the order is able to process the amendment, the Process Amendment transaction is run on the latest amendment in the queue, and the transition is made to the Amending state. Not every order in the queue is processed:

- A revision for the same order might have been received while the order is queued. In that case, the later revision is used instead.
- Restrictions in the life-cycle policy might prevent an amendment from being processed by the Process Amendment transaction.

Unless multiple revisions are common and frequent, the order state transition to Amending will happen almost immediately after the Submit Amendment transaction.

The configurable exit transactions for the Amending state are:

- Submit Amendment: An order can process a Submit Amendment transaction while the order is in the Amending state. This can occur because additional revision orders can be submitted while the order is in the Amending state. In this case, the Submit Amendment transaction adds the amendment to the amendment queue.
- Suspend Order: Transitions to the Suspended state.
- Abort Order: Transitions to the Aborted state.

An order can transition from the Amending state to the In Progress state, but there is no transaction involved. This transition is handled internally by OSM.

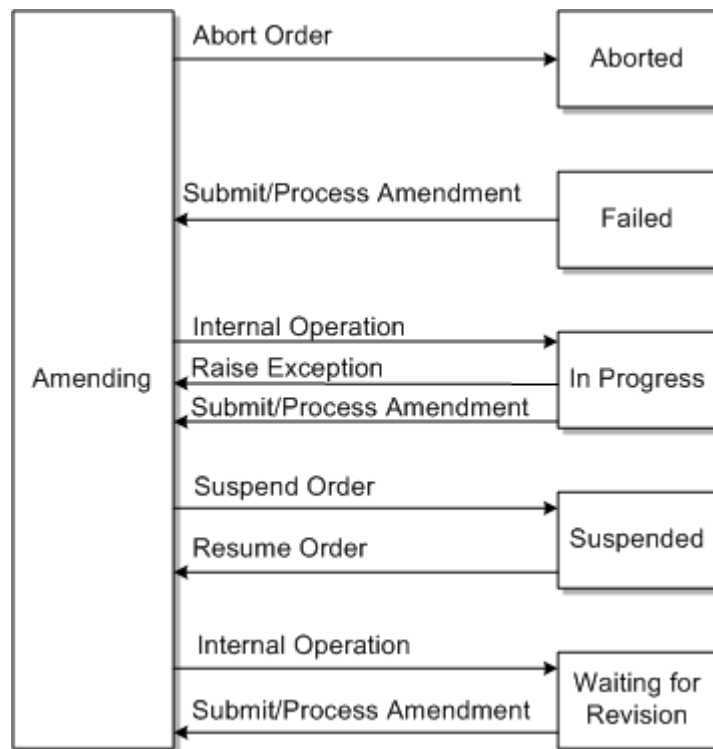
An order can transition from the Amending state to the Waiting for Revision state. However, there is no transaction required to transition from the Amending state to the Waiting for Revision state. This transition happens when fallout occurs, and OSM has found that the fallout is caused by the submitted order. In that case, OSM cannot use further compensation (redo/undo) to fix the problem. Instead, OSM waits for a revision to be submitted from the upstream order-source system to fix the problem.

You can also enable the Manage Order Fallout transaction for this state that controls the following for managing tasks in the failed state:

- RetryOrder and ResolveFailure OSM Web Service operations
- Retry Order and Resolve Order Failure Order Management web client actions

See *OSM Developer's Guide*, *OSM Order Management Web Client User's Guide*, and *OSM Task Web Client User's Guide* for more information.

[Figure 3-7](#) shows the order states that can transition to or from the Amending order state.

Figure 3-7 Order States that Can Transition to or from the Amending Order State

About the Cancelled Order State

When an order is in the Cancelled state, all tasks have been undone back to the creation task.

The actions allowed when an order is in the Cancelled state are different depending on if the order has an orchestration plan:

- If an order has an orchestration plan, the Cancelled state is the final state. The order cannot be resumed.
- If the order does not have an orchestration plan, the order can be resumed at the In Progress state, either by manually opening the order at the creation task and submitting it or by programmatically transitioning the order state using the OSM APIs.

The transaction that causes the Cancelled state is the same Cancel Order transaction that was used for canceling the order.

If the order includes an orchestration plan, the configurable exit transactions are:

- Update Order: Allows the order data to be changed but does not transition the order to another order state.
- Abort Order: Transitions to the Aborted state.
- Delete Order: Removes the order from the OSM system.

If the order does not have an orchestration plan, the configurable exit transactions are:

- Resume Order: Transitions to the In Progress state.
- Update Order: Allows the order data to be changed. This transaction can also transition the order to the In Progress state if the **startOrder** option is used. See the discussion of the Update Order transaction in [Table 3-2](#) for more information.

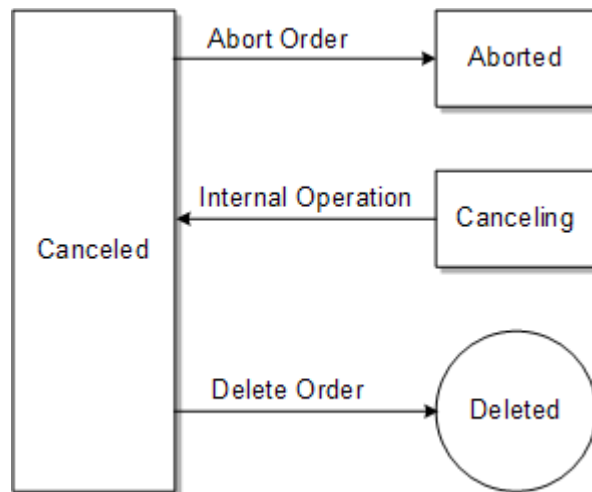
- Abort Order: Transitions to the Aborted state.
- Delete Order: Removes the order from the OSM system.

Note

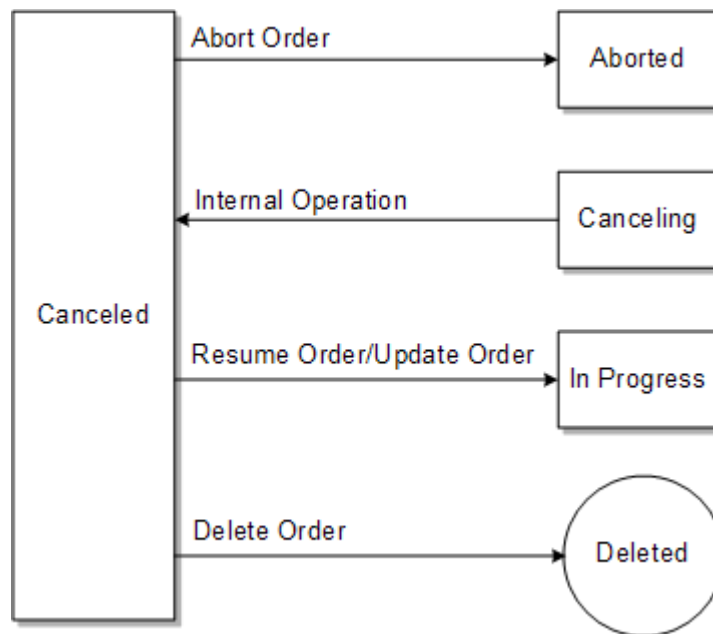
When resumed after being canceled, the order begins again at the beginning of the execution; it is not resumed at the point in the execution it was in when canceled.

[Figure 3-8](#) shows the order states that can transition to or from the Cancelled order state if the order has an orchestration plan.

Figure 3-8 Order States that Can Transition to or from the Cancelled Order State if the Order Has an Orchestration Plan



[Figure 3-9](#) shows the order states that can transition to or from the Cancelled order state if the order does not have an orchestration plan.

Figure 3-9 Order States That Can Transition To Or From the Aborted Order State if the Order Does Not Have an Orchestration Plan

About the Cancelling Order State

When an order is in the Cancelling state, at least one live task is running in a cancellation compensation mode. OSM undoes all completed tasks to return the order to the creation task. When OSM has finished, the order transitions to the Cancelled state.

The entrance transaction for the Cancelling order state is the Cancel Order transaction. An order can be canceled from the following order states:

- In Progress
- Completed
- Suspended
- Waiting
- Waiting for Revision
- Failed

The configurable exit transactions for the Cancelling order state are:

- Suspend Order: Transitions to the Suspended state.
- Abort Order: Transitions to the Aborted state.

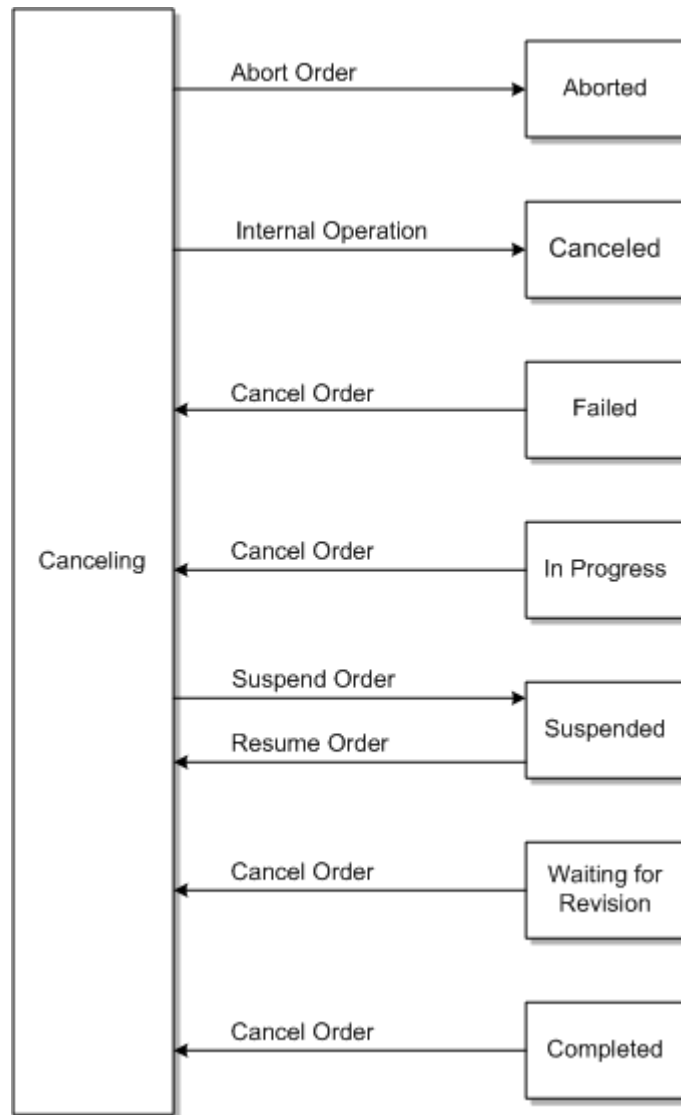
You can also enable the Manage Order Fallout transaction for this state that controls the following for managing tasks in the failed state:

- RetryOrder and ResolveFailure OSM Web Service operations
- Retry Order and Resolve Order Failure Order Management web client actions

See *OSM Developer's Guide*, *OSM Order Management Web Client User's Guide*, and *OSM Task Web Client User's Guide* for more information.

[Figure 3-10](#) shows the order states that can transition to or from the Cancelling order state.

Figure 3-10 Order States That Can Transition To Or From the Cancelling Order State



About the Completed Order State

The order has been fulfilled. There are no live tasks and processing is complete.

The entrance transaction for the Completed state is the Complete Task transaction. It transitions from the In Progress state.

The Complete Task transaction is used internally whenever the last task is completed in the order, which is determined automatically by OSM. Therefore the Complete Task transaction is not shown as part of the life-cycle policy in Design Studio.

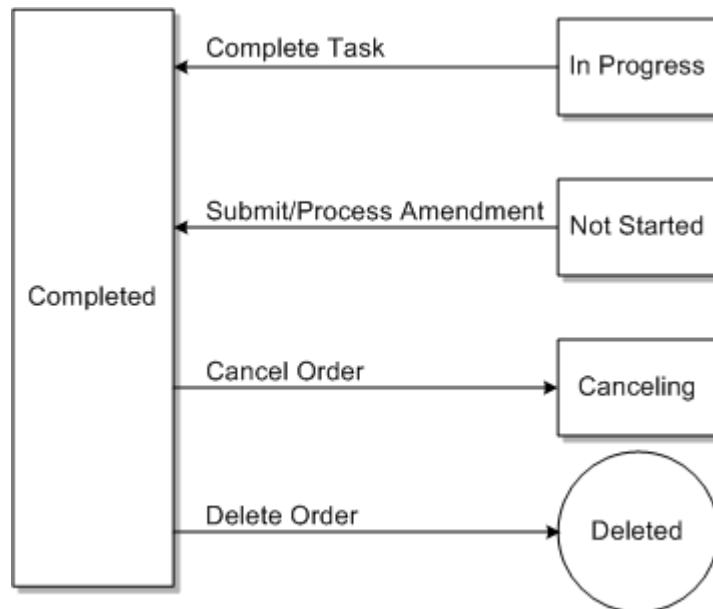
The transition from the Not Started state to the Completed state is specific to revision orders. When a revision order that has been submitted and accepted transitions to the Completed state directly, because the compensation for the revision happens on the base order being revised.

The configurable exit transactions for the Completed order state are:

- Delete Order: Removes the order from the OSM system.
- Update Order: Allows the order data to be added, changed, or deleted but does not transition the order to another order state.
- Cancel Order: Allows the order to be canceled.

[Figure 3-11](#) shows the order states that can transition to or from the Completed order state.

Figure 3-11 Order States that Can Transition to or from the Completed Order State



About the Failed Order State

If an order is the Failed state, the order failed during fulfillment, after the order was submitted by the order-source system or during order recognition when validating the incoming order data.

The entrance transaction for the Failed order state is the Fail Order transaction. An order can transition to the Failed state from the following states:

- Not Started
- In Progress
- Suspended
- Waiting for Revision

The configurable exit transactions for the Failed order state are:

- Manage Order Fallout: Transitions back to the state that the order was in when the Fail Order transaction occurred. For example, if the order was in the Not Started state and then failed, the Manage Order Fallout transaction returns the order to the Not Started state. It can exit to the following states:
 - Not Started

- In Progress
- Waiting for Revision
- Suspend Order: Transitions to the Suspended state.
- Update Order: Allows the order data to be added, changed, or deleted but does not transition the order to a different order state.
- Submit Amendment/Process Amendment: Submits an amendment and is followed by the Process Amendment transaction and transitions the order to the Amending state.
- Cancel Order: Transitions to the Cancelling state.
- Abort Order: Transitions to the Aborted state.

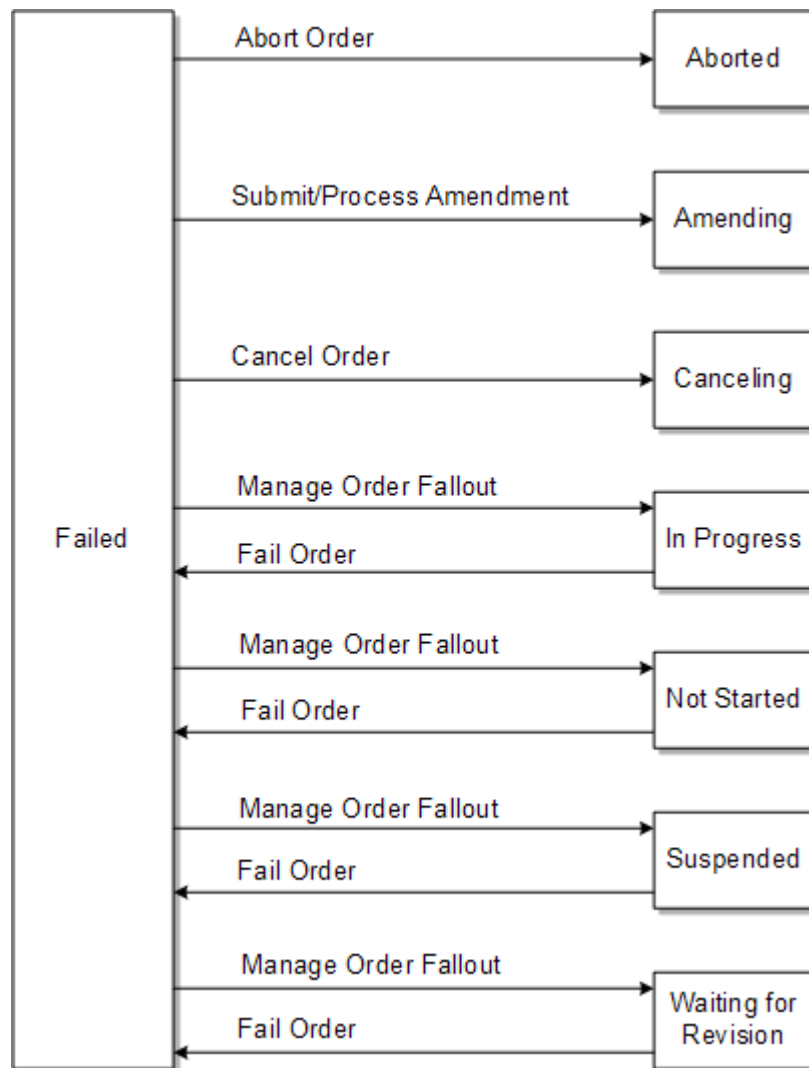
You can also enable the Manage Order Fallout transaction for this state that controls the following for managing tasks in the failed state:

- RetryOrder and ResolveFailure OSM Web Service operations
- Retry Order and Resolve Order Failure Order Management web client actions

See *OSM Developer's Guide*, *OSM Order Management Web Client User's Guide*, and *OSM Task Web Client User's Guide* for more information.

[Figure 3-12](#) shows the order states that can transition to or from the Failed order state.

Figure 3-12 Order States that Can Transition to or from the Failed Order State



About the In Progress Order State

An order in the In Progress state is actively running. Future-dated orchestration orders have an In Progress state while they wait for dependencies to be resolved.

The entrance transactions for the In Progress state are:

- Update Order: Transitions from the Not Started state or Cancelled state when the **startOrder** option is used. Programmatic creation of an order typically begins the execution of the order, transitioning it to the In Progress order state when the **startOrder** option is set to **true** on the CreateOrder or CreateOrderBySpecification OSM Web Service operation. See the discussion of the Update Order transaction in [Table 3-2](#) for more information.
- Resume Order: Transitions from the following states:
 - Suspended
 - Waiting for Revision
 - Canceled

 Tip

The Cancelled state returns the order to the creation task, so the Resume Order transaction does not resume from the state it was in when canceled. Instead, it resumes at the beginning of the process.

- Manage Order Fallout: Transitions from the Failed state.

An order can transition from the Amending state to the In Progress state, but there is no transaction involved. This transition is handled internally by OSM.

The exit transactions for the In Progress order state are:

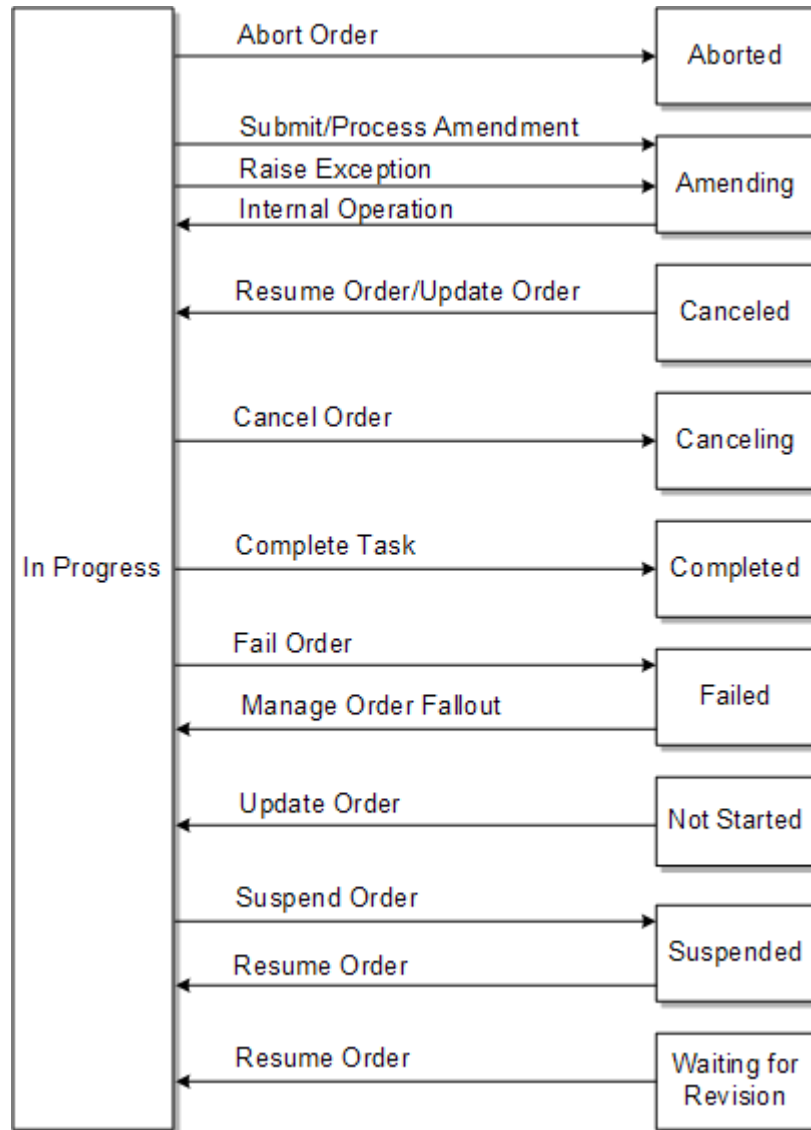
- Update Order: Allows the order data to be added, changed, or deleted.
- Submit Amendment/Process Amendment: Submits an amendment (typically from an external CRM system) and is followed by the Process Amendment transition. Transitions to the Amending state.
- Suspend Order: Transitions to the Suspended state.
- Cancel Order: Transitions to the Cancelling state.
- Fail Order: Transitions to the Failed state.
- Abort Order: Transitions to the Aborted state.
- Raise Exception: The Raise Exception transaction is a special type of transaction from the In Progress state. For order fallout scenarios, the Raise Exception transaction can transition the order to the Amending state to perform compensation for the error. However, for backward compatibility with orders that use process exceptions, the Raise Exception transactions starts an exception handling process, but the order remains in the In Progress state. See the discussion of the Raise Exception transaction in [Table 3-2](#) for more information.
- Complete Task: Transitions from the In Progress state, but only when the last task in the order is completed. This transaction is also used internally whenever a task is completed in the order. It is not shown in the life cycle display in Design Studio.

You can also enable the Manage Order Fallout transaction for this state that controls the following for managing tasks in the failed state:

- RetryOrder and ResolveFailure OSM Web Service operations
- Retry Order and Resolve Order Failure Order Management web client actions

See *OSM Developer's Guide*, *OSM Order Management Web Client User's Guide*, and *OSM Task Web Client User's Guide* for more information.

[Figure 3-13](#) shows the order states that can transition to or from the In Progress order state.

Figure 3-13 Order States That Can Transition To Or From the In Progress Order State

About the Not Started Order State

When an order is in the Not Started state, the order has been created but has not started. There are no live tasks other than the creation task.

The entrance transactions for the Not Started state are:

- **Resume Order**: Transitions from the Suspended state if the order was in the Not Started state when it was Suspended.
- **Manage Order Fallout**: Transitions from the Failed state if the order was in the Not Started state when the Fail Order transaction occurred.

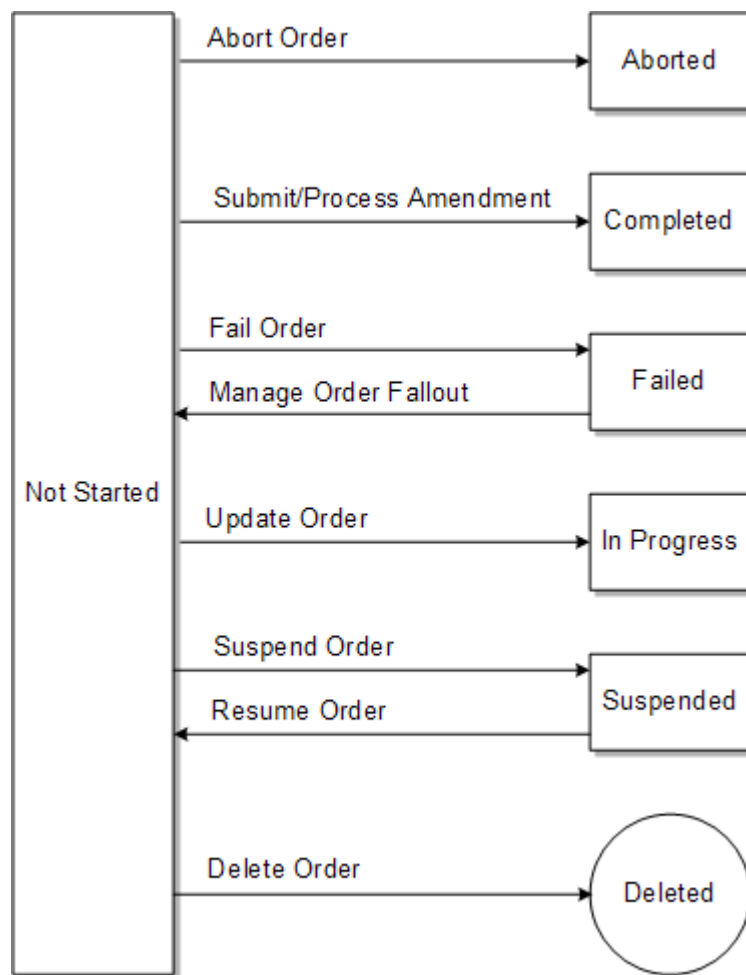
The exit transactions for the Not Started state are:

- **Update Order**: Allows the order data to be added, changed, or deleted. Can also transition the order to the In Progress state if the **startOrder** option is used. See the discussion of the Update Order transaction in [Table 3-2](#) for more information.

- Suspend Order: Transitions to the Suspended state.
- Fail Order: Transitions to the Failed state.
- Abort Order: Transitions to the Aborted state.
- Submit/Process Amendment: Transitions to the Completed state. This transition is specific to revision orders. When a revision order is submitted, if it is accepted it transitions to the Completed order state directly, because the compensation for the revision happens on the base order being revised.
- Delete Order: Removes the order from the OSM system.

[Figure 3-14](#) shows the order states that can transition to or from the Not Started order state.

Figure 3-14 Order States that Can Transition to or from the Not Started Order State



About the Suspended Order State

In the Suspended state, all processing on the order has been halted. No task can be updated or transitioned.

The only entrance transaction for the Suspended state is the Suspend Order transaction. Orders can be suspended from the following order states:

- Not Started
- Failed
- Canceling
- In Progress
- Amending

The exit transactions for the Suspended order state are:

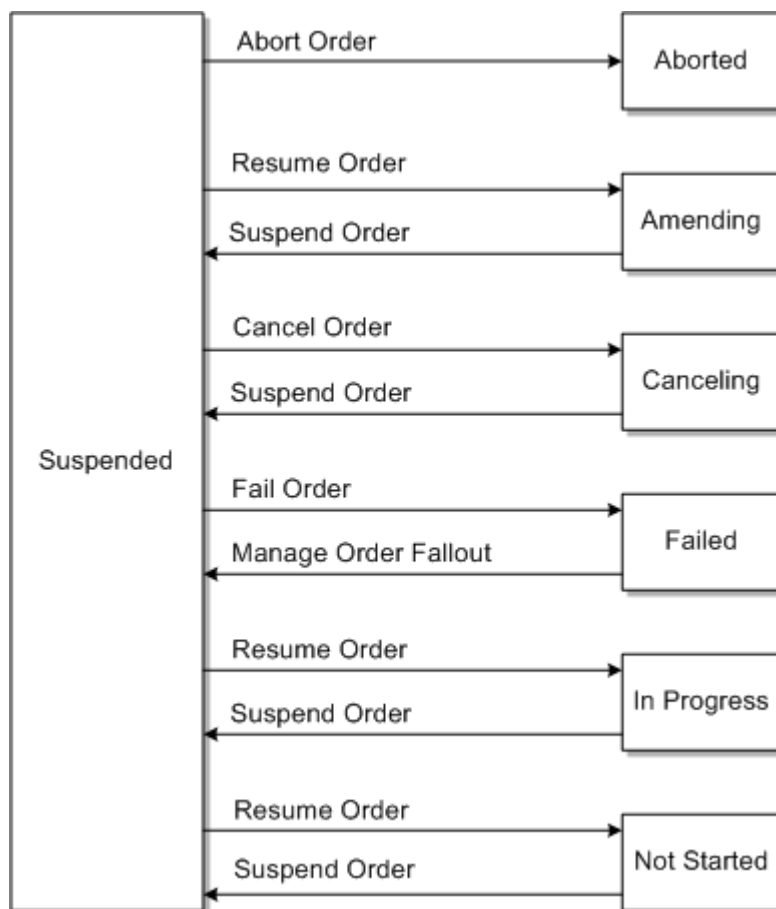
- Resume Order: Transitions the order to the state that it was in when it was suspended.
- Submit Amendment: Submits an amendment (typically from an external CRM system) to the amendment queue. Typically, the Submit Amendment transaction is followed by the Process Amendment transaction, which transitions the order to the Amending state. However, an order in the Suspended state must be resumed with the Resume Order transaction before amendments can be processed. After the order is resumed, the Process Amendment transaction is run on the latest amendment in the queue and the order transitions to the Amending state.
- Update Order: Allows the order data to be added, changed, or deleted.
- Cancel Order: Transitions to the Cancelling state.
- Fail Order: Transitions to the Failed state.
- Abort Order: Transitions to the Aborted state.

You can also enable the Manage Order Fallout transaction for this state that controls the following for managing tasks in the failed state:

- RetryOrder and ResolveFailure OSM Web Service operations
- Retry Order and Resolve Order Failure Order Management web client actions

See *OSM Developer's Guide*, *OSM Order Management Web Client User's Guide*, and *OSM Task Web Client User's Guide* for more information.

[Figure 3-15](#) shows the order states that can transition to or from the Suspended order state.

Figure 3-15 Order States That Can Transition To Or From the Suspended Order State

About the Waiting Order State

This state indicates orders that have been created but are not ready to start. The reasons orders can enter this state are:

- The order is future-dated
- The order is a follow-on order whose predecessor has not completed
- The order is subject to inter-order dependencies that have not completed

The Waiting order state is usually entered from the Not Started state and transitions to the In Progress state when the blocking condition listed above has been resolved, for example the start date for a future-dated order has been reached.

An order can transition from the Amending state to the In Progress state, but there is no transaction involved. This transition is handled internally by OSM.

The exit transactions for the In Progress order state are:

- Update Order: Allows the order data to be added, changed, or deleted.
- Submit Amendment/Process Amendment: Submits an amendment (typically from an external CRM system) and is followed by the Process Amendment transition. Transitions to the Amending state.
- Suspend Order: Transitions to the Suspended state.

- Cancel Order: Transitions to the Cancelling state.
- Fail Order: Transitions to the Failed state.
- Abort Order: Transitions to the Aborted state.
- Raise Exception: The Raise Exception transaction is a special type of transaction from the In Progress state. For order fallout scenarios, the Raise Exception transaction can transition the order to the Amending state to perform compensation for the error. However, for backward compatibility with orders that use process exceptions, the Raise Exception transactions starts an exception handling process, but the order remains in the In Progress state. See the discussion of the Raise Exception transaction in [Table 3-2](#) for more information.
- Complete Task: Transitions from the In Progress state, but only when the last task in the order is completed. This transaction is also used internally whenever a task is completed in the order. It is not shown in the life cycle display in Design Studio.

The entrance transactions for the Waiting order state are:

- Resume Order: Transitions from the Suspended state.
- Resolve Failure: Transitions from the Failed state.

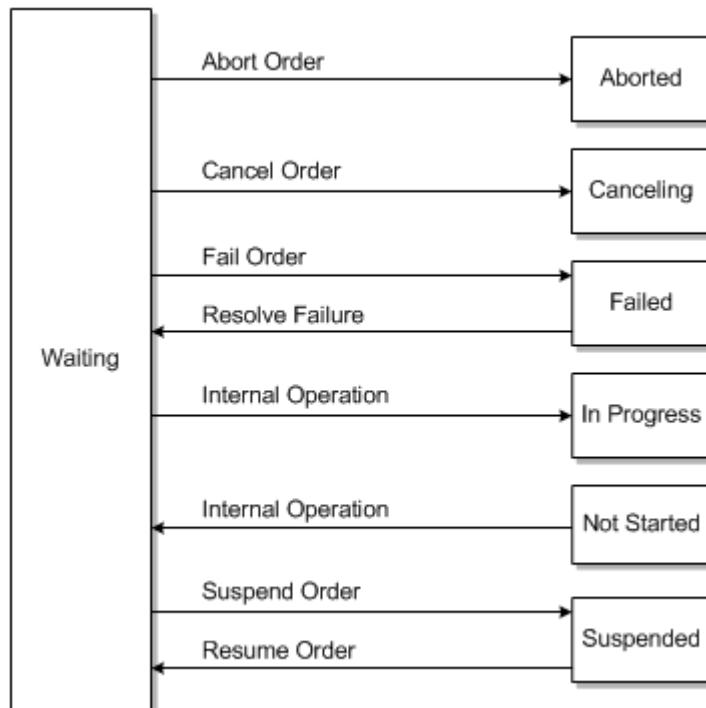
An order can transition from the Not Started state to the Waiting state when the order is ready for processing, but is either future-dated or blocked by another order as described earlier in this section.

The exit transactions for the Waiting order state are:

- Suspend Order: Transitions to the Suspended state.
- Cancel Order: Transitions to the Cancelling state.
- Fail Order: Transitions to the Failed state.
- Abort Order: Transitions to the Aborted state.

An order can transition from the Waiting state to the In Progress state when the future date is reached or the blocking by another order is resolved.

[Figure 3-16](#) shows the order states that can transition to or from the Waiting order state.

Figure 3-16 Order States that Can Transition to or from the Waiting Order State

About the Waiting for Revision Order State

This state is common following compensation to an order for fallout, when the order is awaiting a revision from the order-source system to correct something that caused a failure in the originally submitted order.

The entrance transaction for the Waiting for Revision order state is the Manage Order Fallout transaction, which runs from the Failed state.

An order can transition from the Amending state to the Waiting for Revision state. However, there is no transaction required to transition from the Amending order state to the Waiting for Revision order state. This internal transition is triggered by the Raise Exception transaction and it happens when fallout occurs and OSM has found that the fallout is generated by the submitted order instead of by a task in the process. Therefore, OSM cannot use compensation (redo/undo) to fix the problem. Instead, OSM waits for a revision to be submitted from upstream to fix the problem.

The exit transactions for the Waiting for Revision order state are:

- **Submit Amendment/Process Amendment:** Submits an amendment (typically from an external CRM system) and is followed by the Process Amendment transition. Transitions to the Amending state.
- **Update Order:** Allows the order data to be added, changed, or deleted.
- **Resume Order:** Transitions to the In Progress State
- **Cancel Order:** Transitions to the Cancelling state.
- **Fail Order:** Transitions to the Failed state.
- **Abort Order:** Transitions to the Aborted state.

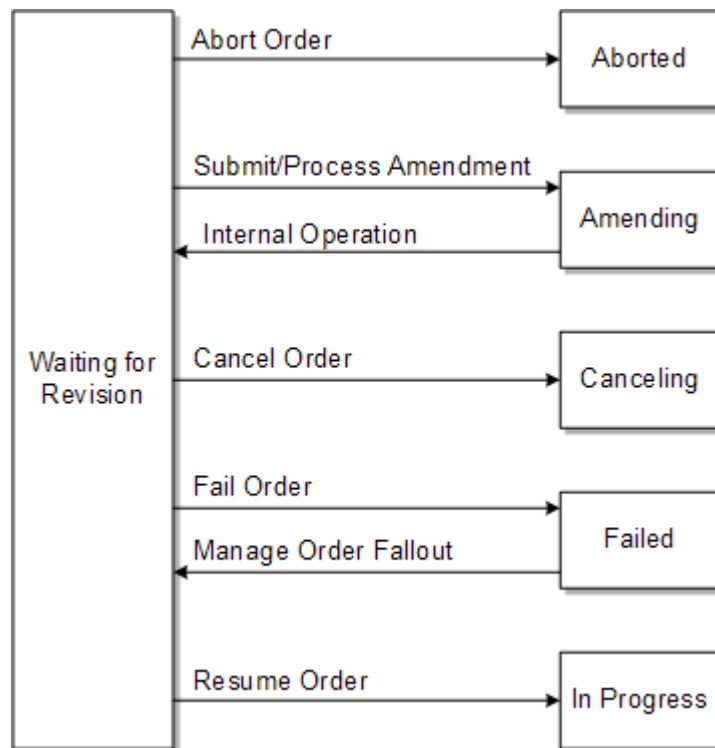
You can also enable the Manage Order Fallout transaction for this state that controls the following for managing tasks in the failed state:

- RetryOrder and ResolveFailure OSM Web Service operations
- Retry Order and Resolve Order Failure Order Management web client actions

See *OSM Developer's Guide*, *OSM Order Management Web Client User's Guide*, and *OSM Task Web Client User's Guide* for more information.

[Figure 3-17](#) shows the order states that can transition to or from the Waiting for Revision order state.

Figure 3-17 Order States that Can Transition to or from the Waiting for Revision Order State



About Deleting Orders

You cannot use either of the OSM web clients or any web service operation to delete orders from the OSM system. Instead, use the `orderPurge` command. See *OSM System Administrator's Guide* for more information.

Modeling Order Recognition

This chapter describes how to model order recognition rules to receive incoming orders from external systems in an Oracle Communications Order and Service Management (OSM) solution.

About Sending Orders to OSM and Order Recognition

This section describes the order capture and submission process and how OSM recognizes and resolves various incoming orders from CRM systems to specific order types.

The following process flow is for a new order:

1. The order data is captured in a CRM system; for example, as a Siebel order. There are several order types that OSM can process (see "[About Order Types](#)" for more information). Before submitting the order to OSM, the CRM system usually performs validations, such as validating customer information from its customer database. For some orders, the order may require technical qualification, such as validating that the network has enough capacity to offer the purchased products.
2. The CRM system sends the customer order to OSM by using the OSM CreateOrder Web Service operation. The CreateOrder operation contains order data that is in the XML format of the order-source system, which is different from the OSM order format (see "[Modeling OSM Data](#)").

The OSM Web Service API is the primary API for external clients that you can use to communicate with OSM (see *OSM Developer's Guide* for more information).

Note

A single OSM instance can receive orders from multiple order-source systems.

3. The OSM order request processor receives the customer order and evaluates the order against order recognition rules until the order request processor finds an order recognition rule that matches the incoming customer order. Then the order request processor uses the order recognition rules to transform the requests to the OSM internal order format before creating the order.
4. After OSM has recognized and validated the incoming customer order, internally, the OSM recognition rule calls the CreateOrderBySpecification web service operation. This operation does the following:
 - Creates the order in OSM
 - Sets the order priority
 - Populates the data in the creation task

Note

You can also send a `CreateOrderBySpecification` operation directly from external client systems in the OSM native XML format that would bypass the order recognition and transformation functionality. The `CreateOrderBySpecification` operation references an order specification that you define in Oracle Communications Service Catalog and Design - Design Studio, and the order details must conform to that order specification.

5. If OSM is unable to create the order by using the `CreateOrderBySpecification` operation, the inbound order is handled in one of two ways:
 - If the order type is not valid, a failed order is created with the inbound order attached.
 - If the order type and source are valid, the inbound order is put on the JMS redelivery queue. OSM attempts to receive the order again, up to the receive limit configured for the queue. When that limit is reached, the failed message is moved to an error queue.

To receive and create orders, you need to do the following:

- Configure your order-source system to output orders in XML format (see "[Modeling OSM Data](#)").
- Do the following in Design Studio:
 - Populate the Data Dictionary with the data elements that the order needs. See "[Modeling OSM Data](#)" for more information.
 - Create recognition rules to recognize, validate, and transform the data.
 - Create order specifications for the types of orders you need to create in OSM. See "[Modeling Orders and Permissions](#)" for more information.

Incoming orders can use the process layer or the orchestration layer. See "[Modeling Orders and Permissions](#)" for more information.

Note

An order can be created without recognition rules and without an orchestration plan. This is common when the order has a limited set of tasks that do not have dependencies; for example, an order that only manages service activation.

Modeling Order Recognition Rules

You model order recognition rules to accept, evaluate, and transform OSM Web Service API `CreateOrder` requests. The content of every `CreateOrder` request must match a specific order recognition rule that associates the incoming order with a target OSM order specification. If you have more than one version of the target order specification, you can target a specific version of the order specification.

During order recognition, OSM reviews a prioritized list of recognition rules to determine which rule applies to the inbound order. Each recognition rule is associated with an order specification. OSM evaluates each order based on how high the relevancy of a particular order recognition rule is. For example, a recognition rule with a relevancy of 12 evaluates first, then the rule with the relevancy of 11, then 10, and so on. An order recognition rule with a 0

relevancy should be modeled as a catch-all recognition rule. See "[Modeling a Catch-All Recognition Rule](#)" for more information.

Validating Incoming Order Data

You can model a validation rule to validate any number of things in the order data. For example, you can ensure that:

- All mandatory fields are populated on the incoming order.
- Valid characters (numeric or alphanumeric) are used for fields.
- The order has a valid status code, such as Open.

Transforming Order Data

OSM provides the following transformation rules in an order recognition rule:

- order priority rules: define the priority of the order in relation to others.
- order reference rules: define the order reference number.
- order data rules: add to or modify incoming customer order data.

At run time, the OSM server always runs all transformation rules, regardless of the failure of any transformation rule. Running all transformation rules ensures that the order is populated with all available data.

If a transformation rule fails, the order is populated with whatever data is available, and the order is placed in a Failed state with reasons corresponding to the type of transformation rule that failed:

- Could not set order priority.
- Could not set order header reference.
- Could not create order data.
- Could not store incoming message. Message stored as attachment.

Modeling the Order Data Rule to Populate the Creation Task

An internal transformation rule always stores the raw XML input message in an XML data field as part of the order data (see "[Adding the Input Message to an Order Recognition Rule](#)" and "[Adding the Input Message to the Order Template](#)"). However, that data does not populate the fields in the creation task.

You can use an order data rule to modify data in the order. For example, you can concatenate the area code and phone number into a single data element.

You can retrieve data from external systems if it does not exist on the incoming customer order using a data instance behavior associated with the order data rule (see "[Evaluating Data Instance Behaviors](#)"). For example, the incoming customer order might have a customer address, but you need to add the geographic region to the order, which is not in the input data. You can use a web service operation, or an SQL call to an external system, to look up the region, based on the customer's address. You can then add the region code to the order.

When modeling a creation task, create a manual task, even if the order is intended to be processed automatically. Using manual tasks as creation tasks ensures that task behaviors are supported at run time if you manually create an order. This can be useful for testing purposes.

Modeling Order Priority

The order priority range specifies the acceptable range of numeric priority (between 0 and 9) that orders of a single type may use. For example, this could allow you to configure a fixed-line order type with a lower range (0 to 4) and a mobile order type with a higher priority range (5 to 9), ensuring that mobile orders are prioritized higher than fixed-line orders.

You create an order priority range by specifying a minimum and maximum priority for the order. OSM rounds priority values up or down to ensure they conform to the order priority range. For example, if you specify a priority range of 5 to 7 and an order is created with a priority of less than 5, the system assumes the intent is to provide the lowest priority allowed for the order, and the priority value of the order is set to 5. Similarly, if a priority higher than 7 is provided for another order of the same type, the system assumes the intent is to provide the highest priority allowed for the order, and the priority value of the order is set to 7.

[Table 4-1](#) shows examples of how the order priority is set by using the order priority from the recognition rule, and the order priority range from the order specification.

Table 4-1 Order Priority Examples

Order Priority Range	Recognition Rule Order Priority 1	Recognition Rule Order Priority 5	Recognition Rule Order Priority 9
Order Priority Range 1 - 3	Priority = 1	Priority = 3	Priority = 3
Order Priority Range 3 - 5	Priority = 3	Priority = 5	Priority = 5
Order Priority Range 5 - 9	Priority = 5	Priority = 5	Priority = 9

You can set the order priority range in the Design Studio Order editor Details tab.

The order priority value is also considered when an order's tasks are run, so that automated tasks are run according to order priority. This requires that Java Messaging Service (JMS) message priority settings are configured for the JMS queues. For information about configuring JMS message priority on JMS queue, see "[Configuring JMS Message Priority on JMS Queue](#)".

You can change the order priority of an in-flight order by using the Order Management web client. You can specify permissions for which roles can change the priority. See the discussion of changing order priority in *OSM Order Management Web Client User's Guide*.

The automation plug-ins are run using JMS. For internal plug-ins, OSM relays Order Priority into JMSPriority and thus ensures Order Priority to take effect during the execution of plug-ins.

For external plug-ins, for Order-Priority to take effect during the execution, the external system needs to update JMSPriority in the JMS message response with the one received in the JMS message request.

Note

This is an optional activity and is relevant only when the execution of external plug-ins needs to acknowledge Order-Priority.

Configuring JMS Message Priority on JMS Queue

As messages arrive on a specific destination, by default, they are sorted in FIFO (first-in, first-out) order, which sorts the messages in the ascending order based on each message's unique JMSMessageID. However, you can use a destination key to configure a different sorting scheme based on other message properties such as JMSPriority and JMSCorrelationID for a destination. In traditional OSM, the OSM installer creates the **osmDescendingPriorityDestinationKey** destination key with **JMSPriority** as the Property and **Descending** as the Sort order. OSM cloud native comes configured with **osmDescendingPriorityDestinationKey**.

To configure JMS Message priority on JMS queue, do the following:

- Create a JMS Destination Key. See "[Creating a JMS Destination Key](#)".
- Configure a destination key for a JMS resource. See "[Configuring Destination Key for a JMS resource](#)".

Creating a JMS Destination Key

To create a JMS Destination Key:

1. In the WebLogic Remote Console, click on **Edit Tree**.
2. Navigate to **JMS Modules**, within **Services**.
3. In the **JMS Modules** table, select the JMS module that contains the configured resource and select the **Destination Keys**.
4. In the Summary of table, click the **New** button to create a destination key.
5. Enter a meaningful name for the key and click **Create**.
6. In the Summary of resources table, select the newly created JMS destination key.
7. Select the **Sort key** field and specify a message property name or the name of a message header field on which to sort messages.
8. Save the changes.
9. Click on shopping cart and **Commit Changes**.

Configuring Destination Key for a JMS resource

To configure destination key for a JMS resource:

1. In the WebLogic Remote Console, click on **Edit Tree**.
2. Navigate to **JMS Modules**, within **Services**.
3. Select the JMS module that contains the configured resource, for example: `osm_jms_module`.
4. Select the required **Queues**, **Topics**, **Uniform Distributed Queues** or **Uniform Distributed Topics** where the created destination key needs to be configured.
5. In the **General** tab, enter the required destination key exact name(s) in **Destination Key(s)** field. Make sure the destination key already exists.
6. Save the changes.
7. Click on shopping cart and select **Commit Changes**.

8. Restart the WebLogic server.

Creating and Configuring JMS Destination Key in OSM Cloud Native

You will need to provide the WDT for your OSM instance to provide the appropriate configuration for a new JMS Destination Key. See "Extending the WebLogic Server Deploy Tooling (WDT) Model" in *OSM Cloud Native Deployment Guide* for further details.

Modeling the Order Reference Number

The order reference number is an alphanumeric value supplied by the order-source system. It is usually unique, but it does not have to be unique. When OSM creates the order, OSM gives the order an OSM order ID. The original order reference number is stored as well, so the order reference number is associated with the OSM order ID.

Modeling a Catch-All Recognition Rule

An order that fails to be recognized by any recognition rule is rejected by OSM, and an error is returned by the web service operation to the order-source system. To make sure that all orders are entered into OSM, create a catch-all recognition rule that accepts all incoming customer orders.

To configure this recognition rule:

- Set the relevancy to 0, and set the relevancy for all other recognition rules higher than 0, so they are processed first.
- Include the following recognition rule XQuery:

```
fn:true()
```
- Select the **Fail Order** check box, and enter a reason. For example:

```
No valid recognition rule found.
```

Using this lowest-level recognition rule, an invalid order is recognized and then fails during validation. It then transitions to the Failed state and is kept by OSM.

Common Order Recognition Errors

There are two possible errors during order recognition:

- A recognition rule fails to run; for example, because of bad syntax. Evaluation of other rules continues.
- The inbound order is not recognized. If all recognition rules run and fail to find a match, then no OSM order can be created. This failure generates fallout, which you can view and manage as an order failure in the Order Management web client.

To avoid this kind of failure, you can create a lowest-relevancy catch-all rule that recognizes any inbound order and maps it to a default order specification. See "[Modeling a Catch-All Recognition Rule](#)" for more information.

5

Modeling Orchestration Plans

This chapter describes how to model orchestration plans in an Oracle Communications Order and Service Management (OSM) solution.

Orchestration Plan Overview

An orchestration plan includes the order items, order components, and dependencies. An order-specific orchestration plan is generated for each order that requires orchestration.

The orchestration plan for an order specifies the following:

- How order items are grouped into order components for processing
- The dependencies between the order components

In the OSM Order Management web client, you can view graphical representations of an order's orchestration plan and dependencies. You can use this information as you model orders to validate that order decomposition and orchestration plan generation is functioning as intended. The graphical representation shows exactly how an order is fulfilled.

The Order Management web client provides a graphical representation of the orchestration plan in two views:

- Orchestration plan decomposition
- Orchestration plan order item dependencies

[Figure 5-1](#) shows three orchestration stages, represented in three columns:

- Determine the fulfillment function
- Determine the fulfillment system
- Determine the processing granularity

Note

You can model any number of orchestration stages.

At each orchestration stage, the graph shows the order components created by that stage. The final column on the right shows the order components that are run as part of the orchestration plan. Each component includes a name, which is based on the orchestration stages. Components also list their included order items.

The inset in [Figure 5-1](#) shows details for three executable order components, as displayed in the orchestration plan decomposition.

Figure 5-1 Decomposition Tree

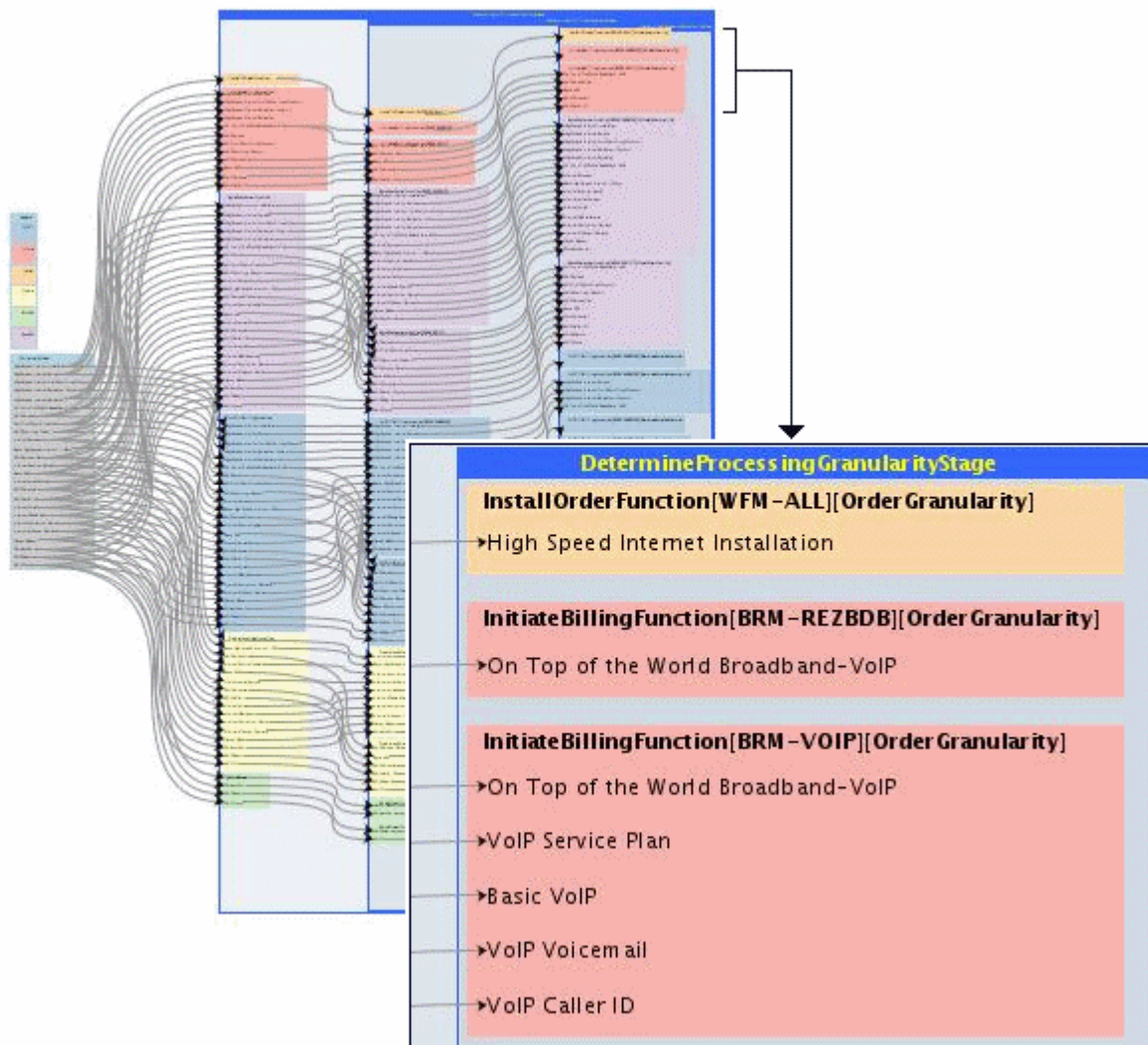
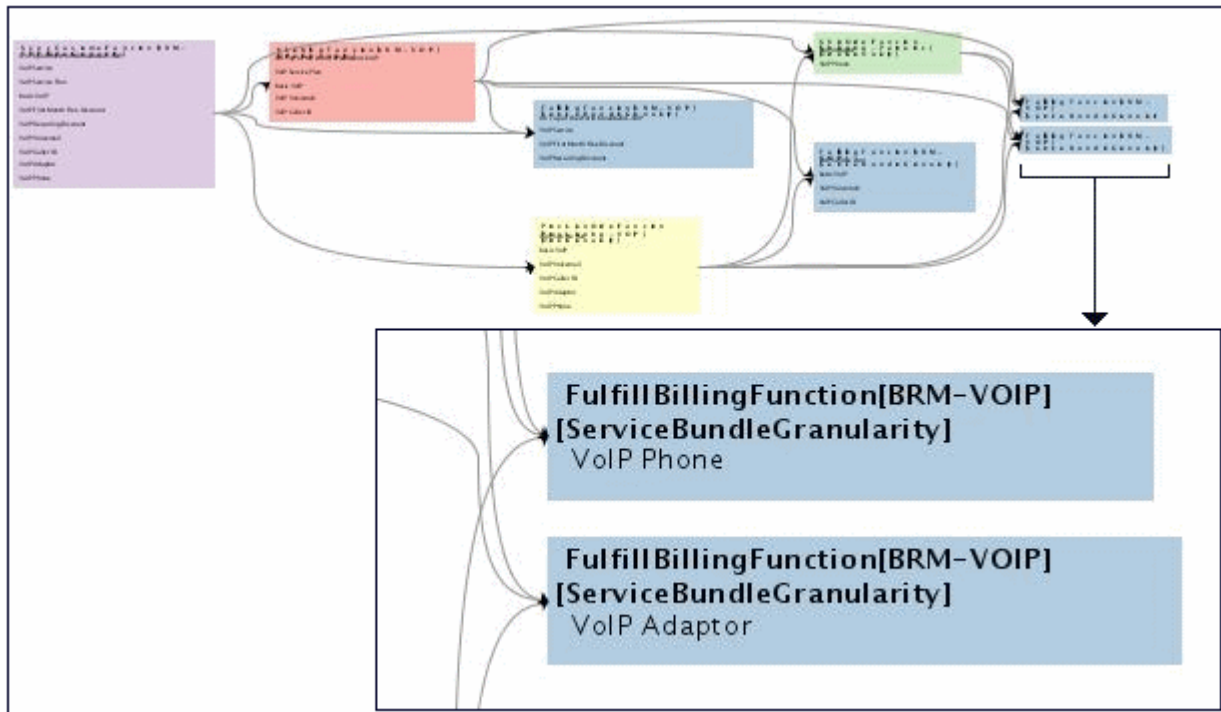


Figure 5-2 shows the orchestration plan displayed in the Order Management web client dependency graph. The dependency graph shows the executable order components which are the components shown in the final stage of the decomposition display. In this case, executable components are based on three orchestration stages corresponding to fulfillment function, fulfillment system, and processing granularity. The different colors represent fulfillment functions, such as InitiateBilling or FulfillBilling. The inset shows a detailed view of two order components. Even though the two fulfillment functions are targeted to the same system (BRM-VOIP), processing granularity rules defined for this order require that they take place as two separate actions.

Figure 5-2 Dependency Graph



Both of these representations are useful at design time and when debugging orchestration plans. For example, you can use the dependency graph to confirm that an order goes to all of the correct systems in the correct order. Use the decomposition tree to verify that decomposition happens as expected at a particular stage and that the order was decomposed into the correct components, each containing the correct order items.

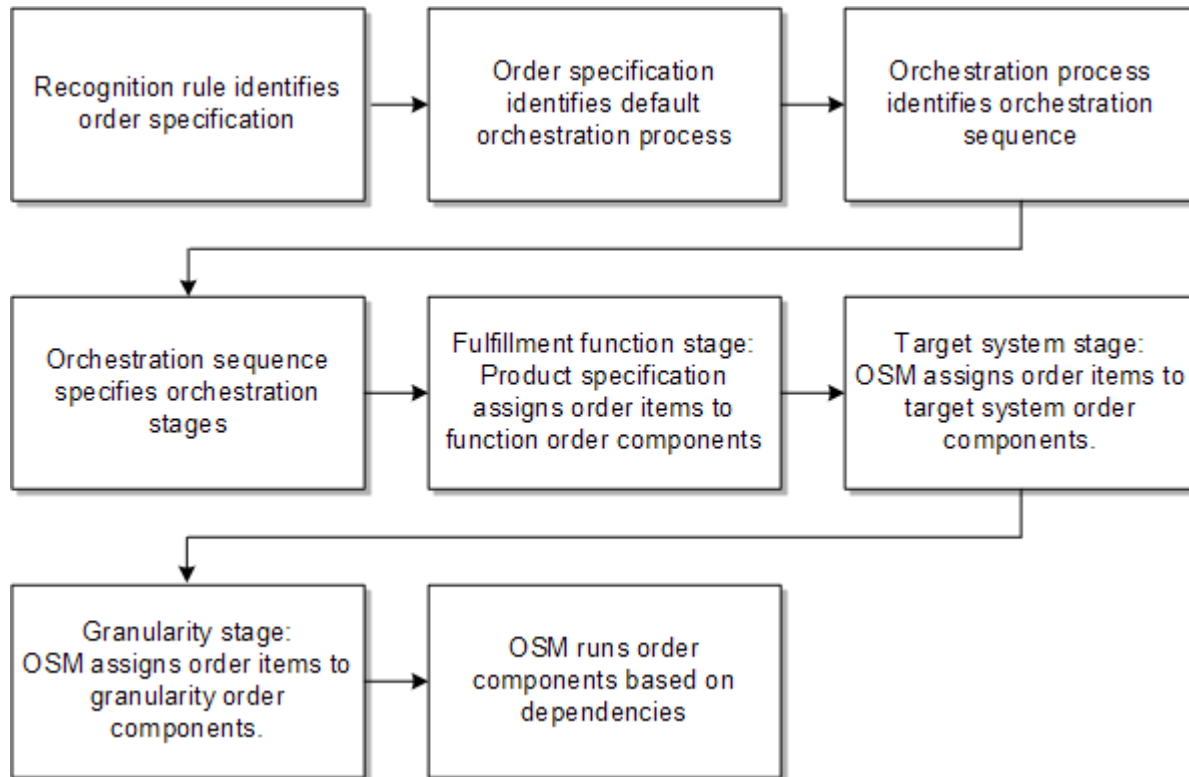
Modeling an Orchestration Plan

To model how orchestration plans are generated, you model several OSM entities in Oracle Communications Service Catalog and Design - Design Studio.

- Orchestration processes. An orchestration process specifies which orchestration sequence to use for the order.
- Orchestration sequences. The orchestration sequence specifies the fulfillment mode (for example, Deliver or Qualify), the orchestration stages, and selects the customer order line item node-sets that OSM uses in orchestration.
- Order item specification. The order item specification includes the order item properties that are used for decomposition, including how to retrieve order items from order line items. Order item properties define data that is used for decomposition; for example, the fulfillment pattern.
- Order components. Order components specify how to organize order items in the decomposition process.
- Orchestration stages. Orchestration stages specify the order components to assign order items to.

[Figure 5-3](#) shows a generalized process flow for orchestration.

Figure 5-3 Orchestration Process



The following process flow shows how OSM uses the orchestration entities to create orchestration plans.

1. After receiving and validating an incoming customer order, OSM creates the order according to the order specification chosen by the recognition rule. At this point, the following has been accomplished:
 - The order has been populated with the creation task data.
 - OSM has used the order item specification to identify order items from the order line items in the incoming customer order.
2. The order specification includes a default process. For an orchestration order, the order specifies an **orchestration process**. (If no orchestration is required, you should define a non-orchestration OSM process. See "[Modeling Processes and Tasks](#)" for more information.)
3. The orchestration process specifies an orchestration sequence.
4. The orchestration sequence specifies the following:
 - The order item specification to use for the order. The order item specification defines the order item properties that are used for decomposition and for displaying the order item in the Order Management web client. See *OSM Concepts* for more information.
 - The order item selector that identifies the customer order line item node-sets to use as order items.
 - The fulfillment mode that the order requires; for example, Deliver or Cancel.
 - The orchestration stages that produce the order components. For example, the orchestration stages might be:

- **Produce function order components.** This stage organizes order items into order components based on the fulfillment functions required for each order item. Fulfillment functions are the activities that must be performed to process the item, such as billing, shipping, provisioning, and so on.
 - **Produce target system order components.** This stage organizes order items into order components based on the target fulfillment systems required to perform the fulfillment functions. For example, this step might determine that certain items need to be fulfilled by a billing system called BRM_Residential and others by a BRM_Wholesale system.
 - **Produce granularity order components.** This stage organizes order items that need to be processed together into order components. For example, you might need to fulfill billing requirements for mobile and fixed services. You can use different order components to process the billing requirements for those services separately.
5. Each orchestration stage produces a set of order components.
 6. Based on the default orchestration process, and the orchestration sequence and stages that are defined, OSM can start the process of assigning order items to order components. The first step is to find the fulfillment pattern used by each order item.

Each order item belongs to a product specification. A **product specification** is a group of related products that share common attributes. For example, the products Broadband Light, Broadband Medium, and Broadband Ultimate would all belong to the ServiceBroadBand product specification. OSM maps the product specification to a fulfillment pattern.

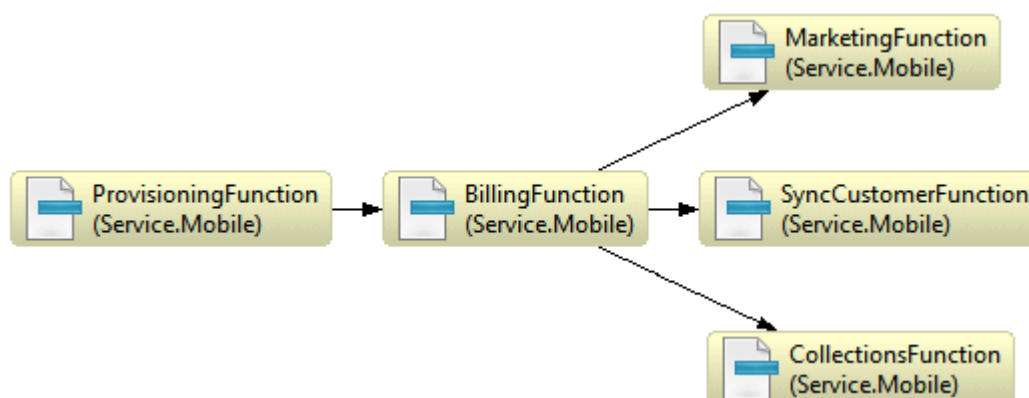
The **fulfillment pattern** manages the first stage of orchestration. It assigns order items to function order components in the first stage of orchestration. It also specifies the dependencies between the function order components. For example, the fulfillment pattern might specify to process function order components in this order:

- a. ProvisioningFunction
- b. BillingFunction
- c. CollectionsFunction

The fulfillment pattern also specifies the fulfillment mode that the order items can be used for. See "[About Mapping Order Items to Fulfillment Patterns](#)" for more information.

Provisioning must occur before billing, which must occur before marketing, customer updates (SyncCustomer), and collections.

Figure 5-4 Dependency Relationships for Order Item Dependency



7. After assigning order items to function components, OSM further decomposes the order into target system order components and granularity order components, following the specifications defined in the orchestration stages. See "[About the Decomposition of Function to Target System Components](#)" and "[About the Decomposition of Target System to Granularity Components](#)" for more information.
8. While decomposing the order, OSM finds dependencies between order components and generates an orchestration plan. Dependencies determine the order in which order components can be processed.
9. After generating the orchestration plan, OSM runs it. Each executable order component runs a process. Each process includes the tasks that fulfill the order requirements.

Order components are usually modeled by extending order component specifications in Design Studio. For example, you can create a base order component for all function types and extend it for individual function types such as billing or collections.

About Component Names and Component IDs

Each order component has an order component name and an order component ID. (This component ID is stored in the order template in `ControlData/Functions/OrderComponentName/componentKey`). The component name is specified at design time. The component ID is generated for each instance of the order component at run time.

The component name is the name of the order component specification; for example, **BillingFunction** or **BillingSystem**. By default, the component ID is a concatenation of the names of the order components in the orchestration stages. For example, if the component names are modeled as:

- BillingFunction
- BillingSystem
- Bundle

The component IDs generated at run time are:

- BillingFunction
- BillingFunction.BillingSystem
- BillingFunction.BillingSystem.Bundle

You can use customized order component IDs when assigning order items to order components. See "[About the Decomposition of Target System to Granularity Components](#)" for more information. For more information about creating valid data keys, see, "[Modeling Valid Data Keys](#)."

About Order Items

Prior to generating an orchestration plan, OSM processes each customer order line item in the incoming customer order and turns it into an order item. The **order item properties** define the data that is included from these order items using XQuery expressions.

Order items are individual products, services, and offers that need to be fulfilled as part of an order. Each item includes the action required to implement it: Add, Suspend, Delete, and so on. For example, a new order might add a wireless router; the order item created in OSM is **Add Wireless Router**.

When you model order items, you do not model every possible order item. Instead, you create an **order item specification**, which defines:

- The data that each order item can include
- The structure of the data; for example, the hierarchy between order items
- Data needed for orchestration

There must be one order item specification for each type of order received from the order-source system. When you model an order item specification, you can configure the following:

- **Order item properties.** Order item properties represent the data that is included in order items. See *OSM Concepts* for more information.
- **Orchestration conditions.** Use orchestration conditions to customize how order items are added to order components. For example, you can use the **region** order item property to assign order items to different target system order components. See "[About the Decomposition of Function to Target System Components](#)" for an example of how orchestration components are used.
- **Order item hierarchies.** You use order item hierarchies to model how parent and child items are identified. For example, you can use line IDs and parent line IDs. See "[Modeling Order Item Hierarchies](#)" for more information.
- **Order template.** This data is the order item control data, which is used by OSM when generating an orchestration plan. You can also assign behaviors to order items. See *OSM Concepts* for more information.
- **Order item dependencies.** Use order item dependencies to create inter-order dependences. See "[About Inter-Order Dependencies](#)" for more information.
- **Permissions.** Use permissions to allow specific roles access to order item search queries in the Order Management web client and to specify if the query returns summary data or detailed data. See "[Modeling Roles and Setting Permissions](#)" for more information.

Most order items properties must be created in Design Studio and associated with corresponding customer order element values using XQuery expressions (see "[About Order Item Specification Order Item Property XQuery Expressions](#)"). However, in some cases the order item property is not provided in the customer order. In this case, you must use an XQuery expression to derive the missing property value from the existing customer order element values.

[Example 5-1](#) shows an order line item. This order line item adds a Commercial Fixed Service order item. In the following example, notice that the items in bold correspond to the order item properties. However, there are order item properties, such as **productSpec** and **region**, that are not in the order line item. Instead, you specify to create those order item properties by using XQuery expressions in the order item specification.

Example 5-1 Order Line Item in an Incoming Customer Order

```
<im:salesOrderLine>
  <im:lineId>4</im:lineId>
  <im:parentLineReference>
    <im:parentLineId>3</im:parentLineId>
    <im:hierarchyName>default</im:hierarchyName>
  </im:parentLineReference>
  <im:rootParentLineId>2</im:rootParentLineId>
  <im:promotionalSalesOrderLineReference>1
</im:promotionalSalesOrderLineReference>
  <im:serviceId>552131313131</im:serviceId>
  <im:requestedDeliveryDate>2001-12-31T12:00:00</im:requestedDeliveryDate>
  <im:serviceActionCode>Add</im:serviceActionCode>
  <im:serviceAddress>
    <im:locationType>Street</im:locationType>
    <im:nameLocation>0Loughlin</im:nameLocation>
```

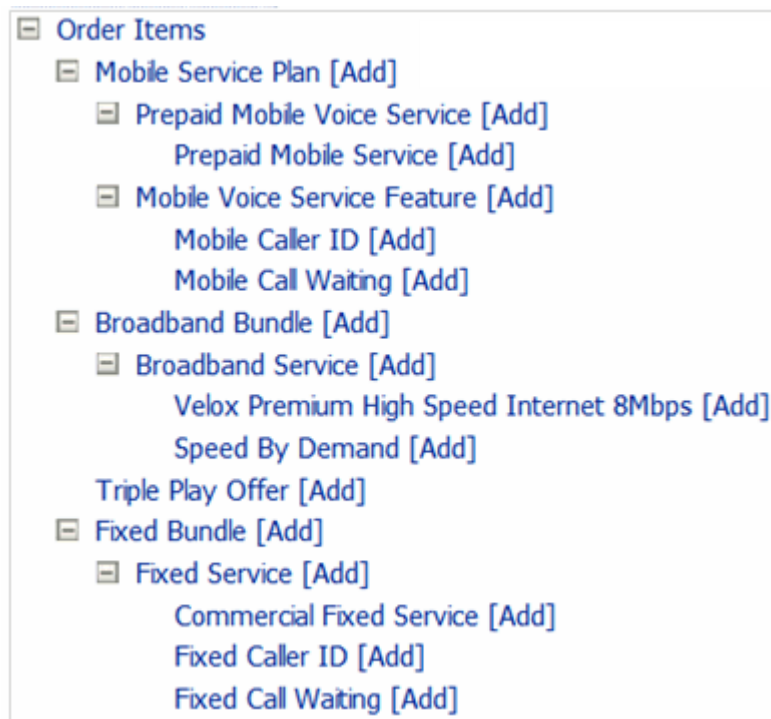
```

    <im:number>48</im:number>
    <im:city>Toronto</im:city>
  </im:serviceAddress>
  <im:itemReference>
    <im:name>Commercial Fixed Service</im:name>
    <im:typeCode>PRODUCT</im:typeCode>
    <im:primaryClassificationCode>Fixed Service Plan Class</im:primaryClassificationCode>
  </im:itemReference>
</im:salesOrderLine>

```

[Figure 5-5](#) shows all of the order items derived from an order, including the order item shown in [Example 5-1](#).

Figure 5-5 All Order Items in an Order



In [Figure 5-5](#), notice that order items are hierarchical. For example, the Fixed Service order item shown in [Example 5-1](#) is part of the Fixed Bundle order item. In addition, the Fixed Service order item includes three more order items: Commercial Fixed Service, Fixed Caller ID, and Fixed Call Waiting. When you model orchestration, you ensure that the hierarchy in the incoming customer order is duplicated in the OSM order items. See "[Modeling Order Item Hierarchies](#)" for more information.

The order item specification defines the order item properties that are required for generating the orchestration plan and the data to display in the Order Management web client. This typically includes the display name, product specification, line ID, requested delivery date, and so on. By contrast, the order item usually would not include supplementary account and customer details such as the street address or mailbox size. That type of data is defined in the task data for each task in the fulfillment data, and in the creation task data when the order is created.

⚠ Caution

Order item properties do not represent all of the data in an order. For example, they do not define creation task data. That data is captured by transformation rules. Order item properties are a subset of the data and are used for orchestration.

Figure 5-6 shows part of an order input file and how the city field is mapped to the **region** order item property in Design Studio. In this example, the **<city>** element in the XML file is used in the order item property expression.

See "[About the Decomposition of Function to Target System Components](#)" for an example of how the **region** order item property is used in orchestration.

Figure 5-6 Order Line and Definition in Order Item Specification

```

<im:serviceAddress>
  <im:locationType>Street</im:locationType>
  <im:nameLocation>Jangadeiros</im:nameLocation>
  <im:number>48</im:number>
  <im:typeCompl>floor</im:typeCompl>
  <im:numCompl>6</im:numCompl>
  <im:district>Ipanema</im:district>
  <im:codeLocation>5000</im:codeLocation>
  <im:city>Rio de Janeiro</im:city>
  <im:state>RJ</im:state>
  <im:referencePoint>Gen. Osorio Square</im:referencePoint>
  <im:areaCode>22420-010</im:areaCode>
  <im:typeAddress>Building</im:typeAddress>
</im:serviceAddress>

```

Order Item Properties

- lineId
- lineItemName
- lineItemPayload
- parentLineId
- productClass
- productSpec
- region**
- requestedDeliveryDate
- serviceId
- typeCode

Property Expression: region

Location

XQuery ☒ Instances ☐ Information

☒ Expression ☐ File ☐ URI

```

declare namespace im="http://xmlns.oracle.com/InputMessage";
fn:normalize-space(im:serviceAddress/im:city/text())

```

A single order item specification is used for generating all of the order items that can be created for an order. This ensures a consistent order item structure. Therefore:

- Order item properties should not be product or service specific. The only product information you need to include is the product specification, which is a generic value used for identifying the fulfillment pattern. By not applying order items to a specific product, you can use the order item specifications for multiple products, and you can support new products and services and multiple order entry systems.

- Order item properties should not be specific to any order entry system.

 Caution

When defining order item properties, include only the data required by OSM for orchestration. Performance is impacted by the number and size of order item properties.

The properties you define for your order items will be different from those pictured in [Figure 5-6](#). However, this selection provides a good example of the type of order properties that are commonly configured:

- **productSpec:** This property retrieves the product specification from the incoming customer order. OSM maps each order item to a fulfillment pattern based on the item's product specification (defined in the order-source system). The fulfillment pattern specifies the order components in the first level of decomposition.
- **fulfillmentPattern:** This property stores the fulfillment pattern that the order item uses. This value is obtained by mapping the **productSpec** value in a mapping file. See "[About Mapping Order Items to Fulfillment Patterns](#)" for more information.
- **lineId:** This is the line ID of the order line item in the incoming customer order. Each order line item in the incoming customer order has a unique line ID. This property is used for determining the hierarchy of the order items. You can determine a hierarchy of order items based on the lineId order item property and the parentLineId order item property. For example, an order item with lineId 4 also specifies a parentLineId as 3 which is the lineId of the parent order item. You can use this function to hierarchically relate various types of order line items, such as offers, products, and bundles of products, services, and resources. For example, an order could include a Broadband offer with a High Speed Internet bundle and an Internet Services service bundle. Both bundles would have the Broadband offer as parent. You can also use order item hierarchies to aggregate order item status. See "[About the Decomposition of Target System to Granularity Components](#)" for an example of how this property is used.
- **lineItemName:** This property is the display name used in OSM web clients.
- **requestedDeliveryDate:** This property is the requested completion date for the order item.
- **parentLineId:** This property defines the parent of the order line item in the incoming customer order. This property is used for determining the hierarchy of the order items. See "[About the Decomposition of Target System to Granularity Components](#)" for an example of how this property is used.
- **region:** This property is an example of data that can be used to manage decomposition into target system order components. See "[About the Decomposition of Function to Target System Components](#)" for more information.
- **serviceId:** This property is used to display the service ID in the OSM web clients.
- **lineItemPayload:** This property stores the entire incoming customer order in OSM as an XML file. This property is typically used in a development environment as an aid to modeling.

About Creating Order Items from Customer Order Line Item Node-Sets

To create order items from customer order line items, OSM needs to know what nodes in the incoming customer order include the data to use in order items. OSM creates orchestration control data from these nodes (see *OSM Concepts*).

[Example 5-2](#) shows the **salesOrderLine** node-set in an incoming customer order. You can specify these node-sets as order items by creating an XQuery expression in the Orchestration Sequence editor that returns every instance of **<salesOrderLine>** contained in the customer order (see "[About Order Item Specification Order Item Property XQuery Expressions](#)"). These node-sets produce the **Broadband Bundle** and the **Mobile Bundle** order items. The elements in these node-sets can then be specified as order item properties in the order item specification.

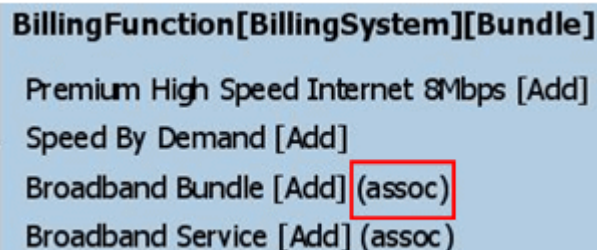
Example 5-2 The **<salesOrderLine>** Element in an Incoming Customer Order

```
<im:salesOrderLine>
  <im:lineId>13</im:lineId>
  <im:promotionalSalesOrderLineReference>1
</im:promotionalSalesOrderLineReference>
  <im:serviceId></im:serviceId>
  <im:requestedDeliveryDate>2001-12-31T12:00:00</im:requestedDeliveryDate>
  <im:serviceActionCode>Add</im:serviceActionCode>
  <im:itemReference>
    <im:name>Broadband Bundle</im:name>
    <im:typeCode>BUNDLE</im:typeCode>
    <im:specificationGroup></im:specificationGroup>
  </im:itemReference>
</im:salesOrderLine>
<im:salesOrderLine>
  <im:lineId>14</im:lineId>
  <im:promotionalSalesOrderLineReference>2
</im:promotionalSalesOrderLineReference>
  <im:serviceId></im:serviceId>
  <im:requestedDeliveryDate>2001-12-31T12:00:00</im:requestedDeliveryDate>
  <im:serviceActionCode>Add</im:serviceActionCode>
  <im:itemReference>
    <im:name>Mobile Bundle</im:name>
    <im:typeCode>BUNDLE</im:typeCode>
    <im:specificationGroup></im:specificationGroup>
  </im:itemReference>
</im:salesOrderLine>
```

About Associated Order Items

[Figure 5-7](#) shows the associated order items, displayed with **(assoc)** in the orchestration plan.

Figure 5-7 Associated Order Items Displayed in the Order Management Web Client



```
BillingFunction[BillingSystem][Bundle]
Premium High Speed Internet 8Mbps [Add]
Speed By Demand [Add]
Broadband Bundle [Add] (assoc)
Broadband Service [Add] (assoc)
```

⚠ Caution

Associated order items are not considered as part of the decomposition and dependency calculations when OSM generates an orchestration plan. Therefore, you cannot reference associated order items in decomposition or dependency rules.

You model order item associations in fulfillment patterns. [Figure 5-8](#) shows an order item association modeled for the Bundle order component in the Service.Mobile fulfillment pattern.

Figure 5-8 Order Item Associations in a Fulfillment Pattern

Orchestration Fulfillment Pattern : Service.Mobile

Description: Service.Mobile

Namespace: http://oracle.communications.ordermanagement.unsupported.centralom

Fulfillment Mode: Deliver

Order Components | Dependencies | Duration

Order Components

Base Component: All Components

Functions: All Components

Search:

- Function
- TargetSystem
- ProcessingGranularity
 - WholeOrder
 - Offer
 - Bundle**

Selected Order Component: Bundle

Order Item Associ... | Conditions »₂

Order Item Associations

bundle

Remove Add

Selected Order Item Association: bundle ?

☒ Fulfillment Pattern
☐ Matching Order Component ID
☐ Property Correlation

XQuery | Instances | Notes

☒ Expression
☐ File
☐ URI

There are three ways to associate order items:

- Fulfillment pattern: This is the default entry. It associates order items by fulfillment pattern, which is the normal orchestration method.
- Matching Order Component ID: This associates order items by matching component ID.
- Property correlation: This associates order items by using order item properties. See ["About Associating Order Items Using Property Correlations XQuery Expressions"](#) for more information.

Modeling Order Item Hierarchies

Order items can be organized hierarchically based on the content of the original customer order. You can configure OSM with the following types of order item hierarchies:

- **Physical Hierarchy:** The hierarchy can include various types of order line items, such as offers, products, and bundles of products or services. For example, an order could include a Broadband-VoIP offer with a High Speed Internet bundle, an Internet Services service bundle, and a Wireless Router product item. OSM maintains the order line item hierarchy from the customer order in the order item hierarchy.
- **Composition Hierarchy:** You can use composition hierarchies with fulfillment state composition rule sets to determine the parent/child relationship between order items so that OSM can determine aggregate fulfillment states for parent order items. See *OSM Concepts* for more information.
- **Dependency Hierarchy:** You can specify a dependency hierarchy that OSM uses to automatically configure dependencies between order items on an order. For more information, see "[About Processing Order Items Sequentially](#)".

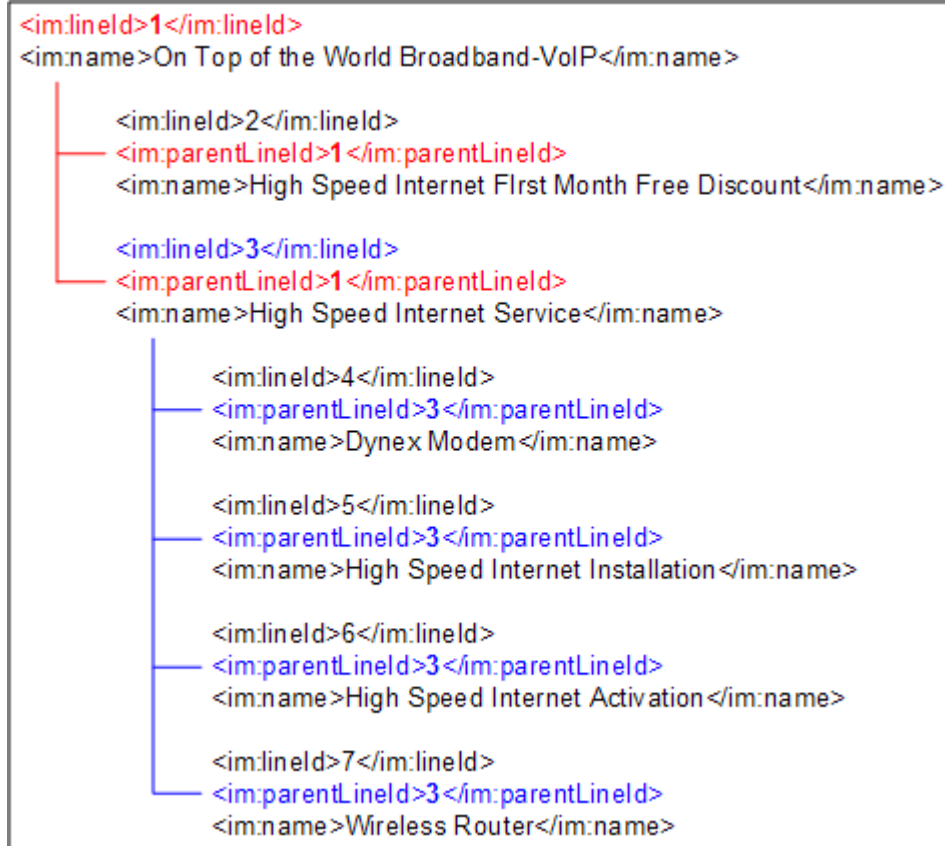
[Figure 5-9](#) shows a physical order item hierarchy that reflects the structure of the original customer order.

Figure 5-9 Physical Item Hierarchy



The hierarchy is defined in the `<lineID>` and `<parentLineID>` elements. [Figure 5-10](#) shows the first part of [Figure 5-9](#), as it appears in an incoming customer order.

Figure 5-10 Item Hierarchies in an Incoming Customer Order



To define the order item properties that specify the hierarchy, you configure the **order item hierarchy** in the order item specification using an XQuery expression. See "[About Order Item Specification Order Item Hierarchy XQuery Expressions](#)" for more information.

An order item hierarchy is invalid when:

- The hierarchy refers to a non-existent parent or child line ID.
- When the key or parent key XQuery is wrong.
- When the hierarchy specifies a circular relationship. For example, the parent of an order item is itself, or if order item A is the parent of order item B and order item B is also the parent of order item A.

OSM does not apply invalid order item hierarchies, but instead runs the order without any hierarchy.

About Using a Distributed Order Template

The distributed order template is a structure data type that is available only for order item specifications. It improves performance and also has the following benefits:

- Reduces order node conflicts: Without the distributed order template, data elements in the data dictionary that have the same name need to have the same definition (type, description, etc.) regardless of whether they appear in different structures in different places in the data dictionary. With the distributed order template, this is no longer necessary.

- Allows data changes without having to redeploy the entire solution: Without the distributed order template, any changes to the data defined for the order (including order item property updates) requires redeployment of the entire solution. With the distributed order template, if you change order item properties, you need to deploy only the cartridge containing the changed order item.

You decide whether to use the distributed order template by selection the appropriate box in the order item creation wizard or in the Order Item Specification editor in Design Studio. For more information, see Design Studio Modeling OSM Orchestration Help.

If you use a distributed order template, any references you make to order item data in XQuery expressions or automation must include a namespace. References to data in data change notifications and flexible headers do not need to change. For any order item that is not a transformed order item, the namespace will always be the namespace of the order item specification. Following is an example of an XQuery reference to the **lineItemID** property on the **InputOrderItem** order item with the namespace **http://ex_input.com**:

```
/ControlData/OrderItem[@type='{http://ex_input.com}InputOrderItem']/lineItemID
```

If you are using the order transformation manager, see "[Using the Distributed Order Template with the Order Transformation Manager](#)" for information about the namespace that will be used for transformed order items.

About Mapping Order Items to Fulfillment Patterns

The first orchestration stage assigns order items to function order components, by using fulfillment patterns. You need to model how to map order items to fulfillment patterns and implement the model using an XQuery expression (see "[About XQuery Expressions for Mapping Product Specifications and Fulfillment Patterns](#)" for more information).

Each order item in an order must have an order item property that specifies a value that represents a product, service, resource, or action. You map the value of the order item property to a corresponding fulfillment pattern designed to fulfill the order items mapped to them. Fulfillment patterns organize the functions into which order items decompose, any conditions that govern when an order item can be included in a function, and any associated order items that might be included in a function from different fulfillment patterns. Ideally, there ought to be a many-to-one relationship between order items and fulfillment patterns.

The way order items decompose to fulfillment patterns and further into functions depends on what kind of order item it is. For example, at the central order management (COM) level, you might group bundle order items as children of offer order items. The bundle order items would in turn be parents to product order items. [Example 5-3](#) is a possible hierarchy where each product order item maps to either an Service.VoIP or Service.CPE.VoIP fulfillment pattern:

Example 5-3 Sample COM Order Item Hierarchy

```

1 On Top of the World Broadband-VoIP (OFFER)
  5 High Speed VoIP Service (Bundle)
    6-VoIP Services (Product) ---> Service.VoIP
      7-VoIP PS (Product) ---> Service.VoIP
        20-Value Added Features PS (Product) ---> Service.VoIP
        22-VoIP Adaptor PS (Product) ---> Service.CPE.VoIP
        25-VoIP Phone PS (Product) ---> Service.CPE.VoIP
        26-VoIP Soft Phone PS (Product) ---> Service.CPE.VoIP
        27-VoIP Visual Voicemail PS (Product) ---> Service.VoIP
        28-VoIP Voicemail PS (Product) ---> Service.VoIP

```

Those order items destined to the Service.VoIP fulfillment pattern would decompose to the following functions:

- ProvisionOrderFunction
- InitiateBillingFunction
- SyncCustomerFunction
- FulfillBillingFunction

You can configure conditions where an order item might not be included in a specific function. For example, if a customer decides to move their VoIP service from one residence to another, you could configure a condition on the InitiateBillingFunction that would block the VoIP service order items from being included in the InitiateBillingFunction since the customer is already being billed for the VoIP services.

Sometimes, you need to assign order items to order functions that would not be assigned to the current fulfillment pattern by their product specification. This requirement can occur when an interaction with an external system requires a specific context for an order item.

For example, a billing system might need to process billing-related order items in the context of a bundle, to manage the relationships between balances, discounts, and so on. Billing charges are often order line items, such as an installation service, that are included in the order outside of the service billing bundle hierarchy. However, they might need to be associated with the billing bundle to ensure that the charge is made against the correct service. In that case, you can associate the billing charges with a bundle order component. By contrast, billing order items might be sent to the billing system in the context of a whole order. In that case, you do not need to associate the order items to a bundle, because they are already in context.

About Modeling Product Specifications

New product specifications should be imported (which will create conceptual model products) or created in the conceptual model. If you have an existing configuration, however, you can still use product specifications (formerly called product classes) that were created in OSM.

You can map multiple product specifications to one fulfillment pattern. This enables you to introduce new products in existing product specifications without needing to create new fulfillment patterns or fulfillment flows.

The Design Studio conceptual model functionality helps you model data as part of an end-to-end solution in an application agnostic way. You create conceptual model projects to:

- Define products.
- Define the services that the products represent.
- Define the resources that implement those services.
- Define service domains, such as broadband (ADSL, VDSL, DOCSIS, and Fiber), VoIP, email, Mobile, and so on.
- Define actions and relationships between products, services, and resources

Conceptual model items are not built into OSM cartridges or deployed to the OSM server directly. They are included into OSM by something called realization. Realization refers to converting the abstract entities in the conceptual model into actual instances in the OSM configuration. You can use this conceptual model metadata as part of your OSM run-time solution to help define order item to fulfillment pattern mappings and to give you an representation of what you need to implement in OSM as part of your overall fulfillment solution.

See *Design Studio Concepts* for more information about conceptual model projects. See ["About XQuery Expressions for Mapping Product Specifications and Fulfillment Patterns"](#) for more information about using conceptual model entities to map order items to fulfillment

patterns. See "[OSM Solution Modeling Overview](#)" for more information about how OSM can be modeled in an end-to-end solution.

Modeling Fulfillment Modes

The fulfillment mode is the overall purpose of the order. For example:

- Deliver a service.
- Qualify a service before delivering it. This ensures that a service can be fulfilled before attempting to fulfill it.
- Cancel an entire order.

Every incoming customer order can specify a fulfillment mode.

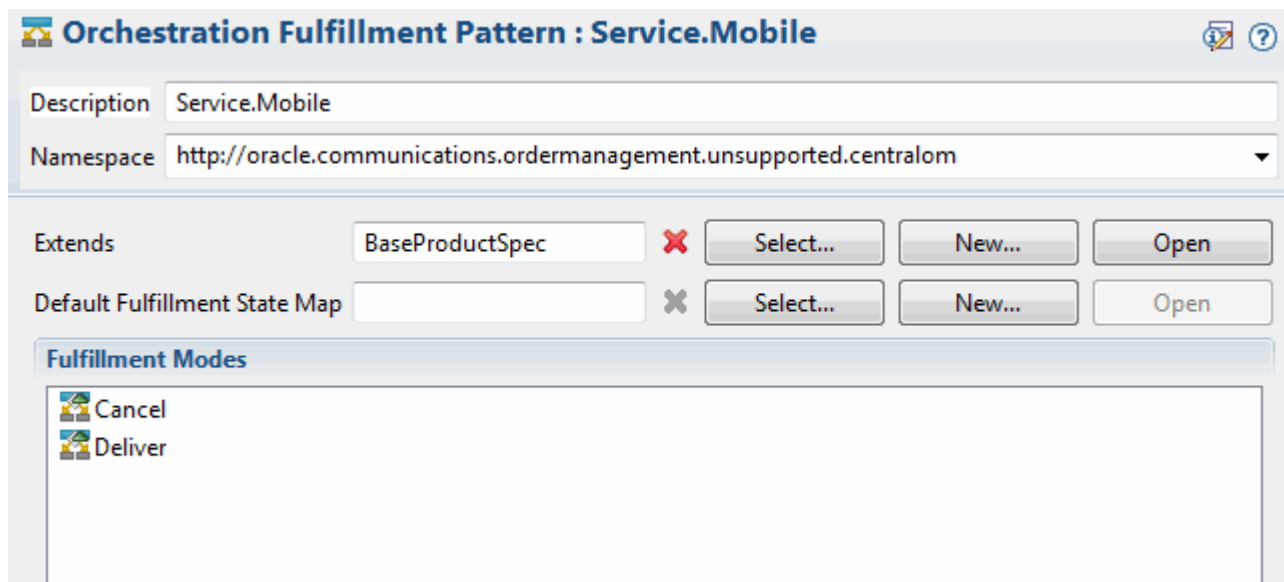
OSM can use the fulfillment mode as part of the orchestration process. For example, if OSM receives two identical incoming customer orders with different fulfillment mode order item properties, it generates a different orchestration plan for each order. The two plans include different executable order components with different dependencies among order items.

Fulfillment modes are configured in the following places:

- Fulfillment mode entities: These entities include no data other than a name. They provide the means to assign fulfillment modes to other entities, such as orchestration sequences and fulfillment patterns.
- Orchestration sequences define a single fulfillment mode using an XQuery expression based on a customer order attribute (see "[About Order Sequence Fulfillment Mode XQuery Expressions](#)").
- Fulfillment patterns list the fulfillment modes that the associated order items can be used with.

[Figure 5-11](#) shows the fulfillment modes defined in a fulfillment pattern. Any order item that uses this fulfillment pattern can be processed in either the Cancel or Deliver fulfillment mode.

Figure 5-11 Fulfillment Modes Defined in a Fulfillment Pattern



When a fulfillment pattern includes multiple fulfillment modes, you can model a different set of order components and dependencies for each fulfillment mode.

About the Decomposition of Order Items to Function Order Components

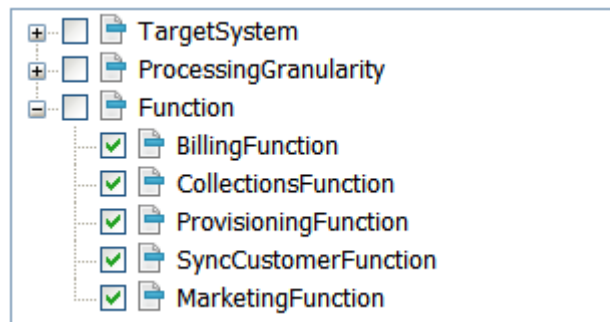
The following sections describe the decomposition of order items to function order components.

About Assigning Order Items to Fulfillment Pattern Function Components

The first step in decomposition is to assign order items to function components. To do so, OSM uses the product specification to find the fulfillment pattern that the order item uses. (See ["About Mapping Order Items to Fulfillment Patterns"](#) for more information.) The fulfillment pattern defines the order components to add the order item to.

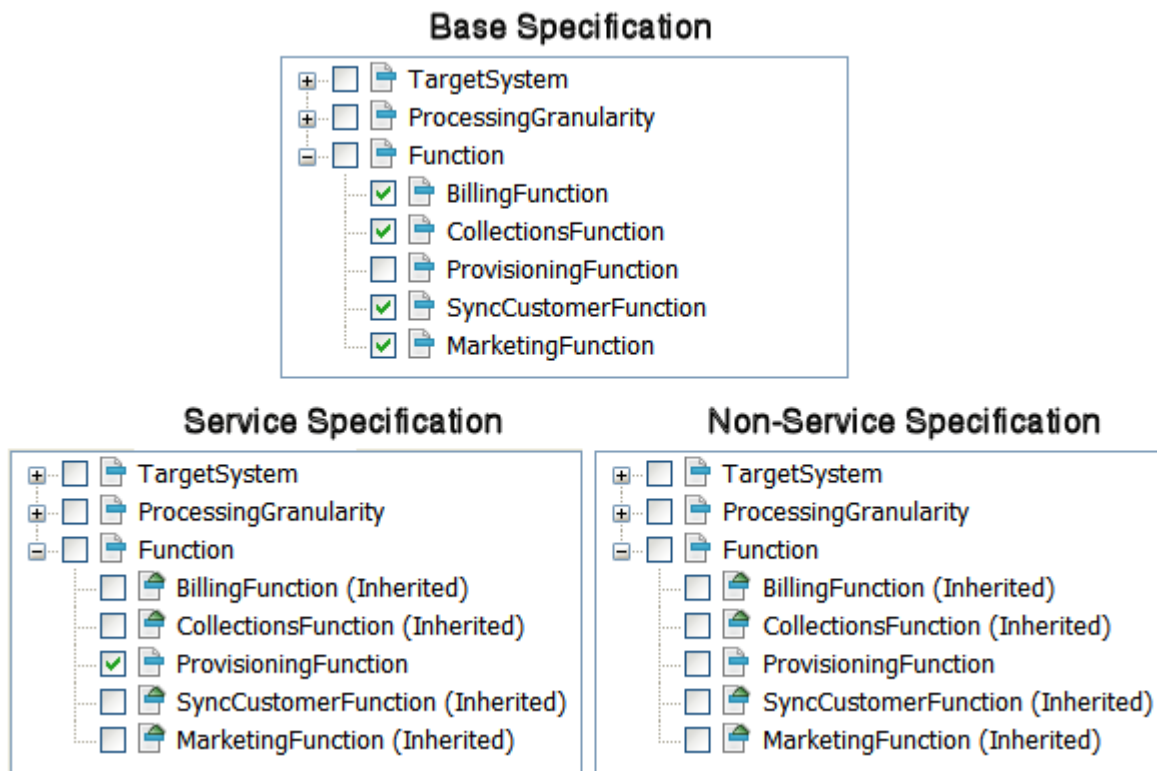
[Figure 5-12](#) shows the function order components selected in the Service.Broadband fulfillment pattern. In this case, order items that use this fulfillment pattern need all of the functions; billing, collections, provisioning, and so on.

Figure 5-12 Function Order Components Selected for a Service Fulfillment Pattern



[Figure 5-13](#) shows how to use a base specification to define the same function order components as described above. In this case, the base fulfillment pattern selects all of the function order components except provisioning. The service and non-service fulfillment patterns inherit the selections. The service fulfillment pattern adds the provisioning function. The non-service fulfillment pattern does not add it.

Figure 5-13 How to Use a Base Specification to Define Function Components



About the Function Components Stage

In addition to using the fulfillment pattern to assign order items to function components, you model an orchestration stage, which specifies to create the function order components to create.

About Order Component Control Data

When OSM creates the order items and order components, it produces a set of control data. The control data provides information OSM requires to fulfill the order. OSM uses the control data to track the status of the entire order and to track the status of the individual order items. During fulfillment, order component transactions update this control data with system interaction responses.

Design Studio automatically generates control data for function order components provided that orchestration entities are preconfigured correctly and you use the **OracleComms_OSM_CommonDataDictionary** model project. If you do not use the **OracleComms_OSM_CommonDataDictionary** model project, you must manually model order component control data. See "About Modeling Order Component Control Data" in *Modeling OSM Orchestration* for information on how order component control data is automatically generated or how to manually model it.

See "[Modeling OSM Data](#)" for more information on adding function order components to the order control data.

About Fulfillment Pattern Conditions for Including Order Items

You can use conditions to add order items to an order component only when the XQuery for the condition evaluates to true. For example, you might include an order item based on an XQuery that checks the action code (Add or None). This is useful in the case of an update to a service that changes some features while leaving other features unchanged. See "[About Order Item Specification Condition XQuery Expressions](#)" for more information.

Summary of Order Item to Function Components Decomposition

To summarize this example, to model the decomposition from a order items to a function component, you model the following:

- The fulfillment pattern order item property so that order items can be mapped to fulfillment pattern function components.
- Any XQuery expressions that evaluate conditions to include or exclude order items.
- The Order control data for orchestration.
- The orchestration stage that produces the function components

About the Decomposition of Function to Target System Components

The following sections describe the decomposition of order items from functional components to target systems.

About Decomposition Rules from Function Components to Target Systems

After the order items have been assigned to function order components, they need to be further decomposed into target system order components. To do so, you use **decomposition rules**.

A decomposition rule specifies a source order component and a target order component. [Figure 5-14](#) shows a decomposition rule from the billing function component to the billing target system component.

Figure 5-14 Decomposition Rule

Decomposition Rule : BillingToBillingSystem

Description: Billing To Billing System

Namespace: http://oracle.communications.ordermanagement.unsupported.centralom

Source Order Components	Target Order Components
Base Component: All Components	Base Component: All Components
Functions: All Components	Functions: All Components
Search:	Search:
- TargetSystem - ProcessingGranularity - Function ☒ BillingFunction - CollectionsFunction - ProvisioningFunction - SyncCustomerFunction - MarketingFunction	- TargetSystem - CollectionsSystem - FixedProvisioningSystem - MarketingSystem - MobileProvisioningSystem - DSLProvisioningSystem_Region2 - CustomerSystem - DSLProvisioningSystem_Region1 ☒ BillingSystem - ProcessingGranularity - Function

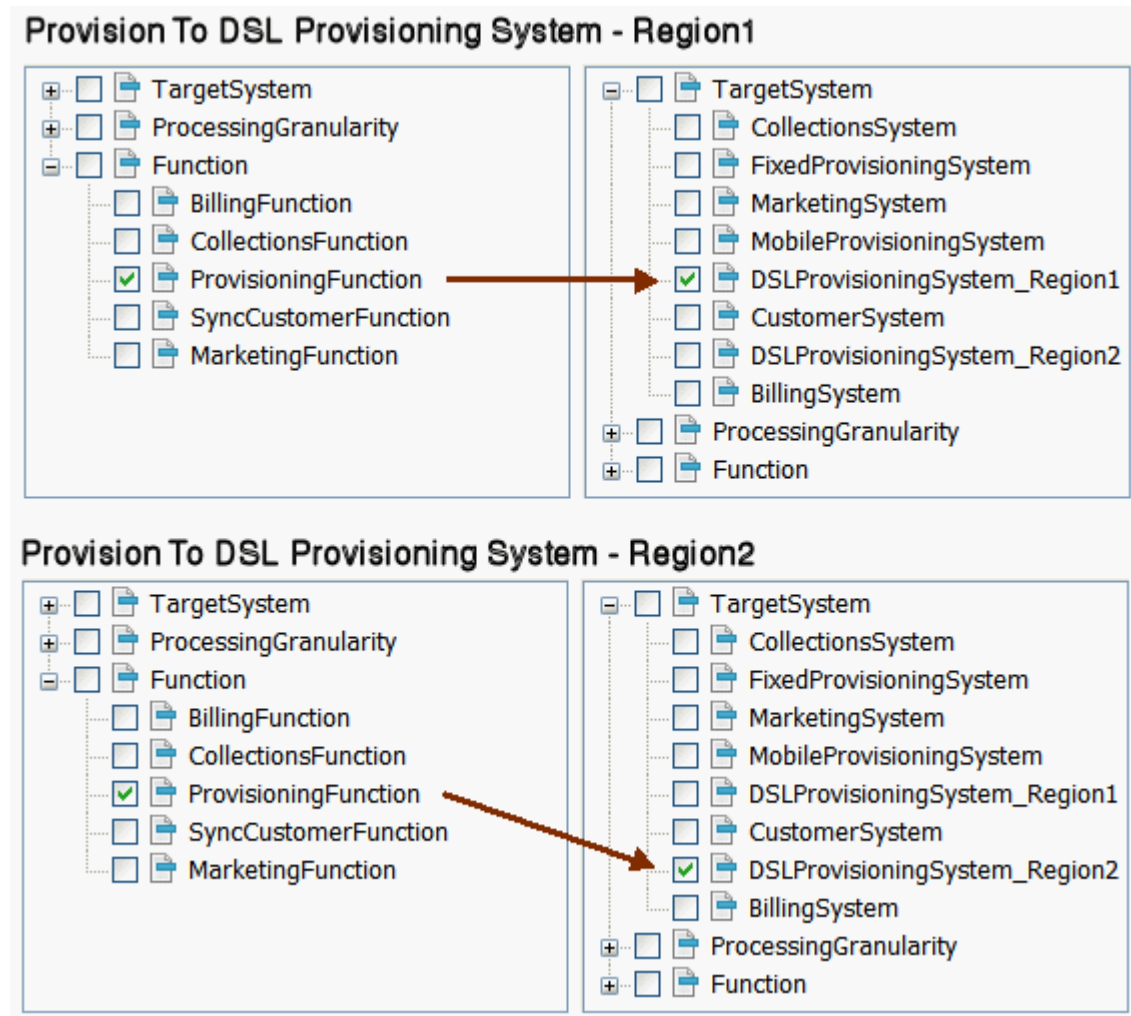
About Decomposition Rule Conditions for Choosing a Target System

You can use decomposition rules to decompose order items from one function component to multiple target system components. [Figure 5-15](#) shows the source and target order components for two decomposition rules:

- Provision to DSL Provisioning System - Region1
- Provision to DSL Provisioning System - Region2

These two decomposition rules decompose the order items in the ProvisioningFunction order component into two target system order components based on Region 1 and Region 2.

Figure 5-15 Source and Target Order Components for Two Decomposition Rules



Each of the decomposition rules uses decomposition conditions to specify which target system to use for a particular order. The target system is selected if the XQuery expression associated with the condition evaluate to true. In this example, the XQuery expression uses the value of the **region** order item property to make this evaluation. If the value of region is Toronto, then OSM selects the condition and target system for Region 1. If the value of region is New York, then OSM selects the condition and target system for Region 2. See "[About Order Item Specification Condition XQuery Expressions](#)" for more information about creating an XQuery condition expression that can be used for with a decomposition rule.

About the Target Systems Stage

In addition to creating the decomposition rules that define the source and target components, you need to create an orchestration stage that produces the target system order components.

Summary of Configuring Target System Components Decomposition

To summarize, to configure how order items are decomposed from a function order component to a target system order component, you do the following:

- Define an orchestration stage to produce the target system order components.
- Create dependency rules to specify the source order components and target order components.
- If a function order component decomposes order items to more than one target system order component, create decomposition conditions. Decomposition conditions depend on data specific to the order items, so decomposition rules typically use XQuery expressions to retrieve the data that is used for evaluating the condition.

About the Decomposition of Target System to Granularity Components

The following sections describe the decomposition of order items from target system components to granularity components.

About Decomposition Rules from Target System to Granularity Components

After order items have been decomposed into target system order components, the next step is to decompose them into the granularity order components.

Some examples of the granularity requirements are:

- A billing system might require the entire order in the message to calculate discounts.
- A billing system might require separate bundles for mobile billing and fixed billing, to handle different completion times (fixed billing typically has more dependencies and takes longer).

To decompose target system order components items into bundle granularity components you configure the following:

- Create a decomposition rule, which decomposes the target system order component into bundle granularity components.
- Create customized component IDs (stored in the **componentKey** data element in the control data) that are used to create separate order components for each bundle. See ["About Customized Component IDs for Separating Bundled Components"](#) for more information. The componentKey data element is used as the data key for the order component. (See ["About Order Data Position and Order Data Keys"](#) for information about the use of data keys in OSM.)

About Customized Component IDs for Separating Bundled Components

You create the customized order component by editing the bundle order component specification.

You need to configure a decomposition rule and a bundle granularity order component specification to make sure that order items for a fixed service and a broadband service are decomposed into separate bundle granularity components, based on their customized component IDs. The customized component IDs result in separate instances of bundle order components, with separate component keys. This allows OSM to process the order components for the fixed service and the broadband service separately. If you do not create customized component IDs, the order items are processed together in the same order component.

This customization also ensures that the component ID is the same for order items within the same granularity (for example, a bundle) but not for order items at a higher granularity.

In addition, you may want to group order items into custom component IDs based on order item requested delivery date. For example, you might want an order component to process all order items with a requested delivery date that falls within the first two days of when an order start, and another order component for the next two days. You can further combine these grouping by requested delivery date within order item hierarchy groupings.

See "[About Component Specification Custom Component ID XQuery Expressions](#)" for more information about configuring custom order component hierarchies using XQuery.

About the Granularity Components Stage

In addition to creating the decomposition rules that define the source and target components, you need to create an orchestration stage that produces the granularity order components.

Summary of Configuring Granularity Components Decomposition

To summarize, to model the decomposition from a target system order component to a bundle order component, you model the following:

- The decomposition rule, which decomposes the target system order component into bundle granularity components
- The orchestration stage that produces the bundle order component
- The order item hierarchy that the XQuery **ancestors** function uses in the order item specification
- The XQuery for the customized order component in the bundle order component specification

About Dependencies

An orchestration plan is based on two main factors: decomposition, which derives the order components, and dependencies, which dictate when the order components are allowed to run. OSM calculates order item decomposition first before calculating dependencies.

Dependencies are relationships in which a condition related to one order item must be satisfied before another item can be processed successfully. For example, a piece of equipment must be shipped to a location before the action to install it at that location can be taken.

You typically create dependencies between order items in the same order (intra-order dependencies). You can model the following types of intra-order dependencies using fulfillment patterns:

- **Order Item dependency:** A dependency that requires the completion of one type of fulfillment function for an order item before starting another type of fulfillment function for the same order item within a single fulfillment pattern. For example, for a single order item included in a VoIP.Service fulfillment pattern, you can specify that the provision function order component must process an order item before the bill function order component can begin processing the same order item.
- **Fulfillment pattern dependency:** A dependency that requires the completion of a fulfillment function for an order item in a fulfillment pattern before starting a fulfillment function for another order item in a different fulfillment pattern. For example, for a single order item included in a VoIP.Service fulfillment pattern, you can specify that the provision

function order component can only process an order item after the provision function order component on the BroadBand.Service fulfillment pattern has completed.

- **Order item property dependency:** A dependency that requires the completion of one order item before starting another order item based on order item properties.
- **Order item hierarchy dependency:** You can configure an order item hierarchy that automatically creates dependencies between predecessor and successor order items based on order item properties so that order items run sequentially. The successor order item can only begin after the predecessor order item completes. For example, order item A would have to complete all functions within its fulfillment pattern before order item B could begin processing its functions within its own fulfillment pattern. The fulfillment patterns could be identical or different, but they would have to be run separately for each order item with the parent child relationship.

For more information, see "[About Intra-Order Dependencies](#)".

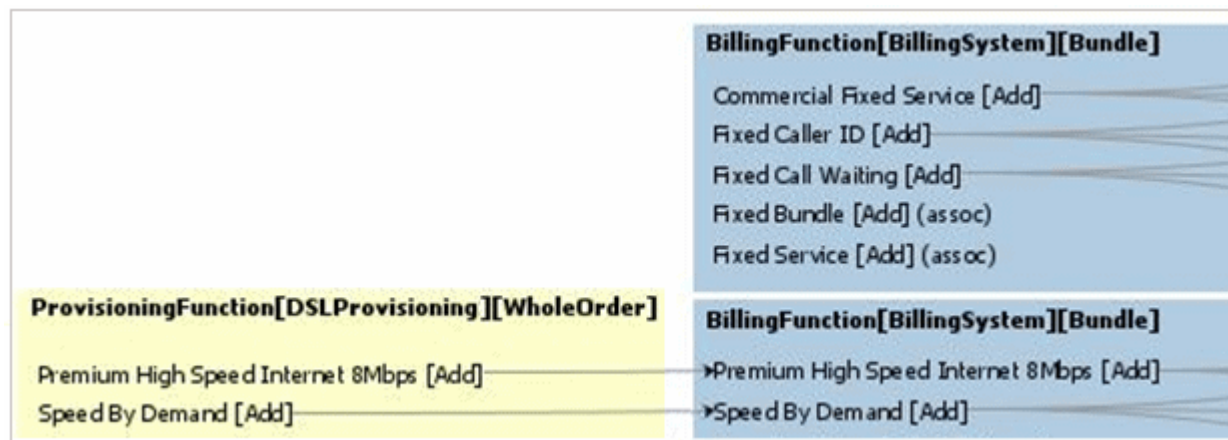
You can also create dependencies between order items in different orders (Inter-order dependencies). For example, the order items in a follow-on order for VoIP provisioning might depend on the execution of the order items in the original order for DSL provisioning. See "[About Inter-Order Dependencies](#)" for more information.

You can model dependencies in two ways in Design Studio:

- As order item dependencies. These dependencies are modeled as part of fulfillment patterns. Most dependencies are modeled in this manner.
- As orchestration dependencies. These dependencies are modeled outside of fulfillment patterns. While not as common as those modeled in fulfillment patterns, orchestration dependencies are useful in specific circumstances; for example, if you need to define a generic dependency or want to model one without having to modify a fulfillment pattern.

[Figure 5-16](#) shows order items displayed in the Order Management web client. In this example, the billing order items for a fixed service can start immediately because they have no dependencies. The billing order items for high-speed Internet must wait until the provisioning order items have completed.

Figure 5-16 Dependencies Displayed in the Order Management Web Client



About Intra-Order Dependencies

A dependency requires two order components: the waiting order item and the blocking order item. The blocking order item is the order item that must complete before the waiting order item is started.

Dependencies can be based on several different factors, including:

- Completion status. For example, the blocking order item must be complete before the waiting order item can start or you can specify to start billing only after provisioning has completed.
- Actual and relative date and time. For example, you may want an order component that contains order items for an installation to start two days after the completion of the order component that contains the order items for shipping the equipment.
- Data change. For example, you can specify that shipping must wait until a specified order item property in the blocking order item has a specified value.

Order items can have combinations of dependencies. For example, an order item for an installation can depend on a combination of a completion status dependency (item successfully shipped) and date dependency (wait two days after shipment to schedule installation).

Note

You can manage dependencies during amendment processing; for example, when you submit a revision order. See "[Modeling Changes to Orders](#)" for more information.

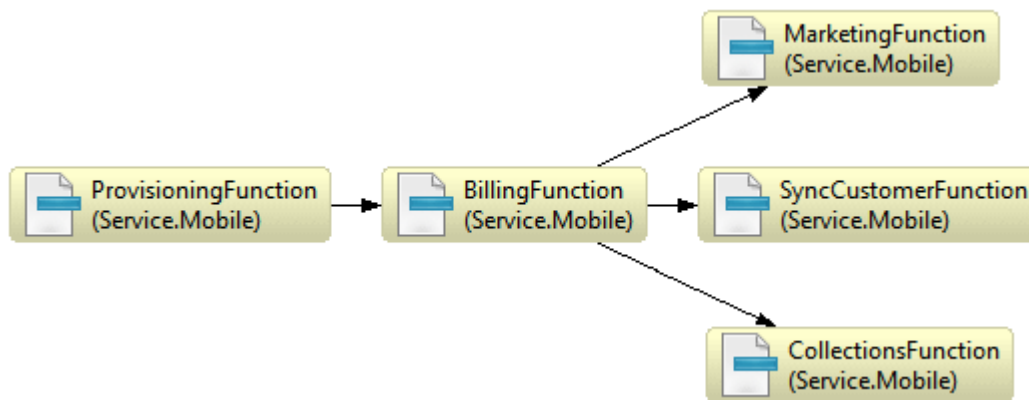
Although dependencies exist logically between order items, they are managed by order components. In other words, if any item in a component has a dependency, the component as a whole cannot be started until the dependency is resolved. In the Order Management web client, order items include dependency IDs to indicate items whose dependencies are managed together. See *OSM Order Management Web Client User's Guide* for more information.

Modeling an Order Item Dependency

The simplest form of dependency is an order item dependency, configured in a fulfillment pattern. This type of dependency is based on function order components; for example, the billing order component cannot start until the provisioning function has completed.

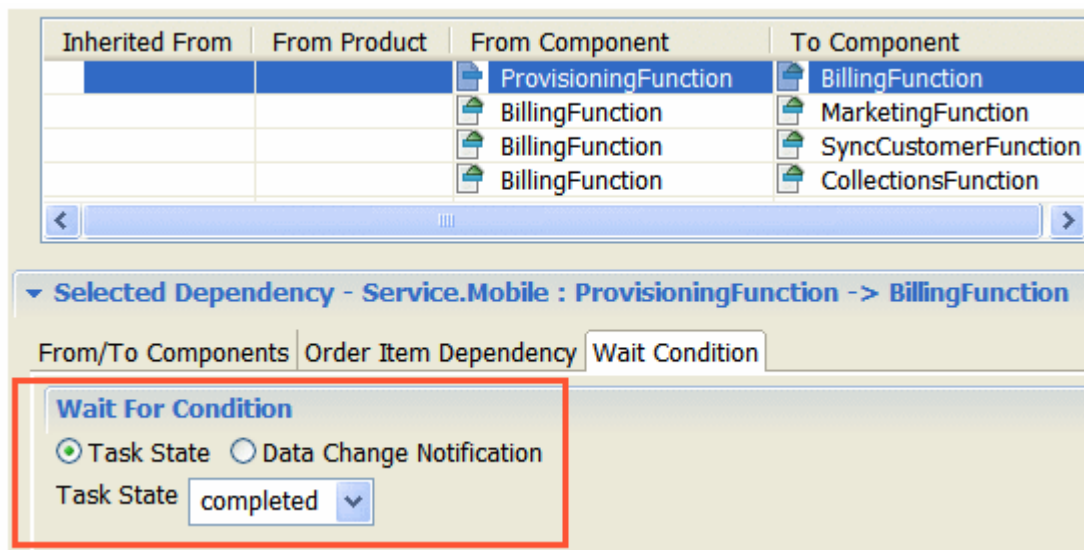
[Figure 5-17](#) shows a dependency relationships. Note the two layers of dependency: billing is dependent on provisioning, and everything else is dependent on billing.

Figure 5-17 Dependency Relationships for Order Item Dependency



In addition to defining the function order components, you need to define the conditions that govern the dependency. The default condition is to wait until the final task associated with the order item has completed. [Figure 5-18](#) shows a wait condition defined in Design Studio. In this case, the waiting order item must wait until the blocking order item task has reached the Completed state. See "[About Order Item Dependency Wait Conditions](#)" for more information.

Figure 5-18 Wait Condition in Design Studio



About Order Item Dependency Wait Conditions

Dependency wait conditions specify the condition that the blocking order item must be in before the waiting order item can start. For example, the default wait condition is to start the waiting order item when the last task associated with the blocking order item reaches the Completed state.

You specify wait conditions in fulfillment patterns and orchestration dependencies. You can set different wait conditions for each dependency. The wait conditions can be:

- The task state of the final task associated with the blocking order item
- A change in the data for a specified field. See "[About Order Item Dependency Wait Conditions Based on Data Changes](#)" for more information.
- A specified duration after the task state or data change condition has been met. You can specify a value in months, weeks, days, hours, or minutes, or you can specify an XQuery expression to determine the delay (see "[About Wait Delay Duration XQuery Expressions](#)"). For example, you can specify to start the waiting order item two days after the blocking order item has completed.
- A specific date and time based on the result of an XQuery expression (see "[About Wait Delay Date and Time XQuery Expressions](#)"). For example, you can specify to start the To Component order component on a date specified in an order item property.

The orchestration dependency wait condition options are identical.

About Order Item Dependency Wait Conditions Based on Data Changes

You can base a dependency on a change to data. The data must be included in an order item property, and it must be in the task data of the task associated with the blocking order item.

To configure the dependency, you define the following:

- The order item property that is evaluated. Any change to the data in the order item property triggers an evaluation of the data to determine if it matches the conditions required for the dependency.
- An XQuery expression that evaluates the data retrieved from the blocking order item. The expression returns **true** or **false**; if true, the dependency has been met.

[Figure 5-19](#) shows a data change dependency in Design Studio.

Figure 5-19 Data Change Dependency in Design Studio

Wait For Condition

☐ Task State ☒ Data Change Notification

Order Item:

Data Change Notification Property:

Relative Path:

Data Change Condition Expression

☒ XQuery ☐ Instances ☐ Information

☐ None ☒ Expression ☐ File ☐ URI

```
declare variable $blockingIndexes as xs:integer* external;
let $expectedMilestoneCode := "PROVISION STARTED"
```

In [Figure 5-19](#):

- The **Order Item** field specifies the order item specification to use.
- The order item property that the dependency is based on is **milestone**.

The **Relative Path** field (not used in this example) is an optional field you can use to specify a child data element in the order item properties.

- The XQuery expression evaluates the data in the **milestone** property to determine if the dependency has been met. See "[About Order Data Change Wait Condition XQuery Expressions](#)" for more information.

Modeling a Fulfillment Pattern Dependency

You can define dependencies across different order items by basing the dependency on the fulfillment patterns of the order items. For example, you can create a dependency that specifies to provision fixed services only after broadband services have been provisioned.

[Figure 5-20](#) shows a dependency based on fulfillment pattern. In this example, the dependency requires that fixed services be provisioned before broadband services. To configure this type of dependency, you edit the fulfillment pattern of the waiting order item. In the fulfillment pattern, you provide a list of waiting and blocking order components.

Figure 5-20 Dependency Based on Fulfillment Pattern

The screenshot displays the 'Orchestration Fulfillment Pattern : Service.Broadband' configuration window. The 'Description' field is set to 'Service.Broadband' and the 'Namespace' is 'http://oracle.communications.ordermanagement.unsupported.centralom'. Below these fields is a table with three columns: 'Inherited From', 'From Fulfillment Pattern', and 'From Component'. The table lists three components: 'BillingFunction' (inherited from 'Service.Fixed'), 'BillingFunction' (from 'Service.Fixed'), and 'ProvisioningFunction' (from 'Service.Fixed'). Below the table are 'Remove' and 'New' buttons. A section titled 'Selected Dependency - Service.Fixed : ProvisioningFunction -> ProvisioningFunction' shows the 'From/To Components' as 'Order Item Dependency' and 'Wait Condition'. Under 'Order Item Dependency', three radio buttons are present: 'Order Item', 'Fulfillment Pattern' (which is selected), and 'Property Correlation'.

Inherited From	From Fulfillment Pattern	From Component
		BillingFunction
		BillingFunction
Service.Fixed		ProvisioningFunction

Remove New

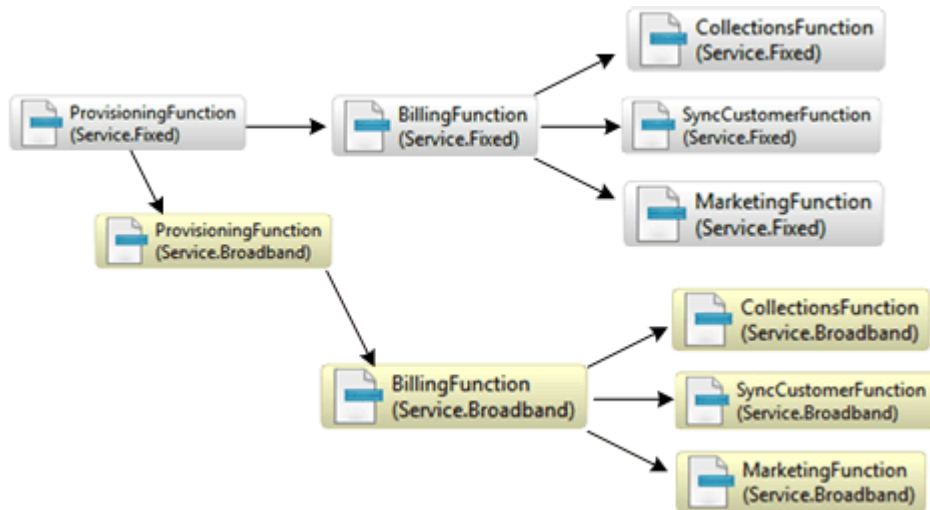
Selected Dependency - Service.Fixed : ProvisioningFunction -> ProvisioningFunction

From/To Components Order Item Dependency Wait Condition

Order Item Dependency

☐ Order Item
☒ Fulfillment Pattern
☐ Property Correlation

[Figure 5-21](#) shows the dependency relationships shown in [Figure 5-20](#). Note that fixed provisioning is the blocker for broadband provisioning and for fixed billing.

Figure 5-21 Dependency Relationships for Fulfillment Pattern Dependency

Modeling an Order Item Property Correlation Dependency

Using properties correlation is the most flexible way to configure dependencies. You use this method to create a dependency on two different order items that share the same order item property. As with other dependencies, you specify a blocking component (the **From Component** field) and a waiting component (the **To Component** field), but you also enter an XQuery expression to select the order item property that order items in the **To Component** field must share with order items in the **From Component** field (see ["About Order Item Dependency Property Correlation XQuery Expressions"](#) for more information).

About Inferred Dependencies

OSM is able to create dependencies at run time by inferring dependencies. For example, you might create this series of dependencies:

Provisioning - Billing - Marketing

If the order item has no billing function, there is an inferred dependency between Provisioning and Marketing, even though you have not modeled that dependency. Provisioning must complete before Marketing can start.

Inferred dependencies mean that whenever A is dependent on B and B is dependent on C, A is dependent on C. This avoids the need to model every dependency that might be possible.

[Figure 5-22](#) shows a sample dependency configuration. [Figure 5-23](#) shows the run-time view of the same configuration when there is no billing function. In this case, the Order Management web client shows dependencies from provisioning to marketing, synchronize customer, and collections.

Figure 5-22 Dependency Relationship

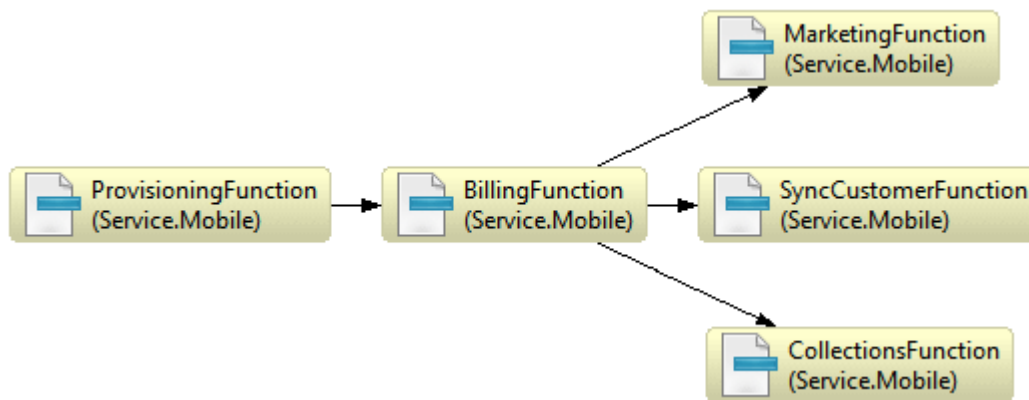
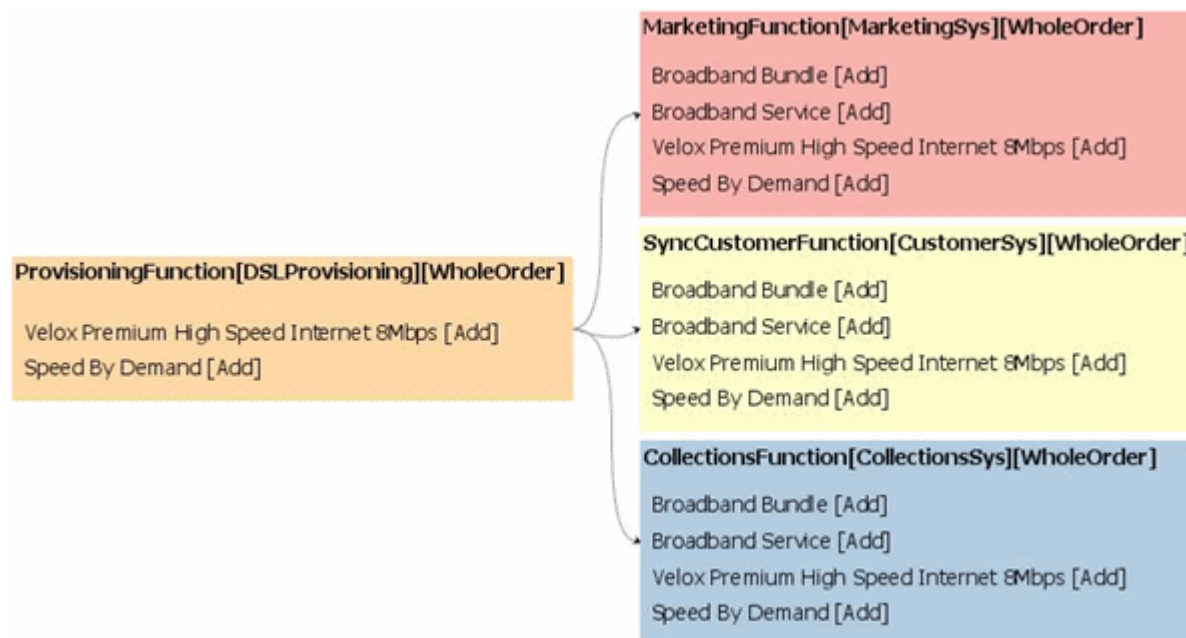


Figure 5-23 Inferred Dependencies at Run Time



Inferred dependencies are supported within a fulfillment pattern, but they are not supported across fulfillment patterns. For example, OSM does not infer a dependency from ProvisioningFunction(Service.Fixed) to BillingFunction(Service.Broadband). You must specifically model that dependency.

About Modeling Orchestration Dependencies

You use orchestration dependencies to create dependencies between order components that are not based on fulfillment patterns. For example, if you need to define a generic dependency or want to model one without having to modify a fulfillment pattern, you can use an orchestration dependency specification.

As with dependencies defined in fulfillment patterns, you can specify wait conditions and the type of order item dependency (for example, order item, fulfillment pattern, and property correlation).

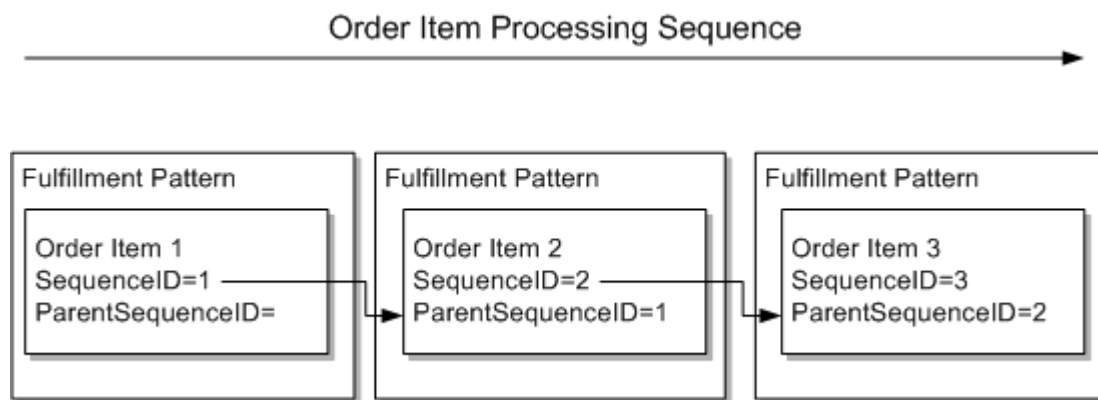
About Processing Order Items Sequentially

You can enable order items to process sequentially at run-time by setting an order item dependency hierarchy in the order item specification editor **Order Item Hierarchies** tab. When you model order items to run sequentially, avoid creating circular dependencies by ensuring that you do not include order items with a predecessor successor relationship into the same order component.

For example, you can ensure that only one order item processes at a time by configuring the orchestration granularity for a component to process only one order item at a time. Or you could also set the granularity for a component to process only a bundle of order items at a time. For example, between a bundle for VoIP and another bundle for Broadband. If parameters designating the successor predecessor relationship always establish a relationship between order items across two different bundles, then you avoid circular dependencies in this way as well.

[Figure 5-24](#) shows how order items can be configured to process sequentially based on two order item properties defined in an order item specification order item hierarchy. You can use any order item property, so long as you can use the properties to establish the predecessor and successor relationship.

Figure 5-24 Order Item Processing Sequence



See "[Modeling Order Item Hierarchies](#)" for more information about modeling order item hierarchies.

About Inter-Order Dependencies

An inter-order dependency is a dependency between order items in different orders. You typically configure this type of dependency to manage changes to an order when that order has passed the PONR and cannot be amended. However, you can also use inter-order dependencies for other purposes, such as managing fulfillment functions on different systems, load balancing, and so on.

When using inter-order dependencies, the blocking order is the base order, and the waiting order is a follow-on order. A typical scenario is:

1. A customer has ordered a broadband service.
2. The next day, while the order is still in-flight but past the PONR, the customer requests a change to the service bandwidth.
3. Because a revision to the base order cannot be submitted, the customer service representative creates a follow-on order.
4. The follow-on order is submitted to OSM; however, it does not begin processing until the base order has completed.

You typically model inter-order dependencies between a base order that has reached its point of no return (PoNR) (where a revision order is no longer possible) and a follow-on order (see "[Modeling a Point of No Return](#)" for more information). A follow on order does not trigger amendment processing on the original base order, but does have a dependency on one or more order items on the base order through the an inter-order dependency. You configure the inter-order dependency on the follow-on order so that it can check that the blocking order items on the base order have completed so that the waiting order items on the follow-on order can start processing.

Here are some important points to know about inter-order dependencies:

- Inter-order dependencies are based on order items. After the base order completes the blocking order item, the follow-on order can start, even though the base order is still in-flight.
- Inter-order dependencies are sometimes used to manage technical dependencies when a specific fulfillment requirement cannot be handled by a revision. However, they can also be based on business reasons, when it is simpler or more efficient to use a follow-on order than to model revisions.
- A follow-on order does not perform amendment processing on the base order. A follow-on order can be used to add, modify, or cancel services, similar to any order. The key feature is that a follow-on order has a dependency on another order.

You must model the inter-order dependencies into both the base order and the follow-on order.

- The follow-on order must be able to find the base order and be able to recognize if the blocking order item has completed.
- The base order must contain a reference to allow the follow-on order to find it.

To configure an inter-order dependency, you use the **Order Item Dependencies** tab. The configuration typically consists of the name of the dependency and its XQuery or data instance (see "[About Order Item Inter-Order Dependency XQuery Expressions](#)" for more information about inter order item XQuery expressions).

You can create inter-order dependencies that involve order item hierarchies. For example, you can specify that the blocking order item include all of the order items in its hierarchy. To do so, you select the child completion dependency when specifying an order item hierarchy (see [Figure 5-25](#)). For more information about order item hierarchies, see "[Modeling Order Item Hierarchies](#)".

Figure 5-25 Use for Child Completion Dependency Selected in Design Studio

The screenshot shows the Oracle Design Studio interface for the 'Order Item Specification : CustomerOrderItemSpecification'. The 'Description' field contains 'CustomerOrderItemSpecification' and the 'Namespace' field contains 'http://orade.communications.ordermanagement.unsupported.centralom'. The 'Order Item Hierarchies' section has three tabs: 'Physical Hierarchy', 'Composition Hierarchy', and 'Dependency Hierarchy'. The 'Physical Hierarchy' tab is selected, and the 'default' value is entered in the 'Physical Hierarchy' field. The 'Use For Child Completion Dependency' checkbox is checked and highlighted with a red box. The 'Key Expression: default' section is visible, showing the 'XQuery' tab selected. The 'Expression' radio button is selected, and the XQuery text area contains the following code:

```
declare namespace  
im="http://orade.communications.ordermanagement.unsupported"
```

About Modeling Orchestration Dependencies

[Figure 5-26](#) shows an orchestration dependency in Design Studio.

Figure 5-26 Orchestration Dependency in Design Studio

Orchestration Dependency : Provision_To_Billing

Description:

Namespace:

▼ Dependency Source

Base Component:

Functions:

Search:

- ☒ Function
 - ☐ BillingFunction
 - ☐ CollectionsFunction
 - ☒ ProvisioningFunction
 - ☐ SyncCustomerFunction
 - ☐ MarketingFunction

▼ Dependency Target

Base Component:

Functions:

Search:

- ☒ Function
 - ☒ BillingFunction
 - ☐ CollectionsFunction
 - ☐ ProvisioningFunction
 - ☐ SyncCustomerFunction
 - ☐ MarketingFunction

Selected Order Component: ProvisioningFunction

XQuery | Instances | Information

☒ None ☐ Expression ☐ File ☐ URI

Using Task States to Manage Orchestration Dependencies

You can use task states when defining orchestration dependencies. For example, you can specify to wait until a task has reached a specified state before an order component can be processed.

6

Modeling the Order Transformation Manager

This chapter describes how to model the order transformation manager in an Oracle Communications Order and Service Management (OSM) solution.

Understanding the Order Transformation Manager

The order transformation manager provides users with the ability to transform order items. For example, you can use the order transformation manager to transform customer-focused order items (what the customer bought) to service-focused order items (the services that equate to what the customer bought). It enables you to set up guidelines for order transformation that do not need to be changed due to product changes. Instead of writing a lengthy XQuery, users can model the order transformation in Oracle Communications Service Catalog and Design - Design Studio. The order transformation manager also provides visibility in the Order Management web client into service processing, making it easier to see how customer services are being transformed into the services being processed by OSM. In addition, the order transformation manager enables you to propagate data upstream and assists in status consolidation.

Order Transformation Manager in Runtime

In runtime, when the order transformation manager is triggered, OSM initiates the following process for each domain that has order items associated with it:

1. The appropriate **transformation sequence** is accessed to determine the appropriate transformation stages.
2. The **transformation stages** are run in sequence. For each transformation stage:
 - a. The stage condition is evaluated to determine whether the stage should be run. If not, OSM moves to the next stage.
 - b. The list of source order items is gathered: both context order items (the order items to be transformed) and related order items (order items that might contribute data to the transformed order items).
 - c. The list of mapping rules that apply to the named relationships for the transformation stage is gathered.
 - d. The mapping rules are processed, creating transformed order items and mapping parameters to them.
3. The transformed order items are processed in the same way as original order items, for example being processed by order components.

The Order Transformation Manager and the Conceptual Model

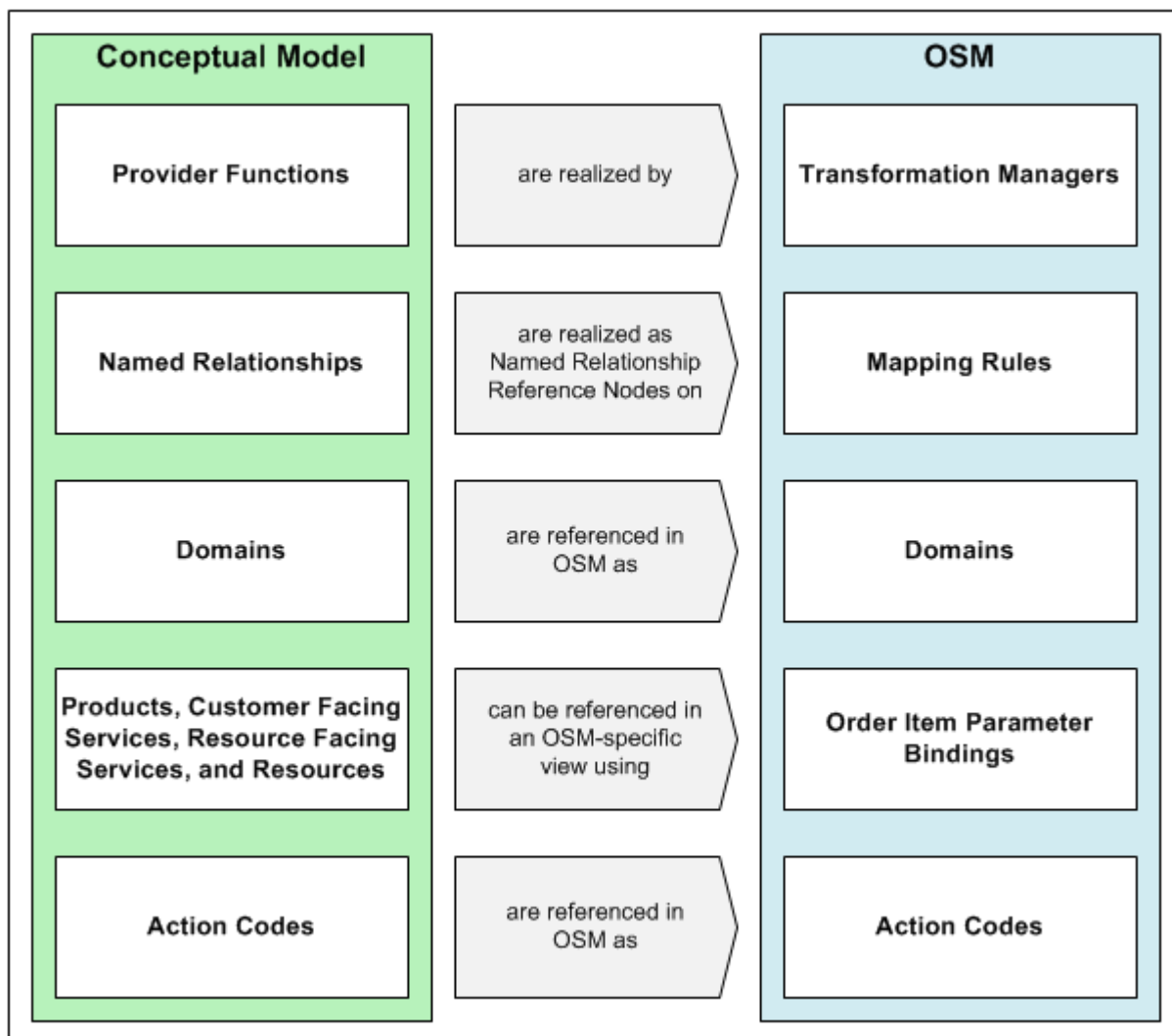
Entities are realized into the OSM cartridges by different means. Following is a description of how the different entities are realized into OSM or referenced by OSM.

- **Provider Functions:** Provider functions in the conceptual model are realized into OSM as transformation managers.

- **Named Relationships:** These entities are realized into OSM when they are referenced by OSM entities, such as mapping rules.
- **Domains:** Domains are referenced in OSM by transformation managers and mapping rules.
- **Products and Customer-Facing Services:** These entities are realized into OSM when they are included in relationships that are used by the order transformation manager and when their parameters are mapped to OSM order items using order item parameter bindings.
- **Action Code:** These are referenced in OSM as action codes.

[Figure 6-1](#) depicts general relationships between conceptual model entities and OSM entities that are used by the order transformation manager.

Figure 6-1 Relationships Between Conceptual Model Entities and OSM Entities



OSM Entities Used in the Order Transformation Manager

The order transformation manager uses several entities in Design Studio for OSM.

- **Transformation manager:** The transformation manager entity enables you to select the transformation sequences for the service domains within a provider function. This entity is the entry into the order transformation functionality.
- **Transformation sequence:** The transformation sequence enables you to define the transformation stages and the logic to be used at each transformation stage. Transformation stages define the source and target order items and the relationship between them for each step of the transformation.
- **Order item specifications:** You must define an original (source) order item specification that defines the structure of the incoming order items and a transformed (target) order item specification that defines the structure of the output of the order transformation for the order transformation manager. If the same structure is used for both, the same order item specification can be defined for both original and transformed order items. See "[About Order Items](#)" for more information about configuring and using order items.
- **Mapping rules:** Mapping rules define the way that original order items are transformed into transformed order items. You use mapping rules to define how transformed order items are generated and how their parameters and properties are populated. The data elements you can use as a source for the mappings are the parameters on the original order item in addition to the parameters on the actions defined for the order item. There are many different ways to generate the parameters and properties for the transformed order items. These methods include:
 - You can map parameters from the source order item to the target order item. You can copy the value from the source to the target, transform the value of the source parameter or property to a value on the target based on pre-defined value mappings or on the units of measure for each, and you can write XQuery expressions to do the value transformation.
 - You can map order item instances from the source order item to parameters or properties on the target order item. You can either set up a specific value to use on the transformed order item based on the presence of the source order item, or you can use XQuery to determine the value for the parameter or property on the transformed order item.

For more information about mapping rule types, see the Design Studio Modeling OSM Orchestration Help.

Mapping rules also enable you to map actions for the transformed order item either using the actions defined in the named relationship or defining the actions specifically for the mapping rule, based on the input, output, and current actions of the order items.

- **Order Item Parameter Bindings:** The order item parameter bindings enable you to bind the parameters from a conceptual model entity to parameters on an order item. They also enable you to determine the mapping between the parameters on the conceptual model entity and the properties on an order item. In addition, they enable you to transform the parameters from the customer order line before they are added to the conceptual model entity. One use for this would be to transform name/value-pair-type parameters from the incoming order into more strongly typed parameters on the conceptual model entity.
- **Transformation Tasks:** If you want to call the order transformation manager from a process instead of before the orchestration plan is generated, you do this using a transformation task. See "[Calling the Order Transformation Manager](#)" for more information. The transformation task is very much like an automated task, except that by default it has an appropriate automation plug-in defined for it and provides the ability to define the transformation manager to call.

Calling the Order Transformation Manager

There are two methods for calling the order transformation manager:

- If you want the order transformation manager to run before the orchestration plan is generated, select **Invoke Order Transformation Manager** in the Orchestration Process and select a provider function. This is the recommended practice, as it causes the order transformation manager to be run in the context of the whole order and with one call.
- If you want to call the order transformation manager at a different place in the order process, you can include a transformation task in an OSM process. The transformation task calls a specific transformation manager that you define in the task. This option provides flexibility in the following ways:
 - It enables you to call the order transformation manager multiple times in the process flow for different provider functions. You should not call the order transformation manager more than once for the same provider function.
 - It provides the option not to persist the results of the transformation to the order template. This is useful if the order transformation manager results are transient or going to be passed through directly to a southbound system. Additionally, this gives the user the flexibility to format any results that are going to be persisted in whichever structure they want.
 - It provides the ability to filter the order items passed into the order transformation manager. This enables a user to ensure that the order transformation manager only processes relevant order items.

The order transformation manager works the same regardless of the way it is called.

Using the Distributed Order Template with the Order Transformation Manager

When you are using the order transformation manager, you must use the distributed order template for the order item specification that contains transformed order items. For the order item specification that contains original order items, using the distributed order template is optional. See "[About Using a Distributed Order Template](#)" for general information about the distributed order template.

The distributed order template uses namespaces to determine the data structure that should be used. For transformed order items, the namespace depends on the source of the data for the transformed order item. Data that is defined in the order item specification itself will use the namespace for the order item specification, the same way that data would be referenced for an input order item. Following is an example of an XQuery reference to the **lineItemID** property on the **OutputOrderItem** order item with the namespace **http://ex_output.com**:

```
/ControlData/OrderItem[@type='{http://ex_output.com}OutputOrderItem']/lineItemID
```

Data that has been derived from a common model entity, for example an action, will use a different format. In the following situation:

- Order item namespace: **http://ex_output.com**
- Order item name: **OutputOrderItem**
- Name of the parameter assigned as the Dynamic Parameter Property in the order item specification: **dynamicParams**

- Conceptual model cartridge name: **Model_Broadband**
- Conceptual model cartridge version: **1.0.0.0.0**
- Conceptual model entity (in this case an Action) name: **SA_Add_Internet**
- Parameter name on SA_Add_Internet: **serviceLevel**

The reference would look like this:

```
/ControlData/OrderItem[@type='{http://ex_output.com}OutputOrderItem']/
dynamicParams[@type='{Model_Broadband/1.0.0.0.0}SA_Add_InternetType']/serviceLevel
```

The parameters from the conceptual model entity are contained in the dynamicParams element on the transformed order item. The type for the parameters contained in the conceptual model entity has the string "Type" appended to the name of the entity. Thus, the type contains **SA_Add_InternetType** rather than just SA_Add_Internet.

Modeling OTM With Calculate Service Order

Calculate Service Order is a specific provider function that is delivered via design patterns in Design Studio. The Calculate Service Order provider function is the functional module that transforms customer orders into service orders.

Using Calculate Service Order has two parts. First, you must run the relevant design patterns to set up the framework, and then you must configure the other required entities that are specific to your implementation.

Calculate Service Order Design Patterns

Calculate Service Order includes two design patterns:

- The Design Studio core software contains a design pattern (Common Model Base Data) that sets up the base data for the conceptual model. The following entities that are created in the conceptual model support Calculate Service Order:
 - A Design Studio project to contain the conceptual model entities (optional, an existing project can be used)
 - The Calculate Service Order provider function (see "[About the Calculate Service Order Provider Function](#)")
 - The Primary and Auxiliary relationship types (see "[About Calculate Service Order Relationship Types](#)")

For more general information about these entities, see the information about designing solutions in *Design Studio Concepts*.

- Design Studio for OSM contains a design pattern (Calculate Service Order) that contains OSM entities to support Calculate Service Order:
 - A Design Studio project to contain the OSM entities (optional, an existing project can be used)
 - The Calculate Service Order transformation sequence (see "[About the Calculate Service Order Transformation Sequence](#)")

About the Calculate Service Order Provider Function

The Calculate Service Order provider function is a logical entity that groups all the metadata required to perform the transformation. It also provides the ability to determine what types of

entities and relationships can be used in the transformation and the method used to realize the provider function into OSM.

The Calculate Service Order provider function defines the following associations:

- The input (Product) and output (Customer Facing Service and Resource) conceptual model entities
- The relationship types (Primary and Auxiliary)

About Calculate Service Order Relationship Types

Calculate Service Order also contains the definitions of the following relationship types:

- **Primary:** In this relationship type, transformed order items are created from original order items. Action codes are normally transferred to the target without being changed, or you can define rules to change the action types.
- **Auxiliary:** In this relationship type, transformed order items are enriched, but no new transformed order items are created. Action codes are translated based on the action type of the source item combined with the current action type of the target item. If the target action type is None, the source action type will be transferred to the target without being changed. If the source and target action types are both defined to something other than None, the action code of the target is changed to Modify. Otherwise, the target action code is unchanged.

These action types are the default for the relationship type. In a mapping rule, you can either use the default from the relationship type or you can define specific rules for a named relationship to be used for the mapping rule.

About the Calculate Service Order Transformation Sequence

The transformation sequence (CalculateServiceOrder) that is created by the OSM design pattern for Calculate Service Order contains the following transformation stages. These stages process order items based on an order item hierarchy. See "[Modeling Order Item Hierarchies](#)" for more information about the way order items can be arranged in hierarchies. You can edit these stages using Design Studio, if you need the transformation to work differently.

1. **ProcessPrimaryRelationships:** This stage creates transformed order items from original order items. Parameters from the original order item are also mapped to parameters on the transformed order item.
2. **ProcessDescendantItems:** This stage looks at child order items of the original order items and uses them to provide auxiliary data on the transformed order items. This can happen in two ways: the child order item itself may map to a data element on the transformed order item, or parameters from the child order item may map to parameters on the transformed order item. The child order items considered in this stage are not only the immediate children of the original order item, but also their children, to the bottom of the order item hierarchy.
3. **ProcessSiblingItems:** This stage is similar to the **ProcessDescendantItems** stage, except that the order items that are contributing data to the transformed order item are the siblings, rather than the descendants, of the original order item. As in the **ProcessDescendantItems** stage, the order items can provide auxiliary data by the sibling order item mapping to a data element on the transformed order item, or by parameters from the sibling order item mapping to parameters on the transformed order item.
4. **ProcessAncestorItems:** This stage is also similar to the **ProcessDescendantItems** stage. In this stage, the order items considered are the parent order items instead of the children. As in the **ProcessDescendantItems** stage, the order items can provide auxiliary

data by the parent order item mapping to a data element on the transformed order item, or by parameters from the parent order item mapping to parameters on the transformed order item. The parent order items considered in this stage are not only the immediate parents of the original order item, but also their parents, to the top of the order item hierarchy.

User-Created Entities for Calculate Service Order

In addition to the entities created by the design patterns, you must also create entities with information specific to your implementation. Some of these entities are in the conceptual model, and some are in OSM.

In the conceptual model, you will need to model at least some of the following:

- Products
- Customer Facing Services
- Resources
- Resource Facing Services
- Actions
- Action Codes
- Data elements

In OSM, you will need to model all of the following:

- Order item specifications for the original (source) and transformed (target) order items
- Transformation manager
- Mapping rules
- Order item parameter bindings: OSM has a design pattern to facilitate creating these bindings

Modeling OTM Without Calculate Service Order

If the supplied Calculate Service Order order transformation does not transform the order items the way you need, to such an extent that you do not think that editing the supplied entities would work for your situation, you have the option of configuring the order transformation manager from scratch instead.

To configure the order transformation manager if you are not using Calculate Service Order:

1. Model conceptual model entities:
 - a. Create a provider function.
 - b. Create relationship types.
 - c. Create one or more functional areas.
 - d. Create a domain in the conceptual model.
 - e. Model customer-facing services in the conceptual model.
 - f. Model products in the conceptual model.
 - g. Model named relationships in the conceptual model.
 - h. Add the products to the domain in the conceptual model.
 - i. Model a provider function in the conceptual model.

- j. Model data in the conceptual model, including keys for conceptual model entities.
For more information, see "Working with Conceptual Models" in *Modeling Basics*.
2. Model the order item specifications for the original and transformed order items:
 - a. Model the order item recognition. This is usually a parameter on the customer order line item, such as Fulfillment Item Code.
 - b. Model order item properties, including a property for order item recognition, a property to contain dynamic parameters created by the order item parameter binding, and properties for the order item ID and action.
3. Model order item parameter bindings to create typed and named parameters from parameters that may have been in name/value pairs in the incoming customer order line item.
4. Model mapping rules. These rules create order items and order item parameters on transformed order items based on original order items (that is, the order items and parameters from the customer order). The following types of mappings are available:
 - Entity-to-entity mapping: This creates a new transformed order item from an original order item. For example, you can use this to create a transformed order item representing a line from an original order item representing a major service.
 - Attribute-to-attribute mapping: This type of mapping creates new parameters on the transformed order item based on parameters on the original order item.
 - Entity-to-attribute mapping: This type of mapping creates new parameters on the transformed order item based on the presence of particular original order items. For example, an original order item representing a feature might be mapped to a parameter for that feature on an order item representing a new line.
5. Model a transformation sequence. This involves modeling a series of transformation stages. Each transformation stage includes the following steps:
 - a. Identify context order items for the transformation stage. These nodes are the original order items that will be available for transformation. You can select these nodes either by selecting an order item property that the original order items will have in common or by defining an XQuery expression to select them.
 - b. Identify related order items for the transformation stage. These order items will be able to contribute data to the transformed order items. You can select these nodes either by their relation to the context order items (parent, sibling, child) or using an XQuery expression. The relationships between the order items will be based on the physical order item hierarchy defined in the order item specification.
 - c. Select the relationship and relationship type that will be available to the transformation stage. For example, the transformation stage may be set up to include a Primary relationship between the Broadband product and the BroadbandInternetAccess customer-facing service.
 - d. Determine whether the stage should be conditional, and if so, write a condition for it.
6. Create a transformation manager that links the service domains and transformation sequences that you have created.

Modeling Processes and Tasks

This chapter describes how to model process, rules, and tasks in an Oracle Communications Order and Service Management (OSM) solution.

Overview of Processes and Tasks

The Process editor in Oracle Communications Service Catalog and Design - Design Studio is where you define the flow of tasks for a particular process. Processes have a single entry point and one or more exit points. When you create the process structure, you must place the tasks in the order in which the process is to complete them.

In addition to running tasks and subprocesses, you can control how a process runs; for example, specify to delay processing a task or create multiple possible transitions from one task to another based on task status.

Order processes can contain automated tasks, manual tasks, and task status transitions from one task to another task, as well as other process actions such as task transition delays, joins, redirects, rules, subprocesses, and end process points.

A **task** is a specific activity that must be carried out to complete the order; for example, if an order needs to verify that an ADSL service was activated, you might model a task named Verify ADSL Service. Tasks can be manual or automated. Manual tasks must be processed by an order manager, using the Task web client. Automated tasks run automatically with no manual intervention.

OSM also provides specialized automated task types called the activation task for communicating with Oracle Communications ASAP and the transformation task for initiating the order transformation manager functionality from within a process flow.

Modeling Processes

The following sections provide information about modeling processes.

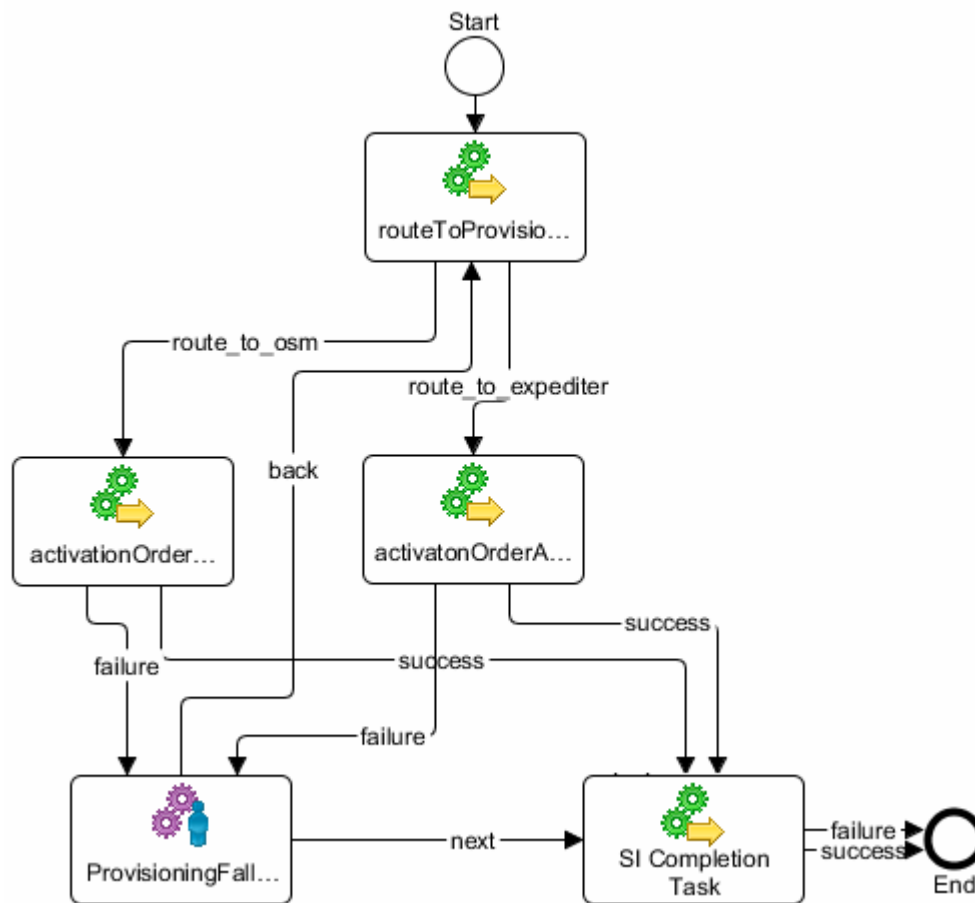
About Process Flows

Process flows define the sequence of tasks that the process performs. You can design flows for specific scenarios, including:

- A flow that ends in a successful process completion (Success) or a process failure (Failure).
- Flows for various activities, such as Cancel, Next, and Back.

[Figure 7-1](#) shows how flows appear in a process in Design Studio. In this figure, flows are labeled with the task status; for example, **route_to_osm**.

Figure 7-1 Process Flows in Design Studio



You can control flows in the following ways:

- You can use an order rule to apply conditions that must be met before the flow can continue.
- You can ensure that the system verifies that mandatory fields are present when a task completes. (This option is not available for tasks with a Rollback status.)
- You can specify a reporting status to display in an OSM web client. This status is tracked in the web client's OSM history.

[Figure 7-2](#) shows flow properties in Design Studio.

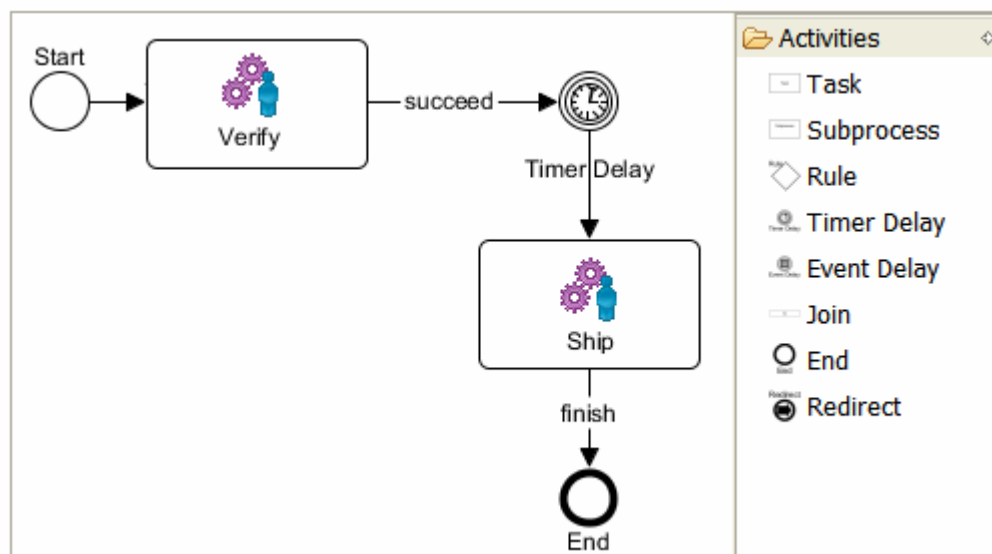
Figure 7-2 Flow Properties

General routeToProvisioningTask to activationOrderAdslRegion2Task		
Events	Property	Value
	Condition	
	From	routeToProvisioningTask
	Mandatory Check	True
	Reporting Status	
	Status	route_to_osm
	To	activationOrderAdslRegion2Task

Adding Process Activities

You use process activities to design how the process runs. [Figure 7-3](#) shows the Activities options in Design Studio. The example process includes a timer delay between the two tasks.

Figure 7-3 Process Activities Options in Design Studio



In addition to the tasks and subprocesses that the process runs, you can control the process by using the following:

- Rules
- Timer delays
- Event delays
- Joins
- Ends
- Redirects

Rules evaluate a condition and then specify the next step in the process. For example, a rule task might evaluate the data that describes the geographic region of the order and branch the process appropriately. Rule tasks perform as follows:

- They typically read and evaluate data to determine what to do.
- They always evaluate to true or false.
- They are always run automatically, with no manual involvement.

Timer delays delay the process until a rule evaluates to true. Timer delays perform as follows:

- The rule is evaluated at specified timed intervals.
- The data evaluated in the rule must be data that is included in the order.
- The rule always evaluates to true or false.
- The delay is always run automatically, with no manual involvement.

Event delays delay the process until a rule evaluates to true. Event delays perform as follows:

- The rule is evaluated only when the data specified in the rule changes.
- The data evaluated in the rule must be data that is included in the order.
- The rule always evaluate to true or false.
- The delay is always run by OSM, with no manual involvement.

Joins combine a set of flows into a single flow. (Process flows define the sequence of tasks that the process performs. See "[About Process Flows](#)" for more information.) The unified flow can join flows based on all transitions completing or any one transition completing (by selecting **All** or selecting **Any**). Selecting **Any** will create one instance of the flow for each incoming transition.

Ends stop the process from continuing.

Redirects redirect the process to another task in the same process or to a different process.

Note

Timer and event delays are not used during amendment processing.

Configuring Subprocesses

When you model subprocesses, you specify the following properties:

- If you want the associated tasks to appear in the Process History window in the Task web client.
- The pivot data element on which OSM spawns individual subprocess instances. For example, if you have subprocess that creates an email address for every person in a list, you might select the **Person** data element as the pivot data element, so the subprocess spawns an instance for each person. See "[Generating Multiple Task Instances from a Multi-Instance Field](#)" for more information.
- How to display the associated tasks in the Task web client. For example, you can display them sequentially, sorted, or unsorted.

- The process to run, based on rules. The rules in an order control how various actions take place; for example, when to trigger a jeopardy notification and how delays in the order process should be handled.
- How the subprocess handles exceptions. For example, you might have a process called `create_vpn`. Within that process, there is a subprocess called `validate_address`. The subprocess `validate_address` can throw an exception when an address is invalid. Using the exception mapping functionality, you can instruct the parent process and subprocesses to take specific actions when the subprocesses throw exceptions. Exception mapping enables you to indicate whether the parent process `create_vpn` should terminate all of the invoked instances, terminate only the offending instance, or ignore the exception altogether.

Understanding Parallel Process Flows

There are two ways to model parallel processes:

- Subprocesses branching from a task. This allows multiple tasks to run within the same time frame. Parallel flows can be rejoined at an appropriate point if needed. Typically, there are no dependencies defined between parallel flows, but whether these tasks actually run simultaneously depends on the order data, how order tasks are fulfilled, and other factors.
- Subprocesses running from a pivot data element. Multi-instance subprocesses are subprocesses that can be instantiated multiple times. When a subprocess has a pivot data element defined, multiple instances of the subprocess, running in parallel, are created. For example, if the pivot data element for a subprocess is defined as **interested_party**, and an order contains three instances of `interested_party`, each containing a different person's name and contact information, OSM creates three separate instances of the subprocess, one for each set of data.

When planning your order specifications, give careful consideration to which data you make available to each parallel process. Excessive and unnecessary data can have negative impacts on performance, and on usability if manual tasks are involved. Also, make sure to flag data as non-significant if the data is not needed for revision orders. By default, OSM assumes that all data is significant.

About Amendments and Multi-Instance Subprocesses

An amendment to an order on which some of the data affecting a multi-instance subprocess has changed can cause *all* subprocess instances to be redone, instead of only directly affected subprocesses to be redone. This can result in unneeded processing for the subprocesses with no data changes.

In amendment processing with multi-instance subprocesses, it is important to contain compensation to only the subprocess instances that require compensation. This is achieved by specifying a key. You specify a key in the **Key** subtab on the **Order Template Node** editor for the data element specified as the pivot data element of the subprocess in the order template. When a key is specified for a subprocess, OSM maps the revised data to the current data using the key field and redoes only the subprocess that was affected.

About Order Rules in Processes and Notifications

Order rules control how various actions take place; for example, when to trigger a jeopardy notification and how delays in the order process should be handled. Rules are used in process flow decisions, conditional transitions, subprocess logic, delay activities, jeopardies, and events.

OSM evaluates order rules by comparing data to data, or data to a fixed value. [Figure 7-4](#) shows an order rule in the Design Studio Order editor Rules tab. This rule identifies residential customers in a specific city. This is an example of a rule that might be used to send a fallout notification to a regional fallout manager.

Figure 7-4 Example of an Order Rule Defined in Design Studio

The screenshot shows the 'Order : OsmCentralOMExampleOrder' window. The 'Display Name' is 'OsmCentralOMExampleOrder'. The 'Rules' tab is active, showing a list of rules on the left: 'dslRegion1Rule', 'dslRegion2Rule' (selected), and 'null_rule'. The 'Definition' tab is selected, showing a table with the following data:

Condition	Data	Operator	Data/Value
	/CustomerDetails/city	=	Sao Paulo
And	/AccountDetails/category	=	Residential

Buttons at the bottom include 'Rename', 'Refresh', 'Properties', 'Remove', and 'Add'.

Modeling Order Rules in Notifications

All jeopardy notifications and most event notifications use order rules to determine if the notification should be triggered. (Event notifications that are used only for running an automation plug-in do not use order rules.)

[Figure 7-5](#) shows an example of a rule defined in Design Studio. This rule finds the city that the customer lives in and the type of account, (Business or Residential). When the jeopardy notification uses this rule, the notification is sent only if the order came from a residential customer in Sao Paulo.

Figure 7-5 Rule Example

This screenshot is identical to the one in Figure 7-4, showing the 'Order : OsmCentralOMExampleOrder' window with the 'Rules' tab active and the 'Definition' tab selected. The table shows the same conditions: '/CustomerDetails/city = Sao Paulo' and 'And /AccountDetails/category = Residential'.

You can use rules such as the one shown in [Figure 7-5](#) to route notifications to specific roles. For example, you can combine rules and roles as follows:

Table 7-1 Example Rule and Role Combinations

Notification Type	Triggered By	Rule Specifies	Sent to Role
Notification_Residential	Expected duration exceeded	Residential account	Residential
Notification_Business	Expected duration exceeded	Business account	Business

In this example, two identical notifications are created, both triggered by the order processing time exceeding the expected duration. If the order is for a residential account, the notification is triggered and sent to the role that handles residential accounts.

OSM uses a system-based **null_rule**. This rule always evaluates to true. Therefore, if you do not specify a rule for a notification, the **null_rule** is used; because it is set to true, the notification is triggered. If you do not specify any conditions to trigger the notification, and the notification uses the **null_rule**, the notification is triggered every time it is polled.

Note

The polling interval cannot be changed at run time.

See "[About Order Rules in Processes and Notifications](#)" for more information about rules.

Using the System Date in Delays

You can create a rule that uses the system date as part of a condition. For example, you can create a rule used in a delay that delays a task transition until the system date is at least the value of a particular order data element of the **dateTime** data type. [Figure 7-6](#) shows a rule that triggers when the system date is at least the value of the date when a particular poll is run.

Figure 7-6 Using the System Date in a Rule

See "[Adding Process Activities](#)" for more information about delays in process flows.

Process and Task Design and Data Considerations for Compensation

There are aspects of compensation that you need to consider when you are designing data, tasks, and processes.

Order Perspectives and Data Elements in Compensation

There are some aspects of compensation that you should consider when designing your processes. Compensation takes place using the data in the contemporary order perspective, but must be reconciled with the data in the real-time order perspective. (For more information about the different order perspectives, see "[About Order-Level and Task-Level Compensation Analysis](#).")

The issue relates to data elements that have been added in tasks that are later in the process than the task currently being compensated. The data that has been added is not present in the contemporary order perspective, since it was not present when the task performed its do operation. However, it is present in the real-time order perspective. If the redo operation checks whether the data element exists, it will be checking the contemporary perspective and will not find it. This will cause the redo operation to attempt to add the data element instead of updating it, which will cause problems when the data is reconciled with the real-time order perspective.

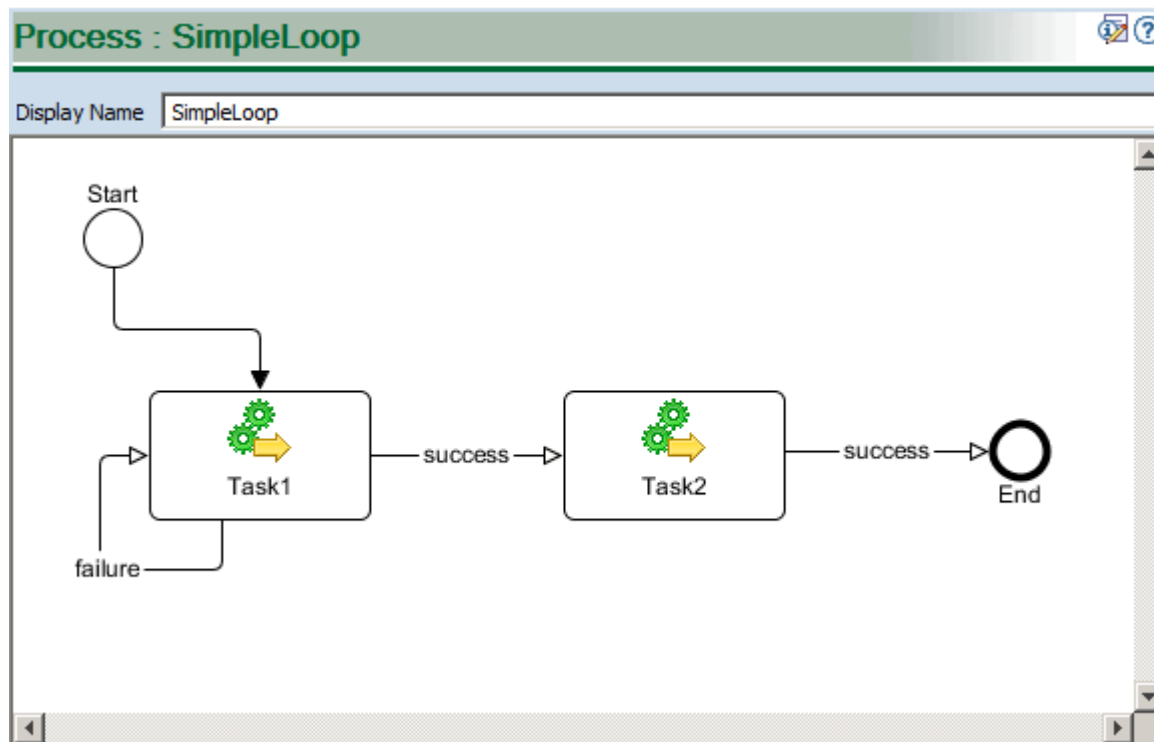
To avoid this situation, you should create any needed data elements before executing tasks that may be compensated. If the data is order-level data, you should initialize the data in the creation task for the order. If the data is function-level data, initialize the data needed by the process in a task that is run early in the process, before tasks that may be compensated.

Effects of Process Loops on Compensation

When you have loops in your OSM processes that cause your tasks to run multiple times and the process is compensated, each instance of the task that ran will be compensated. If entire sub-processes are being looped, this can cause a large number of tasks to require compensation.

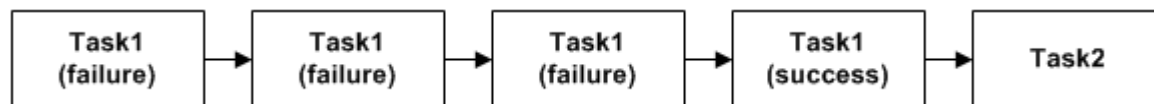
For example, consider the process in [Figure 7-7](#):

Figure 7-7 Simple Loop Process



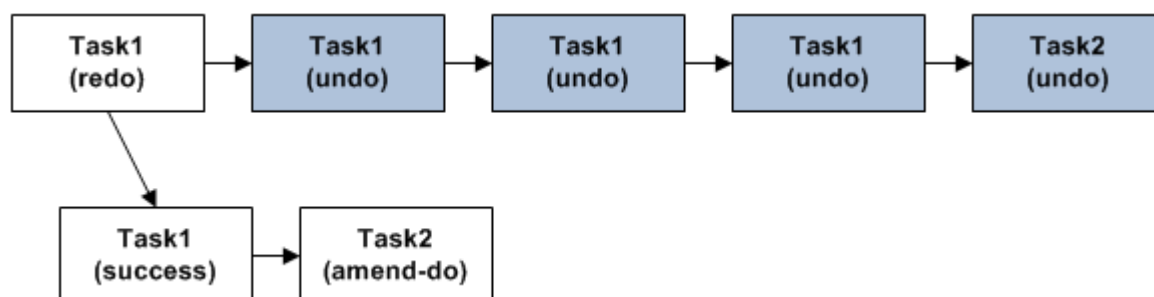
In this very simplified process, Task1 can run multiple times if it fails. In our current example, it is run four times: three times exiting with failure and once with success, as shown in [Figure 7-8](#).

Figure 7-8 Example of Initial Simple Loop Process Sequence



If the process needs to be compensated, the task will first be run once in redo mode. If this is successful, it will make the rest of the initial flow obsolete, so the tasks remaining in that flow would be run in undo mode, as shown in [Figure 7-9](#).

Figure 7-9 Example of Compensation of Simple Loop Process



Then, in the new branch of the process, Task2 will also be run in amend-do mode.

This example shows that while looping inside a process is supported by OSM, solution designers must carefully consider the implications of such loops when OSM compensates them as a result of an amendment. Most solutions include more complicated loops with more tasks per iteration, so you need to consider the impact that looped processes will have on the performance of your overall solution.

Modeling Tasks Entities Common to All Task Types

The following sections provide information about modeling task entities common to all task types.

Modeling Task States

All OSM tasks use states that determine various milestones in the progress of a task. The default task states are:

- **Received:** The task has been received in the system and is waiting to be accepted by a user (normally automatic for automated tasks) or assigned to a user (only in manual tasks).
- **Accepted:** The assigned user (system user account or a manual operator's user account). The task is locked so that it cannot be modified or completed by other users.
- **Completed:** The task is finished.
- **Assigned:** (Manual tasks only) The task has been assigned to a user.
- **Create Activation Work order Failed:** (Activation Task only) The task attempted to create a work order in the activation system but work order creation failed.

These tasks are mandatory and cannot be removed, but you can create custom task states.

Task states are important because they often trigger various functionality. For example, automation task automation plug-ins only run the task is in the Accepted state. You can configure task-level events to trigger when a task state is reached.

Modeling Task Permissions and Execution Modes

When you model tasks, you can specify which roles can perform which task execution modes (Do, Redo, Undo, Failed-Do, Failed-Redo, and Failed-Undo). For example, you may want to configure a specific role for normal Do, Redo, and Undo execution modes with a second role for fallout management that also operates in fallout execution modes. OSM users that are part of the fallout workgroup can work on failed automated and manual tasks. For more information about task execution modes and change order management, see "[About Task Execution Modes](#)".

[Figure 7-10](#) shows roles used in a task specification.

Figure 7-10 Task Permissions

Task Permission						
Role Name	Do	Redo	Undo	Failed-Do	Failed-Redo	Failed-Undo
DefaultRole	☒	☒	☒	☒	☒	☒
ProvisionRole	☒	☒	☒	☐	☐	☐

About Normal and Fallout Execution Modes and Task States

OSM provides the following execution mode groups:

- **Normal:** Task execution modes that run in normal mode include the Do, Undo, Redo, and Amend-Do modes for normal task processing activities.
- **Fallout:** Task execution modes that run in the fallout mode include Do in Fallout, Undo in Fallout, Redo in Fallout, and Amend-Do in Fallout modes for troubleshooting tasks that have failed.

Note

If an amendment is received while a task is in a fallout execution mode, the following will happen:

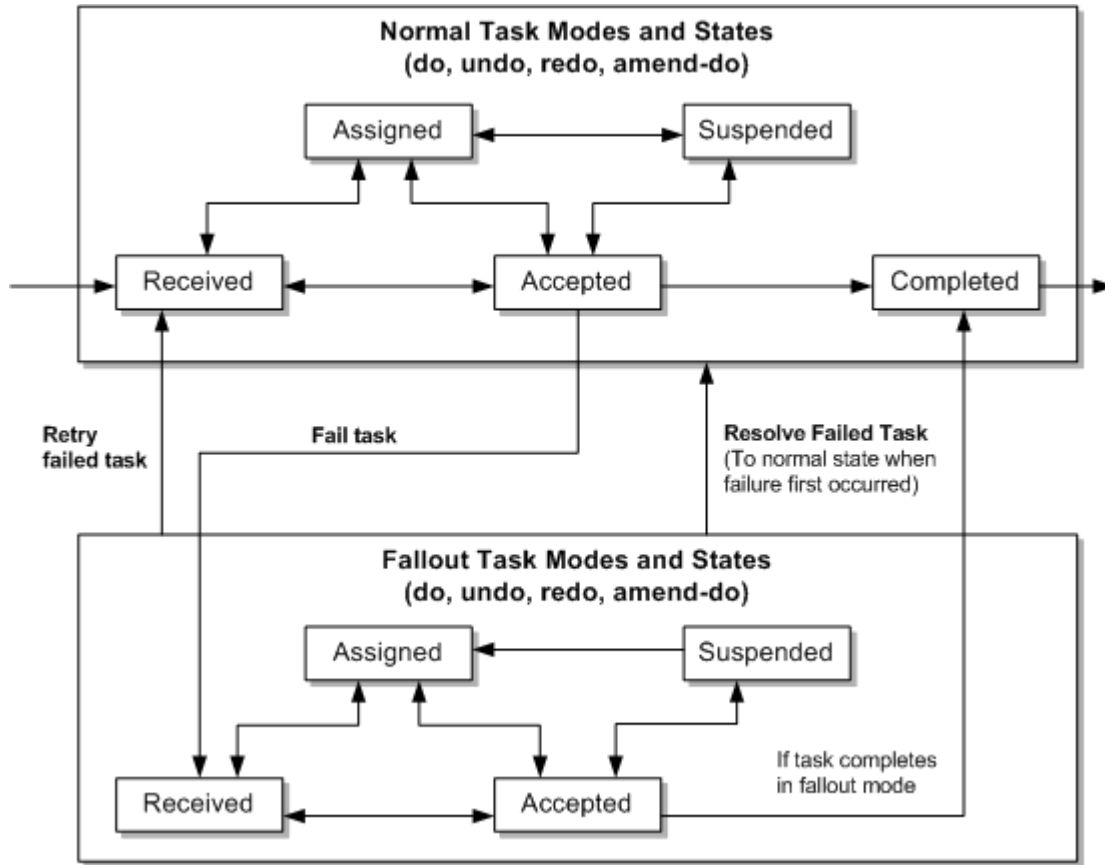
- If the task is not configured to be compensated if it is in progress, the execution mode of the task will not change as a result of the amendment order.
- If the task is configured to be compensated if it is in progress, and the amendment contains changes to significant data:
 - If the task is still needed after the changes to the order from the amendment are considered, it will transition automatically to (normal) Redo mode.
 - If the task is no longer needed after the changes to the order from the amendment are considered, it will transition automatically to (normal) Undo mode.

In both of these cases, your automation code (for either Redo or Undo execution mode) should contain a check to see if the task has been in a fallout execution mode, and also whatever code is needed to resolve any actions that have been taken in the fallout execution mode. For example, if your automation for Do in Fallout mode opens a trouble ticket, your Redo automation should check to see whether it needs to close a trouble ticket.

- If the amendment order contains no changes to significant data, the execution mode of the task will not change as a result of the amendment order.

[Figure 7-11](#) shows how OSM transitions tasks to the fallout execution modes and back to normal execution modes and how these modes relate to task states.

Figure 7-11 Normal and Fallout Execution Mode and Task States



The following shows how the tasks in [Figure 7-11](#) processes through each state in a Normal execution mode:

1. When OSM starts a task, it enters into the **Received** state in a normal execution mode.
2. In Manual tasks, an operator can optionally assign the task to themselves or have the task be assigned to them. When the task is assigned, it enters into the **Assigned** state. Automation tasks do not use this state.
3. When an operator or the system begins working on the manual or automated task, the task enters into the **Accepted** state.
4. While the task is in the **Accepted** state, the system or the operator can:
 - Move the task to a customer defined state like the **Suspended** state for a business reason defined for the task. From the **Suspended** state, the system or the operator can return the task to the **Accepted** state or move it to the **Assigned** state.
 - Move the task to the **Completed** state by completing the task.
 - Fail the task. A failed task automatically moves to the **Received** state in a fallout execution mode. You can fail a task in the following ways:
 - Task web client for manual tasks
 - OSM Java API for automated tasks in automation plug-in code.
 - OSM XML API for manual and automated tasks in automation plug-in code.

The following shows how the tasks in [Figure 7-11](#) processes through each state in a fallout execution mode:

1. A task enters a failed execution mode in the **Received** state from a normal execution mode in the **Accepted** state.
2. In Manual tasks, an operator must assign the task to themselves or have the task be assigned to them. When the task is assigned, it enters into the **Assigned** state. Automation tasks do not use this state.
3. When an operator or the system begins working on the manual or automated task, the task enters into the **Accepted** state.
4. While the failed task is in the **Accepted** state, the system or the operator can:
 - Move the task to a customer defined state like the **Suspended** state for a business reason defined for the task. From the **Suspended** state, the system or the operator can return the task to the **Accepted** state or move it to the **Assigned** state.
 - Move the task to the normal execution mode **Completed** state to complete the task.
 - Retry the failed task. Retrying a task moves the task back to the normal execution mode to the **Received** state to retry the task from the beginning. You can retry a failed task in the following ways:
 - Task web client for one task or for all tasks on the order
 - Order Management web client for all failed tasks on a specific order component within an order, for all failed tasks on each order, or for all failed tasks of many orders as a job control order. You cannot retry a specific task type in bulk across multiple orders using a job control order.
 - OSM Java API in automation plug-in code
 - OSM XML API in automation plug-in code
 - OSM Web Service API operation for all failed tasks on a specific order component within an order, for all failed tasks on each order, or for all failed tasks of many orders as a job control order. You cannot retry a specific task type in bulk across multiple orders using a job control order.
 - Resolve the task. Resolving a task moves the task back to the original normal execution mode and state it had been in before failing. You can resolve a failed task in the following ways:
 - Task web client for one task or for all tasks on the order
 - Order Management web client for all failed tasks on a specific order component within an order, for all failed tasks on each order, or for all failed tasks of many orders as a job control order. You cannot resolve a specific task type in bulk across multiple orders using a job control order.
 - OSM Java API in automation plug-in code
 - OSM XML API in automation plug-in code
 - OSM Web Service API operation for all failed tasks on a specific order component within an order, for all failed tasks on each order, or for all failed tasks of many orders as a job control order. You cannot resolve a specific task type in bulk across multiple orders using a job control order.

Modeling Task Status Transitions

You model task status the define how a task completes and to determine what the next task is in the process flow. You define the status transitions available to a task in the task editor Status/Status tab, and then you apply the status transition of process flows you create between tasks.

You can use the default status transitions defined in manual, automated, activation, and transformation tasks or you can create new status transitions that may better describe what is happening during a status transition from one task to another.

The default statuses for a manual task are:

- Back
- Cancel
- Finish
- Next

The default statuses for a automated and transformation task are:

- Failure
- Success

The default statuses for a activation task are:

- Success
- Activation Failed
- Updated OSM Order Failed

You can also select from the set of additional predefined statuses (Delete, False, Rollback, Submit, Failed, and True), and you can also define your own.

You can also use constraint behaviors with status transitions and manual tasks to better control when an operator can transition from one task to another task. See "[Using the Constraint Behavior to Validate Data](#)".

Specifying the Expected Task Duration

You can specify the expected length of time to complete a task. This information can be used to trigger jeopardy notifications and for reporting. See "[Modeling Jeopardy and Notifications](#)" for more information. This information is also used by OSM to calculate the order component duration.

You can specify the length of time in weeks, days, hours, minutes, and seconds. The default is one day.

You can also calculate the duration based on your workgroup calendars. If you have more than one workgroup with different calendars all responsible for the same task, the calculation is based on the first available workgroup that has access to the task. This ensures that a the task only exceeds it's duration based on the workgroup calendar time.

For example, there might be a task with an expected duration of two hours, and the workgroup that processes the task only works 9 AM - 5 PM Monday to Friday as indicated on their workgroup calendar. If such a task is received at 4 PM on Friday, then the expected duration of the task will expire at 10 AM Monday, as there was only two hours of the workgroup calendar

time that had elapsed (4-5 PM Friday, then 9-10 AM Monday). This ensures that notifications and jeopardies are triggered appropriately.

See *OSM Task Web Client User's Guide* for more information.

Specifying the Task Priority

Task priority is the same as the order priority unless a priority offset is defined. Priority of orders and their tasks becomes effective when the system is under heavy load, ensuring that high priority orders and tasks are not starved of resources by lower priority orders and tasks.

You define the task priority as an offset from the priority of the order itself. This specifies the priority of the task in relation to other tasks in the order.

For example, if the order is created at priority 6, and this task is assigned a priority offset of -2, then this task would run at priority 4 while tasks in the order with no offset would run at priority 6. Similarly, you could assign a task a priority offset of +2, which would mean that the task would run at a slightly higher priority than other tasks in the order.

See "[Modeling Order Priority](#)" for more information about order priority.

About Extending Tasks

You can create a new task by extending from an existing task. The new task inherits all of the data, tasks, rules, and behaviors of the base task from which it was extended. Changing something on the base task is reflected in all tasks extending from it.

For example, if you have multiple tasks that all require the same data subset, you can create a base task that contains this data, then extend from this task to create as many new tasks as necessary. You can add new data and behaviors to each of the new tasks to create unique task and behavior functionality. Extending tasks can significantly reduce duplication and maintenance.

About Task Types

The following sections provide information about different task types.

Modeling Automated Tasks

You add automated tasks to processes whenever you need a task that can run automation plug-in instances without user intervention. Automated task automation plug-ins can do various tasks such as connect to a database to query data, transform data, or communicate with external fulfillment systems. OSM runs the automation plug-in instances on an automated task whenever the automated task transitions to the received state in a normal or fallout execution mode (see "[About Normal and Fallout Execution Modes and Task States](#)").

Automation plug-in user task can perform multiple tasks based on the code you write in the automation plug-ins states. Among the many functions you can implement in the code, you must also ensure that the automation plug-ins manage task status transitions to complete a task and move the task to another task on the process (see "[Modeling Task Status Transitions](#)"). You can also specify task execution modes that determine what roles (workgroups) can perform the task and in what ways (see "[About Normal and Fallout Execution Modes and Task States](#)"). If an automated task does not have any automation plug-ins that can run in fallout execution modes, and then the automated task runs as a manual task so long as there are users associated with roles designated to manage the fallout execution modes (see "[Modeling Task Permissions and Execution Modes](#)").

Automated tasks can also trigger a jeopardy notifications based on the duration of the task and event notifications based on task state changes (see "[Modeling Jeopardy and Notifications](#)").

About Automation Plug-in and Automated Tasks

When you add an automated task to a process, you must associate at least one automation plug-in for the task. To associate an automation plug-in for a task, you open the automated task entity in the Automated Task editor, and add the plug-in to the task in the Automation tab. When you deploy your cartridge to the run-time environment, the OSM server detects a task that has an automation plug-in associated with it, the server triggers the plug-in to perform its processing.

An automated task might have only a single automation plug-in associated with it. For example, you might associate a built-in Automator plug-in with the task to interrogate the task data, perform some calculation, update the order data, and transition the task. In this example, as soon as the Automator plug-in has finished processing, it updates the task with an exit status, and the OSM server moves to the next task.

An automated task can have multiple associated automation plug-ins. For example, you might want to associate multiple plug-ins with a task to represent conversations with external systems. You can associate a built-in Sender plug-in to receive the task data and send it to an external system for processing. That external system might send an acknowledgement back to a queue, where a second Automator plug-in--one that is defined as an external event receiver (it receives data from external system queues)--consumes the reply and updates the order data with the response. A third Sender plug-in might send the external system a message to begin processing, and a fourth Automator plug-in can receive the "processing complete" message from the external system, update the order, and transition the task.

See "[About Automation Plug-ins](#)" for more information.

Completing an Automation Task That Handles Concurrent Status Updates

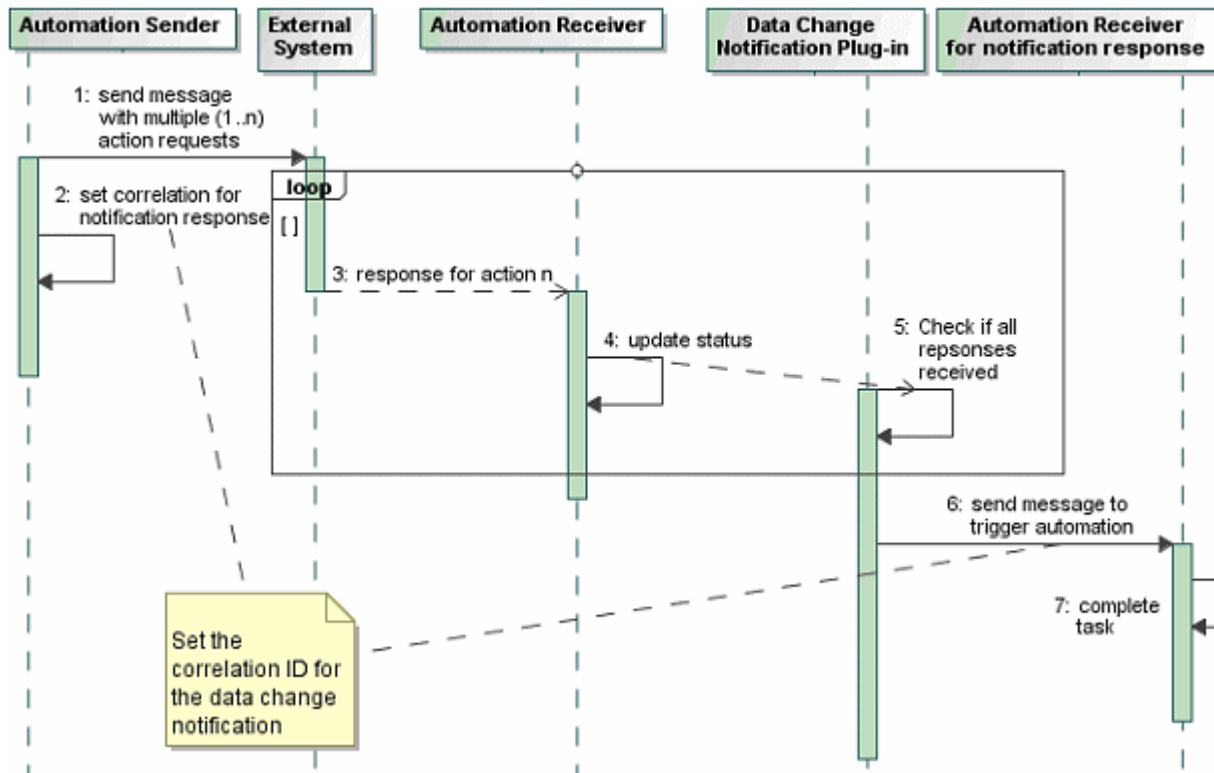
An automated task can process multiple responses from external systems. For example, an activation task might receive the status for each service on the activation request. The activation task needs this information to determine when the activation has been completed by the external system, at which point the task can transition to the Completed state.

- The external system can include data that indicates that all of the requests have been completed. Typically, this is a message indicating that the response is the last response, and there will be no further messages.
- If the external system cannot report that the last request has been processed, the automation task must ensure that a response has been received for each request sent to the external system.

When OSM must determine the last response, there are special considerations for concurrent status updates. If the automated task needs to track the status of all responses, and multiple responses are processed concurrently, the automation receiver instances executing concurrently do not have visibility to status updates from the other receivers. The receiver may never run with the task data that contains all status updates and so never encounters a condition where it can complete the task.

This situation can be handled by configuring an automated notification plug-in that monitors the status fields and creates a notification whenever the data changes.

Figure 7-12 Sequence Diagram for Concurrent Status Update Notification Process



The notification plug-in is triggered every time the status field is updated by the automation receiver. The notification plug-in runs in a separate transaction after each receiver update, and can check the status responses to determine if all responses have been received for each action request. When all responses are received, the notification plug-in can generate a message to trigger an automation receiver. This receiver is correlated to the original sender by means of an ID set by the sender specifically for tracking the status updates. The receiver is then run with the task data that contains all of the status responses and it can complete the task.

Modeling Manual Tasks

You add manual tasks to processes whenever you need a task that requires direct user intervention. Users work with manual tasks in the OSM Task web client whenever a manual task transitions from the received state to the assigned state in a normal or fallout execution mode (see "[About Normal and Fallout Execution Modes and Task States](#)"). You assign manual tasks to OSM users in the following ways:

- **Manually:** The task appears in the OSM Task web client in the received state and an operator has the responsibility to assign the task to a user.
- **Automatically (pre-defined in Design Studio):** You can optionally choose a round robin task assignment algorithm that distributes tasks evenly between all users associated with the role (workgroup) that can work on the task, or load balancing task assignment algorithm that distributes tasks based on user workload.
- **Automatically (customized task assignment algorithm):** You can develop a custom task assignment algorithm using OSM's cartridge management tools. See "[Deploying a Custom Task Algorithm using the OSM Cartridge Management Tool](#)".

When an operator is working in a manual task, they must directly update task data in the OSM Task web client. You can add behaviors to manual task data that perform various function. For example:

- Performing calculations on numerical task data.
- Adding constraints on task data fields to validate the data that users enter. You can also use constraints to control whether a user can transition from one task to another.
- Making a field read-only.
- Making a field visible to some users only.

See "[Modeling Behaviors](#)" for more information about all behavior options that OSM provides.

Manual tasks user task states to managed the progress of the task (see "[Modeling Task States](#)") and task status transitions to move from one task to another task (see "[Modeling Task Status Transitions](#)"). You can also specify task execution modes that determine what roles (workgroups) can perform the task and in what ways (see "[About Normal and Fallout Execution Modes and Task States](#)").

Manual tasks can also trigger a jeopardy notifications based on the duration of the task and event notifications based on task state changes (see "[Modeling Jeopardy and Notifications](#)").

Manual tasks are often used when initially developing OSM solutions to better understand the what needs to happen in various points of an OSM solution. When solution developers have a better understanding of what a task is doing, they can then consider transforming the task into an automated task with associated automation plug-ins. In addition, you can insert manual tasks in a process that function as breakpoints for debugging. This allows you to control a process when you test it.

Deploying a Custom Task Algorithm using the OSM Cartridge Management Tool

The OSM Cartridge Management Tool is only applicable for traditional OSM deployments. To use the custom task algorithm in OSM cloud native, see "[Using a Custom Task Algorithm in OSM Cloud Native](#)".

In addition to the round robin or load balancing algorithms for assigning workgroups to tasks provided by OSM, you can create a custom task assignment algorithm that assigns tasks based on custom business logic. Before you can use OSM CMT to deploy a custom task assignment algorithm, ensure that:

- You can access and reference a WebLogic Server and ADF installation home directory from the OSM CMT build files. See *OSM Installation Guide* for version information.
- You must download and install Ant. See *OSM Installation Guide* for version information.
- You install the **SDK Tools** and the **SDK Samples** components using the OSM installer. You do not need to install the other options. See *OSM Installation Guide* for more information about using the OSM installer.
- You have created a custom task assignment algorithm. See the **SDK/Samples/TaskAssignment/code /CustomizedTaskAssignment.java** reference sample for more information about creating a custom task assignment algorithm.

To deploy a custom task algorithm to an OSM server using OSM CMT:

1. From a Windows command prompt or a UNIX terminal, go to `WLS_home/server/lib` (where `WLS_home` is the location of the base directory for the WebLogic Server core files).
2. Copy the following files required by OSM CMT to the `Ant_home/lib` folder (where `Ant_home` is the location of the Ant installation base directory).

- `WLS_home/server/lib/weblogic.jar`
 - `WLS_home/server/lib/wlthint3client.jar`
 - `WLS_home/modules/com.bea.core.descriptor.wl.jar`
 - `SDK/deploytool.jar`
 - `SDK/Automation/automationdeploy_bin/automation_plugins.jar`
 - `SDK/Automation/automationdeploy_bin/xmlparserv2.jar`
 - `SDK/Automation/automationdeploy_bin/commons-logging.jar`
3. Set the following environment variables and add them to the command shell's path:
 - **ANT_HOME**: The base directory of the Ant installation.
 - **JAVA_HOME**: The base directory of the JDK installation.

For example, for a UNIX or Linux Bash shell:

```
ANT_HOME=/home/user1/Middleware/modules/org.apache.ant_1.10.15
JAVA_HOME=/usr/bin/local/jdk-21.0.7
```

For example, for a Windows command prompt:

```
set ANT_HOME=c:\path\to\oracle\home\Middleware\modules\org.apache.ant_1.10.15
set JAVA_HOME=c:\path\to\oracle\home\Middleware\jdk-21.0.7
```

4. Open the **SDK/Samples/config/samples.properties** file.
5. Set the following variables:
 - Set **osm.root.dir** to the OSM installation base directory.
 - Set **oracle.home** to the Oracle Middleware products base directory.

For example, for a UNIX or Linux Bash shell:

```
/home/oracle/Oracle
```

For example, for a Windows command prompt:

```
C:/Oracle
```

6. Copy the custom task assignment algorithm file you created to **SDK/Samples/TaskAssignment/code**.
7. Open the **SDK/Samples/TaskAssignment/code/build.properties** file.
8. Set the following variables:
 - Set **weblogic.url** to the WebLogic Administration Server URL. The format is:
`t3://ip_address:port`
 where:
 - *ip_address* is the IP address for the WebLogic Administration Server.
 - *port* is the port number for the WebLogic Administration Server.
 - Set **weblogic.domain.server** to the name of the WebLogic Administration Server.
 - Set **weblogic.username** to the WebLogic Administration Server user name.
 - Set **webLogicLib** to the path to the `WLS_home/server/lib` folder.
 - Set **ejbname** to the Enterprise Java Bean (EJB) name for the task assignment behavior.

- Set **ejbclass** to the class name for the task assignment behavior.
- Set **jndiname** to the Java Naming and Directory Interface (JNDI) bind name for task assignment behavior.
- Set **targetfile** to the deploy target file name for a target file that does not contain a suffix like **.ear** or **.jar**.

Note

ejbname, **ejbclass**, **jndiname**, and **targetfile** are preconfigured to deploy the **SDK/Samples/TaskAssignment/code/CustomizedTaskAssignment.java** sample task assignment algorithm. Replace these default values with those for the custom task assignment algorithm.

9. Create and deploy a Design Studio cartridge that includes a manual task that you want to associate to the custom task assignment algorithm. You can associate the custom task assignment algorithm in the **Details** tab of the manual task using the **Assignment Algorithm** and **JNDI Name** fields. See "Task Editor Details Tab" in *Modeling OSM Processes* for more information.

Note

You can import the sample task assignment cartridge from **SDK/Samples/TaskAssignment/data/ taskassignment.xml**. For more information about importing an OSM model into Design Studio, see "Working with Existing OSM Models" *Modeling OSM Processes*.

10. From the **SDK/Samples/TaskAssignment/code** directory, at the Windows command prompt or UNIX shell, type:

```
ant
```

The Ant script begins to run.

11. When the ant script reaches Input **WebLogic Password for user weblogic ...**, enter the WebLogic Administration Server password.

The ant tool compiles, assembles, and deploys the custom task assignment algorithm to the OSM WebLogic Server.

Note

You can also individually compile, assemble, deploy, or undeploy using the following Ant commands:

```
ant compile
ant assemble
ant deploy
ant undeploy
```

Using a Custom Task Algorithm in OSM Cloud Native

To use a custom task algorithm in OSM cloud native, ensure that you have followed these steps:

- You have created a custom task assignment algorithm. See the **SDK/Samples/TaskAssignment/code /CustomizedTaskAssignment.java** reference sample for more information about creating a custom task assignment algorithm.
- Traditional deployment mechanisms do not apply in an OSM cloud native environment. To deploy an application to WebLogic in OSM cloud native, see the "Deploying Entities to an OSM WebLogic Domain" section in *OSM Cloud Native Guide*.
- Create and deploy a Design Studio cartridge that includes a manual task that you want to associate to the custom task assignment algorithm. You can associate the custom task assignment algorithm in the **Details** tab of the manual task using the Assignment Algorithm and JNDI Name fields. See Design Studio Help for more information.

Note

You can import the sample task assignment cartridge from **SDK/Samples/TaskAssignment/data/ taskassignment.xml**. For more information about importing an OSM model into Design Studio, see Design Studio Help.

Modeling Transformation Tasks

You can use a transformation task if you want to call the order transformation manager from a process instead of before the orchestration plan is generated. See "[Calling the Order Transformation Manager](#)" for more information. The transformation task is very much like an automated task, except that by default it has an appropriate automation plug-in defined for it and provides the ability to define the transformation manager to call.

Modeling Activation Tasks

Before you can model Activation tasks in Design Studio, you must install the Design Studio for Order and Service Management Integration feature. This feature includes the Design Studio for Activation feature for integrating with ASAP and IP Service Activator. To model activation tasks, you must also install the Design Studio for Activation feature.

1. OSM transforms order data into an operations support system through Java (OSS/J) message or a web service message and sends it to ASAP or to IP Service Activator. To model this, you configure service action request mapping, to map OSM data to ASAP data or to map OSM data to IP Service Activator data. See "[About Service Action Request Mapping](#)" for more information.
2. ASAP or IP Service Activator receives the data, activates the service, and returns a success or failure status to OSM. To allow OSM to handle the returned data, you model service action response mapping. See "[About Service Action Response Mapping](#)" for more information.

Other elements specific to activation tasks are:

- You can configure state and status transitions for completion events and exceptions returned by ASAP or IP Service Activator.
- You can configure how to handle amendment processing with activation tasks.

- If you are sending JMS OSS/J messages, Oracle recommends that you configure JMS store and forward (SAF) queues to manage the connection to ASAP or to manage the connection to IP Service Activator.
- If you are sending web service messages, Oracle recommends that you configure web service SAF queues to manage the connection to ASAP or to manage the connection to IP Service Activator.

About Service Action Request Mapping

You send fulfillment data to ASAP or to IP Service Activator as a service action request. To model a service action request, you map OSM header data (information that applies to the customer or to all order line items on the order) and OSM task data to the following service order activation data:

- Activation order header: Information that applies to the entire work order.
- Service action: Information that is required to activate a service.
- Global parameters: Information that you define once and which applies to multiple service actions.

About Service Action Response Mapping

After ASAP or IP Service Activator activates a service, it returns information to OSM. You create data structures in OSM to contain the response information returned from ASAP or IP Service Activator. For each event and exception returned by ASAP or IP Service Activator, you select the ASAP or IP Service Activator data that you want to retain, and then identify the OSM data structure to which that data is added. When ASAP or IP Service Activator returns an event or exception, OSM updates the order data with the ASAP or IP Service Activator data that you specified.

✓ Tip

The amount of response data from ASAP or IP Service Activator can be very large, though the data that is needed might be small. Parsing large amounts of ASAP or IP Service Activator response data can affect OSM performance. If you notice a reduction in OSM performance due to large amounts of ASAP or IP Service Activator response data, you can specify a condition on specific parameters to limit the ASAP or IP Service Activator response data.

About Activation Tasks and Amendment Processing

You can configure how to manage an activation task if the associated order undergoes amendment processing. The options are:

- Intervene manually.
- Do not perform any revision/amendment.
- Have OSM redo the activation task, using the previously defined request mapping.
- Have OSM redo the task, using different request mapping.

About State and Status Transition Mapping for Activation Tasks

You can configure state and status transitions to manage completion events (for example, **activation complete**) and errors returned by ASAP or returned by IP Service Activator. You can define multiple transitions to model different scenarios for variations in the data received from ASAP or received from IP Service Activator. For example, if an ASAP parameter or IP Service Activator parameter returns the value **DSL**, you may want the task to transition to a DSL task; when the same parameter returns the value **VOIP**, you want the task to transition to a different task.

You can define state transitions for user-defined states only; you cannot define transitions for system states, such as Received, Accepted, and Completed. At run time, OSM evaluates the conditions in the order and stops evaluating when a condition evaluates to true. Completion events and errors must include a default transition in case all specified conditions fail.

About Automation Plug-ins

You use automation plug-ins to implement specific business logic automatically. You can create automation plug-ins to update order data, complete order tasks with appropriate statuses, set process exceptions, react to system notifications and events, send requests to external systems, and process responses from external systems.

There are two basic types of delivered automation plug-ins, Sender and Automator. Each type can be implemented using XSLT or XQuery, and each type can be defined as an internal event receiver (the JMS message that triggers the call to the plug-in is generated by OSM), or as an external event receiver (the JMS message that triggers the call to the plug-in is generated by an external system).

- Automator plug-ins receive information from OSM or an external system, and then perform some work. Depending on how you configure the plug-in, it can also update the order data.
- Sender plug-ins receive information from OSM or from an external system. They perform some business logic, and may or may not update an order, depending on your configuration. Additionally, they can produce outgoing JMS or XML messages to an external system. When generating JMS messages, you can define JMS messages to connect to a topic or queue.

① Note

XQuery automation types cannot be implemented when using releases prior to OSM 7.0.

OSM assigns automated task plug-in instances to a user account specified in the plug-in **Properties** subtab **Details** subtab **Run As** field. The user account must belong to the OSM_automation WebLogic group. When you install OSM, the OSM installer automatically creates the oms-automation user that belongs to the OSM_automation group. You can use this user account to run automation plug-in instances or create new ones. You can also use the DEFAULT_AUTOMATION_USER model variable in the Run As field that you define at in the Order and Service Management Project editor **Model Variable** tab or in the Environment editor **Model Variables** tab.

When referring to an automation, the following meanings can apply:

- The automation plug-in code that you create and associate with an automation task in Design Studio.
- The instance of an automation plug-in that the OSM run-time server creates in response to an event that triggers an automation. OSM creates and reuses such instances as required when processing automated tasks. OSM maintains these plug-in instances even if the instance is no longer required and only creates additional plug-in instances when the current pool of instances are insufficient to handle the number of incoming orders. OSM only destroys automation plug-in instances in the following scenarios:
 - When you shut down the OSM server, OSM destroys all plug-in instances.
 - When you undeploy a cartridge, OSM destroys all plug-in instances associated with the undeployed cartridges.
 - When OSM detects an error condition in the instance, OSM destroys the instances.

See *OSM Developer's Guide* for detailed information about automated tasks and automation plug-ins.

Specifying Which Data to Provide to Automation Plug-ins

The data that is available for each automation plug-in should be the minimum subset of order data necessary for the plug-in to be performed. You can choose the data to provide to automation plug-ins using the following methods:

- Use the **task data** contained in an automation task to specify which data to provide to an automation plug-in.
- Use **query tasks** to specify which data to provide to an automation plug-in associated with order notification, events, and jeopardies. A query task is a manual task that is associated with a role that has permissions to use some or all order data to run an automation plug-in. See "[Modeling Query Tasks for Order Automation Plug-ins](#)" for more information.

Modeling Query Tasks for Order Automation Plug-ins

In automated tasks, the data that is available to automation plug-ins associated with automated task is already defined in the **Task Data** tab. However, automation plug-ins used with order notifications, events, and jeopardies do not have immediate access to this task data, and, as a result, must reference a manual task called a query task that defines the task data and behavior data available to the automation plug-in.

You can select any manual task as the query task. You can also create special tasks that are only used as query tasks. Their only function is to specify which data to provide to an automation plug-in.

[Figure 7-13](#) shows the **Permissions** tab in the Design Studio order editor. The upper screen shows the permissions for the provisioning role, with the provisioning function task as the query task. For the billing role, the billing function task is assigned as the query task.

Figure 7-13 Roles Assigned to Query Tasks

Display Name: OsmCentralOMExampleOrder

Roles

- DefaultRole
- ProvisionRole**
- ProvisioningUpdateRole
- SummaryRole
- BillingUpdateRole

Role Settings

Details Filters **Query Tasks**

Name	Summary	Detail	Default
ProvisioningFunctionTask	☐	☒	☒

Display Name: OsmCentralOMExampleOrder

Roles

- DefaultRole
- ProvisionRole
- ProvisioningUpdateRole
- SummaryRole
- BillingUpdateRole**

Role Settings

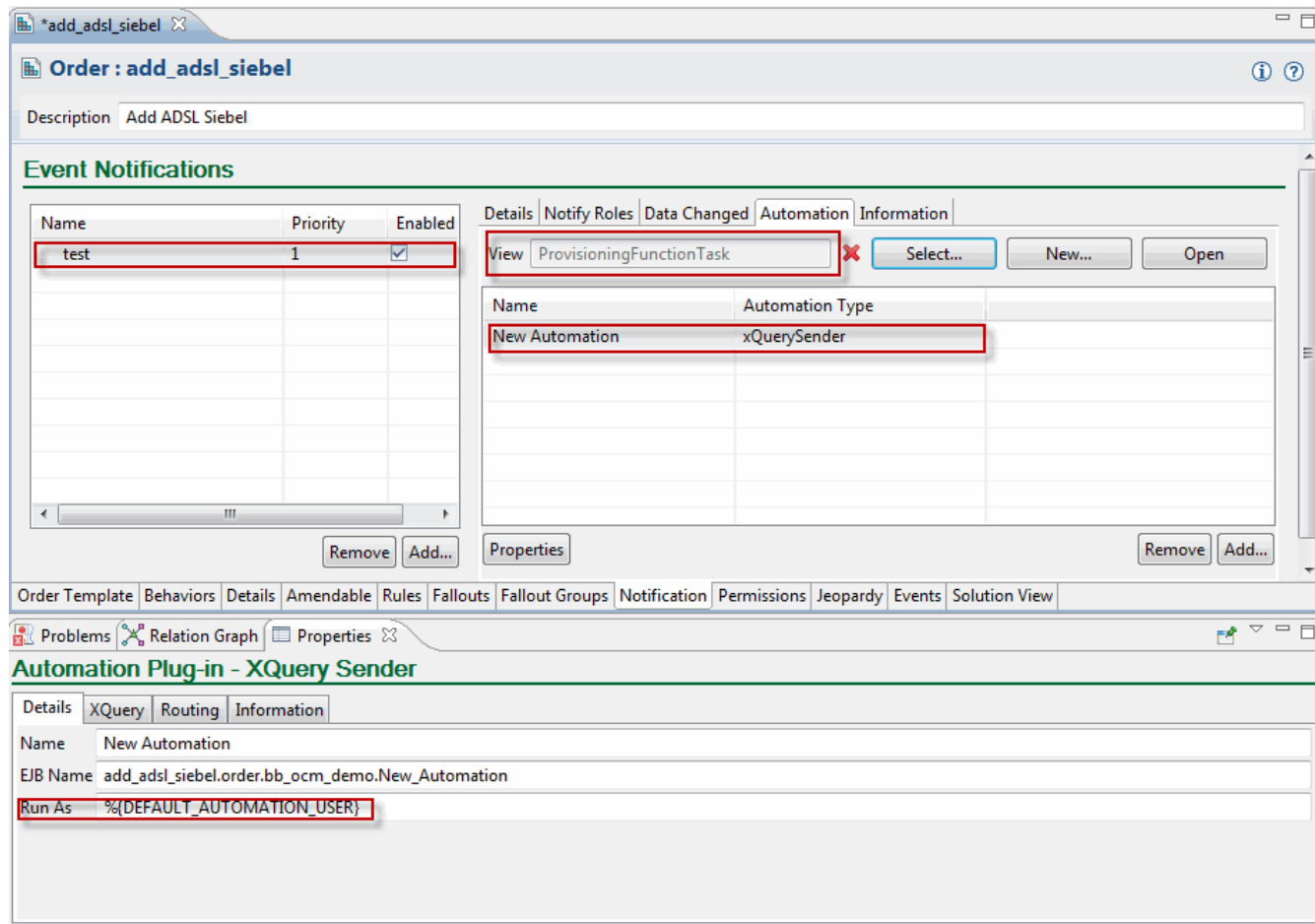
Details Filters **Query Tasks**

Name	Summary	Detail	Default
BillingFunctionTask	☐	☒	☒

To associate a query task with an automation plug-in, use the Default check box, as shown in [Figure 7-13](#).

[Figure 7-14](#) shows an event notification with an automation plug-in that uses the **ProvisioningFunctionTask** query task that is defined as the default query task for the provisioning role. This role must be associated with the **Run as OSM** user that runs the automation plug-in as shown in the Properties **Details** tab. For more information about associating roles to OSM users, see the *OSM Order Management Web Client User's Guide*.

Figure 7-14 Order Event Notification Automation Query Task



About Automation Message Correlation

Automation plug-ins defined as external event receivers are designed to process JMS messages from external systems. JMS messages are asynchronous, therefore external event receivers provide a method of correlating responses to requests previously delivered to enable you to map OSM orders to external system orders.

To correlate responses, the plug-in sets a property on the outbound JMS message, with name of the value set for correlation property in the **automationmap.xml** file, and a value decided by your business logic. For example, business logic might dictate that you correlate on a reference number. The external system copies the properties that you defined for the correlation on the request and includes that data in the response.

You can use the **Message Property Selector** field to filter messages placed on the queue and determine which automation to run. You define the **Message Property Selector** value as a boolean expression that is a String with a syntax similar to the where clause of an SQL select statement. For example, the syntax may be:

```
"salary>64000 and dept in ('eng','qa')"
```

When the condition evaluates to true, the message is picked up and processed by the automation that defined that condition.

In a second example, consider that an external system defines five order types and OSM defines a different automation to process each order type. Each automation defines a different **Message Property Selector**, such as `orderType=1`, `orderType=2`, and so forth. When a message is sent to the queue by the external system, and the message includes the `orderType` upon which the condition is based, the automation framework evaluates each condition until one evaluates to true. If more than one automation defines the same condition, the first one that evaluates to true is picked up and processed.

Note

When you define only one automation plug-in external event receiver for each automation task, you are not required to enter a selector in the **Message Property Selector** field. In this case, automation tasks can share the same JMS queue without a message property selector being set. You must set a message property selector when you do either of the following:

- Define multiple automation plug-in external event receivers for the same automation task.
- Use the Legacy build-and-deploy mode to build and deploy cartridges with automation plug-ins.
- Use the Both (Allow server preference to decide) build-and-deploy mode to build and deploy cartridges with automation plug-ins and configure the OSM server dispatch mode for the Internal mode.

For information on build-and-deploy modes, see "About Automation Message Correlation" in *Modeling OSM Processes*.

Example: Modeling a Basic Automator Plug-in for an Automated Task

This example demonstrates how to configure an Automator type plug-in that receives data from an internal OSM JMS queue and updates order data using an XSLT style sheet. In the example, assume that the XSLT style sheet includes conditional logic to apply a level 1 priority to the order if the order is from a specific customer.

This example demonstrates how to:

1. Create an automated task and add the relevant task data.
2. Add an automation plug-in to the automated task.
3. Configure the automation plug-in properties.

Note

An automated plug-in exists within the context of a Design Studio cartridge project, order, process, and automated task. For purposes of demonstration, this example assumes the existence of multiple Design Studio entities. For example, it assumes the existence of a cartridge project called `DSLCartridge`, an order called `DSLOrder`, a process called `DSLProcess`, and an XSLT style sheet called `check_customer.xslt` that populates default values in the order data. It assumes that the Data Dictionary includes the two data nodes, `customer_name` and `order_priority`. It also assumes that the new automated task will be added to the `DSLProcess` entity. The naming conventions used in this example are for illustrative purposes only.

Step 1: Creating the automated task

1. Select **Studio**, then **New**, then **Order and Service Management**, then **Order Management**, and then **Automated Task**.

The Automated Task wizard appears.

2. In the Automated Task wizard, enter or select the following values:

- In the **Project** field, enter **DSLCartridge**.
- In the **Order** list, select **DSLOrder**.
- In the **Name** field, enter **Check_Customer**.

3. Click **Finish**.

The new automated task appears in the Automated Task editor.

4. Click the **Task Data** tab.

In this example, you will update the `order_priority` field with a default value of 1 if the order is from a specific customer.

Note

Normally, the task data includes all of the data that the task requires to complete. To simplify the example, this task includes only the two pertinent fields: `customer_name` and `order_priority`. See "[Modeling Data for Tasks](#)" for more information.

5. Right-click in the Task Data area.

The context menu appears.

6. Select **Select from Data Schema**.

The Select Data Elements dialog box appears.

7. Select the data nodes **customer_name** and **order_priority**.

8. Click **OK**.

The two data nodes appear in the Task Data area.

9. Click the **Permissions** tab.

On the **Permissions** tab, you can ensure that only the automation role has permissions for automated tasks. See the note in "[Modeling Roles and Setting Permissions](#)" for more information.

You are now ready to add a plug-in to the automated task.

Step 2: Adding the automation plug-in to the automated task

1. In the Automated Task editor, click the **Automation** tab.

2. Click **Add**.

The Add Automation dialog box opens.

3. In the **Name** field, enter **Check_Customer**.

4. In the **Automation Type** field, select **XSLT Automator**.

5. Click **OK**.

The `Check_Customer` plug-in appears in the **Automation** list.

6. In the Automation list, select the **Check_Customer** plug-in.
7. Click **Properties**.

The Automation Plug-in Properties tabs appear.

You are now ready to define the automation plug-in properties.

Step 3: Defining automation plug-in properties

1. In the **Automated Task editor Properties View Details** tab, accept the default value in the **EJB Name** field.
2. Ensure that the model variable that defaults to the **Run As** field points to a user name set up in the Oracle WebLogic console. When you deploy the cartridge, the user in the **Run As** field is added automatically to the OSM_automation group.

For more information about users and groups, see the discussion of setting up security in *OSM System Administrator's Guide*. For more information about model variables, see the Design Studio Help.

3. Click the **XSLT** tab.

On the **XSLT** tab, you define where the XSLT style sheet is located and the status to set if the automation fails. In this example, you'll define a location on your local machine where the XSLT file is stored.

4. Select **Absolute Path**.
5. In the **XSLT** field, enter the location of the XSLT file.

For this example, enter

**C:\oracle\user_projects\domains\osmdomain\xslt\DSLCartridge1.0.0\check_custom
er.xslt.**

6. Do one of the following:
 - In the **Exit Status on Exception** field, select **Failure**.

This field represents the exit status that the plug-in should use if it throws an exception. The options available in this field include any status values you assigned to the task. You use this option if you want to transition the task to a fallout task.
 - Click the **Details** tab and select the **Fail Task on Automation Exception** check box.

This check-box transitions the task to a fallout execution mode if an exception occurs when running the automation plug-in. Using the option allows you troubleshoot task failures within the task that generated the failure.

7. Select **Update Order**.

This option ensures that the default values obtained from the XSLT style sheet will be saved to the order data.

8. Click **Save**.

You have completed the basic configuration for an Automator-type plug-in defined as an internal event receiver.

Note

Successful automation requires a complete automation build file in the cartridge. If no automation build file exists, **Quick Fix** will generate one.

8

Modeling OSM Data

This chapter describes how to model OSM data in an Oracle Communications Order and Service Management (OSM) solution.

Data Modeling Overview

The entity that provides a unified view of all order data relating to various order activities is the order specification order template. All other entities relating to order processing contain a subset of the order data you define in the order template.

You can either model data directly in the order template or you can model data in various OSM entities in Oracle Communications Service Catalog and Design - Design Studio. When you add data to these other entities, Design Studio automatically adds the data into the order specification order template.

In general, there are four groups of data that you must model in any OSM solution. These general groups of data are:

- **Incoming order data:** You must understand the data structure and contents and decide what pieces are important to supporting the orchestration process. While there can be a large amount of data, orchestration is only concerned with modeling and extracting out information needed to support decomposition and dependency processing. For example, the orchestration functionality is primarily driven by the elements and structures within the **ControlData** structure.
- **External fulfillment system data:** You must determine what data you need to model in OSM for tasks that communicate with external systems or communicate order or task notifications to northbound systems or external users.
- **OSM web client user data:** You must determine what data you want users to access when using the OSM Order Management web client to manage orders or when using the OSM Task web client when managing orders or processing tasks.

In addition to identifying and modeling these order data groups, you must also understand how the data flows from each point during order processing. In addition, you must understand whether the data you receive from system A must be transformed or modified before sending it to system B. For example, system A that sends an order with a requested delivery date and time for broadband server may use a different date and time format than system B.

Common areas where data transformation occurs are:

- **Order recognition rule order data rules:** You must use an XQuery to map order data to the data specified in the creation task of the order. The data defined on the order may be identical to what is on the creation task, and so the XQuery must map the data into the corresponding parameters, or the data on the order may be different requiring you to manipulate the data so that it conforms with the data you have define in the creation task. See "[Modeling the Order Data Rule to Populate the Creation Task](#)" for more information.
- **Within Orchestration using the order transformation manager (OTM):** OTM provides OSM the ability to transform order items within the orchestration plan. For more information, see "[Modeling the Order Transformation Manager](#)".

- Within orchestration when OSM identifies order items from order data and maps the data to order item properties. For more information, see "[Modeling Orchestration Plans](#)".
- Between tasks: Automated tasks are the primary means that OSM employs for communicating with external systems. In some cases, the data required by the external system that Task A communicates with may require different parameters or formats than those generated by the creation task and those of other tasks communicating with other systems.

The task of dealing with different message types and formats can be simplified if you use an integration application such as Oracle Application Integration Architecture (Oracle AIA) which defines a canonical order structure for communication between OSM and external fulfillment systems. However, OSM can also directly integrate with external fulfillment systems and transform data immediately at the task within the automated task automation plug-in code.

You also use the data you create in OSM for a variety of other purposes. For example:

- You can model OSM to add the input message (the entire order) to the order. The order recognition rule that receives the message adds the message to an element designated as XML Type which contains the entire order data. See "[Adding the Input Message to the Order Template](#)" for more information.
- You can use data in the order template to manage orders; for example, you can create order keys used by amendment processing. See "[About Order Keys](#)" for more information.
- You can specify which data in the order template should be considered for amendment processing (data significance). See "[About Data Significance](#)" for more information.
- You can assign behaviors to data in the order template. See "[Modeling Behaviors Overview](#)" for more information.

Modeling Order Data

Consider the following data modeling approaches:

- **Data-centric:** First model data for a cartridge project and then model the cartridge project entities using specific data, as needed.
- **Entity-centric:** First model business processes and entities, and then model the data specifically required by the entities used by the business process.

About the Data Dictionary

Before OSM can receive an order from an order-source system, you need to create the OSM Data Dictionary.

The Data Dictionary is the repository of data elements used in Design Studio. The Data Dictionary defines data types and structures that can be used within OSM orders. For example, you can define a simple type that represents an IP address or a phone number, or more complex types representing addresses, product attributes and so on.

Data elements in a Data Dictionary are used as building blocks of an OSM order. The data elements within a Data Dictionary project can be referenced by other projects in a work space.

Design Studio automatically creates a Data Dictionary when you create an OSM cartridge project. You can use this default Data Dictionary or create multiple data schemas to add data elements or structure within the same project.

Each data schema includes a set of data relevant to the how that data is used. For example, a data schema for mobile services could include mobile-related data such as IMSI and MSISDN.

About the Order Template

When you create a new order model in Oracle Communications Service Catalog and Design - Design Studio, you can base the order on an existing order. When you extend an order specification, the extended specification inherits all of the data, tasks, rules, and behaviors of the base specification. You can add new data and behaviors to define unique order specifications and functionality. When you modify a base order specification, the order specifications extended from it are also modified. This means that you can make changes in one place, in the base specification, and those changes apply to the orders that are extended from the base specification.

For example, you might have three order specifications that share a common set of data. You can create a base order that includes configurations common to all three orders. You can then add configurations to each of the three order specifications for the data that is unique to each order specification.

When defining an order specification that is inherited from a base order specification, you cannot edit the inherited order data. For example, you cannot remove or rename data elements inherited from the base order specification. To implement changes to the inherited data, you must edit the data in the base order specification. Design Studio automatically implements those changes among all of the extended order specifications.

The data elements that you can use in an order are defined in the Design Studio **Data Dictionary**. When you define order data, you can use data elements that already exist in the Data Dictionary data schemas, or you can create new data elements and add them to the Data Dictionary. See ["About the Data Dictionary"](#) for more information.

In the data dictionary, you can model the same data element in one or more locations, and assign different type definitions for the elements, such as string or integer, and so on. For example, you might have a data dictionary that contains two instances of a data element called **EmployeeID**: one defined as a string (defined by the employee's name and a two-digit number), the other defined as an integer (defined by a 6-digit number). Although you can do this in the data dictionary, you cannot have the same data instance with different type definitions in the order template.

To avoid such data element conflicts, you can rename the first instance of the parameter after you import it into an order template using the refactoring function which allows you to rename an imported parameter at the order template level without changing the data dictionary instance from which it is derived. This creates an alias for the imported data element and you can then import the second instance of the data element without any data conflict errors. See Design Studio Modeling OSM Processes Help for more information about renaming data elements in the order template.

Identifying Data Requirements for Order Payload

The incoming order data contains important information about the hierarchy of sales item lines, which can consist of offers, bundles, products, and so on. This data structure information can be used to manage the data when it is passed between different fulfillment systems.

You must model incoming order data in a Design Studio data dictionary. You can either manually build the data dictionary for the incoming order data or you can import an XSD file defined in some other application into Design Studio.

To import the Data Dictionary for the data received in orders, you import the XSD file for that incoming customer order into OSM. The elements in the XSD file are loaded into the Data

Dictionary as OSM data elements. [Example 8-1](#) shows part of an XSD file that could be used for importing customer data.

Example 8-1 Elements in Input Message XSD File

```
<element name="order" type="im:OrderType"/>
<element maxOccurs="1" minOccurs="1" name="numSalesOrder" type="string">
</element>
<element maxOccurs="1" minOccurs="1" name="typeOrder">
</element>
```

For each data element, you specify attributes about the data element; for example, the data type and display name. [Figure 8-1](#) shows the configuration for a **requestedDeliveryDate** data element.

Figure 8-1 Data Element Defined in Design Studio

The screenshot shows the 'Element /SalesOrderLine/requestedDeliveryDate' configuration window. The 'Details' tab is active, displaying the following fields:

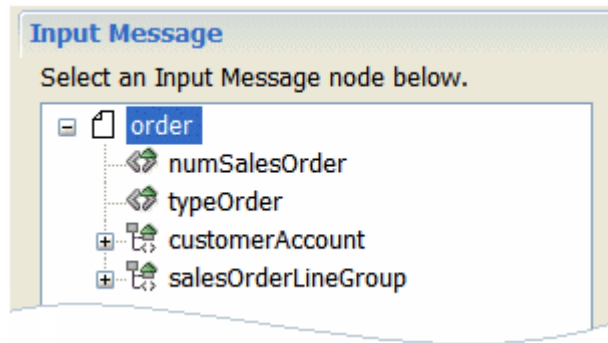
- Name:** requestedDeliveryDate
- Display Name:** requestedDeliveryDate [default] ▼
- Type:** dateTime ▼ [Select ...] [Open ...]
- Max Length:** (empty field)
- Minimum:** 1 ▼ **Maximum:** 1 ▼
- Path:** /SalesOrderLine/requestedDeliveryDate
- Namespace:** http://xmlns.oracle.com/InputMessage

Child XML elements are imported as child data elements. The **Path** field shows the parent data elements. In this example, the parent data element of **requestedDeliveryDate** is **SalesOrderLine**.

Adding the Input Message to an Order Recognition Rule

You must add the order data structure of an incoming order to the Input Message area on the **Details** tab in an order recognition rule.

[Figure 8-2](#) shows an input message specified in a recognition rule.

Figure 8-2 Input Message Specified in a Recognition Rule

The order recognition rule **Order Data Rule** XQuery transforms order data into the OSM order format. However, you can also add the input order data to an order by adding the order data to the order template. For more information, see "[Adding the Input Message to the Order Template](#)".

Adding the Input Message to the Order Template

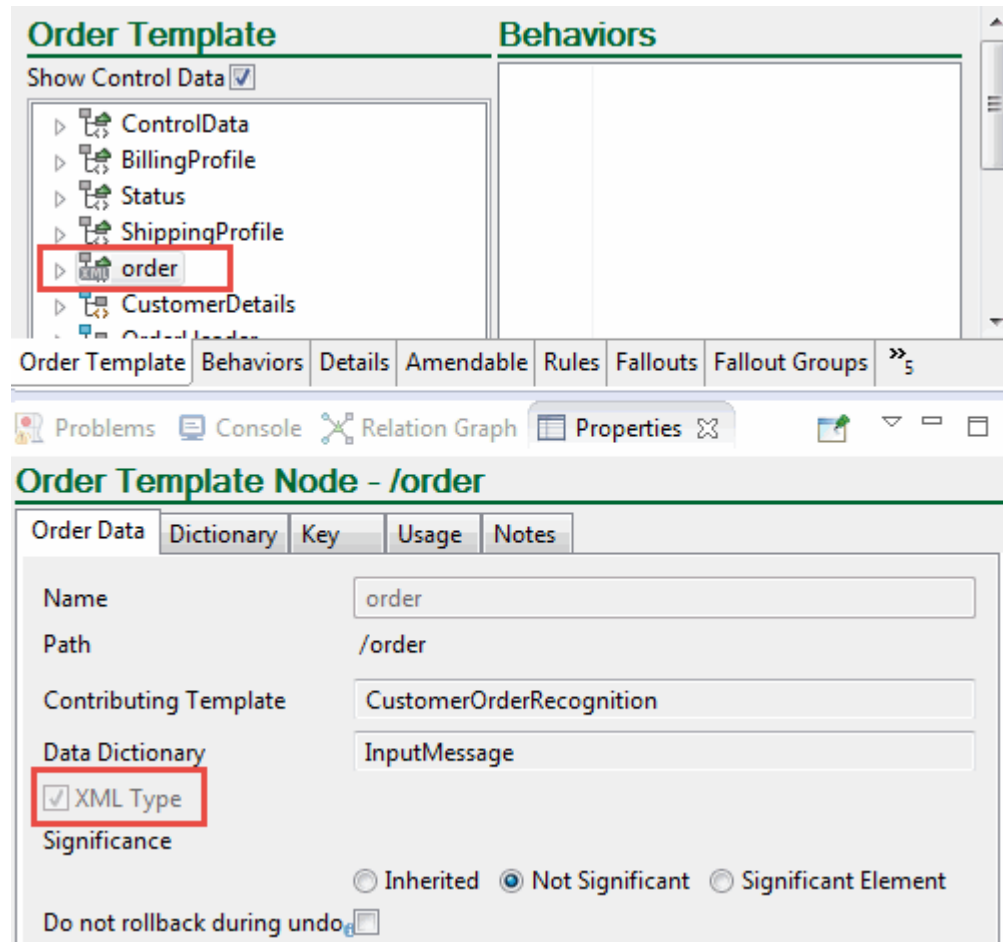
You can model an order template with the incoming order data so that OSM automatically add the incoming order data to the OSM order in addition to the data generated by an order recognition rule **Order Data Rule** Xquery. You should not use the incoming order data for order processing, but the order data information can be useful for debugging, order tracking, or reference purposes.

To add the incoming order data to an order you must add the incoming order data to the target order specification Order Template tab Order Template area.

You must designate the root incoming order data element as an XML Type so OSM can store the data more efficiently. In addition, you must also add the incoming order data structure to the creation task so that OSM can add the incoming order data to the OSM order.

[Figure 8-3](#) shows the input order element **order** with **XML Type** selected in the **Properties** tab **Order Data** sub-tab.

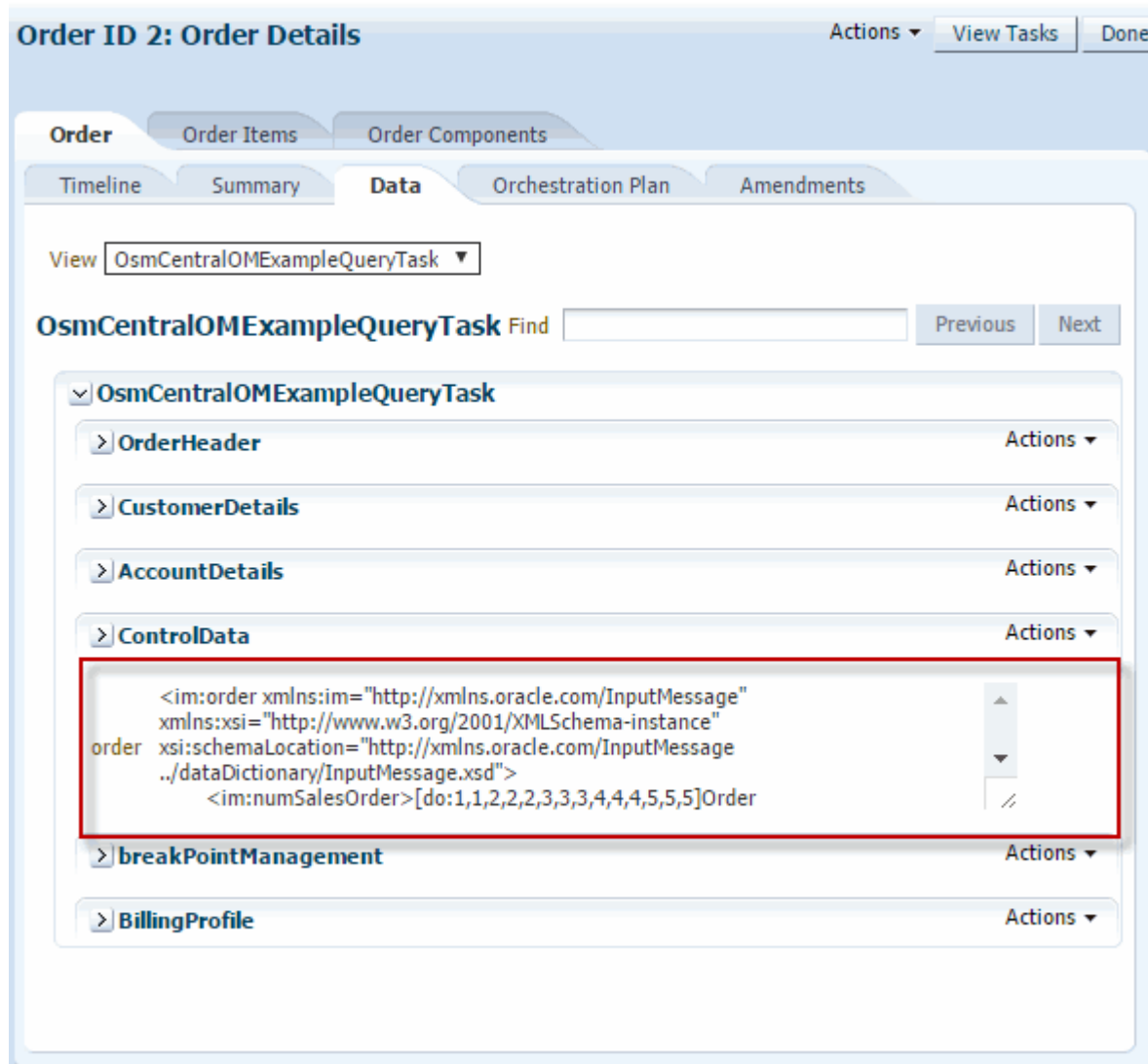
Figure 8-3 Input Order Data as XML Type



For debugging, order tracking, or reference purposes, you can add the incoming order data to a query task so that operators can view data from the Task web client or the Order Management web client.

[Figure 8-4](#) shows the input order data in the Order Management web client **Data** tab with the `OsmCentralOMExampleQueryTask` selected in the **View** field.

Figure 8-4 Order Management Web Client Input Order Data XML Structure



Modeling Valid Data Keys

Data keys are elements that identify specific instances of multi-instance nodes. For more information about the use of data keys, see "[About Order Data Position and Order Data Keys](#)."

When modeling a data key, you must follow the following guidelines.

- Data key expressions must always return a value. For example, if you could use the following expression to return a key value:

```
../login/username/text()
```

However, if either login or username are optional parameters, then this expression might not return a value. Ensure that you model your data and write your key expressions so that data in a valid order will always cause the key expression to return a value.

- Ensure that parallel key expressions always evaluate to unique values. That is, all of the instances of one multi-instance data element must have unique keys.

If a multi-instance data element is inside another multi-instance data element, the child elements' keys must be unique within each parent element. For example, in the following data structure, you could use **.host/text()** as the expression to generate the key for the **location** element. This would work because it is unique within each of the **email-service** parent elements, even though it is not unique across the whole order.

```
<email-service>
  <address>john.doe@example.com</address>
  <name>John Doe</name>
  <quota>5Gb</quota>
  <location>
    <host>host_a</host>
    <operating-system>Linux</operating-system>
  </location>
  <location>
    <host>host_b</host>
    <operating-system>Solaris</operating-system>
  </location>
</email-service>
<email-service>
  <address>jane.doe@example.com</address>
  <name>Jane Doe</name>
  <quota>2Gb</quota>
  <location>
    <host>host_a</host>
    <operating-system>Linux</operating-system>
  </location>
  <location>
    <host>host_c</host>
    <operating-system>MacOS</operating-system>
  </location>
</email-service>
```

- Data key expressions must not use children of reference data elements.

Design Studio allows order template elements to be references to elements elsewhere in the order. For example, this allows more than one order component to refer to the same order item. In this way, the different order components can see the latest version of the order item data, while allowing the data to exist (and be updated) in only one place. Data key expressions that refer to descendants of references are not valid.

- When using descendant data elements in your key expression, consider restricting data key expressions to refer only to direct child elements.

While it is valid for data key expressions to refer to descendants beyond direct child elements, it is easier to ensure compliance with the other criteria when only direct child elements are used.

Modeling Data for Tasks

The following sections describe modeling data requirements for tasks.

Determine Task Data for Manual and Automated Tasks

Each task includes a set of data, which you specify when modeling the task.

The data included in a task is data relevant to the function of the task. [Table 8-1](#) shows some example tasks and the task data they include.

Table 8-1 Examples of Tasks and Task Data

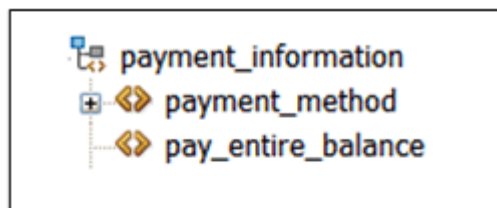
Task	Task Data
Add capacity	Bandwidth
Send customer survey	Name, phone number, address
Query task (to display data in the Task web client)	Name, phone number, bandwidth, port ID

When you model a task, you assign it to an order. The available task data is limited to the data that the order requires. At run time, task data can be entered by an OSM user, provided on an incoming order, or provided from a previous task in the order.

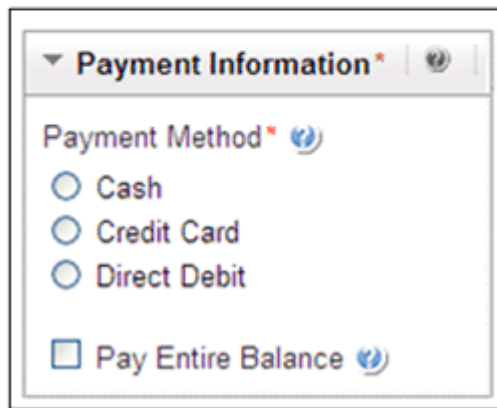
[Figure 8-5](#) shows task data defined in a task in Design Studio and how the data is displayed in an order in the Task web client.

Figure 8-5 Task Data in a Task Specification and in an Order

Task data defined in Design Studio:



Task data displayed in an order in the Task Web client:



✓ **Tip**

To improve performance, usability, and security, include only the data that is necessary to perform the task. Unnecessary data is not exposed to the user performing the task, even though the order may contain much more data.

When modeling orders, it is common to include the entire XML representation of the order in the order data as an XML data type. If you include the XML data, consider defining smaller

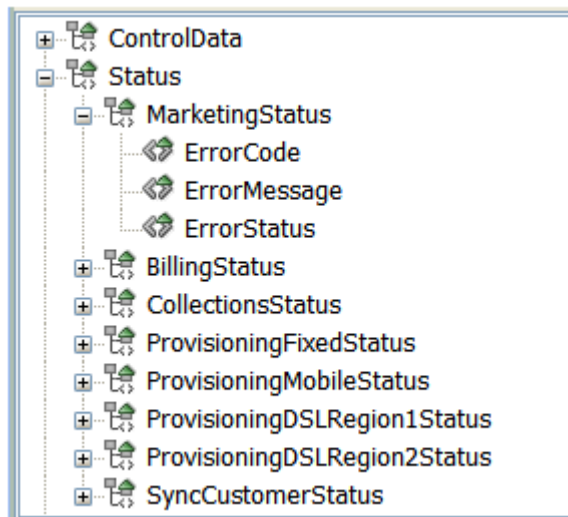
XML elements for storing sections of a sales order rather than including a single XML data type that contains the entire sales order. This allows you to map only the parts of the order that are needed for each task. Including the XML representation is typically done only in the modeling process as an aid to development.

In addition to defining the data included in each task, you can use **behaviors** in manual tasks to manipulate many aspects of how the data is displayed, formatted, and validated. For example, you can specify if data is read-only, or you can modify the value of the data in a task. See "[Modeling Behaviors Overview](#)" for more information.

Determine Task Data for Data Returned from Fulfillment Applications

You can configure the order template to hold status data returned from external systems. [Figure 8-6](#) shows an order template structure that holds status data.

Figure 8-6 Status Data in the Order Template



Generating Multiple Task Instances from a Multi-Instance Field

Some tasks require multiple task instances to complete. For example, you might need to create three task instances to retrieve three different address fields. To accomplish this, you designate a field as a **pivot data element** for the task. When OSM runs the task at run time, the system generates a separate task instance for each separate instance of the pivot data element in the order. The system creates as many instances of the task as there are instances of the data field or data structure, up to the maximum number defined for the field. This feature works for a structure of data also. For example, if the address is a structure called Address, with nested elements of Street, City, and Postal Code, the system generates an instance of a task for each instance of the structure. The data that is visible to the task instance will be restricted to data structure that it is for, and that task will not have visibility to the other instances of the data.

Note

OSM compensation processing does not support task pivot data elements.

Pivot subprocesses are normally "spawned" all at once, meaning that all subprocess instances are available for execution at the same time. Compensation for this type of scenario works as expected.

Pivot subprocesses can also be spawned sequentially, meaning that subprocess instances are only available for execution one-at-a-time, each new instance only becoming available after the previous one completes. The solution defines the order via "Sequential" and "Sort Element" attributes of the pivot node in Design Studio. OSM does not support compensation involving pivot subprocesses that are configured to execute sequentially. If your solution involves compensation of pivot node subprocesses, you must not use sequential execution.

Modeling Data for Orchestration

Define the orchestration data on the entity that best reflects its structure, rather than defining all of the data on the order specification. Design Studio generates the order level order template by aggregating the order template definitions for the order item specifications and order components with any data defined at the order level.

You should define data at the level where it is needed:

- Order Item specification: Define **ControlData/OrderItem** and all of the order item properties on the order item specification.

The **OracleComms_OSM_CommonDataDictionary** model project contains predefined base data elements for control data. It is recommended that you use the data schema of this model project to add the **ControlData/OrderItem** base data element to the order item specification **Order Template** tab.

- Order component: Define **ControlData/Functions/OrderComponentName** and any other data needed by the tasks in the process that run this component in the appropriate order component template.

If you use the **OracleComms_OSM_CommonDataDictionary** model project (recommended) and your orchestration entities are preconfigured correctly, Design Studio automatically generates this structure on the order template of the order component and the order template of the order.

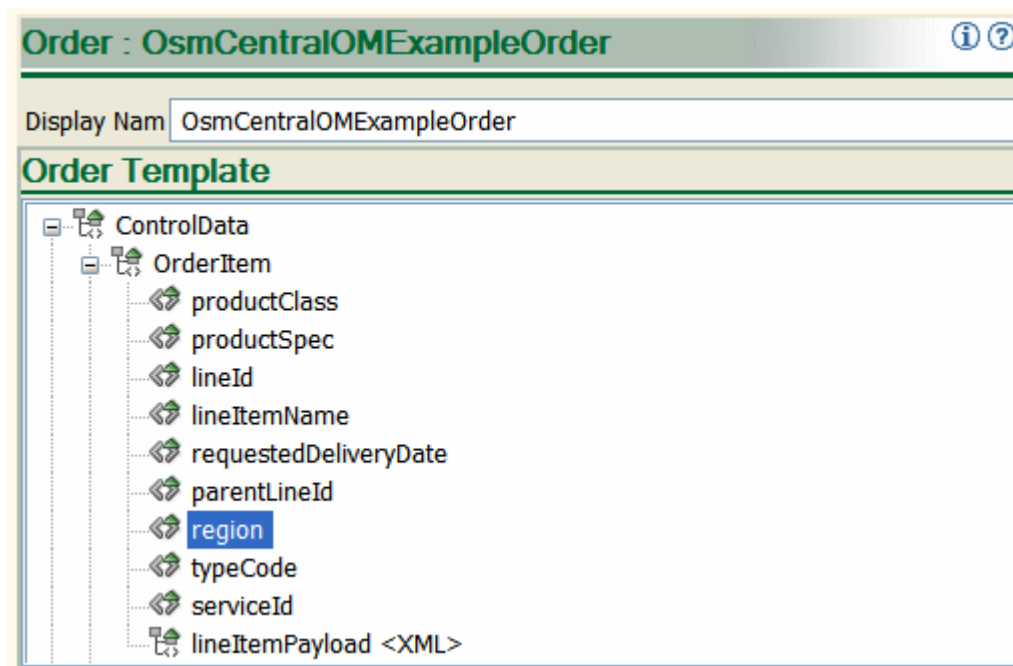
Using this method supports:

- Encapsulation
- Re-factoring: Modify order template data at the entity level to which it is associated because this highlights the connection between an entity and its order template data.
- Maintenance: Modifications to order item specification and order component templates help the designer understand the impact of changes, including possible breaks in compatibility.
- Traceability: Using this method provides direct traceability from order template data to the modeling entity to which it is attached.

About Order Item Control Data

In addition to defining order item properties in the order item specification, you need to provide a storage area for the order item properties. You do so by adding control data to the order item specification **Order Template** tab. This definition is automatically added to the order's order template. This makes it easier to track which entity is the master of the data and enables easier refactoring and maintenance of the overall order specification. [Figure 8-7](#) shows the order item properties in the control data in an order template.

Figure 8-7 Order Item Properties Included in the Order Template



When you define the control data, note the following:

- The name used in the control data must exactly match the spelling and case of the order item property name.
- Make sure that the Data Dictionary properties are correct for the type of data; for example, string or number.
- Configure each data element as a multi-instance data element.
 - Minimum = 0
 - Maximum = Unbounded

Note

To define data properties, you edit the entry in the data schema, not in the order item specification.

An instance of **ControlData/OrderItem** is created for each data element returned by the order item selector from the orchestration sequence (see "[About Creating Order Items from Customer Order Line Item Node-Sets](#)").

The **OracleComms_OSM_CommonDataDictionary** model project contains predefined base data elements for control data. Oracle recommends that you use the data schema of this model project to add the **ControlData/OrderItem** structure to the order item specification **Order Template** tab.

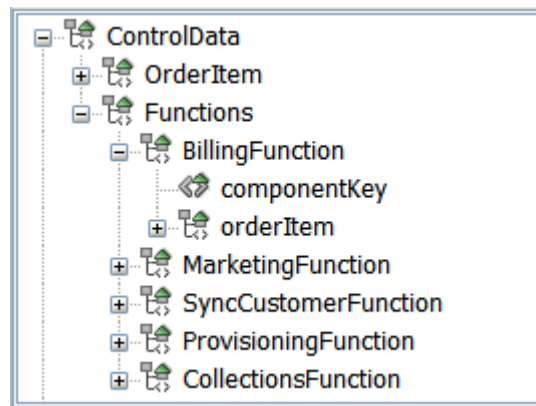
About Order Template Data

The order template includes control data. **Control data** is used by OSM to generate the orchestration plan. Control data is used **only** for orchestration.

There are typically two areas of the order control data:

- ControlData/OrderItem provides the data and structure of order items received in the incoming customer order.
- ControlData/Functions stores the structure of the function order components generated by the first level of decomposition. [Figure 8-8](#) shows function components represented in the order template. The types of functions (BillingFunction, MarketingFunction, and so on) represent the function-level order components.

Figure 8-8 Functions Data in the Order Template



You manually model the order control data of order items in Design Studio. Control data for function order components is automatically generated by Design Studio. See "About Modeling Control Data" in *Modeling OSM Processes* for information on how control data is modeled and generated.

Orchestration plan generation requires a specific order template structure which you must model at design time.

```

ControlData
  OrderItem
  Functions
    OrderComponentName
    componentKey
    calculatedStartDate
    duration
    OrderItem
      orderItemRef
  
```

About Order Item Specification Data

This is a multi-instance node that OSM populates with a set of order items generated off the inbound message. The children of this structure must exactly match the set of order item properties defined on the Order Item specification editor in Design Studio.

The **OracleComms_OSM_CommonDataDictionary** model project contains predefined base data elements for control data. It is recommended that you use the data schema of this model project to add the **ControlData/OrderItem** data element to the order item specification **Order Template** tab.

See "About Modeling Control Data" in *Modeling OSM Processes* for instructions on modeling the **ControlData/OrderItem** structure.

About ControlData for Order Component Data

Order component information is stored in the template in **ControlData/Functions/OrderComponentName**. This is a multi-instance node that OSM populates with the set of order components generated by executing the decomposition rules through an orchestration sequence. *OrderComponentName* must be defined for each order component included in a fulfillment pattern's orchestration plan. This section of the **ControlData** represents all of the order components in the orchestration plan. If you use the **OracleComms_OSM_CommonDataDictionary** model project, Design Studio automatically generates data (*OrderComponentName*) and adds it to the **ControlData/Functions** structure for each order component that is associated with the fulfillment pattern that is part of the orchestration plan.

Each order component is assigned a unique key, called the order component ID, which is stored in the **componentKey** element. For information about how the component ID is determined, see "[About Component Names and Component IDs](#)."

OSM populates the **calculatedStartDate** (dateTime type) and **duration** (string type) nodes for each **ControlData/Function**. With **calculatedStartDate** and **duration** per **Function**, both central order management and service order management solutions can use these values as the requested delivery date for the order line in a downstream system. based on the modeling done in the Order Component Specification entity, the date does affect the runtime behavior of the order component. If there is a **Duration Value** associated with a dependency, it is used in the order component start date calculation since this value is relative value to the orchestration dependency.

OSM populates the multi-instance **orderItem** node with the set of order items that have been decomposed into this order component. The order items are accessed through **orderItemRef**, which is a reference node to **ControlData/OrderItem**. A reference node is used to point to the actual storage location of the order item so that updates to the order item data are reflected in all order components the order item is referenced from.

You can also store status data in the order item data and in the function data. [Figure 8-9](#) shows a structure for storing status data. In this example:

- The **LineID** data element provides a reference to the order line item in the incoming customer order.
- The **SystemInteraction** data element stores data about status events; for example, a status code, description, and timestamp.

Figure 8-9 Status Data in Order Item

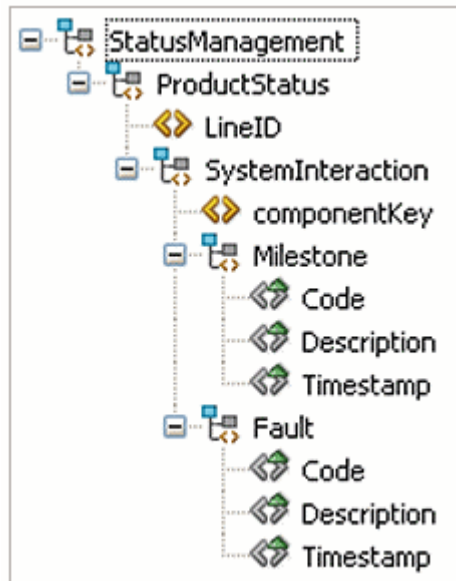
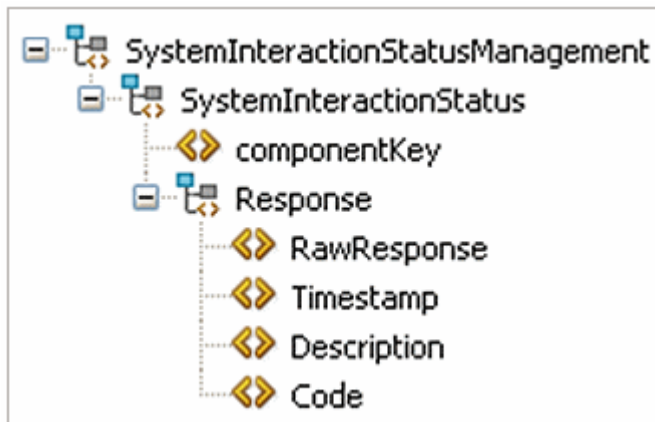


Figure 8-10 shows a structure for storing status data for functions. In this example:

- The **componentKey** data element provides a reference to the order component instance.
- The **Response** data element stores the message from the external system, as well as the timestamp, description, and status code.

Figure 8-10 Status Data in Functions



Modeling Data for Fulfillment States

Fulfillment state processing requires specific structures and data elements inside the order template. The specific locations of the data can be changed using XML catalog; the default locations are presented here. See ["About XML Catalogs"](#) for more information about using XML catalogs in OSM. See ["Sample XQuery for Changing Default Data Locations"](#) for more information about changing the default data locations.

About ControlData for External Fulfillment States

External fulfillment state information is populated for order components.

The default location for external fulfillment state information is **ControlData/Functions/OrderComponentName/orderItem/ExternalFulfillmentState**.

Write the automation code so that it populates the information in the correct place. For example, the following automation code updates the ExternalFulfillmentState value whenever a response containing a fail value returns or passes on any other value in the response:

```
</UpdatedNodes>
{
  for $orderItem in $component/oms:orderItem
  let $lineId := $orderItem/oms:orderItemRef/centralom:lineId/text()
  return
  (
    if ($responseRoot/res:lineResponses/res:response[@id=$lineId]/text() = 'fail') then
    (
      <Update path="/oms:ControlData/oms:Functions/oms:BillingFunction/
oms:orderItem[@index='{ $orderItem/@index }']">
      <ExternalFulfillmentState>{$responseRoot/res:status/text()}</ExternalFulfillmentState>
      </Update>
    )
    else
    (
      <Update path="/oms:ControlData/oms:Functions/oms:BillingFunction/
oms:orderItem[@index='{ $orderItem/@index }']">
      <ExternalFulfillmentState>{$responseRoot/res:status/text()}</ExternalFulfillmentState>
      </Update>
    )
  )
}
</OrderDataUpdate>
```

About ControlData for Order Fulfillment State

OSM populates the order fulfillment state based on the configuration in the order fulfillment state composition rule set.

The default location for OSM to populate the order fulfillment state is **ControlData/OrderFulfillmentState**. The Data Dictionary contains a root-level **OrderFulfillmentState** element. For cartridges created in a pre-7.2 version of OSM, drag the root-level **OrderFulfillmentState** element into the **ControlData** node on the order. For new cartridges, the element will get added automatically to the order template as a child of **ControlData**.

About ControlData for Order Item Fulfillment State

OSM populates the order item fulfillment state based on the configuration in the order item fulfillment state composition rule set.

The default location for OSM to populate the order item fulfillment state is **ControlData/OrderItem/OrderItemFulfillmentState**. The Data Dictionary contains a root-level **OrderItemFulfillmentState** element. For order items in cartridges created in a pre-7.2 version of OSM, drag the root-level **OrderItemFulfillmentState** element into the **ControlData/OrderItem** node on the order. For new cartridges and order items, the element will get added automatically to the order template as a child of **ControlData/OrderItem**.

Fulfillment States and Point of No Return

If points of no return have been configured using fulfillment states, OSM populates the point of no return when processing the order item fulfillment state composition rules. For more information about points of no return, see *OSM Concepts*.

The default location for OSM to populate the point of no return value is **ControlData/OrderItem/PointOfNoReturn**.

Fulfillment State and Point of No Return Initial Values

You can set initial values for order item fulfillment states and points of no return, so that these values will appear on the order before any processing takes place. See "[Sample XQuery for Changing Default Data Locations](#)" for more information about setting these values.

Sample XQuery for Changing Default Data Locations

To change the default locations and set initial values for point of no return and order item fulfillment state, include an XQuery file in the XML catalog. To use the defaults, do not provide a file.

To include your custom XQuery file in the cartridge, include a line similar to the following in the XML catalog file for your cartridge:

```
<rewriteURI uriStartString="cp:oracle/communications/ordermanagement/execution"
rewritePrefix="osmmodel:///CartridgeName/CartridgeVersion/resources/Directory"/>
```

For more information about using XML catalogs, see "[About XML Catalogs](#)".

If you choose to configure a custom file, you should include all of the functions, even those for defaults you are not changing. This will clarify the configuration and assist in maintenance activities. The purpose of each function is indicated in comments in the file. For all values that specify order template locations (for example /OrderLifeCycleManagement), begin the value with a forward slash, as shown below.

```
xquery version "1.0";
module namespace fulfillmentstatemodule = "http://xmlns.oracle.com/communications/
ordermanagement/fulfillmentstatemodule";

declare namespace saxon="http://saxon.sf.net/";
declare namespace xsl="http://www.w3.org/1999/XSL/Transform";
declare namespace oms = "urn:com:metasolv:oms:xmlapi:1";

(: Returns the composite fulfillment state path for an order. :)
declare function fulfillmentstatemodule:getOrderCompositeFulfillmentStatePath
($orderMnemonic as xs:string) as xs:string {
  "/ControlData/OrderFulfillmentState" };

(: Returns the composite fulfillment state path for an order item. :)
declare function fulfillmentstatemodule:getOrderItemCompositeFulfillmentStatePath
($orderMnemonic as xs:string) as xs:string {
  "/ControlData/OrderItem/OrderItemFulfillmentState" };

(: Returns the default order item external fulfillment state path. :)
declare function fulfillmentstatemodule:getOrderItemExternalFulfillmentStatePath
($orderMnemonic as xs:string) as xs:string {
  "ExternalFulfillmentState" };
```

```

(: Returns the default type of the order item external fulfillment state path.
Valid values are RELATIVE_PATH and ABSOLUTE_PATH. :)
declare function fulfillmentstatemodule:getOrderItemExternalFulfillmentStatePathType
($OrderMnemonic as xs:string) as xs:string {
    "RELATIVE_PATH" };

(: Returns the point of no return path for an order item. :)
declare function fulfillmentstatemodule:getOrderItemPoNRPath ($OrderMnemonic as
xs:string) as xs:string {
    "/ControlData/OrderItem/PointOfNoReturn" };

(: Returns the name of the initial fulfillment state. :)
declare function fulfillmentstatemodule:getOrderInitialFulfillmentStateName
($OrderMnemonic as xs:string) as xs:string {
    "" };

(: Returns the namespace of the initial fulfillment state. :)
declare function fulfillmentstatemodule:getOrderInitialFulfillmentStateNamespace
($OrderMnemonic as xs:string) as xs:string {
    "" };

(: Returns the initial point of no return value of an fulfillment state. :)
declare function fulfillmentstatemodule:getOrderItemInitialPoNR($OrderMnemonic as
xs:string) as xs:string {
    "" };

declare function fulfillmentstatemodule:getExternalFulfillmentStates(
    $OrderData as element()) as element()?
{
    let $OrderMnemonic :=
        if (fn:exists($OrderData/OrderType))
        then $OrderData/OrderType/text()
        else ""
    let $OrderItems := $OrderData/_root/ControlData/OrderItem
    where (fn:exists($OrderItems))
    return
        <oms:ExternalFulfillmentStates>
        {
            for $OrderItem in $OrderItems
            let $OrderItemIndex := $OrderItem/@index
            let $components := $OrderData/_root/ControlData/Functions/*[orderItem/
orderItemRef/@referencedIndex=$OrderItemIndex]
            let $externalFulfillmentStatePath :=
fulfillmentstatemodule:getOrderItemExternalFulfillmentStatePath($OrderMnemonic)
            let $externalFulfillmentStatePathExistsCheck :=
fn:concat($externalFulfillmentStatePath, "[text()!='']")
            let $externalFulfillmentStateExists := fn:exists($components/
orderItem[orderItemRef/@referencedIndex=$OrderItemIndex]/
saxon:evaluate($externalFulfillmentStatePathExistsCheck))
            where (fn:exists($components) and $externalFulfillmentStateExists=fn:true())
            return
                <oms:OrderItemExternalFulfillmentState index="{ $OrderItemIndex }">
                {
                    for $component in $components
                    let $componentKey := fn:normalize-space($component/componentKey/
text())
                    let $componentId := $component/@index
                    let $externalFulfillmentStateValuePath :=
fn:concat($externalFulfillmentStatePath, "[last()]/text()")
                    let $externalFulfillmentState := fn:normalize-space($component/
orderItem[orderItemRef/@referencedIndex=$OrderItemIndex]/
saxon:evaluate($externalFulfillmentStateValuePath))

```

```

        where (fn:exists($externalFulfillmentState)
and $externalFulfillmentState != "")
        return
            <oms:OrderItemComponentState componentId="{ $componentId}">
                <oms:ComponentKey>{ $componentKey}</oms:ComponentKey>
                <oms:ExternalFulfillmentState>{ $externalFulfillmentState}</
oms:ExternalFulfillmentState>
            </oms:OrderItemComponentState>
        }
    </oms:OrderItemExternalFulfillmentState>
}
</oms:ExternalFulfillmentStates>
};

```

Modeling Data for Processing States

Processing states requires specific structures and data elements inside the order template.

About ControlData for Order Component Order Item Processing States

Order component order item processing state information is populated for order components.

The default location for order component order item fulfillment state information is **ControlData/Functions/OrderComponentName/orderItem/FunctionProcessingState**.

Write the automation code so that it populates the information in the correct place. For example, the following automation code updates the FunctionProcessingState to the UndoFailed value whenever a response containing a fail value returns or to the Completed value whenever any other response returns:

```

</UpdatedNodes>
{
  (
    for $orderItem in $component/oms:orderItem
    let $lineId := $orderItem/oms:orderItemRef/centralom:lineId/text()
    return
      (
        if ($responseRoot/res:lineResponses/res:response[@id=$lineId]/text() = 'fail') then
          (
            <Update path="/oms:ControlData/oms:Functions/oms:BillingFunction/
oms:orderItem[@index='{ $orderItem/@index}']">
              <oms:FunctionProcessingState>UndoFailed</oms:FunctionProcessingState>
            </Update>
          )
        else
          (
            <Update path="/oms:ControlData/oms:Functions/oms:BillingFunction/
oms:orderItem[@index='{ $orderItem/@index}']">
              <oms:FunctionProcessingState>Completed</oms:FunctionProcessingState>
            </Update>
          )
        )
      )
  )
}
</OrderDataUpdate>

```

About ControlData for Order Item Processing States

OSM populates the order item processing state based on the order component order item processing state.

The default location for OSM to populate the order item processing state is **ControlData/OrderItem/OrderItemProcessingState**.

Modeling Orders With Data Fields Above 1000 Characters

Standard OSM Design Studio data elements and structures can support a maximum of 1000 characters. However, in some cases it may be necessary to model data that exceed this limit. Before you model order data fields that can contain more than 1000 characters, you must carefully decide whether these fields are necessary. Unnecessary data within an order can reduce the order processing performance of OSM.

The following sections describe ways to achieve data length for OSM data above 1000 characters.

Using XML Types for Data Fields Above 1000 Characters

In Design Studio, you can model data dictionary structures as XML types from the **Order** specification, **Order Template**, **Properties** sub-tab, **Order Data** sub-tab. The structure must be empty and contain no children elements or structures for it to be designated as XML type. Structures defined as XML types in the data dictionary can contain XML documents. You can also use XML schema files to validate the XML structures in the XML types.

Oracle recommends this option when the data is not human editable or readable in the OSM user interfaces because the data is represented as XML. For example, the XML data can be captured as follows, where **<largetext>** is the name of the structure designated as XML type:

```
<largetext>
Text to be inserted here
</largetext>
```

When you have defined the XML type structures in the Order specification Order Template, then included them as a part of Manual or Automated Task Data, you can access the XML data using:

- The OSM Task web client Order Editor screen (see *OSM Task Web Client User's Guide* for more information).
- XML API GetOrder and UpdateOrder transactions (see *OSM Developer's Guide* for more information).
- OSM Web Service GetOrder and UpdateOrder OSM operations (see *OSM Developer's Guide* for more information).
- Order access and updates performed using Automated Task automation plug-ins (see "[About Automation Plug-ins](#)" for more information).

This approach has the following limitations:

- You cannot specify XML type data as significant for amendment processing. Changes to this data does not trigger compensation.
- XML types are not visible in the OSM reporting interface.

To enable XML schema validation:

1. Create schema files for the required XML data type.
2. Use the Java perspective Package Explorer view to copy the schema files into the cartridge project data dictionary folder where the XML data type has been defined.

Using Order Remarks for Data Fields Above 1000 Characters

You can add Remarks that contain text to orders during order processing. Remarks can be retrieved and updated using:

- The OSM Task web client Remarks and Attachments screens (see *OSM Task Web Client User's Guide* for more information).

Note

The attachments created using the Remark and Attachments screens can be accessed through the OSM Database.

- XML API Getorder and UpdateOrder transactions (see *OSM Developer's Guide* for more information).
- OSM Web Service GetOrder and UpdateOrder operations (see *OSM Developer's Guide* for more information).
- Order access and updates performed using Automated Task automation plug-ins (see "[About Automation Plug-ins](#)" for more information).

This approach has the following limitations:

- Remarks can store up to 4000 bytes of data. Depending upon the character set configured in your database, the number of characters will vary.
- Remarks associated with orders are only editable for a certain time after you add them. This time limit is defined by the **remark_change_timeout_hours** parameter contained in the **oms-config.xml** file. You can edit the value associated with this parameter to change the number of hours that remarks are editable. The default value is 24 hours. See *OSM System Administrator's Guide* for more information about working with the **oms-config.xml** file.

Using Attachments for Data Fields Above 1000 Characters

You can also add file attachments to remarks. File attachments can contain large amounts of data and you can store them in different formats. You can access attachments with:

- The OSM Task web client using the Remarks and Attachments screens (see *OSM Task Web Client User's Guide* for more information).

Note

The attachments created using the Remark and Attachments screens can be accessed through the OSM Database.

- XML API Getorder and UpdateOrder transactions (see *OSM Developer's Guide* for more information).
- OSM Web Service GetOrder and UpdateOrder operations (see *OSM Developer's Guide* for more information).

Attachments are governed by the **max_attachment_size** parameter in the **oms-config.xml** file. You can edit the value associated with this parameter to change the maximum attachment

size. The default value is 10MB. See *OSM System Administrator's Guide* for more information about working with **oms-config.xml**.

Note

When the remark change threshold is exceeded (**remark_change_timeout_hours**), you can no longer add or delete attachments to the remark.

Using Data Providers to Retrieve Data

This section describes how to use data providers to retrieve data when modeling orders in OSM.

About Data Providers and Adapters

An Oracle Communications Service Catalog and Design - Design Studio data provider is an instantiation of adapter (which is a Java class) that can retrieve data in an XML format from external systems. Data Providers are used when defining Data Instance behaviors (see "[Using the Data Instance Behavior to Retrieve and Store Data](#)" for more information). Design Studio provides several built-in Data Providers to retrieve external XML instances from specific sources such as an Objectel server extension or a SOAP web service. Additionally, you can create your own custom Data Provider (see "[Custom Data Providers](#)" for more information).

In Design Studio, the Data Provider editor **Settings** tab ([Figure 8-11](#)) allows you to set the Data Provider type using **Provider Type**. Types of Data Providers include:

- [Objectel](#)
- [Order](#)
- [Property File](#)
- [SOAP](#)
- [XML Attachment](#)
- [XML File](#)
- [XML Validation](#)
- [JDBC](#)
- [Web Service](#)
- [Custom Data Providers](#)

When you select any of the above choices other than a custom data provider, the **Provider Class** field becomes disabled and is populated with the OSM implementation of the adapter. When you select **Custom**, the **Provider Class** field is enabled because you must supply the class name of the custom adapter that you write. See "[Custom Data Providers](#)" for detailed information.

Figure 8-11 Data Provider Settings Tab

The screenshot shows a window titled "Data Provider : Test" with a green header bar. Below the header, there is a "Display Name" field containing "Test1". A section titled "Settings" is highlighted with a green underline. Within this section, the "Provider Type" is set to "Objectel" in a dropdown menu, and the "Provider Class" is "(Builtin Objectel Adapter)" in a text field. Under the "Cache Settings:" heading, the "Scope" is "System" in a dropdown menu, "Maximum Time (Milliseconds)" is "15000" in a text field with up/down arrows, and "Maximum number cached" is "50" in a text field with up/down arrows. At the bottom, there are two tabs: "Settings" (which is active) and "Interface".

Data Provider Interface Tab

Data providers, both built-in and custom, can take parameters as input, as shown in the Interface tab ([Figure 8-12](#)). Parameter names are free-form text, but are dictated by the data provider's expected input. An asterisk (*) appears next to mandatory parameters, and each parameter's corresponding value can be specified as either XPath 1.0 or XQuery 1.0. In addition to the functions provided by the XPath 1.0 or XQuery 1.0 standards, OSM provides a custom function, `instance(string)` that allows the output of one data provider to be used as the input of another. The parameters required by each of the built-in data providers is documented in the sections that follow.

Figure 8-12 Data Provider Interface Tab

The screenshot shows the 'Data Provider : Test1' interface. At the top, the 'Display Name' is 'Test1'. Below this is the 'Input Parameters' section, which includes a 'Current Param' field set to 'objectel:extensionName'. A 'Parameters' list contains 'objectel:extensionName*', 'objectel:jmsFactory', 'objectel:queue', and 'objectel:allowErrorResponse'. The 'Default Value' is set to 'XPath' with a dropdown arrow, and the value field contains 'iLibPlus:runStoredQuery'. There are 'Remove' and 'Add' buttons for the parameters. Below this is the 'Result Documents' section, which is currently empty, with 'Remove' and 'Add' buttons at the bottom right. At the very bottom, there are tabs for 'Settings' and 'Interface', with 'Interface' being the active tab.

For instructions on how to define these data providers in Design Studio, including field-level detail, see "Data Provider Editor" in *Modeling OSM Processes*.

Accessing Data through Data Providers

To use a Data Provider, you include a data element in the order template, define a behavior for it and use an XPath expression to access the Data Provider and extract the data that you wish to display in the data element.

For example, the following XPath illustrates how to call a web service provider instance named "DataInstance" and return the value of the "my_element" view data element.

```
instance('DataInstance')/Data/_root/my_element
```

For XQuery, you would use `vf:instance()`.

Augmenting or Overriding Data

In most cases, a data provider references order data from an external source, another behavior, or as static values defined within the data provider. In addition to these options, you can also add explicit parameter values from within an XQuery or XPath that augment or override the parameters defined in the OSM data dictionary.

For example, the following variable can be declared with parameters that have not been defined within the OSM data dictionary from within an XQuery:

```
declare variable $dataInstanceParams :=
  <params>
    <oms:url>file://users/bdueck/catalog.xml</oms:url>
    <fooParam>barValue</fooParam>
  </params>;
```

You can call a data instance function using a sequence of parameters declared in the variable above. For example:

```
log:info($log, local-name(vf:instance($order/oms:GetOrder.Response/oms:_root/
oms:data[1], 'DataInstance', $dataInstanceParams/*)/*[1]))
```

You can call a data instance function using parameters passed as parameters on the function one by one. For example:

```
log:info($log, local-name(vf:instance($order/oms:GetOrder.Response/oms:_root/
oms:data[1], 'DataInstance', <oms:url>file://us/catalog.xml</oms:url>)*[1])),
```

You can call a data instance function using parameters passed as parameters on the function one by one and include two parameters. For example:

```
log:info($log, local-name(vf:instance($order/oms:GetOrder.Response/oms:_root/
oms:data[1], 'DataInstance', <oms:url>file://us/catalog.xml</oms:url>, <foo>bar</foo>)/
*[1]))
```

Objectel

This adapter provides a reliable transport call into Objectel. Although JMS is an asynchronous protocol, the Objectel adapter itself is not. While JMS simplifies transaction management, recovery, offline capabilities, and security, these benefits are not relevant when considered within the context of a behavior. The JMS adapter utilizes additional resources in the application server in the form of temporary JMS destinations to which Objectel sends the response. These can be expensive if an order has many adapters being called concurrently. It is not recommended to use this adapter in this scenario.

Parameters

- **objectel:extensionName**
Description: the name of the Objectel server extension to call.
Mandatory/Optional: Mandatory
- **objectel:jmsFactory**
Description: the name of the JMS factory to be used to access Objectel's JMS queue.
Mandatory/Optional: Optional
Default value - `com.oracle.objectel.XMLJMS.QueueConnectionFactory`
- **objectel:queue**
Description: The name of the Objectel receive queue.
Mandatory/Optional: Optional
Default value: - `com.oracle.objectel.XMLJMS.QueueConnectionFactory`
- **objectel:allowErrorResponse**
Description: an optional Boolean parameter name that if specified controls what happens if Objectel returns an error response. If this parameter is set to false (default), an error response from Objectel triggers an exception to be thrown which is in turn displayed as a constraint violation. If this parameter is set to true, the error response is returned by the ObjectelAdapter as a valid instance. This allows another Constraint behavior to apply to that same instance and display an error message accordingly. The benefit of using the default (false) is that you do not have to write an additional behavior to display a default error message. The constraint violation message looks like an exception with a stack trace, but shows the error description returned by Objectel at the top of the message. The benefit

of setting this parameter to true is that you have greater control over when the error is shown, at what severity, and what message is displayed.

- false: If this parameter is set to false (the default), an error response from Objectel throws an exception, which is then displayed as a constraint violation. By using false you can avoid writing an additional behavior to display only a default error message. With this method, the constraint violation message looks like an exception with a stack trace, but shows the error description returned by Objectel at the top of the message.
- true: If this parameter is set to true, the error response is returned by the ObjectelAdapter as a valid instance. This allows another Constraint behavior to apply to that same instance and display an error message accordingly. By setting the parameter to true, you have greater control over:
 - * When the error should be shown
 - * The severity level displayed
 - * The exact error message to display.
- All other parameters are passed directly as name/value pairs to the server extension.

Order

This adapter lets you use order data from any OSM order as an external instance. This is useful for using related order data from other orders within OSM.

Parameters

- oms:OrderID
Description: The order ID of the order to be retrieved.
Mandatory/Optional: Mandatory
- oms:View
Description: The view (query task) to use when retrieving order data.
Mandatory/Optional: This is required if the oms:OrderHistID is not supplied.
- oms:OrderHistID
Description: The order history ID to use when retrieving order data.
Mandatory/Optional: This is required if **oms:View** is not supplied.

Adding a New Order Data Provider

To add a new Data Provider which uses the Order adapter:

1. In Design Studio, add a new Data Provider. From the **Studio** menu, select **New**, then select **Order and Service Management**, and then select **Data Provider**.
2. In the Data Provider Wizard, enter a name and folder for the Data Provider and set **Provider Type** to **Order**.
The new Order Data Provider is added to the Design Studio project.
3. Edit the Data Provider.
4. In the Data Provider editor, on the **Input Parameters** section of the **Interfaces** tab, specify values for either the **oms:View** or **oms:OrderHistID** parameters.

5. Set the **Default Value** to either XQuery or XPath and enter your request code in the **Default Value** edit box.
6. Optionally specify the XML structure of the data in the **Results Document** edit box.
The definition of GetOrderResponse is located in the order management web service schema at **SDK\XMLSchema\GetOrder.xsd**.

For more information, see "About Modeling Control Data" in *Modeling OSM Processes*. Also, see "[Accessing Data through Data Providers](#)".

Property File

This adapter retrieves an external Java property file with a given name from the classpath. The format of the XML instance returned by this adapter is specified as:

<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Properties.html>

Parameters

- `oms:url`

Description: Specifies the file name of the Java property file. The file must be on the classpath and must be in the format of a Java property file.

Mandatory/Optional: Mandatory

SOAP

This adapter lets you access web services from OSM or an external web service server, using the HTTP protocol. You can call SOAP web services from OSM or an external web service server and use the responses within behaviors.

Note

If you need to configure a proxy server to access the internet, add the following parameters to the OSM WebLogic server startup script:

```
JAVA_OPTIONS="${JAVA_OPTIONS} -Dhttp.proxyHost=ip_address -Dhttp.proxyPort=port
```

where *ip_address* and *port* are the IP address and port of the proxy server.

For web service calls specific to OSM, use the Web Service adapter. See "[Web Service](#)".

For general web services calls, use the SOAP adapter.

Parameters

- `soap.endpoint`

Description: Specifies the URL to which the SOAP request will be sent.

Mandatory/Optional: Mandatory

- `soap.action`

Description: Contains the URI that identifies the intent of the message.

Mandatory/Optional: Optional

- `soap.envelope`
Description: Specifies the root element of a SOAP message.
Mandatory/Optional: Mandatory, if the `soap.body` parameter is not defined.
- `soap.body`
Description: Contains the SOAP message intended for the endpoint. If the SOAP body node is not included in the `soap.body` content, it will be added by the SOAP Adapter.
Mandatory/Optional: Mandatory, if the `soap.envelope` parameter is not defined.
- `soap.header`
Description: Contains XML data that affects the way the application-specific content is processed by the message provider. If the SOAP header node is not included in the `soap.header` content, it will be added by the SOAP Adapter.
Mandatory/Optional: Optional
- `oms:credentials.username`
Description: Specifies an authentication user name.
Mandatory/Optional: Optional
- `oms:credentials.password`: An optional authentication parameter
Description: Specifies an authentication password.
Mandatory/Optional: Optional
- `oms:credentials.scope.host`: An optional authentication parameter
Description: Specifies an authentication host parameter.
Mandatory/Optional: Optional
- `soap.allowErrorResponse`:
Description: When set to true, the adapter returns SOAP fault messages to the calling behavior; otherwise, the adapter throws an exception when a SOAP fault response is returned.
Mandatory/Optional: Mandatory

Example of `soap.body` Parameter

The following is an example of a SOAP body, which would be populated in the `soap.body` parameter.

```
<instance name="us-addr" xsi:type="externalInstanceType">
<adapter>com.mslv.oms.view.rule.adapter.SOAPAdapter</adapter>
<parameter name="soap.endpoint">'http://ws2.serviceobjects.net/av/AddressValidate.asmx'</parameter>
<parameter name="soap.body" xsi:type="xqueryType">
<soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<sa:ValidateAddress xmlns:sa="http://www.serviceobjects.com/">
<sa:Address xsi:type="soapenc:string">{ ../street/text() }</sa:Address>
<sa:City xsi:type="soapenc:string">{ ../city/text() }</sa:City>
<sa:State xsi:type="soapenc:string">{ ../state/text() }</sa:State>
<sa:PostalCode xsi:type="soapenc:string"/>
<sa:LicenseKey xsi:type="soapenc:string">{ ../soap_license_key/text() }</sa:LicenseKey>
</sa:ValidateAddress>
</soap:Body>
```

```

</parameter>
</parameter name="soap.action">'http://www.serviceobjects.com/ValidateAddress'</
parameter>
<cache>
<scope>NODE</scope>
</cache>
</instance>

```

Example of soap.envelope Parameter

The following is an example of a SOAP envelope, which would be populated in the soap.envelope parameter.

```

<?xml version="1.0" encoding="UTF-8"?>
<com:modelEntity xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:adapt="http://xmlns.oracle.com/communications/sce/osm/model/adaptor"
xmlns="http://xmlns.oracle.com/communications/sce/osm/model/adaptor"
xmlns:com="http://www.mslv.com/studio/core/model/common"
xmlns:prov="http://xmlns.oracle.com/communications/sce/osm/model/provisioning"
xsi:type="adapt:adaptorType" name="Send_Order">
  <com:displayName>Send_Order</com:displayName>
  <com:saveVersion>49</com:saveVersion>
  <com:interface>
    <com:inputParameter xsi:type="adapt:xpathInputParameterType" name="soap.endpoint">
<adapt:contentType>XPATHT</adapt:contentType>
<adapt:defaultValue>'http://localhost:7001/osm/wsapi'</adapt:defaultValue>
</com:inputParameter>
<com:inputParameter xsi:type="adapt:xpathInputParameterType" name="soap.envelope">
  <adapt:contentType>XQUERY</adapt:contentType>
  <adapt:defaultValue>
    <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement">
      <soapenv:Header>
        <wsse:Security soapenv:mustUnderstand="1"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
secext-1.0.xsd">
          <wsse:UsernameToken wsu:Id="UsernameToken-10570647">
            <wsse:Username>username</wsse:Username>
            <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
username-token-profile-1.0#PasswordText">password</wsse:Password>
          </wsse:UsernameToken>
        </wsse:Security>
      </soapenv:Header>
      <soapenv:Body>
        <osm:CreateOrderBySpecification xmlns:osm="http://xmlns.oracle.com/communications/
ordermanagement" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
          <osm:Specification>
            <osm:Cartridge>
              <osm:Name>LGT_PSTN_Corp</osm:Name>
              <osm:Version>1.0.0</osm:Version>
            </osm:Cartridge>
            <osm:Type>LGT_PSTN_CorpOrder</osm:Type>
            <osm:Source>LGT_PSTN_CorpOrder</osm:Source>
          </osm:Specification>

```

```

<osm:Reference>Create by WebService</osm:Reference>
<osm:Priority>5</osm:Priority>
<osm:AutoAddMandatoryData>true</osm:AutoAddMandatoryData>
<osm:StartOrder>true</osm:StartOrder>
<osm:Data>
  <_root>
    <Customer_info>
      <Customer_name>Sample_cust</Customer_name>
      <Customer_Address>Anytown</Customer_Address>
      <Customer_region>1</Customer_region>
      <Customer_contact>391-322-1323</Customer_contact>
    </Customer_info>
    <Order_info>
      <Order_Id> 1000006</Order_Id>
      <Order_version>1 </Order_version>
    </Order_info>
    <Service_info>
      <Service_Type>1</Service_Type>
      <Corp_TelephoneNumber>900893322 </Corp_TelephoneNumber>
    </Service_info>
  </_root>
</osm:Data>
</osm:CreateOrderBySpecification>
</soapenv:Body>
</soapenv:Envelope></adapt:defaultValue>
</com:inputParameter>
  <com:inputParameter xsi:type="adapt:xpathInputParameterType"
name="oms:credentials.username">
    <adapt:contentType>XPATH</adapt:contentType>
    <adapt:defaultValue>'osm'</adapt:defaultValue></com:inputParameter>
  <com:inputParameter xsi:type="adapt:xpathInputParameterType"
name="oms:credentials.password">
    <adapt:contentType>XPATH</adapt:contentType>
    <adapt:defaultValue>'password'</adapt:defaultValue>
  </com:inputParameter>
</com:interface>
<com:implementation xsi:type="adapt:adapterImplementationType">
  <adapt:builtinType>SOAP</adapt:builtinType>
</com:implementation>
<adapt:cache enabled="true">
  <adapt:scope>SYSTEM</adapt:scope>
  <adapt:timeout>15000</adapt:timeout>
  <adapt:maxsize>50</adapt:maxsize>
</adapt:cache>
</com:modelEntity>

```

XML Attachment

This adapter lets you use an attachment from any OSM order as an external instance. It is useful for using related-order-data from other orders within OSM.

Parameters

- **oms:OrderID**
Description: The order ID of the order to be retrieved.
Mandatory/Optional: Mandatory
- **oms:FileName**
Description: The name of the attachment to use when retrieving the order data.

Mandatory/Optional: Mandatory

XML File

This adapter lets you use an XML file accessible from any URL as an external instance. It is useful for integrating external XML data located in a file system, FTP site, from HTTP, or in a Java JAR file.

Parameters

- `oms:url`

Description: The URL of the file to retrieve.

Mandatory/Optional: Mandatory

XML Validation

This adapter validates a provided XML instance document according to a user-defined schema. The document may be provided either as a URL or as an element. The schema may also be provided as a URL or as an element. The returned document conforms to the element specified by `http://xmlns.oracle.com/communications/ordermanagement#ValidationResult`.

Parameters

- `document`

Description: The file name of the XML document to validate.

Mandatory/Optional: Mandatory

- `schema`

Description: The file name of the XSD used to perform the XML validation.

Mandatory/Optional: Mandatory

JDBC

This adapter lets OSM query any JDBC database, then use the results within a behavior. This adapter is particularly useful for acquiring information stored in an external database.

Parameters

- `oms:dataSource`

Description: The JNDI name of the data source providing the database connection information. For example `'mslv/oms/oms1/internal/jdbc/DataSource'`. The data source must be defined through the WebLogic remote console. To do so, navigate from **Edit Tree** to **Services**. Within **Services**, open **Data Sources**.

Mandatory/Optional: Mandatory

- `oms:sql`

Description: The SQL that is sent to the database. To run a SQL stored procedure, this parameter must comply with the format specified by:

<https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/CallableStatement.html>

Mandatory/Optional: Mandatory

- in:1 . . . in:n

Description: 1 to n additional optional input parameters may be supplied that are bound to parameters defined in the oms:sql value.

Mandatory/Optional: Optional

- out:1 . . . out:n

Description: 1 to n additional optional output parameters that are used when calling SQL stored procedures that have output parameters defined. The parameter value specifies the SQL type name of the parameter, and must be defined at:

<https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/Types.html>

Mandatory/Optional: Optional

Web Service

This external instance adapter lets you invoke the GetOrder and FindOrder OSM Web Service operations. The adapter acts as a wrapper around OSM's Web Service API for these two web service operations, allowing them to be called from external instances.

For other web service calls, use the SOAP adapter. See "[SOAP](#)" for more information.

Parameters

- soap.request

Description: Set this parameter to one of the following:

- The contents of what would normally be in the Body element of the web service request. For example, ord:GetOrder or ord:FindOrder.
- A soap:Envelope element, that is, the entire soap request.
- A soap:Body element, that is, the body element of the soap request.

Mandatory/Optional: Mandatory

See *OSM Developer's Guide* for more information about GetOrder and FindOrder web service transactions.

Adding a New Web Service Data Provider

To add a new Data Provider which uses the Web Service adapter:

1. In Design Studio, add a new Data Provider. From the **Studio** menu, select **New**, then select **Order and Service Management**, and then select **Data Provider**.

2. In the Data Provider Wizard, enter a name and folder for the Data Provider and set **Provider Type to Web Service**.

The new Web Service Data Provider is added to the Design Studio project.

3. Edit the Data Provider.

4. In the Data Provider editor, on the **Input Parameters** section of the **Interfaces** tab, select the **soap.request*** parameter.

5. Set the **Default Value** to XQuery and enter the request XQuery code in the **Default Value** edit box. See "[Sample soap.request XQuery](#)" for an example.

You can optionally specify the request as an XPath instance instead by setting the **Default Value** to XPath and entering the request XPath code in the **Default Value** edit box.

6. Optionally specify the XML structure of the data in the **Results Document** edit box.

Definitions of FindOrderResponse and GetOrderResponse declarations are located in the order management web service schema at **SDKIWebServiceIwsdlIOrderManagementWS.xsd**.

For more information, see "About Modeling Control Data" in *Modeling OSM Processes*.

Sample soap.request XQuery

The following is a soap.request XQuery example for a web services Data Provider. You can also specify the input as a SOAP envelope or a SOAP Body.

```
<ord:GetOrder xmlns:ord="http://xmlns.oracle.com/communications/ordermanagement">
  <ord:OrderId>1</ord:OrderId>
  <ord:View>review_details_view</ord:View>
  <ord:AmendmentFilter>
    <ord:LevelOfDetail>AmendmentsSummary</ord:LevelOfDetail>
  </ord:AmendmentFilter>
  <ord:LifecycleEventFilter>
    <ord:RetrieveLifecycleEvents>true</ord:RetrieveLifecycleEvents>
  </ord:LifecycleEventFilter>
</ord:GetOrder>
```

Accessing Data

To use the Data Provider, include a data element in the order template, define a behavior for it, and use an XPath expression to access the Data Provider and extract the data to display in the data element. See ["Accessing Data through Data Providers"](#).

Whenever the Web Service adapter is called through a Data Provider, GetOrderRequest is executed and a response returned. If logging is set to **debug** for the OrderAdapter, a message similar to the one below is displayed on the WebLogic Remote Console:

```
<09-Feb-2012 2:57:57,884 IST PM> <DEBUG> <adapter.OsmWebServiceAdapter> <ExecuteThread:
'10' for queue: 'oms.web'> <<GetOrderResponse xmlns="http://xmlns.oracle.com/
communications/ordermanagement">
  <OrderSummary>
    <Id>16</Id>
    <Specification>
      <Cartridge>
        <Name>view_framework_demo</Name>
        <Version>1.0.0.0.0</Version>
      </Cartridge>
      <Type>vf_demo_web</Type>
      <Source>vf_demo_web</Source>
    </Specification>
    <State>open.running.in_progress</State>
    <Reference>N1</Reference>
    <CreatedDate>2012-02-08T17:55:31.000+05:30</CreatedDate>
    <ExpectedDuration>P1D</ExpectedDuration>
    <ExpectedOrderCompletionDate>2012-02-09T17:55:37.000+05:30</ExpectedOrderCompletionDate>
    <ProcessStatus>n/a</ProcessStatus>
    <Priority>5</Priority>
  </OrderSummary>
  <Data>
    <osmc:_root
      xmlns:osmc="urn:oracle:names:ordermanagement:cartridge:view_framework_demo:1.0.0.0.0:view
```

```

:enter_account_information" index="0">
  <osmc:account_information index="1328703937231">
    <osmc:first_name index="1328703937242">name</osmc:first_name>
    <osmc:last_name index="1328703937243">lastname</osmc:last_name>
    <osmc:country index="1328703937244">US</osmc:country>
    <osmc:address_information index="1328703937233">
      <osmc:address_details_us index="1328703937236">
        <osmc:validate_address_via_soap index="1328703937238">No</
osmc:validate_address_via_soap>
        <osmc:street index="1328703937239">street</osmc:street>
        <osmc:city index="1328703937241">city</osmc:city>
        <osmc:state index="1328703937240">MI</osmc:state>
        <osmc:zip_code index="1328703937237">12323</osmc:zip_code>
      </osmc:address_details_us>
      <osmc:address_details_ca index="1328703937234">
        <osmc:validate_address_via_soap index="1328703937235">No</
osmc:validate_address_via_soap>
      </osmc:address_details_ca>
    </osmc:address_information>
  </osmc:account_information>
  <osmc:info_roopa index="1328703937245">nikhil</osmc:info_roopa>
</osmc:_root>
</Data>
</GetOrderResponse

```

Custom Data Providers

In addition to the built-in data providers described in previous sections, Design Studio supports custom data providers. You can develop a custom data provider class in a project in Design Studio as part of a solution. This provider class must implement the `com.mslv.oms.view.rule.ExternalInstanceAdapter` interface. This interface is documented in the Javadocs distribution found in the OSM SDK.

The implementation class can be made available to the OSM run time system in two ways:

- Package the class into an Java archive (jar file) with an arbitrary name and place the jar file in the resources directory of the Studio project(s) that define Behaviors referencing the data provider. The class will be available as soon as the project is deployed
- Add the compiled adapter class to the **customization.jar** file in the **oms.ear** file. The class will be available as soon as the OSM application is redeployed. See *OSM Developer's Guide* for information about unpacking, packing, and redeploying the **oms.ear** file.

The `ExternalInstanceAdapter.retrieveInstance(ViewRuleContext, Map)` method provides a Map of name/value pairs of arguments defined in the data provider's Design Studio definition and their corresponding values for an invocation of an instance of this class. The `com.mslv.oms.view.rule.adapter.AbstractAdapter` class provides a number of methods to assist in extracting properly type cast parameter values from that Map. `AbstractAdapter` is included in the `automation_plugins.jar` archive found in the **osmLib** directory of a Design Studio OSM project, as well as in the **automation/automationdeploy_bin** subdirectory of an OSM SDK installation.

Handling Parameters

Custom data providers, like built-in providers, support input parameters. The following examples illustrate how to access those parameters.

Example 1 (incorrect usage)

`String stringParamValue = (String) parameters.get(MY_STRING_PARAM);` The value returned by `parameters.get(...)` may not be a `String`, resulting in a `ClassCastException`.

Example 2 (incorrect usage)

`String stringParamValue = parameters.get(MY_STRING_PARAM).toString();` The `parameters.get()` call may return a null value resulting in a null pointer exception. Also, the value returned may be an XML DOM fragment, requiring a more sophisticated mechanism for value extraction than simply calling `toString()`.

Example 3 (correct usage)

`String stringParamValue = getStringParam(parameters, MY_STRING_PARAM);` The `getStringParam(Map, String)` call automatically performs the appropriate conversion to coerce a parameter value into a `String`. This method is intended for extracting a required parameter value. If a value for `MY_STRING_PARAM` was not provided, or if the value cannot be coerced into a `String`, a `BadParameterException` is thrown. To retrieve optional parameter values, use `getStringParam(Map, String, String)` instead; see Example 4.

Example 4 (correct usage)

`String stringParamValue = getStringParam(parameters, MY_STRING_PARAM, "a default value");` The `MY_STRING_PARAM` parameter is retrieved as an optional parameter. If a value for `MY_STRING_PARAM` is provided, it is returned, otherwise, "a default value" is returned. The `AbstractAdapter` class also provides similar methods to extract boolean, numeric, and XML DOM Node parameter values:

- `boolean booleanParamValue = getBooleanParam(parameters, MY_BOOLEAN_PARAM);`
- `int intParamValue = getIntParam(parameters, MY_NUMBER_PARAM);`
- `Node nodeParamValue = getNodeParam(parameters, MY_NODE_PARAM);`

The following code snippet illustrates a simple custom data provider class:

```
/*
 * Copyright © 1998, 2012, Oracle and/or its affiliates. All rights reserved.
 */
package oracle.communications.ordermanagement.example;

import java.util.Map;

import oracle.communications.ordermanagement.util.xml.XMLHelper;

import org.w3c.dom.Document;
import org.w3c.dom.Element;

import com.mslv.oms.view.rule.ExternalInstanceAdapter;
import com.mslv.oms.view.rule.ViewRuleContext;
import com.mslv.oms.view.rule.adapter.AbstractAdapter;

/**
 * <p>
 * This class exemplifies a custom Data Provider. In particular, it demonstrates a
 * provider that returns the familiar "Hello World!"
 * example. The data returned by this provider can in turn be used as input to any
 * Behavior type.
 * </p>
 */
```

```

* <p>
* Like all data providers, this class implements the {@link ExternalInstanceAdapter}
interface. This interface defines a single method,
* {@link ExternalInstanceAdapter#retrieveInstance(ViewRuleContext, Map)
retrieveInstance(ViewRuleContext, Map)}. The
* {@link ViewRuleContext} argument provides various context hooks to this Data Provider
implementation instance. The Map
* argument contains the name/value pairs of arguments defined in the Data Provider's
Studio definition and their corresponding values for
* the current invocation of this Data Provider implementation instance. It additionally
extends the {@link AbstractAdapter} class.
* AbstractAdapter provides a number of utility methods for retrieving
properly type-cast parameters from the parameter
* Map.
* </p>
*
* @author Copyright © 1998, 2012, Oracle and/or its affiliates. All rights reserved.
*
*/
public final class ExampleProvider extends AbstractAdapter implements
ExternalInstanceAdapter {

    /**
     * The name of a parameter that specifies the salutation to return from {@link
     * #retrieveInstance(ViewRuleContext, Map)}. For example, if
     * a value of Goodbye is specified, the message Goodbye World!
code> will be returned. This example does not require
     * this parameter to exist. If it does not, the message Hello World!
will be returned.
     */
    public static final String SALUTATION_PARAM_NAME = "salutation";

    private static final String DEFAULT_SALUTATION = "Hello";

    /**
     * <p>
     * This implementation simply returns the root {@link Element} of a {@link Document}
containing the String "Hello World!"
     * in the contents, i.e., the root of the XML:
     *
     * <pre>
     * <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
     * <response>
     *   <message>Hello World!</message>
     * </response>
     * </pre>
     *
     * </p>
     * <p>
     * The instance('name') Behavior function resolves to the
document root element returned by this method.
     * Therefore, the syntax for locating this provider's message (assuming the Data
Provider associated with this class is named
     * ExampleProvider) is instance('ExampleProvider')/message.
     * </p>
     *
     * @param ctx
     *         provides various context-specific hooks for use by this instance
     * @param params
     *         Map of name/value pairs, where the key is the parameter
name defined in the Data Provider definition that is
     *         associated with this class, and the value is the resolved value of
that parameter for a specific invocation of this

```

```
        *          method. This example does not expect or require any parameters.
        * @return the root <code>Document Element</code>
        */
    @Override
    public Element retrieveInstance(final ViewRuleContext ctx, final Map<String, Object>
params) throws Exception {
        /*
        * This demonstrates how to use the utility methods inherited from
        AbstractAdapter to return a parameter value, though here the
        * "salutation" parameter is not expected to exist.
        */
        final String salutation = getStringParam(params, SALUTATION_PARAM_NAME,
DEFAULT_SALUTATION);

        /*
        * Create the response. An actual provider implementation would likely calculate
        or retrieve the response from an external system.
        */
        final String response = "<response><message>" + salutation + " World!</message></
response>";

        /*
        * The code invoking this method expects a org.w3c.dom.Document root
        org.w3c.dom.Element. The XMLHelper utility class provides a
        * number of DOM manipulation methods, including various String parsers.
        */
        final Document responseDoc = XMLHelper.parseText(response, false);
        return responseDoc.getDocumentElement();
    }
}
```

9

Modeling Behaviors

This chapter describes how to model behaviors in an Oracle Communications Order and Service Management (OSM) solution.

Modeling Behaviors Overview

You can use behaviors to specify how OSM manages data. For example:

- You can specify the maximum allowed number of characters for text string data.
- You can add the values of multiple fields and display the sum in another field.
- You can specify the minimum and maximum times that a data element can be used in an order. For example, an order might require that exactly two IP addresses are added.

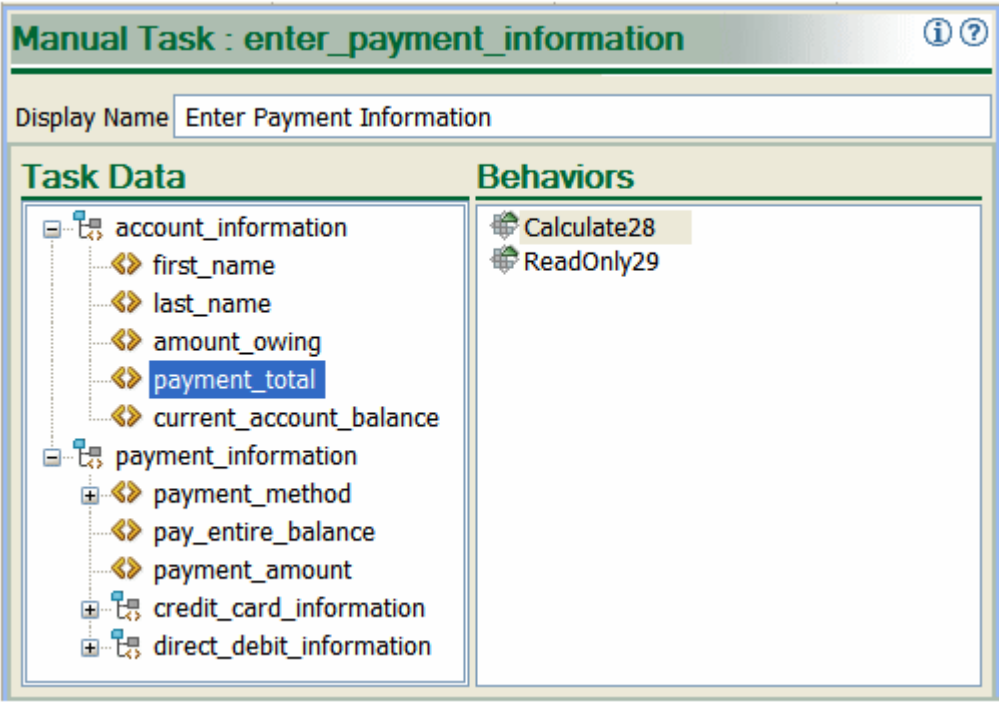
You can model behaviors in tasks and in orders. [Figure 9-1](#) shows how behaviors are modeled in a task that enters payment information. In this figure, the field that shows the payment total uses two behaviors:

- A Calculation behavior that adds values in multiple other fields to create the total payment value.
- A Read Only behavior that makes the field read-only in the Task web client.

Note

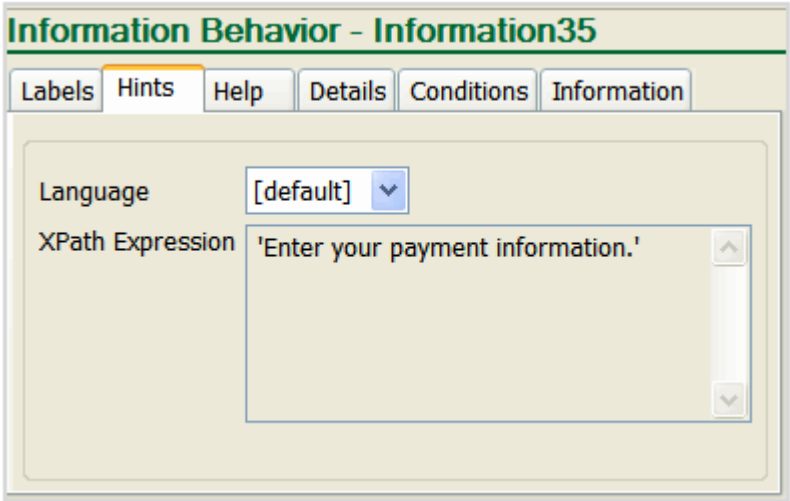
The examples are for illustrative purposes only; OSM is not typically used for payment handling.

Figure 9-1 Behaviors Used in a Task



[Figure 9-2](#) shows a behavior modeled in an order. This behavior is used by an order to display a tool tip for the payment information field.

Figure 9-2 Information Behavior Modeled in Oracle Communications Service Catalog and Design - Design Studio



[Table 9-1](#) lists the OSM behaviors.

Table 9-1 OSM Behaviors

Behavior Name	Descriptions
Calculation	Computes the value of a field value based on a formula that references order data. See " Using the Calculation Behavior " for more information.
Constraint	Specifies a condition that must be met for the data to be considered valid. See " Using the Constraint Behavior to Validate Data " for more information.
Data Instance	Declares an instance that can be used by other behaviors. See " Using the Data Instance Behavior to Retrieve and Store Data " for more information.
Event	Specifies an action that is performed when data is modified. See " Using the Event Behavior to Re-evaluate Data " for more information.
Information	Specifies the label, hint, and help information for the data element instance. See " Using the Information Behavior to Display Data and Online Help " for more information.
Lookup	Specifies a set of dynamically generated choices from which you can select. See " Using the Lookup Behavior to Display Data Selection Lists " for more information.
Read Only	Specifies whether a value can be modified or not. See " Using the Read-Only Behavior " for more information.
Relevant	Specifies whether data is visible or hidden. See " Using the Relevant Behavior to Specify if Data Should Be Displayed in the Web Client " for more information.
Style	Specifies the visual appearance of fields. See " Using the Style Behavior to Specify How to Display Data in the Task Web Client " for more information.

About Behavior Evaluation

It is possible that multiple behaviors can be applied to the same data. At run-time, OSM determines which behavior should be applied by evaluating the conditions defined for behaviors using a combination of server rules and behavior attributes that you model by using Design Studio configuration options. The following configuration options affect the manner in which OSM evaluates behaviors at run-time:

- The level at which you define the behavior. See "[Evaluating Behavior Levels](#)" for more information.
- The manner in which you define the Design Studio Override and Final configuration options. See "[Evaluating Design Studio Final and Override Options](#)" for more information.
- The type of behavior defined for the element. See "[Evaluating Behavior Type Precedence and Sequence](#)" for more information.
- Whether multiple behaviors of the same type are defined for an element at the same level.

Note

The style behavior is the only behavior applied to **Redo**, **Undo**, and **Do Nothing** compensation strategies and the **historical order perspective** displayed in the Task web client. See "[Modeling Compensation for Tasks](#)" for more information about compensation strategies and the historical order perspective.

Evaluating Behavior Levels

In Design Studio, you can create behaviors for data nodes at three levels:

- Data element level (most general)
- Order level (more specific)
- Task level (most specific)

OSM evaluates behaviors from the general level to the specific level. For example, OSM evaluates behavior conditions defined at the data element level first, and evaluates behaviors defined for data nodes at the task level last. At run-time, OSM determines which level to use for a behavior type and data node combination and evaluates rules from that level only.

For example, consider that you create a Calculation behavior at the data element level, and for the same data node you create a Calculation behavior at the order level. In this scenario, OSM would never evaluate the conditions defined for the Calculate behavior at the order level (unless you force evaluation using the **Override** or **Final** options), even if all of the conditions defined for the behavior at the data element level evaluate to false.

OSM does, however, evaluate different types of behaviors defined for a data node at different levels. For example, if for the same data node you define a Calculation behavior at the data element level and a Constraint behavior at the order level, OSM evaluates the conditions for both behaviors at run-time.

Note

The Constraint behavior is an exception to the way in which behaviors are evaluated. When the run-time environment evaluates Constraint behaviors, it evaluates all of them, regardless of the level at which they are declared.

Evaluating Design Studio Final and Override Options

You can force local, specific exceptions to the way behaviors are evaluated for a given node by selecting the **Override** and **Final** check boxes on the appropriate Properties view **Behaviors** tab **Details** subtab in Design Studio. You can select the **Override** attribute to allow the behavior to take precedence over any other behavior:

- Of the same type
- For the same node
- Declared at the same or more general level

For example, consider that you have a data element called *customer* that you declare twice: at the data element level and at the task level. For each occurrence of *customer*, you create a behavior called *styleBehaviorType*. At the specific task level, you select the behavior's **Override** check box. At run-time, OSM evaluates the behavior conditions defined at the task level, as the task-level version of *styleBehaviorType* overrides the data element-level version of the same behavior type.

Note

Override does not function if the behavior that you are trying to override has the **Final** check box selected.

When selected, the **Final** check box prevents another behavior of the same type, for the same node, declared at the same or more specific level, from overriding that behavior.

For example, you define the element *customer* at the data dictionary level (highest), and add it at the task level (lowest). For each occurrence of *customer*, you define a Style behavior. On the data dictionary level (most general) of the behavior definition, you select the **Final** check box. On the task level (lowest) of the behavior definition, you select the **Override** check box. When OSM evaluates the behaviors, the selection of the **Final** check box at the data dictionary level prevents the task level (lowest) definition of the Style behavior from overriding the data dictionary level (highest) definition of the behavior.

Evaluating Behavior Type Precedence and Sequence

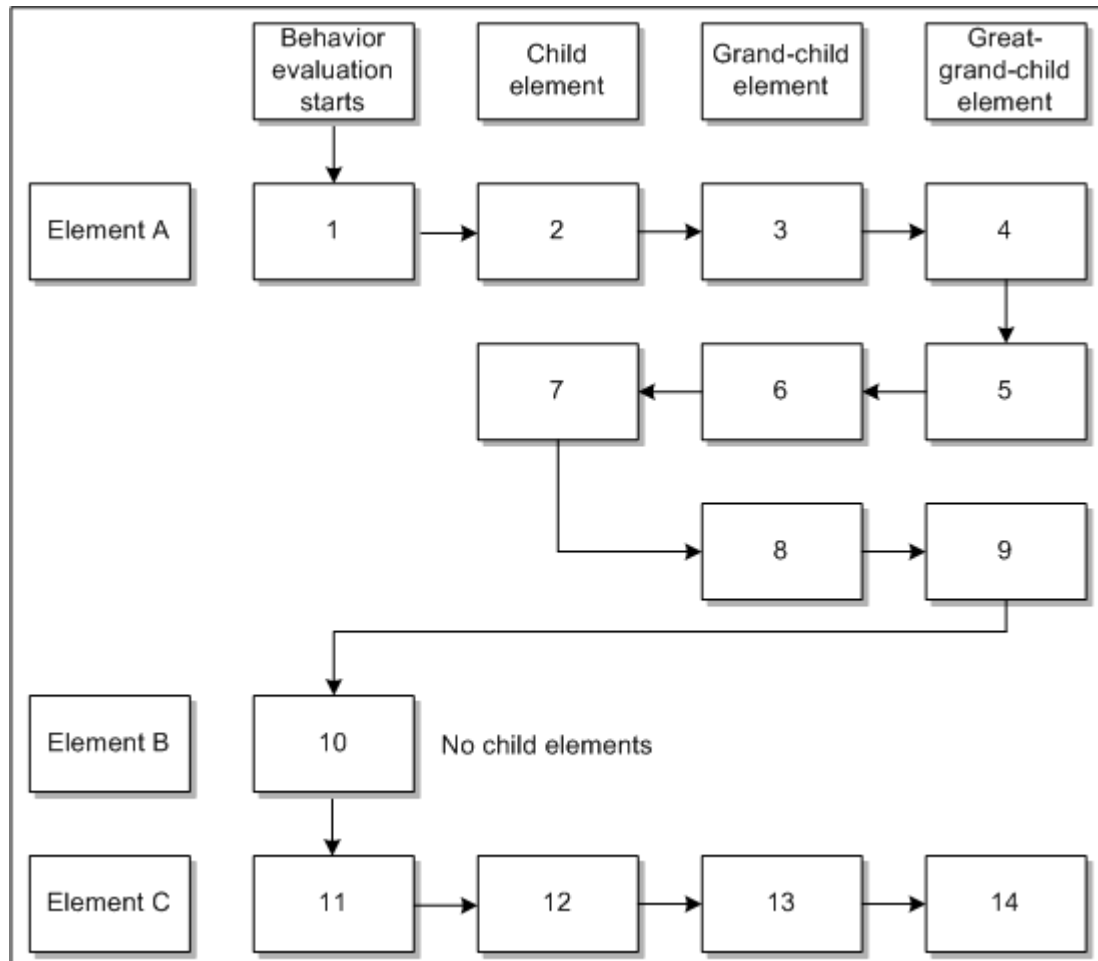
OSM automatically evaluates behaviors whenever you retrieve, save, or transition an order. OSM evaluates the behaviors in a specific nested sequence, as outlined below:

1. The system evaluates all behaviors for a given node before moving to the next node in the order.

The next node in the order is based on a depth first, left-to-right traversal.

[Figure 9-3](#) shows the element selection order.

Figure 9-3 Element Selection



2. Behaviors within a given node are evaluated based on the following precedence of type:

- 1st: Calculate
- 2nd: Style
- 3rd: Information
- 4th: Relevant
- 5th: Lookup
- 6th: Constraint
- 7th: Read-only
- 8th: Event

Note

Relevant rules can prevent other rules from being evaluated. For example, if the Relevant rule of a data node evaluates to false, then rule types with a precedence lower than the Relevant rule are not evaluated (the Lookup, Constraint, Read-only, and Event rules). Additionally, if a data node's Relevant rule evaluates to false, no rule evaluation is done for any descendants of that node.

3. Within an order, within an element, within a specific behavior type, all behaviors defined at a specific data level are evaluated before moving to the next data level.

The evaluation process prioritizes data levels, which are evaluated in the following order:

- Data dictionary level
- Order level
- Task level

Behaviors defined on a task can override behaviors defined on an order if you have enabled the behavior's **Override** option at the task level and if you have disabled the behavior's **Final** option at the order level.

Note

The Constraint behavior is an exception to the way behaviors are evaluated: When OSM evaluates Constraint behaviors, it evaluates all of them, regardless of the level at which they are defined.

Evaluation Process

Within an order, within an element, within a behavior type, within a data level, the evaluation proceeds as follows:

1. **Is the behavior enabled?**

- If the behavior is enabled, the final and override options are evaluated simultaneously.
- If the behavior is not enabled, the behavior is not applied.

2. **Is the behavior finalized or overridden?**

- If the behavior is not finalized and not overridden at a lower level, the condition defined for the behavior is evaluated.
- If the behavior is finalized and not overridden at a lower level, the behavior is final and the condition defined for the behavior is evaluated.
- If the behavior is finalized and overridden at a lower level, the override has no affect; the behavior is final and the condition defined for the behavior is evaluated.
- If the behavior is not finalized and is overridden at a lower level, the condition defined for the overridden behavior is evaluated (not the condition defined for the behavior that is currently being evaluated). If the condition is met, the overridden behavior is applied.
- If the behavior is not finalized and is overridden by more than one lower level, the condition defined for the lowest level overridden behavior is evaluated (not the condition defined for the behavior that is currently being evaluated). If the condition is met, the overridden behavior is applied.

3. **Is the condition defined for the behavior met?**

- If the condition is met, the behavior is applied.
- If the condition is not met, the behavior is not applied.

Note

If you define two or more behaviors for an element at the same level, to avoid unpredictable behavior you should define mutually exclusive conditions. OSM does not guarantee the order of evaluation for the same behavior types defined at the same level.

4. The evaluation process continues.

- If a condition is met, and a behavior is applied, the evaluation process no longer checks lower levels; it moves to the next occurrence of the behavior.
- If a condition is not met, the evaluation process continues with the next occurrence of this behavior type defined at this data level. If there are no more at this level, the evaluation process moves to the next lower level. If there are no lower levels, the evaluation process continues with the next occurrence of this behavior type defined at the highest level, and so on. When there are no more occurrences of the behavior type, the evaluation process moves to the next behavior type, and so on. When there are no more behavior types, the evaluation process moves to the next element.

When the evaluation process determines that a behavior is to be applied at a particular level, some behavior types stop evaluating behaviors of the same type, while others continue evaluating behaviors of the same type at that level for the same element.

For example, you define three behaviors of the same type on the same data element at the same level, and all go through the evaluation process ending with the condition being met (the behavior is applied). For behaviors that stop evaluating, only the first behavior is applied. For behaviors that continue evaluating, multiple behaviors of the same type may be applied, and their effect on the UI is cumulative.

The following behaviors stop evaluating behaviors of the same type after a condition is met and a behavior of the type is applied:

- Calculation
- Lookup

The following behaviors continue evaluating behaviors of the same type after a condition is met and a behavior of the type is applied:

- Constraint
- Event
- Information
- Read Only
- Relevant
- Style

Note

The behaviors in both lists above are presented in alphabetical order, not in behavior type evaluation order.

For example, if three Constraint behaviors are defined, and all go through the evaluation process ending with the behavior being applied, all three Constraint violation messages display

in OSM. In another example, if three Read Only behaviors are defined, if any of them get applied, the field is set to read-only (even if prior and/or subsequent Read Only behaviors evaluate to false). Style and Information behaviors are a bit more complicated in that they have multiple *facets*. The end result is the cumulative effect of these facets. For example, you can define hints and labels with an Information behavior. If one behavior has a hint and another behavior has a label, the end result is that both are applied. If two behaviors define hints, then the second behavior's hint is applied.

Evaluating Multiple Behaviors of Similar Type and Level

When modeling behaviors of the same type, at the same level, for the same data node, ensure that the conditions you define for each behavior are mutually exclusive. When evaluating behaviors of the same type and defined on the same data node and level, the OSM run-time server has no ability to guarantee a predictable order of evaluation. When modeling behaviors for a data node, when it's necessary to define behaviors of the same type at the same level, ensure that you configure conditions that do not rely on a specific order of evaluation.

Additionally, the OSM server evaluates the conditions of each behavior until the conditions of one behavior evaluate to true. Subsequently, OSM does not continue to evaluate any conditions defined for behaviors of the same type and for the same data node.

About Setting Conditions in Behaviors

Conditions enable you to specify when a behavior should function. You set a condition by defining an XPath expression. If the XPath expression evaluates to false at run time, the condition is not met and the behavior is not applied. If the XPath expression evaluates to true at run time, the condition is met and the behavior may or may not be applied, depending on the outcome of evaluation of the behavior at run time.

If no conditions are defined, OSM considers the condition to be met. If multiple conditions are defined, all conditions must evaluate to true for the condition to be met.

Note

The Constraint behavior is the only exception to the way conditions are handled.

Constraint behaviors specify a condition that must be met for the data to be considered valid.

XPath Examples

This section provides XPath examples that are applicable to setting a condition on any behavior type.

- This example shows a condition that evaluates to true when the value of myNumericField is less than 100, and evaluates to false when the value of myNumericField is 100 or greater:

```
../myNumericField<100
```
- This example shows a condition that evaluates to true when the value of myTextField is populated, and evaluates to false when the value of myTextField is an empty String:

```
../myTextField!=""
```
- This example shows a condition that evaluates to true when the value of all three fields are zero, and evaluates to false if any one of three fields are greater than or less than 0:

```
../myNumericField1=0 and myNumericField2=0 and myNumericField3=0
```

Using the Calculation Behavior

You use the Calculation behavior to calculate a field's value based on a formula that references other field values. You can perform numeric operations and string concatenations.

OSM supports the Calculation behavior in the Task web client and in the Order Management web client Data tab.

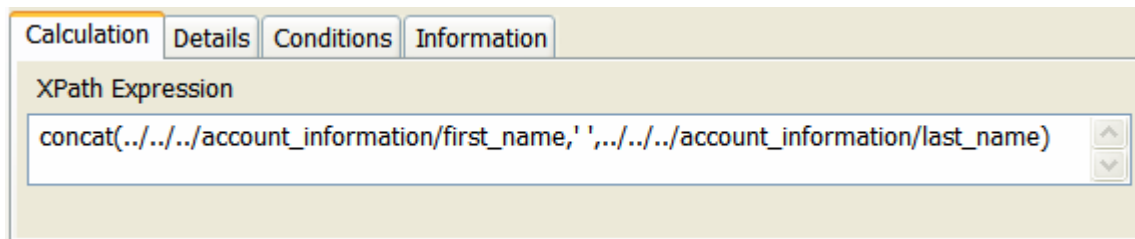
For example, you can use the following expression in a Calculation behavior to calculate the current balance for a customer:

```
../amount_owing - sum(.././payment_information/payment_amount)
```

In this example, the current balance displays the value from the **amount_owing** field after subtracting the value from the **payment_amount** field; the balance = (amount owed) - (amount paid).

[Figure 9-4](#) shows an XPath expression that combines the **first_name** field and the **last_name** field. The Calculation behavior is applied to a field that contains the card-holder name field, where the first and last names are combined into a single field value.

Figure 9-4 Calculate Behavior Formula for Combining String Values



Calculation Behavior XPath Examples

The following examples show how to use XPath statements in the Calculation behavior.

- This example shows how to set a constant value of 100 for a numeric field (whatever number you specify is the number that displays for the field):
`100`
- This example shows how to prefix a constant value to a text field (whatever text you define is the text that displays along with the text value of the field):
`append("any text here",../fieldName)`
- This example shows how to display a numeric field as a result of adding three other numeric fields:
`../fieldName1 + ../fieldName2 + ../fieldName3`
- This example shows how to see the user name of the user who accepted a task:
`/GetOrder.Response/AcceptedUserName`

Calculation Behavior Overview

[Table 9-2](#) shows Calculation behavior attributes.

Table 9-2 Calculation Behavior Attributes

Attributes	Value
Order of evaluation	1st
Default value	None
Applies to	All elements
Parent/child inheritance	Does not inherit (This applies to element relationships within a structure, which is different than the inheritance of behaviors between the data dictionary, order, and task levels.)

Using the Constraint Behavior to Validate Data

You can use the Constraint behavior to validate data that is entered in an order. For example:

- Validate the format of a field. For example, 10 digits for a telephone number, 5 digits for a ZIP code, or an IP address format.
- Validate the range of a field. For example, ensure that a numeric value is between 0 and 100.
- Validate the field value is within a valid list.

In addition to specifying how data is validated, you can:

- Configure messages that indicate the results of the validation; for example, a warning or error message.
- Specify how the order should be processed if the validation fails; for example, stop processing or continue processing.

For example, you might want to ensure that value in a Payment Amount field is less than the amount owed and greater than 0. The Constraint behavior would include this condition:

```
. <= ../../account_information/amount_owing and . >= 0
```

The same Constraint behavior would include the following message to display if the behavior was not met:

```
concat('Invalid payment amount[',..,']. Payment must be greater than 0 and less than  
amount owing of [',../../account_information/amount_owing,']')
```

OSM supports constraint rules in the Task web client.

Displaying Constraint Behavior Error Messages

OSM only displays a Constraint behavior error message if there is a constraint violation caused by the failure of a Constraint behavior condition or by an exception that occurred during the behavior evaluation while you are attempting to either:

- Save an order with invalid field content
- Transition an order with invalid or null values

Otherwise, OSM cues you that a field requires some value by placing a red dot to the left of the field label.

Note

The red dot behavior does not apply to read-only fields. If an error occurs in a read-only field (for example, a failed lookup prevents the display of data) OSM always displays an error message.

The red dot is the same UI element that OSM uses to alert you that a field is mandatory, as defined in the order template. If the field fails the constraint condition and is defined as mandatory, only one red dot appears.

Evaluating Constraint Behaviors

OSM always evaluates Constraint behaviors except when the element or parent element is not relevant, as defined through the Relevant behavior. OSM does not evaluate the Constraint behavior when the task to which the Constraint behavior is associated is at the rollback status. In cases when data is rolled back, it is understood that the Constraint behavior was already evaluated.

Constraint behavior evaluation is different from that of other behaviors. Constraint behaviors are evaluated only when one or more specified conditions evaluate to *false*. All other behaviors are either:

- Always evaluated
- Evaluated only when one or more specified conditions evaluate to *true*.

In addition, when OSM *does* evaluate Constraint behaviors, it always evaluates all of the Constraint behaviors, regardless of where they are defined. This is different from other types of behaviors, where only the first instance of each behavior is selected and applied. However, the **Override** and **Final** check boxes give you control over inheritance. See "[Evaluating Design Studio Final and Override Options](#)" for more information.

Using Task Statuses to Control Process Transitions

You can use task status Constraint values to determine how Constraint behavior violation severity return values affect whether or not a process can make a transition to the next task or activity. Task status Constraint values include:

- **Critical**
- **Error**
- **Warning**
- **None**
- **Valid**

The task status Constraint value represents the highest allowable Constraint behavior violation value with which the task transition will be allowed to occur. When Update is clicked, in the Task web client Order editor, the transition action taken depends on the task status Constraint severity value in conjunction with the Constraint behavior violation severity level, if any.

For example, if the task status Constraint value is set to Error, then Error is the highest allowable Constraint behavior violation value with which the task can be transitioned. The task is not allowed to transition if a Constraint behavior violation of Critical occurs, but is allowed if an Error, a Warning, or a Valid Constraint violation occurs.

The following table explains whether task transition is allowed for all combinations of Constraint behavior violation severities and task status Constraint values.

Table 9-3 Constraint Behavior Actions

Task status Constraint value (highest allowable constraint violation):	Task transition allowed for Critical constraint violation?	Task transition allowed for Error constraint violation?	Task transition allowed for Warning constraint violation?	Task transition allowed for Valid constraint violation?
Critical	Yes	Yes	Yes	Yes
Error	No	Yes	Yes	Yes
Warning	No	No	Yes	Yes
Valid	No	No	No	Yes
None	No	No	No	No

Task Statuses and Constraint Behavior Violation Severity Levels

You can use task statuses in combination with Constraint behaviors to specify the conditions under which a process can make a transition to the next task or activity in the process.

You use Constraint behaviors to validate order data. For example, you can validate that a telephone number has 10 digits or ensure that a numeric value is between 0 and 100.

Constraint behaviors include a **Display as** violation severity level and a message to be displayed in the Task web client when a constraint behavior violation occurs. When Save is clicked in the Task web client Order editor, the save action taken depends on the constraint behavior violation severity level.

Table 9-4 Constraint Behavior Actions

Constraint behavior violation severity levels, from highest severity to lowest	Message display:	When Save is clicked:
Critical	OSM displays the message in bold red text, with the label "ERROR".	The data is not saved.
Error	OSM displays the message in red text, with the label "ERROR".	The data is saved.
Warning	OSM displays the message in yellow text, with the label "WARNING".	The data is saved.
Valid	OSM displays the message in green text, with the label "INFO".	The data is saved.

Constraint Behavior Overview

[Table 9-5](#) shows Constraint behavior attributes.

Table 9-5 Constraint Behavior Attributes

Attributes	Value
Order of evaluation	6th
Default value	True
Applies to	All elements and structures
Parent/child inheritance	Does not inherit (This applies to element relationships within a structure, which is different than the inheritance of behaviors between the data dictionary, order, and task levels.)

Using the Data Instance Behavior to Retrieve and Store Data

You can use the Data Instance behavior to get data from external sources. For example, an order processor using the Task web client can retrieve a set of available ports in real time from an ADSL inventory system.

This behavior differs from all other behaviors in that it has no affect on the UI display of the element for which the behavior is defined. You can think of the Data Instance behavior as a "supporting" behavior because it provides functionality that can be used with other behaviors.

You can use the Data Instance behavior to:

- Store data from an external system and make it accessible to other behaviors.
- Store data that is defined in-line in an XML or XQuery and make it accessible to other behaviors.
- Store data from OSM that is housed in multiple fields but commonly referenced collectively as a single field and make it accessible to other behaviors. For example, the fields **first_name** and **last_name** can be combined in a new data instance **customer_name**.

When you use the Data Instance behavior, you need to specify the **data provider** that you get data from (see "[Using Data Providers to Retrieve Data](#)" for more information). OSM supports several data providers; for example, Oracle Communications Unified Inventory Management (UIM), XML files, and data in the incoming customer order. You can also configure your own data provider.

See "[About Mapping Order Items to Fulfillment Patterns](#)" for an example of how to use a Data Instance behavior.

Evaluating Data Instance Behaviors

When a Data Instance behavior is defined for an element, regardless of the data level, the container is available to the element on all data levels. Because of this:

- The **Override** and **Final** check boxes have no effect on the Data Instance behavior.
- The Data Instance behavior is not part of the evaluation process in terms of prioritization of behavior type, or in terms of prioritization of data level.

Data Instance Behavior XML, XPath, and XQuery Examples

This section provides XML, XPath, and XQuery examples that are applicable to defining a Data Instance behavior.

- This example shows an in-line XML that defines constant values (this could be used to define the values that appear in a dropdown field):

```
<bookStore>
  <books>
    <titles>
      <AlgebraForDummies> <price>30</price> </AlgebraForDummies>
      <GeometryForDummies> <price>35</price> </GeometryForDummies>
      <TrigonometryForDummies> <price>40</price> </TrigonometryForDummies>
    </titles>
  </books>
</bookStore>
```

- This example shows an XPath expression that selects data from an XML file that defines elements (nodes) of bookstore, book, price, and title. This example returns a list of titles with a price greater than \$30:

```
xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
xmlDoc.async=false;
xmlDoc.load("books.xml");
xmlDoc.selectNodes("/bookstore/book[price>35]/title ");
```

- This example shows an XQuery expression that selects data from an XML file that defines elements (nodes) of bookstore, book, price, and title. This example returns a list of ordered titles with a price greater than \$30. The list is returned in variable x:

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

Data Instance Behavior Overview

[Table 9-6](#) shows Data Instance behavior attributes.

Table 9-6 Data Instance Behavior Attributes

Attributes	Value
Order of evaluation	Not applicable. The data instance type is unique in that it doesn't perform any action. It's just a container for data provider instances.
Default value	None
Applies to	All elements and structures
Parent/child inheritance	Children inherit instances declared on parent (This applies to element relationships within a structure, which is different than the inheritance of behaviors between the data dictionary, order, and task levels.)

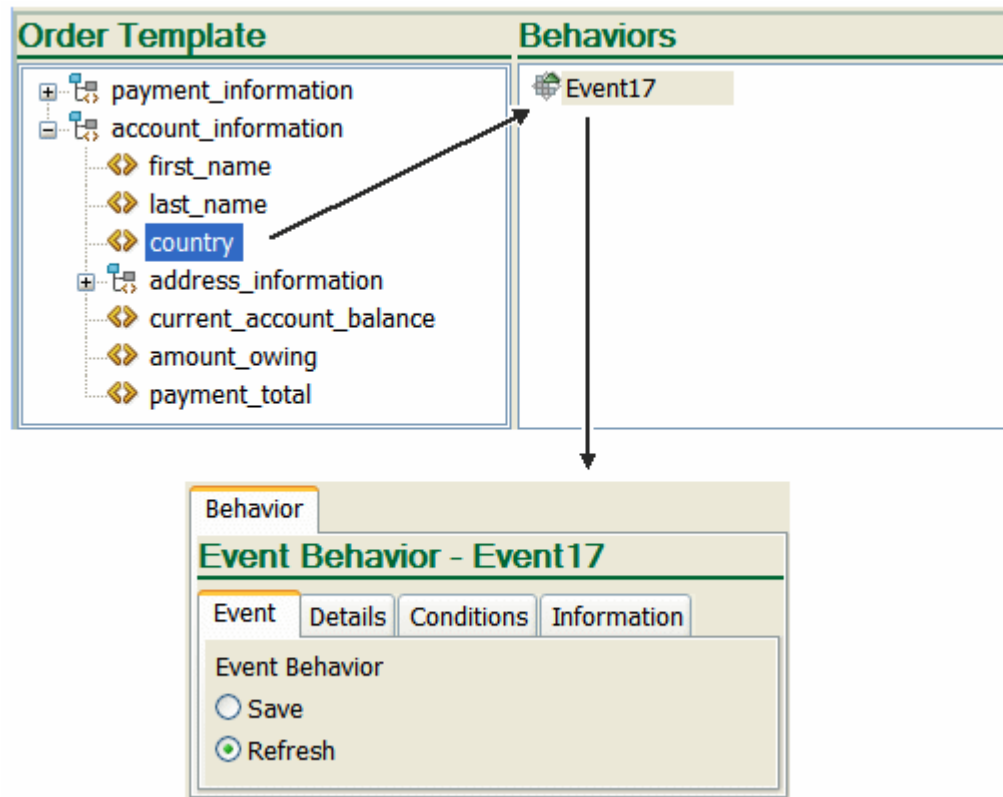
Using the Event Behavior to Re-evaluate Data

You can use the Event behavior to save or refresh data when the data changes. This is useful when a change in a field can cause a behavior to automatically occur in the same field or in another field. For example, you might include an Event behavior in the **account_information/country** field, that causes the data to refresh. That refreshed data might in turn be used by a Relevant behavior assigned to the address details fields that display address information based on the country.

Refreshing causes OSM to re-evaluate all the behaviors associated with the task but does not save the order. **Saving** re-evaluates the behaviors and automatically saves the contents of the order.

[Figure 9-5](#) shows an Event behavior defined in Design Studio. In this figure, the Event behavior refreshes the data in the **account_information/country** field.

Figure 9-5 Event Behavior Defined in Design Studio



See "[Using the Relevant Behavior to Specify if Data Should Be Displayed in the Web Client](#)" for more information on the Relevant behavior, and this scenario in particular.

OSM supports the Event behavior in the Task web client.

Event Behavior Overview

[Table 9-7](#) shows Event behavior attributes.

Table 9-7 Event Behavior Attributes

Attributes	Value
Order of evaluation	8th
Default value	None
Applies to	All elements

Table 9-7 (Cont.) Event Behavior Attributes

Attributes	Value
Parent/child inheritance	Does not inherit (This applies to element relationships within a structure, which is different than the inheritance of behaviors between the data dictionary, order, and task levels.)

Using the Information Behavior to Display Data and Online Help

You can use the Information behavior to specify how data is displayed in the OSM Task web client. You can do the following:

- Set an alternative label for the field. For example, instead of the standard label **State**, the field can be changed to **State or Province** when processing the type of order that uses this behavior setting.
- Localize the field label to one or more different languages.
- Set a tool tip on a field.
- Provide an online help topic for the field.

In the Order Management web client, any information rule on the first instance of a group node that uses a table layout style is used to determine the text of the table panel header. The first instance of each of this group instance's child field nodes are used to determine the column header text for that field node. Hint text for the group instance row and child field instance cells are displayed as tooltip text. Help defined for the group can be run with either a menu item in the table's Actions menu or a row-level context menu and displays help in a modal window in the page containing the table. The implementation of this help behavior differs from the Task web client implementation, which uses an icon in each table cell to load the help in a separate browser window.

OSM triggers information rules when the data element or structure contains data, (for example, from the incoming order or derived from other data sources). If the data element or structure is empty, OSM does not display any label, hint, or help topic information behaviors associated with the empty element or structure. For example, if you defined a label for an element, the label does not appear when the element does not contain a value. Instead, the OSM uses the **Display Name** of the element as defined in the data dictionary.

Information Behavior XPath Examples

This section provides XPath examples that are applicable to defining an Information behavior.

- This example shows an Information behavior label that could be used in conjunction with a Calculation behavior that calculates the current balance based on other fields such as endingBalance + currentCharges + fees - payments:
"Current Balance"
- This example shows an Information behavior label that displays in place of the existing label assigned to the element. For example, the existing label "State" can be changed to display as:
"State or Province"
- This example shows an Information behavior hint that displays when you hover over the *Current Balance* data field:

"The current balance reflects the customer's ending balance, plus any current charges and fees, minus any applied payments."

- This example shows an Information behavior hint that displays when you hover over the *Billing Address* data field:

"The billing address is the address of the party responsible for payment of account. The billing address may differ from the service address. For example, the service address may be a college student's address, and the billing address may be the student's parents address."

Information Behavior Overview

[Table 9-8](#) shows Information behavior attributes.

Table 9-8 Information Behavior Attributes

Attributes	Value
Order of evaluation	3rd
Default value	None
Applies to	All elements and structures
Parent/child inheritance	Does not inherit (This applies to element relationships within a structure, which is different than the inheritance of behaviors between the data dictionary, order, and task levels.)

Using the Lookup Behavior to Display Data Selection Lists

You can use the Lookup behavior to display data in a GUI field that users can select from. You can specify the order of the labels in the list, such as alphabetically.

You can look up data from the following sources:

- Data that is in the incoming customer order.
- Data from an internal source, such as an XML file.
- Data from an external data provider.

Data can be retrieved dynamically based on input. For example, you can look up and populate a list of phones that cost less than \$100, where \$100 is a value obtained from another field in the order.

Note

The Task web client supports two types of lookups: simple lookups with single label value entries, and table lookups, where a single lookup value has multiple associated labels. This latter lookup type is displayed as a text field with an associated icon that launches a secondary window which displays a table of label/value relationships.

In the Order Management web client, simple lookups are fully supported, but complex lookups are rendered as if they were simple: the first-defined label is shown as the display label. In both cases, the field is displayed as a read-only list of values.

Lookup Behavior XPath Example

This section provides an XPath examples that is applicable to defining a Lookup behavior.

This example shows an XPath expression that selects data from an XML file that defines elements (nodes) of bookstore, book, price, and title. This example returns a list of titles with a price greater than \$35:

```
xmlDoc=new ActiveXObject("Microsoft.XMLDOM");  
xmlDoc.async=false;  
xmlDoc.load("books.xml");  
xmlDoc.selectNodes("/bookstore/book[price>35]/title ");
```

Lookup Behavior Overview

[Table 9-9](#) shows Constraint behavior attributes.

Table 9-9 Lookup Behavior Attributes

Attributes	Value
Order of evaluation	5th
Default value	The static lookup values (if any) that are specified in the data dictionary.
Applies to	Elements of data type: • Lookup • Number • Text
Parent/child inheritance	Does not inherit (This applies to element relationships within a structure, which is different than the inheritance of behaviors between the data dictionary, order, and task levels.)

Using the Read-Only Behavior

You can use the Read Only behavior to specify that data displayed in the Task web client is read only. You can specify that data can be read only based on conditions; for example, data can be read only depending on other data in the order.

You typically create read-only fields for fields where the value is derived from other fields. For example, in your order display, you might have two windows: an account window and a payment window. Both windows might have an **Amount Owed** field, which displays the same data. However, you could make the **Amount Owed** field in the payment window the field where the data is collected, and the **Amount Owed** field in the account window read only. In that case, the field in the account window uses two behaviors:

- A Calculate behavior, to get the data from the payment window.
- A Read Only behavior.

Read-Only Behavior Overview

[Table 9-10](#) shows Read-Only behavior attributes.

Table 9-10 Read-Only Behavior Attributes

Attributes	Value
Order of evaluation	7th
Default value	The default specified by the static read-only value.
Applies to	All elements and structures
Parent/child inheritance	If any ancestor evaluates to true , this value is treated as true . Otherwise, the local value is used. (This applies to element relationships within a structure, which is different than the inheritance of behaviors between the data dictionary, order, and task levels.)

Using the Relevant Behavior to Specify if Data Should Be Displayed in the Web Client

You can use the Relevant behavior to specify if data should be displayed in the Task web client or in the Order Management web Client **Data** tab, based on specified conditions.

For example, you can use the Relevant behavior to display address-input fields appropriate to the country that the order applies to. In this example, the Relevant behavior can be used as follows:

- The data for the customer's country is included in the order's **account_information/country** field. This data is displayed in the Task web client in the **Country/Region** field.
- Based on the data in the **account_information/country** field, the customer address fields (**address_information**) can include different values, depending on the country.

Note

The **account_information/country** field includes an Event behavior, which refreshes the data in the field, making it available to the Relevant behavior.

[Figure 9-6](#) shows the address fields for the United States (**address_details_us**) and Canada (**address_details_ca**). The Relevant behavior applies to the selected data, **address_details_ca**.

Figure 9-6 Address Fields in Design Studio

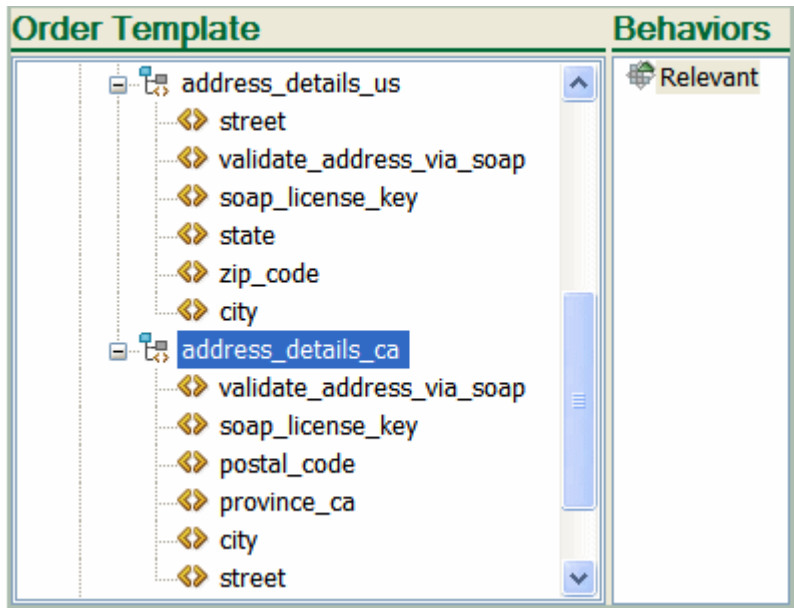
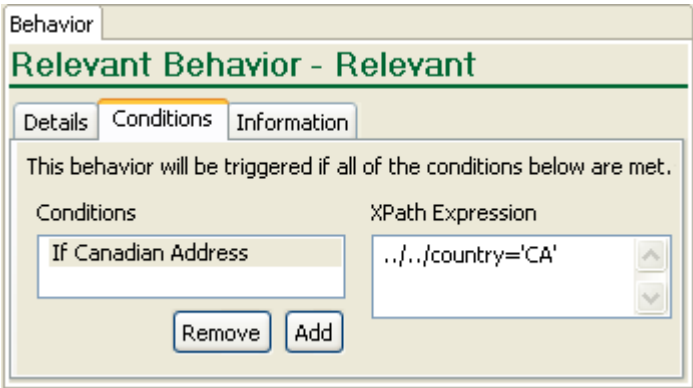


Figure 9-7 shows the XPath expression that specifies the condition (country = Canada) under which the Relevant18 behavior is enabled.

Figure 9-7 Relevant Behavior Properties



In the Order Management web client, if a group instance displayed with a table style behavior is not relevant, then the entire associated table row is omitted. If a particular field is not relevant, the associated table cell is rendered empty.

Relevant Behavior Overview

Table 9-11 shows Relevant behavior attributes.

Table 9-11 Relevant Behavior Attributes

Attributes	Value
Order of evaluation	4th
Default value	True
Applies to	All elements and structures
Parent/child inheritance	If any ancestor evaluates to false , this value is treated as false . Otherwise, the local value is used. (This applies to element relationships within a structure, which is different than the inheritance of behaviors between the data dictionary, order, and task levels.)

Using the Style Behavior to Specify How to Display Data in the Task Web Client

You can use the Style behavior to specify where and how to display data in the Task web client. You can do the following:

- Control the placement of an element on a specific page.
- Specify to display data on tabbed pages. You can display data in columns and tables.
- Hide or mask sensitive data; for example, passwords or credit-card information. You can specify who can read passwords, and you can display a history of password changes. Masked data appears similar to *********.
- Control the layout of a multi-valued field, such as a list of buttons to choose from.
- Apply cascading style sheets (CSS style sheets) to specify how to display data. For example, you could make the current account balance display in red when the data value is greater than zero.

Note

If you define a behavior that contains an apostrophe (') character, OSM will throw an exception error when loading the data. To prevent this from happening, you must include the escape character before and after the apostrophe.

Example:

```
'L'Information De Carte de credit'
```

should be

```
""'L'""Information De Carte de credit""
```

[Figure 9-8](#) shows how the Style behavior changes the appearance of the **Current Account Balance** field in the Task web client.

Figure 9-8 Style Behavior Used in the Current Account Balance Field

Account Information

First Name	Last Name	Amount Owing	Payment Total	Current Account Balance
		1000	0	1000

[Figure 9-9](#) shows the condition that determines if the Style behavior should be applied. In this case, the Style behavior is applied if the account balance is the same as the amount owed.

Figure 9-9 Condition Defined in a Style Behavior

Behavior

Style Behavior - Style

Appearance Layout CSS Style Details **Conditions** Information

This behavior will be triggered if all of the conditions below are met.

Conditions	XPath Expression
Balance = Owing	. > 0 and not(. > ../amount_owing)

[Figure 9-10](#) shows the style definitions to apply to a field.

Figure 9-10 Field Display Colors Defined in a Style Behavior

Behavior

Style Behavior - Style

Appearance Layout **CSS Style** Details Conditions Information

Value

CSS Style Attribute

CSS Class Name

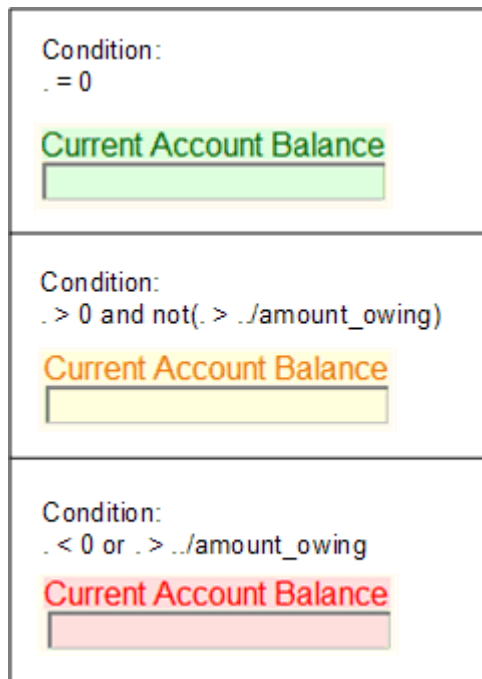
Label

CSS Style Attribute

CSS Class Name

[Figure 9-11](#) shows how three different conditions can change how the field is displayed. If the balance is zero, the field is green. If the balance is the same as the amount that the customer owes, the field is orange. If the balance is less than zero, or greater than the customer owes, the field is red.

Figure 9-11 How Style Behavior Conditions Are Used for Determining the Display Colors



About Style Behavior Layouts

This section provides additional information on table layouts, which you can choose to set as **None**, **Page Layout**, or **Table Layout**.

The **Page Layout** option gives you the ability to organize structure elements onto separate pages that you can access directly, through the use of tabs. This is particularly useful for improving access where there are numerous large structures by eliminating the need to scroll through a single page to find the required structure.

The **Table Layout** option displays multi-instance structures in a grid format. By default, **Table Layout** displays all of the child elements in the structures. However, you can prevent a given child element from being used as a column by setting its `hidden` attribute to true.

Child elements within the structure are represented by columns, and instances of the structure are represented by rows. **Table Layout** displays the columns from left to right in the same order that they appear from top to bottom when displayed without a table layout. If you need to change the order in which the columns appear, you do so by changing their order in the Design Studio order template. The table uses the same child element label to form the column header that it does when displayed without a table layout.

Note

If you use an Information behavior to dynamically change the child element labels, **Table Layout** uses the label associated with the first data instance it encounters.

If you need to hide the value of an individual cell in the resulting table, you can do so by declaring a Relevant behavior for the corresponding child element. See "[Using the Relevant Behavior to Specify if Data Should Be Displayed in the Web Client](#)."

① Note

Table Layout does not support nested structures in the Task web client but does support nested structures in the Order Management web client data tab.

In the Order Management web client data tab, multi-instance child values can be displayed within the corresponding parent value table cell and stacked vertically. You cannot access data history or behavior help from within the cell, but the information about the child multi-instance values can be accessed from the data history for the row. You can access the data history for a row by right clicking on the row and selecting **data history** or by selecting **data history** from the table drop down menu.

OSM uses the first instance of the table group node to determine the CSS style and class of the header text in the Order Management web client. All other style rule attributes of the group instances are ignored. CSS style and class rules, appearance rules, and secret rules are applied to child field (table cell) instances. No other style rules will be applied.

The following figures illustrate the different types of available layouts for the Task web client. Each figure shows the same structure with a different layout type:

- [Figure 9-12](#) shows the structure with no layout applied. With this option, the elements in the structure display within a group box on the original page.
- [Figure 9-13](#) shows the structure with the **Page Layout** option applied. With this option, the elements in the structure display within a group box on a new page that is accessed through a tab on the original page.
- [Figure 9-14](#) shows the structure with the **Table Layout** option applied. With this option, the elements in the structure display within a grid on a new page that is accessed through a tab on the original page.

Figure 9-12 Task with No Layout in the Task Web Client

Figure 9-13 Page Layout in the Task Web Client

Buttons: Preview, Full Preview, Process History, Change State/Status, Save, Add Remark, Exception

▼ Enter Payment Details

Order ID: 61 Order Type: VF Demo Order Source: Web Process: Order Task: Enter Payment Information Priority: 5

Reference #:

Account Information | Payment Information

▼ Account Information

First Name	Last Name	Amount Owning	Payment Total
		9000	0

Figure 9-14 Table Layout in the Task Web Client

Buttons: New Order, Worklist, Query, Reporting, Notifications, Options

Buttons: Preview, Full Preview, Process History, Change State/Status, Save, Add Remark, Exception

▼ Enter Payment Details

Order ID: 61 Order Type: VF Demo Order Source: Web Process: Order Task: Enter Payment Information Priority: 5

Reference #:

Account Information (1.5)

First Name	Last Name	Amount Owning	Payment Total	Current Account Balance
		9000	0	9000

The following figures illustrate the different types of available layouts for the Order Management web client. Each figure shows the same structure with a different layout type:

- [Figure 9-15](#) shows the structure with no layout applied.
- [Figure 9-16](#) shows the structure with the **Table Layout** option applied.

Figure 9-15 No Layout in the Order Management Web Client

Order ID 14: Order Details

Summary **Data** | Orchestration Plan | Dependencies | Amendments

OrderItem List (1...999999999)

OrderItem

Point Of No Return
OrderItemFulfillmentState

lineId 2
lineItemName Fixed Bundle [Add]

OrderItem

Point Of No Return
OrderItemFulfillmentState

lineId 3
lineItemName Fixed Service [Add]

Figure 9-16 Table Layout in the Order Management Web Client

Order ID 15: Order Details

Summary **Data** | Orchestration Plan | Dependencies | Amendments | Activity

Actions View Detach

Row No.	lineId	lineItemName	lineItemPayload	productClass
1	1	Triple Play Offer [...]	<im:salesOrderLin...	
2	2	Fixed Bundle [Add]	<im:salesOrderLin...	
3	3	Fixed Service [Add]	<im:salesOrderLin...	
4	4	Commercial Fixed...	<im:salesOrderLin...	Fixed Service
5	5	Fixed Caller ID [A...	<im:salesOrderLin...	Fixed Service
6	6	Fixed Call Waiting...	<im:salesOrderLin...	Fixed Service
7	7	Mobile Service Pla...	<im:salesOrderLin...	
8	8	Prepaid Mobile Voi...	<im:salesOrderLin...	
9	9	Prepaid Mobile Ser...	<im:salesOrderLin...	Prepaid M...
10	10	Mobile Voice Servi...	<im:salesOrderLin...	

About Style Behavior Password Fields

Behaviors that define password fields can ensure unauthorized users cannot view the contents of elements containing sensitive information. For example, by using this feature you can define a password field in such a way that users in an activation work group can not see the information, but users in the system administrator's work group can.

How Password Fields Display

If you define a password behavior on a writable field, OSM displays the contents of the field as specified by the browser, such as a line of asterisks (*) within a text box. If you define this feature on a read only field, OSM displays the data as specified by the browser, such as line of eight asterisks next to the field label, but not within a text box.

If you open the data history, OSM displays when and by whom the data was modified. When this feature is applied to a field, OSM displays the password field values as specified by the browser, such as a line of eight asterisks.

While you can define a Style behavior on all types of elements, this feature of the Style behavior has no effect on structures.

Do Not Use Password Field Feature with Boolean and Lookup Fields

Because this feature is designed for use with free form entry fields, as opposed to fields that force you to select from a limited number of choices, Oracle recommends that you do not use this feature with Boolean and lookup fields. If you do, you risk exposing confidential information to unauthorized users. This is because OSM displays the value that was previously set in a Boolean or lookup field, even if the field defines this feature through a Style behavior.

Displaying the Data History of Password Fields

OSM only evaluates behaviors at the web UI level, so any password field that you save (that is, create, update, or delete) through the XML API/Automator is not treated as a password field, even if it is defined as such. This can introduce some complexity into how OSM displays the data history for password fields. Use the following general guidelines and examples to understand how OSM displays password field data history.

General Guidelines

1. OSM displays a line of eight asterisks in the data history for any field that it evaluates as a password field (providing the field actually contains data; if the field is empty, OSM displays nothing).
2. If OSM does not evaluate a field as password field, the data history values are shown in plain text.
3. If OSM evaluates a data field as a password field at the time of saving, and the field is later deleted, OSM displays a line of eight asterisks in the data history (providing the field actually contains data; if the field is empty, OSM displays nothing).
4. If OSM evaluates a data field as a non-password field at the time of saving, and the field is later deleted and evaluated as a non-password field at the time of deletion, the data history is displayed as plain text.

Examples

1. If you save the value of a password field through OSM, and OSM is still evaluating the field as a password field when you display the data history, OSM displays the value of the password field as eight asterisks.

2. If you save the value of a password field through the XML API/Automator, and it is still present in the order editor (that is, it has not been deleted by the XML API/Automator) when you display the data history, OSM displays the value as eight asterisks.
3. If you create and delete the password field values through the XML API/Automator, OSM displays the data history values as plain text.
4. If you enter data in a non-password field through OSM and a user subsequently deletes the value through OSM or the XML API/Automator (and OSM evaluates the field as a non-password field at the time of deletion), the history values of this field are displayed as plain text.

Style Behavior Overview

[Table 9-12](#) shows Style behavior attributes.

Table 9-12 Style Behavior Attributes

Attributes	Value
Order of evaluation	2nd
Default value	Data type specific: • For Boolean type fields: Compact • For Lookup type fields: Minimal
Applies to	Elements of data type: • Boolean • Lookup Elements with Lookup behaviors that display only one column.
Parent/child inheritance	Does not inherit (This applies to element relationships within a structure, which is different than the inheritance of behaviors between the data dictionary, order, and task levels.)

Modeling a TMF Solution (Cloud Native Only)

This chapter describes how to model a TMF solution.

Before learning how to model a TMF solution, see the chapter about TMF concepts in *OSM Concepts*.

Note

TMF solution modeling is supported for OSM cloud native deployments only.

About Specifications

The first consideration when modeling a TMF solution is to define the specification to be used.

The canonical TMF specifications (both 622 and 641) have some shortcomings:

- Orders cannot be suspended or resumed
- Orders cannot be aborted
- Existing orders cannot be revised (amended)
- TMF has no state to indicate fallout

Because of this, it is strongly recommended to start with the OSM extended specifications which align with OSM's advanced order management capabilities.

Once you have established the base specification, you should determine whether any schema extensions are needed.

- Does your billing system need extra information that is not found in the existing `billingAccountRef` schema?
- Does your `productOrder` need to track shipment tracking information?
- Are you integrating with an edge system that requires data that does not exist inside the specification of interest?

These are the types of questions that you need to answer and then modify the schema of the specification accordingly. See OpenAPI Specification documentation as well.

All schema changes that you make must result in a version increase inside the specification. The 5th digit of the version is provided for your use.

About Cancelling or Revising an Inflight Order

The canonical TMF 622 and 641 specifications include support for order cancellations but not for revisions. When support for revisions is a requirement, then the OSM extended specifications must be used (either directly or as the base for customer schema extensions).

Cancel and revise are both implemented using the TMF pattern of a task resource. Refer to the TMF 630 guidelines for further details.

Cancel Specific Behavior

Requested cancellation date in the cancel payload is not supported. You cannot request a future date for cancellation via this field. All requests are processed immediately.

Revise Specific Behavior

Revisions to a future-dated order (that are in a TMF state of Pending) are processed immediately. The waiting base order would be updated with the revised data.

The task resource representing the revision request will complete, the OSM revision order will complete and the base order will reflect the amended data and will remain in TMF pending state (OSM waiting state), until the future-dated date arrives.

Design-time Considerations

For TMF cartridges, cancellations are processed as an amendment with no order lines, which would trigger undo mode of all completed tasks.

No order level configuration is necessary to support cancels or revisions, including:

- The cancel fulfillment mode does not need to be created in Design Studio.
- Amendment tab configuration for an order (will be read only for TMF orders).

Automation tasks still need to configure the various execution modes.

Runtime Considerations

OSM Gateway exposes endpoints for cancelling or revising a product or service order and all existing cancel or revise mechanisms (Task Web client, SOAP webservice, and so on) are no longer permitted.

OSM Gateway rejects cancels or revisions where the base order contains any order item in a final TMF state (partial, failed, completed).

Order Key

In Freeform cartridges, order key configuration is needed as a way to correctly identify the base order the revision is applied to. It did not reflect the OSM order id but a piece of data inside the order payload that could be used as a unique identifier of the base order.

In TMF cartridges, identifying the correct base order is done simply by supplying the base order id in the request. Both revise and cancel requests contain a reference to the base order id as shown in the following sample:

```
REVISE
  ReviseProductOrderOSM_Create:
    required:
      - baseOrderRef
      - productOrder
    type: object
    description: OSM extension to create the ReviseProductOrder resource.
    This represents an OSM revision request.
    properties:
      baseOrderRef:
        $ref: '#/components/schemas/BaseOrderRefOSM'
      productOrder:
```

```
$ref: '#/components/schemas/ProductOrder_Create'
.....
```

In the above example, the BaseOrderRefOSM has the id of the targetted base order.

```
CANCEL
  CancelProductOrder_Create:
    required:
      - productOrder
    type: object
    properties:
      cancellationReason:
        type: string
        description: Reason why the order is cancelled.
      productOrder:
        $ref: '#/components/schemas/ProductOrderRef'
```

In the example above, the ProductOrderRef has the id of the targetted base order.

Version

Version is still an important piece of the amendment functionality. However, the handling is different for TMF Orders. Instead of the order carrying the version information inside the payload, it is passed as the HTTP header "X-VERSION". Providing this header is optional for cancels, but mandatory for revise operations.

Callers can supply the HTTP header X-VERSION on the create request, but if omitted (as is generally the case for create), then OSM starts versioning at 1. When a GET is invoked, the header "X-VERSION" indicating the version of the order currently being processed, is returned to the caller.

Once the caller has this information, any subsequent requests for revise or cancel can include an incremented count.

While the mechanism to pass the version information is different for TMF orders, the logic that dictates acceptance or rejection is the same. Orders with a version higher than the order currently processing version are accepted and those with a version lower are rejected.

Impact of PONR

When a cancel request is received, OSM Gateway checks if any order items are in a final state and if so, the request is rejected. This is a form of implicit PONR as TMF states are final (partial, failed, completed). Therefore, a cancel cannot be performed.

If OSM Gateway accepts the cancel request, there is still a chance for the cartridge to reject it if the order or order items have reached PONR. See "[Modeling PONR](#)" for details about how the lifecycle policy can reject orders.

Grace Period

A grace period refers to a configurable period of time that OSM will wait for currently processing tasks to complete, before transitioning an order. In Freeform cartridges, you can pass a grace value during Web Service API invocation.

TMF orders do not allow you to supply a grace period when invoking the REST APIs. Grace period must be defined within the cartridge for TMF orders.

Events about Cancel and Revise Task Resource

The following events are emitted automatically to the event target system in response to lifecycle milestones of the task resource.

Table 10-1 Cancel and Revise Task Resource Events

OSMGW Endpoint	Task Resource Events
/cancelProductOrder	cancelProductOrderCreateEvent cancelProductOrderStateChangeEvent
/reviseProductOrder	reviseProductOrderCreateEvent reviseProductOrderStateChangeEvent
/cancelServiceOrder	cancelServiceOrderCreateEvent cancelServiceOrderStateChangeEvent
/reviseServiceOrder	reviseServiceOrderCreateEvent reviseServiceOrderStateChangeEvent

Task Resource Sequence Diagram

The lifecycle of a task resource is different from the main resource. The following diagrams show how the two lifecycles interact and when events are delivered in relation to the main resource events.

Figure 10-1 Cancel Task Resource Event

Cancel Order

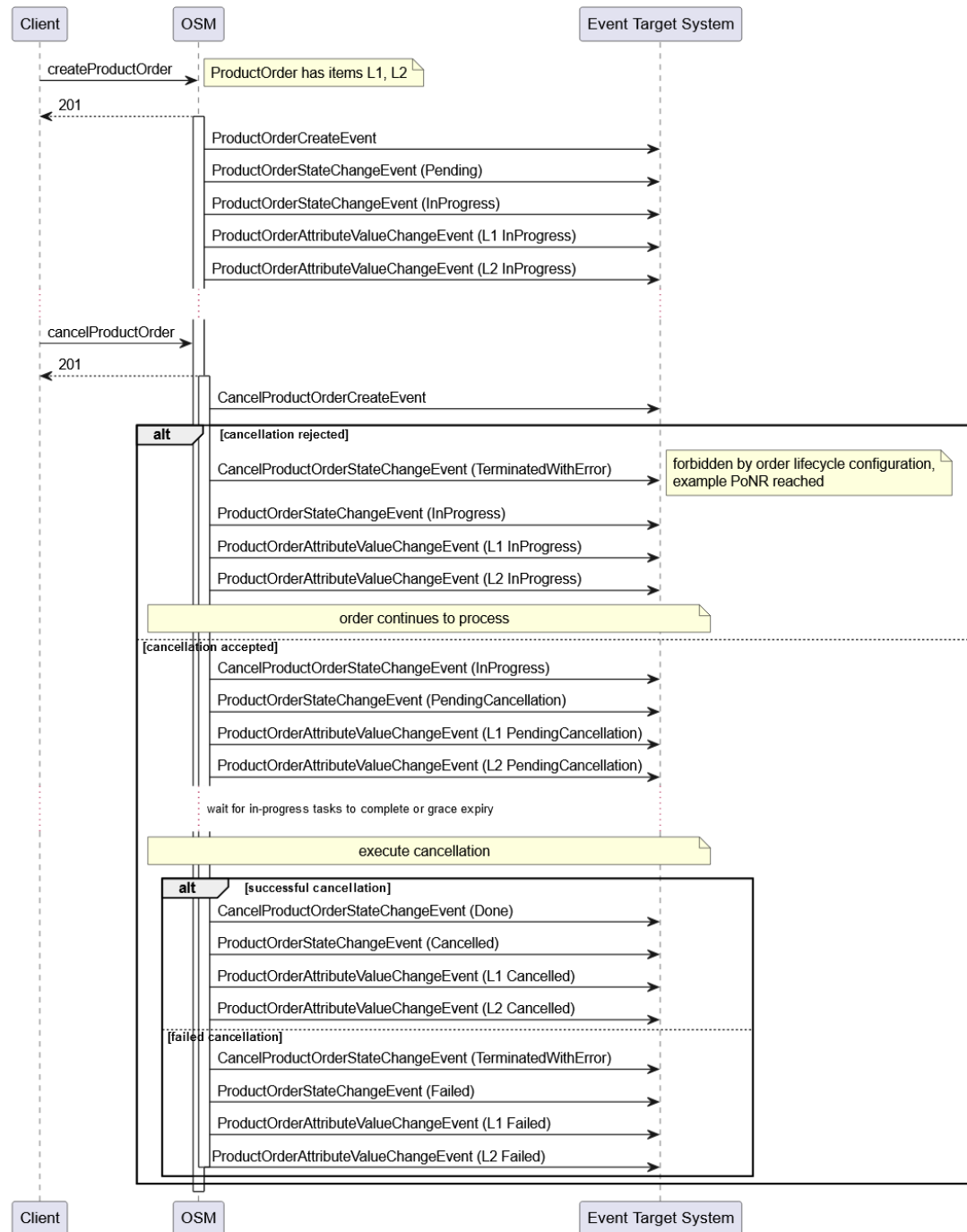
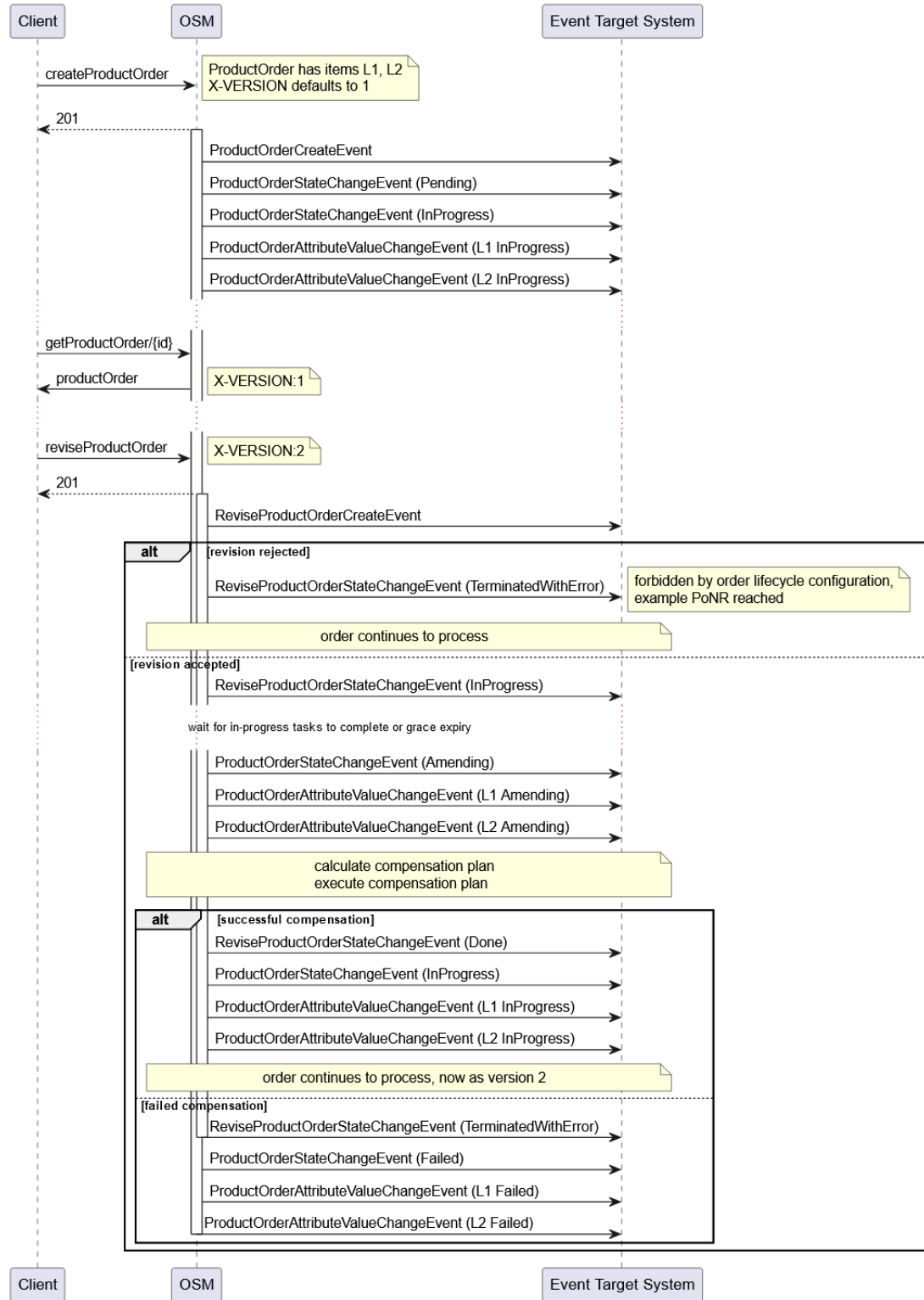


Figure 10-2 Revise Task Resource Event

Revise Order



Modeling PONR

For TMF orders, the modeling pattern with respect to the Point of No Return of an order or order item is different.

PONR handling consists of the following main areas:

- PONR detection
- Propagation of the PONR to the upstream system
- Order amendment and cancellation handling for PONR

PONR detection

For PONR detection, Freeform cartridges and TMF cartridges require slightly different approaches.

- For Freeform cartridges, PONR is configured within a Fulfillment Pattern. An Order Component is selected, and then a specific fulfillment state value can be chosen as the PONR.
- TMF cartridges are required to model the PONR as an order item characteristic within the product model. Population of this field signals PONR has been reached.

In both cases, an automation plugin updates a piece of order data in response to values received from the edge system. With plugins in the Freeform cartridge, this process is a mechanical update to the fulfillment state field.

In the case of TMF cartridge plugins, they must be aware of which characteristic needs to be updated, and also what value from the edge system results in a change to this characteristic. For example, a field "PONRReached" that takes Y and N values would be set to N on order creation. At some point during fulfillment, an automation plugin would change this value to "Y" based on the external fulfillment state received from the edge system.

Propagation of PONR to upstream system

Like PONR detection, PONR propagation is also handled differently for Freeform and TMF cartridges:

- For Freeform cartridges, Model Driven Milestones (MDM) is the mechanism to propagate the PONR to upstream systems.
- TMF cartridges rely on the AttributeValueChangeEvent (Product and Service) to inform upstream systems of a change to the characteristic representing PONR status. Because attribute events are triggered for changes to ANY characteristic value, it becomes the responsibility of the event listener to identify the characteristic holding PONR status and check its value.

Order amendment and cancellation handling for PONR

For both TMF and Freeform cartridges, the handling of amendment and cancellations for orders and order items that may have reached PONR is the same. In both types of cartridges, the objective would be to reject cancellation or revision requests when PONR has been reached. To do this, the order lifecycle policy would have a rule condition defined with a backing XQuery that is responsible for scanning either the fulfillment state field (Freeform cartridges) or the characteristic value (TMF cartridges) to either pass or fail the condition.

Implicit PONR

For TMF cartridges only, there is an implicit PONR. When any order item reaches a final TMF state, then the entire order can be considered as having reached PONR and cancellations and amendments would be rejected by OSM Gateway.

Change Order Support

Change orders that make their way through the stack can be modeled in one of two ways:

- The entire order is included in the change request, lines that have changed as well as lines that have not
- Only the content that has changed is included (and often the necessary parent lines)

Choosing between the two depends on many variables including the product model and capabilities of the edge systems. Neither TMF nor OSM are prescriptive about which modeling pattern is used. However, there are some details within the specifications that may influence which technique is ultimately employed.

Canonical TMF

TMF ordering specifications support action codes on a line item to identify whether it holds a change. Beyond that, there is no support inherent in the TMF specification for identifying *what* the change is. When there is no indicator as to the specific change, the downstream systems must be idempotent, which implies that those systems own the data and therefore have a view of the current state of the object. When passed a new set of data, they can determine what the change is and what action to take. This puts the onus on the systems that OSM talks to, for determining what data is changed and whether it requires an action or not.

This pattern can be expensive for systems and whether or not external systems in the ecosystem behave according to the TMF view, cannot be dictated by the Product Ordering layer.

OSM Extended Specifications

On lines with an action code of "modify", changes are often contained within the characteristic set - new ones added, characteristics no longer required or simply a change of value on an existing characteristic. Examples include an upgrade from 5MB to 10MB bandwidth, or an upgrade to a premium service which results in new characteristics for callWaiting and callForwarding.

OSM recognizes this and offers optimized handling of characteristic changes via the OSM extended specification.

OSM schema extensions have been made to provide additional details about a characteristic - an action code as well as the previous value. This of course relies on the upstream system to populate this data, but if utilized, it can relieve pressure on the downstream systems to calculate the exact nature of the change.

See the TMF 622 REST Specification for change order modeling examples.

Order Fulfillment Modes

The runtime handling for fulfillment mode is different for TMF orders. With Freeform cartridges, something in the order data would need to map to the right fulfillment mode. With TMF orders, the fulfillment mode does not need to be embedded into the payload and for order creation nor

for order cancellations, The calling system in fact does not need to do anything at all. OSM has assumed responsibility for mapping these two order operations properly.

If an additional fulfillment mode was modeled in the TMF cartridge (for example, TSQ), then the calling system would need to supply this value through the HTTP header "X-Fulfillment-Mode". OSM Gateway would forward the FF mode to OSM and would match against deployed fulfillment modes.

Upstream Listener

The following restrictions apply to the outbound TMF messaging:

- An Event Target System must be defined on the hosted order specification in Design Studio.
- Only a single system can be configured. It is expected that this single listener will either consume the events itself, or serve it to a message broker (for example, Kafka).
- The event target system would be the intended recipient of all events that are emitted. The upstream system can ignore messages it is not interested in.

Fault tolerance configuration for target systems is available in the toolkit's specification files. See *OSM Cloud Native Deployment Guide* for details on configuring target systems.

About TMF Order Events For the External Event Listener

The events for TMF orders run by OSM are listed in the REST API reference guide. See REST API reference guide for the details of the events offers and the schema of the events.

Additionally, OSM's framework does the following:

- OSM has restricted the changes that will trigger the AttributeValueChange event. Changes are limited to the order item state or the order item characteristic fields.
- OSM will include multiple updates in a single event. All updates - whether within a single line item or across multiple line items - will be included in an event, so long as they are all made within the same orderUpdate call to OSM core.

See the "About TMF Orders" section in *OSM Concepts* for details about OSM Events Notification.

About Fallout Exception Management

OSM provides a simplified fallout exception management framework for managing fallout exception. See the *OSM Concepts* guide for more information.

For TMF orders, use the simplified fallout management framework only.

Implementing a TMF Solution (Cloud Native Only)

This chapter describes how to implement a TMF solution.

Before reading this chapter, see the chapter about TMF concepts in *OSM Concepts*.

Note

TMF solution modeling is supported for OSM cloud native deployments only. You need to ensure that the cartridge management variable **OSM_RUNTIME_TYPE** value is set to "MultiService".

Accessing the Specifications

The TMF specification files are available in the cloud native SDK download file, in the **TMFSchemas** sub-folder. OSM extensions have the suffix **-OSM** in the filename.

Oracle recommends that you use the specifications with OSM extensions (**-OSM** suffix in the filename) for both Product Order and Service Order. These extensions provide comprehensive access to OSM's advanced capabilities and have been created in line with the guidelines from TMF630.

About Extending the Specifications

The OSM extended specifications can be extended further to meet your implementation requirements by:

- Adhering to TMF630 and OpenAPI guidelines on schema extensions.
- Adding to the schema of the primary object (ProductOrder or ServiceOrder) in terms of order data.
- Keeping the set of Paths (endpoints) and their contents unchanged.
- Keeping the set of Notifications (events) and their contents unchanged.
- Keeping the set of Task Resources and their contents unchanged.
- Keeping the set of State definitions unchanged (preserve the set of states as-is).

The Hosted Order specification consists of the TMForum specification with OSM schema extensions, all expressed as an OpenAPI 3.0 document.

See the following topics:

- [Considerations When Extending the Main Resource](#)
- [About Versioning the Specifications](#)
- [About the "ANY" Schema Type](#)
- [About anyOf, allOf, and oneOf](#)

Considerations When Extending the Main Resource

TMF uses a pattern of defining a schema object for the incoming create order payload, that is different from the main resource. Using TMF 622 as an example, the POST /productOrder endpoint accepts a payload with a schema type **ProductOrder_Create**, which is specific to the creation request. It does not contain some data found in the ProductOrder schema, such as the id, as this is populated by the application not provided by the caller.

On the OSM side, the TMF order template includes the CDT structure representing the main resource, which is "ProductOrder" in this case.

The ORR data transformation xquery is responsible for transforming the **ProductOrder_Create** into the ProductOrder. You are free to do this translation as you see fit. However, the documented solution relies on a simple naming convention. See the Order Data Rule section.

This xquery depends on the main resource name to be related to the incoming schema name in a specific way. When processing the incoming root element, the xquery strips off **_Create** and uses the remaining string as the root for the OSM order data. To use this xquery, any extensions that are made to the main resource must have a schema object for creation that follows the *Resource_Create* convention.

Example: Incompatible naming

```
ProductOrder_Create
ProductOrder_CreateOSM
```

Looking at the above example, if you had named the schema for creation as ProductOrder_CreateOSM, then the xquery would not have done the transformation correctly and order submission would fail.

Instead, the extension is called ProductOrderOSM_Create. The xquery determines correctly that the root resource type is ProductOrderOSM, which is an extension of the canonical ProductOrder.

Example: Compatible naming

```
ProductOrder_Create
ProductOrderOSM_Create
```

If extensions are made to the main resource (ProductOrder or ServiceOrder), consider this naming convention.

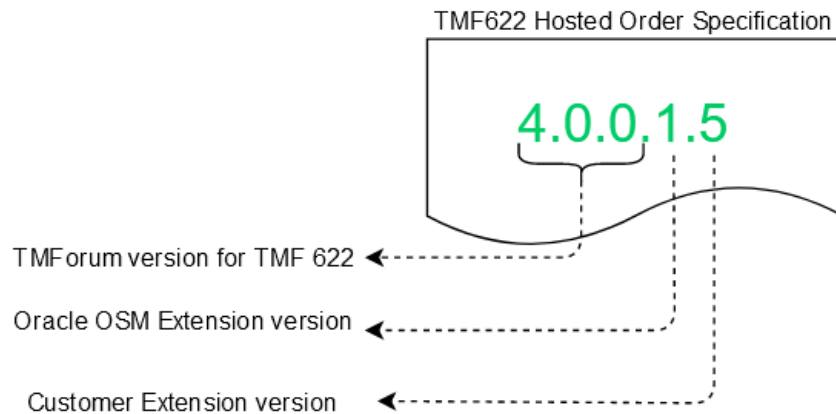
About Versioning the Specifications

When you import a TMF specification into Design Studio, the resulting hosted order specification will have a 5-digit scheme. This is true even if the canonical is imported and there were no extensions added. The TMF specification would say "v4". However, this would be converted to **4.0.0.0.0** for usage inside OSM.

The first three digits are reserved for TMForum as TMF specifications are versioned with a three-number scheme.

The fourth number is reserved for Oracle OSM to version extensions supported by the application itself.

The fifth number is available for solution developers to version their additional extensions.

Figure 11-1 Versioning the Specification Files

Cartridge developers must maintain the integrity of the versioning by updating the version (specifically, the fifth number) if changes are made. While it is mandatory to increment the version when there are non-backward-compatible changes, it is recommended to increment the version for all changes to the specification.

About the "ANY" Schema Type

For schemas that refer to "Any" schema, a true "Any" type is not supported because the type of the object is unknown.

The convention TMF uses is that, with the schema object that refers Any, there must be an accompanying string to identify the type and this should point to a known type in the schema. This is more of a dynamic resolution of what will be passed.

Any is used in the canonical specification, for the characteristic value. The accompanying string to identify its type is valueType.

The following abstract shows characteristic schema from the canonical specification:

```

schema :
  Characteristic:
    required:
      - name
      - value
    type: object
    properties:
      name:
        type: string
        description: Name of the characteristic
      valueType:
        type: string
        description: Data type of the value of the characteristic
      value:
        $ref: '#/components/schemas/Any'
      '@baseType':
        type: string
        description: When sub-classing, this defines the super-class
      '@schemaLocation':

```

```

        type: string
        description: A URI to a JSON-Schema file that defines additional
attributes
        and relationships
        format: uri
        '@type':
        type: string
        description: When sub-classing, this defines the sub-class entity name
        description: Describes a given characteristic of an object or entity
through
        a name/value pair.

```

During the Data Structure Definition (DSD) generation, while importing the hosted specification for a schema which is of Any type, an additional element of `xmlData` type will be generated.

For characteristic, **valueXml**, which is not part of specification, will be generated:

- The **valueType** field identifies whether the value is of primitive type string (string, dateTime, date and so on), boolean, numeric or object type.
- If **valueType** is string, boolean, or numeric, the **value** field would have the value and the **valueXml** field would remain blank.
- If **valueType** is of object type, the value field would be blank and the **valueXml** field would have the objects.
- Either the **value** or **valueXml** field would be available based on the **valueType** field.

Examples of JSON to XML Conversions

The following example shows the JSON payload when **valueType** is string:

```

{
  "productCharacteristic": [{
    "@type": "Characteristic",
    "name": "Call Forwarding",
    "valueType": "string",
    "value": "Y"
  }]
}

```

The following example shows the XML payload when **valueType** is string:

```

<productCharacteristic xsi:type="Characteristic">
  <_type>Characteristic</_type>
  <name>Call Forwarding</name>
  <valueType>string</valueType>
  <value>Y</value>
</productCharacteristic>

```

The following example shows the JSON payload when **valueType** is object:

```

{
  "productCharacteristic": [{
    "@type": "Characteristic",
    "name": "BillingAccount",
    "valueType": "object",

```

```

        "value": {
            "@type": "BillingAccountRef",
            "id": "15"
        }
    }
}

```

The following example shows the XML payload when **valueType** is object:

```

<productCharacteristic xsi:type="Characteristic">
  <_type>Characteristic</_type>
  <name>BillingAccount</name>
  <valueType>object</valueType>
  <valueXml>
    <xmlData>
      <object xsi:type="BillingAccountRef">
        <_type>BillingAccountRef</_type>
        <id>15</id>
      </object>
    </xmlData>
  </valueXml>
</productCharacteristic>

```

The following example shows the JSON payload when **valueType** is boolean:

```

{
  "productCharacteristic": [{
    "@type": "Characteristic",
    "name": "AutoRenewal",
    "valueType": "boolean",
    "value": false
  }]
}

```

The following example shows the XML payload when **valueType** is boolean:

```

<productCharacteristic xsi:type="Characteristic">
  <_type>Characteristic</_type>
  <name>AutoRenewal</name>
  <valueType>boolean</valueType>
  <value>No</value>
</productCharacteristic>

```

The following example shows the JSON payload when **valueType** is numeric:

```

{
  "productCharacteristic": [{
    "@type": "Characteristic",
    "name": "Price",
    "valueType": "numeric",
    "value": 100
  }]
}

```

The following example shows the XML payload when **valueType** is numeric:

```
<productCharacteristic xsi:type="Characteristic">
  <_type>Characteristic</_type>
  <name>Price</name>
  <valueType>numeric</valueType>
  <value>100</value>
</productCharacteristic>
```

About anyOf, allOf, and oneOf

OpenAPI Specification (OAS) describes the various uses of the discriminator object in conjunction with the **allOf**, **anyOf**, and **oneOf** constructs. For details, see the documentation at: <https://spec.openapis.org/oas/v3.0.1#discriminator-object>.

OSM supports only **allOf**. The OSM schema extensions use **allOf** and a discriminator is added to the parent along with propertyName and mapping details. OSM extensions consistently specify the "@type" field as the discriminator propertyName.

In practice, this means that any JSON payload submitted to OSM Gateway when it is hosting an OSM hosted specification must include the @type element on schema objects that have been extended by OSM. You should continue this pattern when making schema extensions and should remember to populate any extensions with the appropriate @type value.

The following is an example of an OSM extension using **allOf**:

```
schema:
  CharacteristicOSM:
    description: OSM extension to include additional (optional) data on a
product characteristic
    allOf:
      - $ref: '#/components/schemas/Characteristic'
      - type: object
        properties:
          unitOfMeasure:
            type: string
            description: Unit of measure of the value associated with a
characteristic. Like MB,GB,Minutes,...
          actionCode:
            type: string
            description: Action taken on a characteristic in a MACD scenario
- is it a new value being introduced - existing indicates no change -
modifiedAttributs changes the value and delete would remove this
characteristic value from the asset
          enum:
            - new
            - existing
            - modified
            - delete
          previousValue:
            type: string
            description: Value of a characteristic in the original New
Product Order (before this Product Order makes a change to it)
          '@baseType':
            type: string
            description: When sub-classing, this defines the super-class
```

```

    '@schemaLocation':
      type: string
      description: A URI to a JSON-Schema file that defines additional
attributes
      and relationships
      format: uri
    '@type':
      type: string
      description: When sub-classing, this defines the sub-class entity
name

```

The following is an example of the parent schema with the discriminator object:

```

Characteristic:
  required:
    - name
    - value
  type: object
  discriminator:
    propertyName: '@type'
    mapping:
      Characteristic: '#/components/schemas/Characteristic'
      CharacteristicOSM: '#/components/schemas/CharacteristicOSM'

```

The following shows the correct portion of the order payload with the @type populated:

```

"productCharacteristic": [
  {
    "@type": "CharacteristicOSM",
    "name": "Authorization Code",
    "value": "AB1234CD"
  }
]

```

Including the discriminator object on the parent is optional when using **allOf**. However, it makes the schema extensions more readable.

About TMF Cartridges and Non-TMF Cartridges

A TMF Cartridge provides the most support to cartridge developers and system administrators when OSM operates on TMF orders. A TMF Cartridge is built around exactly one Hosted Order Specification, and provides the fulfillment logic for that order type (Product Order or Service Order). The TMF Cartridge is a solution cartridge, and consists of one or more component cartridges - all deployed as one unit.

Non-TMF cartridges are referred to as Freeform Cartridges. Freeform cartridges include all cartridges prior to 7.5.0 as well as non-TMF cartridges in 7.5.0. These cartridges are characterized by a fully open Design-time experience, allowing complete flexibility in all of OSM's capabilities - order structure, order state behaviour, eventing/notifications, orchestration, automation, and so on.

Table 11-1 Value for Cartridge Management Variable

Cartridge Type	Variable	Value
TMF Cartridge	OSM_RUNTIME_TYPE	MultiService
Non-TMF Cartridge	OSM_RUNTIME_TYPE	WLS

Note

It is important to note that these two cartridge types cannot be combined. If you have a Freeform cartridge, you cannot simply change the order type to TMF. You must start a TMF cartridge from scratch.

Once the TMF configuration is applied, the cartridge developer can then proceed with the standard cartridge development process, creating orchestration entities, processes, tasks, automation plugins, and so on. Building the cartridge results in Design Studio embedding the Hosted Order Specification into the cartridge par file along with all the other cartridge content.

TMF Cartridge Versioning

The version for auto-generated cartridges is specified during the import process. Once a solution is out of the development phase, if schema changes are introduced to the specification, both the specification and the generated OSM cartridge version should be incremented.

TMF Cartridge Target Version

The target version on all OSM cartridges must be set to 7.5.0.

About Importing the Hosted Order Specification

Access the TMF or OSM extended specifications from the OSM Cloud Native SDK download file, in the **TMFSchema** sub folder. It is strongly recommended to use the OSM extended specification either directly or as a base for custom schema extensions. An OpenAPI parser with support for version 3.0.1 must be able to parse this file once extensions are made. Otherwise, it would be rejected during the import process.

Refer to the *Service Catalog and Design Design Studio Modeling OSM Orchestration Online Help* for instructions about importing a Hosted Order Specification.

Design Studio parses the YAML file and validates it during the import, resulting in two cartridges when the import process is complete:

- An OSM cartridge containing some of the entities that are necessary for the TMF framework.
- A second cartridge would contain a set of Complex Data Types (CDT) created to match the structure and typing as per the schema for the Product Order or Service Order in the Hosted Order Specification.

Updating a Hosted Order Specification

Design Studio does not allow an update to the Hosted Order specification via a menu option. To remove an older version of the Hosted Specification, delete the existing cartridges and re-import the updated specification. You will need to re-apply any changes you have made to the lifecycle policy and fulfillment mode. For this reason, it is important to keep the generated

entities and cartridges free from custom additions as these would get removed during this process.

Deployment of a Hosted Order Specification

The specification is bundled with the cartridge par file and is the mechanism for getting it into the database. It is important to note that the specification lifecycle is independent of the cartridge lifecycle.

When a cartridge is undeployed, the specification is not removed and remains in the database. If an attempt is made to deploy a TMF cartridge (new deploy or re-deploy) that bundles a specification of the same version, but with content differences, then the cartridge deploy would fail as the specification with that version already exists. All specification changes should be accompanied by a version increase.

About Fulfillment Modes

When a Hosted Order specification is imported, one of the auto-generated entities is a fulfillment mode named **deliver**. This fulfillment mode is required for TMF cartridges and should not be renamed or deleted. It should be modified, however, to ensure that its namespace matches the other orchestration entities in the cartridge. Refer to the orchestration process for the correct namespace to be used.

Fulfillment modes for **cancel** are no longer needed for TMF orders. New fulfillment modes should only be added to support additional types of fulfillment such as TSQ.

For TMF orders, OSM will automatically set the fulfillment mode of the inbound order. The fulfillment mode will be defaulted to **delivery** unless an HTTP header is used to provide an alternate mode. Because of this new handling by the OSM core, the cartridge configuration does not need to provide an XQuery to determine the fulfillment mode value. In the Orchestration Sequence Editor, under the **Fulfillment Mode Expression**, the **XQuery** tab can specify the **Expression** radio button and provide **default** in the text area

```
(: Declare OSM name space :)
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/
model";

(: Declare incoming order name space:)
declare namespace tmf="http://oracle.communications.orchestration.com/tmf-api/
productOrderingManagement/{TMF622_VERSION}/productOrder/inputMessage";

declare variable $TMF_FULFILLMENT_MODE := "deliver";
declare variable $TMF_FULFILLMENT_NS := "mycompany.tmf.productorder";
let $fulfillmentModeCode := <osm:fulfillmentMode
name="{ $TMF_FULFILLMENT_MODE }" namespace="{ $TMF_FULFILLMENT_NS }"/>

return $fulfillmentModeCode
```

About TMF Order Lifecycle Policy

A default order lifecycle policy is generated that contains the transitions required for the TMF framework to function correctly. This entity may be moved to another cartridge to prevent cyclic dependencies when roles in other cartridges are referenced.

This policy should be modified to add additional transitions for your custom roles, but the existing configuration for "osm-gateway-internal-role" must be left in place.

Automation Users and Roles

Automation users should continue to be given permissions needed as they would in Freeform cartridges.

Fallout Resolution Users and Roles

A role should be created in Design Studio that can be granted the permissions necessary to perform fallout resolution actions. This workgroup should be distinct and separate from automation groups. (for example "FalloutResolutionRole"). This role should be given the required permissions in the order lifecycle policy to enable all fallout resolution actions.

Table 11-2 Required permissions for fallout resolution actions

State	Transition	Description
InProgress Amending Cancelling	Manage Order Fallout	This permission is required to enable the Retry Task option in the Task Web client (Fallout Exception Actions menu). Not supported for suspended state.
InProgress Suspended Amending Cancelling	Abort Order	This permission is required to enable the Abort Order option (Fallout Exception Actions menu) in the Task Web client.
InProgress Suspended Amending Cancelling	Fail Order	This permission is required to enable the Fail Order option (Fallout Exception Actions menu) in the Task Web client.

One fallout resolution action remains that is not permitted through the lifecycle policy. In order for users in the "FalloutResolutionRole" role to manually complete tasks (Fallout Resolution Actions menu in Task Web client), the following steps must be taken:

- The user performing the fallout action should be added to the same workgroup as the automation user. This enables the fallout user to see the correct content in the Task Web client.
- Any automation task that is capable of raising a fallout exception should have the "assigned" state available on the task editor state/status tab. This allows users that share the same workgroup, to re-assign a task to themselves.
- From the Task Web client, the fallout user should re-assign the task to themselves.
- Using the Task Web client Editor view, or by choosing the menu action "Manually Complete Task", the fallout user can manually complete the task.

About Data Dictionary

The import process creates a cartridge containing a set of Complex Data Types (CDT) that represent the schema of the productOrder or serviceOrder as defined in the specification. This saves time for cartridge developers who would otherwise have to manually construct the data model in Design Studio.

CDT content is always significant. The cartridge developer should be aware that a change to any content within the productOrder will trigger amendment processing, as all of the data is modeled as CDT.

About the Order Template

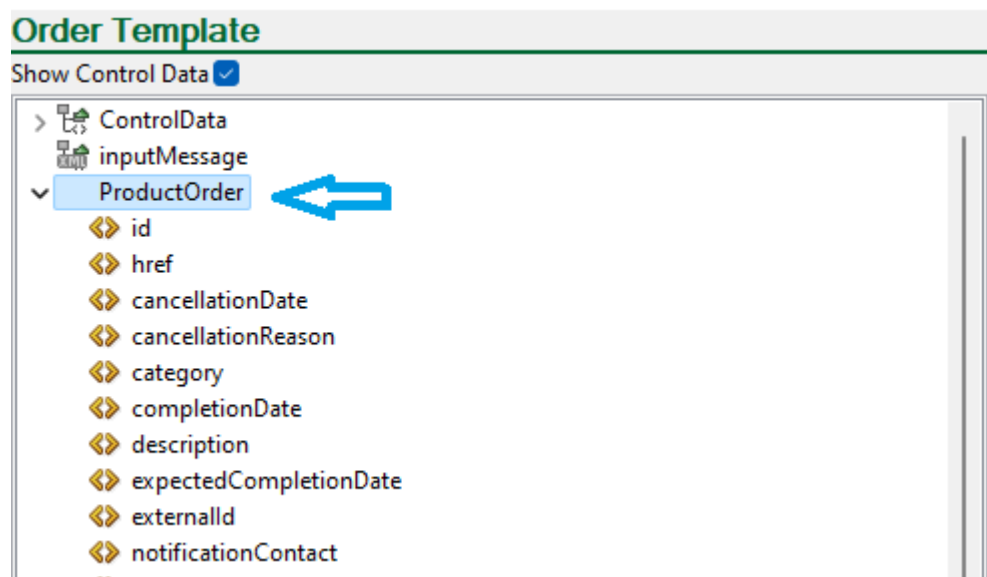
This section describes the following order templates.

- Master Order Template
- Order Item Specification Order Template

About the Master Order Template

To the master order template, you should add the CDT representing your order resource - either `ProductOrder` or `ServiceOrder` to the root (outside `ControlData`). It is important to note that even if there are extensions to the main resource (`ProductOrderOSM`), you should not add these sub-types into the order template. Because the extensions are sub-classed from the main resource and implemented in OSM as CDTs, the exact typing is handled at runtime.

Figure 11-2 Order Template with ProductOrder Information



The payload served to OSM Gateway by the calling system includes a '@type' field that declares the concrete schema type that is populated. The '@type' field should specify the exact schema extension used.

```
{
  "description": "TMF OSM Product Order",
  "category": "salesOrder",
  "externalId": "456855",
  "@type": "ProductOrderOSM_Create",
  .....
}
```

About the Order Item Specification Order Template

The order item specification has an order template that must also be configured properly which includes:

- An order item property must be created named either "ProductOrderItem" or "ServiceOrderItem"
- An XQuery sample for populating this structure follows.
- The CDT representing the order item type from the schema (ProductOrderItem or ServiceOrderItem) must be added to the order template of the order item specification.

① Note

Design Studio restrictions preventing the addition of a CDT to the order template have been lifted for cartridges with a target server version of 7.5.0.

```
(: Sample XQuery to populate the order item specification property -
ProductOrderItem. This XQuery does not retain the hierarchy of order items,
it only returns the incoming data for a single line item :)
declare namespace oms="http://www.metasolv.com/OMS/OrderModel/2002/06/25";
declare namespace tmfbase="%{TMF_CDT_NAMESPACE}";

(: Declare OSM name space :)
declare namespace model="http://xmlns.oracle.com/communications/
ordermanagement/model";
declare namespace tmf="http://oracle.communications.orchestration.com/tmf-api/
productOrderingManagement/%{TMF622_VERSION}/productOrder/inputMessage";

declare variable $tmfbase := "%{TMF_CDT_NAMESPACE}";

(: Ensure that incoming order has mentioned name space:)
declare variable $line := if (fn:exists( fn:root())/tmf:productOrderItem))
then fn:root()/tmf:productOrderItem else .;

declare function local:copyButTrimChildProductOrderItem(
    $parentName as xs:string?,
    $element as element(),
    $first as xs:boolean,
    $trim as xs:boolean) as element()*
{
    let $concreteType := fn:data($element/@xsi:type)
    let $elementName := local-name($element)
    return
        if (fn:exists($concreteType) ) then (
            let $qName :=
                if ($first = fn:true()) then (fn:QName($tmfbase,
"ProductOrderItem"))
                else (fn:QName($tmfbase, $elementName))
            return
                if($elementName = "productOrderItem" and $trim = fn:true())
then ( )
                else (
                    element { $qName }
                    {
                        attribute { "type" } { fn:concat("{", $tmfbase,
"}", $concreteType) },
                        attribute { "xsi:type" }
{ fn:concat(xs:string("tmfbase"),":", $concreteType) },
```

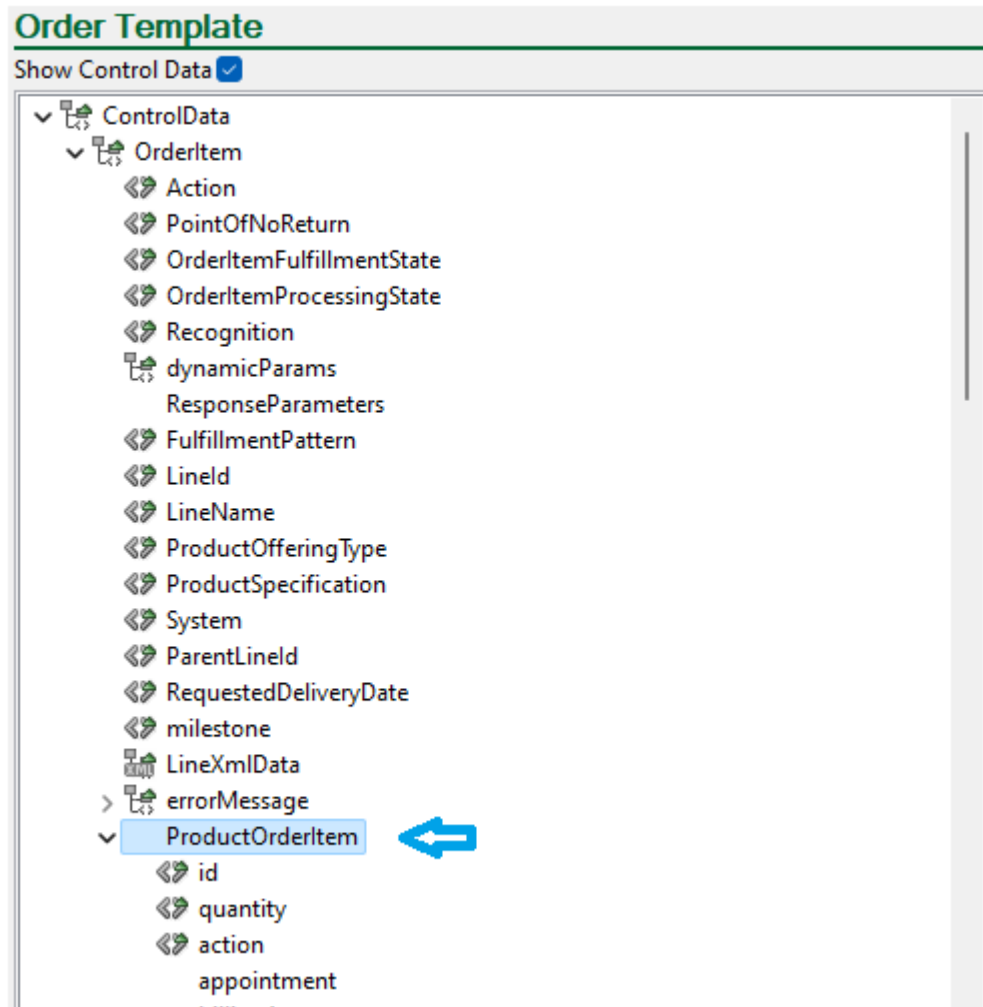
```

        attribute { "tmfbase:type" }
{ fn:concat(xs:string("tmfbase:"), $concreteType) },
    for $child in $element/node()
    return
        if ($child instance of element())
        then
local:copyButTrimChildProductOrderItem($concreteType, $child, fn:false(),
fn:true())
        else $child
    }
)
)
else (
    if($elementName = "productOrderItem" and $trim = fn:true()) then
()
    else (
        element {node-name($element)}
        {
            $element/@*,
            (
                for $child in $element/node()
                return
                    if ($child instance of element())
                    then
local:copyButTrimChildProductOrderItem((), $child, fn:false(), fn:true())
                    else $child
            )
        }
    )
)
);

local:copyButTrimChildProductOrderItem((), $line, fn:true(), fn:false())

```

Figure 11-3 Order Template with Product Order Item Information

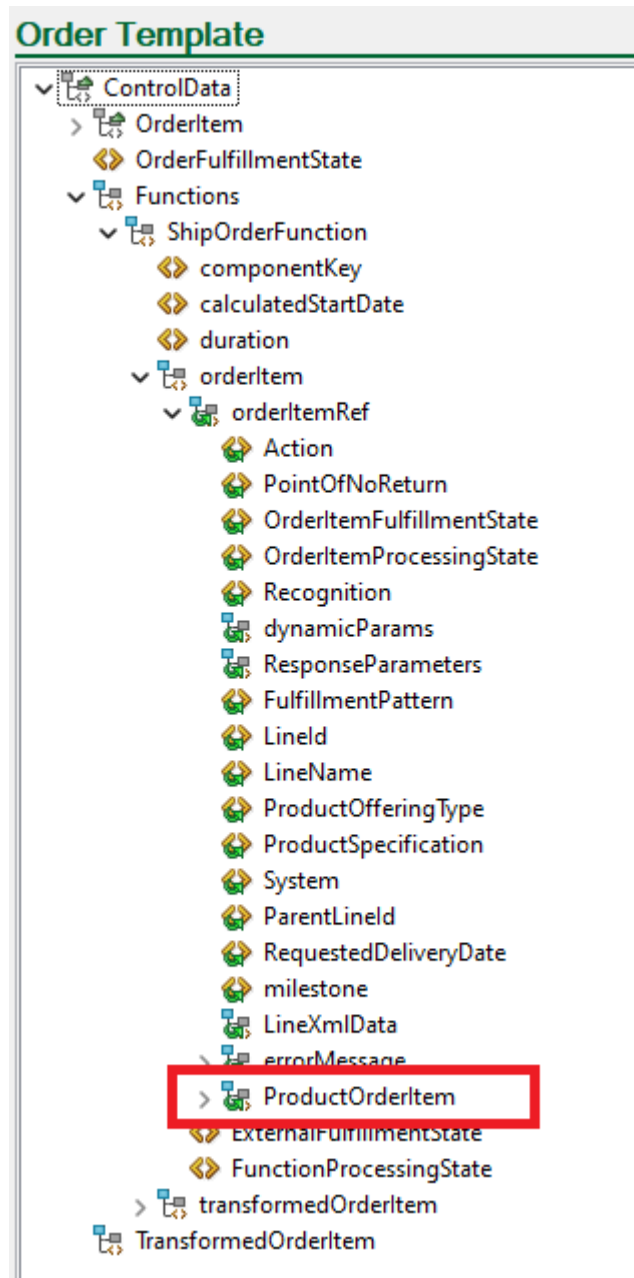


As is the case on the master order template, the CDT structure that should be placed under ControlData/OrderItem should be the base type from the TMF specification and not any extended types. The concrete schema will get resolved at runtime by specifying the '@type' property on the productOrderItem (or serviceOrderItem) on the incoming order payload.

```
"productOrderItem": [
  {
    "@type": "ProductOrderItem<extension>",
    "quantity": 1,
```

The TMF Order Item (ProductOrderItem or ServiceOrderItem) is exposed to fulfillment functions as a property of the orderItemRef within a function. This allows automation tasks associated to update the ProductOrderItem or ServiceOrderItem directly.

Figure 11-4 ProductOrderItem within a Function



The data returned when a GET endpoint is invoked consists of all POI or SOI under ControlData/OrderItem as well as the PO or SO at the root.

About the Significance of CDT

During amendment processing, OSM analyzes the changed data at an element level. Only changes to data elements that have been marked as significant will trigger amendment processing. In OSM, CDTs are always considered significant which means that for TMF orders, a change to any data field within the main resource (ProductOrder or ServiceOrder) will trigger compensation.

Cartridge developers that want more control over this process can configure a rule on the order lifecycle policy (the same functionality as Freeform cartridges) where a more thorough evaluation can be done. This can result in rejecting orders with changes that are deemed insignificant to the fulfillment logic of the cartridge.

About TMF Orders and Permissions

The Details tab for an Order has some required details related to TMF.

- The auto-generated lifecycle policy should be referenced.
- The TMF Order checkbox must be selected.
- The hosted order specification must be selected.

This configuration controls the inclusion of mandatory cartridge configuration that can be produced internally during the design studio build.

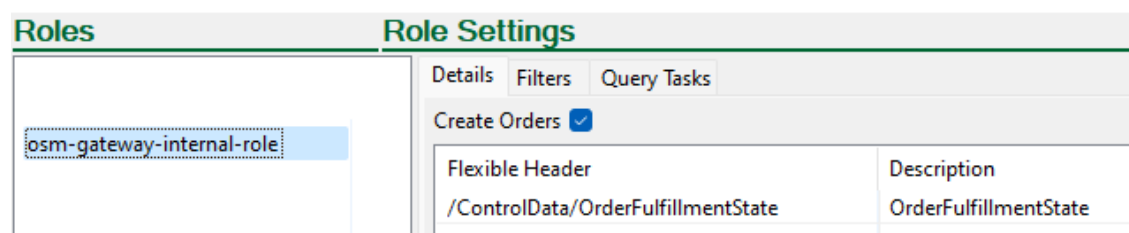
Once an order is designated to be a TMF order, the cartridge developer is responsible for adding the required permissions.

Permissions for Internal Gateway Role

The following are the permissions required for an internal gateway role:

- The osm-internal-gateway-user auto-generated during the import process must be added to the permissions tab of an order.
- On the details tab, Create Order access must be given.
- A flexible header for ControlData/OrderFulfillmentState must be added.
- A query task with the default settings must also be defined. This view does not need to follow a specific naming convention, but does need access to everything in the order template.

Figure 11-5 Internal Gateway Role Configuration



About Order Recognition

Recognition Rule

With freeform cartridges, clients send an XML create order request directly to OSM. Cartridge developers must understand the payload structure and the XML namespace associated with the order as this information is typically used to identify matching order requests.

When OSM hosts a TMF specification, OSM Gateway creates the actual request to OSM. While the payload structure is known (defined inside the TMF specification), the XML namespace carried on the order is crafted by OSM Gateway. As cartridge developers are still

responsible for coding order recognition, they must be made aware of the namespace that will be used. The incoming document namespace reflects the hosted specification name and version number. If you open the hosted specification editor in Design Studio, you can verify both of these values.

The prefix of the document namespace is a constant - **http://oracle.communications.orchestration.com/**

The suffix can be formatted from the following information:

hostedSpecName/<hostedSpecVersion/resourceName/inputMessage

Cartridge developers should use OSM best practices to capture version information as a model variable. As the hosted specification is up-versioned through its lifecycle (resulting in a version change), the model variable must be updated but not the xquery code.

The order recognition xquery provided below can be used in conjunction with a cartridge model variable that defines 'TMF622_VERSION' to match the hosted specification version.

Recognition

```
(: Declare TMF name space :)
declare namespace tmf="http://oracle.communications.orchestration.com/tmf-api/
productOrderingManagement/{TMF622_VERSION}/productOrder/inputMessage";

(: Ensure that incoming order is found and only one :)
fn:count(//tmf:productOrder)=1
```

Order Data Rule

This section provides information about model variable and sample xquery.

Model Variable

The auto-generated OSM cartridge contains an auto-generated model variable called TMF_CDT_NAMESPACE populated with *CDTCartridgeName/CDTCartridgeVersion*.

For example, if the cartridge was called "PO_Base" with version 1.1.1.1.1, then the variable value would have a value of "PO_Base/1.1.1.1.1"

This variable is necessary in many cartridge XQueries to reference the namespace of the OSM CDT entities correctly. In the XQuery sample mentioned below, we can see it is needed to set the namespace on the order data being created and returned to OSM.

XQuery

The following xquery sample will populate the TMF resource anchor root node (i.e / ProductOrder) in the order template from the TMF Order data submitted at runtime.

OrderDataRule

```
declare namespace oms="http://www.metasolv.com/OMS/OrderModel/2002/06/25";
declare namespace tmfbase="{TMF_CDT_NAMESPACE}";
(: Declare OSM name space :)
declare namespace model="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace tmf="http://oracle.communications.orchestration.com/tmf-api/
productOrderingManagement/{TMF622_VERSION}/productOrder/inputMessage";

declare variable $tmfbase := "{TMF_CDT_NAMESPACE}";

(: Ensure that incoming order has mentioned name space:)
```

```

declare variable $root := //tmf:productOrder;

declare function local:copyButTrimProductOrderItem(
    $element as element(),
    $first as xs:boolean) as element()*
{
    let $type := fn:data($element/@xsi:type)
    let $elementName := local-name($element)
    return
    if ($elementName != "productOrderItem") then (
        if (fn:exists($type) ) then (
            let $qName :=
                if ($first = fn:true()) then (fn:QName($tmfbase, "ProductOrder"))
                else (fn:QName($tmfbase, $elementName))
            let $concreteType :=
                if ($first = fn:true()) then (fn:substring-before($type, "_Create"))
                else ($type)
            return
            element {$qName}
            {
                attribute { "type" } { fn:concat("{", $tmfbase,
                "}", $concreteType) },
                attribute { "xsi:type" }
                { fn:concat(xs:string("tmfbase"),":", $concreteType) },
                attribute { "tmfbase:type" }
                { fn:concat(xs:string("tmfbase:"), $concreteType) },
                for $child in $element/node()
                return
                    if ($child instance of element())
                    then local:copyButTrimProductOrderItem($child, fn:false())
                    else $child
            }
        )
    )
    else (
        element {node-name($element)}
        {
            $element/.*,
            (
                for $child in $element/node()
                return
                    if ($child instance of element())
                    then local:copyButTrimProductOrderItem($child, fn:false())
                    else $child
            )
        }
    )
    else ()
};

let $description := $root/tmf:description/text()

return

    <_root>
        {local:copyButTrimProductOrderItem($root, fn:true())}
    </_root>

)

```

About Updating the TMF Order Item with Downstream Data

TMF Order data is held inside the following areas of the Order Template:

- `<resource> (_root/ProductOrder)`
- `ControlData/OrderItem/<resource>OrderItem (ControlData/OrderItem/ProductOrderItem)`

When OSM emits TMF events about the order resource, or when a GET endpoint is invoked, the data returned comes directly from the TMF data in these two locations. There are some considerations for cartridge developers to think about, with respect to where downstream data updates will be stored on the order item.

Is the data update modeled as an order item characteristic that is empty on the incoming create request, but gets populated based on downstream data and then propagated upstream (network address)?

OR is the data something fulfillment related, applying equally to all order items regardless of the product specification (shipment tracking details)?

These different types of data can be modeled in two ways.

Updates to Order Item Characteristics

Updates to a TMF Order Item (ProductOrderItem or ServiceOrderItem) characteristics trigger the automated `<resource>AttributeValueChangeEvent` with a pointer to the exact data change.

If a data update is destined for a line item characteristic, then there is a nuance about the OSM order data that must be considered.

OSM's parameter binding feature will dynamically map the data representing the line attributes on an incoming create order payload to the `dynamicParams` area of the control data.

For TMF orders, a CDT representing the entire order item is added to the control data. This means that the line item characteristics are held in two places in the OSM order data - `ControlData/OrderItem/ProductOrderItem` and the OSM mapped parameters inside `dynamicParams` as shown in [Figure 11-6](#).

Note that one location is OSM order data and the other is TMF data.

Figure 11-6 Order Item Characteristics

PO_QueryTask

☒ **Product Order Query Task**

```
<productOrder xmlns="http://oracle.communications.orchestration.com/tmf-api/productOrderingManagement/4.0.0.1.1/productOrder/inputMessage"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="ProductOrderSupremo_Create">
  <description>TMF Supremo Product Order</description>
```

ControlData

OrderItem List (1...999999999)

OrderItem

OrderItem

OrderItem

Field	Value
Action	add
OrderItemFulfillmentState	In Progress
OrderItemProcessingState	In Progress
FulfillmentPattern	Shipped_Service
ProductOfferin	

dynamicParams (SIMCardPSType)

Field	Value
ICCID	891004234814455936F
Phone Number	6505067000
Port In	N
IMSI	310170265624299
Authorization Code	AB1234CD
IMEI	35175605

ProductOrderItem (ProductOrderItemSupremo)

Id: OOX-1284VX Quantity: 1 Action: Add

BillingAccount (BillingAccountRefSupremo)

Id: OOX-1XRFTTW

Product (ProductRefOrValueSupremo)

Id: OOX-128QML Name: emulator

ProductCharacteristic (CharacteristicOSM) List (0...9999)

ProductCharacteristic (CharacteristicOSM)

Name	Value
Authorization Code	AB1234CD

ProductCharacteristic (CharacteristicOSM)

ProductCharacteristic (CharacteristicOSM)

Which location should cartridge developers update with their plugin script? The two characteristic areas should be kept in sync at all times, and to that end OSM uses the "Reverse Data Propagation" (RDP) feature that is responsible for this.

During PSR modeling, the specific data element that will be populated from downstream systems, should have the "Supports Reverse Propagation" checkbox selected. This should not be turned on for all data elements in a PSR entity, but only the ones where the value will originate downstream.

When checked, OSM will automatically propagate updates to that data element in the OrderItem/dynamicParams to the correct characteristic within the TMF Order Item.

Figure 11-7 Reverse Propagation Configuration

Product : SIMCardPS

Description: SIM Card Product

Data Elements	Details	Enumerations	Tags	Transformation	Usage	Notes
ICCID						
IMSI						
Phone_Number						
Phone_Number_Type						
Authorization_Code						
PortIn						
IMEI						

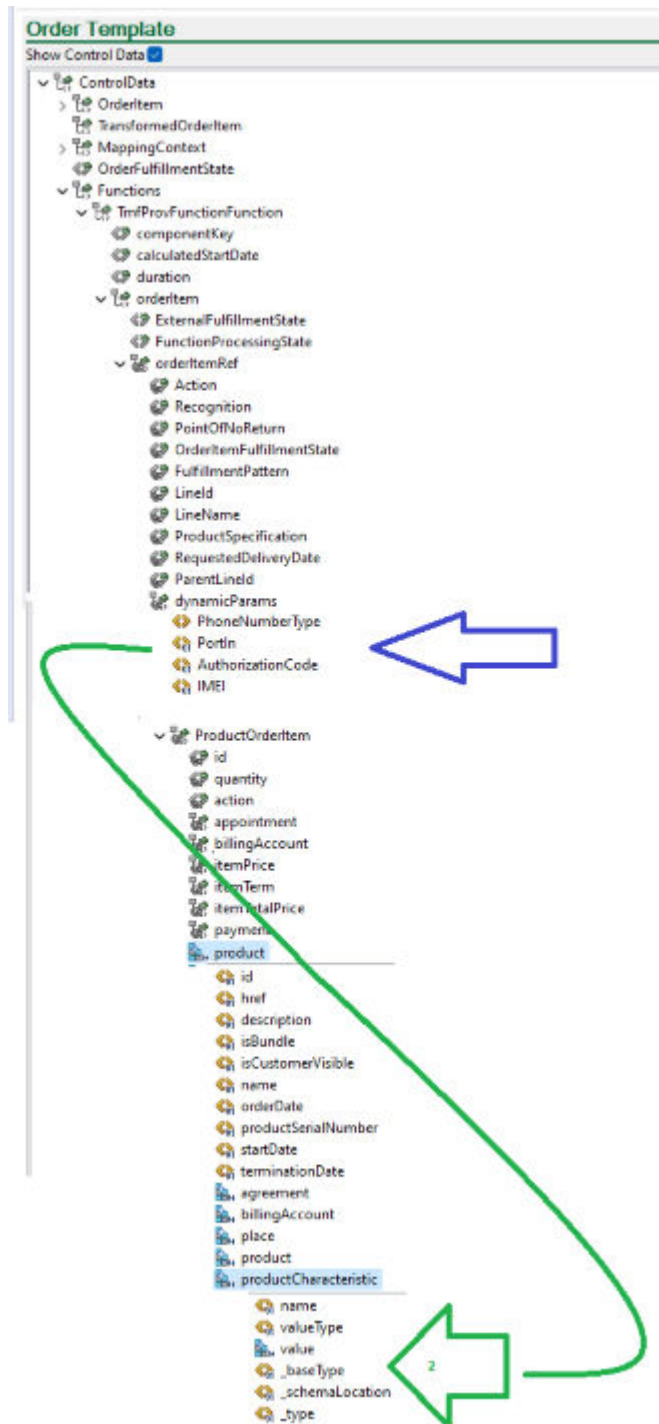
Propagation

Supports Reverse Propagation ☒

Supports Forward Propagation ☐

As shown in [Figure 11-8](#), updates made to data elements where the blue arrow is pointing, will be propagated (when RDP is turned on) to the area pointed to by the green arrow.

Figure 11-8 Data Propagation When RDP is Enabled



This feature is available only when the TMF Order Item Characteristic is a primitive type. Characteristics that are objects or arrays are not supported with RDP.

Note

If the incoming characteristic name is different than what is modeled in design studio, then you should be utilizing the "key" feature along with order item parameter bindings. In this way, the reverse propagation will take into account the key and will be able to find the correct source characteristic. Do NOT use the parameter binding xquery to do any such manipulation from one value to another as data will not be able to be propagated. As an example, if your upstream system will be sending a Characteristic name with underscores "Authorization_Code" but your PSR entity in Design Studio has a data element "AuthorizationCode", you should not use the parameter binding xquery to strip the underscore but rather you should use the "key" feature which is intended to rectify discrepancies between commercial and technical naming.

Transformed Order Items

For line items that are involved in OTM, updates to a TransformedOrderItem/dynamicParams can cause OTM to update an OrderItem/dynamicParams. When combined with the RDP for TMF orders, that will also include propagation to the original source TMF Order Item.

To update the TMF order item characteristic data, cartridge developers only need to focus on updating the dynamic parameter under the transformedOrderItem.

Updates to General Order Item Data

For updates that are more general in nature (ie. shipment tracking info), it does not make sense to model that as Characteristics of a product or service. This class of data would be added under the TMF order item.

General Order Item data such as these examples would be reflected in responses to a GET endpoint invocation, but do not trigger upstream events. In order for callers to be aware of the new data, an explicit GET invocation would be required.

Updates to External Fulfillment State

Updates to Functions/Function/orderItem/ExternalFulfillmentState to trigger cartridge defined fulfillment state calculation, may indirectly trigger an update to TMF OrderItem's state and TMF order state.

About TMF Order State

Calculating and propagating TMF order and order item state are handled internally by OSM.

Changes to the Order state are communicated through StateChangeEvent.

For order processing sequence diagrams showing StateChangeEvent, see "About TMF Orders (Cloud Native Only)" in *OSM Concepts* guide.

[Figure 11-9](#) shows the view of state transitions for a TMF Order (ProductOrder or ServiceOrder). This image includes OSM extensions to the state, meaning a TMF solution must have imported a specification that uses the OSM extensions to follow this diagram exactly.

Figure 11-9 TMF Order (ProductOrder or ServiceOrder) State Transitions

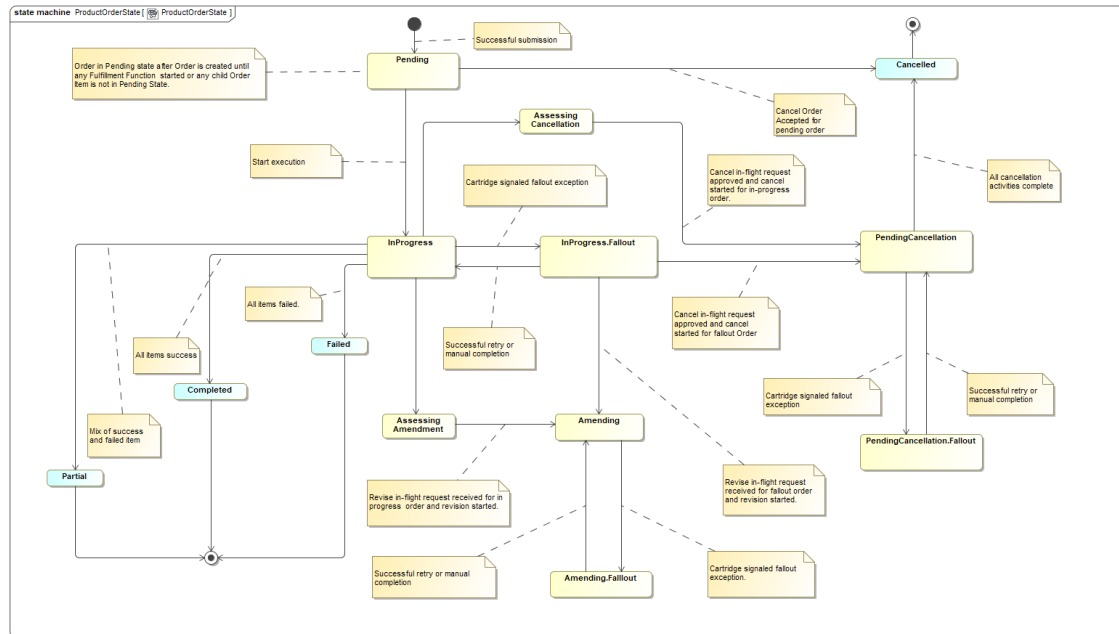


Table 11-4 summarizes the states for a TMF Order (ProductOrder or ServiceOrder).

Table 11-3 States of a TMF Order

TMF State	Usage	OSM Order State
acknowledged	Not used	NA
rejected	Not used Rejected requests are handled through a synchronous HTTP response with appropriate HTTP response code.	NA
pending	Initial State. After order is created, but before any task has started. Initial State for future dated orders.	Created
held	Not used	NA
inProgress	Order is executing.	InProgress
cancelled	Order has finished being cancelled.	Cancelled
completed	Order has finished execution with no failures.	Completed
failed	Order has finished execution and all order lines have failed.	Completed
partial	Order has finished execution with order lines having a mix of success and failure.	Completed
assessingCancellation	Not used.	InProgress

Table 11-3 (Cont.) States of a TMF Order

TMF State	Usage	OSM Order State
pendingCancellation	Order is executing a cancellation request.	Amending

[Table 11-4](#) summarizes the states for a TMF Order (ProductOrder or ServiceOrder).

Table 11-4 States of a TMF Order Extended by OSM

TMF State	Usage	OSM Order State
amending	Order is processing a revision request.	Amending
amending.suspended	Order was processing a revision request but is currently suspended.	Amending
amending.fallout	Order was processing a revision request but is now waiting on fallout exception resolution.	Amending
amending.fallout.suspended	A revision request was waiting on fallout exception resolution but is now suspended.	Amending
pendingCancellation.suspended	Order was executing a cancellation request but is now suspended.	Amending
pendingCancellation.fallout	Order was executing a cancellation request but is now waiting on fallout exception resolution.	Amending
pendingCancellation.fallout.suspended	A cancellation request was waiting on fallout exception resolution but is now suspended.	Amending
inProgress.fallout	Order is executing but is waiting on fallout exception resolution.	InProgress
inProgress.suspended	Order was executing but is currently suspended.	InProgress
inProgress.fallout.suspended	An order waiting on fallout exception resolution but is now suspended.	InProgress

Implications of TMF Partial and TMF Failed State

In all cases, the TMF Failed state and TMF Partial state are final states. This means that no further work is done on the entity that has entered this state. For example, if a line item enters TMF Failed state, no further processing can happen for that line item in the orchestration plan; if the order enters TMF Failed state, no further activity can occur on the order.

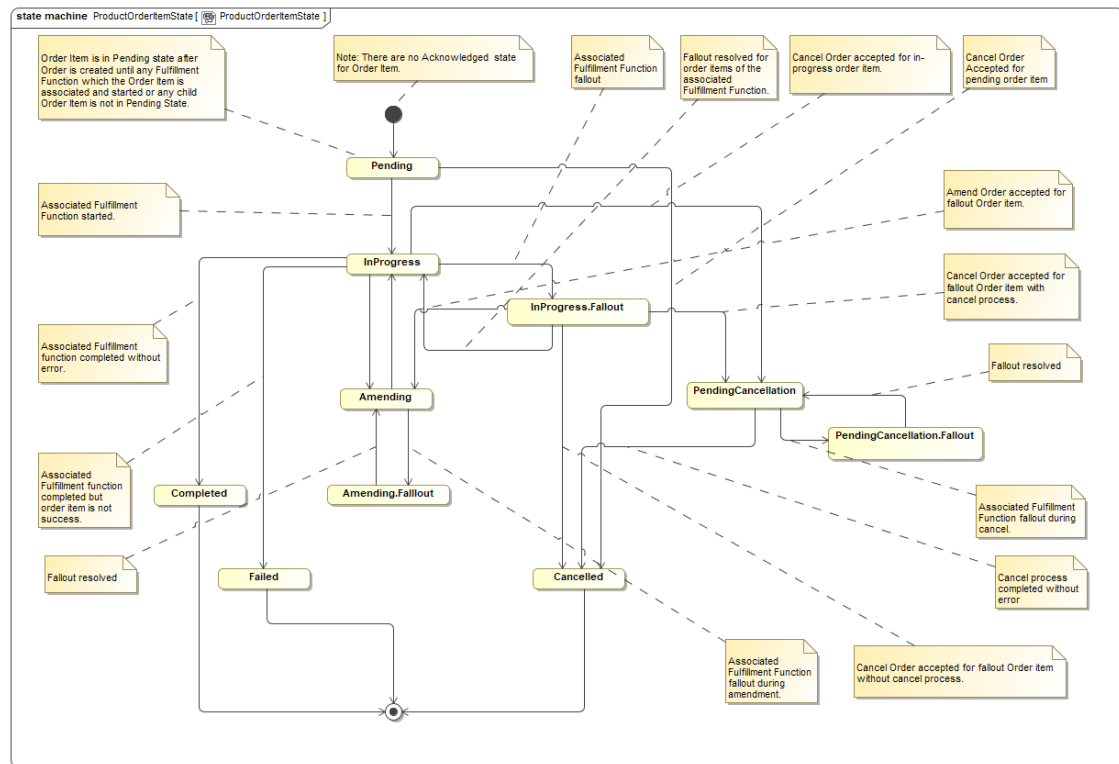
About TMF Order Item State

Changes to order item state are communicated through `AttributeValueChangeEvent`s.

[Figure 11-10](#) shows the state transitions for a TMF Order Item (ProductOrderItem or ServiceOrderItem). This image includes OSM extensions to the state, meaning a TMF solution

must have imported a specification that uses the OSM extensions to follow this diagram exactly.

Figure 11-10 TMF Order Item (ProductOrderItem or ServiceOrderItem) State Transitions



[Table 11-5](#) summarizes the state transitions for a TMF OrderItem (ProductOrderItem or ServiceOrderItem).

Table 11-5 State Transitions for a TMF OrderItem (ProductOrderItem or ServiceOrderItem)

From State/To State	Pending	InProgress	InProgress.fallout	pendingCancellation	pendingCancellation.fallout	amending	amending.fallout	failed	completed	canceled
pending	NA	Associated Fulfillment Function Started.	NA	NA	NA	NA	NA	NA	Pending Order Item is not associated to any Fulfillment Function and contains no child order item.	Cancel Order Accepted For Pending Order Item.
InProgress	NA	NA	Associated Fulfillment Function fallout	Cancel Order accepted for InProgress order item.	NA	Amend Order accepted for InProgress order item.	NA	Cartridge signals Order item is not successful.	Associated Fulfillment function completed without error.	NA
InProgress.fallout	NA	Order Item Fallout resolved and associated Fulfillment Function retry.	NA	Cancel Order accepted for fallout Order item with cancel activities	NA	Amend Order accepted for fallout Order item with amendment activities	NA	Cartridge signals Order item is not successful.	NA	Cancel Order accepted for fallout Order item without cancel activities.
pendingCancellation	NA	NA	NA	NA	Associated Fulfillment Function fallout during cancel.	NA	NA	Cartridge signals Order item is not successful.	NA	Cancel process completed without error.

Table 11-5 (Cont.) State Transitions for a TMF OrderItem (ProductOrderItem or ServiceOrderItem)

From State/To State	Pending	InProgress	InProgress.fallout	pendingCancellation	pendingCancellation.fallout	amending	amending.fallout	failed	completed	canceled
pendingCancellation.fallout	NA	NA	NA	Fallout resolved for cancelling order item and associated Fulfillment Function retry.	NA	NA	NA	Cartridge signals Order item is not successful.	NA	NA
amending	NA	Amendment completed and there are remaining fulfillment function to be executed.	NA	NA	NA	NA	Associated Fulfillment Function fallout during amendment.	Cartridge signals Order item is not successful.	NA	NA
amending.fallout	NA	NA	NA	NA	NA	Fallout resolved for amending order item and associated Fulfillment Function retry.	NA	Cartridge signals Order item is not successful.	NA	NA
failed	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
completed	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
canceled	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA

About Fulfillment State and Processing State

The ControlData/OrderItem/OrderItemFulfillmentState and ControlData/OrderItem/OrderItemProcessingState fields are now reserved for OSM use, for the calculation of TMF State. They will hold calculated TMF state values and will be communicated to upstream systems via the TMF StateChangeEvent.

Cartridges must not update these fields and should not use this field in fulfillment state mapping.

There are two level of Processing State for an Order Item:

- Order Component Order Item Processing State - Processing State of an Order Item within an Order Component. The input value of the state here can be optionally driven by external fulfillment state.
- Order Item Processing State - Processing State of an Order Item aggregated from child order item's processing state as well as all processing state of itself across all order component.

OSM Processing State value is different than the TMF product Order item state. [Table 11-6](#) describes the state mapping between OSM Order Item and TMF Product Order Item.

Table 11-6 OSM Order Item and TMF Product Order Item State Mapping

Processing State	OSM Order State	Processing Direction	Product Order Item State
NotStarted	NA	NA	pending
InProgress	in_progress	FORWARD	InProgress
InProgressWithWarnings	in_progress	FORWARD	InProgress
InProgressWithFailures	in_progress	FORWARD	inProgress.fallout
InProgress	amending	FORWARD	amending
InProgressWithWarnings	amending	FORWARD	amending
InProgressWithFailures	amending	FORWARD	amending.fallout
Completed	in_progress, amending	FORWARD	completed
CompletedWithWarnings	in_progress, amending	FORWARD	completed
PartiallyFailed	in_progress, amending, cancelling	FORWARD, REVERSE	partial
Undoing	amending	REVERSE	amending
UndoingWithWarnings	amending	REVERSE	amending
UndoingWithFailures	amending	REVERSE	amending.fallout
Undoing	cancelling	REVERSE	pendingCancellation
UndoingWithWarnings	cancelling	REVERSE	pendingCancellation
UndoingWithFailures	cancelling	REVERSE	pendingCancellation.fallout
UndoCompleted	cancelling	REVERSE	cancelled
UndoCompletedWithWarnings	cancelling	REVERSE	cancelled
UndoFailed	cancelling	REVERSE	failed

Modeling External REST Interactions using System Interaction (Cloud Native Only)

This chapter describes how to model external REST interactions using System Interaction.

Before reading this chapter, refer to "About REST APIs and System Interaction (Cloud Native Only)" in *OSM Concepts*.

Note

System Interaction is supported for OSM cloud native deployments only.

The System Interaction specifications can be TMForum REST APIs used in fulfillment (such as TMF700 for shipping or TMF 641 for provisioning), but can also be non-TMF APIs expressed as OpenAPI specifications. Parsing technology is used to validate that the structure and syntax of imported REST specifications align with the OpenAPI Initiative v3.0.1.

About Importing the OpenAPI Document into Design Studio

The OpenAPI document involved in a system interaction, is owned by and must be provided by the external application serving the API. The OpenAPI describes the specific capabilities of that system including available operations, HTTP headers, path parameters, HTTP response codes, schema, server url and so on. When the document is a TMF OpenAPI it is likely that extensions have been made by applications. Therefore, care should be taken that the file imported to Design Studio, reflects the current, up-to-date capabilities and schema supported by the external system.

Note that the OpenAPI version information is carried in two locations - at the **info:version** and as part of the **server:url**. Design Studio must be able to determine the version number unambiguously, therefore these numeric values must be an exact match before being imported to Design Studio.

TMF APIs for BSS/OSS System Interactions

For convenience, OSM cloud native makes available a set of TMF OpenAPI specifications that would typically be used for system interactions, as a reference. These are in the OSM SDK in the **TMFSchemas/ClientSpecifications** subfolder.

The samples provided are good for reference and may be used as a development accelerator. However, before actual integration is attempted, the System Interaction Specification should be updated to reflect the actual OpenAPI provided by the external system.

Importing a System Interaction

OSM Order Components have a System Interaction tab where cartridge developers can select the OpenAPI for a particular external system. Design Studio will perform parsing and validation to align with OpenAPI Initiative 3.0.1.

The system interaction tab of an order component, also provides a field to enter the Target System. This should be a logical name for the Target System and should describe the function of the Target System rather than its specific location. For example, a logical name of "Wireless-Activation" would be appropriate, as opposed to "Test-ASAP", as the latter pins it down to a specific system. This is important to allow flexibility between cartridge design and solution deployment design. The cartridge developer can freely reference a logical target system, leaving it up to the deployment scripts and configuration to allocate an actual target for that logical system in the form of a specific URL, authentication, and so on.

This logical name should be obtained from or provided to the individual responsible for maintaining the OSM CNTK deployment scripts. The same value must be defined in the CNTK project and instance specification files.

OSM does not support some aspects of OpenAPI schema for use in a System Interaction specification. For more information about these known issues and their workarounds, refer to ["Known Issues and Workarounds."](#)

Updating a System Interaction Specification

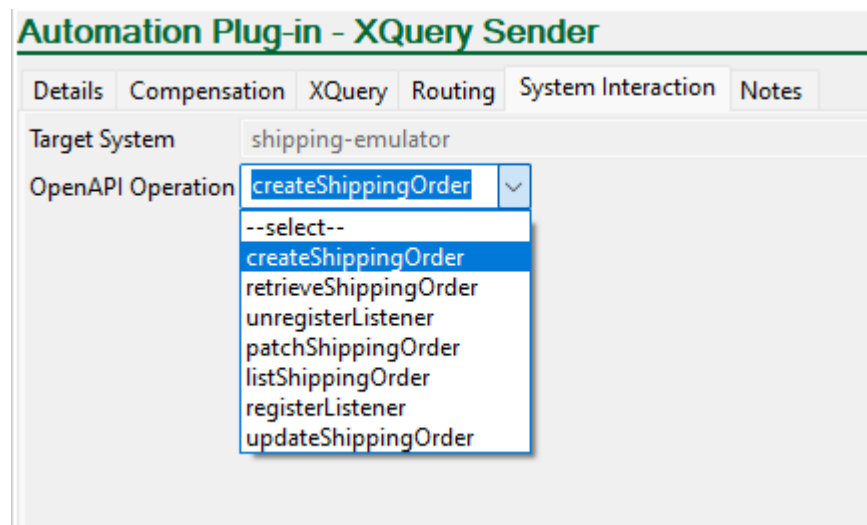
System Interaction Specifications are version specific. If an updated version of the OpenAPI is provided by an external system, the expectation is that the specification will reflect the version change. The Order Component holding the SI must have the old specification removed and the new one re-added. A delete and re-import will cause Design Studio to re-generate the necessary supporting metadata.

If an attempt is made to deploy a cartridge (new deploy or re-deploy) that bundles a System Interaction Specification of the same version, but with content differences then the cartridge deploy will fail as the specification with that version already exists. All specification changes should be accompanied by a version increase.

System Interaction and OSM Order Components

When a system interaction specification is imported to Design Studio, the specific operations and events become available as configuration for automation plugins as shown in the image below.

Figure 12-1 SI Operations Dropdown



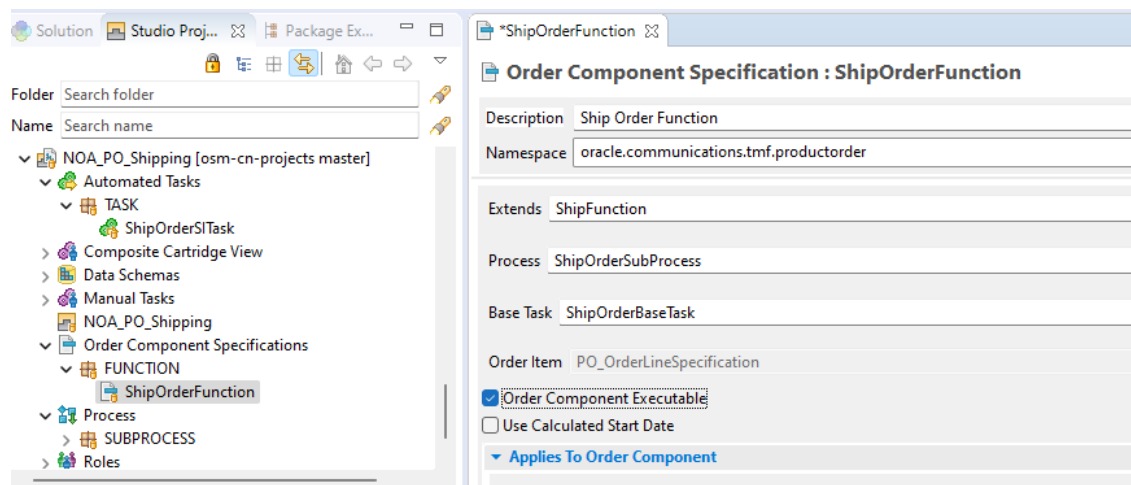
In order to populate this list, Design Studio looks at the OSM subprocess that holds this automation task, and then at the Order Component that contains this subprocess. The System Interaction registered with that order component provides the list of operations.

In the example below, the ShipOrderFunction holds the System Interaction for the Shipping System.

The **ShipOrderSubProcess** is defined on the **ShipOrderFunction**.

Therefore, all automation tasks found in the **ShipOrderSubProcess** will have a list of operations from the System Interaction Specification.

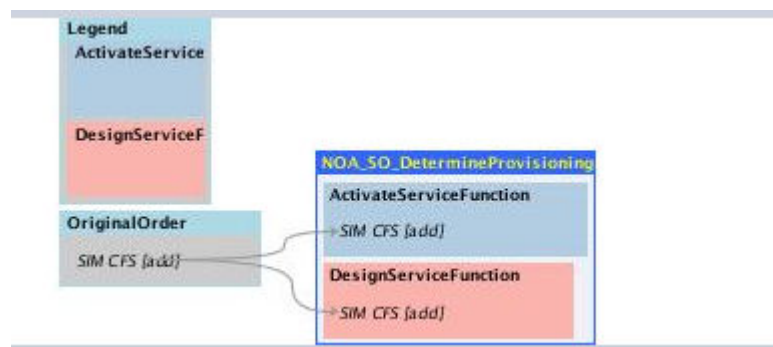
Figure 12-2 ShipOrderSubProcess



Determining the Order Component

OSM supports multiple layers of Order Components as part of the orchestration plan. Typically the layers in sequence are Function, System and Granularity. In case there is only one layer (which would be the Functional Order Component), apply the System Interaction to that component. In the example below, the functional order component - **ActivateServiceFunction** - has the System Interaction Specification.

Figure 12-3 activationOrchPlan

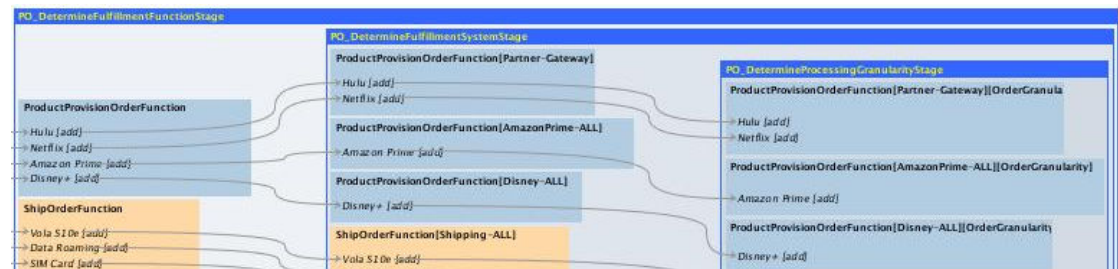


In case there are multiple layers of order components, apply the System Interaction to the earliest layer such that all subsequent layers (and their fan-out) satisfy these conditions:

- Same target system being contacted,
- With the same REST API version,
- And via the same subprocess.

In the next example, the system and granularity order components for the **ProductProvisioning** function (blue boxes) do not deviate from the above conditions, therefore the System Interaction can be imported to the functional order component - **ProductProvisionOrderFunction**.

Figure 12-4 digitaltv



In a final example below, the **Provisioning** function cannot hold the System Interaction for two reasons:

- There are multiple target systems.
- The systems use a different version of the TMF 641 specification.

In this case, the System Interaction Specification is imported to the "system" order component.

Figure 12-5 Emulated SOM

OSM_SOM

Emulated_SOM

Order Component Specification : Emulated_SOM

Description

SOM_Emulator

Namespace

oracle.communications.tmf.productorder

Specification File

serviceOrdering_v4.1.0.yaml

Version

4.1.0

Target System

som-emulator

Figure 12-6 OSM SOM

 **Order Component Specification : OSM_SOM**

Description	OSM_SOM
Namespace	oracle.communications.tmf.productorder
Specification File	serviceOrdering_v4.1.0.1.0.yaml
Version	4.1.0.1.0
Target System	osm-som

About Array of Arrays Support in System Interactions

In the system interaction, all the JSON messages received by OSM must be converted to XML before they can be processed by OSM at runtime. Additionally, all the XML messages from OSM must be converted to JSON before they are emitted to an external system.

The following shows the array of objects (technicalServiceClassChar) and array of arrays (alternativeSupportingResource) as defined in the OpenAPI specification:

```

TechnicalServiceBase:
  type: object
  properties:
    alternativeSupportingResource:
      type: array
      items:
        $ref: '#/components/schemas/MultipleResource'
    technicalServiceClassChar:
      type: array
      items:
        $ref: '#/components/schemas/Characteristic'

MultipleResource:
  type: array
  items:
    $ref: '#/components/schemas/Resource'

Resource:
  type: object
  properties:
    name:
      type: string

Characteristic:
  type: object
  properties:
    name:
      type: string
    valueType:
      type: String
    value:
      type: String
  
```

Array

For an array data in a JSON document, there is an XML node set equivalent of it. The array of type primitives or object data in a JSON document is mapped to the repeated XML element.

The following JSON document consists of an array of objects for the `technicalServiceClassChar`.

```
{
  "technicalServiceClassChar": [{"name": "char1", "value": "sample1"}, {"name": "char2",
"value": "sample2"}]
}
```

The sample XML for the above JSON document looks as follows:

```
<TechnicalServiceBase xmlns="http://oracle.communications.orchestration.com/tmf-api/
{apiName}/{apiVersion}/inputMessage"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="TechnicalServiceBase">

  <technicalServiceClassChar xsi:type="Characteristic">
    <name>char1</name>
    <value>sample1</value>
  </technicalServiceClassChar>
  <technicalServiceClassChar xsi:type="Characteristic">
    <name>char2</name>
    <value>sample2</value>
  </technicalServiceClassChar>
</TechnicalServiceBase>
```

Nested Arrays

The JSON document supports a multidimensional array but there is no equivalent of it in XML. To support an array of arrays, intermediate XML elements are generated that denote the beginning and ending of a nested array.

When receiving a JSON message containing array of arrays, these reserved elements, which are `nestedArray` and `nestedArrayItems` are added into the generated XML to denote the beginning and ending of a nested array and nested array items. When converting XML elements back into JSON, the inserted elements are converted into a JSON document with nested arrays.

The following JSON document consists of an array of arrays for resources.

```
{
  "technicalServiceClassChar": [{"name": "char1", "value": "sample1"}, {"name": "char2",
"value": "sample2"}]
  "alternativeSupportingResource": [
    [
      {
        "name": "Resource1"
      },
      {
        "name": "Resource2"
      }
    ]
  ]
}
```

The sample generated XML for the above JSON document looks as follows:

```
<TechnicalServiceBase xmlns="http://oracle.communications.orchestration.com/tmf-api/
{apiName}/{apiVersion}/inputMessage"
```

```

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="TechnicalServiceBase">
  <technicalServiceClassChar xsi:type="Characteristic">
    <name>char1</name>
    <value>sample1</value>
  </technicalServiceClassChar>
  <technicalServiceClassChar xsi:type="Characteristic">
    <name>char2</name>
    <value>sample2</value>
  </technicalServiceClassChar>
  <alternativeSupportingResource>
    <nestedArray>
      <nestedArrayItem xsi:type="Resource">
        <name>Resource1</name>
      </nestedArrayItem>
      <nestedArrayItem xsi:type="Resource">
        <name>Resource2</name>
      </nestedArrayItem>
    </nestedArray>
  </alternativeSupportingResource>
</TechnicalServiceBase>

```

About the OSM Gateway Functions

OSM Gateway performs the following functions for interactions with an external system using REST APIs:

- With JMS integrations, automation plugin code would be responsible for managing correlation. When REST interactions are modeled using System Interaction, then OSM Gateway becomes responsible for managing correlation. A condition of any integration using System Interaction, is that external systems must honor the **HTTP** header used by OSM for correlation. External system sync responses must echo back the **HTTP** header **X-Correlation-ID** that is sent on the request.
- In a REST exchange with an external system, OSM Gateway translates the XML payloads generated by automation plugins, into JSON payloads destined for the endpoint (and vice versa). Accordingly, System Interaction Specifications are restricted to JSON-based content types, specifically **application/json**, **application/merge-patch+json**, and **application/json-patch+json**.
- Schema validation on the payload content. Tuning parameters for schema validation can be found in the CNTK deployment scripts. This controls the Gateway behavior when unknown data is part of an incoming or outgoing payload.
- Resolves the logical target system name on the System Interaction against deployment artifacts to derive actual system connection details.
- If automation plugins have registered for event notifications then OSM Gateway listens for incoming events and passes them on to the external receiver automation plugin.
- Passes hard failures and unresolvable transient failures back to the automation plugin. New Automation APIs are available to emit fallout exceptions in the event that the order needs to be flagged in the Order Operations UI.

Support for JSON Patch and Merge-Patch Content Types in System Interaction

System Interactions support extended PATCH capabilities by supporting **application/merge-patch+json** and **application/json-patch+json** as valid request body content types, in addition

to **application/json**. This enables a standardized, JSON-based patching for precise and efficient updates when integrating with external systems.

- **Merge Patch (**application/merge-patch+json**)**: Partially updates a resource by merging provided fields. Omitted fields remain unchanged, and fields set to `null` are removed. This is ideal for straightforward updates where the payload structure matches the target resource.
- **JSON Patch (**application/json-patch+json**)**: Applies a sequence of operations (add, remove, replace, move, copy, test) to modify the resource. Each operation is run in order, offering fine-grained control for complex updates.

When constructing a PATCH request body in XQuery, you must adhere to the schema defined in the imported OpenAPI specification. The expected structure depends on the specified **requestBody** content type. This will either be **application/merge-patch+json** for Merge Patch operations or **application/json-patch+json** for JSON Patch sequences.

The OSM Gateway automatically sets the **Content-Type** HTTP header (**application/merge-patch+json** or **application/json-patch+json**) when calling PATCH endpoints on external systems. The header value is determined from the **requestBody** content type defined in the imported OpenAPI specification for the corresponding **operationId**.

The following examples illustrate typical usage of the two PATCH request body content types defined in the OpenAPI specification.

Merge Patch (**application/merge-patch+json**): **updateService**

```
paths:
  /service/{id}:
    patch:
      operationId: updateService
      summary: Update a Service using merge-patch
      description: >
        Updates the Service that matches the specified ID. Update can be
        performed using merge-patch.
      tags:
        - Service
      parameters:
        - required: true
          name: id
          in: path
          schema:
            type: string
          description: The ID of the service to update.
        - name: designOnly
          schema:
            type: boolean
          in: query
        - $ref: '#/components/parameters/fields'
        - $ref: '#/components/parameters/depth'
        - $ref: '#/components/parameters/expand'
      requestBody:
        description: >
          The service to be updated.....
        required: true
        content:
          application/merge-patch+json:
```

```

    schema:
      $ref: '#/components/schemas/Service'

```

The following sample shows the JSON payload for the request type mentioned above:

```

{
  "serviceCharacteristic": [
    {
      "name": "cpeBrand",
      "valueType": "ALPHANUMERIC",
      "value": "HUAWEI",
      "serviceSpecName": "DataCFS",
      "configItemPath": "DataCFS"
    },
    {
      "name": "serialNumber",
      "valueType": "ALPHANUMERIC",
      "value": "654654654",
      "serviceSpecName": "DataCFS",
      "configItemPath": "DataCFS"
    }
  ]
}

```

JSON Patch (**application/json-patch+json**): **patchService**

```

paths:
  /service/{id}:
    patch:
      operationId: patchService
      summary: Patch a service using JSON Patch
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: string
      requestBody:
        required: true
        content:
          application/json-patch+json:
            schema:
              $ref: '#/components/schemas/Service_Update'

```

The following sample shows the JSON payload for the request type mentioned above:

```

[
  {
    "op": "add",
    "path": "/config/cpeType",
    "value": "ONT"
  },
  {
    "op": "replace",

```

```

    "path": "/serviceType",
    "value": "premium"
  },
  {
    "op": "remove",
    "path": "/deprecated"
  }
]

```

Considerations for OSM Cloud Native to OSM Cloud Native Integration using System Interaction

When an OSM instance hosting a TMF specification interacts with another OSM instance that also hosts a TMF specification, there are some additional considerations for the cartridge developer. These are:

- You should take care to ensure OpenAPI version consistency between the two instances. The version of the downstream Hosted Specification should also be used for the upstream System Interaction specification. You cannot successfully integrate with two different versions.
- If a 622 or 641 System Interaction specification is imported, then Design Studio injects additional HTTP header content in anticipation of an **OSM/TMF** to **OSM/TMF** integration. If the external system is not an OSM instance, then this information is benign.

The following HTTP Headers are provided for integrations with OSM:

- **X-VERSION:** This is required for revisions. A **GET** call on the main resource (**GET/serviceOrder**) provides the caller with an HTTP header where X-VERSION reflects the version of the currently processing order. This value can then be incremented for any revision requests that are sent to downstream OSM.
- **X-Fulfillment-Mode:** If the TMF cartridge in the downstream OSM instance uses a non-standard fulfillment mode, then plugins should specify the fulfillment pattern name using this header. This is not required for standard **create** requests (**deliver**), cancellations (**cancel**) or amendments.

Developing Automation Plugins

For information about developing automation plugins, refer to "Using Automation with a System Interaction (Cloud Native Only)" in *OSM Developer's Guide*.

Known Issues and Workarounds

OSM does not support some aspects of OpenAPI schema for use in a System Interaction specification. This section describes these aspects and offers workarounds:

- **Inline nested schema objects:**

While OpenAPI schema allows a nested schema object to be fully defined inline with its usage, OSM requires this to be done by reference instead. The inline schema must be extracted as an independent element definition and the nested element should reference this definition.

- **Open-ended properties using additionalProperties field:**

OpenAPI schema allows for partial specification of child elements by adding the **additionalProperties** field. OSM requires all child elements to be enumerated in the specification. This can be done by extending the OpenAPI schema to include all the child elements of interest.

- **Default value for schema object in specification:**

OpenAPI schema can contain default values to use for schema objects if they are not present in the payload document. OSM does not honour these defaults. The payload originator must ensure such values are part of the generated payload document if they are required to be present.

Part III

Modeling Run-time Order Management

Part III contains the following chapters about modeling run-time functionality in an Oracle Communications Order and Service Management (OSM) solution:

- [Modeling Changes to Orders](#)
- [Modeling Fallout](#)
- [Modeling Fulfillment States and Processing States](#)
- [Modeling Jeopardy and Notifications](#)
- [Modeling Order Scheduling](#)

Modeling Changes to Orders

This chapter describes how to model change order management in an Oracle Communications Order and Service Management (OSM) solution.

About Amendment Processing and Compensation

To revise or cancel in-flight orders, OSM performs amendment processing. Amendment processing analyzes the requested changes, determines how to make the changes, and processes them. Amendment processing functions as follows:

1. A base order is submitted and is currently processing; it is *not* in the Not Started, Completed, or Aborted state. The upstream system submits a revision or a cancel order. The new version of the order includes all of the data relevant to the order, not just changed data. The upstream system does not need to identify the changes to OSM or explicitly provide the discrepancies; OSM determines the discrepancies during amendment processing by comparing the new version with the version of the order currently being processed.

To submit the revision order, the upstream system can use either the CreateOrder web service operation or the CreateOrderBySpecification web service operation.

The new version of the order can:

- Change existing data
- Remove existing data
- Add new data

Note

You can create revision orders by using the Task web client. This is typically used only for testing or for low-volume order processing.

2. OSM receives the revision order. OSM checks to see if the base order is amendable. You enable amendment processing on the order specification. If the base order is not amendable, the order is not a revision order.

Note

When you model orders, make sure that orders that are expected to be amended are configured to be amendable. If not, an order that is sent as a revision order is instead processed as a new order. This can cause errors during fulfillment because there are two orders fulfilling the same services for the same customer.

3. OSM checks in-flight orders for a matching value to an order key. For example, you can specify to use the sales order number as the order key. In that case, when OSM processes an order, it looks for an in-flight order that has the same sales order number. If OSM finds an in-flight order with a matching sales order number, OSM treats the new incoming

customer order as a revision on the existing order. See "[About Order Keys](#)" for more information.

OSM now has two orders to work with: the revision order and the base order.

Note

Many types of orders do not require an orchestration plan; for example, some service orders are created specifically for a simple service provisioning task and therefore require no dependencies.

4. OSM performs further checks on the base order to determine if the order is allowed to be amended. OSM does the following:
 - OSM checks to see if the base order is in a state that can be amended. Orders in the Not Started, Completed, or Aborted state cannot be amended. You can customize the allowed transitions to the amending order state by configuring the order life-cycle policy. See "[Modeling Order Life-Cycle Policy States and Transitions](#)" for more information.
 - OSM ensures that the base order has not passed the point of no return (PONR). The PONR is the point in the processing of an order item after which order amendments are either impossible or incur some penalty. In this case, a revision order might not be possible. See "[Fulfillment Pattern Point of No Return](#)" for information.
 - OSM checks to see if the incoming customer order has a version identifier. If OSM has a version identifier, OSM compares the value of the version to the version of the in-flight order. If the version of the in-flight order is greater than the version of the incoming customer order, the incoming revision is ignored.

If a revision cannot be processed, or if the order life-cycle has not allowed the revision, the revision order is set to the Failed state, and the base order continues to be processed.

5. OSM determines whether amendment processing is needed by analyzing order data at the following levels:
 - OSM compares the revision order data and the base order data (or the revision order data and the last submitted revision order data) to see if a compensation is needed. (Compensation defines the actions that need to be taken to perform amendment processing; for example, undo and redo.) See "[About Order-Level and Task-Level Compensation Analysis](#)" for more information.
 - During compensation, OSM compares task data for each task in the order process to further validate the compensation requirements. See "[About Order-Level and Task-Level Compensation Analysis](#)" for more information.
 - OSM uses the significance of the data to determine if compensation is needed at both stages. Data significance enables you to optimize amendment processing in a way that compensation is considered only for changes to data that is marked as significant. Data that is not marked significant is updated but does not get included in the compensation plan if its value is changed. See "[About Data Significance](#)" for more information.

Note

If an amendment is received while a task is in a fallout execution mode, OSM does the following:

- If the task is not configured to be compensated if it is in progress, the execution mode of the task does not change as a result of the amendment order.
- If the task is configured to be compensated if it is in progress, and the amendment contains changes to significant data:
 - * If the task is still needed after the changes to the order from the amendment are considered, it transitions automatically to (normal) Redo mode.
 - * If the task is no longer needed after the changes to the order from the amendment are considered, it transitions automatically to (normal) Undo mode.

In both of these cases, ensure that your automation code (for the Redo or Undo execution mode) contains a check to see if the task has been in a fallout execution mode, and also whatever code is needed to resolve any actions that have been taken in the fallout execution mode. For example, if your automation for Do in Fallout mode opens a trouble ticket, your Redo automation should check to see whether it needs to close a trouble ticket.

- If the amendment order contains no changes to significant data, the execution mode of the task does not change as a result of the amendment order.

6. After determining that amendment processing is needed, OSM transitions the order to the Amending state.

Note

OSM queues orders that need amending. Therefore it is possible for multiple revisions of the same order to exist in the queue. If amending the order is allowed, OSM chooses the latest version of the amendments in the queue by comparing the optional version identifiers (if configured) or, if there is no configured version identifier, by comparing the dates and times that the amendments were received.

In the Process Amendment state, OSM determines the compensation required. For example, OSM might redo a task with different values for one or more data elements on the task data that were used for input into the task.

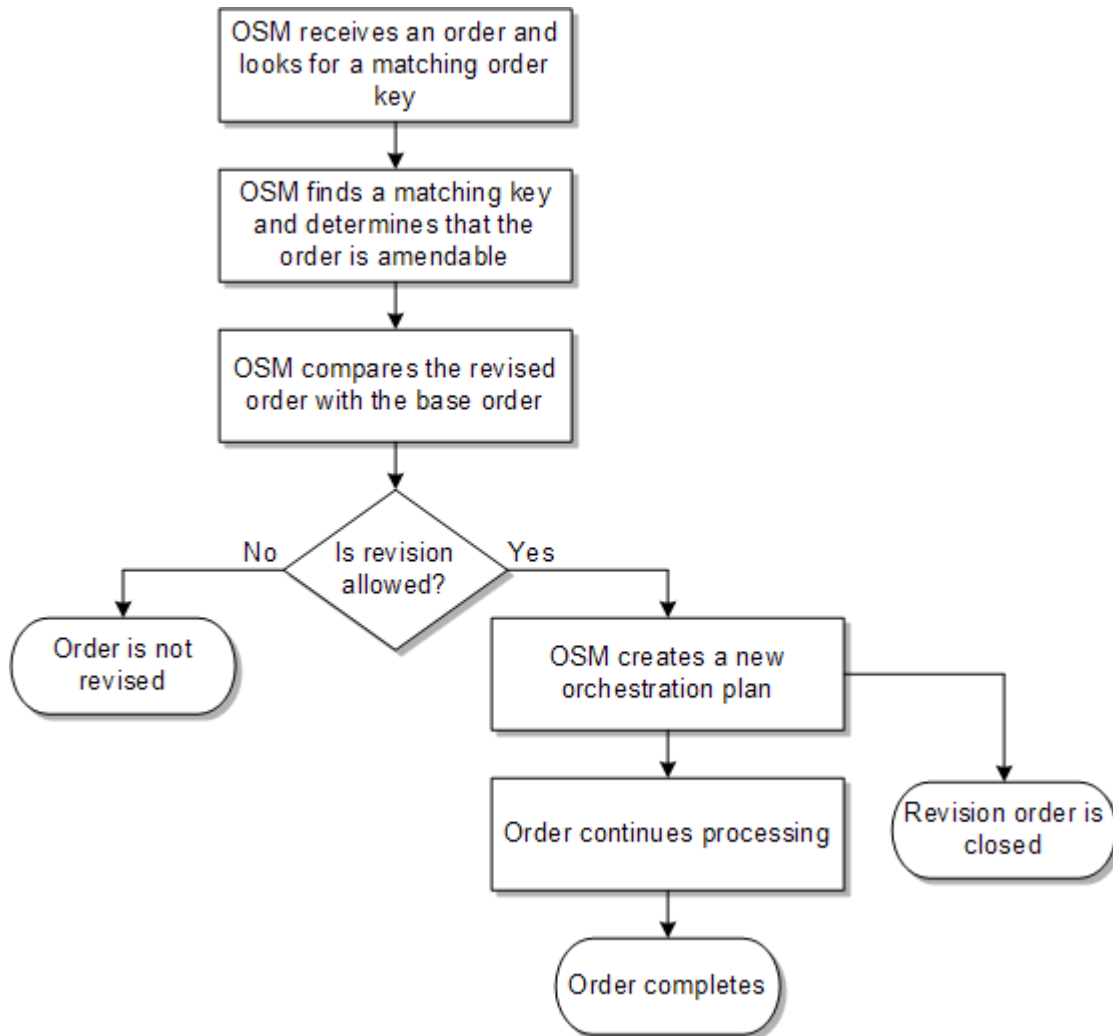
For process-based orders, the tasks are analyzed to find the impact of the changes. That impact determines the compensation plan. For example, OSM might need to redo a task with different data values or undo a task if it is no longer required. The data comparison is based on the data in the creation task of the base order and the revision order. See *OSM Concepts* for information.

For orchestration orders, the order components of the orchestration plan are analyzed to determine which order components need to be redone, undone, or done for the first time (amend do). The tasks of the sub-processes run for each of those order components to be compensated are also analyzed.

7. OSM handles the base order and the revision order as follows:
 - For the base order, OSM creates a new orchestration plan that includes the order components and their dependencies. Any order components with data that has changed as a result of the revision are redone. Any order components that have been processed but are no longer required in the revision are undone in reverse dependency sequence. Any order components that are inserted as new requirements are fulfilled. The order state is set to Amending.
 - For the revision order, OSM transitions it to the Completed state because its only purpose was to revise the base order.
8. OSM processes the changes according to the compensation plan it calculated and re-calculates the compensation plan needed after every change. OSM performs the necessary undo, redo, and amend do operations on order components (for orchestration orders) and on tasks (for both orchestration orders and process-based orders).

[Figure 13-1](#) shows a simplified amendment processing flow.

Figure 13-1 Amendment Processing



Note

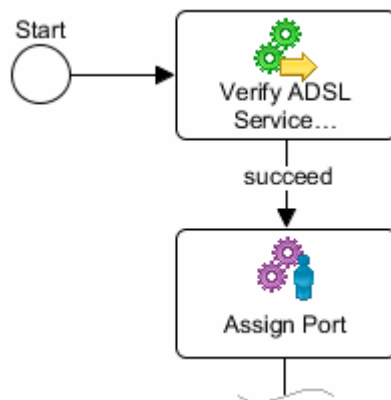
Messages from external systems can be returned to a task for which the receiver is temporarily unable to receive a response. This can happen, for example, if an order is being amended or is suspended. When this happens, OSM saves the returned message to the database, to wait until the order is ready for the task. This message will be removed when the message is resent to the receiver or when it becomes irrelevant (for example, because the order has been purged). This functionality is on by default, but you can turn it off, for example if your solution already handles messages of this type in a different way. To turn this feature off, use the **oracle.communications.ordermanagement.AutomationResponseMessageParkingEnabled** parameter in the **oms-config.xml** file. See *OSM System Administrator's Guide* for more information about this parameter and about the **oms-config.xml** file.

A simple example of a revision order is as follows:

1. A customer orders a DSL service at 3 MBps. An order is created and sent to OSM.

[Figure 13-2](#) shows the start of the process. In this example, the process begins with the **Verify_ADSL_Service** task and then transitions to the **Assign_Port** task.

Figure 13-2 Amendment Order Example



2. OSM verifies that the 3 MBps service is available and transitions to the next task, **Assign_Port**.
3. While the order is waiting for port assignment, the customer calls back and asks a customer service representative (CSR) to change the order to 5 MBps. The CSR creates a revision order in the CRM system with the revised bandwidth value of 5 MBps and submits the order to OSM.
4. OSM receives the incoming customer order, and detects that it is a revision to an in-flight order.
5. OSM accepts the revision order, calculates the compensation plan, and begins to run it. OSM knows that compensation is necessary because the data (bandwidth) that was on the order as input data when this task ran previously has now changed. The revision order requests that the **Verify_ADSL_Service_Availability** task must be redone to ensure that the 5 MBps service is available.
6. The value set by the **Verify_ADSL_Service_Availability** task is changed.

[Figure 13-3](#) shows the order displayed in the Task web client. In this figure, the **Verify_ADSL_Service_Availability** task has an execution mode of Redo. Because the

port has not been assigned yet, the **Assign Port** task has an execution mode of Do, but it cannot be worked on until the order completes compensation for the revision.

The task execution mode can be Undo, Redo, Do, Amend Do, Undo in Fallout, Redo in Fallout, Do in Fallout, and Amend Do in Fallout. (See "[About Task Execution Modes](#)" for more information.)

Figure 13-3 Amendment Displayed in the Task Web Client

Order ID	Order State	Type	Task	Execution Mode	State
412	Amending	Add ADSL Siebel	Assign Port	Do	Received
412	Amending	Add ADSL Siebel	Verify ADSL Service Availability	Redo	Received

- The revision order transitions directly to the Completed state. This is because the revision order is used only for updating the base order. For revision tracking, OSM maintains a record of the revision order as part of the order history.
- After verifying that the revised bandwidth is available, the base order continues processing.

You can monitor revisions in the web clients. [Figure 13-4](#) shows a revision order (Order 7) and [Figure 13-5](#) shows the base order that it revised (Order 6).

Figure 13-4 Revision Order in the Order Management Web Client

Order ID 7: Order Details

Order Order Items Order Components

Summary Data Orchestration Plan Amendments

General

Order ID	7	Expected Order Completion Date	06/02/2015 08:25:25 AM
Reference	Order1433258622976	Requested Order Delivery Date	12/31/2001 12:00:00 PM
Version		Expected Order Start Date	06/02/2015 08:24:25 AM
Priority	5	Expected Order Duration	Amends 6
Order State	Completed		
Target Order State	No State		
Order Creation Date	06/02/2015 08:24:25 AM		

Figure 13-5 Amended Order in the Order Management Web Client

Order ID 6: Order Details Actions ▾

Order Order Items Order Components

Summary Data Orchestration Plan Amendments

General

Order ID	6	Expected Order Completion Date	06/02/2015 08:24:43 AM
Reference	Order1433258622976	Requested Order Delivery Date	12/31/2001 12:00:00 PM
Version	0 [6/2/2015 8:24 AM]	Expected Order Start Date	06/02/2015 08:23:43 AM
Priority	5	Expected Order Duration	1m 0s
Order State	Amending	Amended By	7 [Total 1]
Target Order State	No State		
Order Creation Date	06/02/2015 08:23:43 AM		

About Revising or Canceling Orders by Using the Task Web Client

In most cases, revision orders are submitted from an order-source system. You can also revise and cancel orders by using the OSM Task web client; for example, by using the **Amend Order** menu command. This is useful for testing revisions and cancellations within OSM, however, this method is not appropriate for production systems.

You should use the Task web client to submit amendments only when the order was submitted from the Task web client originally or when the upstream system cannot submit an amendment. If the upstream system submits an amendment after you manually submit an amendment, data synchronization errors can occur.

When you use the Task web client to amend an order:

1. OSM creates another order, with a new order ID number, for the revision. The new order includes all of the creation task data from the in-flight order.
2. The Task web client displays the revision order.
3. You can then change the data required for the revision and submit the revision order.

⚠ Caution

If you use revision versioning, increment the revision version.

About Order Keys

When receiving an order flagged as amendable, OSM checks in-flight orders for a matching value in an order key. (You configure the order key when you model the order specification.) For example, you can specify to use the sales order number as the order key. In that case,

when OSM processes an order, it looks for an existing order that has the same sales order number and amends that order.

✓ **Tip**

Because OSM must check the order key for all in-flight amendable orders, you should make orders amendable only if they might need to be amended. That way, OSM does not need to check for an order key for orders that would not be amended.

You define the order key in the order specification as one or more XPath expressions that reference one or more data elements in the incoming customer order. If you use multiple data elements, the values are concatenated in the order key.

OSM generates an order key when the order is created. To assign an order key:

- The order key data elements must be part of the creation task data.
- The order key must be identical between the base order and the revision orders and must not change.
- The order must be flagged as amendable.

Order key values should not be modified after an order is submitted. For more information about creating valid order keys, see, "[Modeling Valid Data Keys](#)."

About Submitting Multiple Revisions of an Order

In some cases, multiple revisions to a single order are submitted. Each revision is expected to be a new revision of the in-flight order, not a cumulative comparison of previous revisions. The latest amendment is assumed to be the most complete revision containing all of the changes from earlier revisions. Intermediate revisions are not processed by OSM.

You can use versioning in the revision orders to recognize the order of the revisions as OSM receives them. For example:

- If revisions are received out of sequence, OSM ensures that the latest revision is used. If a revision is received while a current revision on the same order is being compensated, and if processing of revising in-flight revision orders is enabled (see "[About Revising In-flight Revision Orders](#)" for more details), OSM initiates the termination of the current revision and changes the compensation state of the current revision to Terminating and queues the latest revision. After the current revision reaches a safe point, OSM terminates the current revision and starts processing the latest revision. If processing of revising in-flight revision orders is not enabled, OSM completes the compensation for the current revision before processing the latest version. If a version is received that is earlier than the current revision being processed, the earlier version is ignored.
- If several revisions are received, OSM discards interim revisions and applies the latest revision because it represents the latest customer instructions for the order and is a complete copy of the base order.

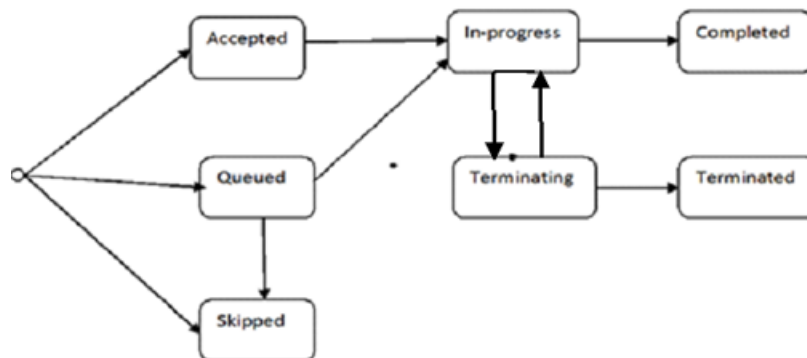
To configure revision versioning, you specify a data element on the incoming customer order that OSM checks when processing revisions for the order. You specify the data element as an XPath expression in the order specification **Amendable** tab. For example, if the data element is **<version>**, the XPath expression is:

```
_root/version
```

About Compensation States

The following diagram shows compensation state transitions and the life-cycle of a revision order.

Figure 13-6 Compensation State Transitions



- **Accepted:** This state indicates that the revision order is accepted by OSM. OSM evaluates order life cycle policy when the revision order is received before accepting it. This state is a transitional state until the revision processing on the base order starts.
- **In progress:** This state indicates that the revision processing has started on the order.
- **Completed:** This state indicates that the revision processing is complete.
- **Queued:** This state indicates that the revision order is queued.
 - OSM queues the revision on a base order that is in the In-progress state if the Process Amendment transaction in the order state policy is not enabled. If the Process Amendment transaction is enabled, OSM re-evaluates the condition on order data changes and dequeues the revision. See "[Disabling Processing of Revisions on In-flight Revision Orders](#)" for more details.
 - OSM queues the revision order when the base order is in the Amending state. Revision order remains queued until either the current revision processing is terminated (default configuration) or revision processing is completed.
- **Skipped:** This state indicates that the revision order is skipped. This happens on a queued revision, when it is replaced by a new revision.
- **Terminating:** This state indicates a transition period before OSM starts processing the latest revision on the order. During this period, OSM provides support to clean up all Started compensation tasks by ensuring they reach a known state.
- **Terminated:** This state indicates that the compensation is terminated safely.

About Revising In-flight Revision Orders

OSM can process a revision order while it is still processing a revision on the same order that it received earlier, without having to wait for the ongoing revision order to complete. When a revision on an in-flight revision order is received, OSM initiates the termination of the current

revision and changes the compensation state of the current revision to Terminating and queues the latest revision. After the current revision reaches a safe point, OSM terminates the current revision and then starts processing the latest revision. The compensation processing of the current revision transitions to a new revision safely, at the earliest, instead of waiting for the completion of the current revision processing. With this functionality, OSM can process order changes quickly, while reducing the operational expenses by optimizing the work needed to be done for subsequent order changes, and carries forward pending tasks that were not run in the previous revisions to the latest revision.

Note

By default, processing of revisions on in-flight revision orders is enabled for cartridges with target version 7.4.0.0.0. For information on enabling and disabling processing of revisions on in-flight revision orders, see "[Disabling Processing of Revisions on In-flight Revision Orders](#)".

OSM processes a revision on an ongoing revision order as follows:

- While amendment processing is still in progress for the revision order, OSM receives another revision on the order.

OSM initiates the termination of the current revision and changes the compensation state of the current revision to Terminating and the latest revision is queued. Once the current revision reaches a safe point, the current revision is terminated. See "[About Terminating Compensation](#)" for details about what happens in the Terminating state and when OSM terminates in-flight revision processing.

- OSM merges the compensation plan of the new revision order with that of the previous revision that was terminated.

OSM does the following when it merges the compensation plans:

- OSM carries forward all the Not Started Redo compensation tasks from the terminated revision.

Note

OSM compensates these tasks only if there are significant data changes compared to their prior execution.

- If there are pivot- sub-processes that have not started prior to the arrival of the new revision, OSM carries them forward into the latest order revision processing and runs them in a proper sequence.

About Insignificant Revision

When OSM receives a revision that has only insignificant data changes, the changes are applied immediately, while the processing of the ongoing revision is still in progress. Thus, a revision with insignificant data is not delayed. Also, it does not interrupt or impact the current revision processing.

About Terminating Compensation

When OSM receives a revision order while it is still processing a revision on the same order that it received earlier, OSM terminates the compensation for the ongoing revision order before processing the new revision. The Terminating state provides a transitional stage to ensure that the current compensation plan runs further until it reaches a safe point before starting and processing the latest revision that is queued. To avoid unnecessary processing, OSM does not create successors on the redo tasks upon its completion during the Terminating state.

The revision processing of an ongoing revision remains in the Terminating state until the following conditions are met:

- The compensation remains in the Terminating state until all Undo tasks are processed and completed.
- If there is a new component/pivot- sub-process in the current revision and it has started prior to the arrival of the new revision, the compensation of the current revision remains in the Terminating state until the execution of the new component/pivot sub-process is completed through the execution of tasks in the Amend-Do execution mode.
- If there is a change in flow during compensation (for example, if a Redo task is completed at a status that is different from the status that it was run earlier). The compensation of the current revision remains in the Terminating state until the dead path is rolled back, which is done by undoing all tasks in the dead path, and the new path is run and completed by processing the tasks in the Amend-Do mode.

OSM terminates the compensation when it considers the current state of the compensation has reached a safe point (when the aforementioned conditions are met) and starts processing the queued revision. OSM merges the terminated compensation plan with the new compensation plan to ensure that the compensation tasks which were skipped are now run as part of the processing of the new revision order.

Disabling Processing of Revisions on In-flight Revision Orders

By default, processing of revisions on in-flight revision orders is enabled for cartridges with target version 7.4.0.0.0.

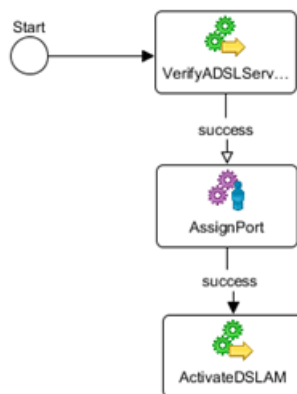
To disable this functionality, you configure the Process Amendment transaction for the Amending order state. You configure the Process Amendment transaction for the Amending order state by removing the permission to the Amending - Process Amendment transaction for a selected role. For details on removing and granting permissions to transactions, see the topic about Configuring Order Lifecycle Policies in *Design Studio Modeling OSM Processes Online Help*.

Example: Revising an In-flight Revision Order

A simple example of how OSM processes revision on an in-flight revision order is as follows:

1. A customer orders a DSL service with 3 MBps bandwidth. An order is created and sent to OSM.

[Figure 13-7](#) shows the start of the process. In this example, the process begins with the **Verify_AD_SL_Service** task and then transitions to the **Assign_Port** task.

Figure 13-7 Amendment Order Example

2. OSM verifies that the service with 3 MBps is available and transitions to the next task, **Assign_Port**.
3. On availability of a port, it transition to the **Activate DSLAM** task.
4. While the **Activate DSLAM** task is being completed, the customer calls back and requests to change the bandwidth to 5 MBps.

The CSR creates a revision order in the CRM system with the revised bandwidth value of 5 MBps and submits the order to OSM.

5. OSM receives the incoming customer order and detects that it is a revision to an in-flight order.
6. OSM accepts the revision order, calculates the compensation plan, and begins to run it. OSM recognizes that compensation is necessary because the data (bandwidth) that was on the order as input data when this task ran previously has now changed.
7. The revision order requests that the **Verify_ADSL_Service_Availability** task be redone to ensure that the 5 MBps service is available.

The value set by the **Verify_ADSL_Service_Availability** task is changed. The **Verify_ADSL_Service_Availability** task has an execution mode of Redo.

8. While the order is running the **Verifying_ADSL_Service_Availability** task, the customer calls back and requests to change the bandwidth to 10 MBps.

The CSR creates another revision order in the CRM system with the revised bandwidth value of 10 MBps and submits the order to OSM.

9. OSM receives the incoming customer order, and detects that it is a revision to an in-flight revision order.
10. OSM accepts the revision and initiates the termination of the current revision and queues the latest revision. You can monitor the amendment in the Amendments tab in the Order Management Web Client.
11. On completion of the **Verify_ASDL_service_availability** compensation task, the current revision gets Terminated and the **Assign_Port** compensation task is not started. This task is carried forward to the latest revision.
12. OSM carries forward the Not Started **Assign_Port** compensation task from the terminated revision.

Note

OSM compensates these tasks only if there are significant data changes compared to their prior execution.

About Controlling When Amendment Processing Starts

You can delay amendment processing for an order. For example, the order might be in the middle of running an automated task that is executing system interactions with fulfillment systems, so you want to postpone the processing of the revision until after the tasks complete. After the system interaction is complete, OSM can begin processing the revision.

During amendment processing, the order is in the Amending state, which prevents normal processing such as task updates. This allows compensation to deal with one set of data changes without also needing to carry out normal processing activities at the same time. To manage the transition to the Amending state, OSM does the following:

1. Checks permissions to allow or postpone the processing of the revision.
2. Checks if a grace period is set to allow all order activity to settle. If so, it waits for the grace period to end.
3. Transitions the order to the Amending state.

To control when amendment processing starts, you use the order life-cycle policy to control OSM transactions. A **transaction** is an action taken by the OSM system. For example, for the In Progress state, you can prevent the Process Amendment transaction from occurring until a condition is true.

See "[Modeling Order Life-Cycle Policy States and Transitions](#)" for more information about transactions.

To manage amendment processing, OSM uses two order state transactions, in the following order:

1. **Submit amendment.** This transaction occurs when the revision order is submitted. You can specify conditions that determine if the order can be amended or not. Because the evaluation of the condition is triggered when the revision order is submitted, the condition does not need to be based on data, but it can include data as part of the condition.
2. **Process amendment.** If the revision order is accepted, OSM evaluates this transaction to determine if the amendment can be processed now, or if it needs to wait for a specified amount of time, or if it needs to wait until all accepted tasks are completed. This condition is evaluated based on data in the order. If the condition returns false, the amendment is queued. The condition is re-evaluated whenever the data changes. When the condition evaluates to true, the transition to the Amending state can occur.

A **grace period** specifies a period of time to wait for all accepted tasks to complete before an order can transition to a different state. For example, if an automated task has sent a request to an external system, but the external system has not responded, OSM does not know if the task has been completed and therefore does not know if the task needs to be compensated. A grace period set on the Process Amendment order state transaction can allow the order the opportunity to reach a known state for all current tasks before transitioning to the Amending state.

Grace periods are defaulted to be indefinite, so OSM waits until all currently accepted tasks are completed before transitioning to the target state. You can limit the grace period:

- You can set the grace period to zero, which specifies that OSM not wait for any accepted tasks to complete before transitioning to the target state
- You can provide a time limit; for example, one hour (to give all accepted tasks a limited time to complete before transitioning to the target state).

If an automation response is received for a task after the order has transitioned to the Amending state, an automation exception is thrown, because the automation plug-in cannot process the response when the order is in the Amending state. The automation exception is sent to the JMS response queue and is retried. When the retry limit is reached, the message is forwarded to an error destination, if one is configured. To manage exceptions that occur during amendment processing, you can review the errors to determine if the messages can be resubmitted or handled by fallout.

If there are multiple queued revisions waiting for the grace period to end, OSM selects the latest version among the queued amendments to process. The other versions are assumed to be out of date and are ignored. See "[About Submitting Multiple Revisions of an Order](#)" and "[Modeling Order Life-Cycle Policy States and Transitions](#)" for more information.

About Compensation

The following sections describe how compensation occurs.

About Order-Level and Task-Level Compensation Analysis

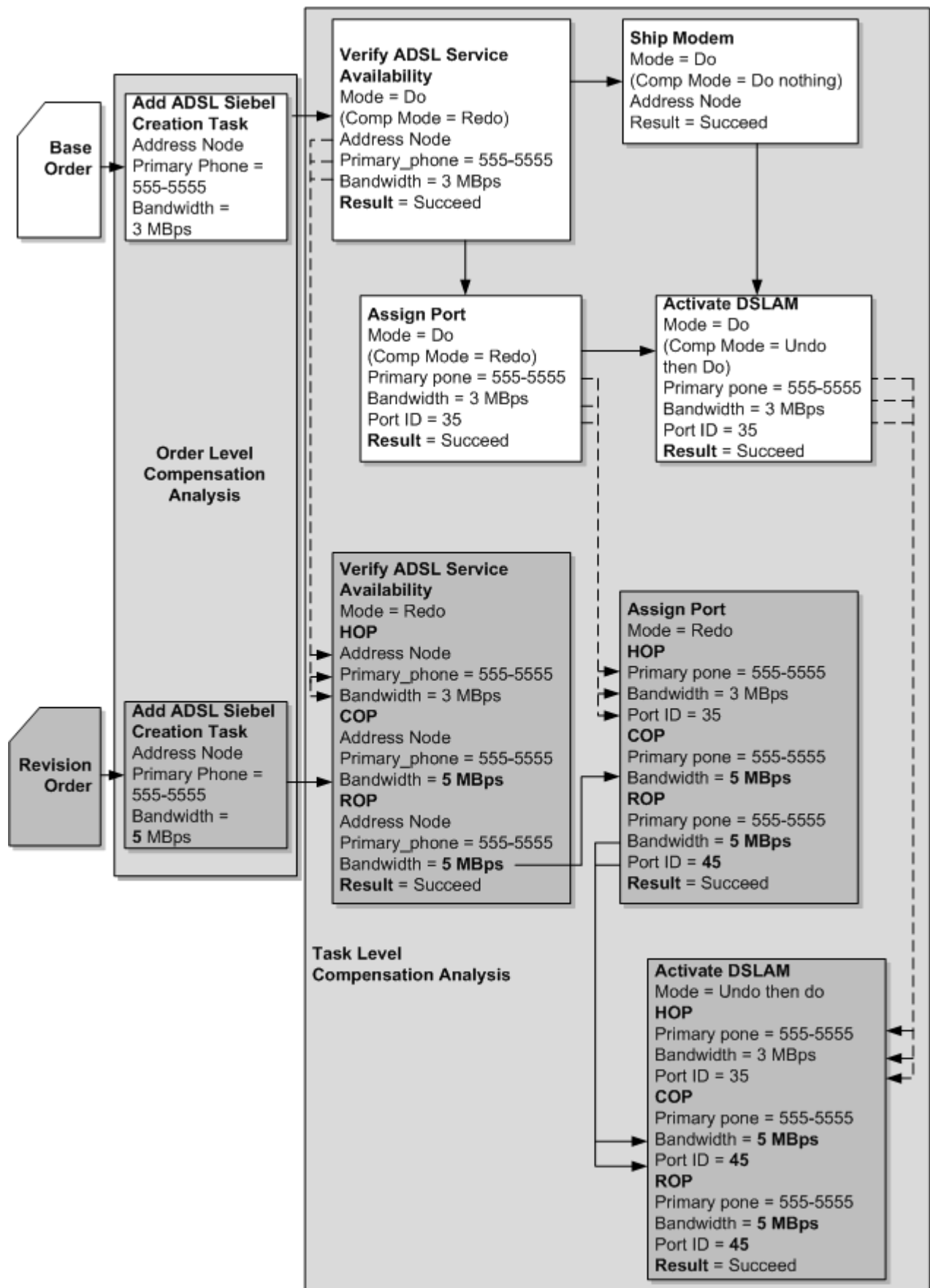
When the revision order is received, OSM analyzes the differences between the revision order data and the base order data (or between this revision order data and the last submitted revision order data) to see if a compensation is indicated. Changes and updates to order data can occur in the context of task data views or order data views.

OSM then begins analyzing impacted tasks. OSM provides the following data perspectives for each individual task which are snapshots of data that OSM uses to calculate whether a task needs to be compensated. These data perspectives are:

- **Historical order perspective (HOP):** Specifies the data used when the task last ran in Do mode and changed to the Completed state (or Redo mode if the task last ran as part of compensation for a previously submitted revision order).
- **Contemporary order perspective (COP):** Specifies the unchanged task data from the last time the task completed in Do or Redo mode (for example, from the tasks run for the base order or for a previous revision order). COP also shows any new or changed data from the current revision order and from the tasks triggered from that revision order that compensated prior in the process flow to the compensation task currently being analyzed.
- **Real-time order perspective (ROP):** Specifies the last change to a parameter value by any task or at the order level (for example through order-level updates). This perspective may be different from the COP because the COP only provides a view of task data for previously run compensation tasks and revision order data and may not represent the last change to a parameter value. For example, the COP may include unchanged data from when the parameter that was originally processed by the Task, but that same data parameter could have been updated in a later task and so the current data would have a different value than the one displayed in the COP.

[Figure 13-8](#) describes a process-based order, where a subscriber requests ADSL service with 3MBps speed. The order is submitted to OSM and service fulfillment begins. The subscriber calls back while the base order is in-flight and has just completed the **Activate DSLAM** task and requests the order be changed from 3MBps to 5MBps speed. In this scenario, the existing port does not support 5MBps. The compensation process proceeds as follows:

Figure 13-8 Changing a Service Request



1. When OSM receives the revision order, OSM compares the creation task data of the revision order with the creation task data of the base order to determine if any data changes have occurred to significant data.
2. Because the bandwidth changed from 3 MBps to 5 MBps and the bandwidth parameter is designated as significant, OSM begins task-level analysis for the first task in the process. OSM compares the **Verify ADSL Service Availability** HOP and COP and determines that the task must be redone because of the bandwidth change and because the compensation strategy for that task is redo.

OSM updates the results of the task and any data changes because of redoing the task to the ROP. The **Verify ADSL Service Availability** ROP becomes the COP for the **Ship Modem** task and the **Assign Port** task.
3. The compensation mode for **Ship Modem** is Do Nothing, so no compensation analysis occurs for that task. The compensation mode for **Assign Port** is Redo, so compensation analysis begins for that task. OSM compares the HOP and COP for the Assign Port task and determines that the task must be redone because of the bandwidth change. OSM adds the result of redoing the task to the ROP which includes the bandwidth change and a new port ID because the original port ID could not handle the increased bandwidth requirement. The ROP becomes the COP for the **Activate DSLAM** task.
4. The compensation mode for **Activate DSLAM** is Undo then Do, so compensation analysis begins for that task. OSM compares the HOP and the COP for the **Activate DSLAM** task and determines that the task must be undone then redone because of the new port ID and the bandwidth changes. OSM adds the results to the ROP. Processing continues normally after this task.

Note

In this scenario, **Activate DSLAM** is the last task; however, if there had been additional tasks that had completed after **Activate DSLAM**, OSM would have had to undo them all prior to undoing Activate DSLAM regardless of the compensation strategy associated with those subsequent tasks. This scenario only applies to tasks running in Undo then Do mode.

You can use update order transactions to make changes to order data using automation plug-ins from the task context (this includes automated task, task event, and task jeopardy notification automations) and also from the order context (this includes order-level notification, event, and jeopardy automations). OSM captures any data update made from a task context in the HOP and COP and are therefore guaranteed to be reflected in any compensation analysis for that task initiated by new revision orders. Order updates can also be applied to the order-level data by referencing the view for that order data defined in the query task that you can associate to an order in the Order Specification editor **Permissions** tab, **Query Task** sub tab (see "[Modeling Query Tasks for Order Automation Plug-ins](#)" for more information about query tasks for order-level data). Updates at the order data level should be done with care because these updates are not included as part of compensation analysis and do not generate a HOP or COP. OSM attempts to apply any order-level change to the closest task that has been created or completed, but these updates are not guaranteed deterministically like the task-level updates are. For more information about how update orders can be used in automation plug-ins, see *OSM Developer's Guide*.

OSM does the following when discrepancies occur between the contemporary order perspective and the historical order perspective:

- Adds revision order nodes if they do not match nodes of the last submitted order data or the nodes in the historical task perspective.

- Changes revision order node values if the nodes do match the values found in the last submitted order data or the nodes in the historical task perspective.
- Deletes nodes if the nodes are in the last submitted order data or in the historical task perspective but not in the revision order.

In Oracle Communications Service Catalog and Design - Design Studio, you can model compensation strategies for manual and automated tasks statically from a predefined list or dynamically from revision order data. If you model the compensation task dynamically, you can create an XQuery that has access to order data provided in the contemporary and historical perspectives as well as a comparison between the two. You can use the results of this comparison to dynamically select an appropriate task-level compensation strategy. For more information about compensation strategies, see "[Modeling Compensation for Tasks](#)".

About Order Data Position and Order Data Keys

OSM compares order data in the following ways:

- By comparing the position of the XML nodes of the base order (or last submitted revision order), with the position of the XML nodes in the current revision order. This is not the recommended method, since the result of the comparison can be unexpected and cause compensation to behave in a way you do not want.
- By comparing order data keys in the order specification order template tab for specific data elements. This is the recommended method. When OSM receives a revision order, it compares the order data keys from the revision order with the order data keys in the base order (or last submitted revision order). When OSM finds a matching order data key, it compares the values in each element.

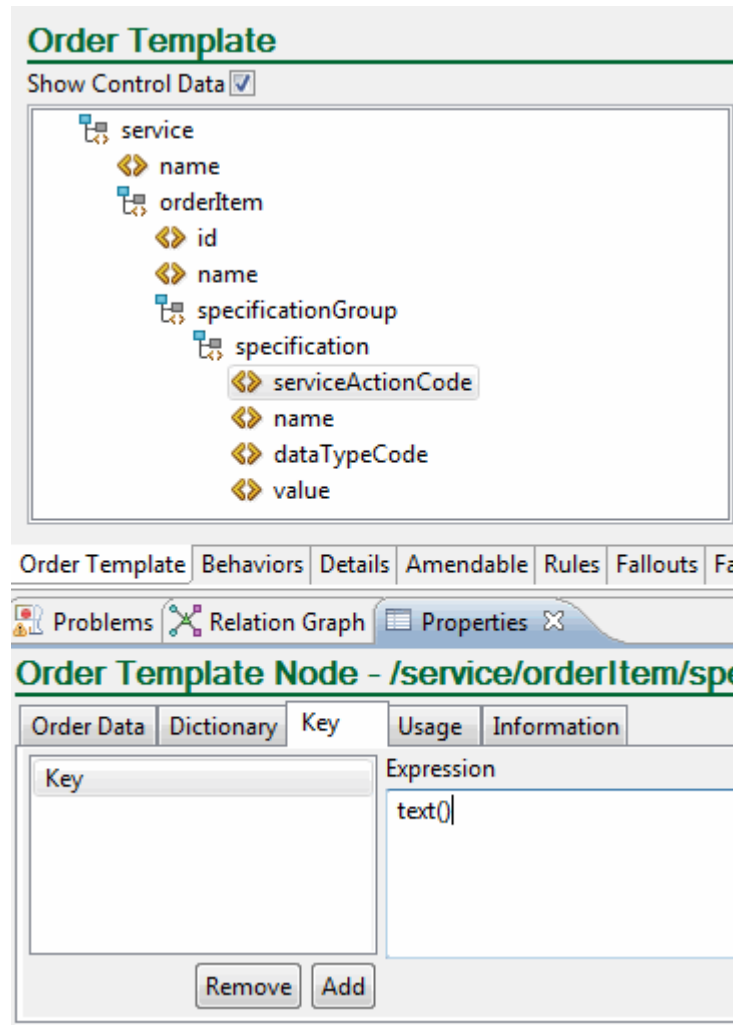
Note

OSM uses order data keys to determine order data changes during compensation and to identify pivot nodes that generate multiple task instances based on multi-instance data nodes (see "[Generating Multiple Task Instances from a Multi-Instance Field](#)") and should be distinguished from order keys used to match base orders with revision orders (see "[About Order Keys](#)").

To set an order key for a data element value, you must specify the data element as an XPath expression in the **Key** subtab on the **Order Template Node** editor.

Oracle recommends using order data keys for multi-instance data nodes to differentiate between instances of the same data node, because the results are predictable. For example, the data structure in [Figure 13-9](#) can be used multiple times to identify different product specifications. You can associate an order data key to the children nodes of **specification** to uniquely identify each instance of a product specification contained in a customer order.

Figure 13-9 Order Data Key Defined in Design Studio



For example, you could set a key on **specification** that points to the **name** child node. For expression for this key would be:

`./name`

For more information about creating valid order keys, see, "[Modeling Valid Data Keys](#)."

About Data Significance

During amendment processing, OSM identifies all tasks in the order that are affected by the changed order data. It then determines whether the data being changed is flagged as significant. (When you define orders or tasks, you can mark data as **Significant** or **Not Significant**. By default, all data is flagged significant.) OSM compensates only those tasks that process significant data.

If any of the data changes are significant, OSM transitions the order to the Amending state and builds a compensation plan based on all affected tasks, creating redo or undo compensation tasks as necessary.

Changes to non-significant data are updated on the in-flight order. For example, if the customer's preferred contact method (email or text message) is marked as non-significant, a revision order that changes only that data does not trigger amendment processing. Instead, the base order is changed, and the revision order is completed without starting amendment processing. The next task that uses the changed data uses the updated values.

You can configure data significance at the following levels:

- Data Dictionary
- Order template data
- Task data

Each level can inherit or override the significance flag of its parent level. The Data Dictionary is at the top parent level. You can also configure significance for data structure definitions, but they do not participate in inheritance.

In addition to the data significance levels mentioned above, you can access the data in the order template from the **Order Template** tabs in the Order Item editor and the Order Component editor. If you change the significance of data in these tabs, you are actually altering the data in the order template.

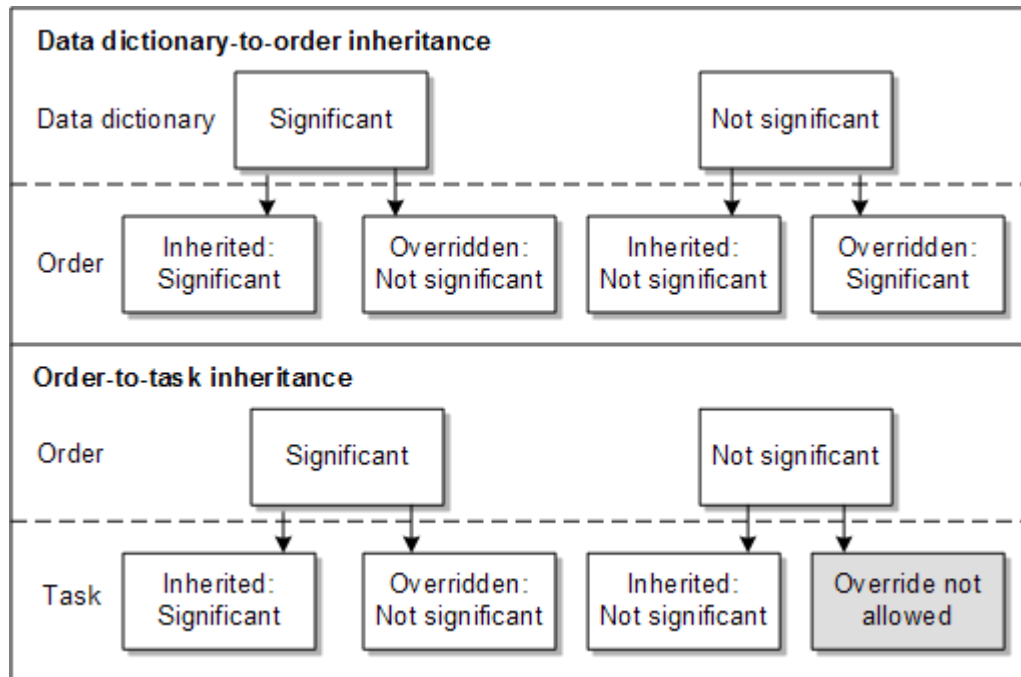
The order template can inherit or override the data significance specified in the Data Dictionary. This allows one order type to consider the data significant while another order type does not.

The task data can inherit the data significance set in the order template only to override it as non-significant data. This allows data to be significant in one task and not significant in another. In that case, a revision with that one data element changed would cause the task that considers the data element significant to be compensated: the task that does not consider it significant will not be compensated. The exception to this is that you cannot override the significance of the following types of data at the task level:

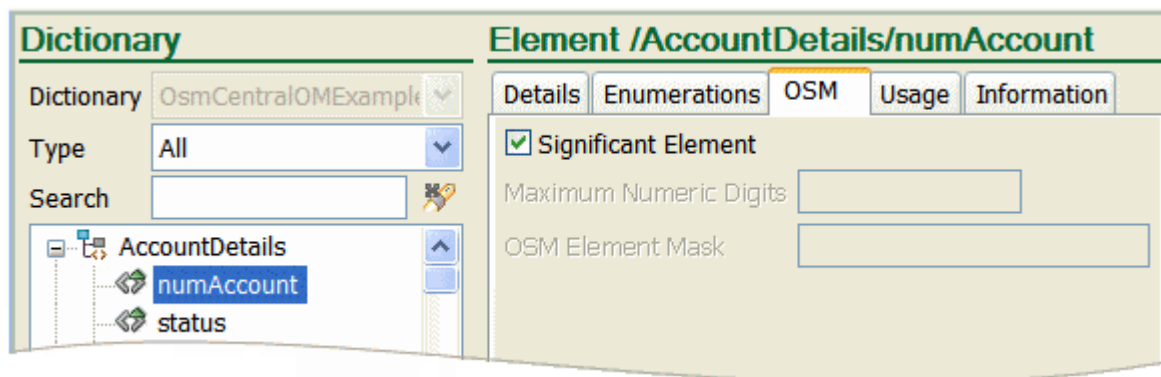
- data elements defined in data structure definitions
- the ControlData/OrderItem element and its children, if you have selected **Support Distributed Order Template** in the Order Item Specification editor **Property References** tab

It is not possible to specify a data element as not significant at the order level and significant at the task level.

[Figure 13-10](#) shows how data significance can be inherited and overridden.

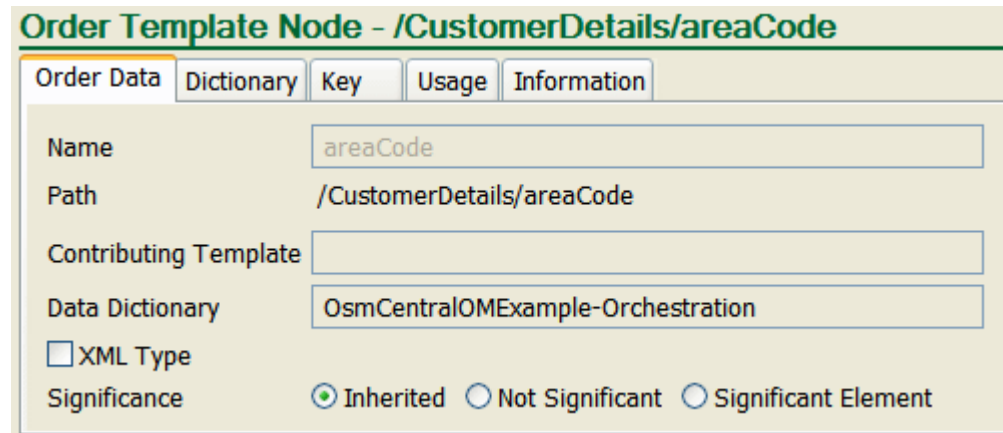
Figure 13-10 Data Significance Inheritance

[Figure 13-11](#) shows data significance specified in the Data Dictionary. Because this is the top level, there is nothing to inherit the significance from, so there is no inheritance option.

Figure 13-11 Data Significance Specified in the Data Dictionary

[Figure 13-12](#) shows data significance specified in the order template. In this example, the significance is inherited from the Data Dictionary.

Figure 13-12 Data Significance Specified in the Order Template



Order Template Node - /CustomerDetails/areaCode

Order Data Dictionary Key Usage Information

Name: areaCode

Path: /CustomerDetails/areaCode

Contributing Template:

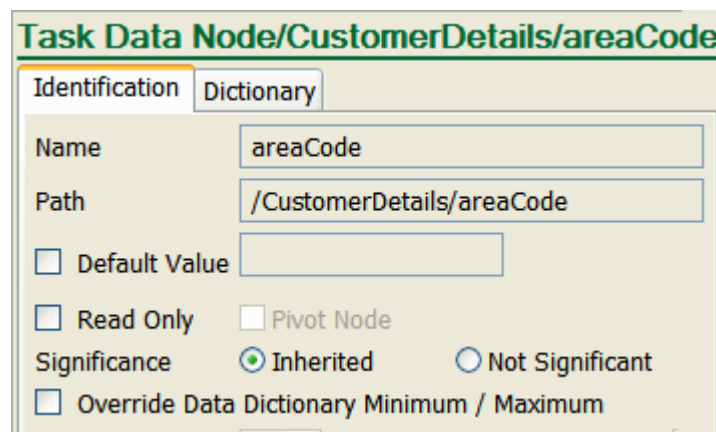
Data Dictionary: OsmCentralOMExample-Orchestration

☐ XML Type

Significance: ☒ Inherited ☐ Not Significant ☐ Significant Element

Figure 13-13 shows significance specified in the task data. Note that the significance is either inherited, or it is not significant. There is no option for significant: instead, that value is inherited from the order template.

Figure 13-13 Data Significance Specified in the Task Data



Task Data Node /CustomerDetails/areaCode

Identification Dictionary

Name: areaCode

Path: /CustomerDetails/areaCode

☐ Default Value

☐ Read Only ☐ Pivot Node

Significance: ☒ Inherited ☐ Not Significant

☐ Override Data Dictionary Minimum / Maximum

About Task Execution Modes

Tasks run in the following modes:

- **Do.** This is the normal execution mode of a task when the order is in the In Progress state.
- **Undo.** This execution mode is used when the task must undo work that has already been done; for example, to un-assign a port when an order is canceled.

Undoing tasks is performed in reverse order to how they were run. For example, if task B was completed after task A, then task B is undone before task A is undone.

Undo is used when the order component in the base order has completed, and the revision order has no corresponding order component. A cancellation order, therefore, can include no order components. This causes all of the order components in the base order to be undone. The **Orchestration Plan** tab in the Order Management web client displays nothing when this is the case, indicating that the order may have been canceled.

- **Redo.** This execution mode is used when the task must redo work that has already been done; for example, a port assignment task that needs to be performed again because the input value of bandwidth is different as a result of the revision order. Redoing tasks is performed as an optimization of the Undo and Do operations for a task in a single operation.

The Redo execution mode is used when an order component has completed in the base order, and the revision order has the same order component, but specifies different order items or data values.

- **Amend Do.** This execution mode is used when a new task must be performed while the order is in the Amending state. For example, the revision order might specify to add a service that was not in the base order. Because normal processing is not allowed during amendment processing, the Do mode cannot be used; Amend Do is used instead.

The Amend Do execution mode functions like the Do execution mode. When a task runs in the Amend Do mode, all of the permissions and automation plug-in logic for the Do mode of that task apply.

See "[Example 3: Amend Do Compensation](#)" for an example of how the Amend Do execution mode is used.

Note

You can specify which tasks can be amended by the Redo and Undo compensation modes, but Amend Do is not configurable. This is because OSM determines when Amend Do is required, and the logic followed is that of the Do mode.

- **Do in Fallout.** This is the mode for a task that runs when the task fails while running in Do mode.
- **Undo in Fallout.** This is the mode for a task that runs when the task fails while running in Undo mode.
- **Redo in Fallout.** This is the mode for a task that runs when the task fails while running in Redo mode.
- **Amend Do in Fallout.** This is the mode for a task that runs when the task fails while running in Amend-Do mode.

The Amend Do in Fallout execution mode functions like the Do in Fallout execution mode. When a task runs in the Amend Do in Fallout mode, all of the permissions and automation plug-in logic for the Do in Fallout mode of that task apply.

See "[Example 3: Amend Do Compensation](#)" for an example of how the Amend Do execution mode is used.

Note

You can specify which tasks can be amended by the Redo in Fallout and Undo in Fallout compensation modes, but Amend Do in Fallout is not configurable. This is because OSM determines when Amend Do in Fallout is required, and the logic followed is that of the Do in Fallout mode.

[Table 13-1](#) summarizes the possible combinations and the required compensation for a revision order.

Table 13-1 Compensation Types

Base Order Component	Revision Order Component	Compensation Type
Exists	Does not exist	Undo or Undo in Fallout
Does not exist	Exists	Do or do in Fallout (run after compensation is complete) or Amend Do or Amend Do in Fallout (while the order is in the Amending state).
Exists	Exists, no changes found	No compensation required
Exists	Exists, changes found	Redo or Redo in Fallout
Exists	Exists, changes found that also causes process flow changes during compensation	Redo or Redo in Fallout for the impacted tasks that do not require a new process flow. Undo or Undo in Fallout for tasks in a process flow that are undone. Amend Do or Amend Do in Fallout for completely new process flows.

Modeling Compensation for Tasks

To perform compensation, OSM must identify the tasks that need to be compensated and then do, undo, or redo them in the appropriate sequence. OSM applies these compensation execution modes regardless of whether the task is running in normal mode or in failed mode.

A task needs to be compensated if it was completed and a change to at least one significant data element in the task's data has been made. Tasks in the Received, Accepted, Assigned, or a user-defined state can also be compensated.

Note

When a task is compensated, all its successors must be compensated, whether or not they have significant changes.

Determining Task Compensation Strategy

In the Design Studio **Task Editor Compensation** tab (see [Figure 13-14](#)), you can model:

- Static amendment processing compensation strategies for manual and automated tasks using a predefined list. Static compensation strategies are appropriate when the compensation requirements for a task are invariable.
- Dynamic amendment processing compensation strategies for manual and automated tasks based on revision order data using an XQuery expression. Dynamic compensation strategies are appropriate when more than one compensation strategy is required for a task. For example, you could model the XQuery expression to select an **Undo then do** compensation strategy if the revision order bandwidth parameter is greater than 50 MB, and only a **redo** compensation strategy if the bandwidth parameter is less than 50 MB. For

more information about dynamically modeling amendment processing compensation strategies, see "[About Task Compensation Strategy XQuery Expressions](#)".

Figure 13-14 Task Compensation Options

As shown in [Figure 13-14](#), there are two scenarios that need to be compensated:

- The task needs to be re-evaluated. This means that the task includes significant data and needs to be compensated.

The static amendment processing compensation options are:

- Redo in one operation. This option is recommended because it performs the fewest number of Undo and Do operations necessary for compensation.

In the case of a manual task, the task will appear in the worklist in Redo mode, and the user can display the historical perspective and the contemporary perspective of the task data (from the last time the task was run) in two separate tabs. The user updates the data on the **Contemporary Perspective** tab and completes the task.

- Undo and Redo in two operations. Use this option when you need to roll back all order changes and perform the task again from the beginning. This option is useful when interacting with an external system that has no redo action but can process equivalent do and undo actions (for example, in the external system, implement and cancel).

Note

When this option is used, it forces all completed tasks subsequent to this task to be undone in reverse sequence prior to executing the undo and then do of this task. To redo the task, you need to roll back all subsequent tasks first, then undo the task and redo it.

- Do nothing. Use this option if redoing the task is not necessary. For example, a task that sends a customer survey email would not need to be redone, even if it includes significant data.

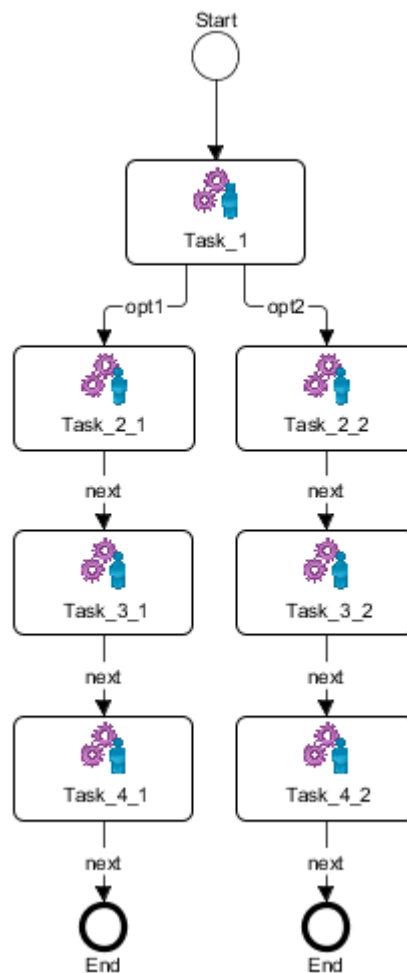
In addition, you can select the **Compensation Expression** check box and enter an XQuery that dynamically selects an amendment processing compensation option at run

time based on order data. The dynamic compensation takes precedence over the static compensation unless there is an error in the XQuery itself. If there is an XQuery error, then OSM reverts to the compensation selected with the static radio buttons.

- The task is no longer required. This occurs when an order is canceled or when a branch of completed tasks in a process becomes obsolete due to a revision.

[Figure 13-15](#) shows a process that has two potential paths. In this example, the base order followed the path from Task_1 to Task_2_1. The revision caused the path to change to follow Task_1 to Task_2_2. This means that Task_2_1, Task_3_1, and Task_4_1 do not need any compensation, because they are no longer on the path required to fulfill the order.

Figure 13-15 Process with Two Paths



The static amendment processing compensation options are:

- **Undo.** This option rolls back the task by executing the task in Undo mode to perform the roll-back operation. In the case of a manual task, this requires that the rollback be acknowledged manually in the Task web client. You cannot update the task data for an undo of a manual task in the Task web client, because the system will automatically put the data back to what it was prior to the task executing.
- **Do nothing.** This option rolls back the task automatically, without creating an undo task.

In addition, you can select the **Compensation Expression** check box and enter an XQuery that dynamically selects an amendment processing compensation option at run time based on order data. The dynamic compensation takes precedence over the static compensation unless there is an error in the XQuery itself. If there is an XQuery error, then OSM reverts to the compensation selected with the static radio buttons.

About Compensating In Progress Tasks

You can configure whether a task is included in compensation when it is completed or in progress, or you can use an XQuery expression that evaluates whether an in progress task can be included in compensation based on order data. Compensating in progress tasks is important for long running tasks where a response to a request takes hours or even days to return but the task still needs to be compensated. If you specify that a task can be compensated while it is in progress, you can also specify whether a grace period should be observed before performing the compensation. In addition, you must use an XQuery expression to evaluate any changes to the compensating task data to identify when the compensation has completed and the task can enter into normal do mode again.

For example, some automated plug-ins communicating with workforce management systems may involve the dispatching of personnel to perform work over several days. In such cases the automation plug-in sends the dispatch request to the workforce management system, and remains in progress until such time as the work completes. If a revision order were to arrive that changes some aspects of the work, then the in progress automation plug-in responsible for sending the original request should be included in the compensation plan. You can specify an XQuery that evaluates data on the in progress task communicating to the work force management system that determines if the task needs to be compensated. In addition, you can specify whether a wait period should be observed before starting compensation. You must also write an XQuery that determines when compensation has completed, for example, when the task receives the response from the new request indicating the workforce management system has received the new work details and has begun to processing the request.

[Figure 13-16](#) shows the configuration options for determining when compensation should occur on an in progress task, whether a grace period should be observed before starting task compensation, and when compensation should complete.

Figure 13-16 In Progress Task Compensation Options

In the **When an amendment occurs this task will be compensated if it is:** area, you can select:

- **Completed:** OSM only considered the task in compensation if it is in the completed state.
- **Completed or In Progress:** OSM considers the task in compensation if it is in the completed state and also if it is in progress (for example, in the received, assigned, accepted, or a custom states.)
- **In Progress Compensation Include Expression:** You can specify an XQuery that uses task data to determine whether the in progress task should be compensated. This XQuery expression overrides the **Completed** and **Completed or In Progress** options except when the XQuery is invalid.
- **In Progress Compensation Complete Expression:** You can specify an XQuery expression that uses task data to determine whether the in progress task should be compensated.

In the **When an amendment occurs if this task is in progress it will:** area, you can specify what grace period should be observed before beginning task compensation on the in progress task:

- **Wait for the grace period:** OSM observes the grace period specified on the order-life cycle for the Process Amendment transition.
- **Be excluded from the grace period:** OSM does not observe a grace period.
- **Wait for specified duration:** OSM observes the grace period statically configured for the task in seconds, minutes, hours, or days.
- **Dynamic Expression:** OSM uses an XQuery expression that dynamically specifies the wait duration based on revision order data. This XQuery expression overrides the other options except when the XQuery is invalid. For more information about compensation strategy XQuery expressions, see "Compensation XQuery Expression".

About Task Compensation Strategy XQuery Expressions

You can dynamically assign compensation strategies to tasks by creating XQuery expressions in the Design Studio **Task Editor Compensation** tab for re-evaluation compensation strategies or compensation strategies for when a task is no longer required.

Note

If the XQuery expression is invalid OSM logs the error but does not rollback the transaction. Instead, OSM uses the static compensation strategy as the default.

This section refers to the Design Studio OSM **Automated Task** or **Manual Task** editor, **Compensation** tab Compensation Expression XQuery field for re-evaluation compensation strategies:

- **Context:** The context for this XQuery is the current order data. You can get the current order data using the XML API `GetOrder.Response` function.
- **Prolog:** You can declare the XML API namespace to use the `GetOrder.Response` function in the XQuery body to extract the order information. You must declare the `java:oracle:communications.ordermanagement.compensation.ReevaluationContext` OSM Java package that provides methods that access the contemporary and historical order perspectives and compares the two. You can use the results of this comparison to determine what compensation strategy is required for a task based on revision order data.

For example:

```

declare namespace osm = "urn:com:metasolv:oms:xmlapi:1";
declare namespace context =
"java:oracle:communications.ordermanagement.compensation.ReevaluationContext";
declare namespace log = "java:org.apache.commons.logging.Log";

declare variable $log external;
declare variable $context external;

```

For more information about the classes in the OSM packages, install the OSM SDK and extract the OSM Javadocs from the **SDK/osm7.w.x.y.z-javadocs.zip** file (where *w.x.y.z* represents the specific version numbers for OSM). See *OSM Installation Guide* for more information about installing the OSM SDK.

- **Body:** The body must return a valid compensation option.

For example, the following XQuery expression creates variables for the `ReevaluationContext` methods. The expression then checks that a specific value exists in the `$value` variable and that the value in the `$significantValue` variable both exists and is significant. If the value exists and is significant, then the expression sets the compensation strategy for the task to **Undo then Do** (`undoThenDo` in the `ReevaluationContext` Java class). If not, then the expression sets the compensation strategy to **Redo** (`redo` in the `ReevaluationContext` Java class).

```

let $inputDoc := self::node()
let $shopDoc := context:getHistoricalOrderDataAsDOM($context)
let $ropDoc := context:getCurrentOrderDataAsDOM($context)
let $diffDoc := context:getDataChangesAsDOM($context)
let $value := $inputDoc/GetOrder.Response/_root/service[name='BB']//orderItemRef/
specificationGroup//specification[value='100']
let $significantValue := $diffDoc/Changes/Add[@significant='true']//
specification[value='100']
let $currentValue := $ropDoc/GetOrder.Response/_root/service[name='BB']//
orderItemRef/specificationGroup//specification[value='100']

return if (fn:exists($value) and fn:exists($significantValue))
then
    context:undoThenDo($context)
else
    context:redo($context)

```

This section refers to the Design Studio OSM **Automated Task** or **Manual Task** editor, **Compensation** tab **Compensation Expression** XQuery field for when a task is no longer required. The context, prolog, and body are similar to the XQuery expression for the re-evaluation strategy, except that the XQuery expression implements the `java:oracle:communications.ordermanagement.compensation.RollbackContext` package.

For example:

```

declare namespace osm = "urn:com:metasolv:oms:xmlapi:1";
declare namespace context =
"java:oracle:communications.ordermanagement.compensation.RollbackContext";
declare namespace log = "java:org.apache.commons.logging.Log";

declare variable $log external;
declare variable $context external;

let $inputDoc := self::node()
let $shopDoc := context:getHistoricalOrderDataAsDOM($context)
let $ropDoc := context:getCurrentOrderDataAsDOM($context)
let $diffDoc := context:getDataChangesAsDOM($context)

let $value := $inputDoc/GetOrder.Response/_root/service[name='BB']//orderItemRef/
specificationGroup//specification[value='100']

```

```
return if (fn:exists($value))
then
    context:undo($context)
else
    context:doNothing($context)
```

About Managing Compensation in the Task Web Client

When an automated task is redone, it is redone automatically. When a manual task is redone, the Task web client displays the task with an execution mode of Redo. The manual task must be processed in the Task web client.

To manage compensation in the Task web client, you can do the following:

- Perform manual undo and redo operations.
- Display the execution mode under which the tasks is running (Do, Redo, or Undo).
- Display the order state; for example, Amending.
- Display the historical data (the data as it was when the task last run) in the historical order perspective when editing a task.

Note

You can assign roles in Design Studio to specify who can redo and undo tasks in the Task web client. OSM also supports the ability to assign the different execution modes of a task to different roles. This is useful because OSM can compensate using both manual and automated tasks. For example, the regular processing of a task in Do mode could be automated, and the Undo and Redo modes for the same task could be set to a special role to be done manually.

See *OSM Task Web Client User's Guide* for more information.

Modeling Compensation for Rules in Processes

You can specify to redo a rule in a process, or to do nothing. Because rules only evaluate data, and therefore do not modify data or interact with other systems, there is no undo necessary for a rule.

Modeling Compensation for Task Automation Plug-Ins

An automated task can include multiple automation plug-ins; for example, senders and receivers. Each automation plug-in can be associated with one or more execution modes. For example, if you create an automated task to activate a service, you can use the same logic to handle the initial activation request and the redo compensation for the activation request.

Each automated task can have separate plug-ins for each of the three modes; Do, Redo, and Undo. When an automated task runs in Redo or Undo mode, OSM provides information about the task data that was present when the task was last ran. For redo tasks, the Automation framework provides the historical data, the contemporary data, and the delta to the automation plug-in for use in the plug-in logic you write. See *OSM Developer's Guide* for more information.

Compensation Examples

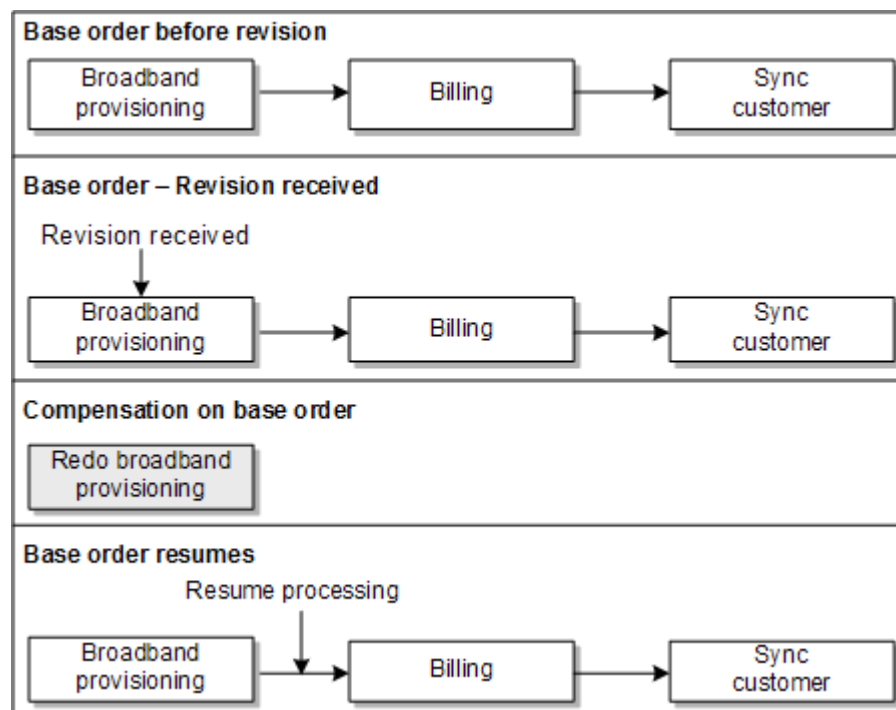
The following examples show different compensation scenarios.

Example 1: Compensation During Provisioning

[Figure 13-17](#) shows a compensation scenario for an orchestration order. In this example, OSM is running in the central order management role, fulfilling multiple functions.

1. The base order requires provisioning, billing, and customer account order components (sync customer).
2. A revision order is submitted while the order is carrying out provisioning tasks. The revision order replaces a medium-capacity service (3 MBps) with a high-capacity service (8 MBps). In this case, the content of the order components has changed in the revision order's orchestration plan, but the order components it contains and their dependencies remain the same.
3. Because the revision order was received during the base order provisioning, the compensation specifies that the provisioning order component must be redone, after which the order returns to the In Progress state, and the billing and sync customer components are then run with no compensation required.

Figure 13-17 Example of Compensation that Occurs During Provisioning



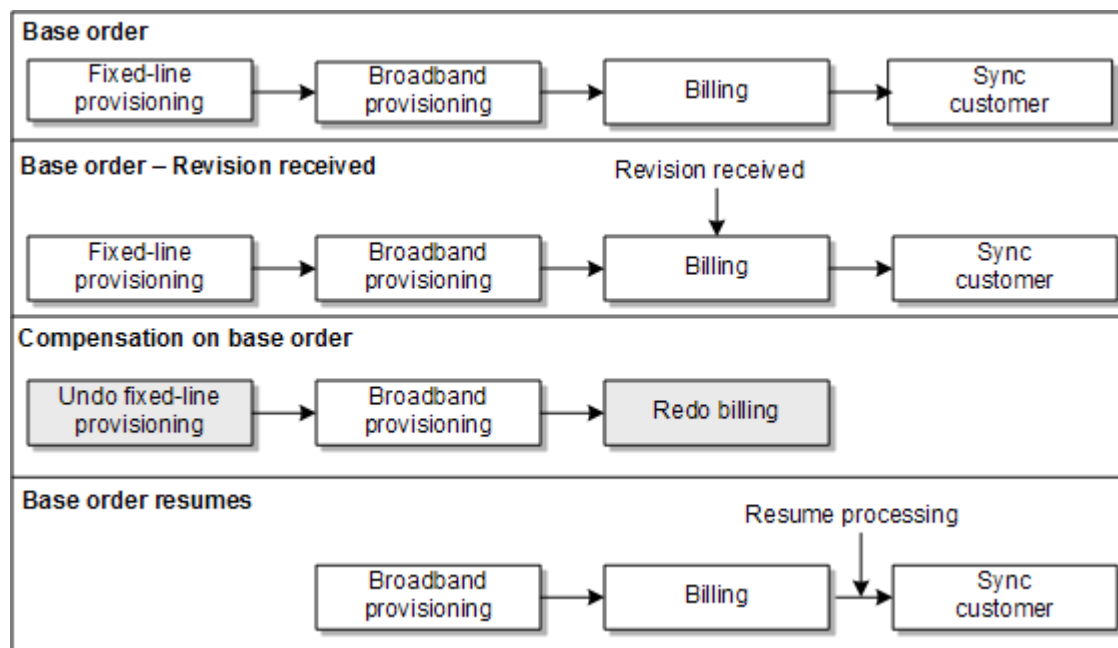
Example 2: Compensation During Billing

[Figure 13-18](#) shows a compensation scenario for billing:

1. The base order includes provisioning order components for a fixed-line service and a cable broadband service.

2. A revision order is received after the provisioning order components are complete but while the billing order components are being processed. The revision order cancels the fixed-line service.
3. The compensation plan specifies to undo the fixed-line service and to redo the billing order components. The cable broadband service requires no compensation. Following the redo of the billing order components, the order resumes normal processing.

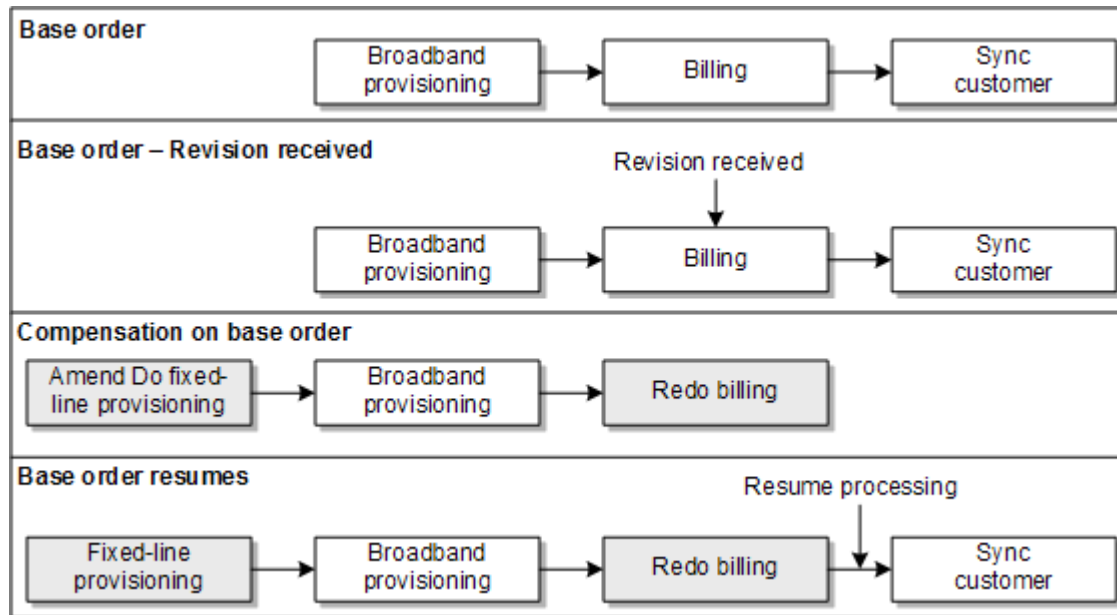
Figure 13-18 Compensation for Removing a Service



Example 3: Amend Do Compensation

[Figure 13-19](#) shows a scenario for Amend Do compensation:

1. The base order includes provisioning order components for a broadband service.
2. A revision order is submitted while billing order components are being processed. The revision order adds a fixed-line service.
3. The compensation plan creates a new dependency between the fixed-line service and the broadband service. Therefore, OSM must use Amend Do to first perform the new task and then process the broadband service order components. The billing order components are redone, and processing continues normally.

Figure 13-19 Compensation for Adding a Service

Examples of Changes to Orchestration Plans

You can use the OSM Order Management web client to see how compensation affects an order's orchestration plan.

[Figure 13-20](#) shows how an orchestration plan changes when a single service attribute changes. In this example, the connection speed changes from 8 MBps to 16 MBps. The order components remain the same, but the value of the connection speed changes in the provisioning component and in the billing component.

Figure 13-20 Orchestration Change Due to Revision: Change Service Attribute

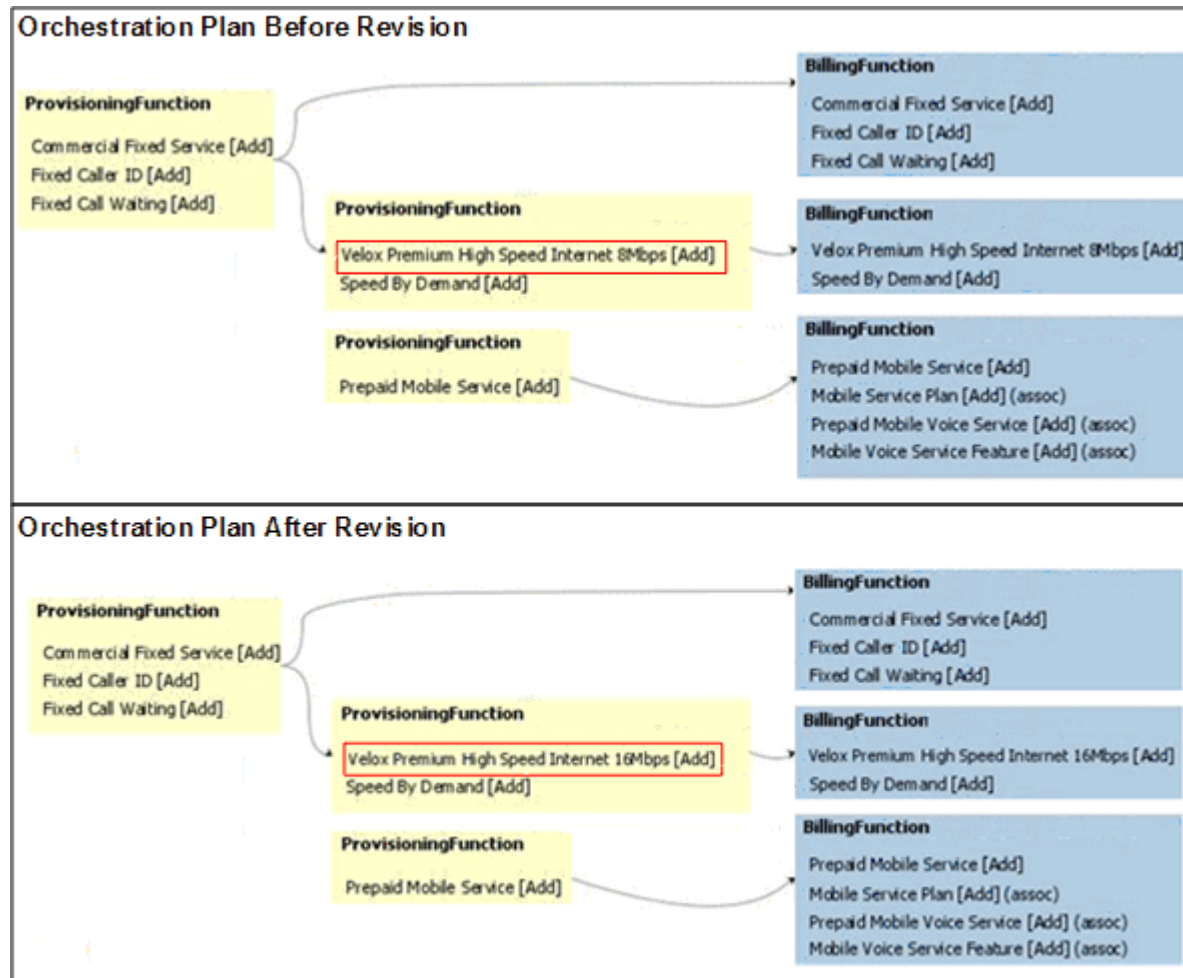


Figure 13-21 shows how an orchestration plan changes when a revision order removes a service from the base order. In the example, the Fixed service was ordered in the base order, but it was removed in the revision order. The provisioning and billing components are removed, and the DSL provisioning component no longer has a dependency on the Fixed order component.

Figure 13-21 Orchestration Change Due to Revision: Remove Service From Order

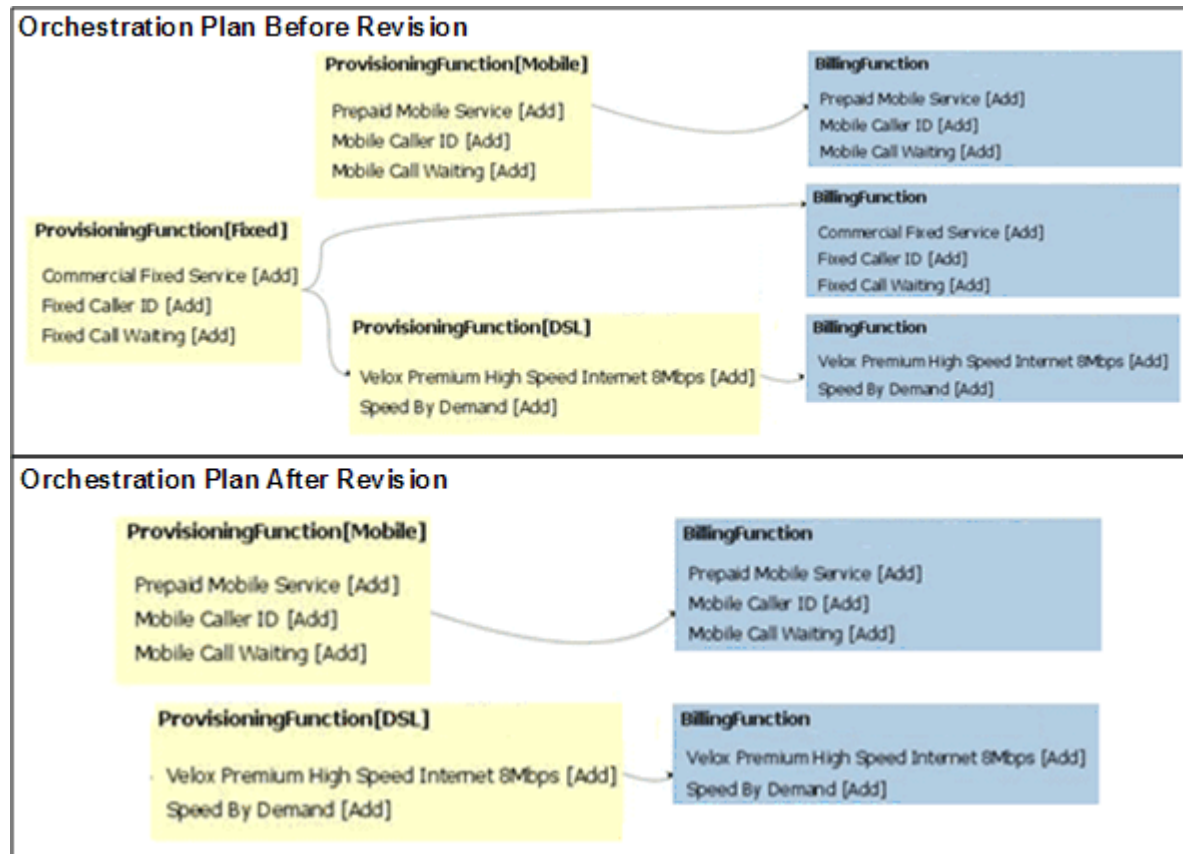
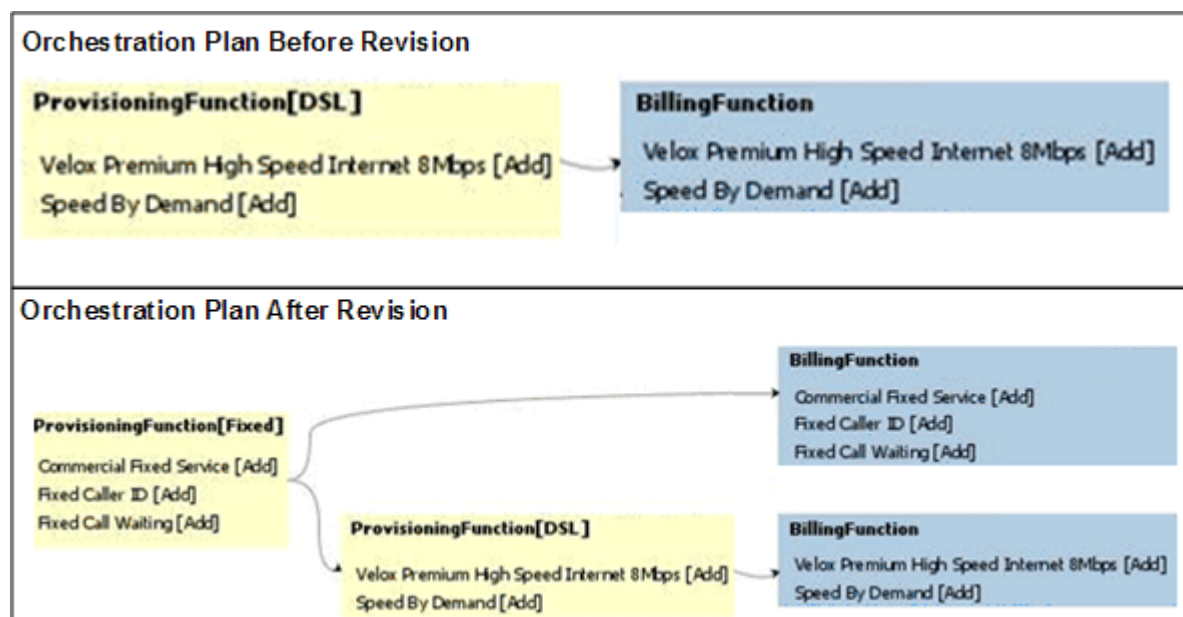


Figure 13-22 shows how an orchestration plan changes when a service is added by a revision order. In this example, the Fixed service is added. This creates a new dependency for the DSL provisioning component.

Figure 13-22 Orchestration Change Due to Revision: Add Service to Order



Modeling a Point of No Return

The following sections describe how to model a point of no return on an order where a revision order is no longer possible.

Fulfillment Pattern Point of No Return

There are two ways to set a point of no return. The first, only available for orchestration orders, is to set it on the fulfillment pattern using fulfillment states. The second is to write an expression in the order life-cycle policy.

When you use the fulfillment pattern to set a point of no return, the point of no return rules set a point of no return value for that order component. Order life-cycle policy conditions can then leverage this point of no return value for restricting order amendments. See "[Life-Cycle Policy Point of No Return](#)" for more information.

When you create a point of no return, model the following in Design Studio:

- Define fulfillment states. These are required before configuring a point of no return on the fulfillment pattern. See "[Modeling Fulfillment States and Processing States](#)" for more information.
- Define a point of no return value list in the fulfillment pattern. Create a name for your point of no return and indicate whether it is a hard point of no return or not. Alternatively, you can create a point of no return value on the fulfillment pattern extended by your fulfillment pattern and allow the point of no return values to be inherited.
- Define point of no return rules for the point of no return values you created. Point of no return rules are specified at the order component level. Point of no return rules involve selecting one or more fulfillment states to map to the specified point of no return value. Additionally, because order component definitions are hierarchical, a sub-component of the order component associated with the orchestration plan inherits the point of no return rules defined on the orchestration plan order component. This sub-component may also specify its own additional point of no return rules.
- Define one or more transition conditions in the Order Lifecycle Policy to check the point of no return value.

Life-Cycle Policy Point of No Return

When you use life-cycle policies to set a point of no return, you define the point of no return as an expression in the order life-cycle policy, by setting conditions on the Submit Amendment transaction.

The following example shows a simple point of no return expression:

```
declare namespace oms="urn:com:metasolv:oms:xmlapi:1";
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";

let $taskData := fn:root()./GetOrder.Response
let $rootData := $taskData/_root

return
  if (($rootData/PoNR/text() = "HARD"))
  then (
    true()
  ) else (
```

```
false()  
)
```

When a revision order is received, OSM checks the life-cycle policy to see if there are any point of no return conditions preventing the transition to the Amending order status. If OSM finds any point of no return conditions that are met, the order is not allowed to be amended. In the example above, if the broadband service is billed before the fixed-line service is provisioned, the order has passed the point of no return, even though the fixed-line service has not passed its point of no return.

If the life-cycle policy determines that the revision is not allowed, an `OrderTransactionNotAllowedFault` message is returned to the order-source that submitted the revision order.

About Modeling Order Change Management

When you model order change management, you configure the following OSM entities:

- Data dictionary. When you create data elements, you can assign them significance. If data is significant, it is considered for amending. See "[About Data Significance](#)" for more information.
- Order specification. When you create an order specification, you configure the following:
 - Data significance at the order level. You can inherit significance from the Data Dictionary, or you can define order-specific significance. See "[About Data Significance](#)" for more information.
 - If the order is amendable or not.
 - The order key. See "[About Order Keys](#)" for more information.
 - The data element that defines the order version. See "[About Submitting Multiple Revisions of an Order](#)" for more information.
 - Whether or not to publish order events about amendment processing. You can choose to publish events when an amendment is started, completed, queued, being terminated, terminated or abandoned. An amendment can be abandoned when it is queued for processing and a subsequent amendment supersedes it. See "[Modeling Order Life-Cycle Policy States and Transitions](#)" for more information.
- Tasks. You can configure the following:
 - Data significance at the task level. See "[About Data Significance](#)" for more information.
 - How the tasks should be compensated. See "[Modeling Compensation for Tasks](#)" for more information.
 - The roles that can redo and undo tasks.
 - If automated, the automation plug-ins for redo and undo modes of tasks.
- Rules in processes. You can configure if the rule should be redone or not. See "[Modeling Compensation for Rules in Processes](#)" for more information.
- Order life-cycle policy. You configure the conditions that allow an order to be amended. See "[About Controlling When Amendment Processing Starts](#)" and "[Life-Cycle Policy Point of No Return](#)" for more information. See "[Modeling Order Life-Cycle Policy States and Transitions](#)" for information about order states.

Troubleshooting Order Change Management Modeling

You can use the following methods to troubleshoot your order change management modeling:

- You can use the following OrderManagementDiag.wsdl web service operations:
 - GetOrderCompensations: Returns a list of compensations against a given order.
 - GetOrderProcessHistory: Returns multiple process history perspectives.
 - GetCompensationPlan: Returns a set of compensation tasks and the dependencies between them.
- See the **PONR_OrderID.xml** file. This file is generated when a Submit Amendment transaction is called.

About Order Change Management at the Orchestration Layer

To manage changes to an orchestration order, OSM uses **order compensation**. OSM analyzes the required order changes and their impact on everything that has already been completed by the in-flight order including manual updates from Task web client users and order updates from automated tasks. OSM then creates a **compensation plan** to define the actions that need to be carried out to amend the in-flight order. After compensation has ended, the in-flight order will have incorporated the required order changes and continues executing normally. You can recognize when compensation is happening to an orchestration order when the order is in the Amending state or the Cancelling state.

Note

If you submit a revision order that uses a different cartridge version from the one that the original base order was created with, OSM uses the original base order cartridge version to run any required compensation tasks and not the cartridge version used to create the revision order.

Triggering amendment processing using revision orders is the most efficient way to manage changes made to in-flight orders. OSM automatically detects the revisions that must be made and changes the orchestration plan as necessary. No manual work is required to find changes that need to be made.

A revision order is sometimes called a supplemental order. This order contains all the relevant data for the order, including the updated requirements. During the **amendment processing** phase, OSM compares the data in the revision order with the data in the base order and makes the changes as required (a single revision order can make multiple changes to an order). This allows the base order to continue processing with, and compensating for, the customer's new order requirements provided in the revision order. The customer does not have to wait for the base order to be completed or canceled before changing it. A revision order can also be used to correct a failed order.

When you model orders and tasks, you can control the amendment processing that is allowed for the order. For example:

- If the order is allowed to be amended
- At which point in the order processing the order is no longer allowed to be amended (the PONR)

- Who can manage revision orders in the Task web client
- Which data needs to be compensated, and which does not

For more information about amendment processing and compensation, see "[About Compensation and Orchestration](#)".

About Compensation and Orchestration

OSM performs compensation on both process-based orders and orchestration orders. When compensating an order that has an orchestration plan, the compensation can change the orchestration plan.

Each orchestration order has its own unique orchestration plan, generated specifically for that order. Therefore, to manage a revision order for the base order, OSM must generate a new orchestration plan for the revision order. The orchestration plan for the revision order can be different from the orchestration plan for the base order; for example, it might include different order components, with different dependencies and different order items.

By contrast, a process-based order has a predefined process; the process is not generated when the order is created. The tasks that make up that process and the flow of those tasks in the process do not change. The data values for those tasks change as a result of a revision, and the path through the predefined process may change as a result of compensation, but the overall process remains the same.

To manage compensation for an orchestration plan, OSM needs to recognize and account for dependencies between the order components in the order that is being amended. The compensation required depends on whether components exist in one or both orders' (revision order and base) orchestration plans and on whether changes to the contents of those order components (such as different order items) exist.

Redoing an order component in an orchestration plan is performed by redoing the tasks run by the order component. In redoing order components, OSM follows the sequence of dependencies in the orchestration plan. OSM takes into account the dependencies from the revised orchestration plan, unless a successor component has previously started in the original base order, in which case the dependency is considered resolved.

If a cartridge is built with its target OSM version set to 7.5.0 or newer, additional logic comes into play with respect to data change dependencies during revision. For such cartridges, these dependencies are re-evaluated afresh in the context of the revised orchestration plan. This contrasts with the approach for cartridges built with older target OSM version, where if a data change dependency had been satisfied in the base order, it is automatically deemed satisfied in the revision order.

OSM analyzes the order component compensation type and component dependencies to determine the sequence of component compensation. OSM performs order component compensations in the following stages:

- Reverse compensation: In this stage, OSM performs only undo compensation tasks for order components. OSM performs undo tasks for order components in the reverse order of dependencies that existed between the components in the original.

For example, OSM performs undo tasks for order component B before performing any undo task for order component A if B was dependent upon A in the base order.
- Forward compensation: In this stage, OSM fulfills order components that have changed (redo) or been introduced (amend do) based on the order of dependencies, which is derived after taking into account dependencies from the revised orchestration plan.

The revised orchestration plan may include new components to be completed using Amend Do and Redo compensation types.

Note

When switching from reverse to forward compensation, OSM identifies the new order components that need to be completed using the Amend Do compensation type. These components participate in the compensation plan as compensation items. This facilitates appropriate compensation sequencing for compensation tasks of existing downstream order components or other components that require amend do compensation.

All processing not related to compensation is suspended for an orchestration plan until compensation is complete. After compensation is complete, the order is restored from the Amending state to an In Progress state and normal processing continues.

About Point of No Return

In some cases, there may be a point in the order process after which it becomes impossible or undesirable to make changes to an order. This is called a point of no return.

There are two types of points of no return in OSM.

- A **hard point of no return** indicates that amendments to the relevant part of the order are either impossible or undesirable. In the case of a hard point of no return, a revision order is not possible. Instead, you can create a **follow-on order**. See "[About Inter-Order Dependencies](#)" for more information about follow-on orders.

Note

A follow-on order is not a change to an in-flight order but is an alternative when revising the in-flight order is not possible. Follow-on orders are used to make changes to items on an order that have not yet been completed but are past the point of no return. OSM manages follow-on orders to ensure they do not run until the order items upon which they depend are completed.

- A **soft point of no return** indicates that order amendment processing is still possible, but there are consequences for the customer. For example, you can specify to bill a customer for an extra charge if the order is revised after the soft point of no return has been reached.

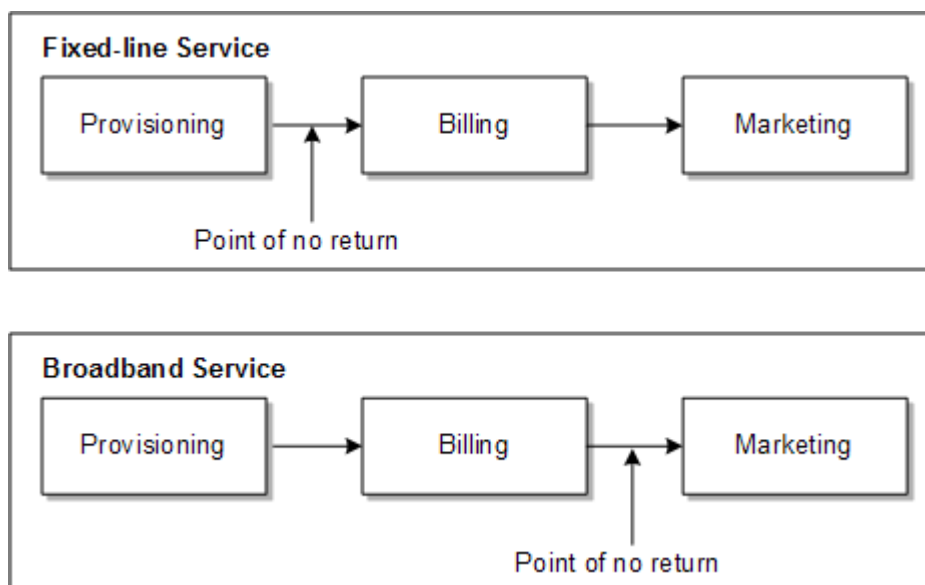
You can define multiple point of no return milestones in an order's fulfillment flow. For example:

- For a fixed-line service, a point of no return after provisioning.
- For a broadband service, a point of no return after billing.

All soft or hard points of no return depend on the order life cycle policy conditions which control whether orders can transition from the In Progress state to the Amending state. See "[Modeling Order Life-Cycle Policies](#)" for more information. A point of no return value that the order life cycle policy evaluates is typically set at the order item level. This allows order components with varying processing durations to run, instead of stopping the entire order at the first order item with a point of no return. You can also set the point of no return to a fulfillment state which provides aggregated states for a group of order items and for the overall order. See "[Modeling Fulfillment States](#)" for more information.

[Figure 13-23](#) shows two different point of no return scenarios.

Figure 13-23 Point of No Return for Different Services



Modeling Fallout

This chapter describes order fallout modeling best practices in an Oracle Communications Order and Service Management (OSM) solution.

Overview of Fallout

Modeling fallout involves considerations in the following areas:

- **Prevention:** Identify possible sources of errors in the solution and model entities to prevent the error from occurring.
- **Detection:** Identify ways in which you can configure OSM modeling entities to detect failures on orders or tasks within orders. For example:
 - OSM receives an error message from a downstream system at an automation plug-in that transitions the task to a fallout execution mode.
 - An OSM operator working on a manual task uses the Task web client to transition a task to a fallout execution mode when progress on the manual task is no longer possible.
 - OSM detects a failure at order creation. For example, an order recognition rule recognizes an order but the order fails because of a validation error.
- **Inform/Investigate:** Identify ways in which you can configure OSM modeling entities to provide information to fallout specialists so they can investigate the problem. For example:
 - When an automated task transitions to a fallout execution mode because of an error message returned from a downstream system, you can also configure the task to update the order item processing state that the task was processing that also changes the order item processing states of parent order items, and so on up the order item hierarchy. Fallout specialists can search for and view orders, order items, and tasks based on order item processing states or based on task execution mode.
 - Depending on the error or order processing state change, you might need to notify other systems or fallout specialists that a problem occurred and why.

For example, the customer relationship management (CRM) system must know if an order has failed because of incorrect data. You can configure OSM to send email notifications to fallout managers or to notify an external trouble-ticketing application. Or if a task on an order or the order itself is taking too long to complete, you can configure a jeopardy notification.
 - Order management personnel can monitor the progress of orders and tasks in the Task web client or in the Order Management web client. You can configure OSM to send failed tasks and orders to specific personnel associated with fallout workgroups. You can search for orders with failure and warning indicators on them.
- **Resolution:** You can model OSM to automatically fix problems and then resume or restart the order or tasks within the order or you can model OSM to notify fallout specialists so they can manually investigate and resolve problems.

You can recover from order and task failures in various ways. For example:

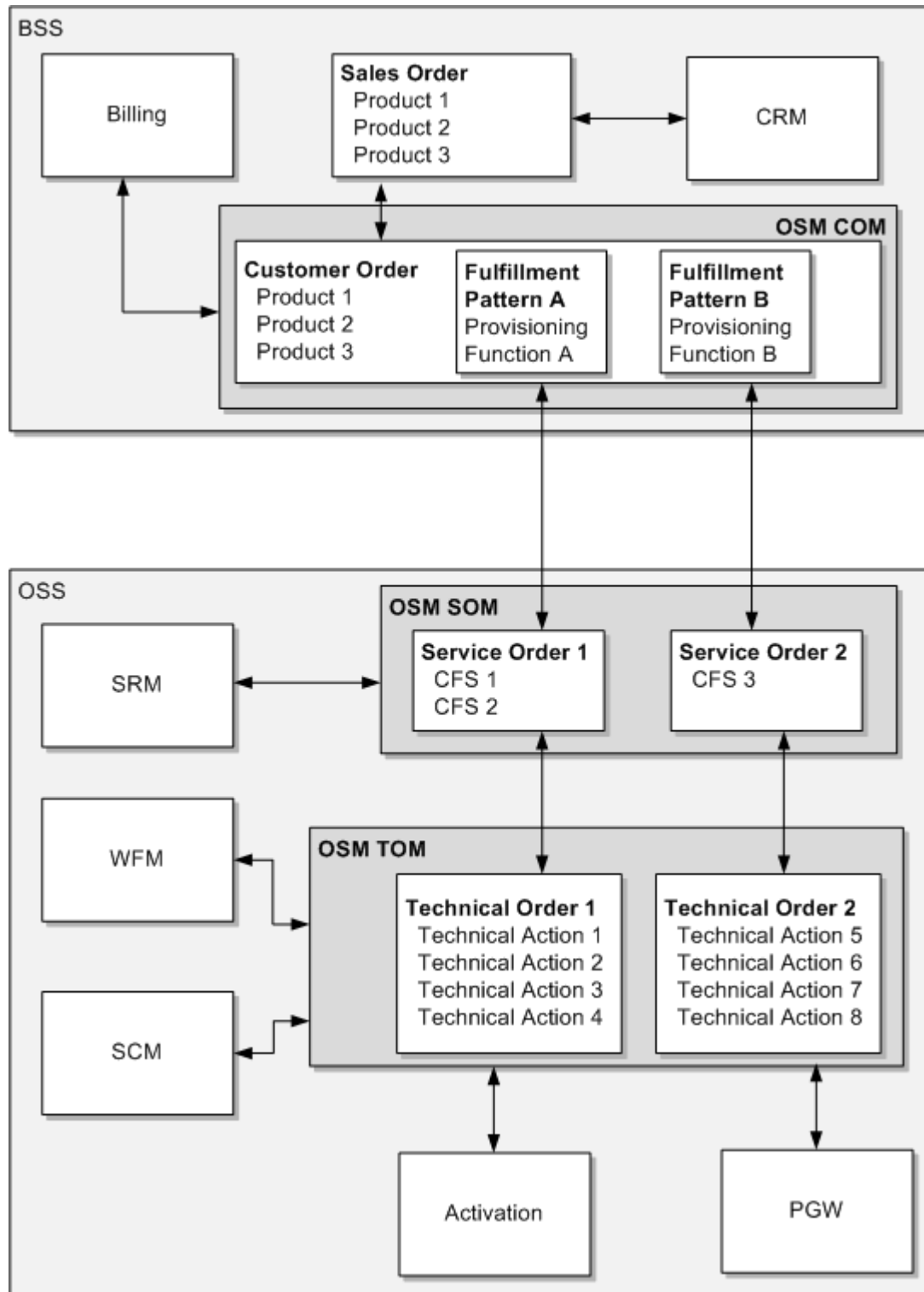
- A revision order can be submitted with corrected data. A revision order may be required between CRM systems and OSM in the central order management (COM) role, for example, if the original order contained faulty data.
- Order management personnel can edit faulty order data in the Task web client and resume processing of the order by retrying and resolving the order or tasks.
- Order management personnel can use a job control order to perform several functions in sequence such as updating faulty data on multiple orders and tasks and then retrying or resolving failed tasks in each order.
- The order can be terminated and a new order submitted.
- **Escalation:** If a particular instance of OSM is unable to resolve an issue and must escalate the problem to the upstream order and system, such as between OSM technical order management (TOM) and service order management (SOM), OSM SOM and COM, or between COM and the originating CRM system. For example, OSM TOM receives a technical order with incorrect inventory data from OSM SOM. When OSM TOM tries to use the data in activation, OSM TOM receives an error and must escalate the problem back to OSM SOM.

Understanding Fallout Across OSM Roles

[Figure 14-1](#) shows how OSM in the COM, SOM, and TOM roles processes a single sales order from a CRM system. The sales order generates the following hierarchically related orders:

- One customer order in the OSM COM role. The customer order contains three order items that decompose into two fulfillment patterns. The customer order is the parent order of all other orders sent to OSM in the SOM role.
- Two service orders in the OSM SOM role. The two fulfillment patterns from the COM role generate the two service orders that OSM in the SOM role processes. These two orders are sibling orders, typically related to each other through reference number. The service orders are the parent orders of all other orders sent to OSM in the TOM role.
- Two technical orders in the OSM TOM role. SOM sends two separate technical orders to OSM in the TOM role also related to one another by reference number.

Figure 14-1 Order Processing Across OSM Roles



The total number of orders generated from the one CRM sales order is five. Ancestor orders are completed when descendant orders are complete. Customer orders are always the first orders created and the last orders to be completed.

Order fallout can occur in any one of these COM, SOM, and TOM orders and in any one order item being processed by an order component instance for an order. Failures can occur in system communication between OSM and the following systems:

- The billing system
- The service resource management (SRM) system
- The workforce management (WFM) system
- The supply chain management (SCM) system
- The activation system
- The partner gateway (PGW) system

You can use order item processing states to help keep track of the processing state of each order item within each COM, SOM, and TOM order that the OSM solution produces. You can configure automation plug-ins and manual tasks to update order processing states when error messages return from external fulfillment systems or when an error condition occurs when personnel are processing manual tasks. See "[Modeling Processing States](#)" for more information.

You can use various notification mechanisms, such as order data change notifications, to send fulfillment state changes about child orders up to parent orders. For example, between TOM and SOM orders, between SOM and COM orders, and all the way up to the CRM system as a trouble ticket if there is a fallout situation that requires changes to the original sales order. See "[Modeling Jeopardy and Notifications](#)" for more information.

If at all possible, it is important to try and resolve errors within the same order and order item hierarchy where the error occurred. However, there are cases where errors can originate from data introduced in other instances of OSM. In such scenarios, it is important to correct faulty data that caused the error at the source. For example, inaccurate inventory data may have been introduced at the SOM level from the SRM system that created a failure at the TOM level on the order component and automation task communicating with the activation system. Although the error can be manually corrected directly on the task communicating with the activation system, allowing the order to continue making progress, this would cause a data inconsistency issue between OSM SOM and the SRM system and OSM TOM and the activation system. In such a case, it is important to define which instance of OSM and related fulfillment systems own the faulty data that causes the fallout in the other OSM instances.

Understanding Fallout Sources

The following sections define typical areas where OSM can experience fallout scenarios.

Managing Business Related Fallout Sources

Business errors can cause problems with downstream systems in the following ways:

- A business error in data generated by OSM such as insufficient or incorrect data can lead to OSM sending an invalid message request to an external system. For example the SRM system generates network information that does not represent the resources in the actual network such as assigning a port that is already in use.
- A business error in the downstream system may occur. For example, an account might be incorrect in the billing system that OSM is communicating with.

The following actions are possible when business errors occur in downstream systems:

- The personnel responsible for the external fulfillment system must notify OSM personnel that the configuration error has been corrected so that the OSM personnel can resolve or retry the task.
- The personnel responsible for the external fulfillment system can resolved the configuration error and can also complete the work that OSM wanted accomplished. In this scenario, the OSM personnel, after having been notified that the work has been completed on the external fulfillment system, can then transition the task to the complete state.

In both these scenarios, OSM personnel are responsible for manually updating any required OSM task data from the fulfillment system if any.

To inform fallout management personnel about a failure in a downstream system, set the failed task to a fallout execution mode. See "[About Failed Tasks and Execution Modes](#)" for more information.

Managing Fallout from Failures in Network or System Resources

OSM typically detects network or system resource problems when automated task plug-ins send messages to external fulfillment systems and receive responses back. Network and system resource problems have to do with software or hardware infrastructure issues unrelated to business configuration or data errors. For example, a power outage may render certain system resources unavailable or the network on which OSM transmits message may experience a failure.

[Figure 14-2](#) shows a normal synchronous message exchange between OSM and an external system. The following lists typical locations where fallout can occur in this exchange:

- The message generated by automation plug-in A-1 may fail to reach the external message queue. This can occur because of a network failure or a middleware failure. In this scenario, OSM performs a rollback of Task A and the failure message returns to the oms_events queue. Oms_events retries Task A up to a predefined limit. When oms_events exceeds the retry limit, oms_events forwards the failure message to the omsErrorQueue. You can configure OSM to automatically transitions the task to a fallout execution mode when an automation plug-in receives a failure exception by selecting the Fail Task on Automation Exception check box in the automation plug-in Details tab. See "[About Failed Tasks and Execution Modes](#)" for more information.

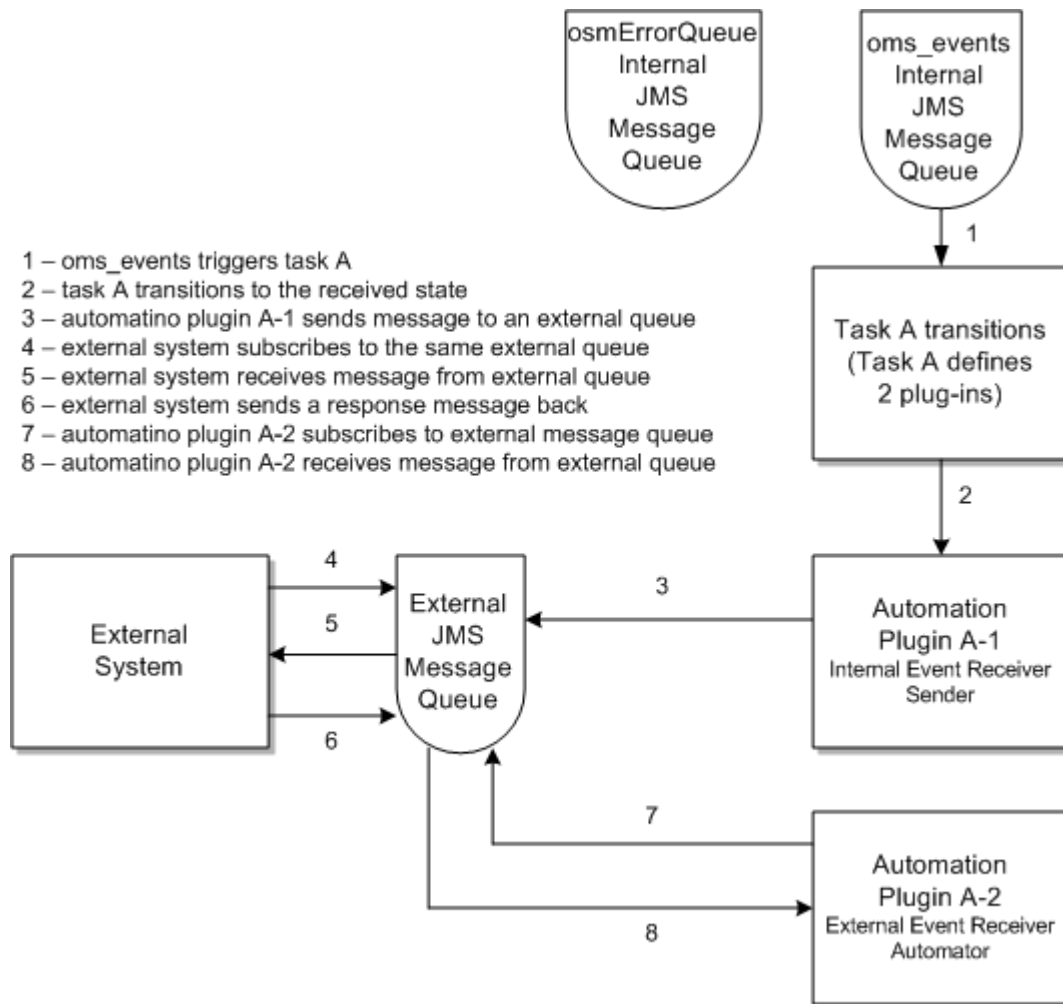
Note

The default settings for oms_events are 15 retries with a 10 second delay between attempts. Do not change these queue settings because OSM relies on this queue for internal order processing. For more information about the oms_events queue, see *OSM Developer's Guide*.

- The message generated by automation plug-in A-1 reaches the external system, but the external system does not respond. This can occur because of a middleware failure, because the external system is unavailable, or because the external system is too busy to respond. To deal with such scenarios, you can configure jeopardy notifications on the automated task to trigger after a specific duration. The jeopardy can run an automation plug-in that can perform a variety of tasks such as transitioning the task to a fallout execution mode, sending a notification to a fallout specialist to manually investigate the problem on the external system, retrying the task, and so on.
- The message generated by automation plug-in A-1 eventually produces a response from the external system. Although the process may be delayed, such problems can be ignored

unless the problem occurs on a regular basis causing performance issues. At this point, further investigation is required on the external system to determine the cause of the delay.

Figure 14-2 Automation Flow: Simple Synchronous



Managing Fallout During Order Creation

The following failure scenarios can occur when using the CreateOrder web service operation:

- Failure to recognize the order. To resolve order recognition failures, model a catch all order recognition rule for such orders.

An order that fails to be recognized by any recognition rule is rejected by OSM and lost. No record of it is sent to the order-source system. To make sure that all orders are captured in OSM, create a recognition rule that accepts all incoming customer orders. Prioritize it at the lowest level (0) and prioritize all other recognition rules higher so they are processed first. Using this lowest-level recognition rule, an invalid order is recognized, and then it fails during validation. It then transitions to the Failed state and is kept by OSM.

- Recognition rules are global entities. An incoming customer order could be recognized by a recognition rule deployed in the system that you did not intend to be matched if you are not careful with the relevancy settings and the recognition rule.

- The order recognition rule accepts the order but a validation rule error occurs. For example, a missing field on the order. The order fails, and the original incoming customer order is attached. You can publish an event based on order failure.
In the case of validation errors, revise the order request and resubmit it.
- The order recognition rule accepts the order but a transformation rule error occurs. For example, where the data you are transforming does not match the creation task. In the case of transformation errors, troubleshoot and fix the transformation logic, and resubmit the order.
- Failure due to incorrect authentication credentials. In such cases, verify that the credentials are still valid and the account has not been locked out.
- Failure to create the correct control data for the orchestration plan. In such cases, verify how the cartridge is modeled and the XQuery expressions involved in generating an orchestration plan.
- Failure when the CreateOrderBySpecification web service operation is used, usually because the input data is not valid or permissions are not correctly set. The error response can be:

- InvalidOrderSpecificationFault
- InvalidOrderDataFault

The error response includes error details.

If either of these two faults is returned, revise the order and resubmit it.

- The order is a revision order, and the point of no return based on order state transition has been reached on the base order. In this case, TransactionNotAllowedFault is returned. If you have configured a follow-on order for this scenario, you can submit the follow-on order. Otherwise, you can submit a new order.

You can specify to display a message in the Task web client and the Order Management web client if an order fails during order recognition rule validation and transformation. To do so, specify the fail-order reason when you model the recognition rule in Oracle Communications Service Catalog and Design - Design Studio. In addition, you can configure any validation rule error (returns a non-true response) in the Order Management web Client.

OSM sends an exception response to the sender if an order creation failure occurs.

Managing Fallout in the OSM Web Clients

You can use both the Order Management web client and the Task web client to manage order fallout.

- You typically use the Order Management web client to search for orders with warning and failure order item processing state, failed tasks, or failed orders. You suspend, resume, cancel, retry, resolve, fail, or terminate an order in the Order Management web client. You can also run these operations as job control orders for groups of orders. See "[Managing Fallout in the OSM Order Management Web Client](#)".
- You typically use the Task web client to run fallout management operations within tasks running in a fallout execution mode. You can fail, resolve, retry, and raise exceptions on manual tasks in the Task web client. You can also suspend, resume cancel, retry, resolve, fail, or terminate an order in the Task web client.

Both clients can be used for fallout management, but the primary differences are:

- You use the Order Management web client to search for orders, order items, and tasks that have failed based on order item processing states, fulfillment states, and failed task

execution modes. The View Faults search page is particularly useful in this regard. You can view the problem that is causing the order to fail, but you cannot resolve the order failure by changing order data in the Order Management web client.

- You use the Task web client to manage problems with tasks and processes; for example, you can manage failed orders by working on tasks running in a fallout execution mode. You can change order data that may resolve the order failure. You can manually trigger a fallout exception.

Each client can launch the other client when required. To learn more about navigating between the clients so you can quickly access the orchestration view and task-level view of an orchestration order, see the getting-started discussions in each web client's user guide.

Modeling Fallout in Tasks

The following sections describe how to model order fallout in tasks.

About Failed Tasks and Execution Modes

OSM manual and automated tasks include execution modes for normal forward processing operations and change order management operations. OSM manual and automated tasks also include these operations in fallout execution modes that you can assign to fallout workgroups with responsibilities for troubleshooting failed tasks.

Fallout execution modes allow:

- Separate fallout workgroup (roles) can be associated with a task that has failed. Fallout users associated with the fallout workgroups can then receive and be assigned to the failed tasks. This is important, for example, if you have a dedicated team of fallout specialists who constantly monitor orders and tasks for fallout. Having a fallout workgroup associated with the task that failed means that these fallout specialists have direct access to the task that generated the failure.
- Visibility of failed tasks in the OSM Order Management web client and the OSM Task web client.
- Avoid additional modeling. Although you can create separate fallout tasks to handle fallout scenarios, modeling fallout on the original task using fallout execution modes helps you avoid additional modeling.
- Run recovery operations in the OSM Clients, such as Retry Order, Resolve Order, Retry Task, and Resolve Task and corresponding OSM Web Service API and XML API. You can also run many of these recovery operations as job control orders to correct failures in bulk.
- Failures that occur in amending states can be detected and managed. For example, a failure during cancelation can be corrected so that the cancelation process can continue or a failure during revision can also be corrected so that the revision process can continue.
- You can specify query tasks and roles to specify the data available to fallout managers.
- Automation plug-ins to run on tasks that have failed if they are configured to run in the corresponding fallout execution mode.

You can model automated tasks to transition to a fallout execution mode based on error messages received from downstream systems by using the OSM Java API

TaskContext.failTask or **TaskContext.failTaskOnExit** methods in the automation plug-in code. You can also use the Task web client to fail manual tasks causing the manual tasks to transition to a fallout execution mode. See *OSM Task Web Client User's Guide* for more information.

About Alternate Task Fallout Management Methods

Alternate fallout management methods that OSM supports for backward compatibility include:

- Set the task to a user-defined failed state. The order remains in the In Progress state, and other tasks can still be carried out while the recovery is managed. To correct the problem, you can manually complete the task or reset the state of the task to Received, which retries the task.

This method requires additional modeling of task states and when the task is in the failed state, the other states cannot be used at the same time. This means that plug-ins cannot run against tasks that have failed, users cannot be assigned to failed tasks, and there cannot be a separate fallout workgroup associated with the task in the user defined failed state.

- Transition the task to a manual fallout recovery task using a task status transition. This provides a specific set of data that applies to redoing the task. You can then use a status transition from the recovery task to the failed task to retry the failed task. This option can cause data consistency issues because it requires the order manager to maintain data integrity within OSM instead of allowing OSM to handle the data changes through compensation.

This method requires additional modeling so that every task that can have a failure requires a status transition from the failed task to a recovery task. In addition, when compensation scenarios occur, the recovery tasks may be included in the compensation plan even though the recovery task is no longer required.

Note

Oracle recommends using fallout execution modes instead of these alternative task fallout methods. For more information, see "[About Failed Tasks and Execution Modes](#)".

Modeling Task Notifications for Fallout

Within the Design Studio automated or manual task editors, you can model jeopardy notifications that send email, display jeopardy notifications in the Task web client, and trigger automation plug-ins when the order is exceeding a specified duration or pass rules that evaluate them. For example, you can configure a jeopardy to run an automation plug-in when there is a problem in the network or system resource that OSM is communicating with and no response has returned within the specified time. See "[About Jeopardy Notifications](#)" for more information about task jeopardy notifications.

About Modeling Fallout Exceptions

You can designate parameters that can potentially contain problematic or inconsistent data as fallout data that you can use to trigger a fallout exception in the Task web client. This functionality causes compensation to occur from the point where the data was introduced. This functionality should only be used from well-known points where such problematic data can be generated.

For example, data received from an SRM system can sometimes be faulty and not reflect what is in the network. In this case, the data received from the SRM system can be designated as having the potential to trigger a fallout exception. When a task attempts to use the data to send

an activation request to an activation system, the activation system returns an error message that causes the task to raise the fallout exception. OSM then calculates what compensation is required up to the point where the data was introduced at the SRM system. As part of the compensation, the task that originally communicated with the SRM system runs in the redo execution mode and the SRM system returns corrected data. All tasks between the task that communicated with the SRM system and the one that sent the faulty data to the activation system are compensated accordingly.

A similar scenario may involve OSM in the SOM and TOM roles as illustrated in [Figure 14-1](#). In this scenario, the faulty data may have been introduced by the SRM system to OSM in the SOM role, but the data only triggers a fallout exception in OSM in the TOM role. In this case, OSM TOM traces the fallout exception back to the creation task of the TOM order. OSM transitions the TOM order to the Waiting for Revision state. At this point, the problem must be escalated back to the OSM SOM system and the parent service order that generated the technical order.

In addition to these data-centric examples, you can also use fallout exceptions in well-defined points during order processing where errors can occur, although the errors are not tied to specific data and the resolution could involve reverting back to a specific point in order processing. For example, you could designate a parameter called point A at task A as fallout data that would allow you to trigger compensation back to task A from any task after task A that contains the parameter point A. It is not the data on point A that causes the error, but you can use point A to revert back to task A.

You can configure fallout entities in Design Studio to specify the data that you want to display in the Order Management web client. To do so, when modeling an order, create a fallout entity and include it in the order model. A fallout entity includes one or more data elements that you want the Order Management web client to display.

[Figure 14-3](#) shows a fallout configured in OSM. In this example, the fallout is named PortAlreadyAssigned. It is used when a task for activating a service fails because a port was assigned that is not available. The data element is **asdl_service_details/port_id**.

Figure 14-3 Fallout Defined in an Order

The screenshot shows the 'Order : add_adsl_siebel' configuration page. The 'Display Name' is 'Add ADSL Siebel'. The 'Fallouts' section is active, showing a list of fallout entities. The 'PortAlreadyAssigned' entity is selected. Its 'Display Name' is 'Port Already Assigned' and its 'Nodes' are '/adsl_service_details/port_id'. The 'Nodes' field is a table with one row and one column.

Nodes
/adsl_service_details/port_id

Buttons for 'Rename', 'Remove', and 'Add...' are visible. The 'Fallouts' tab is selected in the bottom navigation bar.

After you configure fallouts in the order specification, you can assign those fallouts to manual tasks that need them. This association enables OSM to identify the task that generated the error, transition the order to the Amending state, and initiate amendment processing.

To resolve fallout, OSM follows the same process as when it performs amendment processing: It builds a compensation plan, and then applies the required changes.

Fallout can be triggered based on a single incorrect field in a single task. Because fallout can be mapped to one or more data elements, it is possible to have multiple errors in a single task view.

You can also create fallout groups to simplify assigning fallout data to orders. A fallout group is a group of fallout specifications, each of which includes a set of data to display in the Order Management web client. This enables you to review multiple fallouts together in the Order Management web client when the corresponding types of fallout occur.

To trigger fallout in an automated task, use the XML API `FalloutTask.Request` through `com.mslv.oms.automation.OrderContext.processXMLRequest`.

Managing Fallout Exceptions in the Task Web Client

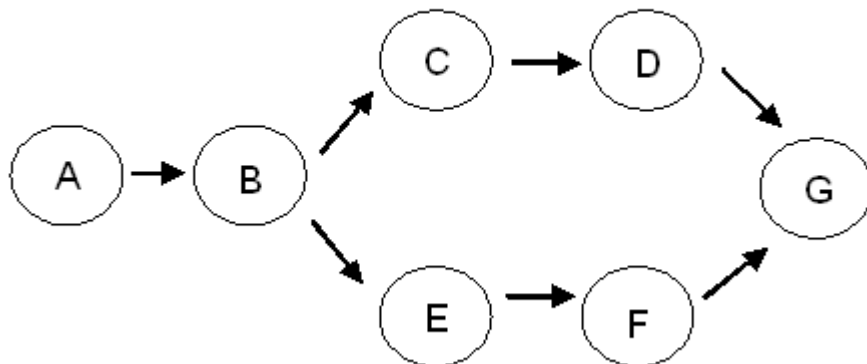
You can initiate fallout in the Task web client by raising an exception. An **exception** is a mechanism used to interrupt or stop an order or to redirect it to any task in the same process or any other process. You can use two types of exceptions: process exceptions and fallout exceptions.

You can use a **fallout exception** to initiate fallout to correct an error. A fallout exception allows you to initiate fallout from a particular task to correct an error caused by a previous task. When you raise a fallout exception, the system identifies the task that generated the error, transitions the order to the Amending state, and initiates amendment processing.

To recover from order fallout, the order might require a revision order to redo some of the order processing. [Figure 14-4](#) shows how the system manages compensation tasks due to fallout.

In this scenario, Task B is responsible for the error and Tasks C and D include the error data. The fallout exception is raised at Task G.

Figure 14-4 Order Fallout Corrected by a Revision Order



In this figure:

1. An order is processed using the above workflow following the path A, B, C, D, G.

2. A fallout exception is raised at Task G.
3. OSM determines that Task B first output the error and initiates amendment processing as follows:
 - **Same branch:** If, during the redo processing of Task B, the task completes with the same completion status as it did in normal processing, subsequent Tasks C and D are also redone and the flow is complete.
 - **Different branch:** If, during the redo processing of Task B, the task completes with a different completion status causing Task E to be the next task, the obsolete branch of Tasks C and D must be undone and the new branch of Tasks E and F must be done while still in the Amending state.

① Note

If the error data was generated by the creation task, the order transitions to the Waiting For Revision state. No compensation tasks are created and the order must be corrected through a revision order.

You can use a **process exception** to stop or redirect an order. Process exceptions are typically part of the configured order flow and can be used to manage the order manually.

① Note

Process exceptions is deprecated. Exception processes are incompatible with OSM's built-in compensation functionality. An order for which an exception process is configured cannot accept revisions, cancellations, or fallout.

Simplified Fallout Exception Automation Framework (Cloud Native Only)

The OSM automation framework provides cartridge automation plugins with an interface to raise fallout exceptions. OSM enriches fallout exception to include execution context (OSM Order Id, histId, run time task instance identifier, order component key, and so on). The interface provides generic APIs for the cartridge's automation plugin to create fallout exceptions and enrich them with any additional data available to the cartridge via "fallout exception attributes".

① Note

Automation tasks that can throw fallout exception should have the **Assigned** state enabled.

Using FalloutContext API in XQuery Plugin

Fallout Context is created by OSM Automation and is bound to scripted plugins such as the XQuery plugin via OrderContext. The following snippet shows how XQuery plugin uses the FalloutContext interface to create and shape up a fallout exception:

```
declare namespace affectedLines="java:java.util.HashSet";
declare namespace throwable="java:java.lang.Exception";
declare variable $cause := throwable:new();
context:createFalloutException($context, 'Detailed fallout message example',
'test','test', $cause ),
context:emitFalloutException($context)
```

Note

The FalloutContext Java object lifecycle is limited to the duration of the script execution. Fallout Exception is created in memory and used when the automation framework is triggered to emit the fallout exception. Only one fallout exception can be created via a single execution of a plugin. Hence, calling on the **createFalloutException** method repeatedly only replaces the previously created object.

Fallout Exception Attributes and Emit-fallout context class interface

The methods for adding custom attributes to the fallout exception are as follows:

```
/**
 * Allows scripted plugin to create Fallout Exception.
 * <p>
 * Example usage in XQuery:
 *
 * <pre>
 *   declare namespace affectedLines="java:java.util.HashSet";
 *   declare namespace throwable="java:java.lang.Exception";
 *   declare variable $cause := throwable:new();
 *   context:createFalloutException($context, 'Detailed fallout message
example', 'test','test', $cause ),
 *   context:emitFalloutException($context),
 * </pre>
 *
 * @param message
 * @param location
 * @param category
 * @param cause
 * @since OSM 7.4.2
 */
default void createFalloutException(final String message, final String
location, final String category,
    @Nullable final Exception cause) {

}
/**
 * Emits the fallout exception that has been created via FalloutContext
when automator execution is complete.
 */
default void emitFalloutException() throws AutomationException{
```

```
    }  
}
```

where:

- The `createFalloutException` method is designed for creating a custom exception related to fallout exceptions. This method allows you to construct and throw an exception specific to fallout situations with the ability to provide detailed information about the event. The method is void, meaning it does not return a value, but is responsible for raising the exception with the provided details.
- `message` represents a message or text that provides information about the fallout situation. It could be a description of what is happening, any instructions, warnings, or any other relevant details. For example, "System is not active at the moment" could be a message that informs someone about the fallout.
- `location` is a free form field which contains details about the source of the fallout. For example, Billing System or SOM.
- `category` is used to categorize or classify the type of fallout exception. It helps provide context about the nature of the situation. For example "inventory issue", "activation error". A developer uses the appropriate one for each fallout exception they want to raise.

For details about the Fallout Exception Management REST API, see *REST API Reference for Oracle Communications Order and Service Management Cloud Native*.

Modeling Fallout in Orders

The following sections describe how to model order fallout in orders.

Modeling the Failed Order State

When an order fails and you want no further progress to occur on the order, the Fail Order transaction transitions the order to the Failed state. You can then resolve the problem in the downstream system. When the problem is corrected, reset the status of the order to In Progress by using the `ResolveFailure` web service operation.

This method should only be used when no further processing is possible on the order and failing a task to a fallout execution mode is not sufficient to correct the problem. Solution developers must consider that large orders with many concurrent order components and tasks completely stop.

You can trigger an order transition to the Failed state from the following states:

- Not Started
- In Progress
- Waiting for Revision
- Suspended
- Waiting

When the problem is fixed, the order can be moved out of the Failed state as follows:

- If the order was failed from the Not Started, In Progress, Waiting, or Waiting for Revision states, the Manage Order Fallout transaction moves the order back to the state it was in before being failed.
- If the order was failed from the Suspended state, the order is transitioned back to the Suspended state.

If the order needs a revision to be fixed, the Submit Amendment transaction places the order in the amendment queue, after which the Process Amendment transaction transitions the order to the Amending state. A revision can come from two sources:

- The originating order-source system can enter a revision order.
- A process exception, which includes redo and undo operations, can run.

If the order must be restarted, the Cancel Order transaction transitions the order to the Cancelling state, and then to the Cancelled state. This operation undoes all changes and returns the order to the creation task.

If the order has an orchestration plan, it cannot be restarted after being canceled. The Cancelled state is a final state for orders that have an orchestration plan.

See "[Modeling Order Life-Cycle Policy States and Transitions](#)" for more information.

Modeling Order Notifications for Fallout

Within the Design Studio Order editor, you can model jeopardy notifications that send email, display jeopardy notifications in the Task web client, and trigger automation plug-ins when the order is exceeding a specified duration or pass rules that evaluate them. For example, you can configure a jeopardy to run an automation plug-in when the order is taking too long to complete. See "[About Jeopardy Notifications](#)" for more information about task jeopardy notifications.

You can create automated fallout messages based on the **exception** order milestone in the Order editor, Events tab. The exception milestone is triggered when an order moves from the In Progress state to the Amending state using the **Raise Exception** transition. The Raise Exception transition is triggered whenever an operator initiates a fallout exception from a manual task or whenever an automation plug-in triggers a fallout exception. See "[About Using Order Milestones to Trigger Event Notifications](#)" for more information.

[Figure 14-5](#) shows an automation plug-in configured to run when the exception order milestone occurs.

Figure 14-5 Exception Order Milestone that Triggers Fallout Automation Message

The screenshot displays the 'Order : OsmCentralOMExampleOrder' window. The 'Display Nam' field is set to 'OsmCentralOMExampleOrder'. The 'Order Milestone Automation Events' section shows a table with the following data:

Milestone	Automation				
exception	<table border="1"> <thead> <tr> <th>Name</th> <th>Automation Type</th> </tr> </thead> <tbody> <tr> <td>TroubleTicketAutomator</td> <td>xQuerySender</td> </tr> </tbody> </table>	Name	Automation Type	TroubleTicketAutomator	xQuerySender
Name	Automation Type				
TroubleTicketAutomator	xQuerySender				

Below the table are 'Remove' and 'Add...' buttons. The 'Properties' button is also visible. The bottom navigation bar includes tabs for 'Details', 'Amendable', 'Rules', 'Fallouts', 'Fallout Groups', 'Jeopardy', 'Notification', and 'Events'. The 'Events' tab is selected, showing a 'Properties' window for the 'Automation Plug-in - XQuery Sender'.

The 'Automation Plug-in - XQuery Sender' window has tabs for 'Details', 'XQuery', 'Routing', and 'Information'. The 'Details' tab is active, showing the following fields:

Name	TroubleTicketAutomator
EJB Name	OsmCentralOMExampleOrder.TroubleTicketAutomator
Run As	%{DEFAULT_AUTOMATION_USER}

In addition, you can use order data change notifications in the Order editor Notifications tab to generate messages to users that are members of specific workgroups (roles), display the messages in the Task web client, and trigger automation plug-ins whenever a specific data field changes. For example, you can configure OSM to communicate order fulfillment state or processing state changes of each order that is part of the order fulfillment process running in COM, SOM, and TOM OSM roles including order item failure and warning states. Whenever a TOM order item fulfillment state or processing state changes, you can use an order data change notification to communicate the change to the OSM SOM instance that generated the TOM order. See "[About Using Order Data Changes to Trigger Notifications](#)" for more information.

About Terminating an Order

If the order fallout cannot be resolved by any other means, the Abort Order OSM Web Service operation transitions the order to the Aborted state. In addition, you can terminate an order from the Order Management web client which also transitions the order to the Aborted state.

After terminating the order, you can resubmit the order from the order-source system. Only use this method when all other approaches are impossible.

Managing Fallout in the OSM Order Management Web Client

In the Order Management web client, you can search for faults using the View Faults search page. You can search for failed orders based on the location of the source of the fault. Fault source types can be:

- On the Order when an order transitions to the failed state.
- On an order item when the order item transitions to a failed order item processing state.
- On a task when a task transitions to a fallout execution mode.

From the View Fault search page Results area, you can:

- Select the failed order to view order and order item details in the Order Details Order, Order Items, and Order Components tabs.
- Select tasks running in a fallout execution mode to view and troubleshoot the task in the Task web client.

From the Manage Orders search page and results area, you can:

- Find orders based on whether they have a failure at the task or order level, whether they are running in the failed state, whether order tasks are running in a fallout execution mode, and so on.
- Run operations such as retry and resolve failed orders to retry or resolve all failed tasks within the order.
- Run operations on multiple orders as a job control order.
- View details about individual failed order to determine why it failed.

From within the Order Details page, you can also run actions on an individual order and on the Order Details Order Component tab, you can run retry and resolve actions on all tasks within individual order components. In addition, in the Order Details page Order Components tab, Running & Failed Tasks subtab, you can view all failed tasks and retry or resolve each task individually.

To correct the error that caused the failure, you often must use the OSM Task web client to work on tasks in fallout execution modes. You might also need to work with external systems. There is no functionality in the Order Management web client to manually edit tasks.

Note

If the order failed because of a recognition rule failure or after reaching its point of no return, it cannot be resolved. Also, the ability to suspend, cancel, or terminate an order depends on its life-cycle policy.

If you cannot resolve the order or task failure, you can use the Order Management web client to cancel or terminate the order:

- Canceling an order immediately stops its processing and sets the order state to Canceled. Any tasks that have already completed for the order are rolled back. If the order has an orchestration plan, the order cannot be resumed. If the order does not have an orchestration plan, it can be resumed.

- Terminating an order immediately stops its processing and sets the order state to Aborted. The order cannot be resumed. Unlike canceling an order, terminating an order does not roll back any tasks that have already completed. As a result, clean-up may be required.

Note

Consider the impact on other systems of canceling or terminating orders. Depending on how your solution is configured, upstream systems may not be aware that an order has been canceled or terminated.

You can also use the Order Management web client to fail an order manually. Failing an order stops its processing and sets its state to Failed. It is not possible to change the state of a failed order or to make other changes until you resolve the order failure. Orders you fail manually are treated the same way as orders that are failed automatically by the system. They are considered fallout.

Note

In most environments, fallout-handling rules detect processing problems and automatically fail orders. Manually failing orders is not normally required. There may be some situations and environments when it is necessary to manually fail orders, however.

Make sure you understand how other systems in your order processing solution handle failed orders. Depending on how your solution is implemented, upstream systems may not be aware that an order has been manually failed.

Modeling Fulfillment States and Processing States

This chapter describes how to model fulfillment states and processing states in an Oracle Communications Order and Service Management (OSM) solution.

About Fulfillment States, and Processing States

You can associate the predefined order component order item (OCOI) processing states to messages from external fulfillment systems or to events that occur within OSM to generate an aggregate order item processing state as an order processes. OSM then uses these OCOI processing states to calculate You can also define fulfillment states for orders and order items to report on different business scenarios.

[Table 15-1](#) compares processing states with fulfillment states.

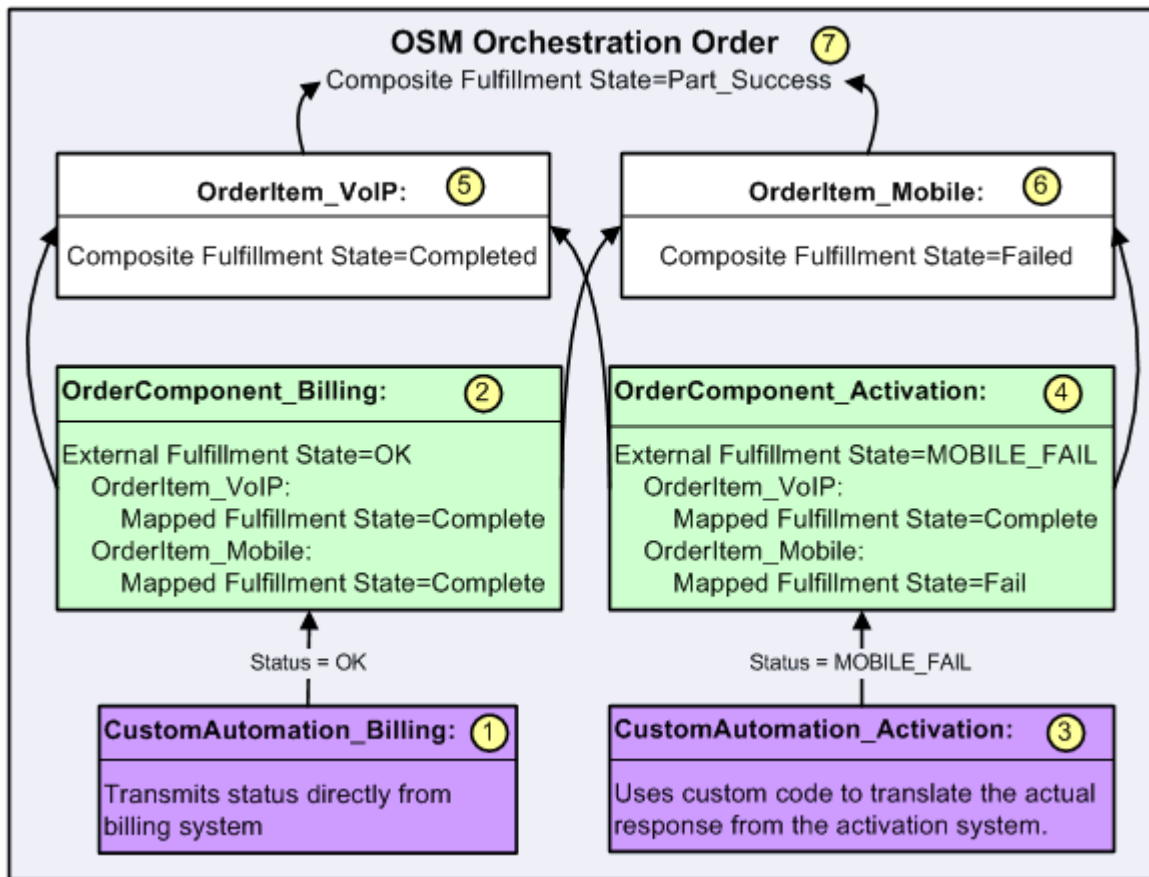
Table 15-1 Comparing Processing States with Fulfillment States

Features	Processing States	Fulfillment States
Auto-configured and predefined	Yes	No
Manually configured and defined	No	Yes
Tracked as normal, warnings, or failures counts in the Order Management web client Order tab, Summary subtab.	Yes	No
Tracked as warning or failure counts in the Order Management web client Order Items tab.	Yes	No
Failure states in the Order Management web client can be traced: - From the order item - To the order components processing the order item - To the tasks processing the order items in each order component. Identifying when a task generates a failure state is easier when the task also transitions to a fallout execution mode.	Yes	Yes
Updated for each order item in the Order Management web client Order Items tab.	Yes	Yes
States reflected up the order item hierarchy	Yes	Yes
States reflected on orders based on order item hierarchy states	No	Yes

Modeling Fulfillment States

[Figure 15-1](#) is a detailed depiction of fulfillment state processing for a small part of a sample implementation. It shows the way multiple external responses can be translated into a single fulfillment state for the order.

Figure 15-1 Fulfillment State Composition



At run time, OSM maps the external fulfillment states to mapped fulfillment states on an order item. Order item fulfillment states are composed using the immediate children of the order item, and order fulfillment states are composed using the root-level order items.

Whenever one of the input fulfillment states for an order item changes, the fulfillment state of that order item (and all of its parents, including the order) is recalculated. For example, if the mapped fulfillment state of "leaf" order item A changes, the composite fulfillment state of order item A is recalculated. If the composite fulfillment state for order item A changes and it has a parent, order item B, order item B's fulfillment state is recalculated as well. If the composite fulfillment state of order item A does not change, the fulfillment state for order item B is not recalculated.

In the figure:

1. The external billing system sends a status of OK, which is used directly as the external fulfillment state for OrderComponent_Billing.
2. The external fulfillment state of OK is mapped to Complete for both the order items that are fulfilled by that order component (OrderItem_VoIP and OrderItem_Mobile) using the fulfillment state mappings.
3. The activation system has sent a complex message indicating the statuses of different parts of the fulfillment request. That message is translated by the custom code in the automation to the external fulfillment state of MOBILE_FAIL.

4. The fulfillment state mappings are configured to map `MOBILE_FAIL` for this order component to mean that `OrderItem_Mobile` has failed and `OrderItem_VoIP` has succeeded.
5. The fulfillment state composition rules for the `OrderItem_VoIP` order item then look at the mapped fulfillment states for `OrderItem_VoIP` for each order component (`OrderComponent_Billing` and `OrderComponent_Activation`) that fulfills that order item. Because the mapped fulfillment states for both of the order components are `Complete`, the composite fulfillment state for the order item is also set to `Completed`.
6. The fulfillment state composition rules for the `OrderItem_Mobile` order item then look at the mapped fulfillment states for `OrderItem_Mobile` for each order component (`OrderComponent_Billing` and `OrderComponent_Activation`) that fulfills that order item. Because the mapped fulfillment states for one of the order items is `Complete` and for the other order item is `Fail`, the composite fulfillment state for the order item is set to `Failed`.
7. The fulfillment state composition rules for the order then take the composite fulfillment state of the highest-level parent order items to determine the fulfillment state of the order. In many cases, the failure of any part of an order might be configured as a failure of the order as a whole. However in this example, fulfillment states have been configured that, because part of the order (VoIP) is ready for customer use, the composite fulfillment state is set to `Part_Success`.

Defining Fulfillment States

Fulfillment states are configured in Oracle Communications Service Catalog and Design - Design Studio. At a high level, configuration of fulfillment state management has the following main steps:

1. Define external fulfillment states for order components: Create a list of values for the order component that matches the statuses returned by the external systems or automations. An external fulfillment state is available on the order component where it is defined and on any order component that extends that order component. See "[Modeling External Fulfillment States](#)" for more information.
2. Create and configure fulfillment state maps: Create one or more lists of values for the common fulfillment states and create mappings to translate external fulfillment states into mapped fulfillment states. **Common fulfillment states** are used as mapped fulfillment states and as composite fulfillment states. Fulfillment state mappings provide the evaluation and normalization of the external system's states into mapped fulfillment states. Common fulfillment states and fulfillment state mappings are available for the entire workspace. See "[Modeling Fulfillment State Maps](#)" for more information.
3. Create and configure order item fulfillment state composition rule sets and order fulfillment state composition rule sets: Create the composition rule sets to determine the fulfillment state of an order or order item from the fulfillment state of its child items. Composition rule sets are based on the order item and order hierarchy, and compose fulfillment states into composite fulfillment states that reflect the state of entire order items or orders. See "[Modeling Fulfillment State Composition Rule Sets](#)" for more information.

The external fulfillment states, order item fulfillment states, and order fulfillment are stored in the `ControlData` for the order. See "[Modeling Data for Fulfillment States](#)" for more information. Mapped fulfillment states are not stored on the order.

Modeling External Fulfillment States

External fulfillment states consist of a list of responses expected by an order component and any order components that extend the order component. When an external fulfillment state is defined, it can be used in a fulfillment state mapping.

Modeling Fulfillment State Maps

You use fulfillment state maps to configure common fulfillment states and fulfillment state mappings. Fulfillment state **mappings** are the entities that contain the actual mapping information, and fulfillment state **maps** are containers for the information. Functionally, it does not matter whether you have one or many fulfillment state maps. Each common fulfillment state is available to all of the fulfillment state mappings, regardless of which fulfillment state map it is configured in. This means that each common fulfillment state needs to be unique in the workspace. There are optional default common fulfillment states that can be used. See Design Studio Modeling OSM Orchestration Help for more information about the default states.

Common fulfillment states have two functions:

- They are used as the result of the fulfillment state mappings. When they are used this way, they are referred to as mapped fulfillment states.
- They are used as the result of the composition rules. When they are used this way, they are referred to as composite fulfillment states. If these fulfillment states are to be sent to an upstream system, you configure these values to match what the upstream system expects. (For more information about composition rules, see "[Modeling Fulfillment State Composition Rule Sets](#)".)

Common fulfillment states, used as either mapped or composite fulfillment states, are configured in a single list in the **States** tab of the Fulfillment State Map editor. You do not need to assign the common fulfillment state as either a mapped fulfillment state or a composite fulfillment state when you configure it. The same common fulfillment state can be used for both purposes at the same time. [Figure 15-2](#) shows the common fulfillment states configured in a fulfillment state map.

Figure 15-2 Detail from Fulfillment State Map Editor States Tab

The screenshot displays the 'Fulfillment State Map : InitiateBilling' editor. At the top, the 'Description' field contains 'Integration' and the 'Namespace' field contains 'http://oracle.communications.centralom'. Below this, the 'Fulfillment States' tab is active, showing a list of common fulfillment states: CANCELED, COMPLETED, FAILED, IN_PROGRESS, and PENDING. Each state is preceded by a red square icon with a white 'O'. To the right of the list, the 'Details' tab is selected, showing 'Fulfillment State: CANCELED' and a 'Description*' field containing 'Canceled'.

After the fulfillment states have been created, you create the mappings in the **Mappings** tab of the Fulfillment State Map editor.

A fulfillment state mapping maps an external fulfillment state to a common fulfillment state. When defining a fulfillment state mapping, you must define when that particular mapping will be used. Each mapping must specify a single fulfillment pattern, order item, and orchestration

sequence, with a single set of orchestration stage and order component combinations. There may be a large number of mappings because wild cards cannot be used.

These criteria are defined in Design Studio and should be specified in the order given. Some of the entries later on the list cannot be set until the earlier ones have been entered.

1. **Fulfillment pattern:** The fulfillment pattern value restricts the fulfillment state mapping to apply only to order components defined on orchestration plans associated with the specified fulfillment pattern. For example, the fulfillment state mappings might be very different between mobile and IP services.
2. **Order item:** The selected value restricts the fulfillment state mapping to apply only to order components responsible for processing the specified order item.
3. **Orchestration sequence:** The available orchestration sequences are those related to the specified order item. The selected value restricts the orchestration stages to which the mapping can apply.
4. **Orchestration stage:** One or more orchestration stages must be specified for the mapping. Any of the orchestration stages in the orchestration sequence can be specified. Use only one orchestration stage per mapping, if possible. Using only one orchestration stage facilitates maintenance of the solution because your decomposition rules may change over time.
5. **Order component:** One order component must be specified for each specified orchestration stage.

You can further restrict the application of the mapping by specifying any of the following:

- **Fulfillment mode:** If specified, the fulfillment mode value, combined with the fulfillment state mapping's fulfillment pattern value, determines the orchestration plan to which the fulfillment state mapping applies. The fulfillment state mapping is evaluated for order components associated only with the identified orchestration plan. The fulfillment state mapping returned for an item with Cancel fulfillment mode could be very different than that for an item with Deliver fulfillment mode.
- **Properties/property value combinations:** After the order item is selected, one or more order item property value criteria values may be specified. The set of order item properties available for selection are those properties that are defined on the fulfillment state mapping's selected order item specification. For example, you might have a property called LineType and have different mappings based on whether the value was VoIP Phone or soft phone.
- **Current Fulfillment State:** If a current fulfillment state is specified, the fulfillment state mapping is evaluated only for those order components where the current fulfillment state of the item on the component matches the specified value. This current fulfillment state is taken from the list of common fulfillment states, meaning that it is the target fulfillment state of another fulfillment state mapping or the result of composition rules. You might use this to set a mapped fulfillment state of Failed if that is the current state; if the current state is In_Progress, the new state might be Complete.

Modeling Fulfillment State Composition Rule Sets

Orders contain one or more order items. Order items can in turn be fulfilled by one or more order components and also contain other order items using the order item hierarchy. See ["Modeling Order Item Hierarchies"](#) for more information.

There is a fulfillment state assigned to the order and order item as a whole that takes into account all of the fulfillment states of its immediate children. This is referred to as a composite fulfillment state.

Fulfillment state composition rules for the order item are defined in order item fulfillment state composition rule sets. These rules aggregate the mapped fulfillment states for any order components that fulfill the order item and also the fulfillment states of any child order items of the order item.

Fulfillment state composition rules for the order are defined in order composition rule sets. These rule sets aggregate the composite fulfillment states of the root-level order items.

Note

To use fulfillment states, you must configure composition rule sets both for orders and for order items.

The configuration processes for order fulfillment state composition rule sets and order item fulfillment state rule sets are similar.

A fulfillment state composition rule set contains rules, which in turn contain conditions, as shown in [Figure 15-3](#).

Figure 15-3 Detail from Order Fulfillment State Composition Rule Set Editor

Order Fulfillment State Composition Rule Set : OrderCom

Description: OrderComposition

Namespace: http://oracle.communications.centralom

Order*: CentralOMManagedServices-OrchestrationOrder

Composition Rules

- IN_PROGRESS_RULE [IN_PROGRESS]
 - ANY_INPROGRESS_CONDITION
 - NONE_FAILED_CONDITION
 - NONE_CANCELED_CONDITION
- COMPLETED_RULE [COMPLETED]
- FAILED_RULE [FAILED]
- CANCELED_RULE [CANCELED]
- PENDING_RULE [PENDING]
- IN_PROGRESS_RULE_2 [IN_PROGRESS]

Rule Details | Rule Information

Rule: IN_PROGRESS_RULE

Description*: IN PROGRESS RULE

Composite State*: IN_PROGRESS

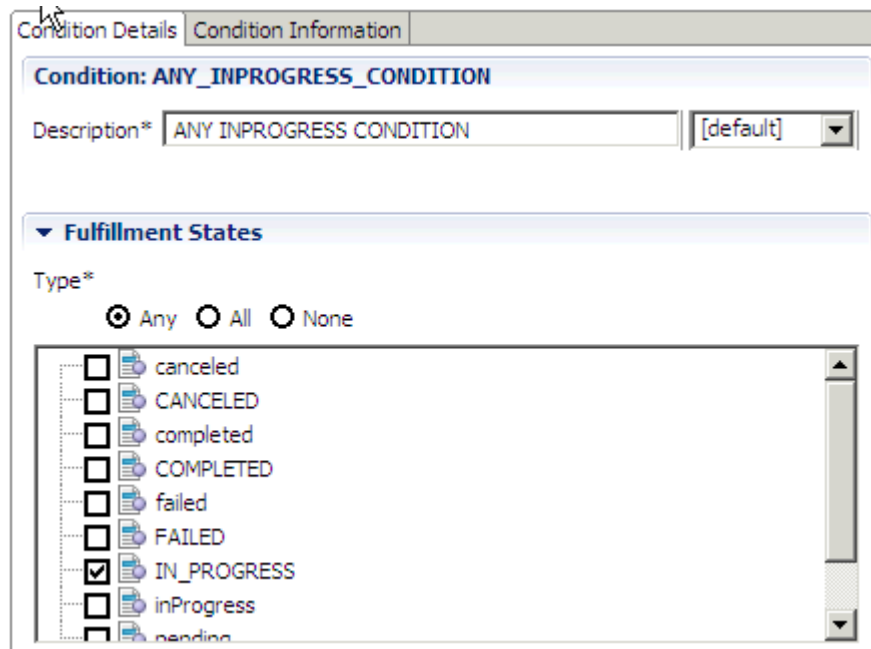
You use composition rules to specify the fulfillment state for the order or order item when all of the conditions are met (logical AND). If there are separate situations that can result in the same fulfillment state (logical OR), create separate rules that evaluate to the same fulfillment state.

For example, say that you have one condition that specifies that all of the input fulfillment states must be FAILED, and another condition that specifies that all of the input fulfillment states must be CANCELLED. Both of these conditions should result in a fulfillment state of NOT_DONE. You also have another condition that allows a mixture of FAILED and

CANCELLED states that should result in a fulfillment state of CHECK_STATUS. In this case you would need three separate rules. The last condition requires its own rule because it results in a different fulfillment state. The other two conditions each require their own separate rule because it would never be possible for both of those conditions to be met at the same time.

The fulfillment state condition based on the input fulfillment states is the same for both order item composition rule sets and order composition rule sets. It allows the inclusion (or exclusion) of one or more fulfillment states according to whether any, all, or none of the input fulfillment states are in a selected list of fulfillment states. [Figure 15-4](#) shows the details for a condition.

Figure 15-4 Fulfillment States Section of Condition Details Subtab



The fulfillment states selected in the condition are constrained by a conjunction that must be true for the condition to evaluate to true. The available conjunctions are:

- **Any:** The condition requires at least one of the input fulfillment states to match one of the selected fulfillment states.
- **All:** The condition requires all of the input fulfillment states to match the selected fulfillment states.
- **None:** The condition requires that none of the input fulfillment states match any of the selected fulfillment states.

The list of fulfillment states that can be assigned as mapped fulfillment states and the list that can be assigned as composite fulfillment states is the same list. The common fulfillment states created in the Fulfillment State Map editor **States** tab apply to both the mapped and composite fulfillment states. Therefore, when you are generating a composite fulfillment state, the list of fulfillment states that you can choose in this condition is the list of common fulfillment states. (See "[Modeling Fulfillment State Maps](#)" for more information about this list.)

Order Item Fulfillment State Composition Rule Sets

In addition to the fulfillment state conditions discussed above, in order item fulfillment state composition rule sets you can set order item property values that must be present for the

composition rule to evaluate to true. If both Any/All/None and property values are defined, both must be true for the composition rule to evaluate to true.

Order Fulfillment State Composition Rule Sets

In addition to the common fulfillment state-related criteria discussed above, in order fulfillment state composition rule sets you can also specify an XQuery expression that must evaluate to true for the condition as a whole to evaluate to true. For example:

```
/GetOrder.Response/_root/OrderHeader/AccountIdentifier > 0
```

This XQuery expression provides the same functionality available to XQuery expressions exposed elsewhere in Design Studio, including access to order data, access to behavior instances, and external configuration.

Modeling Processing States

Order item processing states are a predefined set of states that an order item can enter that derive from a predefined sets of OCOI processing states. Because OSM can process an order item in more than one order component, OSM then aggregates the OCOI values returned from external systems in each order component to determine the overall processing state of the effected order item. You can apply OCOI processing states based on values in response messages from external systems that OSM receives in automated task automation plug-ins or based on direct operator input in manual tasks.

In addition, because order items can be arranged hierarchically, when a child order item processing states changes, OSM also evaluates whether the parent order item should change, and in the same way, if the parent order item is itself the child of another parent order item, OSM evaluates the parent order item when its child order item changes. This process continues up the hierarchy.

The following example shows the Brilliant BroadBand offer and all its descendant order items including their processing states.

Brilliant BroadBand [Add]	InProgressWithFailures
BroadBand Service [Add]	InProgressWithFailures
Basic Internet Access [Add]	In Progress
Internet Media Service [Add]	InProgressWithFailures
Content on Demand [Add]	InProgressWithFailures <--A1=Failed OCOI
Video on Demand [Add]	Not Started
E-Mail Service [Add]	In Progress
Internet 100% TBO [Add]	In Progress
Firewall [Add]	In Progress
Customer Broadband Model	In Progress
Wireless Router	In Progress
Broadband Installation Fee	In Progress
Broadband Activation Fee	In Progress

As this order progresses, one of the order components processing the Content on Demand order item receives a response message at automation plug-in instance **A1**. Based on a value within the response message, the **A1** updates the automation task data with a **Failed** OCOI processing state. This change automatically causes OSM to evaluate all other order components involved in processing the Content on Demand order item and based on this evaluation, assigns the Content on Demand order item with the **InProgressWithFailures** order item processing state.

For more information about OCOI processing states and how OSM aggregates OCOI processing states, see "[Order Component Order Item Processing States](#)".

The change in the processing state of the Content on Demand order item causes OSM to evaluate whether Content on Demand's parent order item, Internet Media Service, also requires an order item processing state change. OSM determines the processing state of the Internet Media Service order item based on its children order items: Content on Demand and Video on Demand. When OSM determines that the Internet Media Service order item also requires an order item processing state change, this causes OSM to further evaluate BroadBand Service and its children (Basic Internet Access, Internet Media Service, E-mail Service, and so on). Likewise, a change in the Broadband Service order item also causes OSM to evaluate Brilliant Broadband based on the processing states of all of its children.

For more information about order item processing states and how OSM aggregates order item processing state changes across an order item hierarchy, see "[Order Item Processing States](#)".

Order Component Order Item Processing States

You can apply OCOI processing states from automated or manual tasks that fall into the normal, warning, or failed categories. These categories impact the overall processing state of an order item (see "[Order Item Processing States](#)" for more information about these categories).

[Table 15-2](#) shows the OCOI processing states and the categories they are included in.

Table 15-2 OCOI Processing States

OCOI Processing State	Category	Description
NotStarted	Normal	Apply the NotStarted OCOI processing state to order items that have not begun processing in an order component. For example, you could create an automated task in the first order component that OSM runs for an order that updates all order items being processed on an order with the NotStarted OCOI processing state.
InProgress	Normal	Apply the InProgress OCOI processing state when the first task in an order component process begins or when tasks within a process resume normal processing, for example, after resolving a failure in a task.
Completed	Normal	Apply the Completed OCOI processing state to indicate that the final task has completed successfully within the order component process. For example, in an automation plug-in, you can update the Completed OCOI processing state in conjunction with the completeTaskOnExit method that completes the final task of the order component process.
Failed	Failure	Apply the failed OCOI processing state to indicate that a failure has occurred in order processing and fallout intervention is required to correct the error. For example, the Failed OCOI processing state could be used in conjunction with the failTaskOnExit method that transitions the task state to the failed-Do execution mode so that an operator can manually troubleshoot the task. A failed OCOI Processing state causes the Failure count to increase by one.
FailedContinue	Warning	Apply the FailedContinue OCOI processing state to indicate a failure condition that does not require fallout intervention to correct the error. The FailedContinue OCOI Processing state causes the Warning count to increase by one.
Undoing	Normal	Apply the Undoing OCOI processing state the entire OCOI is being undone as a result of a revision order or as a result of a fallout exception that triggers order amendment.
UndoCompleted	Normal	Apply the UndoCompleted OCOI processing state when the OCOI is undone.

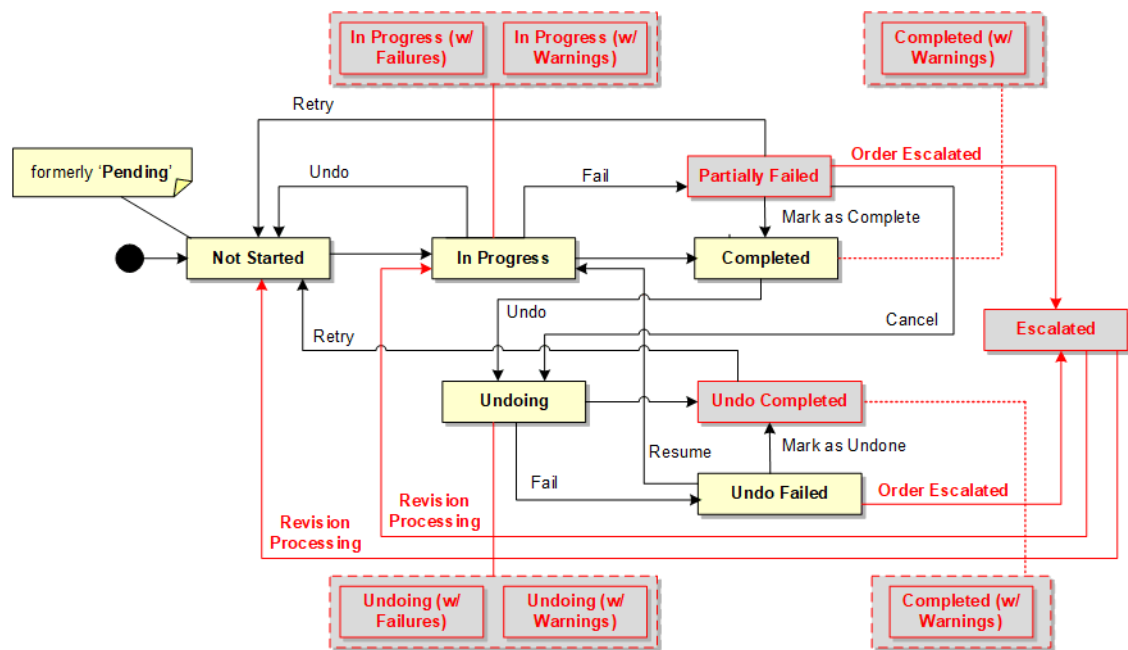
Table 15-2 (Cont.) OCOI Processing States

OCOI Processing State	Category	Description
UndoFailedContinue	Warning	Apply the UndoFailedContinue OCOI processing state to indicate a failure condition that does not require fallout intervention to correct the error. The FailedContinue OCOI Processing state causes the Warning count to increase by one.
UndoFailed	Failure	Apply the UndoFailed OCOI processing state to indicate that a failure has occurred in order processing while undoing the OCOI and fallout intervention is required to correct the error. For example, the UndoFailed OCOI processing state could be used during compensation in conjunction with the failTaskOnExit method that transitions the task state to the failed-Redo execution mode so that an operator can manually troubleshoot the task. An UndoFailed OCOI Processing state causes the Failure count to increase by one.
DownstreamCorrectionRequired	Warning	The downstream system returns a message to an OSM automated task that the downstream system has experienced a failure and is currently working to resolve the problem. This OCOI processing state remains until another response message returns from the downstream system indicating that the problem has been resolved causing the manual or automated task to update the OCOI processing state to Completed or InProgress.
None	Normal	OSM assigns this state automatically if no OCOI processing state has been assigned. You cannot directly use this processing state.

You must write automation plug-in code to map status response values to order component order item processing states. OSM stores order component order item processing state values in the **ControlData** element. See "[About ControlData for Order Component Order Item Processing States](#)" for more information.

Order Item Processing States

OSM evaluates order item processing states differently depending on order item processing directions. If an order item is being fulfilled by OSM, then the order item is operating in the forward direction. If an order item has been removed, for example, in a revision order or because the order itself has been canceled, then the order item is operating in reverse direction.



OSM keeps an overall count of the following processing state categories:

- **Normal:** An order item has a normal category order item processing state when all OCOI processing states from order components processing the order item belong to the normal category or when all the descendant order items of a parent order item belong to a normal category. See [Table 15-2](#) for OCOI processing state categories. The normal count increments by 1 when an order item is processing normally.
- **Warning:** An order item has a warning category order item processing state when one or more OCOI processing states from order components processing the order item belong to the warning category or when one or more of the descendant order items of a parent order item belong to a warning category. See [Table 15-2](#) for OCOI processing state categories. The warning count increments by 1 when an order item contains warnings.
- **Failure:** An order item has a Failure category order item processing state when one or more OCOI processing states from order components processing the order item belong to the failure category or when one or more of the descendant order items of a parent order item belong to a failure category. See [Table 15-2](#) for OCOI processing state categories. The warning count increments by 1 when an order item contains failures.

[Table 15-3](#) shows order item processing states, and the direction and categories they are included in.

Table 15-3 Order Item Processing States

Order Item Processing State	Direction	Category	Description
NotStarted	Forward	Normal	The order item has not begun to process in any order component.
InProgress	Forward	Normal	The order item has begun processing in one or more order components.
InProgressWithWarnings	Forward	Warning	The order item has begun processing within one or more order components, however, one or more of the order components or descendant order items have a processing state from the Warning category.

Table 15-3 (Cont.) Order Item Processing States

Order Item Processing State	Direction	Category	Description
InProgressWithFailures	Forward	Failure	The order item has begun processing within one or more order components, however, one or more of the order components or descendant order items have a processing state from the Failure category.
Completed	Forward	Normal	The order components processing the order item have completed all tasks associated with the order item. All order components processing the order item or all descendant order items or an order item have updated their processing states to Completed.
CompletedWithWarnings	Forward	Warning	The order components processing the order item have completed all tasks associated with the order item; however, one or more of the order components processing the order item or descendant order items have a processing state from the Warning category.
PartiallyFailed	Forward	Failure	All order components processing the order item or descendant order items have returned processing state results, however, one or more have a processing state from the Failure category.
Undoing	Reverse	Normal	The order item has begun processing in the reverse direction in one or more order components.
UndoingWithFailures	Reverse	Failure	The order item has begun processing in the reverse direction within one or more order components, however, one or more of the order components have an OCOI processing state from the Failure category.
UndoingWithWarnings	Reverse	Warning	The order item has begun processing in the reverse direction within one or more order components, however, one or more of the order components have an OCOI processing state from the Warning category.
UndoFailed	Reverse	Failure	One or more order components processing the order item in reverse direction has a failed task that requires fallout intervention to correct.
UndoCompleted	Reverse	Normal	The order components processing the order item in reverse direction have completed all tasks associated with the order item.
UndoCompleteWithWarnings	Reverse	Warning	The order components processing the order item in the reverse direction have completed all tasks associated with the order item; however, one or more of the order components processing the order item or descendant order items have a processing state from the Warning category.
None	Reverse	Normal	OSM assigns this state automatically if no OCOI processing state has been assigned. You cannot directly use this processing state.

OSM stores order item processing state values in the **ControlData** element. See "[About ControlData for Order Item Processing States](#)" for more information.

Modeling Jeopardy and Notifications

This chapter describes how to model jeopardy and notifications in an Oracle Communications Order and Service Management (OSM) solution.

Best Practices for Using Notifications for Status Updates

Status values for an order item and for the whole order often need to be sent to the upstream system that submitted the original request. There are a number of ways to achieve this.

Status Update Strategies

Some common strategies for updating order status are:

- Use an event notification triggered by a change to order data, or when an order reaches an order milestone; for example, completed. The notification runs an automation plug-in that sends a status message to the upstream system. The automation plug-in should have all of the values for status data defined in its view, in order to calculate an aggregated status value.

Be aware that there can be race conditions if multiple status updates are run in parallel. Since each update is taking a snapshot at a particular moment, it is possible that none of the status updates will have a snapshot that includes all of the final values. This strategy is better used when there are no multiple concurrent status updates.

You also can use this strategy in conjunction with fulfillment state or processing states. However, for these two options, the calculation of the aggregated status value is handled by OSM before the event notification is triggered. The event notification can be configured against the order level fulfillment state or order item processing state. In this case there is no race condition as the event is only triggered on the top most data element when it is changed.

- Configure an automation plug-in to generate a status message whenever the order changes state. Because order state changes are generally less frequent than data changes, this may provide better performance.
- It is possible to configure status update functions as order components and make them first-class members of the orchestration plan. However, it is not desirable to do this in most circumstances, because this can quickly lead to a large increase in the number of tasks in the cartridge. If used, this option will work only if status updates are sent at a specific point in the orchestration plan, for example as the last function after provisioning and billing.

Strategies for Using Notifications

Some common uses for notifications are:

- In general, jeopardy notifications are used for alerting order management personnel about something that should have happened but did not happen. By contrast, event notifications are based on events that have happened, and they are used more for communicating status information and for directing the order fulfillment process to the next step.

- Communication with external systems is usually handled by automation plug-ins run by event notifications. For example, the progress of an order is typically monitored in external systems by tracking which parts of the order have been completed. To communicate that, you typically configure event notifications based on a change to order data or a change to task status.
- Notifications intended for an internal audience (OSM users) are typically created using a notification type that, by default, sends a notification to the Task web client. The only notification types that do not are event notifications based on order data change and task state change notifications that run an automation plug-in. See "[About Using Task States and Rules to Trigger Event Notifications](#)" for more information.

Modeling Notifications

The following sections provides information about modeling notifications.

Using Task States and Statuses to Trigger Event Notifications

You can use task states and task statuses to trigger event notifications. For example, changing to the Failure status can trigger a notification to a fallout specialist. See "[About Event Notifications](#)" for more information.

About Notification Priority

You can specify a priority for most types of notifications. For example:

- Notifications can be prioritized to control how they are sorted in the Task web client. You should prioritize jeopardy notifications higher than information messages.
- Prioritizing notifications sent to external systems helps those systems process the more important notifications first.

OSM evaluates notifications with the highest priority first (1 is the highest priority). For notifications that are sent to external systems, the notification priority represents the JMS queue priority.

About Sending Notifications in Email

You can deliver notifications in email. The email message consists of the same information that is displayed in the Notifications window in the Task web client. You cannot customize the message or add information to it. The message template is:

```
You have a notification for Order ID ID number and notification ID
notification ID. Use the following URL to connect to the notification details:
url
```

For most types of notifications, you specify to send email by selecting a check box in the notification configuration. For event notifications that are used only for running an automation plug-in, you configure the automation plug-in to send the email. See *OSM Developer's Guide* for information about automation.

To specify who to send the email to, you do the following:

- When configuring the notification in Oracle Communications Service Catalog and Design - Design Studio, or in your automation plug-in, specify the roles that receive the notification.
- Configure the email recipients for the roles by using the OSM Order Management web client. (Roles are called *workgroups* in OSM Administrator.)

About Configuring Entities to Support Notifications

Before you configure notifications, you need to configure the following entities:

- You must create the roles to assign notifications to.
- To trigger notifications based on a change to order data, you must first model the data. See ["About Using Order Data Changes to Trigger Notifications"](#) for information.

You can model automation plug-ins as you define notifications, but modeling automation plug-ins before you configure notifications is more efficient.

About Jeopardy Notifications

A jeopardy notification is a message that is sent to OSM users or users on other systems (for example, to return status to a CRM system). Jeopardy notifications are not event-driven; they use polling at specified intervals to identify processes or tasks in jeopardy.

OSM uses three methods to deliver jeopardy notifications:

- By displaying a notification in the Task web client.
- By sending email to users.
- By using an automation plug-in to notify an external system. Each order jeopardy notification can map to one automation plug-in.

Jeopardy notifications can be defined for an order using the Order Jeopardy editor or the Order editor, or for a task using the Task editor. Many of the jeopardy properties are the same for orders and tasks; for example, you can specify the roles to notify and the rule to trigger the notification. However, defining a jeopardy notification for an order or a task allows you to use the order or task properties. For example:

- You can trigger a notification based on the state of the order.
- You can trigger a notification if a task has exceeded its expected duration.

You can use two methods to trigger a jeopardy notification:

- Conditions; for example, if the order processing time has exceeded the expected duration.
- Order rules; for example, you can define an order jeopardy notification based on a rule that evaluates a data condition where an order milestone is not equal to the Complete state and has a due date that is greater than the value specified in the condition. For example:

orderMilestone <>completion and dueDate>SpecifiedDate.

This checks to see if there are any orders that are not completed but that are supposed to be completed by today.

About Modeling Jeopardy Notifications

You can model jeopardy notifications for tasks and for orders. The following list describes where in Design Studio you can model jeopardy notifications:

- **Task editor:** The Task editor **Jeopardy** tab is the only place to model task jeopardy notifications. For more information, see the topic on working with tasks in the Design Studio Modeling OSM Processes Help.
- **Order Jeopardy editor alone:** The Order Jeopardy editor enables you to define detailed jeopardy conditions based on order states, including multiple states to be used as start and

end states for the order jeopardy notification timer. The Order Jeopard editor also enables you to define order states during which the timer will pause. Although these order jeopardy notifications are not defined in the Order editor, they are still defined for a specific order. For more information, see the topic on working with jeopardy and event notifications in the Design Studio Modeling OSM Processes Help.

- **Order Jeopardy editor and operational jeopardy file:** When you define an order jeopardy notification in the Order Jeopardy editor, you can choose to make that order jeopardy notification an *operational* order jeopardy notification. This means that you can define the order jeopardy in Design Studio, but the details of the order jeopardy notification can be changed at run-time without having to redeploy any cartridges. You can change the details of operational order jeopardy notifications by editing text files on the OSM system. You specify the names of the text files in the **oms-config.xml** file. For more information about the Order Jeopardy editor, see the topic on working with jeopardy and event notifications in the Design Studio Modeling OSM Processes Help. For more information about using the text file to define operational order jeopardy notifications, see the information about configuring OSM with the **oms-config.xml** file in *OSM System Administrator's Guide*.
- **Order editor:** The Order editor **Jeopardy** tab provides simple order jeopardy modeling capabilities and works in basically the same way as the task jeopardy notification configuration. For more information, see the topic on working with orders in the Design Studio Modeling OSM Processes Help.

About Jeopardy Notification Triggering

OSM triggers jeopardy notifications in one of two ways, depending on where you modeled the order jeopardy notification.

If the order jeopardy notification has been modeled using the Order Jeopardy editor, either with the configuration defined in Design Studio or with an operational jeopardy defined in a text file, OSM triggers the notification using the following process:

1. When the timer starts for an order, OSM adds the notification to an internal list, sorted by the due date of the notification. The system frequently polls this list and retrieves the items that have come due.

Because this is a server-wide, internal, automatically generated list, you do not have to configure polling intervals for order jeopardy notifications defined in the Order Jeopardy editor.
2. OSM checks whether there are any rules that might restrict the notification from being triggered. For example, you might configure two jeopardy notifications, one that is triggered for orders from only business accounts and one that is triggered for orders from only residential accounts. Each notification might have a different email recipient, so the notification is only triggered for the correct recipient. The rule is checked after the conditions have been met to ensure that the latest version of the order data is used in the evaluation.
3. If a rule evaluates to true, the notification is triggered.

If the order jeopardy notification has been modeled using the Order editor or using the Task editor, OSM triggers the notification using the following process:

1. OSM polls in-flight orders and tasks to determine if a condition has been met. For example, the condition might be that a task has been in progress for longer than one hour. If the condition is met, OSM begins to process the notification.

You can specify how often OSM should poll to reevaluate the jeopardy condition. You can specify a polling interval in hours, days, weeks, or months. You can specify the day of the

week (for example, Monday), or the day of the month (for example, the first day of the month). You can specify a date and time for OSM to begin polling. The default is the current date.

✓ **Tip**

When configuring notifications in the Order editor or the Task editor, consider the performance impact from polling for jeopardy notifications. For example, a configuration that polls every minute on one million orders has a much greater performance impact than polling every hour on one thousand orders.

2. OSM checks whether there are any rules that might restrict the notification from being triggered. For example, you might configure two jeopardy notifications, one that is triggered for orders from only business accounts and one that is triggered for orders from only residential accounts. Each notification might have a different email recipient, so the notification is only triggered for the correct recipient. The rule is checked after the conditions have been met to ensure that the latest version of the order data is used in the evaluation.
3. If a rule evaluates to true, the notification is triggered.

About Jeopardy Notification Conditions

You can trigger jeopardy notifications based on an order or task condition. For example, you can specify to send a jeopardy notification if a task has exceeded its expected duration.

The conditions you can use depend on whether you define the jeopardy notification in the Order Jeopardy editor, the Order editor, or the Task editor.

Specifying Jeopardy Notification Conditions in the Order Jeopardy Editor

When you define a jeopardy notification for an order using the Order Jeopardy editor, you can specify to trigger the notification based on the following:

- The amount of time that an order has spent in one or more states. You can specify lists of order states that can do the following:
 - start the timer
 - stop and reset the timer
 - pause the timer

For example, you can trigger a notification because an order entered the In Progress state 30 days ago, without counting any time the order spent in the Suspended state, and has not yet entered the Completed or Aborted states.

You can set the expected duration in one of the following ways:

- Setting a specific duration value
- Using the expected duration for the order
- Using an XQuery expression to set the value
- Using the value of a field on the order
- Whether the order has reached a certain date without having reached one of a list of specified end states.

You can set the expected date in one of the following ways:

- Using an XQuery expression to set the value
- Using the value of a field on the order

Specifying Jeopardy Notification Conditions in the Order Editor

When you define an order, you can specify to trigger a jeopardy notification based on the following:

- The amount of time that an order has been in the In Progress state. For example, you can trigger a notification if the order has been in the In Progress state for longer than 30 days.
- The amount of time that an order has been in the Completed state. For example, you can trigger a notification if the order has been in the Completed state for longer than 30 days.
- If the process duration has exceeded the expected duration. The value is based on elapsed time, regardless of the order states that the order might transition in and out of.
- If the process duration has exceeded a duration that you define; for example, five days. This duration value starts at the creation task.

To determine the duration that the order has been in any of these conditions, OSM polls the system at an interval that you define.

Specifying Jeopardy Notification Conditions for a Task

When you define a task, you can specify to trigger a jeopardy notification based on the following:

- If the process that the task is associated with has exceeded the expected duration.
- If the process that the task is associated with has exceeded a duration that you define. This duration is measured starting with the creation task.
- If the task has exceeded the expected duration.
- If the task has exceeded a duration that you define.
- If the order has exceeded a specified amount of time past when it was received (when the order is created in OSM).

To determine the duration that the order has been in any of these conditions, OSM polls the system at an interval that you define.

When you define a jeopardy notification in a task, and the task can have multiple instances, you can specify if the notification should be triggered for every task instance.

About Event Notifications

Event notifications are triggered by events. You do not specify polling intervals for event notifications. You can configure them to occur in the following cases:

- When a task transitions through a task status. For example, you might trigger an event notification when a task transitions to the Failed status.

Event notifications triggered by transitions can be sent to a workgroup. See "[About Using Task Transitions to Trigger Event Notifications](#)" for more information.

- When a task reaches a specified state. You can use two methods:
 - You can use the task state to trigger an automated event notification. In this case, only the task state is evaluated (no rules are applied to evaluate a condition), and the notification runs an automation plug-in that handles the notification actions. For

example, when a task reaches the Assigned state, you can automate an external lookup before allowing the workflow to continue. You do not specify roles or email delivery for the notification. See "[About Using Task States to Trigger Automated Event Notifications](#)" for more information.

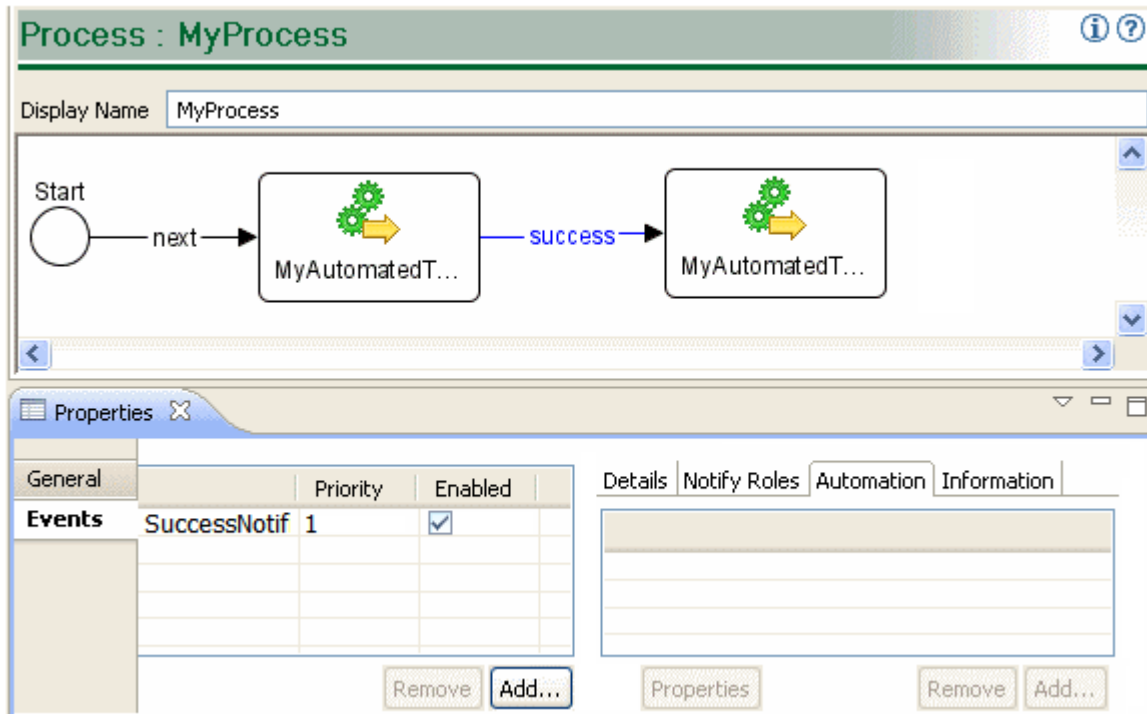
- You can use the task state in combination with rules to trigger the event notification. In this case, you can specify a rule to evaluate conditions, the priority, if the notification can be delivered by using email, and the workgroups that receive the notification. See "[About Using Task States and Rules to Trigger Event Notifications](#)" for more information.

When you use the task state to trigger an automated event notification, the notification is run from all processes that include the task. When you configure a notification based on a task state change in a process, the notification is applicable only to the task within the process in which it is defined.

- You can trigger an event notification when an order passes an order milestone. You use this type of notification to trigger an automation plug-in that handles the notification actions. You do not specify roles or email delivery for the notification. See "[About Using Order Milestones to Trigger Event Notifications](#)" for more information.
- You can trigger an event notification when a change is made to order data. You typically use these notifications to update external systems (such as a CRM) with information about the progress of the order when a specific data element in the order data is changed. See "[About Using Order Data Changes to Trigger Notifications](#)" for more information.
- You can trigger an event notification based on order life-cycle changes. OSM posts the these notifications to a pre-defined JMS queue. See "[About Enabling Order Life-Cycle Events](#)" for more information.

About Using Task Transitions to Trigger Event Notifications

An event notification based on a task transition does not apply to all instances of the task. It applies to a task only as it is used in a specific process. Therefore, to configure an event notification based on a task transition, you edit the process that includes the transition and apply the event notification to the transition. [Figure 16-1](#) shows the configuration for a success transition in Design Studio. In this figure, the success transition is selected, and the event notification properties are defined below the process window.

Figure 16-1 Event Notification Based on Task Transition

The event notification for a status change works as follows:

1. When the task status changes to the status that you define for the notification, the notification runs a rule to evaluate if the conditions are true.
2. If the conditions are true, the event notification is triggered.

When you use a task transition to trigger an event notification, you can specify an automation plug-in that the notification runs; however, an automation plug-in is not required.

About Using Task States and Rules to Trigger Event Notifications

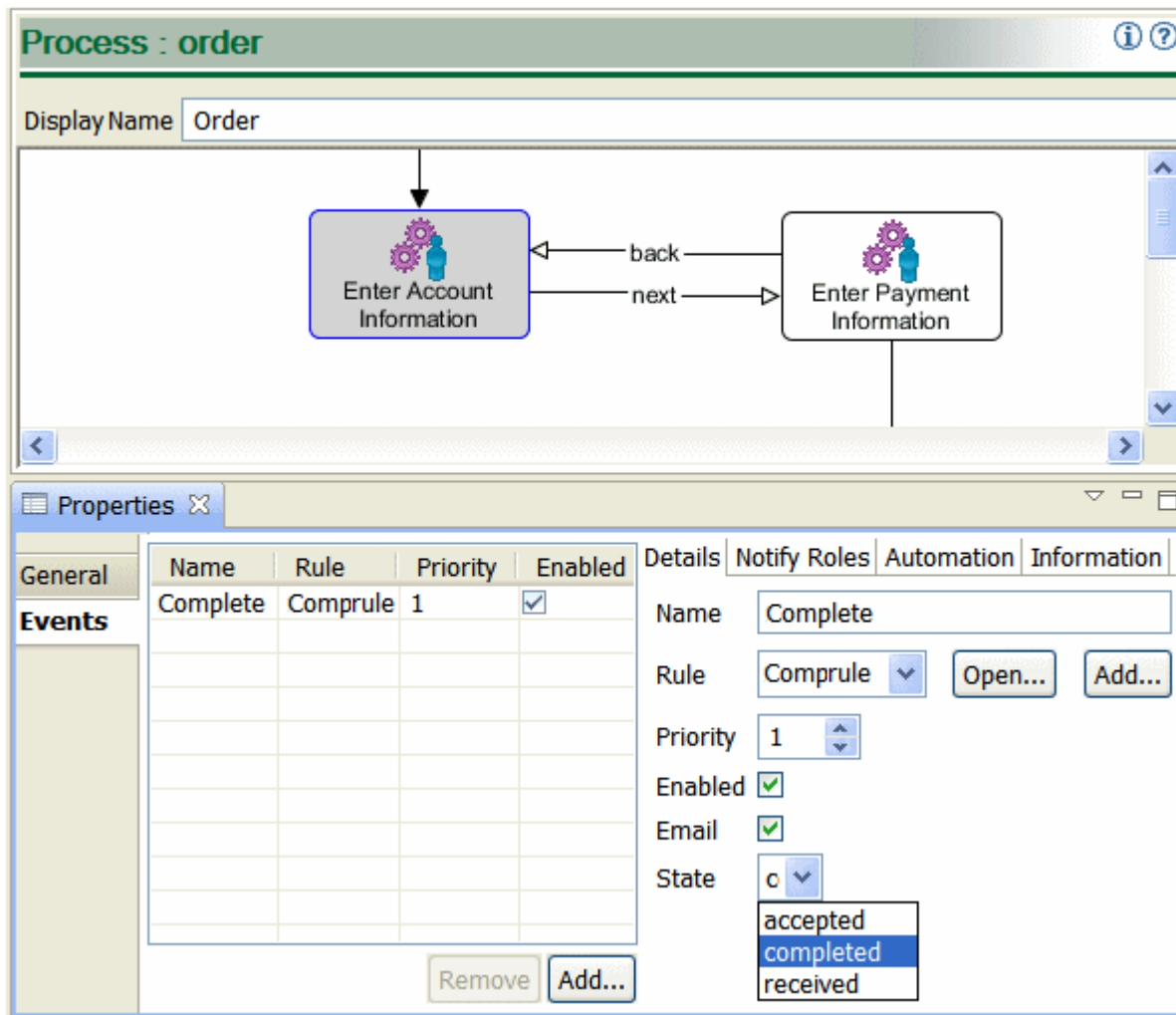
An event notification triggered by a task state change and rules works as follows:

1. When the task state changes to the state that you define for the notification, the notification runs a rule to evaluate if the conditions are true.
2. If the conditions are true, the event notification is triggered.

For example, you can specify that when the Completed task state is reached, a rule evaluates if the billing address is in California.

This type of notification does not apply to all instances of the task. It applies to a task only as it is used in a specific process. Therefore, you create this type of notification when you create processes in Design Studio. [Figure 16-2](#) shows how to assign an event notification to a task in a process. In this figure, the EnterAccountInformation task is selected, and the rule and state are defined in the Properties window Events tab below.

Figure 16-2 Notification Based on Task State and Rule



You can specify an automation plug-in that the notification runs; however, an automation plug-in is not required.

About Using Task States to Trigger Automated Event Notifications

You can use a task state to trigger an automated event notification. In this case, only the task state is evaluated (no rules are applied to evaluate a condition), and the notification triggers an automation plug-in which handles the notification actions. This type of notification runs for every instance of the task, independent of the process that it is in. Event notifications triggered by task states are not displayed in the Task web client.

For example, you can define an automated notification that sends a notification when the task reaches the Assigned state. The event notification works as follows:

1. When the task reaches the Assigned state, a notification is created.
2. When the notification is created, the automation plug-ins run.

[Figure 16-3](#) shows an event notification configured in Design Studio. Any time this task runs, the event notification is triggered when the task reaches the Completed state.

Figure 16-3 Event Notification Based on Task Status

The screenshot shows the 'Automated Task : activate_dslam' configuration window. The 'Display Name' is 'Activate DSLAM'. The 'Task State Automation Events' section shows a table with one event: 'TaskComplete' using the 'xQuerySender' automation type. Below this is a 'Properties' window for the 'Automation Plug-in - XQuery Sender'. The 'Details' tab is active, showing the event name 'TaskComplete', EJB Name 'activate_dslam.TaskComplete', Run As '%{DEFAULT_AUTOMATION_USER}', and Event Type set to 'Internal Event Receiver'.

State	Automation				
completed	<table border="1"><thead><tr><th>Name</th><th>Automation Type</th></tr></thead><tbody><tr><td>TaskComplete</td><td>xQuerySender</td></tr></tbody></table>	Name	Automation Type	TaskComplete	xQuerySender
Name	Automation Type				
TaskComplete	xQuerySender				

Task Data | Automation | Details | States/Statuses | Compensation | Permission | Jeopardy | Events

Automation Plug-in - XQuery Sender

Details | XQuery | Routing | Information

Name: TaskComplete
EJB Name: activate_dslam.TaskComplete
Run As: %{DEFAULT_AUTOMATION_USER}
Event Type: ☒ Internal Event Receiver ☐ External Event Receiver

About Using Order Milestones to Trigger Event Notifications

You can use an order milestone to trigger an event notifications. [Figure 16-4](#) shows an event notification based on an order milestone.

Figure 16-4 Event Notification Based on an Order Milestone

Order : vf_demo_web

Display Nam VF Demo Web

Order Milestone Automation Events

Milestone	Automation				
completion	<table border="1"> <thead> <tr> <th>Name</th> <th>Automation Type</th> </tr> </thead> <tbody> <tr> <td>OrderComplete</td> <td>xQueryAutomator</td> </tr> </tbody> </table>	Name	Automation Type	OrderComplete	xQueryAutomator
Name	Automation Type				
OrderComplete	xQueryAutomator				

Remove Add... Properties Remove Add...

Details Amendable Rules Fallouts Fallout Groups Jeopardy Notification Events »₃

Properties

Automation Plug-in - XQuery Automator

Details XQuery Information

Name OrderComplete

EJB Name vf_demo_web.OrderComplete

Run As %{DEFAULT_AUTOMATION_USER}

Only the order milestone is evaluated (no rules are applied to evaluate a condition), and the notification triggers an automation plug-in that handles the notification actions. Each event notification maps to one or more automation plug-ins. For more information about automation plug-ins, see "[About Automation Plug-ins](#)".

For example, you can define an event notification that specifies the Completion milestone. The event notification works as follows:

1. After all tasks within a process successfully complete for an order, the order Completion milestone is reached and a notification is created.
2. When the notification is created, the automation plug-ins run.

Note

You cannot define custom order milestones. Order milestones are based on order states; for example, the Completion milestone occurs when the order transitions to the Completed state.

When you create event notification that is triggered by an order milestone, you specify the order milestone that triggers the notification. You can use the following order milestones:

- Creation: The order was created in the OSM system.

- Completion: The final task in the order has completed, and the order transitioned to the Completed state.
- Deletion: The order was removed from the OSM system by transitioning to the Deleted state.
- Exception: A process exception or fallout was initiated.
- State change: The order transitioned to a different state.

About Using Order Data Changes to Trigger Notifications

You define event notifications based on order data changes when you create orders in Design Studio. For example, you can define an event notification that sends a notification when a telephone number is entered. Event notifications triggered by data changes are shown in the Task web client.

When you create an event notification based on order data changes, you can specify the data field that triggers the notification when the data is changed. Any change to the field causes the notification to trigger. However, this value is not evaluated for content. To trigger the notification based on the value of the data, you must configure a rule to evaluate it.

For example, to trigger a rule when the billing address is changed to California, you specify the billing address field as the field that triggers the notification and run a rule that evaluates if the address was changed to California.

You can specify an automation plug-in that the notification runs; however, an automation plug-in is not required.

[Figure 16-5](#) shows an event notification based on data change in an order. In this example, when a credit card number changes, the notification is triggered.

Figure 16-5 Event Notification Based on Data Change in an Order

Order : vf_demo_web

Display Name VF Demo Web

Event Notifications

Name	Details	Notify Roles	Data Changed	Automation	Information					
cc_number	<table border="1"><thead><tr><th>Nodes</th></tr></thead><tbody><tr><td>/payment_information/credit_card_information/credit_card_number</td></tr><tr><td> </td></tr><tr><td> </td></tr><tr><td> </td></tr></tbody></table>					Nodes	/payment_information/credit_card_information/credit_card_number			
Nodes										
/payment_information/credit_card_information/credit_card_number										

Remove Add...

Behaviors Details Amendable Rules Fallouts Fallout Groups Jeopardy Notification »₃

About Enabling Order Life-Cycle Events

You can configure orders to publish events when any of the following occurs:

- The order is created.
- The order is removed.
- The order state changes.
- Amendment processing starts.
- Amendment processing is queued.
- Amendment processing completes.
- Amendment processing is terminating.
- Amendment processing is terminated.
- Amendment processing is abandoned.

Order life-cycle events are published to the `oms_order_events` queue as Java Message Service (JMS) messages containing order identification and state information. You can configure which life-cycle events you want to be generated for an order type in Design Studio.

Summary of Notification Functionality

[Table 16-1](#) shows a summary of notification functionality.

Table 16-1 Summary of Notification Functionality

Notification Type	Sends Email	Displays in Task Web Client	Can Be Evaluated By a Rule	Can Be Sent to Different Roles	Runs Automation Plug-in	Has a Priority
Jeopardy - Task editor	Yes	Yes	Yes	Yes	Optional	Yes
Jeopardy - Order Jeopardy editor	Yes	Yes	Yes	Yes	Optional	Yes
Jeopardy - Order editor	Yes	Yes	Yes	Yes	Optional	Yes
Event - Task status	Yes	No	Yes	Yes	Optional	No
Event - Task state, automation	Sent by automation plug-in only	No	No	Defined by automation plug-in only	Mandatory	Yes
Event - Task state, in a process	Yes	No	Yes	Yes	Optional	Yes
Event - Order milestone	Sent by automation plug-in only	No	No	Defined by automation plug-in only	Mandatory	No
Event - Order data change	Yes	Yes	No	Yes	Optional	Yes

Modeling Milestone Events

This chapter describes how to model milestone events in an Oracle Communications Order and Service Management (OSM) solution.

Before reading this chapter, read the following to learn about general OSM concepts:

- [OSM Solution Modeling Overview](#)
- [Modeling Fulfillment States and Processing States](#)

See the following topics for details about Model-driven Milestones:

- [About Milestones and Model-driven Milestones](#)
- [Usage of Milestone Events](#)
- [Modeling Model-driven Milestones](#)

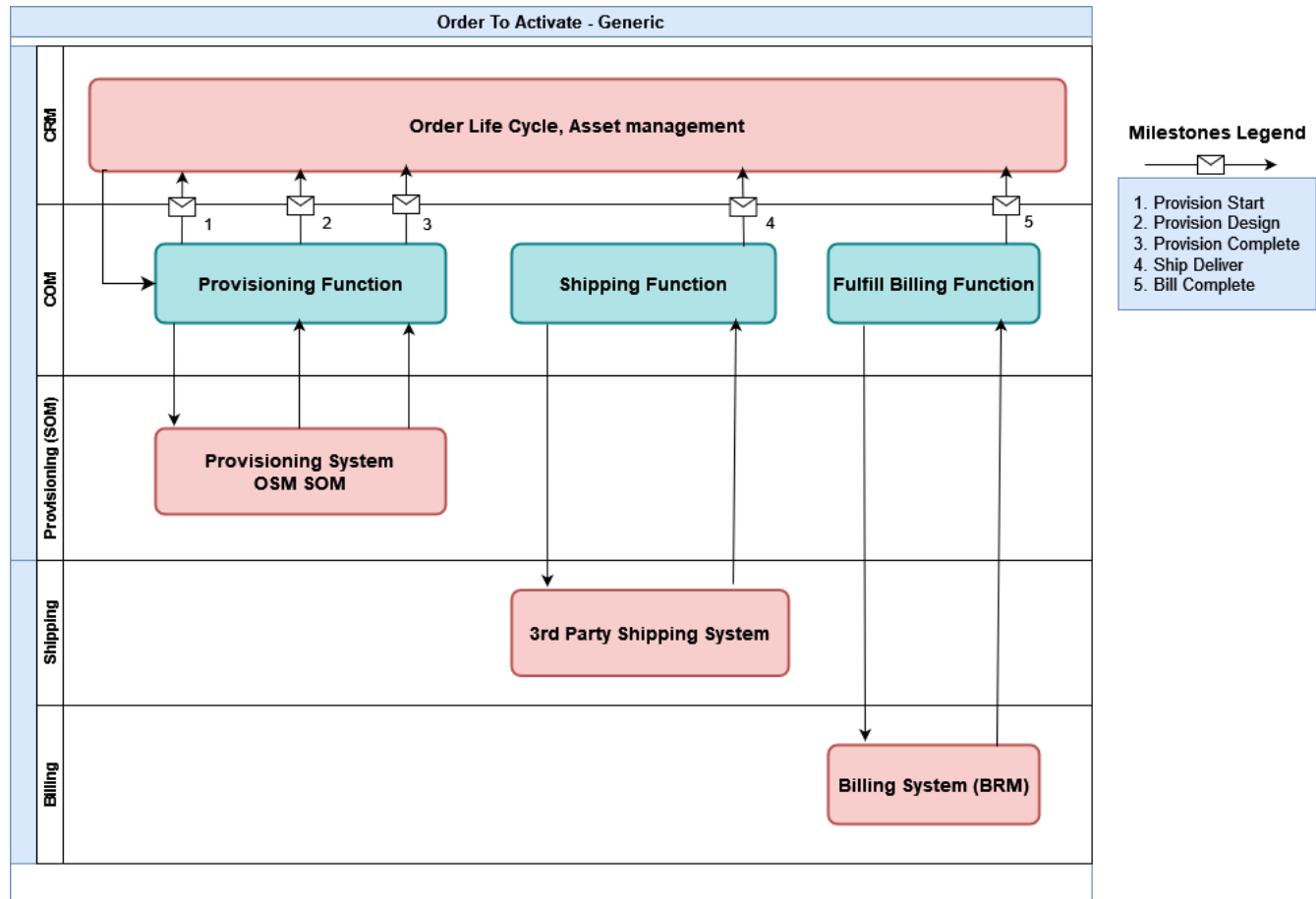
About Milestones and Model-driven Milestones

For order fulfillment, milestone represents an achievement of the process being reached. While an order state mainly represents the fixed operation state of the order such as in-progress, completed, cancelled, amending and so on, it does not represent the complexity and the variation of order fulfillment in different business domains. OSM uses fulfillment states that provide flexible modeling to define the order fulfillment achievement for a domain. Fulfillment state is a mechanical evaluation of the order fulfillment process as it relates to an order item or to the whole order.

Model-driven Milestone considers a specific fulfillment state for a specific order item or order, or the external fulfillment state that triggers the fulfillment state evaluation to declare that as a business-specific checkpoint being reached during order fulfillment. Milestones have meaning to the overall solution as they provide a progress report on an order that is consumable by human users and external systems. Milestones are propagated from downstream system, when an order item reaches PONR or a specific fulfillment state, or when a whole order reaches a specific fulfillment state.

[Figure 17-1](#) illustrates an example of a sequence of five milestones propagated from OSM COM to an upstream system (CRM).

Figure 17-1 Milestone Interaction



Usage of Milestone Events

Milestone propagation from OSM to upstream system allows upstream systems (such as a CRM system) to keep track of the order fulfillment progress. The current milestone of a fulfillment order may trigger interaction between an upstream system and other edge systems or customer interaction. Upstream systems also rely on milestones to decide the type of orchestration action that can be submitted to alter the orchestration process. These orchestration actions include cancel order, amendment order and follow-on order.

For example, when an upstream system is required to amend an in-progress order fulfillment, the milestone will be used to decide whether an in-flight amendment should be sent (if PONR is not reached) or a follow-on order should be sent (if PONR is reached). For cancelling, the same PONR milestone may be used to decide whether an in-flight cancel order should be sent (if PONR is not reached) or a disconnect order is needed (if PONR is reached). The upstream system creates a new version of assets after receiving a milestone with details of the product which may also be enriched with network resource.

Without Model Driven Milestones, the milestone declaration, detection and propagation have to be implemented by cartridge code, which runs within the automation plugin. For example, the receiver plugin needs to check the response data from the downstream system to scan for an expected external fulfillment state value. Such external fulfillment state value may be hard coded or stored in a freeform metadata outside of OSM. The detected milestone is then sent to the upstream system by the same automation plugin that processes the response message.

The implementation of detection and propagation is usually added on top or combined with the normal processing logic of the automation plugin, resulting in a more complicated code.

Note

Message propagation should always be executed in the same transaction of the automation plugin so the current order data reflects the snapshot of this time. Using the data changed or the notification event that is executed in a new transaction is not recommended, since additional data may be updated by the other transaction that is committed prior to the new transaction.

Model-Driven Milestones provide the following:

- A design-time milestone **declaration**
- A design-time milestone **detection**
- A design-time milestone **propagation and configuration**

This allows all implementation related to the milestone activities to be excluded from the automation plugin. If an existing milestone defined is not valid for a new order fulfillment pattern, then disabling, removing or updating the configuration of the milestone can be managed efficiently.

Modeling Model-driven Milestones

A model driven milestone configuration consists of:

- Milestone detection
- Order data associated with the milestone
- Message routing configuration

Milestone Detection

For milestone detection, OSM uses the external fulfillment state, order item fulfillment state, order fulfillment state and order item PONR as the valid data element to trigger the milestone. If update to the state mentioned early and the value matches the configuration, then the milestone propagation process is triggered.

For example, consider automation plugin updating an external fulfillment state value as "**Provision Design**" with the following path:

```
/ControlData/Functions/ProvisionOrderFunction/orderItem/  
ExternalFulfillmentState
```

Milestone propagation is triggered if model-driven milestone is defined for the following:

- For external fulfillment state with "Provision Design" and for fulfillment function with "ProvisionOrderFunction".
- For fulfillment state with "Design" for order Item under fulfillment function with "ProvisionOrderFunction", and the external fulfillment state map with "Provision Design" to fulfillment state with "Design".
- For order item PONR for fulfillment pattern which invokes fulfillment function "ProvisionOrderFunction", the PONR is reached and fulfillment state "Design" is reached.

Order data associated with the milestone

The milestone propagation process creates a payload that describes the details of the milestone. The content is driven by the order data. To support that, the configuration for the second item consists of an xquery or xslt script, which is executed by the milestone propagation process with the order data supplied by an order view. The xquery or xslt script is responsible to create the data payload sent to an upstream system. The implementation details of the xquery or xslt script are the same as automation. For more details, see the "Using Automation" section in the *OSM Developer's Guide*.

Message routing configuration

Message routing configuration contains the target system's JMS queue name. The milestone propagation process delivers the milestone message into this JMS queue.

Modeling Order Scheduling

This chapter describes how to model order scheduling entities in an Oracle Communications Order and Service Management (OSM) solution.

About Order Item Requested Delivery Date and Order Components

OSM can process orders at different times. In many cases, a customer wants an order to be completed as soon as possible, in which case OSM can start processing the order immediately. However, in some cases, the start date of an order should be delayed until a future date. For example:

- A customer might request that a new VoIP service be added at the beginning of the next month, when their current service expires.
- A customer might request the disconnect of an existing service at the end of the current month.

In addition, there may be groups of order items within an order that need to be fulfilled at different times. For example, an order might contain three services, such as internet, IPTV, and VoIP. The internet and IPTV services might have an immediate requested delivery date, but the VoIP service might only be required at the end of the month, after the customer's current phone service plan has expired. In this case, you can enable OSM to calculate a time to start fulfilling the VoIP service at a future date that would allow the service to be activated by the requested delivery date: at the end of the month.

Different groups of order items may have orchestration dependencies configured that have an impact on when a service gets fulfilled. For example, the internet service might be required before you can activate an IPTV or VoIP service. These dependency scenarios are fixed and take precedence over honoring requested delivery dates. In other words, OSM will only honor a requested delivery date for a service if there is enough time to fulfill that service given the time it takes to perform the fulfillment tasks and any dependencies that might exist between one service and another. In such a scenario, the order completion date will be later than the delivery date requested by the customer.

To accurately calculate when an order should start so that it can meet a requested delivery date, you must determine how long it takes to perform certain tasks contained in the order and you must know when a customer wants a service.

Note

Orders must have an orchestration plan to be able to calculate the order completion date.

When viewing an entire order in the Order Management web client **Summary** tab General area, you see the following fields:

- **Order Creation Date:** The date when the order is created in OSM.

- **Expected Order Start Date:** The date when the order is expected to start being processed.
- **Expected Order Completion Date:** The date when the order is expected to be completed.
- **Requested Order Delivery Date:** The date by which the customer requests the order be delivered.
- **Expected Order Duration:** The amount of time the order is expected to take to complete processing.

These fields are used in, or derived from, an orchestration plan algorithm. This algorithm, at its highest level, uses the **Order Creation Date** (for orders that start immediately) or the **Expected Order Start Date** (for future dated orders) in conjunction with the **Expected Order Duration** to determine whether there is enough time to achieve the **Requested Order Delivery** date. If there is enough time, then the **Expected Order Completion Date** is the same date as the **Requested Order Delivery Date**. If there is not enough time, then the **Expected Order Completion Date** is later than the **Requested Order Delivery Date**.

When viewing a specific order item in the Order Management web client **Summary** tab General area, you see the following fields:

- **Expected Order Component Start Date:** The date when the order component that processes the order item is expected to start.
- **Expected Order Item Start Date:** The date when the order item is expected to start. This is always the same date as the first Order Component Start Date to start processing the order item.
- **Expected Order Item Completion Date:** The date when the order item is expected to be complete. In some scenarios, an order item may require processing from more than one order component. For example, one order component may provision the service while another performs the billing function. And so the order item completion date must take into account the total time it takes to complete these two order components.

The following sections describe the design-time and run-time elements that you must model so that the orchestration plan algorithm can generate an order fulfillment timeline.

How OSM Decomposes and Processes Order Items in Order Components

You can model the decomposition of order items into order components that typically share the same function, are destined for the same fulfillment system, and share the same processing granularity. The entity that ultimately processes order items is an executable order component that is linked to a process that contains a sequence of manual and automated tasks that fulfill every order item in the order component.

OSM calculates the order component start dates based on the requested delivery date for order line items in customer orders. This requested delivery date order line item value must be mapped to an order item specification **requestedDeliveryDate** order item property in Oracle Communications Service Catalog and Design - Design Studio.

For example, a group of six order items might be gathered in an executable component that is linked to a process that contains an automated task that generates and sends a service request to an activation system. The service request that the automated task builds would contain all the information from the six order items that the activation system requires to activate services that correspond to the order items in the network.

When OSM has determined the order component start date, all order items in the order component begin processing immediately (regardless of their requested delivery date). Although this can mean that some order items might be delivered early, it ensures that no order items are delivered late.

About Grouping Order Items in Order Components by Date Range

If order items belong to the same function and go to the same fulfillment system need to be fulfilled on substantially different dates, you can model different order components in Design Studio that run at different stages or within the same stage, but that have different start dates.

In addition, OSM provides Java functions that can be used along with order item hierarchies to further delineate and group order component IDs based on order item requested delivery date. For more information about creating custom component IDs using Java function, see "[About Component Specification Custom Component ID XQuery Expressions](#)".

Modeling Order Component Minimum Processing Duration

When you model orders in Design Studio, you need to provide OSM with enough information to be able to meet the order item requested delivery dates with as much accuracy as possible. To do so, you specify a minimum processing duration value that defines how long it typically takes to fulfill all order item within an executable order component. You can model this value at the order component level (see [Figure 18-1](#)) or at the fulfillment pattern order component level (see [Figure 18-2](#)). OSM always uses the larger of the two values. This duration should take into account the total duration of any manual or automated tasks involved in completing the process. For example, if you know that it takes one week to ship a telephone, you specify one week for the minimum processing duration for an order component that is used for shipping a telephone.

You can specify a different minimum processing duration for each fulfillment mode in the fulfillment pattern. For example, the Deliver fulfillment mode can have a different duration than the Cancel fulfillment mode.

[Figure 18-1](#) shows the duration defined for an order component.

Figure 18-1 Processing Duration Defined for an Order Component

Order Component Specification : Bundle

Description: Bundle

Namespace: http://oracle.communications.ordermanagement.unsupported.centralom

Minimum Processing Duration

Duration: 2 Days

Duration Expression

Details | Order Template | **Duration** | Component ID | Permissions

[Figure 18-2](#) shows the processing duration assigned to an order component when it is used in a fulfillment pattern **Order Components** tab, **Selected Order Components** sub-tab.

Note

The **Duration** tab displayed beside the Order Components tab and **Dependencies** tab in [Figure 18-2](#) is no longer used. This tab still appears in Design Studio to support OSM cartridges that target pre-OSM 7.2.2 servers.

Figure 18-2 Processing Duration Defined for an Order Component as Used in a Fulfillment Pattern

The screenshot displays the Oracle Design Studio interface for configuring an 'Orchestration Fulfillment Pattern : Service.Fixed'. The 'Description' is 'Service.Fixed' and the 'Namespace' is 'http://oracle.communications.ordermanagement.unsupported.centralom'. The 'Order Components' tab is active, showing a list of components including 'Function', 'ShipFunction', 'MarketingFunction (Inherited)', 'ProvisioningFunction' (selected), 'BillingFunction (Inherited)', 'SyncCustomerFunction (Inherited)', 'CollectionsFunction (Inherited)', 'TargetSystem', and 'ProcessingGranularity'. The 'Selected Order Component: ProvisioningFunction' is shown on the right, with a 'Duration' of 1 Day. The 'Processing Duration' section shows a 'Duration' of 1 and a unit of 'Days'. A 'Duration Expression' field is also visible.

The minimum processing duration of an order may vary greatly depending on a number of factors:

- The kinds of products or services. Orders for mobile services typically have a very short processing duration, whereas a complex business-to-business order might take weeks.
- What must be done to fulfill the actions on the product or service, such as shipping or installation work.
- Any dependencies within and between the products and services. For example, PSTN provisioning must complete before ADSL provisioning starts.

Because a single order can have multiple values for the minimum processing duration, defined in multiple order components and at the order level, OSM compares all of them (if they are defined) to find the longest processing duration for the order:

1. OSM compares the two possible values of the minimum duration for an order component:

- The duration specified in the order component itself.
- The duration assigned to the order component in its fulfillment pattern.

OSM uses the larger of the two values as the order component minimum processing duration.

2. OSM adds the calculated durations for all of the order components in the order. OSM takes into consideration dependencies between order components. For example, if the order component that provisions a service depends on the order component that processes billing, the minimum processing duration for both components must be used.
3. OSM calculates the order duration based on the expected order completion date minus the start date.

About Minimum Processing Duration Inheritance in Fulfillment Patterns

For the minimum processing duration that is assigned to an order component by a fulfillment pattern, the minimum processing duration for the order component is inherited in fulfillment patterns extended from the parent fulfillment pattern. For example:

1. In the BaseProductSpec fulfillment pattern, the BillingFunction order component is assigned a duration of 2 days.
2. The Service.Fixed fulfillment pattern is extended from the BaseProductSpec fulfillment pattern. Therefore, if you do not specify a duration for the BillingFunction order component in the Service.Fixed fulfillment pattern, it inherits the duration of 2 days from the BaseProductSpec fulfillment pattern.

[Figure 18-3](#) shows how the duration is inherited from a parent fulfillment pattern.

Figure 18-3 Minimum Processing Duration for an Order Component Inherited in a Fulfillment Pattern

Parent Fulfillment Pattern

Order Components

Order Components

Base Component

All Components

Functions

All Components

Search

TargetSystem

ProcessingGranularity

Function

BillingFunction

CollectionsFunction

Selected Order Component: BillingFunction

Duration

Order Item Associations

Conditions

Processing Duration

Duration

2

Days

Duration Expression

Child Fulfillment Pattern

Order Components

Order Components

Base Component

All Components

Functions

All Components

Search

TargetSystem

ProcessingGranularity

Function

BillingFunction (Inherited)

CollectionsFunction (Inherited)

Selected Order Component: BillingFunction

Duration

Order Item Associations

Conditions

Processing Duration

Duration

2

Days

Duration Expression

About Minimum Processing Duration Expressions

In addition to specifying a fixed amount of time as the duration, you can use an XQuery expression. The following expression returns a duration of three hours:

```
PT3H0M0S
```

You typically use a duration expression if you have an external system that keeps track of processing duration and the load levels of systems. You can write a duration expression that uses this information dynamically. For example, the calculation can take into account peak activity periods.

Calculating the Earliest Order Component Start Date (Order Start Date)

The first order component to start processing can contain one or more order items. OSM uses the order item with the earliest requested delivery date to calculate the order component start date. If there were only one level of order component decomposition in the orchestration plan and there were no dependencies between order components, OSM would calculate the order component start date by taking the earliest order item requested delivery date and subtracting the configured minimum processing duration for the order component. This calculated start date would also be the order start date.

In the scenario, the following order component start dates are possible:

- If the component start date (also the order start date) is in the future, OSM does not start the order component until the future date. In the Order Management web client, you would see:
 - The expected order start date would be later than the order creation date.
 - The order component Expected Start Date would be the same as the expected order start date.
 - The expected order item start date for all order items in the order component would be the same as the order component Expected Start Date.
 - The expected order completion date and the requested order delivery date would be identical.
- If the component start date is in the past, OSM starts the order immediately. In the Order Management web client, you would see:
 - The order component Expected Start Date would be the same as the expected order start date.
 - The expected order item start date for all order items in the order component would be the same as the order component Expected Start Date.
 - The requested order completion date would be before the expected order delivery date.
- If no minimum processing duration was configured for the order component, then the order component would start on the same day as the requested delivery date, assuming that day was a future date. In the Order Management web client, you would see:
 - The expected order start date would be later than the order creation date.
 - The order component Expected Start Date would be the same as the expected order start date.
 - The expected order item start date for all order items in the order component would be the same as the order component Expected Start Date.
 - The requested order completion date would be on the same date as the expected order delivery date.
- If the order item contained no value for the requested delivery date property, then OSM starts the order immediately.

About Calculated Order Component Start Dates

The first order component in an order and any initial order component that does not depend on another order component always uses a calculated start date based on order item requested

delivery date values. If the order items do not have values for the requested delivery date, then the order begins processing immediately.

Dependent order items start in the following ways:

- Any dependent order components start immediately after the first or initial order component completes and all dependencies are resolved. This is the default behavior for order components.
- You can enable calculated start dates for dependent order components by selecting the **Use Calculated Start Date** check box in the Order Component Specification (see [Figure 18-4](#)). Dependent order components use the calculated start date based on the earliest order item requested delivery date in the order component, minus the order component duration. See "[Modeling Order Component Dependencies and Requested Delivery Dates](#)" for more information about configuring dependent order component calculated start dates.

Figure 18-4 Enabling Order Component Calculated Start Dates

The screenshot shows the 'Order Component Specification : MarketingFunction' dialog. The 'Description' is 'MarketingFunction' and the 'Namespace' is 'http://oracle.communications.ordermanagem'. The 'Extends' field is set to 'Function'. The 'Process' is 'MarketingSI'. The 'Base Task' field is empty. The 'Order Item' is 'CustomerOrderItemSpecification'. The 'Order Component Executable' checkbox is unchecked, and the 'Use Calculated Start Date' checkbox is checked. Under the 'Applies To Order Component' section, the 'Base Component' and 'Functions' are both set to 'All Components'. At the bottom, there is a 'Search' field and a tabbed interface with 'Details' selected, showing columns for 'Order Template', 'Duration', 'Component ID', and 'Perm'.

For a three stage orchestration cartridge with function, system, and granularity components, you can enable calculated start dates at the function level if you wanted all components related to that function to use a calculated start date. Or you can enable calculated start dates at the system level. In this second scenario, one function might decompose to more than one system level component and a calculated start date might only be required for one of them.

Modeling Order Component Dependencies and Requested Delivery Dates

An OSM orchestration cartridge can have several order components with dependencies configured between them. OSM always honors any order component dependency wait condition before starting a new order component. You can configure dependent order components to start immediately after the blocking order component is complete and all dependencies have been met, or you can use the calculated start date. See "[About Calculated Order Component Start Dates](#)" for more information.

This scenario assumes that the dependency between the order component order items are between different order items. For example, order item 1 is only processed by order component A (the blocking order component) and order item 2, which is dependent on order item 1, is only processed by order component B (the waiting order component).

The following dependent calculated order component start date scenarios are possible:

- If the component start date is in the future, and the blocking order component is complete with all dependencies met, then OSM does not start the order component until the calculated start date arrives.
- If the component start date is calculated to a date before the blocking order component is complete and all blocking order component dependencies are met, then OSM ignores the calculated start date. The order component begins immediately after the blocking order component completes and all dependencies are resolved.
- If the order item contained no value for the requested delivery date parameter, then OSM starts the order immediately.

Modeling Order Items Processed by Multiple Dependent Order Components

If OSM processes an order item in more than one executable order component, and there is a dependency between these executable order components, then OSM calculates the order component start dates for the first order component by subtracting the duration from the longest chain of order component durations involved in processing the order item from the earliest order item requested delivery date. This ensures that all order components can be delivered by the requested delivery date. All dependent order components in this scenario would start immediately after the previous order component was resolved. For example, if order item 1 is processed by order component A, B, and C, and B and C depend on A, then the order component start date for A would be the requested delivery date for order item 1 minus the duration of either order components B or C (whichever was longer) and A. Or, if B was dependent on A, and C was dependent on B, then OSM would subtract the total duration of A, B, and C from the requested order delivery date of order item 1 to determine the start date for order component A.

Revisions of Future-Dated Orders

You can submit revision orders to future-dated orders. The revision order can have a different requested delivery date than the base order or the same requested delivery date. In either case, OSM re-calculates the start date for the revision order based on its requested delivery date and on the minimum processing durations of the revised order components.

Note

Future-dated orders that cancel a future-dated base order are special cases. In this situation, the base order is canceled immediately, regardless of the requested delivery dates.

You can submit a future-dated revision order for an order that has already started processing. Only order components that have not started can have new calculated start dates applied. The new requested delivery date will trigger a compensation only if the order item specification **requestedDeliveryDate** order item property is marked as significant. Any task compensation required (for example, in previous completed order components) also happens immediately.

As a result of changing a significant order item requested delivery date, OSM calculates a new orchestration plan. Order components that have compensation tasks set with undo, redo, or amend do compensation strategies are processed based on the dependency graph of the revised base order orchestration plan. The order item requested delivery date modification may change the calculated start date of the order component that is processing the order item and, by extension, may also change the expected order completion date.

Examples of Calculating the Expected Start Date

The following examples show scenarios for calculating the expected start date for an order and order components.

Example 1: Calculating Start Dates for Order Components with No Dependencies

In this example:

- A billing function order component has a duration of 2 days and processes order item 1 with a requested delivery date of January 3rd.
- A provisioning function order component has a duration of 3 days and processes order item 2 with a requested delivery date of January 5th.
- There are no dependencies between order components.

The start date for each order component is calculated as follows:

1. The calculated start date for the Billing order component is calculated using the following logic:

- Order item 1 requested delivery date January 3th
- Minus Billing order component duration 2 days
- The Billing order component start date is January 1st.

Because there are no dependencies between the order components, OSM calculates the start date for each order component separately.

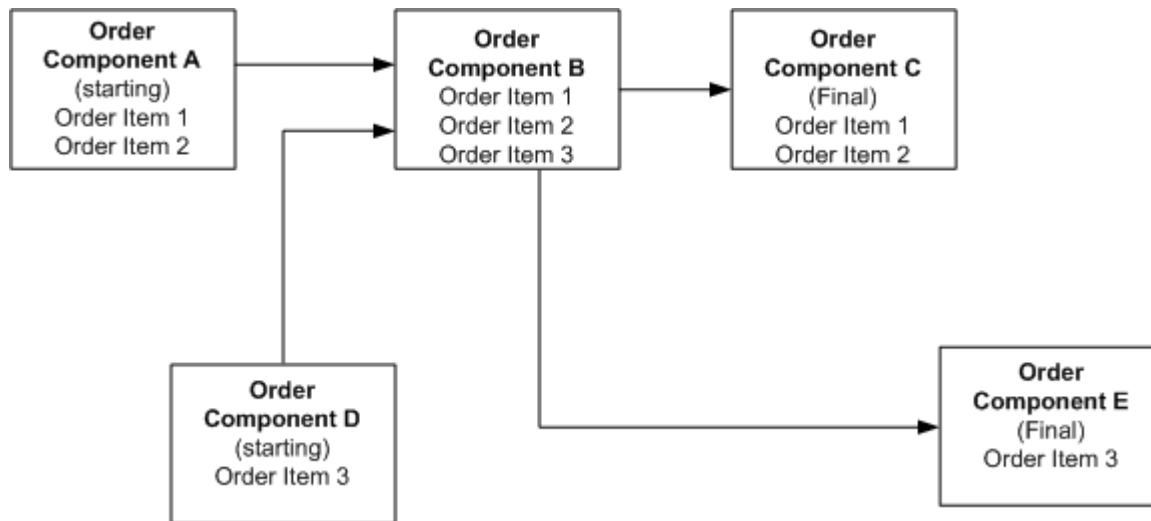
2. The calculated start date for the Provisioning order component is calculated using the following logic:

- Order item 1 requested delivery date January 5th
- Minus Provisioning order component duration 3 days
- The Provisioning order component start date is January 2nd.

Example 2: Calculating Start Dates for Order Components with Dependencies

OSM always uses the final set of order components for in an orchestration plan to determine the start date for the order component. A final order component has no successor order components. For example, [Figure 18-5](#) shows the order component processing flow for three order items. Order components C and E are final order components.

Figure 18-5 Order Component and Order Item Processing Flow



OSM calculates start dates for each order component starting with the requested delivery date of the final order components minus the order duration and any dependency condition wait delay duration. In this example:

- **Order component C** processes order item 1 and 2. Order item 1 has a requested delivery date of January 8, while order item 2 has a requested delivery date of January 10. OSM always uses the earliest requested delivery date to calculate the start date for the order component, which means the January 8 date is used. Because **order component C** is configured with a duration of 2 days, then **order component C** starts on January 6th.
- **Order component E** processes order item 3 that has a requested delivery date of January 18. Because **order component E** is configured with a duration of 2 days, then **order component E** would start on January 16th.

OSM calculates the start date of **order component B** by subtracting the configured duration for **order component B** (2 days) minus the start date for **order component C** (January 6th) resulting in a start date for **order component B** of January 4th.

OSM uses **order component C** instead of **order component E** to calculate the start date for **order component B** because **order component C** is a final order component with an order item that has the earliest requested delivery date. OSM does this to ensure that all order items being processed by an order component are not started late, even though they may start early. In other words, those order items being processed in **order component B** complete earlier than **order component E** needs them, but those order items destined for **order component C** complete with sufficient time for **order component C** to meet order item 1's requested delivery date of January 8th.

Finally, OSM calculates the start dates for **order components A** and **D**. **Order component A** has a configured duration of 3 days minus the start date for order component B (January 4th)

resulting in a start date of January 1st. **Order component D** has a configured duration of 2 days resulting in a start date of January 2nd.

The order start date is the earliest of all starting order components. In this example, the earliest order component start date is January 1st for **order component A**.

Part IV

Managing OSM Projects

Part IV contains the following chapter about managing OSM projects in an Oracle Communications Order and Service Management (OSM) solution:

- [Managing OSM Solution Cartridges](#)

Managing OSM Solution Cartridges

This chapter describes managing cartridges in an Oracle Communications Order and Service Management (OSM) solution.

Solution Management Overview

Cartridges are an important part of every OSM solution. OSM is a metadata-driven system that includes model entities necessary for order management functions. OSM model entities include tasks, processes, data elements, data structures, orchestration stages and sequences, decomposition rules, roles, order life cycle policies, and so on. You define how you want to use and extend OSM to meet your business needs by creating these model entities in Oracle Communications Service Catalog and Design - Design Studio. Design Studio is a client application with editors that allows you to configure the OSM model and model entities. In addition to the model entities, you can create and include artifacts such as automation plug-ins, XQuery and XSLT scripts, XML Catalog, and other resource files in a cartridge. You must package these entities and artifacts and deploy them to the OSM system as cartridges.

Creating and modeling cartridges in Design Studio is a design-time activity. You do not use Design Studio to manage a run-time environment. Once you have created cartridges in Design Studio, you can build the cartridges and then deploy them into a run-time OSM environment. Deploying is the act of installing a packaged cartridge to the run-time environment, which is where orders are processed, and where end users login and use the OSM Order Management web client or Task web client to view and manage orders and tasks. The run-time environment could be an OSM server that is installed on the local computer of a developer along with Design Studio, or could be a dedicated OSM environment setup with some server hardware. You do not require an OSM run-time environment to use Design Studio to model and build cartridges, but you do require an OSM run-time environment to deploy and test cartridges with test orders.

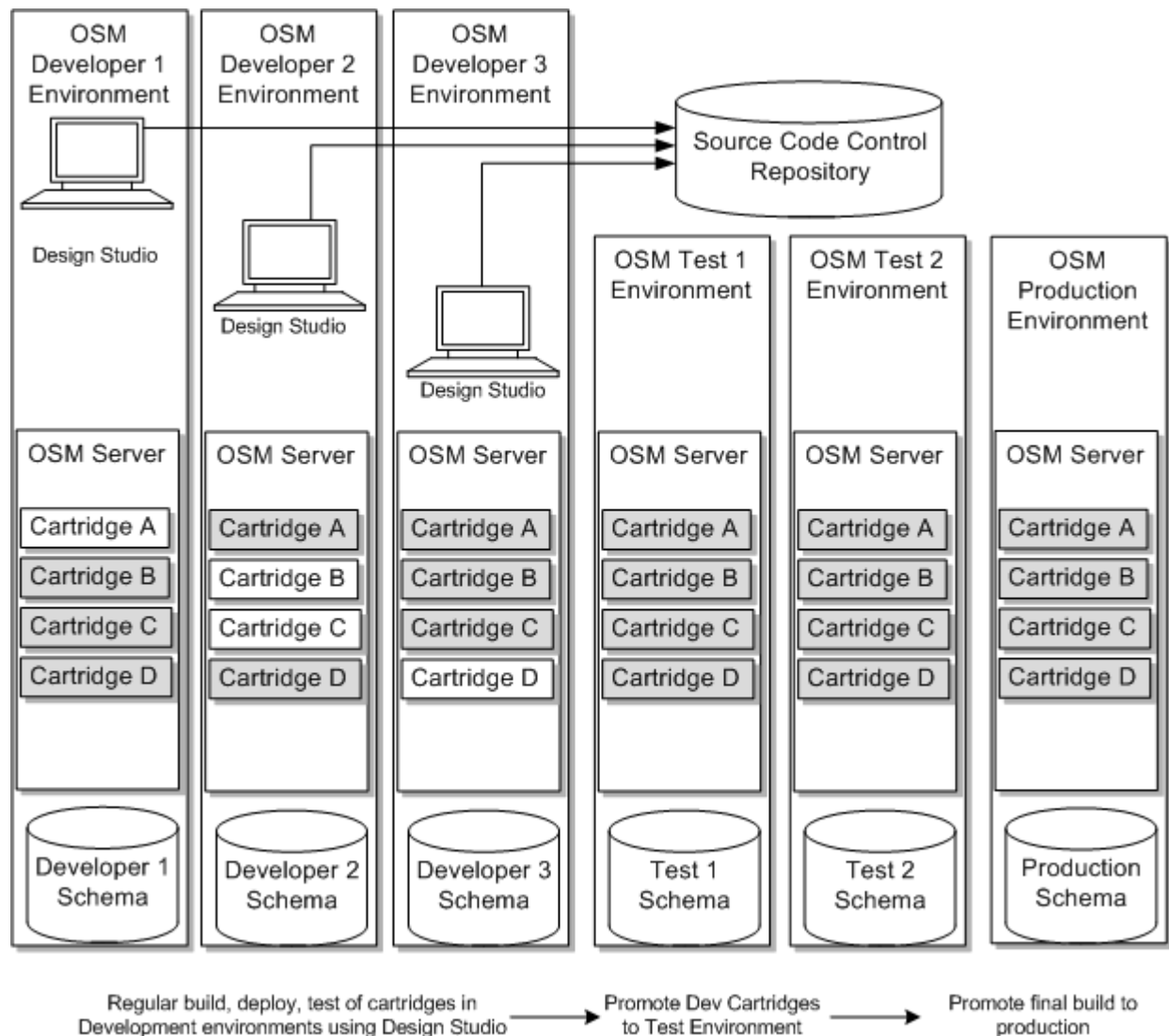
An OSM solution is typically made up of multiple cartridges. Each cartridge is a building block that specifies an aspect of the OSM solution. For example, some cartridges may define how OSM interacts with particular fulfillment systems. Other cartridges may define data dictionaries common across applications, and shared across other OSS applications such as Oracle Communications Unified Inventory Management (UIM) or Oracle Communications ASAP. Some cartridges are more foundational to an OSM implementation, such as those that specify how orders are represented and how order decomposition and orchestration occur. Together, they contain the directives of the OSM implementation for a particular deployment such as OSM running in the central order management (COM), service order management (SOM), and technical order management (TOM) roles.

Cartridges allow you to decouple solution behavior from the core OSM system that the cartridges run on. This decoupling allows you to upgrade to a newer versions of OSM to take advantage of improvements without needing to make extensive changes to the solution cartridges.

Cartridges can also help you manage development cycles. You can use cartridges to divide OSM development work into logical pieces among development team members contributing to the implementation. For example in [Figure 19-1](#), OSM developer 1, 2, and 3 work in parallel on different component cartridges in the solution. You can also port cartridges between OSM

environments. This is useful for sharing between developer team members, promoting cartridges from development to test environments, or from test to production.

Figure 19-1 Cartridge Portability in Development, Test, and Production Systems



About OSM Cartridge Scope

Cartridge scope refers to which entities an OSM order has access to. If a particular OSM entity is considered in scope for an order, that entity can be used to influence how that order is processed.

The scoping mechanism of OSM has evolved over the years. For information about the history of scoping in different releases of OSM, see knowledge article 2077384.1, **Cartridge Management and Versioning**, on the Oracle support website:

<https://support.oracle.com>

Scope of OSM Entities Without Namespaces

Some OSM entities, such as orders, processes, tasks, composite cartridge views, and resource files, do not have namespaces defined for them.

Design Studio Entities

Design Studio entities that do not have namespaces have visibility only to other entities contained within the same cartridge. At run time, they cannot be referenced by an entity that resides in another cartridge or another version of the same cartridge.

However, at design time (in Design Studio), you can reference entities across cartridges even if those entities do not have namespaces. For example, when designing a process flow, you can include a task from another cartridge. When you do this, Design Studio includes the referenced entity in the cartridge when it is built, so that it will be available at run time.

XML Catalogs and Resource Files

In addition to specific entities in Design studio, an OSM implementation typically contains other resource files in the cartridges. XQuery and XSLT files are the most common types of resource files, but there can also be other types of files such as Java files or other XML configuration files. In addition, there may be XML catalog files. The scoping of these resource files is the same as with other entities that do not have namespaces:

- For standalone cartridges, the resource entities belong to a local or global pool, depending on the configuration of the FullScopeAccess parameter in the **oms-config.xml** file. For more information about this setting, see "[Standalone Cartridge Scope](#)."
- For solutions using composite cartridges, the scope of resource files is defined by the contents of the composite cartridge.

Scope of OSM Entities with Namespaces

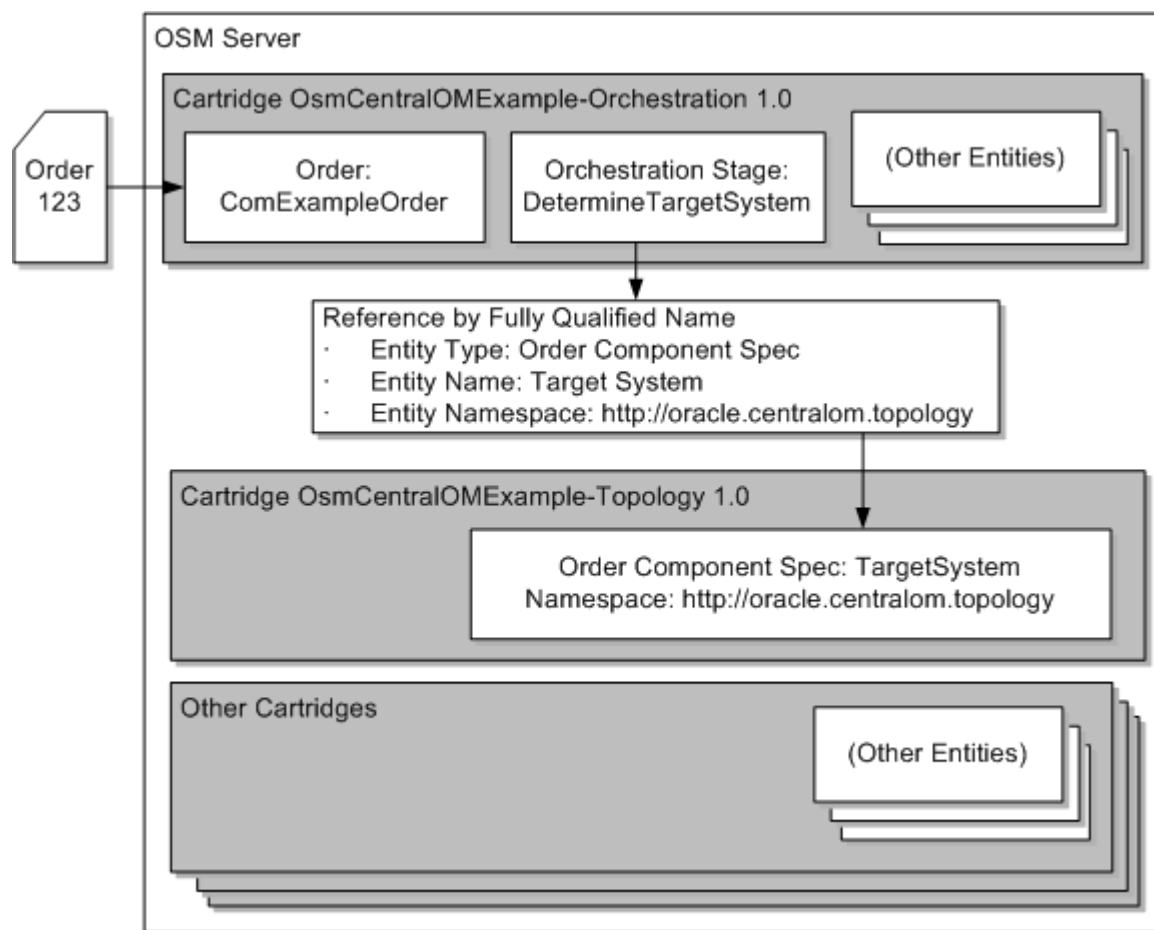
For orchestration entities such as order component specifications, order item specifications, and order recognition rules, there is an explicit namespace attribute that you can specify in Design Studio.

Entities with namespaces are referenced by their fully qualified names. The fully qualified name is the combination of:

- Entity type – for example, **Order Component Specification**
- Entity name – for example, **TargetSystem**
- Entity namespace – for example, **http://oracle.centralom.topology**

[Figure 19-2](#) is a simple example of how an entity with a namespace is referenced by another entity. Referencing by fully qualified name occurs automatically when the cartridges are packaged at build time, as long as you provide appropriate entity namespace values in Design Studio.

Figure 19-2 Referencing an Orchestration Entity by Fully Qualified Name



The cartridge name (OsmCentralOMExample-Topology) is not part of the fully qualified name. Entities can freely reference entities of other cartridges as long as the fully qualified name is unique in the run-time environment.

Standalone Cartridge Scope

Scope considerations are different depending on whether your cartridges are standalone or grouped using composite cartridges. For more information about the different cartridge types, see "[About Cartridge Types](#)."

OSM builds a resource pool for each cartridge that contains an order when it is loaded. The contents of this pool are determined by the **oracle.communications.ordermanagement.resource.FullScopeAccess** parameter in the **oms-config.xml** file. By default, this parameter is set to restrict the resource pool to only the resources in the cartridge. You can also set it to enable either all cartridges or specific cartridges to access all of the resources in your solution. For more information about this parameter, see the discussion of **oms-config.xml** in *OSM System Administrator's Guide*, or see knowledge article 1568944.1, **Cartridge Resources Not Picked Up Correctly for Multiple Cartridge Versions**, on the Oracle support website.

For cartridges that have access to all resources, resources in this pool are accessed according to three levels of priority:

1. Resources in the local cartridge version
2. Resources in other cartridge versions with the same cartridge namespace (with no determined order between versions)
3. Resources in all other cartridges (again, with no determined order)

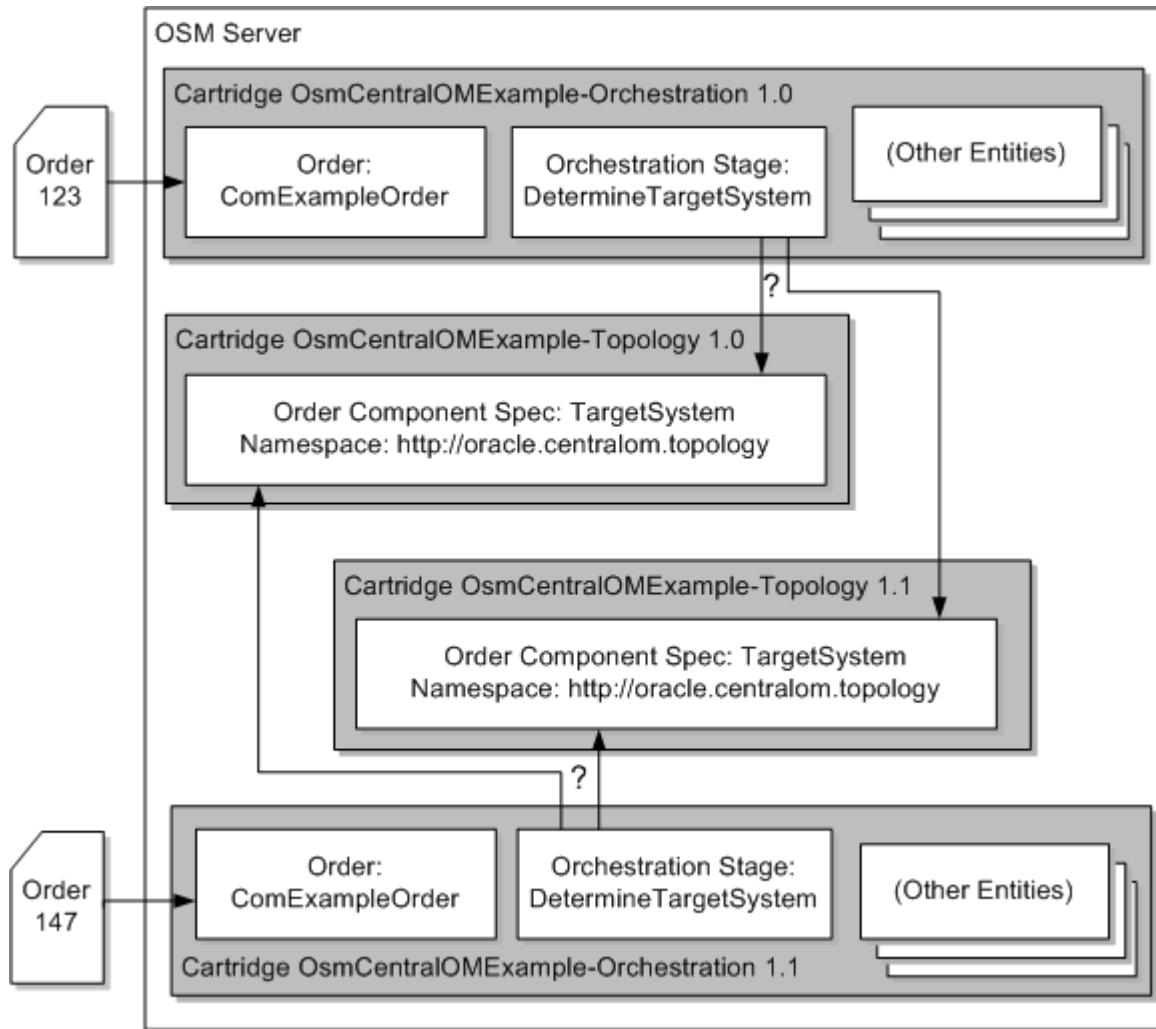
The different access priorities in the pool mean that it is more likely that the OSM server will find the correct copy of an entity with a namespace, but does not completely eliminate the potential for namespace collisions.

XML Catalog Files in Standalone Cartridges

If you are using standalone cartridges, it is important to locate the XML catalog files in the same cartridge as the entities that call them, or the wrong file can be picked up, meaning that the wrong namespace translation will take place, and therefore the wrong resources will be selected.

Avoiding Namespace Collisions for Design Studio Entities

A namespace collision may occur when multiple entities with namespaces have the same fully qualified name in a run-time environment. This is not unusual when multiple versions of the same cartridge are deployed. For example, in [Figure 19-3](#), the TargetSystem order component has the same fully qualified name in both versions 1.0 and 1.1.

Figure 19-3 Namespace Collision of an Orchestration Entity

In the event of namespace collision, it is not possible to predict which version will be used. In the example, it is unknown whether version 1.0 or 1.1 of **TargetSystem** will be loaded by OSM, but both versions 1.0 and 1.1 of the **OsmCentralOMExample-Orchestration** cartridge will use the same version of **TargetSystem**, which means one of them is using the wrong version.

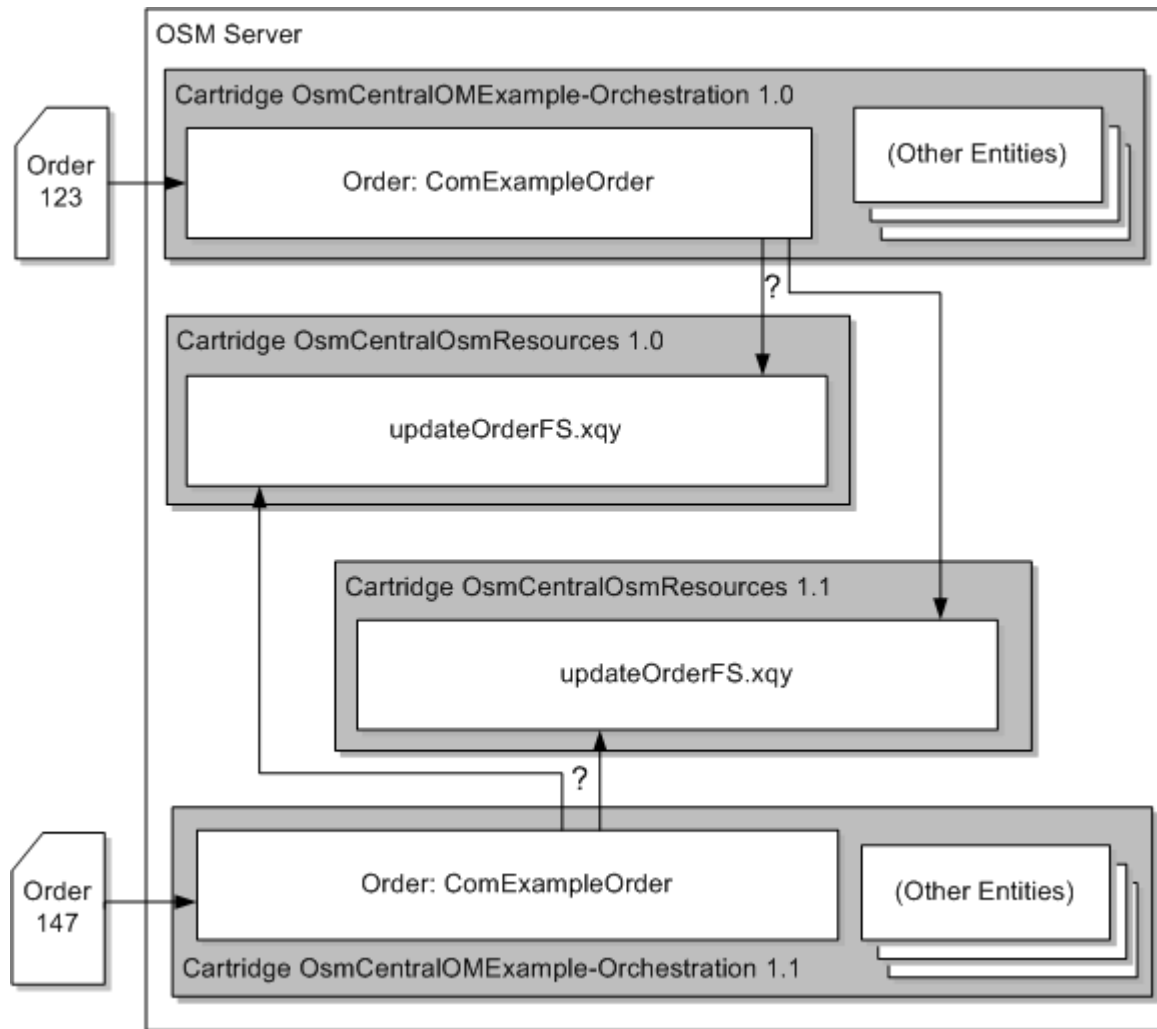
To avoid namespace collision across cartridge versions, you can include the cartridge version in the entity namespace. In the example above, the namespace can be specified as `http://oracle.centralom.topology/1.0.0.0.0`. The best way to do this is using an XML catalog, which you can use to translate the namespaces of entities so that they include the cartridge version. See ["Using XML Catalogs to Support Cartridge Versioning"](#) for details.

Avoiding Namespace Collisions for Resource and XML Catalog Files

In the same way as for other entities without namespaces, if you are using standalone cartridges, it is possible for your solution to reference an unintended version of a resource file when multiple cartridge versions exist. You can avoid this problem for XQuery files if you locate them using a URI that is mapped to a cartridge-version specific location using the XML Catalog.

Figure 19-4 contains an example where the OsmCentralOMExample-Orchestration cartridge references an XQuery file located in a centralized resource cartridge OsmCentralOsmResources, and that two versions, 1.0 and 1.1, have been deployed:

Figure 19-4 XML Catalog Conflict



If the XQuery file is referenced using a URI like the following:

```
http://oracle.centralom/base/xquery/updateOrderFS.xqy,
```

and the XML Catalog for cartridge version 1.0 has a rewriteURI entry like this:

```
http://oracle.centralom/base -> osmmodel:///OsmCentralOsmResources/1.0.0.0.0/resources
```

and the XML Catalog for cartridge version 1.1 has rewriteURI entry like this:

```
http://oracle.centralom/base -> osmmodel:///OsmCentralOsmResources/1.1.0.0.0/resources
```

then there is no ambiguity when locating the XQuery file. Entities in the OsmCentralOMExample-Orchestration cartridge version 1.0 will find the XQuery resource in version 1.0 of the OsmCentralOsmResources cartridge, because the URI of the XQuery file will be resolved to:

```
osmmodel:///OsmCentralOsmResources/1.0.0.0/resources//updateOrderFS.xqy
```

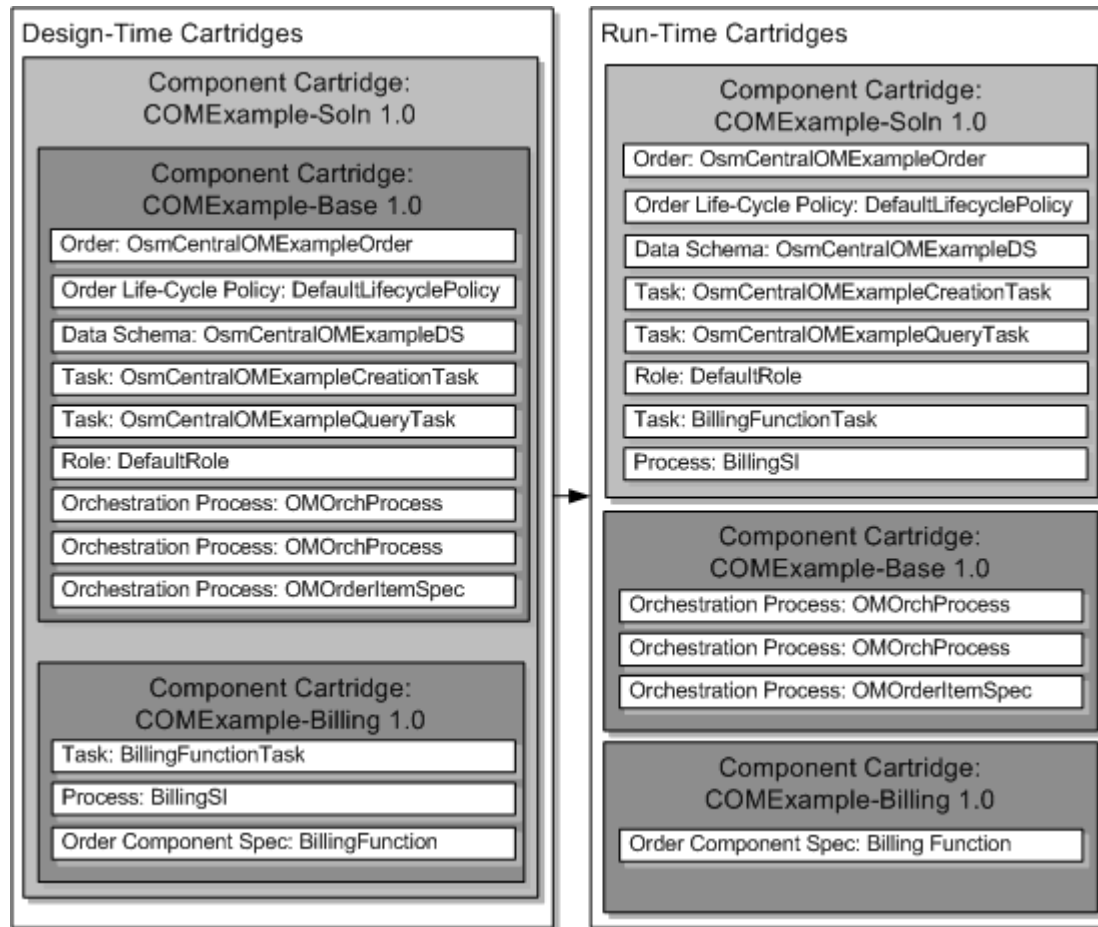
Composite Cartridge Scope

A composite cartridge is a special type of cartridge that does not directly contain any OSM entities at design-time. Instead, it contains references to other cartridges (referred to as component cartridges) that make up an OSM service or solution. The composite cartridge has its own 5-segment version number that you can manage, and contains a list of component cartridges that make up the composite cartridge. For more information about composite cartridges, see *Design Studio Modeling OSM Orchestration Help*. Oracle recommends that you use composite cartridges in your solution, as they help reduce versioning problems and namespace collisions.

The composite cartridge provides solution-level scoping boundaries that are appropriate for running concurrent cartridge versions of an OSM implementation. When the composite cartridge is built, all of the entities without namespaces from the component cartridges are aggregated to become one new deployable cartridge. This run-time cartridge has a namespace that is the name of the composite cartridge. However, all of the entities with namespaces are built into their respective component cartridges as usual.

For XML catalog files in composite cartridges, unlike in standalone cartridges, it is only necessary to ensure that the XML catalog is located somewhere with the scope of the composite cartridge to ensure that the correct version will be used.

[Figure 19-5](#) is an example of a composite cartridge having two component cartridges (a base cartridge and a billing cartridge), and how the run-time cartridges will be generated by the build process:

Figure 19-5 The Use of Composite Cartridges in Design Time and Run Time

In this example, a run-time cartridge named COMExample-Soln is generated to contain all of the entities without namespaces, such as orders, processes and tasks. The component cartridges, COMExample-Base and COMExample-Billing, still contain their entities with namespaces. When the composite cartridge is deployed, all of the design-time component cartridges and the generated component cartridge are deployed to the run-time environment. When an incoming order is handled by an order entity in this composite cartridge version, it is given the namespace of the generated cartridge, which is the name of the composite cartridge, COMExample-Soln.

Because all of the entities without namespaces are aggregated into a single cartridge, there is no ambiguity about which entity is referenced. Even if multiple versions of the cartridge are present in OSM, only one version of the cartridge would be included in the composite cartridge. In our example, if cartridge COMExample-Billing version 1.0 and 1.1 are deployed, the BillingFunctionTask task from the COMExample-Billing cartridge version 1.0 will be used by orders that are tied to COMExample-Soln version 1.0, since the order and the task are now confined to a single cartridge and version.

Composite cartridges also address the problem of namespace collision for entities with namespaces. The composite cartridge establishes a manifest that explicitly specifies the component cartridge version whenever an entity dependency is established across cartridges in the solution. Entities and resource files in the component cartridge version are confined to the OSM solution defined by the composite cartridge. So, in the example, if the COMExample-Billing version 1.0 and 1.1 cartridges are both deployed, orders tied to COMExample-Soln 1.0

will use entities from version 1.0 of COMExample-Billing instead of version 1.1, even when the same fully qualified name exists in both, because the. This means that, when you use composite cartridges, you do not need to include the cartridge version in the namespace of entities to avoid namespace collisions.

Special Cases for Scope

Some entities are special cases with regards to scope. First, there is the order recognition rule. Since the order recognition rule is evaluated before the order is selected, its scope cannot be based on the cartridge in which it is located. Also there is the fulfillment pattern, which works differently because the namespace used is based on the order item.

Order Recognition Rules

Order recognition rules (ORRs) are a special case for scoping. When the CreateOrder API is called, OSM does not know the version or namespace of the incoming order. The purpose of ORRs is to be able to determine the kind of order contained in an inbound message. Even though an ORR has a namespace field, its scope is system-wide: all ORRs in the OSM run-time environment are used to recognize the contents of the CreateOrder message, regardless of which cartridge contains the ORR. This is true for ORRs regardless of whether they are in standalone cartridges or component cartridges contained by a composite cartridge. Essentially, ORRs ignore both cartridge version and namespace.

So if you have two versions of an ORR that are both deployed and will pick up the same order type, there is a non-namespace way to configure which ORR will pick up new orders of the relevant type. The **Relevancy** setting in Design Studio determines the order in which the ORRs are evaluated. OSM will run ORRs on an inbound message from highest to lowest relevancy until an ORR recognizes the order. If two ORRs have the same relevancy, it is not predictable which one will be evaluated first. Once an ORR is matched, the incoming order is tied to the target order type (and thus its cartridge namespace) that the ORR designates, and further rules are not evaluated.

Fulfillment Patterns

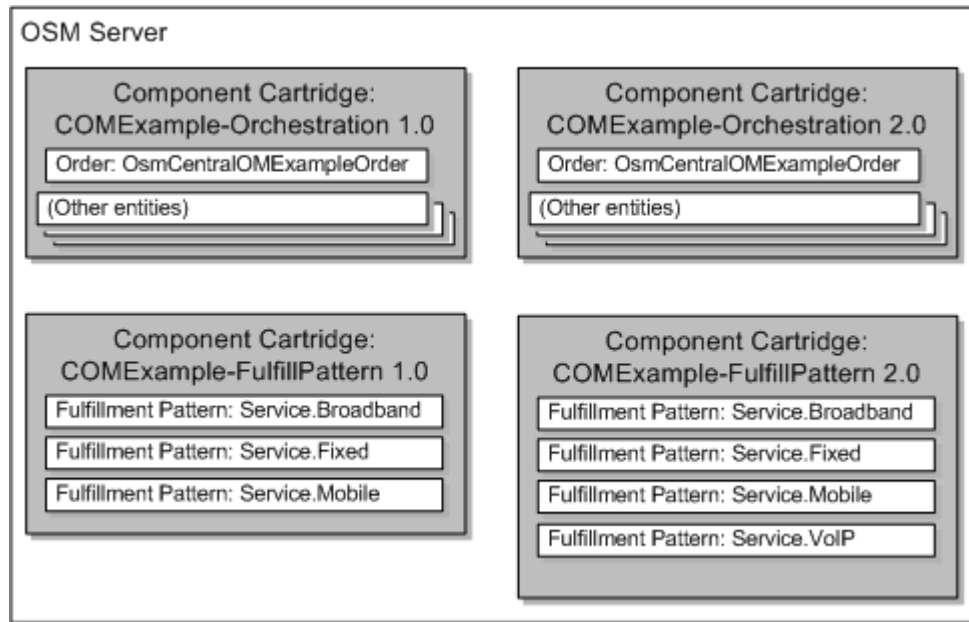
Fulfillment patterns are responsible for determining the fulfillment of an order item in the generated orchestration plan. Fulfillment patterns have a namespace. However, finding the fulfillment pattern of an order item is a unique case in scoping. In Design Studio, the order item is mapped to a fulfillment pattern using the value in the **Fulfillment Pattern Mapping Property** field, which contains a string to map to a fulfillment pattern name. OSM finds the Fulfillment Pattern by fully qualified name (not just the string contained in the property field), using the namespace of the order item to determine what namespace to look for in a fulfillment pattern. Because of this, it is important to use the same namespace for both order items and their related fulfillment patterns in your OSM solution. Otherwise, OSM cannot find the fulfillment patterns for the order items.

Fulfillment Patterns in Standalone Cartridges

In the same way that there can be scoping issues with regular entities in standalone cartridges, there can be scoping issues for fulfillment patterns as well. The example depicted in [Figure 19-6](#) shows what happens if the version number is not included in the namespace of an order item in a standalone cartridges. The COMExample-FulfillPattern version 1.0 cartridge contains the Service.Broadband, Service.Fixed, and Service.Mobile fulfillment patterns. The COMExample-FulfillPattern version 2.0 cartridge introduces the new Service.VOIP fulfillment pattern as well as making changes to existing fulfillment patterns. If the namespaces do not contain cartridge versions, finding **Service.Broadband** results in matching whichever of the

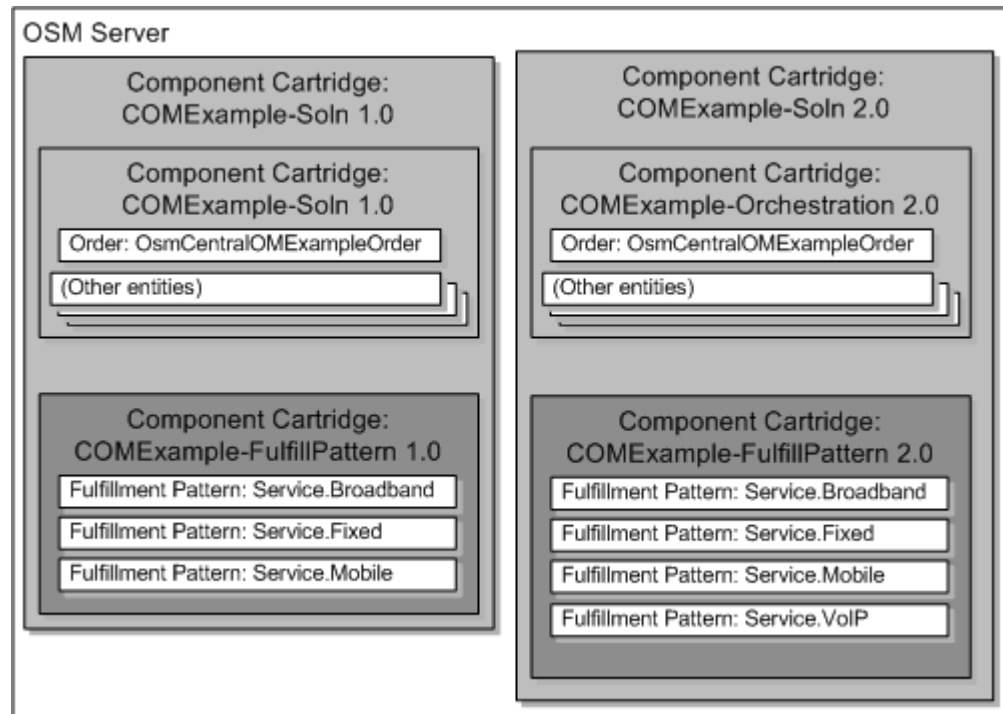
fulfillment patterns was in the version of the cartridge that happens to have been deployed most recently. This results in unpredictable behavior in orchestration plan generation.

Figure 19-6 Fulfillment Patterns in Standalone Cartridges



Fulfillment Patterns with Composite Cartridges

In contrast, with the use of composite cartridges, OSM matches only fulfillment patterns that are encapsulated in the same composite cartridge. In the example in [Figure 19-7](#), for orders in cartridge version 1.0, only Service.Broadband in COMExample-FulfillPattern1.0 is matched.

Figure 19-7 Fulfillment Patterns Encapsulated in Composite Cartridges

Managing Cartridge Versions

To distinguish changes in a cartridge over time, each cartridge has a cartridge version. As your OSM implementation evolves to keep up with business, you can either introduce cartridge changes by overwriting the existing cartridge version, or by deploying as a new cartridge version. The longer the order lifetime is, the greater the likelihood that you will need multiple cartridge versions deployed at the same time.

Changes can be made to cartridges for different reasons, including fixing implementation details and enhancing the cartridge to meet new requirements. Often, the changes that need to be made in a cartridge are not compatible with the way the cartridge currently works, or may be disruptive to the in-progress orders that are running in the current version of the cartridge. This is the primary reason to introduce a new version: deploying a new version of a cartridge allows new orders to use the new cartridge, while in-flight orders can continue without disruption using the version of the cartridge that they started with.

One consideration to keep in mind when planning solution maintenance is the order lifetime. The time needed to fulfill orders for a service provider varies among different domains of products and services. For example, consumer orders for mobile services may be seconds or minutes to complete, while consumer orders for fixed line services may take minutes to days, particularly if those orders require human interactions such as physical equipment adjustments or shipments. Business services may take days, weeks, or even months to complete, where an order can encapsulate the services of multi-site network with lots of equipment, off-net components, cabling, and so on.

In OSM, existing orders that are running in the system will continue to run against the existing cartridge version that they were using. For example, if order 123 is in progress using cartridge A version 1, and then cartridge A version 2 is deployed to the run-time system, then order 123

will continue to run to completion using cartridge A version 1. This is also true for existing future-dated orders for cartridge A version 1 even if the start date has not arrived.

Making Changes to Existing Cartridge Versions

If you have very short-lived orders, it is possible to avoid having different versions of cartridges at the same time, by deploying changes to the cartridges without changing the version. To do this without breaking existing orders, a possible strategy is to allow OSM to complete the fulfillment of all orders, followed by an order purge operation that removes them from the environment (since, once the new cartridge is deployed, OSM may not be able to understand the old orders). However, there are several drawbacks to this strategy:

- It does not allow you to keep completed orders in the system for troubleshooting, auditing, and reporting purposes.
- It limits your implementation's ability to handle inter-order dependencies, because you cannot guarantee that the parent orders are still in the system.
- There are also operational complications that can arise, because there will be some orders that take longer to complete than others, and there may be order fallout that needs to be handled. This would mean that at least some order fallout would need to be handled outside of OSM, rather than managing it using the order fallout capabilities of OSM.

Aside from the scenario mentioned above, where you have only very short-lived orders, you should always plan to use cartridge versioning in your production environment.

However, even if you are using cartridge versioning in your solution, you do not need to introduce a new cartridge version every time you make a change. There are cases where it is better to make changes to an existing cartridge version and redeploy it, instead of creating a new cartridge version, such as:

- If you have critical bug fixes that should affect existing orders, and the fixes would not make the cartridge incompatible with existing orders, it may make sense to deploy the fix to the same cartridge version that causes the problem.
- If you have changes need to become effective immediately, applying to existing in-progress orders, you may want to deploy the changes in the existing cartridge version. For example, if there is an interface change in a downstream system, and it no longer supports the original version of the interface, that would require OSM to adapt to the new interface across all cartridge versions.

Generally speaking, the longer that orders take to complete, the more business pressure there is to make changes that existing orders can use.

When when you need to make changes to existing cartridge versions, it is important that you ensure that all in-progress orders can continue to be run against the cartridge after the changes are made. For technical and best practice guidance towards safely making changes to existing cartridge versions, see knowledge article 2077384.1, **Cartridge Management and Versioning**, on the Oracle support website.

Making changes to existing cartridge versions is inherently risky, and so adequate test coverage is an essential part of the process, to ensure that all in-progress order scenarios are not impacted (for example, do not stop processing) by the changes to the cartridges.

In general, it is important to minimize the number of cartridge versions that you introduce for performance reasons. See the discussion of cartridge management strategy in *OSM System Administrator's Guide* for more information about removing old versions of cartridges when they are no longer necessary.

Handling Multiple Cartridge Versions

The Cartridge and Composite Cartridge editors in Design Studio contain several fields that make up the version number. There are five fields that you set to indicate the version of a cartridge: **Major Version Number**, **Minor Version Number**, **Maintenance Pack**, **Generic Patch**, and **Customer Patch**. These constitute a 5-digit version number, for example, 1.0.0.0.0. In addition to the cartridge version, there is a read-only **Build Number** field that is automatically incremented by Design Studio each time a cartridge is built. It is not reset when the cartridge version is changed.

To deploy a new cartridge version, change the value of the version fields and deploy the cartridge. Multiple cartridge versions can be deployed in an OSM environment at the same time, with orders running against each version.

Some additional configuration is necessary to deploy multiple versions of a cartridge to an OSM environment. All of the following considerations should be taken into account when implementing multiple cartridge versions:

- The `DEFAULT_CARTRIDGE` cartridge management variable:
Ideally, only one version of a cartridge should be set as the default version of the cartridge. For example, if you have versions 1.0.0.0.0 and 2.0.0.0.0 of an OSM cartridge deployed, only one of them should be set as default. For more information, see "[Designation of the Default Cartridge Among Cartridge Versions](#)."
- Composite and component cartridge versions:
When you update a component of a composite cartridge, you do not always need to update the version of the composite cartridge as well.
For example, if CompositeCartridge version 1.0 references ComponentCartridgeA version 1.0 and ComponentCartridgeB version 1.0, when you update ComponentCartridgeA to version 1.2, the composite cartridge and ComponentCartridgeB can both remain at version 1.0.
- Cartridge versioning using the XML Catalog:
In standalone cartridges, the XML Catalog should be used to allow multiple cartridge versions to refer to their own set of resources using the cartridge model variable `CARTRIDGE_VERSION`.
If a **rewriteURI** entry in the XML catalog contains a version-specific portion in the URI such as "1.0.0.0.0" in the following:

```
<rewriteURI uriStartString="http://example.com/" rewritePrefix="osmmodel:///MyCartridge-Resources/1.0.0.0.0/resources"/>
```


the version-specific portion of the `rewriteURI` entry must be updated to point to the correct cartridge version.
See "[Using XML Catalogs to Support Cartridge Versioning](#)" for more information.
- Automation – External Event Receiver:
When there are multiple versions of automation external event receivers listening to the same JMS Source, OSM uses the `JMSCorrelationID` to ensure that the message is consumed by the correct receiver, as long as the external automation receiver is named using the format `taskName.automatorName`, and there is only one external automation receiver associated with the task.
If your receiver does not have the name format `taskName.automatorName`, or there is more than one external automation receiver associated with the task, the message

listening filter criteria of your automation plug-in must guarantee not to pick up a message that should have been picked up by another cartridge version. This may happen if, for example, your system has asynchronous interaction with an external system that takes days to fulfill your request and you have modeled the correlated response to return to a different task than the one that sent the message, or it might happen if you have an old (pre-OSM 7.0.3.1) cartridge that you have not updated.

See "Properties View External Event Receiver Tab" in *Modeling OSM Processes* for the **External Event Receiver** sub-tab of the properties view in the automated task editor **Automation** tab for more information.

- Order recognition rule:

When there are multiple versions of a cartridge with orchestration entities, order recognition rules should be modeled to recognize a specific version of the order instead of the default version. To recognize a specific version of the order, the Target Order Version of the order recognition rule should be set to the version of the cartridge where the specific version of the order resides.

When an order recognition rule is used in a composite cartridge and there are multiple versions of the composite cartridge, the Target Order Version of the order recognition rule should be set to the version of the composite cartridge that contains the target order as part of the solution. For example, you might have version 1.0.0.0.0 of the OsmCentralOMExample-Solution composite cartridge with the following dependent cartridges:

- OsmCentralOMExample-Orchestration version 1.2.0.0.0 – OsmCentralOMExampleOrder is defined here
- OsmCentralOMExample-ProductSpec version 2.0.0.0.0
- OsmCentralOMExample-FulfillmentPattern version 2.0.0.0.0
- OsmCentralOMExample-Topology version 1.1.0.0.0

The target order version of the order recognition rule should be set to 1.0.0.0.0, because that is the version of the composite cartridge.

Migrating Orders to a New Version of a Cartridge

Only orders that are in the Not Started order state can be migrated to another cartridge or cartridge version. All running orders (in any order state other than Not Started), including future-dated orders, must continue to run to completion against the version of the cartridge in which they were created.

It is possible to mimic migrating orders to a new cartridge version by re-submitting in-flight orders as new orders to the new cartridge version. This requires either the in-progress fulfillment flow to be manually cleaned up in the various external systems before the order is resubmitted, or the flow itself to be configured so that past activities can be repeated without needing to be undone (for example, so that you can resend a message to a downstream system without having to undo any previous commands). This is not a recommended option, but can be a possibility, depending on the specifics of your OSM implementation.

Designation of the Default Cartridge Among Cartridge Versions

If there are multiple versions of the same cartridge deployed, you must designate which version is to be used for to an inbound order. It is possible (but not required) to specify the cartridge version in the CreateOrderbySpec request. In the CreateOrder request, the target version cannot be specified as input parameter, but can optionally be defined by the matched ORR. Alternatively, you can configure the ORR to use the default version of the cartridge. In

any case, there must be a way to determine which version of the cartridge should handle the order if the cartridge version is not defined on the order or set by the order recognition rule.

For standalone cartridges, the default designation is configured using the `DEFAULT_CARTRIDGE` cartridge management variable. This variable should be set to **true** for the version that should handle new orders, typically the latest cartridge version. For instance, if version 1 of cartridge A is already deployed and has orders running against it, and version 2 of cartridge A is deployed as the new default version, then new orders created in the run-time system for cartridge A will run against version 2, and any changes made for version 2 will be effective for those new orders.

For composite cartridges, the default is set in the same way, only it is set on the composite cartridge. When the `DEFAULT_CARTRIDGE` cartridge management variable is set to **true**, all the composite cartridge's component cartridges are considered the default versions. The default settings of the component cartridges do not have an effect, only the setting of the composite cartridge.

For both standalone and composite cartridges, the OSM server always recognizes exactly one version as the default for each cartridge namespace. When multiple cartridge versions are deployed that have the default flag set to **true**, the OSM run-time environment will make the last deployed of these versions the default cartridge. Because of this, special attention is required when redeploying an old cartridge version. When you create a new cartridge or composite cartridge in OSM, by default, the `DEFAULT_CARTRIDGE` cartridge management variable is set to **true**. When you deploy the versions in numeric order, the latest version will be the default. However, if you redeploy an earlier version after a later version, you must ensure that you have set the `DEFAULT_CARTRIDGE` cartridge management variable to **false** for that earlier version. There is no warning in Design Studio or on the run-time server that there is an older version of a cartridge being deployed as the default, so you must take care to set the value properly.

Handling Revision Orders When Multiple Cartridge Versions Are Deployed

OSM always creates revision orders with the same cartridge version as the base order. This is because otherwise, generating and executing compensation can cause errors because entities in the new version are not available in the original cartridge version. The detection of order revision and the choosing of cartridge version are handled automatically by the OSM server.

For example, order 123 is created against cartridge A version 1.0, and is currently in an In Progress state. Next, version 1.1 of cartridge A is deployed and is now the default version of the cartridge, and all new orders for cartridge A will be run against version 1.1. Then a revision for order 123 is submitted to the system. When OSM detects that this is a revision of order 123, and that order 123 is running against version 1.0 of cartridge A, it creates the revision order for version 1.0 of cartridge A, then proceeds with the amendment process.

This means that all subsequent revisions of the order will be created against the same cartridge version as the original order. There is no way to override this behavior. Regardless of any information set on the order or by the ORR, revision orders will use the same cartridge version as the original order.

Working with Cartridges in OSM Cloud Native

For cartridge considerations in an OSM cloud native environment, see "Preparing Cartridges for OSM Cloud Native" in *OSM Cloud Native Deployment Guide*.

Building and Packaging a Cartridge

Use Design Studio to package a cartridge by specifying entities to include in the cartridge. By default, all entities created within the cartridge are included unless otherwise specified on the Order and Service Management Cartridge editor **Packaging** tab.

To build cartridges:

- From the **Project** menu, select **Build**.

Oracle recommends that you periodically clean the project prior to a build (see "[Cleaning and Rebuilding Cartridges Prior to Deployment](#).")

For instructions on how to package and build a cartridge, see "Packaging and Deploying OSM Cartridges" in *Modeling OSM Processes*.

About Generating OSM Cartridges and Deployment Options

When you build OSM cartridges, Design Studio generates a portal archive (PAR) file for each cartridge, which is a ZIP file packaged to contain the metadata interpretable by the OSM run-time environment. This PAR file has the .par file extension, and is the artifact sent to the OSM server when deploying a cartridge. The name you choose for the cartridge becomes the name of the PAR file. Design Studio saves the PAR file to the *cartridgeName/cartridgeBin* directory, that you can view from the Java perspective Package Explorer.

Both the design-time project files in Design Studio and the deployable PAR file stored on the OSM run-time environment are referred to as cartridges. Often, the context of the discussion clarifies which artifact is being referred to.

You can deploy OSM cartridges using either of the following tools:

- **Design Studio environment perspective:** Typically, developers and testers managing their own run-time environments use Design Studio to deploy cartridges. Developers may perform build, deploy, and test cycles many times a day. You can deploy cartridges from Design Studio using the Studio Environment Perspective. See *Design Studio Concepts* for more information about deploying cartridges using the Design Studio environment perspective.
- **Design Studio cartridge management tool (CMT) (Traditional OSM Only):** Oracle recommends CMT (packaged with Design Studio) for deploying cartridges to production, pre-production, or automated testing environments. The cartridge management tool is a set of ANT scripts for deploying and un-deploying cartridges to a run-time environment. Script-based deployment is important for run-time environments that are under strict operational control and require an automated, repeatable way to build, deploy and un-deploy cartridges. See *Design Studio Developer's Guide* for more information about CMT.
- **OSM DB Installer:** OSM cloud native uses a different mechanism for deploying cartridges. See "Deploying Cartridges Using the OSM DB Installer" in *OSM Cloud Native Deployment Guide* for more details.

Both tools connect to the OSM server using the Cartridge Management Web Service (CMWS) deployed in WebLogic.

Note

The XML Import/Export (XMLIE) application is a legacy client that can manage cartridges. It does not connect over CMWS. Do not use XMLIE to deploy cartridges unless you are supporting an OSM 6.x implementation. See *OSM System Administrator's Guide* for more information about XMLIE.

You can configure how orders are fulfilled by deploying cartridges in different ways. For example:

- You can deploy different cartridges on different instances of OSM. For example, you can deploy a specific set of cartridges on an instance of OSM that is dedicated to central order management.
- You can make changes to a cartridge after the cartridge has been deployed to the OSM server by making changes to the original cartridge in Design Studio and then redeploying the cartridge.
- You can fulfill orders differently by using functionality deployed by different cartridges.
- You can fulfill orders differently based on functionality deployed by different versions of the same cartridge.

For instructions on how to create a cartridge in Design Studio, see "Packaging and Deploying OSM Cartridges" in *Modeling OSM Processes*.

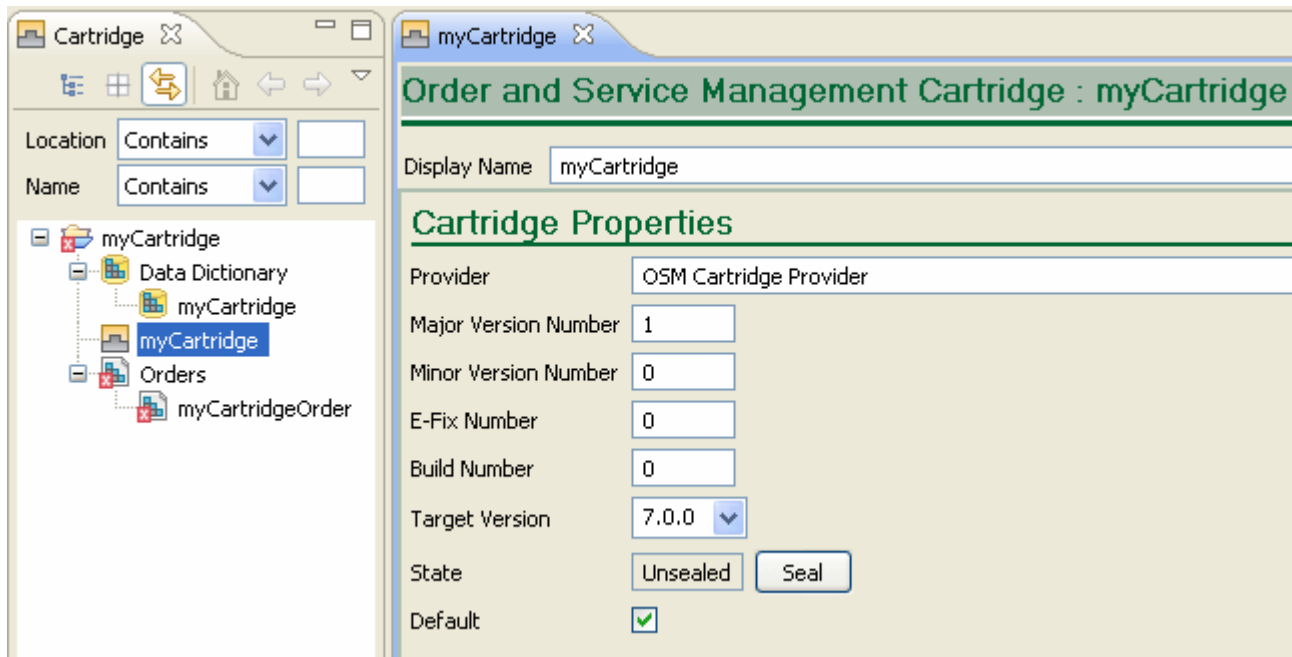
About Cartridge Types

You can create the following OSM cartridge types using Design Studio:

- **Component cartridge:** Component cartridges contain a part of the OSM model entities in an overall OSM solution. For example, some component cartridges may define OSM model entities for interacting with particular fulfillment systems, such as a shipping system, or a billing and revenue management system, or OSM system running in a different role. Other component cartridges may define data dictionaries common across OSS or BSS applications.
- **Composite cartridge:** Composite cartridges designate an OSM solution by referencing a collection of component cartridges. The composite cartridge does not contain any OSM model entities itself but acts as a container that includes the component cartridges it references. For this reason, composite cartridges are also called **solution cartridges**. When you deploy a composite cartridge, all the included component cartridges are also deployed, effectively deploying the entire OSM solution in a single action. Oracle recommends using composite cartridge to manage the component cartridges of an OSM implementation in the production environment. Composite cartridge projects may contain any number of component cartridges, but not other composite cartridges.
- **Standalone cartridge:** Standalone cartridges are component cartridges that are not part of a composite cartridge solution. Standalone cartridges can have dependencies to other standalone cartridges, but cannot be dependent on any component cartridge within a composite cartridge solution.

About Design Studio Editors for OSM Cartridges

[Figure 19-8](#) shows an example of a cartridge, named myCartridge, as it appears in the Design perspective Cartridge view (left side). The corresponding Order and Service Management Cartridge editor is also shown (right side).

Figure 19-8 Cartridge View of a Cartridge

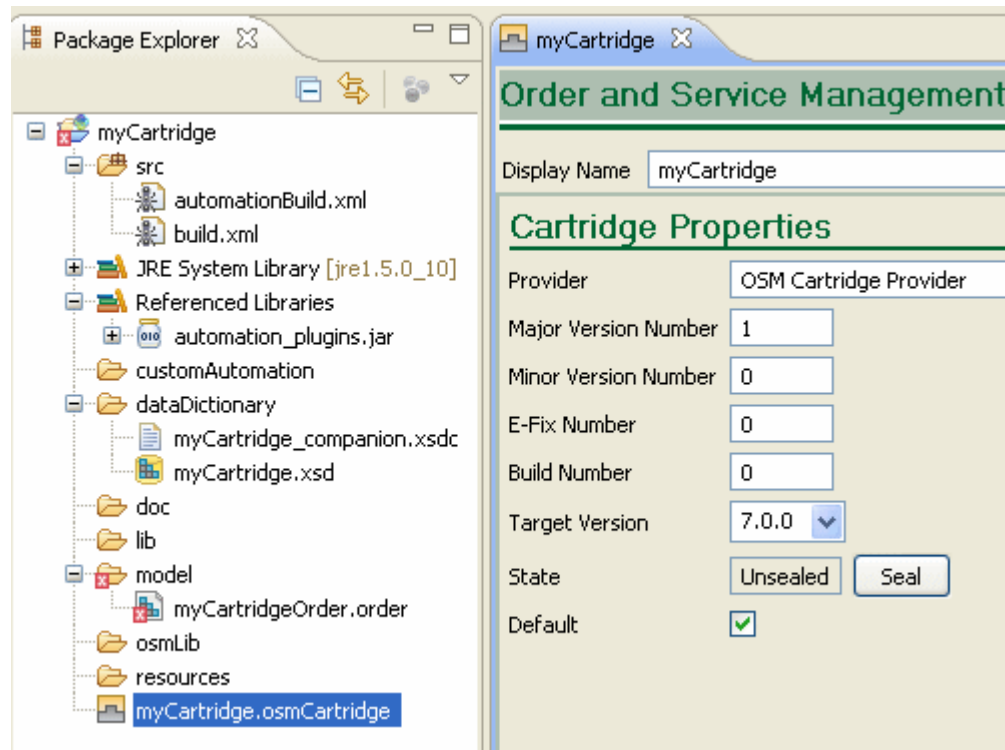
Expand the cartridge in the Cartridge explorer pane (on the left in [Figure 19-8](#)) to see the contents created with each cartridge. For a component cartridge, this includes a default order based on the name of the cartridge. When you initially create a new OSM component cartridge, errors are always present because the default order requires you to define:

- A creation task
- A default process
- A role that grants creation permissions
- An order life-cycle policy
- An order template
- Order permissions

A composite cartridge does not require these entities, so there are no errors when it is initially created. For a component cartridge, after these entities are defined for the order, the errors are resolved, but the graphic will still show the presence of an error by placing a small red "x" box on the lower left corner of the icons in the Cartridge explorer pane. This is because the graphic shows what is present when the cartridge is created. When the errors are resolved, the pane reflects the additional entities of a process and a life-cycle policy that are not part of cartridge creation.

Switching to the Java perspective Package Explorer view and expanding the cartridge displays the file types of the contents created with each cartridge.

[Figure 19-9](#) shows an example of a cartridge, named myCartridge, as it appears in the Java perspective Package Explorer view. The corresponding Order and Service Management Cartridge editor is also shown.

Figure 19-9 Package Explorer View of a Cartridge

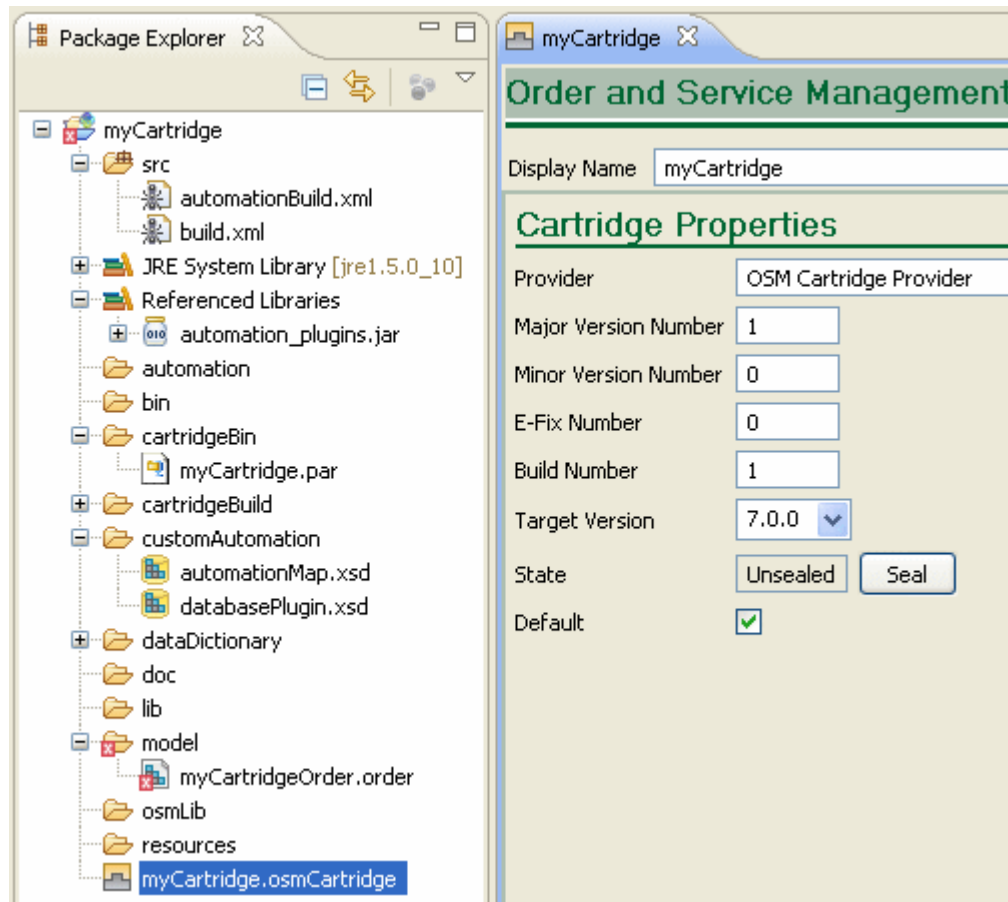
In the example, **myCartridge** was entered in the **Project name** field when creating the cartridge. As a result, the Java perspective Package Explorer view shows:

- **myCartridge:** The Design Studio Order and Service Management Cartridge project.
- **myCartridge/dataDictionary/myCartridge.xsd:** The schema file used internally by Design Studio.
- **myCartridge/model/myCartridgeOrder.order:** The Order editor.
- **myCartridge/myCartridge.osmCartridge:** The Order and Service Management Cartridge editor shown on the right side of [Figure 19-9](#).

After creating the cartridge, an immediate build of the project creates additional directories and files in the cartridge, as shown in [Figure 19-10](#). The directories include, among others:

- **cartridgeBin**
This directory contains the **myCartridge.par** file, which contains the Design Studio entity files and is deployed to the OSM server.
- **customAutomation**
This directory is created with the cartridge, but the **automationMap.xsd** and **databasePlugin.xsd** files are pulled into the cartridge with the build.

Figure 19-10 Package Explorer View of a Cartridge After a Build

**Note**

If working with automation plug-ins, the directories and files listed below are important:

- **src** directory
- **Referenced Libraries/automation_plugins.jar**
- **cartridgeBin/cartridgeName.par**
- **customAutomation/automationMap.xsd**
- **customAutomation/databasePlugin.xsd**
- **resources** directory

For more information, see "[About Automation Plug-ins](#)."

Organizing Design Studio and Naming Conventions

Oracle recommends that you determine a set of naming conventions for the Design Studio entities being created and a directory structure to contain those elements that is appropriate to your implementation. Following is an example set of naming conventions for selected

configuration elements within Design Studio. However, each project team should determine what conventions are suitable for a particular project.

Table 19-1 Suggested Design Studio Naming Conventions

Metadata Element	Naming Convention	Sample
Order recognition rules	Use the convention of <i>OrderType</i> ORR	SalesOrderORR ProvisioningOrderORR
Order item specifications	Use the convention of <i>OrderItemType</i> ItemSpec	CentralOrderItemSpec
Fulfillment patterns	Use a name that indicates the fulfillment flow being supported. For example: <i>PS_FulfillmentType_SpecType</i>	PS_Service_Broadband
Fulfillment modes	Use names that clearly identify the type of action to be taken	DELIVER QUALIFY QUOTE
Order component specifications: Fulfillment actions	Use names that indicate the function of the fulfillment action; for example, <i>Billing</i> , or <i>Shipping</i> .	ResidentialBillingFunction EnterpriseBillingFunction ProvisioningFunction
Order component specifications: Fulfillment target systems	Use names that indicate the function of the target system; for example, <i>Billing</i> , or <i>Shipping</i> .	BillingSystem CRMSystem ServiceManagementSystem
Order component specifications: Processing Granularity	Use names that correspond to the order structure defined in the product catalog; for example, <i>Item</i> , <i>Bundle</i> , and <i>Order</i> .	ItemBased BundleBased OrderBased
Orchestration stages	Use names that describe the stages.	DetermineFulfillmentFunctionStage DetermineFulfillmentSystemStage DetermineProcessingGranularityStage
Orchestration sequences	Use the convention <i>CartridgeNameSequence</i>	SalesOrderFulfillmentSequence
Decomposition rules	Use the naming convention <i>DR_FunctionName_To_SystemName</i> for system decomposition rules Use the naming convention <i>DR_DetermineGranularity_For_FunctionName</i> for granularity decomposition rules	DR_BillingFunction_To_ResBRM DR_DetermineGranularity_For_BillingFunction

Cartridge Packaging Design

When you model OSM entities, you can define separate cartridges and combine them in a single solution. This allows you to create individual cartridges for specific purposes, and to create a library of cartridges which can be shared across multiple solutions. This approach can result in lower maintenance, better performance, and easier collaboration within the implementation team.

While each deployment has its own specific considerations, Oracle recommends that you consider the following guidelines:

- Build separate cartridges based on function. For example, build separate cartridges for COM, SOM, and TOM roles.
- Put configuration elements that are more commonly changed into a separate cartridge for ease of maintenance.
- If a component has distinct notification and status management requirements, use a separate cartridge and pass the data to this "sub-order" from the main order. For example, if the process of shipping a piece of equipment involves interactions with three to four systems and multiple notifications to other systems, consider creating a "Shipping order management cartridge" to handle this requirement.
- For a given cartridge, limit the total number of sub-processes and the number of tasks per sub-process for ease of maintenance. Consider limiting both to ten or fewer, although in some situations more might be required.
- Consider defining a cartridge which contains only a data dictionary with data nodes and structures that are specific to a technology or service or space. Other cartridges can then reference this data.

Modifying the Build

If you need to modify the build performed by Eclipse, you can modify the build files that are provided with the creation of each cartridge. Common modifications include adding logic to the build file for the generation of Java code and the creation of JAR files. The build file for each cartridge is:

- *CartridgeName*/src/build.xml

The *CartridgeName*/src/build.xml file can be customized to add files to the **lib** directory. For example, you may want to get a JAR file from another project as part of the build or do some other custom staging activity. Nothing in the **lib** directory goes on the classpath automatically. You can do this manually as well.

About XML Catalogs

XML Catalogs are logical structures that act like address books or directories. XML Catalogs contain entries that indicate a placeholder location and then provide the path to the location to be used. At run time, when OSM processes a URI you specify as part of the OSM data model, OSM first attempts to resolve the URI against the XML Catalogs you specified. Based on the mapping defined in the XML Catalogs, OSM can update the URI to adapt to different environments by resolving the location of the URI in your data model with the location it is mapped to in the XML Catalog. For example:

- OSM resolves a URI against a test server in a test environment and resolves that URI against a different server in a development environment.
- OSM resolves the location of files in a developer's local workspace to the location of equivalent files available to the OSM server at a generic URI. You might use XML Catalogs in this way for XQuery module import statements that at design time need to refer to files in your local workspace but at run time need to refer to files within the resources directory of a deployed cartridge.
- Your OSM model might reference a resource located on the Internet. If your server deployment runs behind a firewall with no Internet access, you can load the resource behind the firewall and use an XML Catalog to redirect the URI of the Internet location to the location of the resource behind the firewall.

For more information on XML Catalogs and valid XML Catalog entries, see the OASIS web page:

<http://www.oasis-open.org/committees/entity/spec-2001-08-06.html>

See "[Using XML Catalogs in OSM](#)" for information on how you can use XML Catalogs in your OSM development.

Using XML Catalogs in OSM

In Design Studio, you model behaviors such as business rules and other model components, which OSM uses at run time to satisfy your business requirements for order processing. The model components used at run time to manage and fulfill orders are referred to as OSM resources and are often contained in resource files. Examples of resource files include XQuery files, XSLT files, custom JAR files, third-party JAR files, and XML files such as a product class mapping file. There can be a large quantity of resources and some of those resources must reference each other. Resources in OSM can be referenced through URI locators in your data model.

A resource must reside on some physical location on a system. Each system has its own unique directory structure. If you use static values or constants to indicate the location of a resource when defining the URI locator for that resource in your data model, the resource will not be accessible if you deploy your cartridge to other systems where the resource is in a different directory. Thus, using static values to indicate the location of a resource limits the portability of your cartridge solution to other systems or run-time environments. XML Catalogs solve this problem by redirecting the URI defined in your data model to the URI where the resource actually resides in whichever run-time environment you deploy your cartridge. XML Catalogs provide a redirection from a URI to another URI. By redirecting the resource URI locators, XML Catalogs serve to insulate your cartridge solution from environment configuration.

At run time, when OSM processes a URI you specify as part of the OSM data model, OSM first attempts to resolve the URI against the XML Catalogs you specified. Based on the mapping defined in the XML Catalogs, OSM updates the URI to adapt to the environment by resolving the location of the URI in your data model with the new URI you mapped for it in the XML Catalogs.

OSM processes XML Catalogs in the order you specify them, as follows:

- Specified in your OSM cartridge projects

XML Catalogs specified in your OSM cartridge projects are packaged as part of the cartridges and deployed to the OSM server. The XML Catalog manages only the resource files in the resources folder of your cartridge project. When you deploy a cartridge with XML Catalog support enabled, the contents of the resources folder are loaded into a virtual file system. Those resources are available through URI redirection to any other deployed cartridges. XML Catalogs can be defined in any cartridge, and those defined in one cartridge can reference resources in other cartridges. All of the XML Catalogs deployed on the OSM server are stored in memory and rebuilt each time the metadata refreshes. If there are conflicting XML Catalog entries, the latest entry loaded overwrites the earlier entry. See "[Defining rewriteURI Entries in XML Catalogs](#)" for information on how to avoid conflicting entries.
- Specified on the OSM server

XML Catalogs specified on the OSM server are defined in the **oms-config.xml** file and are loaded ahead of the XML Catalogs specified in OSM cartridge projects. XML Catalogs defined on the server are global in scope, applying to all cartridges. XML Catalogs specified on the OSM server override the URI mapping of XML Catalogs in cartridge

projects. URIs mapped in **oms-config.xml** are resolved for each specific environment. For example, a cartridge developer can specify an XML Catalog in **oms-config.xml** to point certain URIs defined in the data model to her own local Design Studio workspace, allowing her to change the contents of the resources locally and test the changes without having to redeploy the entire cartridge. Because OSM uses XML Catalogs that are specified on the OSM server to resolve URIs to be environment specific, XML Catalogs specified in OSM cartridge projects should not reference URI locations that are environment specific (such as drive letters).

Following are some examples of data OSM looks up from resource files at run time that you could use the XML catalog to redefine:

- Automation logic: You can configure XQuery and XSLT automators with the XML Catalog to specify the XQuery/XSLT file that drives the automation logic.
- Data from a data provider: A data instance provider can use the XML Catalog to specify a resource for providing the data loaded by the provider.
- Order item properties (for orchestration orders): Order item properties can be configured to be loaded through a URI locator. You can configure the XML Catalog to redirect the URI to specify the XQuery file that implements determining the property value.
- Decomposition rules (for orchestration orders): Decomposition rules can be configured to be loaded through a URI locator. You can configure the XML Catalog to redirect the URI to specify the XQuery file that implements determining the decomposition condition.

See "[Specifying XML Catalogs for OSM](#)" for instructions on how to specify XML Catalogs.

You can use the XML Catalog as a tool to perform cartridge versioning, to shorten development cycles, to allow for cartridge extensibility, and to insulate test and production environments from development-specific environments. See "[Examples of Using XML Catalogs](#)" for examples of these uses of the XML Catalog.

You can specify a common resources cartridge project that contains all of the shared resources across multiple cartridge projects. Defining the XML Catalog in this common resources cartridge consolidates the XML Catalog entries in one file which makes it easy to identify and eliminate conflicting catalog entries. See "[Resource Packaging Considerations for Using XML Catalogs](#)" for information on how you can package your resources when using XML Catalogs.

You can use any valid XML Catalog entry in your XML Catalog, but the `rewriteURI` entry is the most useful for OSM. See "[Defining rewriteURI Entries in XML Catalogs](#)" for information on defining `rewriteURI` entries for OSM.

Resource Packaging Considerations for Using XML Catalogs

You can specify a common resources cartridge project that contains all of the shared resources across multiple cartridge projects. Defining the XML Catalog in this common resources cartridge consolidates the XML Catalog entries in one file which makes it easy to identify and eliminate conflicting catalog entries. When you specify a common resources cartridge project in this way, other projects with model entities that reference the shared resources do not need to have an XML Catalog defined.

When you define resource properties in Design Studio, you can indicate to retrieve the resource by expression, file, or URI. XML Catalogs apply only to the URI option.

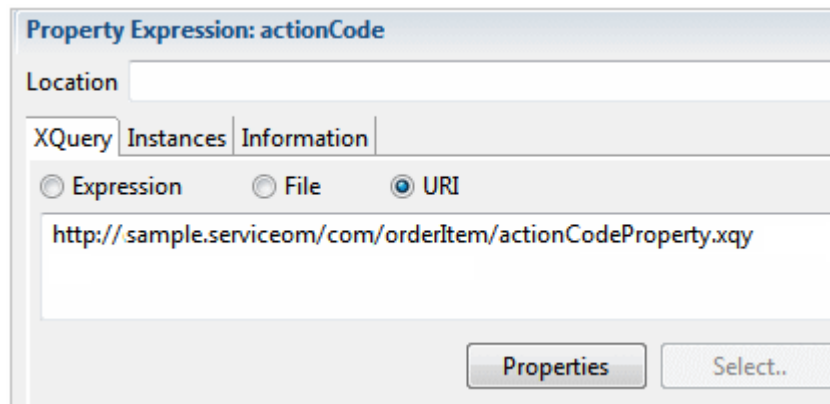
Consider the following when making your decision on which option to choose:

- Select **Expression** when the XQuery expression is short (only a few lines in length) and is not shared by other resources.

- Select **File**, also referred to as **Bundle In**, when the XQuery configuration is longer (more than a few lines in length) and is not shared by other resources. Use this method for resources that are not expected to change. You will not be able to access the resource except in the physical location specified. In addition, a resource referenced through **File** or **Bundle In** must exist in the same project as the entity referencing it.
- Select **URI** when the XQuery configuration is shared by multiple configurations and is located in a remote URI location to be accessed through the specified URI. If the XQuery configuration requires frequent changes, even though it is only used in one cartridge, you may want to use the URI option and also package the XQuery in a separate cartridge. That way, you can modify and redeploy the resource without having to compile and redeploy the possibly larger cartridge that uses it.

[Figure 19-11](#) shows the **Expression**, **File**, and **URI** options in the **XQuery** tab of a Design Studio editor:

Figure 19-11 URI Option for Defining Resource Properties



Oracle recommends you package resources in the following ways:

- Package resources to be used by a single cartridge in the cartridge itself. Select **File** or **Bundle in** when you define the resource properties in Design Studio.
- Package resources to be used by multiple cartridges into a shared or common resources cartridge and do the following:
 - Configure the resources to be retrieved by a URI. Select **URI** when you define the resource properties in the **XQuery** and **XSLT** tabs of the Design Studio editor.
 - Configure OSM to access the resources inside of the deployed common resources cartridge through a URI locator.

Another reason to package resources in a common resource cartridge is when you need to change those resources frequently and they are used by a large cartridge that has automation and model entities that take a long time to build, package, and deploy. By packaging resources that change frequently in a common resources cartridge, you avoid having to rebuild the larger cartridge each time you change the resources.

Defining rewriteURI Entries in XML Catalogs

This section describes how to define a rewriteURI entry in the XML Catalog for OSM. See ["Using XML Catalogs in OSM"](#) for general information about XML Catalogs and how they work with OSM.

You can use any valid XML Catalog entry in your XML Catalog, but the `rewriteURI` entry is the most commonly used entry for OSM. OSM uses the `rewriteURI` entry to replace the starting string of a URI (such as a URL) with an alternative string. For example, OSM could replace **`http://somewhere.org/something`** at run time with **`http://myhost/something`**.

During data modeling, you can define a URI locator (such as a URL) to access a resource as part of the OSM data model by using the **XQuery** and **XSLT** tabs of various Design Studio editors. For example, in the Order Recognition Rule editor you specify a URI to denote that the XQuery configuration for the recognition rule is hosted in a remote URI location such as **`http://osm_server/AIARecognitionRule.xqy`**. You can use the XML Catalog for any of the URIs you specify in the Design Studio editors. OSM uses the `rewriteURI` entry of the XML Catalog to update URIs you defined in your data model to adapt to different environments.

OSM replaces the starting string of a URI/URL with an alternative string as specified by the `rewriteURI` entry in the XML Catalog. For example, for this `rewriteURI` entry:

```
<rewriteURI uriStartString="http://example.org/somewhere" rewritePrefix="http://192.0.2.0/foo"/>
```

when OSM processes a URI that starts with **`http://example.org/somewhere`**, it replaces that starting string with **`http://192.0.2.0/foo`**. A URI you define in Design Studio as **`http://example.org/somewhere/myfolder/myfile.txt`** resolves as **`http://192.0.2.0/foo/myfolder/myfile.txt`** at run time.

Note

The `uriStartString` and the `rewritePrefix` attributes can be any valid URI: they do not have to be an IP address or host name.

`uriStartString` is set to the start of the resource URI you defined in Design Studio and `rewritePrefix` is set to the string OSM replaces `uriStartString` with after you deploy the cartridge.

To reference resources packaged inside of an OSM cartridge, you can use the OSM model scheme ("osmmodel") rather than the traditional URI schemes (HTTP, FTP, and so on) to define the URI. For example, for this `rewriteURI` entry:

```
<rewriteURI uriStartString="http://example.org/somewhere" rewritePrefix="osmmodel:///MyCartridge/1.0.0/resources"/>
```

when OSM processes a URI that starts with **`http://example.org/somewhere`**, it replaces that starting string with **`osmmodel:///MyCartridge/1.0.0/resources`**. A URI you defined in Design Studio as **`http://example.org/somewhere/myfolder/myfile.txt`** is resolved as **`osmmodel:///MyCartridge/1.0.0/resources/myfolder/myfile.txt`**.

This allows you to leverage the contents of the resources directory in each OSM cartridge at run time.

The format of an OSM model schema URI is:

`osmmodel:///CartridgeName/CartridgeVersion/resources`

where:

- **`osmmodel`** indicates a location inside of a deployed OSM cartridge
- *CartridgeName* is the name of your cartridge
- *CartridgeVersion* is the version of the cartridge (specified in the cartridge editor)

The default cartridge version uses the value **default**.

Note

See "[Using XML Catalogs to Support Cartridge Versioning](#)" for more information on cartridge versioning.

To enable cartridges to refer to resources contained in other cartridges in a non-version specific way, you refer to the default cartridge version. To refer to the default cartridge version, use the OSM model schema URI:

osmmodel:///cartridge_name/default/resources

See "[Using XML Catalogs to Support Cartridge Versioning](#)" for information on how the XML Catalog supports cartridge versioning.

Caution

To guarantee the correct resource is located, ensure that resources are always uniquely identifiable to a single catalog entry.

When defining XML Catalog entries, do not define mappings that can be satisfied by more than one entry. The following example shows two `rewriteURI` entries that can be used by OSM at run time to resolve the same URI locator in two different ways:

```
<rewriteURI uriStartString="http://
oracle.communications.ordermanagement.sample.centralom.resources/com"
rewritePrefix="osmmodel:///CommonResourcesCartridge/1.0.0/resources/com"/>
<rewriteURI uriStartString="http://
oracle.communications.ordermanagement.sample.centralom.resources"
rewritePrefix="osmmodel:///CommonResourcesCartridge/1.0.0/resources/comMapping"/>
```

Using the preceding `rewriteURI` entries, OSM can resolve the URI locator **http://oracle.communications.ordermanagement.sample.centralom.resources/com/foo.xml** as:

osmmodel:///CommonResourcesCartridge/1.0.0/resources/com/foo.xml

or

osmmodel:///CommonResourcesCartridge/1.0.0/resources/comMapping /com/foo.xml.

Specifying XML Catalogs for OSM

You specify XML Catalogs for an OSM cartridge project in the `cartridgeProject\xmlCatalogs\core\` directory (where `cartridgeProject` is the root of the project directory). In this directory, you create your XML Catalog file (you can use any filename such as **core.xml** or **catalog.xml**) and define your catalog entries within it. Design Studio automatically generates a template XML Catalog file `cartridgeProject\xmlCatalogs\core\xmlCatalogCoreTemplate.xml`.

You specify XML Catalogs on the OSM server in the OSM configuration entry **oracle.communications.ordermanagement.util.net.CatalogUriResolver.DefaultXmlCatalogsUris**. By specifying XML Catalog files on the OSM server, you can operationally modify how OSM resolves URIs without changing the contents of a cartridge. See "[Using XML Catalogs in](#)

[OSM](#)" for information on how OSM resolves URIs based on the XML Catalogs you specify on the OSM server.

To specify XML Catalogs on the OSM server:

1. Add or modify a configuration entry for **oracle.communications.ordermanagement.util.net.CatalogUriResolver.DefaultXmlCatalogsUris** in the **oms-config.xml** file.

See the chapter on configuring OSM with **oms-config.xml** in *OSM System Administrator's Guide* for detailed instructions on accessing and modifying the **oms-config.xml** file.

2. Enter the XML Catalog entries you require.
 - Multiple XML Catalogs can be specified separated by a semicolon (;).
 - Use any standard XML Catalog entry. The rewriteURI entry is the most commonly used for OSM. See "[Defining rewriteURI Entries in XML Catalogs](#)" for information on defining rewriteURI entries.

Note

The XML Catalog entries you specify are applied system wide. Ensure that resources are uniquely identifiable to a single catalog entry so that the correct resource can be located.

Note

This configuration defines the XML Catalog entries inline in the **oms-config.xml** configuration file.

3. Save the file.

Enabling and Disabling XML Catalog Support

XML Catalog support is enabled by default for all cartridges and is required to be enabled.

If your target run-time software version is OSM 7.0.2 or earlier, you can disable XML Catalog support for a cartridge (or re-enable it) by using the cartridge model variable XML_CATALOG_SUPPORT. For information on disabling or re-enabling XML Catalog support for a cartridge, see "Enabling and Disabling XML Catalogs for a Cartridge Project" in *Modeling OSM Processes*.

Examples of Using XML Catalogs

This section provides the following examples of how you can use the XML Catalog:

- [Using XML Catalogs to Support Cartridge Versioning](#)
- [Using XML Catalogs to Load Resources from a Development File System \(Traditional OSM Only\)](#)
- [Using XML Catalogs to Insulate Run-Time Environments from Development](#)

Using XML Catalogs to Support Cartridge Versioning

Cartridge versioning requires multiple versions of cartridges to reference their own versioned set of resources. For example, if you have version 1.0 and version 2.0 of an OSM cartridge deployed, you might have version-specific XQuery or JAR files that need to be used depending on which cartridge you are using. XML Catalogs ensure that the cartridges reference the correct resources.

To use XML Catalogs to support cartridge versioning:

1. In Design Studio, on the **Model Variables** tab of the cartridge editor, set the `CARTRIDGE_VERSION` model variable to the version number of the cartridge.
For more information about model variables, see "Working with Model Variables" in *Modeling Basics*.
2. In the parts of your model where you need OSM to substitute the version number, use `%{CARTRIDGE_VERSION}`. For example:

```
http://example.com/%{CARTRIDGE_VERSION}/xquery/myFile.xqy
```

3. In the XML Catalog, define the `rewriteURI` entries as follows:
 - If the cartridge is a component of a composite cartridge, use the `CARTRIDGE_VERSION` model variable. For example:

```
<rewriteURI uriStartString="http://example.com/%{CARTRIDGE_VERSION}"
rewritePrefix="osmmodel:///MyCartridge-Resources/%{CARTRIDGE_VERSION}/
resources"/>
```

- If the cartridge is not a component of a composite cartridge, use the specific cartridge version number. Do not use a model variable. For example:

```
<rewriteURI uriStartString="http://example.com/1.0" rewritePrefix="osmmodel:///
MyCartridge-Resources/1.0/resources"/>
```

When you deploy the cartridge, OSM replaces all instances of `%{CARTRIDGE_VERSION}` with the value that you set on the cartridge editor **Model Variables** tab.

4. When you create a new version of a cartridge, update the `CARTRIDGE_VERSION` model variable with the new version number.
5. In the XML Catalog, update the `rewriteURI` entries as follows:
 - If the cartridge is a component of a composite cartridge, no further updates are required. Because you used the model variable in the `rewriteURI` entries, OSM automatically replaces the model variable with the new version number when you deploy the cartridge.
 - If the cartridge is not a component of a composite cartridge, update the cartridge version number in each `rewriteURI`. For example:

```
<rewriteURI uriStartString="http://example.com/1.5" rewritePrefix="osmmodel:///
MyCartridge-Resources/1.5/resources"/>
```

Using XML Catalogs to Load Resources from a Development File System (Traditional OSM Only)

To shorten development cycle times that involve numerous coding, building, deployment, and test cycles, you can use the XML Catalog to load resources from a development file system. By using the XML Catalog in this way, you can test changes to resources located within the cartridge without needing to rebuild and repackage the cartridge. Rebuilding and repackaging

can be slow and CPU intensive because Design Studio needs to rebuild the deployment EAR file before any changes can be tested. By redirecting the URIs to a local resource, you can change XQuery, XSLT, XML, or Java code and immediately test the changes without having to rebuild, repackage, and redeploy (Java code would still need to be rebuilt but not repackaged and redeployed).

For example, use the XML Catalog to instruct OSM to load resources:

- From the development file system during development
- From the cartridge PAR file after testing

Locate resources on the file system instead of from within the cartridge PAR file so that configuration changes made to a resource are picked up by the run-time environment without having to rebuild and redeploy the cartridge. After testing is complete, the URI is redirected to load resources from the cartridge PAR file.

To redirect the URI so that OSM loads resources from the development file system:

1. In the Package Explorer view in Design Studio, navigate to the *cartridgeProject/xmlCatalogs/core/* directory.
2. Create or edit the **catalog.xml** file. You can create the file by renaming a copy of **xmlCatalogCoreTemplate.xml**.
3. Create the XML Catalog entry:

```
<rewriteURI uriStartString="http://example.org/somewhere" rewritePrefix="file://localhost/dev/env1/mycartridge/resources"/>
```

OSM loads all resources that start with **http://example.org/somewhere** from the file system located on localhost at **/dev/env1/mycartridge/resources**.

To redirect the URI so that OSM loads resources from the cartridge PAR file after testing is complete, change the preceding configuration to:

```
<rewriteURI uriStartString="http://example.org/somewhere" rewritePrefix="osmmodel:///MyCartridge/1.0.0/resources"/>
```

OSM loads all resources that start with **http://example.org/somewhere** from the cartridge PAR file.

The XML Catalog supports resource extensibility in a cartridge solution because URIs can be easily rewritten to change the location from which resources are loaded. The XML Catalog allows you to redirect the cartridge solution to use customized resources different from the ones that were originally provided by the cartridge solution.

Using XML Catalogs to Insulate Run-Time Environments from Development

To insulate test and production environments from development-specific environments, you can use the XML Catalog. When you develop your code, you can set your XML Catalog to point to local resources on your file system on your laptop (not applicable for OSM cloud native). Assume you have an automated test environment that runs daily tests on certain cartridges that use resources on the testing box. In production, you would use the XML Catalog to point resources to your production systems. Note that in this example the resources are not bundled inside of the cartridges.

Cartridge Deployment

Design Studio allows you to deploy cartridges to an OSM environment. For more information about creating environment entities and deploying cartridges, see "Deploying Cartridge Projects" in *Modeling Basics*.

Cleaning and Rebuilding Cartridges Prior to Deployment

Cleaning and rebuilding a cartridge is not included as a deployment step because it is not required for a successful deployment. However, Oracle recommends that you periodically clean and rebuild a cartridge prior to deployment because multiple people can work in the same cartridge; cleaning and rebuilding the cartridge picks up these changes, ensuring that the cartridge is in its current state.

Optimizing Cartridge Deployment

During the development process, you can save time by redeploying your changes only, rather than redeploying the entire application. For more information about this option, see "Managing Changes to Deployed Cartridges" in *Modeling OSM Processes*.

Deploying Multiple Cartridges

You can simultaneously deploy multiple cartridges when deploying from the Environment perspective Cartridge Management view. When you select multiple cartridges for deployment, the system deploys the cartridges individually based on any existing cartridge dependencies. The system prevents you from deploying cartridges independently of those cartridges upon which they depend. For more information, see "[Deploying Cartridges with Dependencies](#)."

Deploying Cartridges with Dependencies

A cartridge can be dependent upon information defined in another cartridge. When dependencies exist between cartridges, the build of the cartridge with the dependency extracts the dependent information from the built cartridge upon which it depends and copies the information to the cartridge being built. As a result, the cartridges can be deployed independently from each other.

For example, CartridgeA is created and defines *phoneNumber* as a data element in a data schema. CartridgeB is then created, and *phoneNumber* is added to a CartridgeB order template. This causes CartridgeB to be dependent upon CartridgeA. CartridgeA is built first. When CartridgeB is built, the *phoneNumber* data element is extracted from CartridgeA and copied to cartridgeB. As a result, cartridgeB can be deployed even if CartridgeA is not deployed.

 Caution

Cartridges should not be circularly dependent upon each other (CartridgeA depends on CartridgeB *and* CartridgeB is depends on CartridgeA). If you define cartridges with a circular dependency, the cartridge build will fail, with an error like, "CartridgeA Cartridge Model Dependency Error – Cyclic dependency exists: CartridgeA <- CartridgeB." If there is a composite cartridge that refers to cartridgeX or cartridgeY, the composite cartridge build will also fail, as a result of the component cartridge builds failing.

Deploying Cartridges to the OSM Database Using XMLIE

You can deploy cartridges to the OSM DB directly using XMLIE. This approach is strongly recommended for controlled environments such as production, pre-production or UAT, and for environments managed using a CI/CD pipeline. It is also the preferred approach for semi-formal environments, such as test systems.

Offline Cartridge Deployment

You can deploy cartridges to the OSM DB directly using XMLIE while OSM is shut down.

To deploy a cartridge in offline mode using XMLIE:

1. Ensure that all managed servers are stopped. Shutting down the admin server is optional. See *OSM System Administrator's Guide* for details on stopping managed servers.
2. Build the cartridge PAR file using Design Studio.
3. Do the following to deploy the cartridge PAR file with XMLIE:
 - a. Change the directory to the directory where XMLIE is installed. For example, **cd /opt/osm_sdk/SDK/XMLImportExport**.
 - b. Copy the **config_sample.xml** file located at **\$XMLImportExport/config** to the **config.xml** file and edit it to specify the OSM DB schema connection information.
 - c. Run the **EncryptPasswords.sh** script to encrypt the password of the OSM DB schema.

```
./EncryptPasswords.sh config.xml -dbUser
```

- d. Run the **import.sh** XMLIE script to deploy the cartridge PAR file:

```
./import.sh $cartridge.par config.xml
```

4. Deploy all the cartridges using the same approach and then start the servers.

 Note

For a solution cartridge, the solution PAR file already contains all the PAR files of the components of the cartridges. Deploy only the solution PAR file. Do not deploy PAR files of the individual components.

Online Cartridge Deployment

You can deploy cartridges to your OSM running instance while orders from a cartridge that you deployed earlier are still being processed. When you deploy cartridges in online mode, OSM

availability is uninterrupted and ongoing orders continue to react to new incoming messages. This is achieved by shutting down and restarting the OSM managed servers sequentially. OSM leverages WebLogic Zero Downtime Patching to deploy cartridges without any loss of service.

Online cartridge deployment is built on top of WebLogic's Zero Downtime framework. This framework imposes certain pre-requisites:

- WebLogic Node Manager must be configured and running on all the hosts where OSM servers exist (configured during domain creation).
- "Machines" must be configured in the WebLogic domain, and hosts must be assigned to these machines during domain creation.
- The Proxy Server (if used) and Managed Servers must be started with Node Manager, not via other mechanisms such as scripts. The Admin Server should also be started via Node Manager.
- Admin Server cannot run on the same host as any Managed Server.

For complete control, ensure that each machine runs exactly one managed server. While it is possible to assign two or more managed servers to the same machine, it affects the overall availability of OSM as all managed servers on a given machine undergo maintenance at the same time.

The OSM cluster must have at least two functioning managed servers for online cartridge deployment to work.

The following cartridge deployment operations are supported in an online deployment mode:

- Deployment of the first version of a new cartridge
- Deployment of a new (updated) version of an existing cartridge

Note

To redeploy an existing version of a cartridge, use the regular deployment mechanism via Design Studio. You can also use the Cartridge Management Tool (CMT) to redeploy an existing version of a cartridge.

During the deployment process, the cluster is reduced by one managed server (the one that is actively restarting), while at least one other managed server (the one that just finished restart) is still warming up to full capacity. To avoid transient resourcing issues, it is strongly recommended to perform online cartridge deployment during a low order volume period. The processing of in-progress orders with undelivered incoming messages is paused while the operating managed server undergoes a restart. The processing of the in-progress orders resumes automatically once the restart of that managed server is complete. The processing of in-progress orders without undelivered incoming messages is not interrupted.

To deploy cartridges while OSM is still running:

1. Deploy each cartridge PAR file that needs to be added to OSM. Also, fast-undeploy each cartridge version that needs to be removed.

To deploy a cartridge PAR file:

- a. Change the directory to the directory where XMLIE is installed. For example, `cd /opt/osm_sdk/SDK/XMLImportExport`.
- b. Copy the **config_sample.xml** file located at **\$XMLImportExport/config** to the **config.xml** file and edit it to specify the OSM DB schema connection information.

- c. Run the **EncryptPasswords.sh** script to encrypt the password of the OSM DB schema.

```
./EncryptPasswords.sh config.xml -dbUser
```

- d. Run the **import.sh** XMLIE script to deploy the cartridge PAR file in online mode:

```
./import.sh $cartridge.par config.xml online
```

Note

For a solution cartridge, the solution PAR file already contains all the PAR files of the components of the cartridges. Deploy only the solution PAR file. Do not deploy PAR files of the individual components.

2. Perform rolling restart of the managed servers using WebLogic Remote Console or WLST.
 - a. In WebLogic Remote Console, from **Monitoring Tree**, go to **Environment**. Within **Environment**, select **Servers**.
 - b. Here, manually shut down and restart each managed server, one at a time.

Note

Make sure each server returns to **Running** before proceeding to shut down and restart the next server.

About Performing a Rolling Restart of Managed Servers

After running the XMLIE script to deploy the cartridge, you must restart all the managed servers in the OSM cluster in a sequence. The completion of the restart of one managed server should trigger the restart commencement of the next. The WebLogic Zero Downtime Rolling Restart capability (part of the Zero Downtime Patching functionality) provides this mechanism for online cartridge deployment. This rolling restart can be triggered and monitored either using the WebLogic Remote Console or using WLST.

For details on how to configure the rolling restart workflow, see *Oracle Fusion Middleware Administering Zero Downtime Patching Workflows* available at: <https://docs.oracle.com/en/middleware/fusion-middleware/weblogic-server/14.1.2/wlzdtt/intro.html>.

While configuring rolling restart in WebLogic Remote Console, make sure each server returns to **Running** before proceeding to shut down and restart the next server.

While configuring the restart in WebLogic Remote Console, you need to specify the value for **Graceful Shutdown** to **120**. To do this, do the following:

1. Navigate from **Edit Tree** to **Environment**. Here, navigate to **Servers**, and open **Server**.
2. Here, open the **Advanced** tab, and within it, open the **Start/Stop** subtab.
3. In the **Start/Stop** subtab, specify **120** as the value for **Graceful Shutdown Timeout**.

This is the grace period for orderly shutdown. Tune this timeout value to ensure that your managed servers can shutdown gracefully. In general, the more activity on the server, the longer it takes to shutdown. Monitor how long it takes for your managed server to shutdown normally and use that as the basis for this setting. For instructions about performing rolling

restart, see *Oracle Fusion Middleware Remote Console Online Help for Oracle WebLogic Server 14.1.2.0*.

Building and Deploying Composite Cartridges

When you build and package a composite cartridge, it is packaged as a single PAR file which contains:

- All non-orchestration entities aggregated and packaged into the composite cartridge
- A PAR file for each component cartridge referenced in the composite cartridge

When a composite cartridge is deployed, it includes all of the OSM non-orchestration entities and all component cartridges referenced in the composite cartridge, if they are either changed or not currently deployed.

Setting Cartridge Dependencies

Projects have dependencies on other projects when entities in one project reference entities in a different project. If you configure a cartridge to reference content in other cartridges without declaring project dependencies, Design Studio creates a warning. For information about how to set cartridge dependencies, see "Managing Project Dependencies" in *Modeling Basics*.

Post-Deployment Effect on Numeric Data

When defining a data element in Design Studio, you have the option of defining numeric data as type int, double, float, or decimal. OSM does not directly support these data types. Rather, the OSM Data Dictionary defines the data type *numeric*. When a cartridge containing the data types int, double, float, or decimal is deployed to the OSM server, the data types are converted to the OSM Data Dictionary type *numeric*.

Post-Deployment Changes to Cartridge

You can make changes to a cartridge after the cartridge has been deployed to the OSM server by making changes to the original cartridge in Design Studio and then redeploying the cartridge. Before doing this, you should back up the original cartridge, because exporting a deployed Design Studio cartridge back out of OSM into Design Studio is not supported.

Metadata Errors

Metadata errors can cause order processing failures and can occur in any cartridge with orchestration model entities. Metadata is the information used to represent OSM modeled entities such as order templates, order components, order items, tasks, decomposition rules and so on. If there are no metadata errors, the cartridge models deployed are valid.

Metadata errors occur when OSM references an entity that is missing or the modeling for an entity is incorrect (for example, a data type for an entity is incorrectly entered).

OSM detects and logs metadata errors during the following procedures:

- Deploying a cartridge to a server
- Restarting an OSM server
- Refreshing OSM metadata with the OSM Order Management web client or with an Ant refresh

These actions reload OSM metadata, and errors are detected while running validation constraints against certain orchestration model entities. [Table 19-2](#) lists the orchestration entities that are currently validated.

Table 19-2 Orchestration Entities That Are Currently Validated

Entity Type	Schema Constraint Description
OrchestrationStageType	Verifies that the value for the element dependsOnStage is a valid stage. dependsOnStage is empty if the stage is independent. A stage is valid if it is defined in the orchestrationSequence of orchestrationModel.
OrderComponentSpecRef	Verifies that this reference is pointing to a valid OrderComponentSpec. OrderComponentSpec is valid if it is defined in orchestrationModel.
OrchestrationConditionRef	Verifies that this reference is pointing to a valid orchestration condition. An orchestration is valid if it is defined in orderItemSpec of orchestrationModel.
DurationType	Verifies that a valid duration value is specified.
ProductSpecRef	Verifies that this reference is pointing to a valid ProductSpec. ProductSpec is valid if it is defined in orchestrationModel.
OrderItemSpecRef	Verifies that this reference is pointing to a valid OrderItemSpec. OrderItemSpec is valid if it is defined in orchestrationModel.

After rebuilding or deploying a cartridge, check for metadata errors. Search for the string *Metadata Errors* in the Console view of the Cartridge Management editor in Design Studio. If you are not using Design Studio to deploy cartridges, look in the Oracle WebLogic Server logs for the same string.

Metadata errors appear together in a numbered list. For example:

```
Metadata Modeling Errors*****
1) Metadata error Severity:ERROR Description:Invalid
ProductSpec[name=NonService.Offer,
namespace:CommunicationsSalesOrderFulfillmentPIP]
Cartridge Name:TypicalSalesOrderFulfillment Version:1.0.0
EntityName:NonService.Offer EntityType:ProductSpecRef
```

where

- **Severity** can be an ERROR, WARNING or CRITICAL.
- **Description** describes the failure and provides the entity type, name and name space.
- **Cartridge Name** is the name of the Cartridge that is reporting the problem.
- **Version** is the cartridge version.
- **EntityName** and **EntityType** are the name and type of the entity reporting the metadata error and its name space. In some cases, the modeled entity within the cartridge is invalid. In other cases, the modeled entity is referring to another entity which is missing or invalid.

If you find metadata errors, it most likely means that OSM is calling on an entity that is missing, has the wrong name, or has a value that is incompatible for the entity type.

To fix the problem, clean and rebuild your cartridges, and make sure all related cartridges are deployed. If you still have metadata errors, it may mean that you have errors in your data. In this case you will have to use Design Studio to re-validate your model. See "[Cleaning and Rebuilding Cartridges Prior to Deployment](#)."

Part V

Working with Capabilities Cartridges

Part V contains the following chapter about working with Capabilities Cartridges in Oracle Communications Order and Service Management:

- [Working with Capabilities Cartridges \(Cloud Native Only\)](#)

Working with Capabilities Cartridges (Cloud Native Only)

This chapter describes the guidelines, limitations and restrictions for generating Design Studio workspace content as a capabilities cartridge.

About Capabilities Cartridges

A capabilities cartridge is a packaging concept. It is not a new cartridge type but instead relates to the way a set of OSM cartridge content is packaged for delivery into the dynamic design process.

About Dynamic Cartridge Assembly (Cloud Native Only) in *OSM Concepts* provides an overview of the Dynamic Cartridge Assembly feature. You need to understand this feature to understand how a capabilities cartridge fits within OSM.

About OSM Participant provides an overview of the steps that you need to take in both Solution Designer, and OSM to use Capabilities Cartridges. You need to understand the way this exchange functions to use a capabilities cartridge.

You should have a good working knowledge about writing OSM orchestration cartridges as context for the information in this chapter.

About Capabilities Cartridges Restrictions

Not all OSM cartridges can support generation as a capabilities cartridge. The following are the restrictions that you need to be aware of when generating capabilities cartridges:

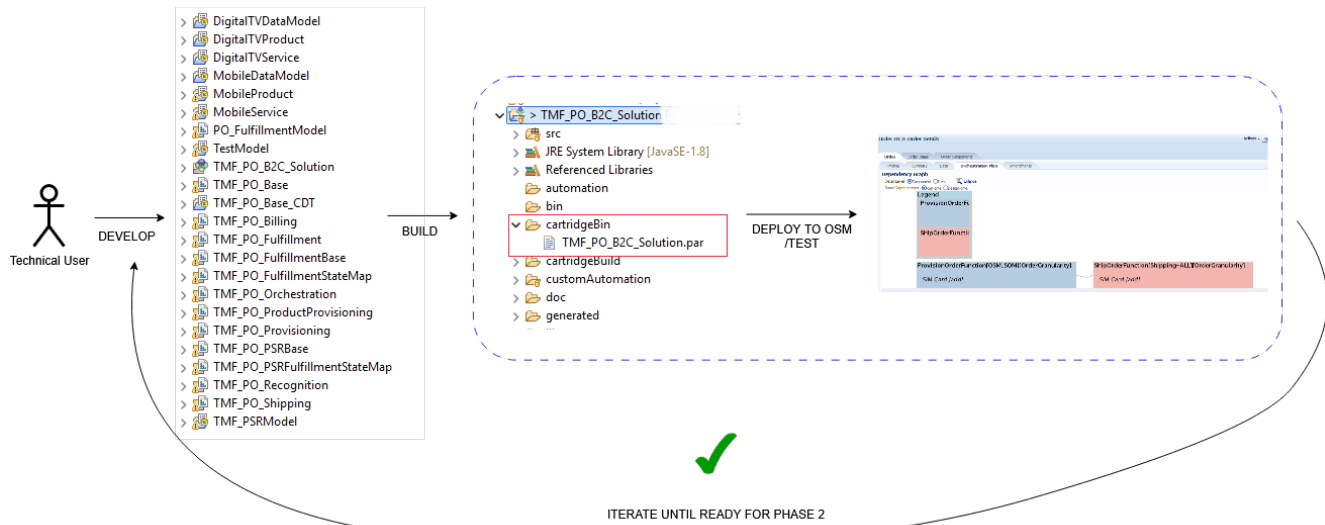
- You can use only TMF cartridges to generate a capabilities cartridge CPAR file.
- It must host a TMF 622 specification. TMF 641 is not yet supported for capabilities cartridge development.
- The OSM target server version in Design Studio must be 8.0 or later.
- Only a single capabilities cartridge can be delivered to dynamic design. Multiple versions of the capabilities cartridge (as identified by the Solution name) are allowed. For example, multiple COM solutions are not allowed, but multiple versions of a single COM solution is possible.

User Workflow in Design Studio

For background information about user journeys, refer to About Dynamic Cartridge Assembly (Cloud Native Only) in *OSM Concepts*.

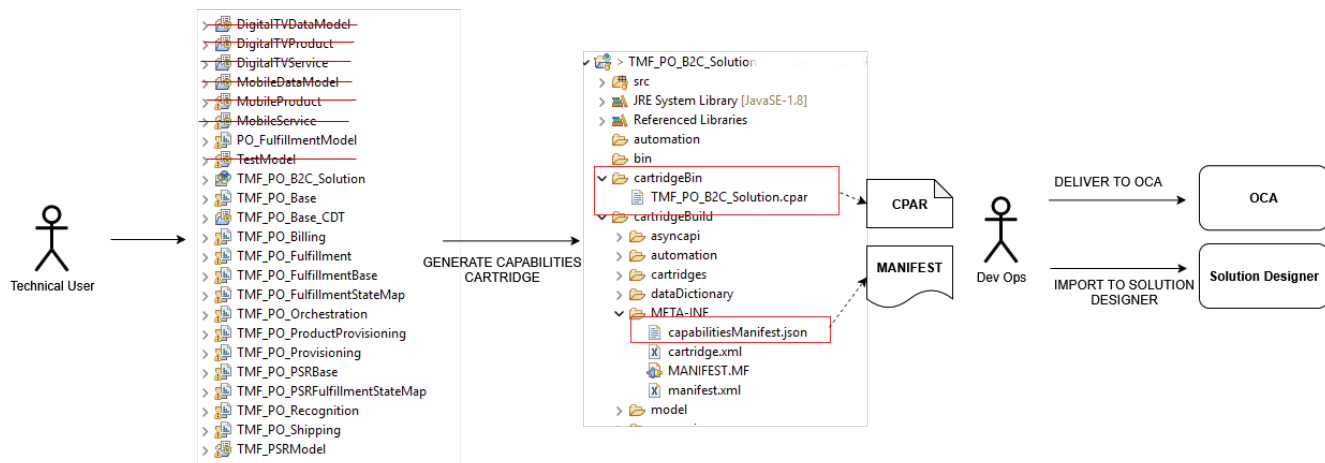
The following image shows the traditional design journey for developing and deploying cartridges using Design Studio and OSM.

Figure 20-1 Traditional Design Journey



In phase one, the workspace is re-packaged so that the necessary artifacts are generated and can then be passed to the OSM dev ops administrator for delivery to the OCA microservice and Solution Designer. The following image illustrates this phase.

Figure 20-2 Phase 1 of Dynamic Design Journey



Capabilities Cartridge Lifecycle

All lifecycle stages must be managed in a coordinated manner between the OSM Cartridge Assembler (OCA) microservice and Solution Designer. Any changes or updates that you make in one application must also be reflected through a corresponding update in the other application. This synchronization ensures that both systems remain aligned, reducing the risk of inconsistencies and operational errors.

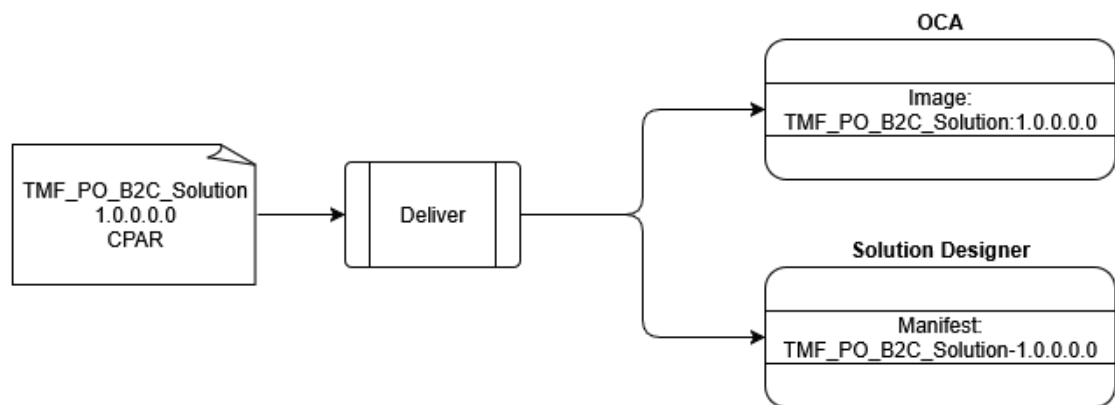
Creating and Delivering the CPAR

A capabilities cartridge is generated from a Design Studio workspace resulting in two deliverable artifacts. These artifacts are:

- A CPAR archive that is used to create an image that is loaded into the OSM Cartridge Assembler Microservice. For more information about this, refer to Container Images in the *OSM Cloud Native Deployment Guide*.
- A manifest file that is extracted from the archive to import into Solution Designer.

The following flowchart illustrates the deliverable artifacts generated from the sample CPAR.

Figure 20-3 CPAR Deliverable Artifacts



Updating and Re-Delivering the Capabilities Cartridge

Once a capabilities cartridge has been delivered, any changes you make to the content can impact:

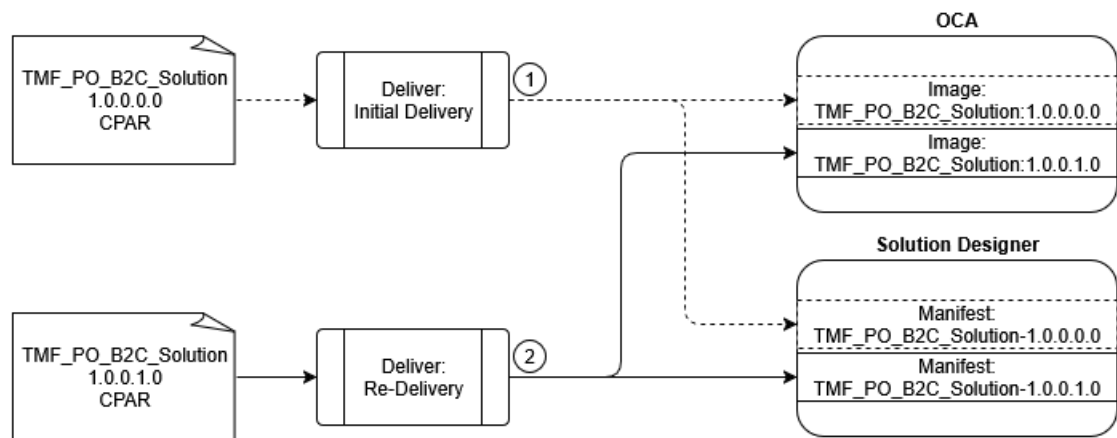
- Only OCA, such as an XQuery change
- Only SCD, such as the OSM entity descriptions
- Both applications, such as adding, removing or updating the OSM capabilities. For example, introducing new or changed fulfillment patterns, fulfillment functions or fulfillment systems

Regardless of its type, you need to exclusively re-deliver any change you make through Design Studio. You need to ensure that all Design Studio content is being maintained under source control to ensure deliverable artifacts can be reliably and repeatedly generated. This approach promotes consistency, traceability, and reproducibility in your deployment process.

To further enhance traceability, Oracle strongly recommends that when you are making changes to the capabilities cartridge produced by Design Studio, you also change the version.

The following flowchart illustrates the Update and Re-deliver process for the capabilities cartridge.

Figure 20-4 Updating and Re-Delivering the CPAR



Retiring a Capabilities Cartridge

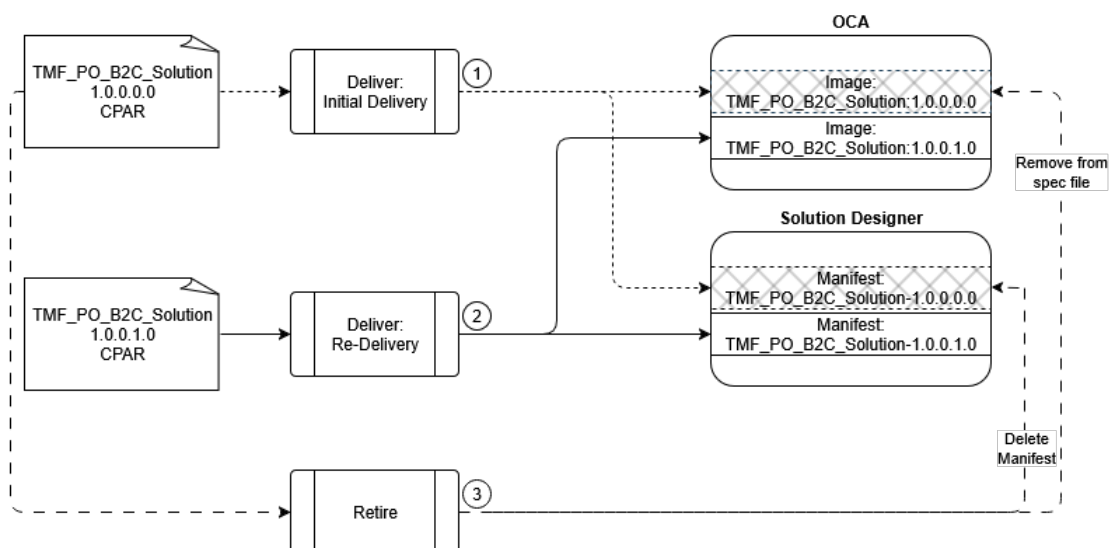
If a capabilities cartridge version should no longer be used, the entry for that image should be removed from all OSM workspaces connected to the Solution Designer instance. The manifest should also be deleted.

Note

Solution Designer does not support retirement of a manifest once it has been released to production.

The following flowchart illustrates the process for retiring a CPAR.

Figure 20-5 Retiring a CPAR



Capabilities Cartridge Content

This section describes some limitations and conventions when packaging content as a capabilities cartridge.

Design Studio Workspace

A Design Studio workspace can be organized in such a way that it can support both a traditional build (PAR) or capabilities cartridge generation (CPAR). While PAR files can be deployed to OSM and take effect immediately (allowing OSM to process orders), a CPAR file needs to be delivered differently. A CPAR file cannot process orders - it needs to be part of the dynamic cartridge process first.

In order to be compatible with the dynamic cartridge assembly process that takes place in the OCA microservice, the capabilities cartridge must adhere to the restrictions and conventions outlined in this chapter. OSM delivers a reference sample inside the OSM SDK that can be built as a capabilities cartridge. It aligns to the correct design studio workspace setup, and follows the guidelines outlined here.

Capabilities Cartridge Conventions

You need to adhere to the guidelines outlined in the following topics to ensure that your capabilities cartridge is compatible with the dynamic cartridge assembly process:

- [Fulfillment Model](#)
- [Order Recognition Rule](#)
- [XML Catalogs](#)
- [Automation Concurrency Map](#)
- [Relationship Types](#)
- [Provider Function](#)
- [Description Fields](#)
- [Fulfillment Pattern Property on the Order Item Specification](#)
- [Transformed Order Item Properties](#)
- [Order Component Organization](#)

Fulfillment Model

A single component cartridge must hold the fulfillment model. In the reference sample, this is PO_FulfillmentModel.

This cartridge should not contain any OSM entity definitions other than what is allowed as Test Content. For more information about what is allowed as test content, refer to [Conceptual Model Test Content](#).

Note

This does not apply to items in the resources (such as XQueries) directory.

Order Recognition Rule

The cartridge version changes every time you publish from Solution Designer. Therefore, you need to set the Target Order Version as **default** in the Order Recognition Rule.

XML Catalogs

Any XML catalog that references the resources held in the fulfillment model cartridge (**PO_FulfillmentModel**), needs to use **default** in the re-write string instead of a version number as this will be updated with each publish from Solution Designer.

```
<!-- Support ParameterBinding -->

<rewriteURI uriStartString="http://oracle.communications.orchestration.com/
tmf-api/productOrderingManagement/{TMF622_VERSION}/productOrder/
parameterBinding" rewritePrefix="osmmodel:///PO_FulfillmentModel/default/
resources/parameterBinding"/>
```

Automation Concurrency Map

This is a file that you can optionally define in the resources directory of the Solution cartridge. If you use this file, you must not include the target Plugin version. In this way, the use of Order Automation Concurrency Control will be scoped to the cartridge namespace and not one specific version.

```
<targetPlugins>
  <cartridgeNamespace>TMF_PO_B2C_Solution</cartridgeNamespace>
  <pluginSelector>.[cartridgeNamespace="TMF_PO_B2C_Solution"][implement/
script/url="http://oracle.communications.orchestration.com/tmf-api/
productOrderingManagement/{TMF622_VERSION}/productOrder/provisioning/
componentInteraction/4.1.0.1.0/serviceOrderStateChangeReceiver.xquery"]</
pluginSelector>
</targetPlugins>
```

Relationship Types

The **Relationship Type** entity is not configurable in Solution Designer. However you can use the associations **Primary** and **Auxiliary** in the product to service relationship.

These entities are still defined in the capabilities cartridge instead of being test content, because of the action configuration these entities carry. Refer to [Mapping Rules](#) for more details.

You need to follow these rules when configuring relationships:

- You must create a relationship called **Primary**.
- You must create a relationship called **Auxiliary**. Design Studio will enforce this when the capabilities cartridge is generated.
- You must not create any additional relationship types.

Provider Function

If you use OTM, the provider function must have the two relationship types defined - **Primary** and **Auxiliary**, with Primary set as the default.

Description Fields

While description fields on various objects in Design Studio are optional, it is strongly recommended that you populate these fields with care. The description is a means of communicating information from the technical user in Design Studio to the business modeler in Solution Designer.

The descriptions should capture meaningful details for the following entities in Design Studio:

- All order components (functions, systems, granularities)
- Fulfillment patterns

You can provide an overall description to summarize the entire cartridge as part of the Capabilities Cartridge wizard. All of these descriptions will be visible to the business modeler in Solution Designer in order to guide them on the nature of these entities and any other relevant aspects, such as functional assumptions made by the technical user.

Fulfillment Pattern Property on the Order Item Specification

Product definitions are pruned from the capabilities cartridge during generation. Due to this, Design Studio cannot populate the **productSpecMapping.xml** file with mapping details. Prior to OSM 7.5.0, this file was used to determine the correct fulfillment pattern for the inbound line item.

A simplified mapping technique was introduced in OSM 7.5.0 and the dynamic assembly feature relies on this pattern.

Note

See the "How to Simplify the Fulfillment Pattern Property on an OSM Order Item Specification (Doc ID 3000040.1)" KM article on My Oracle Support for more information on simplifying the fulfillment pattern property on an OSM Order Item Specification.

In the capabilities cartridge, you need to populate the fulfillment pattern property with the incoming product specification name instead of the fulfillment pattern name. This simplified mapping defined in the capabilities cartridge will be coupled with the compatible OSM metadata during dynamic assembly so that OSM can resolve the correct pattern to use.

Transformed Order Item Properties

You need to create the following properties on the transformed order item so that they can be populated by the cartridge assembly process. These values are needed by the automation plugin constructing the TMF 641 service order.

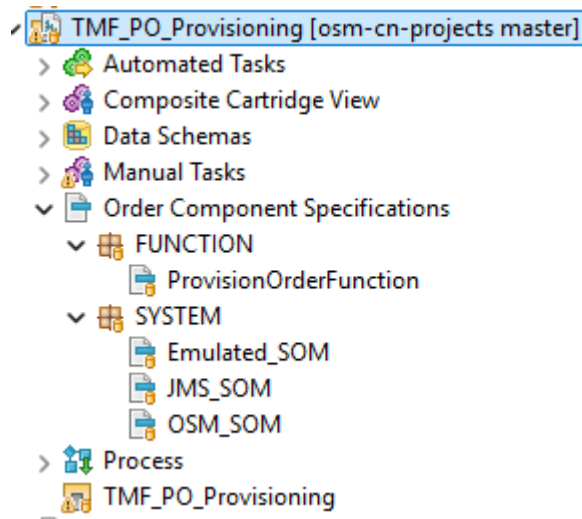
- **ServiceSpecificationId**
- **ServiceSpecificationName**

Order Component Organization

OSM uses cartridge boundaries to create the correct function to system hierarchy. This is used to construct a graphical fulfillment pattern display in Solution Designer. A functional cartridge must therefore contain order components representing the function and all applicable fulfillment systems.

The following image shows the order component organization for a sample cartridge.

Figure 20-6 Order Component Organization



Configurability Limitations

In the dynamic design journey, Solution Designer has ownership for some entity definitions but it does not provide the same level of OSM specific configurability which can be seen in Design Studio. This is necessary to provide a simpler and more streamlined experience for the modeler, but the cartridge developer needs to be aware of the impact this may have on the fulfillment logic in the capabilities cartridge.

General Restrictions

The following are the general restrictions:

- You must use a single cartridge to hold the fulfillment model.
- You must use three stage orchestration which includes decomposition to function, to system, and to granularity.
- You cannot use structured data elements on catalog entities (product or service entities).

Mapping Rules

The limitations in this area are:

- The mapping options for individual attribute mappings support **copy value**, **value map** and **unit of measure**. There is no equivalent for **advanced** mapping in Solution Designer.

- You cannot map order item properties (from source to target). Refer to [Transformed Order Item Properties](#) for a detailed explanation.
- The primary relationship is used to drive order item selection. This is set during dynamic assembly and cannot be changed in Design Studio or Solution Designer
- You cannot configure action codes directly. These come from the action codes defined in the capabilities cartridge and cannot be restricted in Design Studio or Solution Designer
- You cannot configure action mappings directly. These are driven from the relationship action map. This is set during dynamic assembly and cannot be changed, however the relationship type and its action map can be defined in Design Studio
- You cannot configure fulfillment state filters directly. These come from the order component that uses the provider function and cannot be restricted in Design Studio or Solution Designer
- All mappings support bi-directional mapping. This is set during dynamic assembly and cannot be changed in Design Studio or Solution Designer

Transformation Sequence

The transformation sequence editor allows you to define the order item context for the Primary Relationship stage. Design Studio enforces this configuration even though the context can be driven from similar configuration in the mapping rule.

You need to provide the configuration that Design Studio needs to generate a capabilities cartridge successfully and without any build errors. At the same time you need to ensure that the configuration you provide does not conflict with what is applied during dynamic cartridge assembly.

In the editor, select **Advanced** as the context type, and for the expression use **'na'**. The following image shows sample Transformation Sequence configuration.

Figure 20-7 Transformation Sequence

Transformation Sequence : PSRTransformationSequence

Description: Product Service Resource Transformation Sequence

Namespace: oracle.communications.tmf.productorder

- PSRTransformationSequence
 - ProcessPrimaryRelationshipsSt
 - ProcessChildrenRelationshipsS
 - ProcessSiblingRelationshipsSta
 - ProcessAncestorRelationshipsS

Stage Detail | Order Item Context | Related Order Item Select

Details

Reuse Previous Stage Context ☐

Context Type

☐ Simple

☒ Advanced

Property

Value

Expression

XQuery | Instances | Notes

☒ Expression ☐ File ☐ URI

Decomposition and Routing Rules

Routing rules in Solution Designer get translated into decomposition rules in OSM. The following limitations apply to routing rule configuration:

- There is no means to limit the applicability of a routing rule to a fulfillment pattern or a set of fulfillment patterns.
- The conditions on a routing rule apply equally to the rules controlling the second and third stage of decomposition. You cannot configure a condition that applies to the function stage to the system stage differently than it applies to the system stage to granularity stage.
- Cartridge assembly enforces that only a single routing rule can reference a fulfillment function. You cannot have two rules that both target the same function.

Cartridge Versioning

The version defined on most component cartridges in Design Studio is retained through the dynamic assembly process. The fulfillment model and solution cartridges are exceptions to this rule. The **Customer Patch** value of those cartridges will be replaced during dynamic assembly.

Note

Any changes you make to the customer patch value for the fulfillment model and solution cartridges in the Design Studio workspace will not be preserved.

The following image shows the **Customer Patch** value.

Figure 20-8 Customer Patch Value in Cartridge Versioning

The screenshot shows the 'Cartridge Properties' dialog box. The 'Customer Patch' field is highlighted with a red rectangle. The value '0' is entered in this field. Other fields include 'Provider' (OSM Cartridge Provider), 'Identifier' (PO_FulfillmentModel), 'Package Name' (empty), 'Major Version Number' (1), 'Minor Version Number' (0), 'Maintenance Pack' (0), 'Generic Patch' (0), 'Build Number' (111), and 'Target Version' (8.0.0).

Cartridge Properties	
Provider	OSM Cartridge Provider
Identifier	PO_FulfillmentModel
Package Name	
Major Version Number	1
Minor Version Number	0
Maintenance Pack	0
Generic Patch	0
Customer Patch	0
Build Number	111
Target Version	8.0.0

DataTypes for Product and Service Attributes

Products and Services are pruned from the test content when generating the capabilities cartridge. Dynamic assembly supports only the following simple data types as characteristics on Products and Services:

- string
- boolean
- int
- long

Solution Designer supports more data types (including structured data) but these are not permitted by dynamic cartridge assembly. You need to ensure that the cartridge logic relies only on the supported data types so that your orders process successfully.

Transformed Order Item Properties

You can define a set of general properties in the **Order Item Specification** editor in Design Studio. These values are populated based on the mappings on the Product (PS) - Service (CFS) mapping and not by any configuration applied on the **Order Item Specification** editor.

Since Solution Designer does not expose OSM specific configurability for PS-CFS mappings, you cannot map the order item properties.

For an automation plugin to correctly generate a TMF 641 order, it needs access to data about the CFS that is not defined as a service characteristic.

To bridge this gap, dynamic assembly provides the following property mappings for primary PS-CFS relationships:

- **ServiceSpecificationName**: This should be created as an order item property. It will then be populated automatically from the CFS ID defined in Solution Designer.
- **ServiceSpecificationId** : This should be created as an order item property. It will then be populated automatically from the CFS Name defined in Solution Designer.
- The property that is declared as the **Order Item ID Property** will be populated with a random UUID.
- The property that is declared as the **Fulfillment Pattern Mapping Property** will be copied from the source fulfillment pattern and needs to be populated for successful cartridge deployment.

Capabilities Cartridge Test Scaffolding

Test data is created in Design Studio and is required during the traditional design journey, outlined in [User Workflow in Design Studio](#) when the re-usable OSM capabilities are being validated through the traditional build and deploy cycle.

Fulfillment Model

Conceptually, a fulfillment model includes the following items:

- PSR (Product, Service, Resource) Entities
 - Products
 - Services
 - Data Elements
 - Unit of Measure Converters
- OSM Entities
 - Mapping Rules
 - Decomposition Rules

OSM Enrichment Data

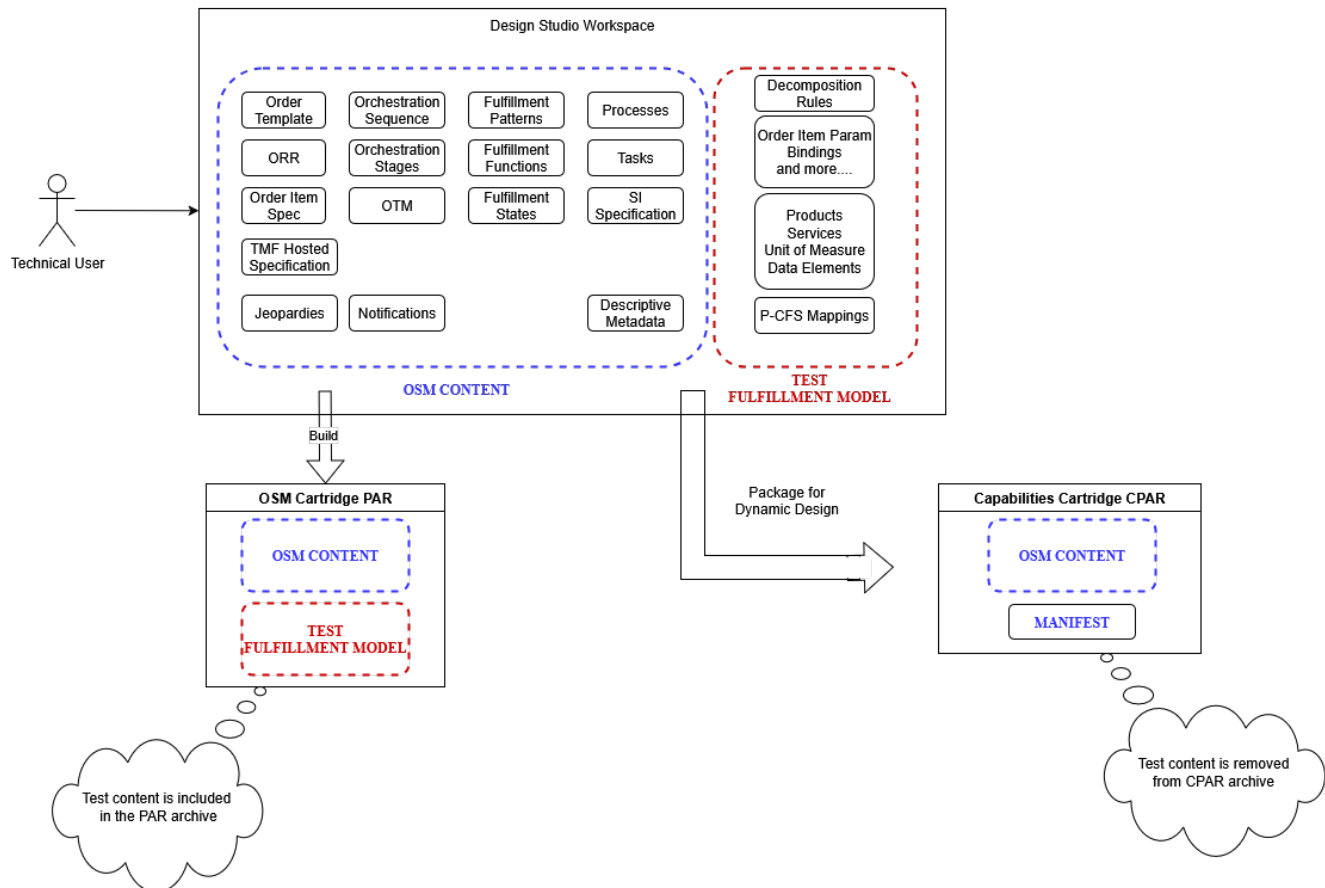
OSM enrichment data includes entities or other metadata created during the assembly process, including:

- Order Item Parameter Bindings

- Transformation Manager
- Product to Fulfillment Pattern Mappings

The following flowchart illustrates the flow of the test data usage during design journeys.

Figure 20-9 Test Data During Design Journeys



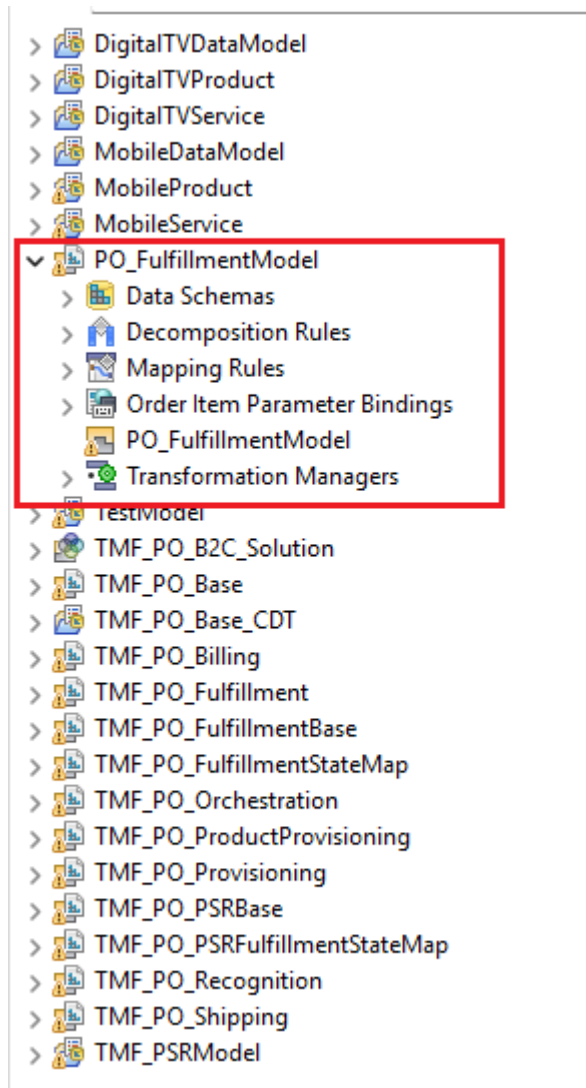
Cartridge Guidelines for Test Data

The cartridges must cleanly separate the re-usable entities (the capabilities cartridge) from the test data. The test data is defined across different cartridges depending on the type of data it holds such as OSM entity, OSM enrichment or PSR (Product, Service, Resource).

OSM Test Content

You must define OSM entities and OSM enrichment data in a single OSM cartridge. This cartridge will be specified as the **Fulfillment Model Cartridge** during the generation of the capabilities cartridge.

The following image shows the sample Fulfillment Model Cartridge and its test data.

Figure 20-10 Sample Fulfillment Model Cartridge

Conceptual Model Test Content

Conceptual model entities comprise a large portion of the capabilities test content. These entities are defined in Design Studio model projects. An OSM cartridge is designated as the **container cartridge** for these model projects so that the required metadata can be generated into the OSM cartridge.

Note

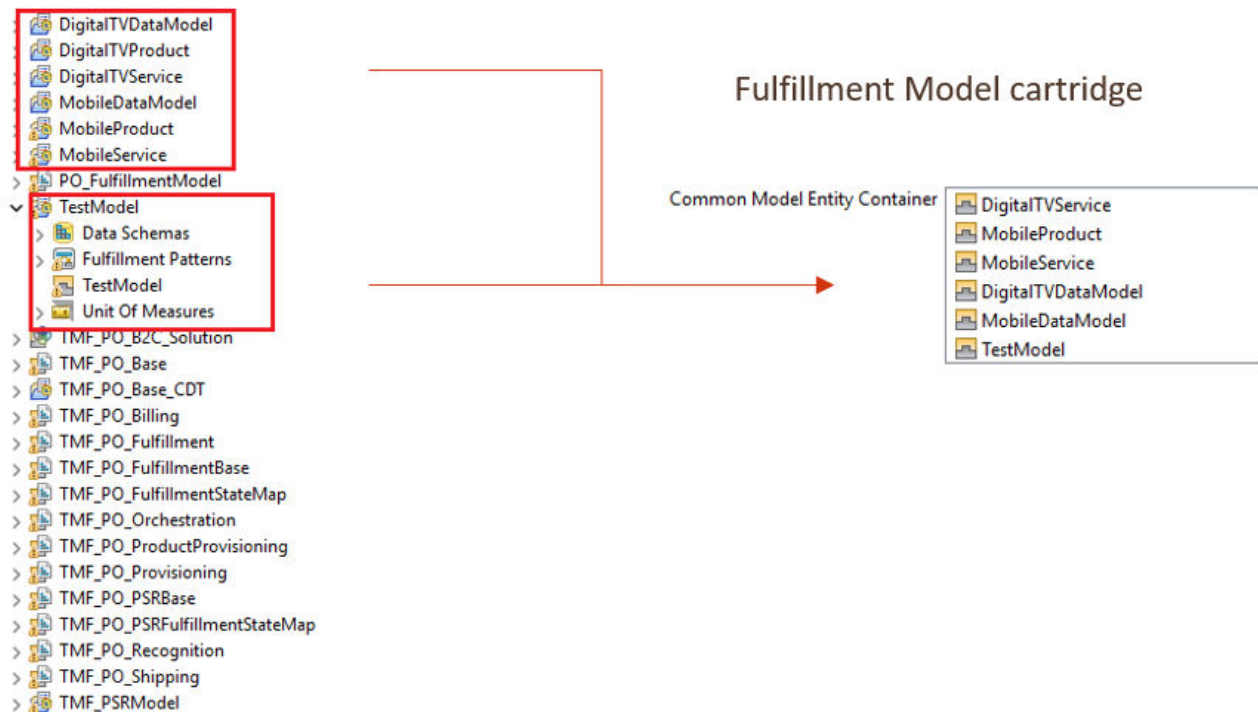
For more information about Conceptual Models, refer to Working with Conceptual Models.

The reference sample workspace contains Product and Service definitions for Mobile and Digital TV domains and the **PO_FulfillmentModel** acts as the container cartridge.

Additionally, unit of measure converters which are available in Solution Designer and conceptual fulfillment patterns which are only required in Design Studio, are defined in model projects and are contained in **PO_FulfillmentModel**.

The following image shows the sample PSR project.

Figure 20-11 Sample PSR Project



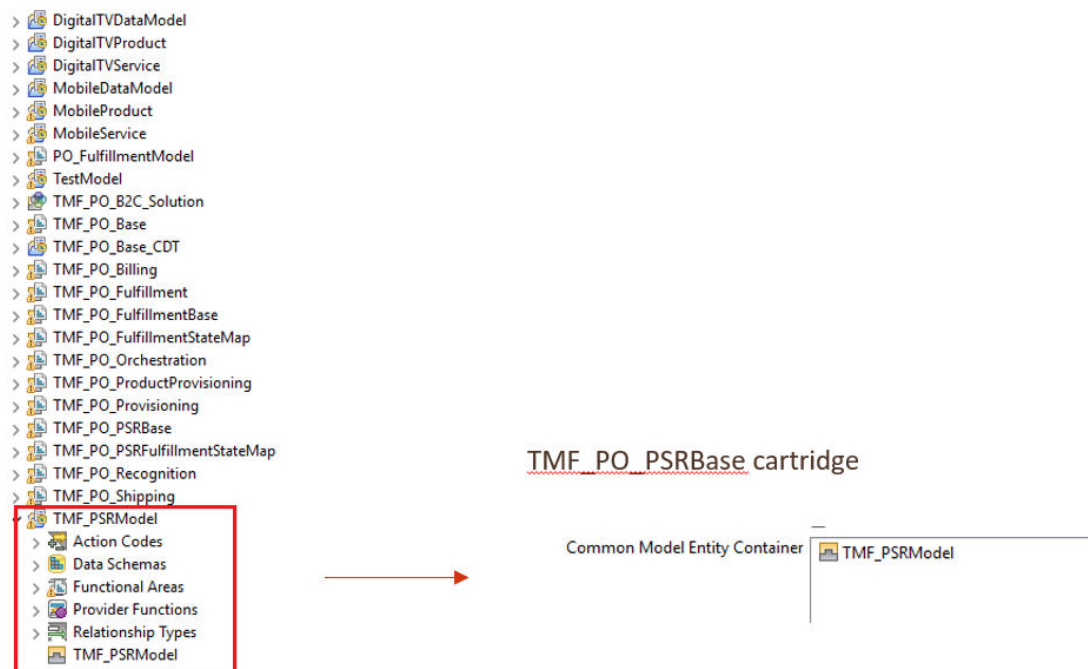
Reusable Conceptual Model Content

When order transformation is used in the fulfillment logic, then you must define several other conceptual entities. They are considered reusable content from a dynamic design perspective.

You must preserve these entities when building the cartridge. Therefore a separate OSM cartridge must act as the container for the associated metadata. In the reference sample workspace, the **TMF_PO_PSRBase** takes on this role.

The following image shows the reusable conceptual model content.

Figure 20-12 Reusable Conceptual Content



Capabilities Cartridge Build

This section provides an overview of the process and the output that you get when building the capabilities cartridge.

Using the Capabilities Cartridge Wizard

When you are using the **Capabilities Cartridge Wizard**, you can select a set of order item properties for exposure in Solution Designer. You should restrict the properties selected to what *needs* to be exposed to correctly configure routing rules (OSM decomposition rules).

Properties that are created for the cartridge fulfillment logic do not need to be exposed to the business modeler and therefore should not be selected. Additionally, you must not select properties that use **date**, **time** or **datetime** as they are not supported by dynamic assembly.

About the CPAR File

When you are using the **Capabilities Cartridge Wizard**, you need to designate a cartridge in the workspace as the **fulfillment model** cartridge. This will be the cartridge that contains the test data.

Based on the inputs you provide to the wizard, Design Studio strips the test content before packaging the CPAR archive. This stripped content includes everything in the **/model** directory.

The cartridge itself is still packaged in the CPAR and retains the following:

- Cartridge dependencies and model variables
- Everything in the **/resources** directory

- **xmlCatalogs**

Cartridge developers need to note that the pruning of test data is not selective, and any non-test data that gets defined in the fulfillment model will be lost when the CPAR is generated.

Capabilities Manifest Details

The capabilities manifest consists of 2 types of data:

- Entities that are exposed in Solution Designer and can be referenced when configuring a fulfillment model
- Metadata that is used internally by the dynamic assembly process

None of this data should be manipulated by a human user.

The table below describes the capabilities that are exposed in Solution Designer and a description of how each is populated.

Table 20-1 Exposed Capabilities in Solution Designer

Field	Description
description	Textual summary of the capabilities offered in the cartridge, to be used by modelers in Solution Designer. Usage summary entered in the Capabilities Cartridge Wizard .
orderItemSpecification/property	List of order item specification properties selected in the Capabilities Cartridge Wizard . These are available for use in routing rule conditions in Solution Designer.
orderComponents	Order components are grouped according to their classification into functions, systems, or granularities. The Product configuration in each orchestration stage drives the classification (only leaf components). • Function: Components produced by stage one. • System: Components produced by stage two. • Granularity: Components produced by stage three. These are available in routing rule configuration in Solution Designer.
fulfillmentMode	The standard fulfillment mode for TMF orders - deliver . This is informational in Solution Designer.
fulfillmentPattern	List of fulfillment patterns and their orchestration plan details. Products are mapped to these patterns in Solution Designer.
component	List of order components that are part of the orchestration plan of a fulfillment pattern. These are used to build a graphical representation of the pattern in Solution Designer.
transition	List of transition dependencies for the order components in a fulfillment pattern. These are used to build a graphical representation of the pattern in Solution Designer.

For more information, refer to Packaging and Deploying a Capabilities Cartridge in *Design Studio Modeling OSM Orchestration*.

A

Behaviors Quick Reference

The following pages contain a quick reference for Oracle Communications Order and Service Management (OSM) behaviors which you can print and keep as a work aid.

For comprehensive information on behaviors, see "[Modeling Behaviors](#)."

OSM Behavior Type Overview

[Table A-1](#) provides an overview of the OSM behaviors.

Table A-1 Behavior Type Overview

Behavior Type Name	Order	Synopsis	Default	Applies To	Parent/Child Inheritance
Calculate Behavior	1st	Calculates the value of the data instance node.	None	All value nodes.	Does not inherit.
Style Behavior: Appearance Facet	2nd	Specifies the appearance of a data instance node: • DEFAULT: the default appearance should be used • FULL: all choices should be rendered at all times. • COMPACT: a fixed number of choices should be rendered, with scrolling facilities as needed. • MINIMAL: a minimum number of choices should be rendered with a facility to temporarily render additional choices.	Data type specific. For Boolean type fields: Compact For Lookup type fields: Minimal	Boolean and Lookup type value nodes. Nodes with Lookup behaviors that have only one displayed column.	Does not inherit.
Style Behavior: CSS Style Facet	2nd	Specifies the HTML CSS style attributes of the data instance node and label.	None	All value and group nodes.	Does not inherit.
Style Behavior: CSS Class Facet	2nd	Specifies the HTML CSS Class name of the data instance node and label.	None	All value and group nodes.	Does not inherit.
Style Behavior: Newline Facet	2nd	Specifies whether a line-break is inserted before the node causing it to be displayed at the start of a new line.	False	All value nodes.	Does not inherit.
Style Behavior: Secret Facet	2nd	Ensures unauthorized users are now allowed to view the contents of nodes containing sensitive information.	True	All value nodes except for modifiable (read/write) lookups and boolean values.	Does not inherit.

Table A-1 (Cont.) Behavior Type Overview

Behavior Type Name	Order	Synopsis	Default	Applies To	Parent/Child Inheritance
Style Behavior: Layout Facet	2nd	Specifies the organization of a group's child nodes into tabbed pages.	None	All group nodes.	Does not inherit.
Style Behavior: Location Facet	2nd	Specifies the tabbed page that this group will be placed in.	None	All group nodes.	Does not inherit.
Information Behavior	3rd	Specifies the label, hint, and help information for the data instance node.	None	All value and group nodes.	Does not inherit.
Relevant Behavior	4th	Indicates whether the data instance node is currently relevant. Data instance nodes with this property evaluating to false are not displayed in the view. If this property is False, other behaviors for this node are not evaluated.	True	All value and group nodes.	If any ancestor node evaluates to false, this value is treated as false. Otherwise, the local value is used.
Lookup Behavior	5th	Specifies a set of dynamic generated choices for the data instance node.	Static lookup values (if any) specified in the OSM Model data dictionary.	All value nodes that are of type lookup, number, or text.	Does not inherit.
Constraint Behavior: Attachment Facet	6th	Specifies a condition that needs to be satisfied for the associated order attachment content to be considered valid. NOTE: This facet is only supported through programmatic behavior implementations.	True	Attachment nodes.	Does not inherit.
Read Only Behavior	7th	Describes whether the value is restricted from changing. This behavior overrides the static read-only value specified in the OSM Model View Node.	Default specified by the static read-only value on the OSM Model View Node.	All value and group nodes.	If any ancestor node evaluates to true, this value is treated as true. Otherwise, the local value is used.
Event Behavior	8th	Specifies an action to perform when a given event occurs.	None	Value nodes.	Does not inherit.
Constraint Behavior	N/A	Specifies a condition that needs to be satisfied for the associated data instance node to be considered valid. If the condition is not satisfied (evaluates to false), then messages are displayed to the user.	True	All value and group nodes.	Does not inherit.

Table A-1 (Cont.) Behavior Type Overview

Behavior Type Name	Order	Synopsis	Default	Applies To	Parent/Child Inheritance
Data Instance Behavior	N/A	Defines a container in which instances can be declared. It has no affect on the user interface display of the element for which the behavior is defined.	None	All elements and structures	Children. (Applies to element relationships within a structure. This is different than the inheritance of behaviors between the data dictionary, order, and task levels.

Common Behavior Elements

This section describes the syntax for declaring common behavior elements.

Annotation Element

```
<annotation>
  <documentation lang="NCName">... </documentation>
</annotation>
```

Description Element

```
<description>string</description>
```

Instance Element

```
<instance name="NCName" lang="NCName"
  xsi:type="inlineInstance|externalInstance">
  For inlineInstanceType, any valid XML document is allowed up to
  4000 characters in length.
  For externalInstanceType, adapter followed by parameter*, and cache*
</instance>
```

Adapter Element [externalInstanceType]

```
<adapter>com.mslv.oms.view.rule.adapter.ObjectelAdapter
|com.mslv.oms.view.rule.adapter.OrderAdapter
|com.mslv.oms.view.rule.adapter.XMLAttachmentAdapter
|com.mslv.oms.view.rule.adapter.XMLFileAdapter
|javaClassNameType<
/adapter>
```

Parameter Element [externalInstanceType]

```
<parameter name="string">string-expr</parameter>
```

Cache Element

```
<cache>
  <scope>NONE|NODE|SYSTEM</scope>
  <timeout>positiveInteger</timeout>
  <maxSize>positiveInteger</maxSize>
</cache>
```

Expression Element

```
<expression>boolean-expr</expression>
```

Declaring Behaviors in OSM XML Model

This section describes the syntax for declaring behaviors in OSM XML model.

Data Dictionary Level

```
<dataDictionary>  element+
  <element name="nameType" xsi:type="booleanType|currencyType|dateType
    |dateTimeType|phoneType|groupType|textType|numericType|lookupType">
    description, viewRule*, followed by type specific content
  </element>
</dataDictionary>
```

Master Order Template Level

```
<masterOrderTemplate>
  dataNode+
  <dataNode element="NCName">
    viewRule*, followed by dataNode*
  </dataNode>
</masterOrderTemplate>
```

View Level

```
<viewNode element="NCName">
  editable?, minOccurs?, maxOccurs?, viewRule*, viewNode*
  <editable>boolean</editable>
  <minOccurs>unsignedInt</minOccurs>
  <maxOccurs>unsignedInt</maxOccurs>
</viewNode>
```

Data Provider Overview

[Table A-2](#) provides an overview of the built-in and custom data providers. See ["Using Data Providers to Retrieve Data"](#) for details.

Table A-2 Data Provider Overview

Data Provider	Synopsis	Parameter
Custom	Uses data provided by a custom-defined Java class.	Implementation-defined

Table A-2 (Cont.) Data Provider Overview

Data Provider	Synopsis	Parameter
JDBC	Lets OSM query any JDBC database, then use the results within a behavior.	oms:dataSource, oms:sql, in:1 . . . in:n?, out:1 . . . out:n?
Objectel	Uses results of an Objectel Server Extension as an instance.	obj:extensionName, obj:jmsFactory?, obj:queue?, obj:allowErrorResponse?. Other parameters passed to Objectel
Order	Uses data from any OSM order as an external instance.	oms:OrderID, oms:View oms:OrderHistID
Property File	Retrieves an external Java property file with a given name from the classpath.	oms:url
SOAP	Lets you open up OSM to web services, using the HTTP protocol.	soap.endpoint, soap.action?, soap.envelope, soap.body, soap.header?, oms:credentials.username?, oms:credentials.password?, oms:credentials.scope.host?, soap.allowErrorResponse
XML Attachment	Uses an XML attachment from any OSM order as an instance.	oms:OrderID, oms:FileName
XML File	Uses an XML file from any URL as an instance.	oms:url
XML Validation	Validates a provided XML instance document according to a user-defined schema. The document may be either a URL or an element. The schema may also be a URL or an element.	document, schema

Programmatic Behavior Implementation Overview

[Table A-3](#) provides an overview of the programmatic behavior implementation.

Table A-3 Programmatic Behavior Implementation Overview

Rule Type Name	Java Interface	Method Names	Parameter Types	Return Types
Calculate Rule	com.mslv.oms.view.CalculateRule	calculate_<mnemonic>	com.mslv.oms.view.rule.ViewRuleContext org.w3c.dom.Node	Any Java primitive or descendent of java.lang.Object
Style Rule: Appearance Facet	com.mslv.oms.view.StyleRule	appearance_<mnemonic>	com.mslv.oms.view.rule.ViewRuleContext org.w3c.dom.Node	int NOTE: Return value must be one of FULL_APPEARANCE, COMPACT_APPEARANCE, MINIMAL_APPEARANCE defined on the StyleRule interface
Style Rule: CSS Style Facet	com.mslv.oms.view.StyleRule	style_<mnemonic> styleForLabel_<mnemonic>	com.mslv.oms.view.rule.ViewRuleContext org.w3c.dom.Node	java.util.Map<String, String>

Table A-3 (Cont.) Programmatic Behavior Implementation Overview

Rule Type Name	Java Interface	Method Names	Parameter Types	Return Types
Style Rule: CSS Class Facet	com.mslv.oms.view.StyleRule	cssClass_<mnemonic> cssClassForLabel_<mnemonic>	com.mslv.oms.view.rule.ViewRuleContext org.w3c.dom.Node	String
Style Rule: Newline Facet	com.mslv.oms.view.StyleRule	newline_<mnemonic>	com.mslv.oms.view.rule.ViewRuleContext org.w3c.dom.Node	boolean
Style Rule: Secret Facet	com.mslv.oms.view.StyleRule	secret_<mnemonic>	com.mslv.oms.view.rule.ViewRuleContext org.w3c.dom.Node	boolean
Information Rule	com.mslv.oms.view.InformationRule	information_<mnemonic>	com.mslv.oms.view.rule.ViewRuleContext org.w3c.dom.Node	java.util.Map<String, String>
Relevant Rule	com.mslv.oms.view.RelevantRule	relevant_<mnemonic>	com.mslv.oms.view.rule.ViewRuleContext org.w3c.dom.Node	boolean
Lookup Rule	com.mslv.oms.view.LookupRule	lookup_<mnemonic>	com.mslv.oms.view.rule.ViewRuleContext org.w3c.dom.Node	String[] String[][] java.util.Map<Object, Object> java.util.Collection<Object>
Constraint Rule	com.mslv.oms.view.ConstraintRule	constraint_<mnemonic>	com.mslv.oms.view.rule.ConstraintContext org.w3c.dom.Node	String[] com.mslv.oms.view.rule.ConstraintResult com.mslv.oms.view.rule.ConstraintResult[] java.util.List<com.mslv.oms.view.rule.ConstraintResult>
Constraint Rule Attachment Facet	com.mslv.oms.view.ConstraintRule	constraint_attachment	com.mslv.oms.view.rule.ConstraintContext org.w3c.dom.Node java.io.InputStream	String[] com.mslv.oms.view.rule.ConstraintResult com.mslv.oms.view.rule.ConstraintResult[] java.util.List<com.mslv.oms.view.rule.ConstraintResult>
Read Only Rule	com.mslv.oms.view.ReadOnlyRule	readonly_<mnemonic>	com.mslv.oms.view.rule.ViewRuleContext org.w3c.dom.Node	boolean

Table A-3 (Cont.) Programmatic Behavior Implementation Overview

Rule Type Name	Java Interface	Method Names	Parameter Types	Return Types
Event Rule	com.mslv.oms.view.EventRule	event_<mnemonic>	com.mslv.oms.view.rule.ViewRuleContext org.w3c.dom.Node	java.util.Map<String, String> NOTE: Map key must be EventRule.VALUE_CHANGED_EVENT. Map value must be one of EventRule.REFRESH_ACTION or EventRule.SAVE_ACTION

B

XQuery Examples

You use XQuery expressions in various locations to implement key aspects of the Oracle Communications Order and Service Management (OSM) orchestration functionality. For information about these XQuery expressions, refer to the following topics:

- [General XQuery Information](#)
- [Order Recognition Rule XQuery Expressions](#)
- [Decomposition XQuery Expressions](#)
- [Dependency XQuery Expressions](#)
- [Order Transformation Manager XQuery Expressions](#)

General XQuery Information

This topic contains general or reference information about XQuery that applies the same in different situations.

When working with XQuery expressions, see the following topics:

- [About Creating XQuery Expressions with Design Studio](#)
- [OSM XQuery Functions](#)
- [Referencing Items from a Distributed Order Template in XQuery Expressions](#)

About Creating XQuery Expressions with Design Studio

In general, the way you enter XQuery information into editors in Oracle Communications Service Catalog and Design - Design Studio is the same, regardless of the editor. The XQuery control in Design Studio generally has three tabs: XQuery, Instances, and Information. Following are general instructions for entering XQuery information into each of these tabs in Design Studio.

Using the XQuery Tab

The **XQuery** tab allows you to configure XQuery-based rules or elements, or identify the source of the XQuery-based rules or elements. Select one of the following options:

- Select **None** if the XQuery configuration is optional and not configured. When you select this option, Design Studio disables the remaining options in the subtabs.
- Select **Expression** and enter the XQuery expression in the corresponding text box. Click **Edit** to open the Edit XQuery dialog box, which displays the configured XQuery expression in a larger and resizable text box. You can edit the expression in the Edit XQuery dialog box and click **OK** to save your changes, or click **Cancel** to dismiss the dialog box without saving the changes.

Note

Design Studio provides XQuery validation on basic syntax and semantics, and denotes errors with Problem markers.

- Select **File** to denote that the XQuery configuration is located in a file saved to the project **resources** directory. This option enables you to write your XQuery expressions using any XQuery editing application you have installed in your Eclipse environment. See the Eclipse online Help topic *Associating editors with file types* for more information.

Click **Select** to open the Select XQuery File dialog box, which displays all XQuery files contained in the project **resources** directory. Select the appropriate XQuery file and click **OK**.

- Select **URI** to denote that the XQuery configuration is located in a remote URI location. For example, you might enter:

`http://osm_server/AIARecognitionRule.xqy`

Click **Properties** to open the Properties view, where you can define the following information for the XQuery:

- **Annotation:** The optional XML annotation element allows you to provide information about the XQuery. Enter information (for example, HTML-formatted information) for external systems into the **Annotation <appinfo>** field. Enter information for human users into the **Annotation <documentation>** field.
- **Language:** When you work with multiple languages, you can select a different language for displaying the description and annotation. For more information, see "Defining Language Preferences" in the Design Studio Modeling OSM Processes Help.

Using the Instances Tab

You can define a Data Instance behavior to obtain data that is not included in the order data and make that data available to the rule. Click **Add** to add a Data Instance behavior. Select the Data Instance behavior and click **Properties** to configure the Data Instance behavior.

For more information, see "Defining Data Instance Behavior Properties" in the Design Studio Modeling OSM Processes Help.

Using the Information Tab

Use this tab if you want to describe the intended use of the rule. For example, you might describe the functionality of a complex rule or provide instructions on its use.

OSM XQuery Functions

OSM-specific XQuery functions are available to you when writing XQuery expressions. These XQuery functions are contained in classes that you can declare in the prolog of your XQuery expression.

To see specifics about the functions available, install the OSM SDK and extract the OSM Javadocs from the **SDK/osm7.w.x.y.z-javadocs.zip** file (where *w.x.y.z* represents the specific version numbers for OSM). See *OSM Developer's Guide* for more information about installing the OSM SDK.

The specific classes that contain XQuery functions you might use are:

- **OrchestrationXQueryFunctions**: This class contains XQuery functions that are used in OSM Orchestration. To declare this class, put the following declaration in the prolog of your XQuery expression:

```
declare namespace osmfn =
"java:oracle.communications.ordermanagement.orchestration.generation.OrchestrationXQueryFunctions";
```

- **XQueryFunctions**: This class contains XQuery functions that are used in the order transformation manager. To declare this class, put the following declaration in the prolog of your XQuery expression:

```
declare namespace otmfn =
"java:oracle.communications.ordermanagement.orchestration.transformation.XQueryFunctions";
```

Referencing Items from a Distributed Order Template in XQuery Expressions

The distributed order template is an option you can set on an order item specification to modify the method used to store order item data. For more general information about the distributed order template, see *OSM Concepts*.

When using a distributed order template, any XQuery expressions that reference order item data must be in a particular format.

For any order item that is not a transformed order item, you must include the namespace of the order item specification. Following is an example of an XQuery reference to the **lineItemID** property on the **InputOrderItem** order item with the namespace **http://ex_input.com**:

```
/ControlData/OrderItem[@type='{http://ex_input.com}InputOrderItem']/lineItemID
```

For transformed order items, the format depends on the source of the data for the transformed order item. Data that is defined in the order item specification itself will use the namespace for the order item specification, the same way that data would be referenced for an input order item. Following is an example of an XQuery reference to the **lineItemID** property on the **OutputOrderItem** order item with the namespace **http://ex_output.com**:

```
/ControlData/OrderItem[@type='{http://ex_output.com}OutputOrderItem']/lineItemID
```

Data that has been derived from a common model entity, for example an action, will use a different format. In the following situation:

- Order item name: **OutputOrderItem**
- Order item namespace: **http://ex_output.com**
- Conceptual model entity (in this case an Action) name: **SA_Add_Internet**
- Conceptual model cartridge name: **Model_Broadband**
- Conceptual model cartridge version: **1.0.0.0.0**
- Parameter name on SA_Add_Internet: **serviceLevel**

The reference would look like this:

```
/ControlData/OrderItem[@type='{http://ex_output.com}OutputOrderItem']/
dynamicParams[@type='{Model_Broadband/1.0.0.0.0}SA_Add_InternetType']/serviceLevel
```

Note that the type for the parameters contained in the conceptual model entity has the string "Type" appended to the name of the entity. Thus, the type contains **SA_Add_InternetType** rather than just SA_Add_Internet.

Order Recognition Rule XQuery Expressions

The following topics provide reference information about order recognition rule XQuery expressions:

- [About Recognition Rule XQuery Expressions](#)
- [About Validation Rule XQuery Expressions](#)
- [About Order Priority XQuery Expressions](#)
- [About Order Reference XQuery Expressions](#)
- [About Order Data Rule XQuery Expressions](#)

About Recognition Rule XQuery Expressions

This topic describes how to use the Order Recognition Rule editor Recognition Rule area **XQuery** tab to write an expression that specifies a customer order and associates it with an OSM target order type. The XQuery has the following characteristics:

- Context: The input document for the Recognition Rule **XQuery** is the customer order. For more information about typical customer order structures, see *OSM Concepts*.
- Prolog: You can declare the namespace for the customer order if you want to use the contents of the order as part of the recognition rule or you can omit the declaration if you only want to check the incoming customer order namespace. For example:

```
declare namespace im="http://xmlns.oracle.com/InputMessage";
```

- Body: You must match the namespace you want to select for order processing with the namespace of the incoming customer order. For example, the following expression retrieves the namespace URI from the incoming customer order (**fn:namespace-uri(.)**) and compares it with this URI: **'http://xmlns.oracle.com/InputMessage'**:

```
fn:namespace-uri(.) = 'http://xmlns.oracle.com/InputMessage'
```

If you have declared a namespace in the prolog, you can also check to see if specific values exist in the order. For example, you can use the **fn:exists** function to check that an element exists. Or you can use a comparison expression such as = (equal to) or != (not equal to) to compare a value in the customer order with a value in the XQuery.

✓ Tip

Recognition rules are global entities within OSM, meaning that they can apply to any CreateOrder operation. Configure the relevancy settings and the recognition rule carefully to avoid having an incoming customer order recognized by a recognition rule that you do not intend. For more information about relevancy, see *OSM Concepts*.

For example, in a simple scenario, the XQuery is based on a namespace:

```
fn:namespace-uri(.) = 'http://xmlns.oracle.com/InputMessage'
```

The input message XML file includes the following line, which matches the namespace specified in the recognition rule:

```
<im:order xmlns:im="http://xmlns.oracle.com/InputMessage"
```

The XQuery expression returns a Boolean expression, for example, `fn:true()` or `fn:false()`

The following example searches in a specific type of order:

```
fn:namespace-uri(.) = 'http://xmlns.oracle.com/communications/sce/dictionary/
CentralOMManagedServices-Orchestration/CustomersalesOrder'
```

In a more complicated scenario, you might create an XQuery expression that looks for a specific namespace and also interrogates the data within the incoming customer order. The following example shows a recognition rule that recognizes an order based on the following criteria:

- Namespace
- Value of the **typeCode** data element in the incoming customer order. In this case, the value must be OSM-BDB. This indicates an OSM business-to-business order.
- The value of the **FulfillmentModeCode** data element in the incoming customer order. In this case, the value can be DELIVER, CANCEL, or TSQ.

```
declare namespace provord="http://xmlns.oracle.com/EnterpriseObjects/Core/EB0/ProvisioningOrder/V1";;
declare namespace corecom="http://xmlns.oracle.com/EnterpriseObjects/Core/Common/V2";;
fn:namespace-uri(.) = 'http://xmlns.oracle.com/EnterpriseObjects/Core/EB0/ProvisioningOrder/V1'
and
fn:exists(..provord:ProcessProvisioningOrderEBM/provord:DataArea/provord:ProcessProvisioningOrder/
corecom:Identification/corecom:BusinessComponentID)
and
..provord:ProcessProvisioningOrderEBM/provord:DataArea/provord:ProcessProvisioningOrder/
provord:TypeCode/text() = 'OSM-BDB'
and
(
..provord:ProcessProvisioningOrderEBM/provord:DataArea/provord:ProcessProvisioningOrder/
provord:FulfillmentModeCode/text() = 'DELIVER'
or
..provord:ProcessProvisioningOrderEBM/provord:DataArea/provord:ProcessProvisioningOrder/
provord:FulfillmentModeCode/text() = 'CANCEL'
or
..provord:ProcessProvisioningOrderEBM/provord:DataArea/provord:ProcessProvisioningOrder/
provord:FulfillmentModeCode/text() = 'TSQ'
)
```

For more information about order recognition rules see *OSM Concepts*.

About Validation Rule XQuery Expressions

This topic describes how to use the Order Recognition Rule editor Validation Rule area **XQuery** tab to write an expression that specifies nodes in the incoming customer order that must evaluate to true to accept the customer order into the system. The XQuery has the following characteristics:

- Context: The input document for the Validation Rule **XQuery** is the customer order. For more information about typical customer order structures, see *OSM Concepts*.
- Prolog: The input document for the Validation Rule **XQuery** is the customer order. You can declare the customer order namespace in the XQuery prolog. For example:

```
declare namespace im="http://xmlns.oracle.com/InputMessage";
```

- Body: The validation rule must specify customer order parameters or parameter values to evaluate to **true** for the validation to be successful. If the validation fails, the expression should return an error message.

In addition, if the Validation Rule fails, OSM automatically creates the order and sets the order state to Failed. The inbound message and validation failure output are attached to the order for reference. You can display and manage the order failure in the Order Management web client.

The following sample XQuery checks for the existence of a sender ID:

```
if (fn:exists(./header/c:Sender/c:ID) and ./header/c:Sender/c:ID != '')
then (true())
else concat("SEVERE", "Message Header should contain Sender ID", header/Sender/ID")
```

The following sample XQuery checks for correct values in the **typeCode** data element in the incoming customer order:

```
if (fn:exists($orderLine/im:ItemReference/im:TypeCode)
and
$orderLine/im:ItemReference/im:TypeCode != '')
then
(
if ($orderLine/im:ItemReference/im:TypeCode = "PRODUCT" or
$orderLine/im:ItemReference/im:TypeCode = "OFFER" or
$orderLine/im:ItemReference/im:TypeCode = "BUNDLE") then ()
else
local:reportIssue("ERROR", "Product Type should be one of: PRODUCT, OFFER, BUNDLE",
$lineNum, "ProcessProvisioningOrderEBM/DataArea/ProcessProvisioningOrder/
ProvisioningOrderLine/ItemReference/TypeCode")
)
```

Given this XQuery sample, the following part of a customer order would evaluate to true because the typeCode element value is **BUNDLE**.

```
<!-- FIXED BUNDLE - BUNDLE -->
<im:salesOrderLine>
  <im:lineId>2</im:lineId>
  <im:promotionalSalesOrderLineReference>1</im:promotionalSalesOrderLineReference>
  <im:serviceId></im:serviceId>
  <im:requestedDeliveryDate>2001-12-31T12:00:00</im:requestedDeliveryDate>
  <im:serviceActionCode>Add</im:serviceActionCode>
  <im:itemReference>
    <im:name>Fixed Bundle</im:name>
    <im:typeCode>BUNDLE</im:typeCode>
    <im:specificationGroup />
  </im:itemReference>
</im:salesOrderLine>
```

For more information about validation rules see *OSM Concepts*.

About Order Priority XQuery Expressions

This topic describes how to use the Order Recognition Rule editor Order Priority area **XQuery** tab to write an expression that specifies an element value in the incoming customer order that identifies the order priority. The XQuery has the following characteristics:

- Context: The input document for the Order Priority XQuery is the customer order. For more information about typical customer order structures, see *OSM Concepts*.

- **Prolog:** You can declare the customer order namespace in the XQuery prolog. For example:

```
declare namespace im="http://xmlns.oracle.com/InputMessage";
```
- **Body:** The Order Priority body must specify the node that contains the order priority value.

For more information about creating order priority XQuery expressions in the order recognition rule and about creating order priority ranges for an order type, see *OSM Concepts*.

About Order Reference XQuery Expressions

This topic describes how to use the Order Recognition Rule editor Order Reference area **XQuery** tab to write an expression that specifies an element value in the incoming customer order that identifies the order reference. The XQuery has the following characteristics:

- **Context:** The input document for the Order Reference XQuery is the customer order. For more information about typical customer order structures, see *OSM Concepts*.
- **Prolog:** You can declare the customer order namespace in the XQuery prolog. For example:

```
declare namespace im="http://xmlns.oracle.com/InputMessage";
```

- **Body:** The Order Reference body must specify the node that contains the order reference value.

The following example shows a transformation rule XQuery expression that retrieves the order reference number (as a string) from the **numSalesOrder** field in the incoming customer order:

```
declare namespace im="http://xmlns.oracle.com/InputMessage";
let $order := ../im:order
return
$order/im:numSalesOrder/text()
```

For more information about order reference, see *OSM Concepts*.

About Order Data Rule XQuery Expressions

This topic describes how to use the Order Recognition Rule editor Order Data Rule area **XQuery** tab to write an expression that specifies nodes in the incoming customer order that must be used in the creation task. The XQuery has the following characteristics:

- **Context:** The input document for the Order Data Rule XQuery is the customer order. For more information about typical customer order structures, see *OSM Concepts*.
- **Prolog:** You can declare the customer order namespace in the XQuery prolog. For example:

```
declare namespace im="http://xmlns.oracle.com/InputMessage";
```

- **Body:** The Order Data Rule body must map the customer order element values to the corresponding creation task Task Data values.

The following example shows the fields in an incoming customer order:

```
<im:customerAddress>
  <im:locationType>Street</im:locationType>
  <im:nameLocation>Jangadeiros</im:nameLocation>
  <im:number>48</im:number>
  <im:typeCompl>floor</im:typeCompl>
  <im:numCompl>6</im:numCompl>
  <im:district>Ipanema</im:district>
```

```

    <im:codeLocation>5000</im:codeLocation>
    <im:city>Rio de Janeiro</im:city>
    <im:state>RJ</im:state>
    <im:referencePoint>Gen. Osorio Square</im:referencePoint>
    <im:areaCode>22420-010</im:areaCode>
    <im:typeAddress>Building</im:typeAddress>
  </im:customerAddress>

```

Following is a sample order data in a creation task. In the example, the following data is contained in a CustomerDetails element:

- locationType
- nameLocation
- number
- typeCompl
- numCompl
- district
- codeLocation
- city
- state
- referencePoint
- areaCode
- typeAddress

The following XQuery expression specifies a variable for the location of the customerAddress node in the customer order that can then be used to map customerAddress child element values to CustomerDetails task data elements:

```
let $details := $customer/mes:customerAddress
```

The following XQuery expression performs this mapping:

```

return<_root>
<CustomerDetails>
  <locationType>{$details/im:locationType/text()}</locationType>
  <nameLocation>{$details/im:nameLocation/text()}</nameLocation>
  <number>{$details/im:number/text()}</number>
  <typeCompl>{$details/im:typeCompl/text()}</typeCompl>
  <numCompl>{$details/im:numCompl/text()}</numCompl>
  <district>{$details/im:district/text()}</district>
  <codeLocation>{$details/im:codeLocation/text()}</codeLocation>
  <city>{$details/im:city/text()}</city>
  <state>{$details/im:state/text()}</state>
  <referencePoint>{$details/im:referencePoint/text()}</referencePoint>
  <areaCode>{$details/im:areaCode/text()}</areaCode>
  <typeAddress>{$details/im:typeAddress/text()}</typeAddress>
</CustomerDetails>
</_root>

```

In the following example, the XQuery expression returns the <_root> portion of the order. The ControlData portion of the order is populated by the system during the generation of the orchestration plan.

```

declare namespace cso="http://xmlns.oracle.com/communications/sce/dictionary/
CentralOMManagedServices-Orchestration/CustomSalesOrder";
let $customer := //cso:CustomerAccount

```

```
return
<_root>
<OrderHeader>
<AccountIdentifier>{$customer/cso:AccountID/text()}</AccountIdentifier>
</OrderHeader>
</_root>
```

For more information about order data rules, see *OSM Concepts*.

Decomposition XQuery Expressions

This topic includes information about order recognition rule XQuery expressions related to order decomposition:

- [About Orchestration Sequence XQuery Expressions](#)
- [About Order Item Specification XQuery Expressions](#)
- [About Fulfillment Pattern Order Component XQuery Expressions](#)
- [About Decomposition Rule Condition XQuery Expressions](#)
- [About Component Specification Custom Component ID XQuery Expressions](#)
- [About Component Specification Duration XQuery Expressions](#)
- [About Fulfillment Pattern Duration XQuery Expressions](#) (deprecated)
- [About Fulfillment Pattern Component Duration XQuery Expressions](#)

About Orchestration Sequence XQuery Expressions

The Orchestration Sequence editor provides the following areas to define XQuery expressions related to order decomposition:

- [About Order Sequence Order Item Selector XQuery Expressions](#)
- [About Order Sequence Fulfillment Mode XQuery Expressions](#)

About Order Sequence Order Item Selector XQuery Expressions

This topic describes how to use the Orchestration Sequence editor Order Item Selector area **XQuery** tab to write an expression that specifies which node-set to use from the customer order as order items and has the following characteristics:

- Context: The input document for the Order Item Selector XQuery is the customer order. For more information about typical customer order structures, see *OSM Concepts*.
- Prolog: You can declare the customer order namespace in the XQuery prolog.
- Body: The XQuery body must specify the customer order node-sets that OSM then uses as order items.

The following example shows an order item selector XQuery where the **<salesOrderLine>** node-set is specified. OSM can now use the data in the **<salesOrderLine>** node-set in the incoming customer order in the order items. There can only be one node-set selected per sequence.

```
declare namespace im="http://xmlns.oracle.com/InputMessage";
./im:salesOrderLine
```

About Order Sequence Fulfillment Mode XQuery Expressions

This topic describes how to use the Orchestration Sequence editor Fulfillment Mode area **XQuery** tab to write an expression that specifies the fulfillment mode for the orchestration sequence from a customer order element and has the following characteristics:

- Context: The input document for the Fulfillment Mode Expression area XQuery is the customer order. For more information about typical customer order structures, see *OSM Concepts*.
- Prolog: The input document for the Fulfillment Mode Expression area XQuery is the incoming customer order. You must declare the customer order namespace in the XQuery prolog.
- Body: The XQuery body must specify the fulfillment mode.

Typically, the fulfillment mode is specified in the order header. For example:

```
<im:FulfillmentModeCode>Deliver</im:FulfillmentModeCode>
```

In the following example, the XQuery looks in the incoming customer order (SalesOrder) for the **<FulfillmentModeCode>** element. It returns the text contained in that element.

```
declare namespace
im="http://xmlns.oracle.com/InputMessage";
<osm:fulfillmentMode name="{fn:normalize-space(../im:SalesOrder/im:DataArea/im:FulfillmentModeCode/
text())}"
```

This is the XML in the incoming customer order:

```
<im:FulfillmentModeCode>Deliver</im:FulfillmentModeCode>
```

In this case, the XQuery returns **Deliver**.

About Order Item Specification XQuery Expressions

The Order Item Specification editor provides the following areas to define XQuery expressions related to order decomposition:

- [About Order Item Specification Order Item Property XQuery Expressions](#)
- [About XQuery Expressions for Mapping Product Specifications and Fulfillment Patterns](#)
- [About Order Item Specification Order Item Hierarchy XQuery Expressions](#)
- [About Order Item Specification Condition XQuery Expressions](#)

About Order Item Specification Order Item Property XQuery Expressions

This topic describes how to use the Order Item Specification editor, **Order Item Properties** tab, Property Expression area, **XQuery** tab to write an expression that specifies order item properties based on the input context. These expressions have the following characteristics:

- Context: The Property Expression area XQuery input document is a node from the node-set returned by the order item selector (see "[About Order Sequence Order Item Selector XQuery Expressions](#)"). OSM runs every order item Property Expression area XQuery against each node (starting with the first and ending with the last node) in the node-set returned by the order item selector.
- Prolog: You can declare the following variables within the prolog to access additional context information:

- The **\$inputDoc** variable can be declared in the prolog of an OSM XQuery to provide access to the original input customer order. This external function can be useful if you need to generate order item properties based on elements outside of the order item node-set defined in the order item selector. The format for declaring this variable in the XQuery prolog is:

```
declare variable $inputDoc as document-node() external;
```

You can then access this variable within the XQuery body. For example, the following XQuery body uses **\$inputDoc** to define the **ItemReferenceName** value:

```
let $inputOrderData:= $inputDoc/GetOrder.Response/_root
fn:normalize-space(cso:ItemReferenceName/text())
```

For more information about typical customer order structures, see *OSM Concepts*.

- The **\$salesOrderLines** variable can be used in an OSM XQuery to provide access to original order item node-set before it is selected by the orchestration sequence's order item selector. This can be useful if the order item selector XQuery changes the selected order item node-set (for example, by rearranging the order of the elements). The format for declaring this variable in the XQuery prolog is:

```
declare variable $salesOrderLines as document-node() external;
```

You can access this variable within the XQuery body. For more information about typical customer order structures, see *OSM Concepts*.

- **Body:** The XQuery body must specify the order item element that provides the values for each order item property you define.

After these XQuery expressions have run against an order item, the order item and the order item properties become internally accessible as an XQuery context for other OSM entities. For example,

```
<osm:orderItem
  xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model" id="1288881040699">
  <osm:name>Commercial Fixed Service [Add]</osm:name>
  <osm:orderItemSpec xmlns="http://xmlns.oracle.com/communications/ordermanagement/model">
    <osm:name>CustomerOrderItemSpecification</osm:name>
    <osm:namespace>
      http://oracle.communications.ordermanagement.unsupported.centralom
    </osm:namespace>
  </osm:orderItemSpec>
  <osm:productSpec xmlns="http://xmlns.oracle.com/communications/ordermanagement/model">
    <osm:name>Service.Fixed</osm:name>
    <osm:namespace>
      http://oracle.communications.ordermanagement.unsupported.centralom
    </osm:namespace>
  </osm:productSpec>
  <osm:properties xmlns:im="http://oracle.communications.ordermanagement.unsupported.centralom">
    <im:typeCode>PRODUCT</im:typeCode>
    <im:parentLineId>3</im:parentLineId>
    <im:requestedDeliveryDate>2013-06-31T12:00:00</im:requestedDeliveryDate>
    <im:lineItemName>Commercial Fixed Service [Add]</im:lineItemName>
    <im:lineId>4</im:lineId>
    <im:ServiceActionCode>UPDATE</im:ServiceActionCode>
    <im:productSpec>Fixed Service Plan Class</im:productSpec>
    <im:serviceId>552131313131</im:serviceId>
    <im:fulfillPatt>Service.Fixed</im:fulfillPatt>
    <im:lineItemPayload> [34 lines]
    <im:region>Sao Paulo</im:region>
  </osm:properties>
</osm:orderItem>
```

The following examples show some ways to map data in an incoming customer order to an order item property. The current context is a single node from salesOrderLines, which is one of the nodes returned by executing the orchestration sequence order item selector against the input message (see "[About Order Sequence Order Item Selector XQuery Expressions](#)").

- Order management personnel need to know what the requested delivery date is for order items. Adding the date to the order item allows the order management personnel to see it in the OSM web clients. In addition, OSM needs the requested delivery date to calculate the order start date.

To retrieve the requested delivery data for an order item, OSM looks in the incoming customer order for the **<requestedDeliveryDate>** element:

```
<im:requestedDeliveryDate>2001-12-31T12:00:00</im:requestedDeliveryDate>
```

The definition of the **requestedDeliveryDate** order item property includes the following XQuery, which returns the text of the **<requestedDeliveryDate>** element:

```
declare namespace im="http://xmlns.oracle.com/InputMessage";
fn:normalize-space(im:requestedDeliveryDate/text())
```

- Order management personnel need to identify order items in the OSM web clients. The **lineItemName** order item property includes the following XQuery:

```
declare namespace im="http://xmlns.oracle.com/InputMessage";
fn:normalize-space(fn:concat(im:itemReference/im:name/text(), '
',im:serviceActionCode/text(),'))
```

This XQuery looks for two elements, **<name>** and **<serviceActionCode>**:

```
<im:name>Fixed Caller ID</im:name>
<im:serviceActionCode>Add</im:serviceActionCode>
```

It then concatenates the text retrieved from the two elements to form the order item name, in this case Fixed Caller ID [Add].

- Order management personnel need to identify the products or product specification from the customer order so that order items can be mapped to fulfillment patterns (see "[About XQuery Expressions for Mapping Product Specifications and Fulfillment Patterns](#)"). The following example shows the product specification data in the message, contained in the **<primaryClassificationCode>** element:

```
<im:primaryClassificationCode>Mobile Service Feature Class
</im:primaryClassificationCode>
```

The **productClass** order item property uses the following XQuery expression to get the data:

```
declare namespace im="http://xmlns.oracle.com/InputMessage";
fn:normalize-space(im:itemReference/im:primaryClassificationCode/text())
```

About XQuery Expressions for Mapping Product Specifications and Fulfillment Patterns

The order item property specified in the **Fulfillment Pattern Mapping Property** field for the order item must map to an existing OSM fulfillment pattern entity. The value could be contained in a customer order, but more often, it is derived from other customer order parameter. This property is mandatory.

The construction of the fulfillment pattern mapping order item property follows the same rules as other order item property XQuery expressions. See "[About Order Item Specification Order](#)

[Item Property XQuery Expressions](#)" for more information about the XQuery context, prolog, and body.

The following describes a common scenario for deriving fulfillment patterns from product or product specification data contained in an order. In other scenarios, the mapping from product or product specification to fulfillment pattern might be simpler; or, there might be cases where some order line items have no product specification, in which case the product specification can be derived from the context of the order item.

You typically create conceptual model products in your OSM system by importing them. (See *OSM Concepts* for more information.) When you import products, Design Studio creates the **productClassMapping.xml** and **productSpecMapping.xml** files. These files contain mappings between conceptual model products and OSM product specifications and fulfillment patterns. The **productClassMapping.xml** file is provided for backward compatibility, so in this topic it is assumed that you are using the **productSpecMapping.xml** file. These files are created in one of the following directories:

- If you have specified a value for the **Product Specification Mapping Folder** field of the Orchestration Preferences in Eclipse, it will create the two files in the directory specified.
- If no value is specified for that field, OSM will create the **productClassMapping.xml** file in the **resources/productClassMapping** directory and the **productSpecMapping.xml** file in the **resources/productSpecMapping** directory.

You can retrieve this mapping data from one of these files by creating a data instance provider that can be referenced from an XQuery expression body using a data instance behavior.

Note

The element names are not the same between the **productClassMapping.xml** and **productSpecMapping.xml** files. Ensure that you are using the correct element names for the file you are referencing. The names in this topic are correct for the **productSpecMapping.xml** file.

For example, the following XQuery creates the **\$productSpecMap** variable that references the data instance that points to the **productSpecMapping.xml** file:

```
let $productSpecMap := vf:instance('dataInstace1')
```

The following code creates a variable that references the product specification value from the customer order. For example:

```
let $productSpecName := fn:normalize-space(im:itemReference/  
im:primaryClassificationCode/text())
```

You can now create an expression that matches the product specification from the order with the product specification contained in the **productSpecMapping.xml** file and returns the fulfillment pattern associated with it or else defaults to the Non.Service.Offer fulfillment pattern. For example:

```
return  
if ($productSpecName != '')  
then  
  fn:normalize-space($productSpecMap/productSpec  
    [fn:lower-case(@name)=fn:lower-case($productSpecName)]/productSpec/text())  
else  
  'Non.Service.Offer'
```

In the following example, OSM retrieves the product specification Mobile Service Feature Class from the incoming customer order. OSM uses the order item property specified in the **Fulfillment Pattern Mapping Property** field for the order item to map the product specification to a fulfillment pattern.

The order item property specified in the **Fulfillment Pattern Mapping Property** field for the order item includes the following XQuery expression:

```
declare namespace im="http://xmlns.oracle.com/InputMessage";
(: Use the ProductSpecMap data instance behavior to retrieve the data in the
productSpecMapping.xml file: :)
let $productSpecMap := vf:instance('ProductSpecMap')
let $productSpecName :=
  fn:normalize-space(im:itemReference/im:primaryClassificationCode/text())
return
  if ($productSpecName != '')
  then
    fn:normalize-space($productSpecMap/productSpec
      [fn:lower-case(@name)=fn:lower-case($productSpecName)]/productClass/text())
  else
    'Non.Service.Offer'
```

The **productSpecMapping.xml** file includes the **<productSpec>** element, that maps the Mobile Service Feature Class product specification to the Service.Mobile fulfillment pattern:

```
<productSpec name="Mobile Service Feature Class"
  cartridge="OsmCentralOMExample-ProductSpecs">
  <fulfillmentPattern>Service.Mobile</fulfillmentPattern>
</productSpec>
```

To summarize, to map an order line item in an incoming customer order to a fulfillment pattern, you configure the following:

- In the order item specification:
 - A property that retrieves the conceptual model product or the OSM product specification from the incoming customer order.
 - The order item property specified in the **Fulfillment Pattern Mapping Property** field, that maps the product or product specification to the fulfillment pattern. To do so, OSM uses the ProductClassMap data instance behavior.
- The ProductSpecMap data instance behavior (and the data provider that supports it), that retrieves data from the **productSpecMapping.xml** file.
- The **productSpecMapping.xml** file used by the ProductClassMap data instance behavior, that maps products and product specifications to fulfillment patterns.

When you update your product catalog, you might need to add new fulfillment patterns. In that case, you need to:

- Create new fulfillment patterns and conceptual model products, if necessary.
- Add mappings to the **productSpecMapping.xml** file.

You do not need to change the order item specification or the data instance behavior.

About Order Item Specification Order Item Hierarchy XQuery Expressions

This topic describes how to use the Order Item Specification editor **Order Item Hierarchies** tab, Key Expression and Parent Key Expression areas, **XQuery** subtabs to write expressions that specify the relative hierarchy of order items, in the same order or between different orders,

based on an order item value, such as **lineId** and **parentLineId** and has the following characteristics:

- Context: The Key Expression and Parent Key Expression area XQuery input document is the order item. Specifically order item properties that indicate the relative hierarchy, such as order item **lineId** and **parentLineId** properties. For example:

```
<osm:orderItem
  xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model"
  id="1288881040699">
  ....
  <osm:properties
    xmlns:im="http://oracle.communications.ordermanagement.unsupported.
    centralom">
    <im:typeCode>PRODUCT</im:typeCode>
    <im:parentLineId>3</im:parentLineId>
    <im:requestedDeliveryDate>2013-06-31T12:00:00</im:requestedDeliveryDate>
    <im:lineItemName>Commercial Fixed Service [Add]</im:lineItemName>
    <im:lineId>4</im:lineId>
    <im:ServiceActionCode>UPDATE</im:ServiceActionCode>
    <im:productClass>Fixed Service Plan Class</im:productClass>
    <im:serviceId>552131313131</im:serviceId>
    <im:productSpec>Service.Fixed</im:productSpec>
    <im:lineItemPayload> [34 lines]
    <im:region>Sao Paulo</im:region>
  </osm:properties>
</osm:orderItem>
```

- Prolog: You can declare the order item specification namespace and the OSM namespace in the XQuery prolog. For example:

```
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace im="http://
oracle.communications.ordermanagement.unsupported.centralom";
```

You can declare the **OrchestrationXQueryFunctions** class in the prolog to use the **ancestors** method that returns the current node and all ancestors of the current node based on the specified hierarchy definition. This method can be useful when creating dependencies between order items based on hierarchy. For example:

```
declare namespace osmfn =
"java:oracle.communications.ordermanagement.orchestration.generation.OrchestrationXQueryFunctions";
```

See "[OSM XQuery Functions](#)" for more information about the **OrchestrationXQueryFunctions** class. See *OSM Concepts* for an example of how the **ancestors** method is used.

- Body: The XQuery body must specify an order item property defined in the order item specification.

For example, for the Key Expression, you can identify a unique key for each order item, typically the order item line ID:

```
fn:normalize-space(osm:properties/im:LineId/text())
```

For example, for the Parent Key Expression, you can identify a parent order line item, typically the line ID for the parent order line item:

```
fn:normalize-space(osm:properties/im:parentLineId/text())
```

In the following example, the key expression uses the parent order line item's **<lineId>** element from the order item property customer order:

```
declare namespace im="http://oracle.communications.ordermanagement.unsupported.centralom";
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
fn:normalize-space(osm:properties/im:LineId/text())
```

The parent key expression uses the child order line item's **<parentLineId>** element from the incoming customer order:

```
declare namespace im="http://oracle.communications.ordermanagement.unsupported.centralom";
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
fn:normalize-space(osm:properties/im:parentLineId/text())
```

About Order Item Specification Condition XQuery Expressions

This topic describes how to use the Order Item Specification editor **Orchestration Conditions** tab, Condition Expression area, **XQuery** subtab to write expressions that specifies an order item property value as a condition that you can then use in an order decomposition rule or in a fulfillment pattern to determine whether an order item gets included in an order component. The XQuery for the condition has the following characteristics:

- Context: The Condition Expression area XQuery input document is the order item properties you want to use as conditions. For example, the following order item contains the **region** and the **ServiceActionCode** order item properties, that could be associated with conditions:

```
<osm:orderItem
  xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model"
  id="1288881040699">
  ....
  <osm:properties
    xmlns:im="http://oracle.communications.ordermanagement.unsupported.
    centralom">
    <im:typeCode>PRODUCT</im:typeCode>
    <im:parentLineId>3</im:parentLineId>
    <im:requestedDeliveryDate>2013-06-31T12:00:00</im:requestedDeliveryDate>
    <im:lineItemName>Commercial Fixed Service [Add]</im:lineItemName>
    <im:lineId>4</im:lineId>
    <im:ServiceActionCode>UPDATE</im:ServiceActionCode>
    <im:productClass>Fixed Service Plan Class</im:productClass>
    <im:serviceId>552131313131</im:serviceId>
    <im:productSpec>Service.Fixed</im:productSpec>
    <im:lineItemPayload> [34 lines]
    <im:region>Sao Paulo</im:region>
  </osm:properties>
</osm:orderItem>
```

See "[About Fulfillment Pattern Order Component Condition XQuery Expressions](#)" for a description of the XQuery condition based on the **ServiceActionCode**. See "[About Decomposition Rule Condition XQuery Expressions](#)" for a description of the XQuery condition based on the **region**.

- Prolog: You can declare the order item specification namespace and the OSM namespace in the XQuery prolog. For example:

```
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace im="http://
oracle.communications.ordermanagement.unsupported.centralom";
```

- Body: The XQuery body must evaluate an order item property defined in the order item specification. These order item properties are available from the OSM namespace using the properties element. For example, the following expression evaluates to true if the value of **region** is anything other than **Sao Paulo** and the order item gets included in the order

component. If the **region** were set to **Sao Paulo**, then the order item would not be included in the order component.

```
fn:not(fn:normalize-space(osm:properties/im:region/text()='Sao Paulo'))
```

Another condition could be created that would only evaluate to true if the value of **region** was set to **Sao Paulo**. In this case, the order item would only be included in the order component if the **region** were set to **Sao Paulo**.

About Fulfillment Pattern Order Component XQuery Expressions

The Fulfillment Pattern editor provides the following areas to define XQuery expressions related to order decomposition:

- [About Fulfillment Pattern Order Component Condition XQuery Expressions](#)
- [About Associating Order Items Using Property Correlations XQuery Expressions](#)

Note

The XQuery expressions discussed in this chapter also apply to the Orchestration Dependency editor.

About Fulfillment Pattern Order Component Condition XQuery Expressions

This topic describes how to use the Fulfillment Pattern editor, **Orchestration Plan** tab, **Order Components** subtab, **Conditions** subtab **XQuery** subtab to write an expression that specifies whether to include or exclude an order item from an order component. You can create a new fulfillment pattern from the Fulfillment Pattern editor or select from conditions created in the Order Item Specification. See "[About Order Item Specification Condition XQuery Expressions](#)" for more information about the context, prolog, and body of condition XQuery expressions.

The following example XQuery expression only evaluates to true if the value of **ServiceActionCode** is not NONE or UPDATE. For example, if the value of **ServiceActionCode** were ADD, then the order item would be included in the order component.

```
fn:boolean
(
  (osm:properties/im:ServiceActionCode/text()!="NONE" and
  osm:properties/im:ServiceActionCode/text()!="UPDATE") or
  (
```

About Associating Order Items Using Property Correlations XQuery Expressions

This topic describes how to use the Fulfillment Pattern editor, **Orchestration Plan** tab, **Order Components** subtab, **Order Item Association** subtab, Property Correlation selection, **XQuery** subtab to write an expression that associates order items to order components that are not assigned by their fulfillment pattern. These order item associations are typically required when external systems need a specific context for an order item and includes the following characteristics:

- Context: The **Order Item Association** subtab XQuery input documents are multiple order items in the order after decomposition contained in the **fromOrderComponent** element and the entire set of order items included in the order contained in the **toOrderComponent** element. You can make an XQuery association based on the

contents of these order items that create an association between the unique order item IDs. For example:

```
<fromOrderComponent xmlns="">
  <osm:orderItem
    xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model"
    id="1234">
    <osm:name>Speed By Demand [Add]</osm:name>
    ....
    <osm:properties
      xmlns:im="http://oracle.communications.ordermanagement.unsupported.
      centralom">
      <im:typeCode>PRODUCT</im:typeCode>
      <im:parentLineId>3</im:parentLineId>
      <im:requestedDeliveryDate>2013-06-31T12:00:00
      </im:requestedDeliveryDate>
      <im:lineItemName>Commercial Fixed Service [Add]</im:lineItemName>
      <im:lineId>4</im:lineId>
      <im:SiteID>10</im:SiteID>
      <im:ServiceActionCode>UPDATE</im:ServiceActionCode>
      <im:productClass>Speed by Demand class</im:productClass>
      <im:serviceId>552131313131</im:serviceId>
      <im:productSpec>Service.Fixed</im:productSpec>
      <im:lineItemPayload> [34 lines]
      <im:region>Sao Paulo</im:region>
    </osm:properties>
  </osm:orderItem>
  <osm:orderItem [37 lines]
  <osm:orderItem [42 lines]
  <osm:orderItem [57 lines]
  <osm:orderItem [57 lines]
</fromOrderComponent>
<toOrderComponent xmlns="">
  <osm:orderItem [35 lines]
  <osm:orderItem [37 lines]
  <osm:orderItem [42 lines]
  <osm:orderItem
    xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model"
    id="5678">
    <osm:name>Broadband Bundle [Add]</osm:name>
    ....
    <osm:properties
      xmlns:im="http://oracle.communications.ordermanagement.unsupported.
      centralom">
      <im:typeCode>PRODUCT</im:typeCode>
      <im:parentLineId>3</im:parentLineId>
      <im:requestedDeliveryDate>2013-06-31T12:00:00
      </im:requestedDeliveryDate>
      <im:lineItemName>Broadband Bundle [Add]</im:lineItemName>
      <im:lineId>4</im:lineId>
      <im:SiteID>10</im:SiteID>
      <im:ServiceActionCode>UPDATE</im:ServiceActionCode>
      <im:productClass>Broadband Bundle Class</im:productClass>
      <im:serviceId>1112223333</im:serviceId>
      <im:productSpec>Broadband.Bundle</im:productSpec>
      <im:lineItemPayload> [34 lines]
      <im:region>Sao Paulo</im:region>
    </osm:properties>
  </osm:orderItem>
  <osm:orderItem [57 lines]
  <osm:orderItem [57 lines]
  <osm:orderItem [42 lines]
```

```

<osm:orderItem [37 lines]
<osm:orderItem [37 lines]
<osm:orderItem [57 lines]
</toOrderComponent>

```

- **Prolog:** You can declare the order item namespace and the OSM namespace in the XQuery prolog. For example:

```

declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace im="http://
oracle.communications.ordermanagement.unsupported.centralom";

```

- **Body:** The XQuery body must specify a dependency between the order item and the associated order item using something similar to the following syntax:

```

let $fromItem := osm:fromOrderComponent/osm:orderItem[osm:name/text()='Speed By
Demand [Add]]
let $toItem := osm:toOrderComponent/osm:orderItem[osm:name/text()='Broadband Bundle
[Add]' and osm:properties/im:SiteID/text() = $fromItem/osm:properties/im:SiteID/
text()]
return
<osm:dependency fromOrderItemId='{ $fromItem/@id}' toOrderItemId='{ $toItem/@id}' />

```

where

- **osm:fromOrderComponent:** Returns the set of order items included in the order component after the decomposition phase.
- **osm:toOrderComponent:** Returns the entire set of order items included in the order.
- **osm:orderItem:** These are the order items in the **fromOrderComponent** or **toOrderComponent** categories.
- **osm:dependency fromOrderItemId='{ \$fromItem/@id}':** The output of the XQuery specifying the source order item ID for the association.
- **toItem='{ \$childOrderItem/@id}' />:** The output of the XQuery specifying the target order item ID for the association.

Given the sample provided in the context bullet, this XQuery would return the following association:

```

<osm:dependency fromOrderItemId='1234' toOrderItemId='5678' />

```

The following example shows an XQuery that associates all child order items with their parent items. (See *OSM Concepts* for more information.) The output of the XQuery expression returns a node-set of `<osm:dependency fromOrderItemId='{ $fromOrderItem/@id}' toOrderItemId='{ $toOrderItem/@id}' />` where item IDs are the `@id` attribute of the order item.

```

declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace prop="http://oracle.communications.ordermanagement.unsupported.centralom";
(: $fromOrderItemList contains all order items in the selected order component: :)
for $fromOrderItem in $fromOrderItemList
let $fromOrderItemList := osm:fromOrderComponent/osm:orderItem
(: $childOrderItems contains all children for the current $fromOrderItem: :)
let $childOrderItems := osm:toOrderComponent/osm:orderItem/osm:properties
[prop:ParentLineID/text() = $fromOrderItem/osm:properties/prop:LineID/text()]
(: Returns the association between all parents and their children: :)
for $childOrderItem in $childOrderItems
return
<osm:dependency fromOrderItemId='{ $fromOrderItem/@id}' toOrderItemId='{ $childOrderItem/@id}' />

```

About Decomposition Rule Condition XQuery Expressions

This topic describes how to use the Decomposition Rule editor, **Conditions** tab, **Conditions Details** subtab, **XQuery** subtab to write an expression that associates a condition with a decomposition rule. You can create the condition in the order item specification and select them in the decomposition rule, or you can create them directly in the decomposition rule. See ["About Order Item Specification Condition XQuery Expressions"](#) for more information about the context, prolog, and body of condition XQuery expressions.

The following is an example of two decomposition rules, each having a condition set that determines whether an order item is included in the target order component or not. In this example:

- The decomposition rule that targets the target system order component for region 1 has the following decomposition condition:
`isRegion1`
- The decomposition rule that targets the a target system order component for region 2 has the following decomposition condition:
`isOtherRegion`

The XQuery for the **isRegion1** decomposition rule condition is:

```
declare namespace im="http://oracle.communications.ordermanagement.unsupported.centralom";
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
fn:normalize-space(osm:properties/im:region/text()='Toronto')
```

This condition specifies the value of the **region** order item property. If the value is **Toronto**, the decomposition rule condition is true, and the order item is included in the region 1 target system order component.

The XQuery for the **isOtherRegion** decomposition rule condition is:

```
declare namespace im="http://
oracle.communications.ordermanagement.unsupported.centralom";declare namespace
osm="http://xmlns.oracle.com/communications/ordermanagement/model";fn:not(fn:normalize-
space(osm:properties/im:region/text()='Toronto'))
```

This condition also specifies the value of the **region** order item property, but evaluates to true only if the value is not **Toronto**. All order items that have any other value are included in the region 2 target system order component.

The following example includes a variation on the **isRegion1** decomposition rule that specifies that all the order items from the source order component to the target order component that have at least one order item with a **region** property of **Toronto** are included in the order component. Otherwise, if the condition evaluates to false then none of the order items in **fromOrderComponent** are included in the resulting order component.

```
declare namespace im="http://oracle.communications.centralom";
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
fn:exists(osm:fromOrderComponent/osm:orderItem[fn:normalize-space(osm:properties/
im:Region/text()='Toronto']])
```

For some functions, there is only one target system in the topology. For example, if you have only one collections system in the topology, you will have one dependency rule that uses a simple mapping from the source collections function order component to the collections target system order component, and no decomposition condition is necessary.

About Component Specification Custom Component ID XQuery Expressions

This topic describes how to use the Order Component Specification editor, **Component ID** tab, Component ID area, **XQuery** subtab to write an expression that specifies a custom component ID for an order component. These custom component IDs are typically required when the default component IDs are not sufficiently specific (see *OSM Concepts* for more information about the default component ID). The Component ID XQuery includes the following characteristics:

- Context: The **Component ID** tab XQuery input document is the order item and the order item properties you want to use to create a custom component ID with. For example, the following order item contains the **SiteID** and **requestedDeliveryDate** order item properties. In a simple scenario, you can use this element to group all order items that share the same **SiteID** value and further delineate groups based on **requestedDeliveryDate** date range.

```
<osm:orderItem
  xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model"
  id="1288881040699">
  ....
  <osm:properties
    xmlns:im="http://oracle.communications.ordermanagement.unsupported.
    centralom">
    <im:typeCode>Bundle</im:typeCode>
    <im:parentLineId>3</im:parentLineId>
    <im:requestedDeliveryDate>2013-06-31T12:00:00</im:requestedDeliveryDate>
    <im:lineItemName>Commercial Fixed Service [Add]</im:lineItemName>
    <im:lineId>4</im:lineId>
    <im:SiteID>10</im:SiteID>
    <im:ServiceActionCode>UPDATE</im:ServiceActionCode>
    <im:productClass>Fixed Service Plan Class</im:productClass>
    <im:serviceId>552131313131</im:serviceId>
    <im:productSpec>Service.Fixed</im:productSpec>
    <im:lineItemPayload> [34 lines]
    <im:region>Sao Paulo</im:region>
  </osm:properties>
</osm:orderItem>
```

- Prolog: You can declare the order item namespace and the OSM namespace in the XQuery prolog. In more complicated XQuery expressions, you can also use the **OrchestrationXQueryFunctions** OSM Java package to specify component IDs based on order item hierarchies, order item requested delivery date, order component duration, order component minimum duration separation, or a combination of some or all of them. For example:

```
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace im="http://
oracle.communications.ordermanagement.unsupported.centralom";
declare namespace osmfn =
"java:oracle.communications.ordermanagement.orchestration.generation.OrchestrationXQueryFunctions";
```

See "[OSM XQuery Functions](#)" for more information about the **OrchestrationXQueryFunctions** class.

- Body: The body must return a string. Every order item that ends with the same string gets included in the order component. For example, if you wanted to group all order items based on the **SiteID** value, you could specify the following XQuery:

```
return osm:properties/im:SiteID/text()
```

The following topics describe OSM Java package methods.

For more information about how the `OrchestrationXQueryFunctions` are used in custom Component ID XQuery expressions and for more complicated custom group ID generation scenarios that use `OrchestrationXQueryFunction`, see the following topics:

- [Custom Order Component IDs Based on Hierarchy](#)
- [Custom Component IDs Based on Requested Delivery Date and Duration](#)
- [Custom Component IDs by Duration and Minimum Separation Duration](#)
- [Combining Order Item Hierarchy with Duration-Based Groupings](#)

Custom Order Component IDs Based on Hierarchy

A more common scenario where custom order component IDs can be used is when you need additional groupings of order components at the granularity level. For example, three levels of decomposition from Function, System, to Bundle, results in the following component IDs:

- `BillingFunction`
- `BillingFunction.BillingSystem`
- `BillingFunction.BillingSystem.Bundle`

If you had order items in the Bundle order components that were part of different bundles that go to different the billing system, you would need to separate each order item bundle into different bundle order component. A component ID for such a scenario could look like this:

- For billing system 1: `BillingFunction.BillingSystem.Bundle.2/BundleGranularity`
- For billing system 2: `BillingFunction.BillingSystem.Bundle.6/BundleGranularity`

To create custom component IDs for this scenario, you could use the following order item properties:

- **typeCode:** This property specifies if the order line item is an offer, bundle, or product. This element also defines the product hierarchy of the order line items. For example:

```
OFFER
  BUNDLE
    PRODUCT
```

- **linelid** and **parentLinelid:** These properties specify the hierarchical relationship between the bundle and product order line items. You can create separate component IDs for every order item bundle and associate all product order items with their corresponding bundle component ID. To identify all ancestor order items that may be a bundle, you can use the XQuery **ancestors** function, as explained later.

For example, the following four order items include two bundles and two associated products. These order items have the following characteristics:

- **Order Item 1** includes:
 - **typeCode:** BUNDLE
 - **linelid:** 2
 - **parentLinelid:** 1 (for example, an order item with an OFFER **typeCode**. This order item is not specified in this example).

```
<osm:orderItem
  xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model"
```

```

id="1234">
  <osm:name>FIXED BUNDLE - BUNDLE</osm:name>
  ....
  <osm:properties
    xmlns:im="http://oracle.communications.ordermanagement.unsupported.
    centralom">
    <im:typeCode>BUNDLE</im:typeCode>
    <im:parentLineId>1</im:parentLineId>
    <im:requestedDeliveryDate>2013-06-31T12:00:00</im:requestedDeliveryDate>
    <im:lineItemName>Fixed Bundle</im:lineItemName>
    <im:lineId>2</im:lineId>
    <im:SiteID>5</im:SiteID>
    <im:ServiceActionCode>UPDATE</im:ServiceActionCode>
    <im:productClass>Fixed Bundle Class</im:productClass>
    <im:serviceId>552131313131</im:serviceId>
    <im:productSpec>Service.Fixed</im:productSpec>
    <im:lineItemPayload> [34 lines]
    <im:region>Sao Paulo</im:region>
  </osm:properties>
</osm:orderItem>

```

- Order Item 2 includes:
 - typeCode:** PRODUCT
 - lineId:** 3
 - parentLineId:** 2 (This matches the **lineId** of order item 1 indicating that order item 1 is the parent of order item 2).

```

<osm:orderItem
  xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model"
  id="56789">
  <osm:name>FIXED CALLER ID - PRODUCT</osm:name>
  ....
  <osm:properties
    xmlns:im="http://oracle.communications.ordermanagement.unsupported.
    centralom">
    <im:typeCode>PRODUCT</im:typeCode>
    <im:parentLineId>2</im:parentLineId>
    <im:requestedDeliveryDate>2013-06-31T12:00:00</im:requestedDeliveryDate>
    <im:lineItemName>Commercial Fixed Service [Add]</im:lineItemName>
    <im:lineId>5</im:lineId>
    <im:SiteID>7</im:SiteID>
    <im:ServiceActionCode>UPDATE</im:ServiceActionCode>
    <im:productClass>Fixed Bundle Class</im:productClass>
    <im:serviceId>552131313131</im:serviceId>
    <im:productSpec>Service.Fixed</im:productSpec>
    <im:lineItemPayload> [34 lines]
    <im:region>Sao Paulo</im:region>
  </osm:properties>
</osm:orderItem>

```

- Order Item 3 includes:
 - typeCode:** BUNDLE
 - lineId:** 6
 - parentLineId:** 1 (This indicates that both order item 1 and order item 3 share the same parent).

```

<osm:orderItem
  xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model"
  id="10111213">

```

```

<osm:name>BroadBand BUNDLE - BUNDLE</osm:name>
.....
<osm:properties
  xmlns:im="http://oracle.communications.ordermanagement.unsupported.
  centralom">
  <im:typeCode>BUNDLE</im:typeCode>
  <im:parentLineId>1</im:parentLineId>
  <im:requestedDeliveryDate>2013-06-31T12:00:00</im:requestedDeliveryDate>
  <im:lineItemName>Broadband Bundle</im:lineItemName>
  <im:lineId>6</im:lineId>
  <im:SiteID>5</im:SiteID>
  <im:ServiceActionCode>UPDATE</im:ServiceActionCode>
  <im:productClass>Broadband Bundle Class</im:productClass>
  <im:serviceId>552131313131</im:serviceId>
  <im:productSpec>Service.Broadband</im:productSpec>
  <im:lineItemPayload> [34 lines]
  <im:region>Sao Paulo</im:region>
</osm:properties>
</osm:orderItem>

```

- Order Item 4 includes:
 - **typeCode**: PRODUCT
 - **lineId**: 7
 - **parentLineId**: 6 (This matches the **lineId** of order item 3 indicating that order item 3 is the parent of order item 4).

```

<osm:orderItem
  xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model"
  id="14151617">
  <osm:name>BroadBand Service - PRODUCT</osm:name>
  .....
  <osm:properties
    xmlns:im="http://oracle.communications.ordermanagement.unsupported.
    centralom">
    <im:typeCode>PRODUCT</im:typeCode>
    <im:parentLineId>6</im:parentLineId>
    <im:requestedDeliveryDate>2013-06-31T12:00:00</im:requestedDeliveryDate>
    <im:lineItemName>Fixed Bundle</im:lineItemName>
    <im:lineId>7</im:lineId>
    <im:SiteID>5</im:SiteID>
    <im:ServiceActionCode>UPDATE</im:ServiceActionCode>
    <im:productClass>Broadband Bundle Class</im:productClass>
    <im:serviceId>552131313131</im:serviceId>
    <im:productSpec>Service.Broadband</im:productSpec>
    <im:lineItemPayload> [34 lines]
    <im:region>Sao Paulo</im:region>
  </osm:properties>
</osm:orderItem>

```

The customer order includes two bundles with two products. The hierarchy is:

```

Fixed Bundle - order item 2
  Fixed Caller ID - order item 5
Broadband Bundle - order item 6
  BroadBand Service - order item 7

```

To create the separate customized component IDs for the bundle order items 1 and 3, and include all their corresponding children order items you need to:

- Return a separate component ID for each BUNDLE **typeCode**. This causes BUNDLE order components to be generated.

- Ensure that the PRODUCT **typeCode** for that bundle are included in its parent order item.

To do so, the XQuery uses the **ancestors** function to find whether the order item has a BUNDLE **typeCode** or has a BUNDLE **typeCode** in one of its parent order items. If the order item is a bundle, then a OSM creates a component ID for the bundle. If the order item has a bundle in one of its parent order items, then OSM includes the order item in its parent order item component ID. The following example shows an XQuery that does this.

```
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace prop="http://
oracle.communications.ordermanagement.unsupported.centralom";
declare namespace osmfn =
"java:oracle.communications.ordermanagement.orchestration.generation.OrchestrationXQueryF
unctions";
(: The following part of the XQuery identifies the order line hierarchy definition
and retrieve all of the predecessor order line items in the bundle: :)
let $ancestors := osmfn:ancestors("CustomerOrderItemSpecification","default","http://
oracle.communications.ordermanagement.unsupported.centralom")

(: The following part of the XQuery finds the BUNDLE order item and generates an ID
based on the bundle order item lineID: :)
return
  if (fn:exists($ancestors[osm:properties/prop:typeCode='BUNDLE']))
  then (
    concat($ancestors[osm:properties/prop:typeCode=('BUNDLE')]
    [1]/osm:properties/prop:lineId/text(),'/BundleGranularity')
  )
  else (
    'ALL_OFFERS_AND_NON_SERVICE_BILLING/BundleGranularity'
  )
)
```

This XQuery finds the child order line items, finds their parent order line items, and creates a bundle order component for each of the bundle lines. The component IDs are:

- BillingFunction.BillingSystem.Bundle.2/BundleGranularity
- BillingFunction.BillingSystem.Bundle.6/BundleGranularity

In another example, there is one offer with two bundles and two products in each bundle. The following table shows the hierarchy of bundles and products. The component IDs use the line IDs of the two bundle items.

Line Number	Line Name	Line typeCode	Parent Line ID	Value to Use in Component ID
1	Triple Play	OFFER	-	-
2	Fixed Bundle	BUNDLE	1	2
2.1	Fixed Service	PRODUCT	2	2
2.2	Call Forwarding	PRODUCT	2	2
5	Broadband Bundle	BUNDLE	1	5
5.1	Broadband Service	PRODUCT	5	5
5.2	High-Speed Internet	PRODUCT	5	5

Custom Component IDs Based on Requested Delivery Date and Duration

In some scenarios, you may want to create custom Order Component IDs based on order item requested delivery date and duration. For example, the following custom component ID XQuery creates order component grouping based on the order item requested delivery dates:

```

declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace prop="http://
oracle.communications.ordermanagement.unsupported.centralom";
declare namespace osmfn =
"java:oracle.communications.ordermanagement.orchestration.generation.OrchestrationXQueryF
unctions";
let $groupDuration := "P2D"
return
osmfn: getGroupIdByDateTime ($groupDuration)

```

The XQuery creates a new order component for an order item based on the order item's requested delivery date and includes all order items within this group that fall within two days of the first order item's requested delivery date in the group. The XQuery does the same thing for all other order items within the order.

The following table shows how five order items would be grouped given a custom Order Component ID XQuery that creates a new component IDs.

Note

The group ID names are static with the first order component always called Group1 and the next Group2, and so on.

Order Item	Requested Delivery Date	Group ID
A	June 9, 2014	Group1
B	June 10, 2014	Group1
C	June 11, 2014	Group2
D	June 12, 2014	Group2
E	June 12, 2014	Group3

See "[About Component Specification Custom Component ID XQuery Expressions](#)" for more information about the context, prolog, and body of this XQuery. See "[OSM XQuery Functions](#)" for more information about the **OrchestrationXQueryFunctions** class.

Custom Component IDs by Duration and Minimum Separation Duration

You can specify a minimum duration separation value for order items that fall very close to a custom Order ID grouping based on order item requested delivery date and duration. For example, the following XQuery adds a minimum separation value of one day:

```

declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace prop="http://
oracle.communications.ordermanagement.unsupported.centralom";
declare namespace osmfn =
"java:oracle.communications.ordermanagement.orchestration.generation.OrchestrationXQueryF
unctions";
let $groupDuration := "P2D"
let $minSeparationDuration := "P1D"
return
osmfn: getGroupIdByDateTime ($groupDuration, $minSeparationDuration)

```

All order item requested delivery dates that fall within one day of a two day grouping, would be included in the two day grouping.

The following table shows how the five order items would be grouped given a one day minimum separation duration.

Order Item	Requested Delivery Date	Group ID
A	June 9, 2014	Group1
B	June 10, 2014	Group1
C	June 11, 2014	Group1
D	June 12, 2014	Group2
E	June 12, 2014	Group2

See ["About Component Specification Custom Component ID XQuery Expressions"](#) for more information about the context, prolog, and body of this XQuery. See ["OSM XQuery Functions"](#) for more information about the **OrchestrationXQueryFunctions** class.

Combining Order Item Hierarchy with Duration-Based Groupings

You can combine the function to create custom Component IDs based on order item requested delivery date, duration, and minimum duration separation, or a combination of these functions with order component ID generation based on order item hierarchy. The following example creates separate component IDs for order items that, although they have the same requested delivery date, are part of different order item hierarchical groupings:

```
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace prop="http://
oracle.communications.ordermanagement.unsupported.centralom";
declare namespace osmfn =
"java:oracle.communications.ordermanagement.orchestration.generation.OrchestrationXQueryF
unctions";
let $groupDuration := "P2D"
let $minSeparationDuration := "P1D"
return
osmfn: getGroupIdByDateTime ($groupDuration, $minSeparationDuration)
let $rootAncestorID := osmfn:ancestors("eboLineItem", "default", "http://
xmlns.oracle.com/communications/ordermanagement")[fn:last()]/osm:properties/
prop:BaseLineId/text()
return fn:concat($rootAncestorId, '/', $groupId)
```

The following table shows how five hierarchically divided order items would be grouped given a one day minimum separation duration.

Order Item	Requested Delivery Date	Group ID	Component ID
A.1	June 9, 2014	Group1	A/Group1
A.1.1	June 11, 2014	Group1	A/Group1
A1.2	June 19, 2014	Group2	A/Group2
A.1.3	June 20, 2014	Group2	A/Group2
B.1	June 9, 2014	Group1	B/Group1
B.1.1	June 11, 2014	Group1	B/Group1
B.1.2	June 12, 2014	Group1	B/Group2

See ["About Component Specification Custom Component ID XQuery Expressions"](#) for more information about the context, prolog, and body of this XQuery. See ["OSM XQuery Functions"](#) for more information about the **OrchestrationXQueryFunctions** class.

About Component Specification Duration XQuery Expressions

This topic applies to the Order Component editor, **Duration** tab, Duration Expression area, **XQuery** subtab.

- Context: There is no input document for this expression.
- Prolog: There is no prolog required for this expression.
- Body: The XQuery body returns a duration value based on the XQuery you enter:

`PyYmMdDTThmMsS`

where

- **P** begins the expression.
- **yY** specifies the year.
- **mM** specifies the month.
- **dD** specifies the day.
- **T** separates the parts of the expression indicating the date from the parts of the expression indicating the time.
- **hH** specifies the hour.
- **mM** specifies the minutes.
- **sS** specifies the seconds.

The following example is a hard-coded duration expression for seven hours:

`PT7H0M0S`

For more information about how OSM uses these fields to calculate order component durations, see *OSM Concepts*.

About Fulfillment Pattern Duration XQuery Expressions

This topic applies to the Fulfillment Pattern editor, **Orchestration Plan** tab, **Duration** subtab, Duration Expression area, **XQuery** subtab. The functionality for this tab has been deprecated and is displayed to provide backward compatibility with older cartridges.

For the recommended functionality for configuring order component durations, see "[About Fulfillment Pattern Component Duration XQuery Expressions](#)" and "[About Component Specification Duration XQuery Expressions](#)".

About Fulfillment Pattern Component Duration XQuery Expressions

This topic applies to the Fulfillment Pattern editor, **Orchestration Plan** tab, **Order Components** subtab, **Duration** subtab, Duration Expression area, **XQuery** subtab.

- Context: There is no input document for this expression.
- Prolog: There is no prolog required for this expression.
- Body: The XQuery body returns a duration value based on the XQuery you enter:

`PyYmMdDTThmMsS`

where

- **P** begins the expression.
- **yY** specifies the year.
- **mM** specifies the month.
- **dD** specifies the day.
- **T** separates the parts of the expression indicating the date from the parts of the expression indicating the time.
- **hH** specifies the hour.
- **mM** specifies the minutes.
- **sS** specifies the seconds.

The following example is a hard-coded duration expression for three hours:

```
PT3H0M0S
```

For more information about how OSM uses these fields to calculate order component durations, see *OSM Concepts*.

Dependency XQuery Expressions

This topic includes information about Orchestration XQuery expressions related to orchestration dependencies:

- [About Order Item Dependency Property Correlation XQuery Expressions](#)
- [About Wait Delay Duration XQuery Expressions](#)
- [About Wait Delay Date and Time XQuery Expressions](#)
- [About Order Data Change Wait Condition XQuery Expressions](#)
- [About Order Item Inter-Order Dependency XQuery Expressions](#)

About Order Item Dependency Property Correlation XQuery Expressions

This topic describes how to use one of the following fields:

- Fulfillment Pattern editor, **Orchestration Plan** tab, **Dependencies** tab, **Order Item Dependency** subtab, **XQuery** subtab for the **Property Correlation** selection
- Orchestration Dependency editor, **Order Item Dependencies** tab, **XQuery** subtab for the **Property Correlation** selection

to write an expression that specifies dependencies between different order items using order item properties. The Property Correlation XQuery has the same context, prolog, and body structure as the Fulfillment Pattern editor, **Order Components** tab, **Order Item Association** subtab, **XQuery** subtab. See "[About Associating Order Items Using Property Correlations XQuery Expressions](#)" for more information.

The following example shows a dependency that requires provisioning of an Internet service before shipping a modem. This involves two order items: provision Internet service and ship modem. The correlating property is the order item ID.

```
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace im="http://sample.broadband";
let $bbProvision := osm:fromOrderComponent/osm:orderItem[osm:name="Internet Service"]
let $bbModem := osm:toOrderComponent/osm:orderItem[osm:name/text()='Broadband Modem'
and osm:properties/im:SiteID/text() = $bbProvision/osm:properties/im:SiteID/text()]
```

```
return
  <osm:dependency fromOrderItemId='{ $bbProvision/@id }' toOrderItemId='{ $bbModem/@id }' />
```

In this example:

- **\$bbProvision** contains the broadband service order item in the blocking Provision order component.
- **\$bbModem** is the broadband modem in the waiting Ship order component.
- The XQuery returns a dependency from the Internet Service order item to its associated Broadband Modem order item, identified by **\$bbProvision/@id** and **\$bbModem/@id**.

If the order item IDs are:

- \$bbProvision/@id = 1301589468772
- \$bbModem/@id = 1301589468785

Then the XQuery returns the following:

```
<osm:dependency fromOrderItemId='1301589468772' toOrderItemId='1301589468785' />
```

About Wait Delay Duration XQuery Expressions

This topic describes how to use one of the following fields:

- Fulfillment Pattern editor, **Orchestration Plan** tab, **Dependencies** subtab, **Wait Condition** subtab, Wait Delay area, Duration Expression area **XQuery** subtab for the **Duration** selection
- Orchestration Dependency editor, **Wait for Condition** tab, Wait Delay area, Duration Expression area **XQuery** subtab for the **Duration** selection

to write an expression that specifies the duration of delay, based on an order item property, before starting a waiting order component after all dependencies have been resolved.

- Context: The **Duration** XQuery input document is the entire set of order items included in the order contained in the **toOrderComponent** element. You can return the value of requestedDeliveryDate to help determine the wait delay duration. For example:

```
<toOrderComponent xmlns="">
  <osm:orderItem [35 lines]
  <osm:orderItem [37 lines]
  <osm:orderItem [42 lines]
  <osm:orderItem
    xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model" id="5678">
    <osm:name>Broadband Bundle [Add]</osm:name>
    ....
    <osm:properties xmlns:im="http://oracle.communications.ordermanagement.unsupported.centralom">
      <im:typeCode>PRODUCT</im:typeCode>
      <im:parentLineId>3</im:parentLineId>
      <im:requestedDeliveryDate>2013-06-31T12:00:00</im:requestedDeliveryDate>
      <im:lineItemName>Broadband Bundle [Add]</im:lineItemName>
      <im:lineId>4</im:lineId>
      <im:SiteID>10</im:SiteID>
      <im:ServiceActionCode>UPDATE</im:ServiceActionCode>
      <im:productClass>Broadband Bundle Class</im:productClass>
      <im:serviceId>1112223333</im:serviceId>
      <im:productSpec>Broadband.Bundle</im:productSpec>
      <im:lineItemPayload> [34 lines]
      <im:region>Sao Paulo</im:region>
    </osm:properties>
  </osm:orderItem [57 lines]
```

```
<osm:orderItem [57 lines]
<osm:orderItem [42 lines]
<osm:orderItem [37 lines]
<osm:orderItem [37 lines]
<osm:orderItem [57 lines]
</toOrderComponent>
```

- Prolog: You can declare the order item namespace and the OSM namespace in the XQuery prolog. For example:

```
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace im="http://
oracle.communications.ordermanagement.unsupported.centralom";
```

- Body: The XQuery body returns a duration value based on the requestedDeliveryDate order item property:

```
let $mydate := osm:toOrderComponent[1]/osm:orderItem[1]/osm:properties[1]/
*[namespace-uri()='http://
oracle.communications.ordermanagement.unsupported.centralom' and local-
name()='requestedDeliveryDate'] [1]/text()
return
if (fn:current-dateTime()- xs:dateTime($mydate) < xs:dayTimeDuration('PT10H')) then
  'PT10H'
else
  'PT10M'
return
```

where

- **osm:toOrderComponent**: Provides the entire set of order items included in the order.
- **osm:orderItem**: These are the order items in the **toOrderComponent** category. The remainder of this expression identifies the namespace of the order item specification and returns the value of the requestedDeliveryDate element.
- The **if** statement checks to see if the value of the requestedDeliveryDate is less than the hard-coded dayTimeDuration value. These values conform to the XSD duration data type.
- The **then** statement returns 10 hours if the **if** statement evaluates to true.
- The **else** statement return 10 months if the **if** statement evaluates to false.

The following example shows the sample XQuery to return a duration value.

```
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace im="http://oracle.communications.ordermanagement.unsupported.centralom";

let $mydate := osm:toOrderComponent[1]/osm:orderItem[1]/osm:properties[1]/*[namespace-uri()='http://
oracle.communications.ordermanagement.unsupported.centralom' and local-name()='requestedDeliveryDate']
[1]/text()
return
if (fn:current-dateTime()- xs:dateTime($mydate) < xs:dayTimeDuration('PT10H')) then
  'PT10H'
else
  'PT10M'
```

About Wait Delay Date and Time XQuery Expressions

This topic describes how to use one of the following fields:

- Fulfillment Pattern editor, **Orchestration Plan** tab, **Dependencies** subtab, **Wait Condition** subtab, Wait Delay area, Duration Expression area **XQuery** subtab for the **Date Time Expression** selection
- Orchestration Dependency editor, **Wait for Condition** tab, Wait Delay area, Duration Expression area **XQuery** subtab for the **Date Time Expression** selection

to write an expression that specifies the date and time, based on an order item property, for starting a waiting order component after all dependencies have been resolved.

- Context: The **Date Time Expression** XQuery input document is the entire set of order items included in the order contained in the **toOrderComponent** element. You can use the requestedDeliveryDate order item property to determine the date and time that the XQuery should start after all blocking items have resolved. For example:

```
<toOrderComponent xmlns="">
  <osm:orderItem [35 lines]
  <osm:orderItem [37 lines]
  <osm:orderItem [42 lines]
  <osm:orderItem
    xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model" id="5678">
    <osm:name>Broadband Bundle [Add]</osm:name>
    . . . . .
    <osm:properties xmlns:im="http://oracle.communications.ordermanagement.unsupported.centralom">
      <im:typeCode>PRODUCT</im:typeCode>
      <im:parentLineId>3</im:parentLineId>
      <im:requestedDeliveryDate>2013-06-31T12:00:00</im:requestedDeliveryDate>
      <im:lineItemName>Broadband Bundle [Add]</im:lineItemName>
      <im:lineId>4</im:lineId>
      <im:SiteID>10</im:SiteID>
      <im:ServiceActionCode>UPDATE</im:ServiceActionCode>
      <im:productClass>Broadband Bundle Class</im:productClass>
      <im:serviceId>1112223333</im:serviceId>
      <im:productSpec>Broadband.Bundle</im:productSpec>
      <im:lineItemPayload> [34 lines]
      <im:region>Sao Paulo</im:region>
    </osm:properties>
  </osm:orderItem [57 lines]
  <osm:orderItem [57 lines]
  <osm:orderItem [42 lines]
  <osm:orderItem [37 lines]
  <osm:orderItem [37 lines]
  <osm:orderItem [57 lines]
</toOrderComponent>
```

- Prolog: You can declare the order item namespace and the OSM namespace in the XQuery prolog. For example:

```
declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace im="http://
oracle.communications.ordermanagement.unsupported.centralom";
```

- Body: The XQuery body returns a date and time value based on the requestedDeliveryDate order item property:

```
osm:toOrderComponent[1]/osm:orderItem[1]/osm:properties[1]/*[namespace-uri()='http://
oracle.communications.ordermanagement.unsupported.centralom' and local-
name()='requestedDeliveryDate'][1]/text()
```

osm:toOrderComponent: returns the entire set of order items included in the order and returns the requested delivery date of all order items for the wait delay date and time.

The following example shows the sample XQuery to return a date time value.

```

declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace im="http://
oracle.communications.ordermanagement.unsupported.centralom";

osm:toOrderComponent[1]/osm:orderItem[1]/osm:properties[1]/*[namespace-uri()='http://
oracle.communications.ordermanagement.unsupported.centralom' and local-
name()='requestedDeliveryDate'][1]/text()

```

About Order Data Change Wait Condition XQuery Expressions

This topic describes how to use the Fulfillment Pattern editor, **Orchestration Plan** tab, **Dependencies** subtab, **Wait Condition** subtab, Wait for Condition area, **XQuery** subtab for the **Data Change Notification** selection,

This topic describes how to use one of the following fields:

- Fulfillment Pattern editor, **Orchestration Plan** tab, **Dependencies** subtab, **Wait Condition** subtab, Wait for Condition area, **XQuery** subtab for the **Data Change Notification** selection
- Orchestration Dependency editor, **Wait for Condition** tab, Wait for Condition area, **XQuery** subtab for the **Data Change Notification** selection

to write an expression that specifies a value that must exist in order item property (typically a blocking order item property) before a waiting order item starts.

- Context: The **Data Change Notification** XQuery input document is the task view task data that was changed using an update order transaction.
- Prolog: You can declare the \$blockingIndexes variable in the XQuery prolog that contains an index of data element for all blocking order items. For example:


```
declare variable $blockingIndexes as xs:integer* external;
```
- Body: The XQuery body returns a specific value and will wait until all blocking order items have the corresponding value and the XQuery returns true.

The following example shows the XQuery that evaluates the data change. The dependency is met when all blocking order items have reached a state of PROVISION STARTED.

```

(: The $blockingIndexes variable contains data element indexes for all blocking order items: :)
declare variable $blockingIndexes as xs:integer* external;
(: Specify "PROVISION STARTED" as the data value that must be met: :)
let $expectedMilestoneCode := "PROVISION STARTED"
(: $milestoneValues contains a set of milestones for all blocking order items: :)
let $milestoneValues :=
  /GetOrder.Response/_root/ControlData/Functions/ProvisioningFunction/orderItem/orderItemRef[
    fn:index-of($blockingIndexes, xs:integer(@referencedIndex)) !=
    0]/milestone[text() eq $expectedMilestoneCode]
(: Return true only if all the milestones in ProvisioningFunction/orderItem/orderItemRef are PROVISION
STARTED: :)
return fn:count($milestoneValues) eq fn:count($blockingIndexes)

```

The following example returns true when at least one blocking item is completed.

```

declare namespace oms="urn:com:metasolv:oms:xmlapi:1";
declare variable $blockingIndexes as xs:integer* external;
let $component := //ControlData/Functions/NetworkProvisionFunction
let $lineItem := $component/orderItem/orderItemRef[fn:index-of($blockingIndexes,
xs:integer(@referencedIndex)) != 0]
return
  if (fn:exists($lineItem))
  then

```

```

let $statusValue := $lineItem/OrderItemStatus/text() = "completed"
return
if (fn:count($statusValue)>0)
then
  fn:true()
else
  fn:false()
else
  fn:false()

```

About Order Item Inter-Order Dependency XQuery Expressions

This topic describes how to use the Order Item Specification editor, **Order Item Dependency** tab, Order Item Selector area, **XQuery** tab to write an expression that creates dependencies between the order items on the follow-on order and the order items on the base order. It is the follow-on order that generates this dependency on the base order.

- Context: The **Order Item Selector** XQuery input document is typically an order item on a follow-on order (the waiting order).
- Prolog: You can declare the OSM namespace, the cartridge namespace for the target order (the base order), and the namespace of the query task that contains the order data you want to view. For example:

```

declare namespace osm="http://xmlns.oracle.com/communications/ordermanagement/model";
declare namespace im="CommunicationsSalesOrderFulfillmentPIP";
declare namespace osmc="urn:oracle:names:ordermanagement:cartridge:
CommunicationsSalesOrderFulfillmentPIP:1.0.0:view:CommunicationsSalesOrderQueryTask";

```

- Body: The CRM that sends the follow-on order must specify the reference number that uniquely identifies the base order and also the order item line ID of the blocking order item. You can define a variable (such as **\$dependingLineId**) that extracts the dependent line ID from the order item context. For example:

```

let $dependingLineId := fn:normalize-space(osm:properties/
im:DependingSalesOrderBaseLineId)

```

You can configure a web service data instance provider that runs a FindOrder Web Service that searches for orders based on the reference value (**osm:properties/prop:Ref/text()**) in the follow-on order that generates a FindOrder response that includes the order ID of the base order. See *OSM Developer's Guide* for more information about configuring web service data instance providers. For example:

```

<ord:FindOrder xmlns:ord="http://xmlns.oracle.com/communications/ordermanagement"
xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model"
xmlns:prop="http://oracle.communications.ordermanagement.unsup.centralom">
  <ord:ViewBy>
    <ord:AmendmentFilter>
      <ord:LevelOfDetail>AmendmentsSummary</ord:LevelOfDetail>
    </ord:AmendmentFilter>
    <ord:LifecycleEventFilter>
      <ord:RetrieveLifecycleEvents>false</ord:RetrieveLifecycleEvents>
    </ord:LifecycleEventFilter>
  </ord:ViewBy>
  <ord:SelectBy>
    <ord:Reference>{fn:normalize-space(osm:properties/prop:Ref/text())} </ord:Reference>
  </ord:SelectBy>
</ord:FindOrder>

```

This data instance provider returns the order ID of the base order which you can capture in an XQuery variable (such as **\$parentOrderID**). You can use this variable in a data

instance provider that runs a GetOrder Web Service to obtain the order item details from the base order. For example, the following XQuery populates the GetOrder request message using the results from the "findOrder" data instance provider to provide the value for the order ID of the base order in the Order ID field:

```
<ord:GetOrder xmlns:ord="http://xmlns.oracle.com/communications/ordermanagement"
  xmlns:osm="http://xmlns.oracle.com/communications/ordermanagement/model">
  <ord:OrderId>{vf:instance("findOrder")/ord:Order[last()]/ord:Amendments/
ord:AmendedOrderSummary/ord:Id/text()}</ord:OrderId>
<ord:View>CommunicationsSalesOrderQueryTask</ord:View>
</ord:GetOrder>
```

This data instance provider returns all order item instances in the base order that you can then search through to find the blocking order item using the **\$dependingLineId** variable. You can capture the results in an XQuery variable (such as **\$parentOrderItemId**). For example:

```
let $parentOrderItemId :=fn:normalize-space(vf:instance("getOrder") /ord:Data/
osmc:_root/osmc:ControlData/osmc:OrderItem [osmc:BaseLineId=$dependingLineId]/@index)
```

The XQuery body returns the order ID of the base order and the order item property that specifies the blocking order item on the base order:

```
<osm:dependency fromOrderId="{ $parentOrderId}"
fromOrderItemId="{ $parentOrderItemId}"/>
```

where

- **<osm:dependency fromOrderId**: Returns the base order ID.
- **fromOrderItemId**: Returns the blocking order item property value that controls the dependency. OSM internally monitors the blocking order item until it is no longer being processed by any order component on the base order.

The following example shows an XQuery for an inter-order dependency.

```
declare namespace ord="http://xmlns.oracle.com/communications/ordermanagement";
declare namespace im="CommunicationsSalesOrderFulfillmentPIP";
declare namespace osmc="urn:oracle:names:ordermanagement:cartridge:
CommunicationsSalesOrderFulfillmentPIP:1.0.0:view:CommunicationsSalesOrderQueryTask";
let $dependingLineId := fn:normalize-space(osm:properties /
im:DependingSalesOrderBaseLineId)
return
  if(fn:not($dependingLineId = ''))
  then
    (: Use the data instance behavior "findOrder" to find the base order: :)
    let $parentOrderId := fn:normalize-space(vf:instance("findOrder")/
ord:Order[last()]/ord:Amendments/ord:AmendedOrderSummary/ord:Id/text())
    (: Use the data instance behavior "getOrder" to find the associated order item ID in the
base order: :)
    let $parentOrderItemId :=
      fn:normalize-space(vf:instance("getOrder")/ord:Data/
osmc:_root/osmc:ControlData/osmc:OrderItem[osmc:BaseLineId=$dependingLineId]/
@index)
    return
      if(fn:not($parentOrderId = '') and fn:not($parentOrderItemId = ''))
      then
        (: Return the dependency: :)
        <osm:dependency fromOrderId="{ $parentOrderId}"
fromOrderItemId="{ $parentOrderItemId}"/>
      else()
    else()
```

Order Transformation Manager XQuery Expressions

The following topics provide reference information about order transformation manager XQuery expressions.

- [About Transformation Sequence XQuery Expressions](#)
- [About Mapping Rule XQuery Expressions](#)
- [About Order Item Parameter Binding XQuery Expressions](#)
- [About Transformed Order Item Fulfillment State XQuery Expressions](#)

About Transformation Sequence XQuery Expressions

When working with Transformation Sequence editor, see the following topics for information about defining XQuery expressions related to transformation sequences:

- [About Order Item Context XQuery Expressions](#)
- [About Related Order Item Selector XQuery Expressions](#)
- [About Stage Condition XQuery Expressions](#)

About Order Item Context XQuery Expressions

This topic describes how to use the Transformation Sequence editor, **Dependencies** tab, **Order Item Context** subtab, Expression area, **XQuery** subtab to write an expression that defines the context order items for the order transformation. To see the **Order Item Context** subtab, you must select a transformation stage in the tree on the Dependencies tab.

- Context: The input document is the complete set of source order items.
- Prolog: You can declare the order item namespace in the XQuery prolog. For example:

```
declare namespace prop='http://oracle.communications.centralom';
```

- Body: The XQuery body returns the source order items that should be considered the context for the transformation stage.

The following example shows an XQuery expression for selecting an order item context.

```
declare namespace prop='http://oracle.communications.centralom';  
osm:orderItem[osm:properties/prop:serviceIntance = 'Y']
```

About Related Order Item Selector XQuery Expressions

This topic describes how to use the Transformation Sequence editor, **Dependencies** tab, **Related Order Item Selector** subtab, Expression area, **XQuery** subtab to write an expression that defines the related order items for a particular context order item. To see the **Related Order Item Selector** subtab, you must select a transformation stage in the tree on the Dependencies tab.

- Context: The input document is a context order item.
- Prolog: You can declare the order item namespace and the namespace for the order transformation manager functions in the XQuery prolog. For example:

```
declare namespace prop='http://oracle.communications.broadband';  
declare namespace
```

```
otmfn="java:oracle.communications.ordermanagement.orchestration.transformation.XQueryFunctions.";
```

- Body: The XQuery body returns the source order items related to the context order items.

The following example shows an XQuery expression that returns sibling order items as related order items to the order transformation.

```
declare namespace prop='http://oracle.communications.broadband';
declare namespace
otmfn="java:oracle.communications.ordermanagement.orchestration.transformation.XQueryFunctions.";
let $siblings := otmfn:siblings (., '{http://
oracle.communications.broadband}default')
return $siblings[! fn:exists(osm:properties[prop:serviceInstance = 'Y'])]
```

For more information about the **transformation.XQueryFunctions** class, install the OSM SDK and extract the OSM Javadocs from the **SDK/osm7.w.x.y.z-javadocs.zip** file (where *w.x.y.z* represents the specific version numbers for OSM). See *OSM Installation Guide* for more information about installing the OSM SDK.

About Stage Condition XQuery Expressions

This topic describes how to use the Transformation Sequence editor, **Dependencies** tab, **Stage Condition** subtab, Expression area, **XQuery** subtab to write an expression that determines whether a particular transformation stage should be run. To see the **Stage Condition** subtab, you must select a transformation stage in the tree on the Dependencies tab.

- Context: The input document is the complete set of target order items.
- Prolog: You can declare the order item property and parameter namespaces in the XQuery prolog. For example:

```
declare namespace prop='http://oracle.communications.broadband';

declare namespace parm='http://oracle.communications.broadband';
```

- Body: The XQuery body returns a Boolean, with true meaning that the transformation stage should be run and false meaning that the transformation stage should not be run.

The following example shows an XQuery expression that returns true if certain parameters have not been defined, and false if the parameters are already defined.

```
declare namespace prop='http://oracle.communications.cso';
declare namespace parm='http://oracle.communications.broadband';
not(fn:exists(osm:properties/prop:Parameters[fn:exists(parm:uploadSpeed) and fn:exists(parm:downloadSpeed)]))
```

About Mapping Rule XQuery Expressions

When working with Mapping Rule editor, see the following topics for information about defining XQuery expressions related to order decomposition:

- [About Mapping Condition XQuery Expressions](#)
- [About Action Mapping XQuery Expressions](#)
- [About Entity-to-Entity Advanced Mapping XQuery Expressions](#)
- [About Entity-to-Data-Element Advanced Mapping XQuery Expressions](#)
- [About Data-Element-to-Data-Element Advanced Mapping XQuery Expressions](#)

- [About Reverse Mapping XQuery Expressions](#)
- [About Multi-Instance XQuery Expressions](#)

About Mapping Condition XQuery Expressions

This topic describes how to use the Mapping Rule editor, **Mapping** tab, **Condition** subtab, Expressions area, **XQuery** subtab to write an expression that defines a condition that must be satisfied to apply this mapping.

- Context: The input document is a target order item.
- Prolog: You can declare the order item namespace in the XQuery prolog. For example:

```
declare namespace prop='http://oracle.communications.broadband';
```
- Body: The XQuery body returns a Boolean, with true meaning that the mapping rule should be run and false meaning that the mapping rule should not be run.

The following example shows an XQuery expression that runs the rule only if the target action is **None**.

```
declare namespace prop='http://oracle.communications.cso';
osm:properties/prop:Action/text() = 'None'
```

About Action Mapping XQuery Expressions

This topic describes how to use the Mapping Rule editor, **Mapping** tab, **Actions** subtab Action Mappings area, **XQuery** subtab to write an expression that defines the mapping for an action code for a particular mapping rule. To access this field, you must deselect **Use Relationship Action Map** and select the **Advanced** option.

- Context: The input document is a source order item.
- Prolog: You can declare the following variables within the prolog to determine the action code.
 - You can declare **\$sourceValue** to access the action code of the source order item. This is the **Order Item Action** property value for the source order item.
 - You can declare **\$currentTargetValue** to access the action code of the target order item. This is the **Order Item Action** property value for the target order item.
- Body: The XQuery body returns an action code, or returns () to leave the current value unchanged.

The following example shows an XQuery expression that returns the source action code if the target action code is not already set and otherwise leaves the target action code unchanged.

```
declare $sourceValue external;
declare $currentTargetValue external;
if (! fn:empty($currentTargetValue))
  $sourceValue
else
  ()
```

About Entity-to-Entity Advanced Mapping XQuery Expressions

This topic describes how to use the Mapping Rule editor, **Mapping** tab, **Mapping** subtab, Mapping Rule Item area, **XQuery** subtab to write an expression that defines an advanced mapping between two entities. This field is displayed when you select the target of an entity-to-entity mapping. This is the only type of mapping available for entity-to-entity mapping.

- Context: The input document is a source order item.
- Prolog: You can declare any namespaces needed to construct the target property (or properties) in the XQuery prolog. For example:

```
declare namespace prop='http://oracle.communications.cso;
```
- Body: The XQuery body returns a list of order item properties to be set on the target order item. If the property already exists on the target order item, it will be overwritten by the value returned from this XQuery expression.

The following example shows an XQuery expression that returns the structured Parameters property for the target order item.

```
declare namespace prop='http://oracle.communications.cso;
<prop:Parameters xmlns:param="http://oracle.communications.broadband">
  <param:AAAAccount>Account1</param:AAAAccount>
  <param:DownloadSpeed>6</param:DownloadSpeed>
  <param:UploadSpeed>0.6</param:UploadSpeed>
  <param:MAC/>
  <param:Brand>Siemens</param:Brand>
  <param:Model>4200</param:Model>
  <param:Firewall>Y</param:Firewall>
</prop:Parameters>
```

About Entity-to-Data-Element Advanced Mapping XQuery Expressions

This topic describes how to use the Mapping Rule editor, **Mapping** tab, **Mapping** subtab Mapping Rule Item area, **XQuery** subtab to write an expression that defines an advanced mapping between an entity and a data element. This field is displayed when you select the target of an entity-to-data-element mapping and select the **Advanced** option in the **Mapping Rule Item** topic.

- Context: The input document is a source order item.
- Prolog: There is no prolog for this XQuery.
- Body: The XQuery body returns a data element value or returns () to leave the current value unchanged.

The following example shows an XQuery expression that returns "Y" if a particular parameter exists, and () if it does not exist.

```
if fn:exists(vf:instance("checkMe")/somevalue)
  "Y"
else
  ()
```

About Data-Element-to-Data-Element Advanced Mapping XQuery Expressions

This topic describes how to use the Mapping Rule editor, **Mapping** tab, **Mapping** subtab **Configuration** subtab, **XQuery** subtab to write an expression that defines an advanced mapping between two data elements. This field is displayed when you select the target of a data-element-to-data-element mapping and select the **Advanced** option in the **Mapping Rule Item** topic.

- Context: The input document is a source order item during normal transformation. If invoked during forward data propagation, the input document is empty.
- Prolog: You can declare the order item namespace in the XQuery prolog. For example:

```
declare namespace prop='http://oracle.communications.centralom';
```

You can also declare the following variable within the prolog to determine the action code.

- You can declare **\$value** to contain the values of the target data elements.
- Body: The XQuery body returns one or more data element values or returns () to leave the current value unchanged.

The following example shows an XQuery expression that returns the target value of a data element based on the value of the source data element.

```
declare variable $value external;  
if (fn:empty($value)) then ('unknown') else (fn:concat('Loc: ', $value))
```

The following example shows an XQuery expression that returns the target value of a data element based on characteristics of the source order item.

```
declare namespace prop='http://oracle.communications.centralom';  
if (fn:exists(osm:properties/prop:ServicePoint/text()))  
then (fn:concat('Loc: ', fn:normalize-space(osm:properties/prop:ServicePoint/string())))  
else ('unknown')
```

About Reverse Mapping XQuery Expressions

This topic describes how to use the Mapping Rule editor, **Mapping** tab, **Mapping** subtab, **Bi-Directional Mapping** subtab, **XQuery** subtab to write an expression that defines an advanced mapping between two data elements. This field is displayed when you select the target of a data element-to-data element mapping and select the **Advanced** option in the **Mapping Rule Item** topic, if **Supports Bi-Directional Mapping** is selected in the **Details** subtab of the **Mapping** tab for the selected mapping.

- Context: The input document is empty.
- Prolog: You can declare the following variables within the prolog to determine the action code.
 - You can declare **\$value** to access the updated target value.
- Body: The XQuery body returns the updated source value.

The following example shows an XQuery expression that returns () if the return value is unknown and otherwise returns the updated value.

```
declare variable $value external;  
if ('unknown' = $value) then() else (fn:substring($value, 5))
```

About Multi-Instance XQuery Expressions

This topic describes how to use the Mapping Rule editor, **Mapping** tab, **Mapping** subtab, **Multi-Instance Expression** subtab, **XQuery** subtab to write an expression that defines key mapping for a multi-instance structure. This field is displayed when you select the target of a data element-to-data element mapping and select the **Advanced** option in the **Mapping Rule Item** topic, if the target data element is a member of a multi-instance structure.

- Context: The input document is a source order item.
- Prolog: You can declare the order item namespace in the XQuery prolog. For example:

```
declare namespace prop='http://oracle.communications.broadband';
```
- Body: The XQuery body returns a key value that identifies a source order item instance.

The following example shows an XQuery expression that returns the concatenation of two source order item properties for the key value.

```
fn:concat(prop:areaCode, '-', prop:localNumber)
```

About Order Item Parameter Binding XQuery Expressions

This topic describes how to use the Order Item Parameter Binding editor, **Parameter Bindings** tab, Binding Expression area, **XQuery** subtab to write an expression that defines the bindings for one or more parameters on a conceptual model entity from an order item.

- Context: The input document is an input order item. Each order item element in this node set is passed into the XQuery as the context.
- Prolog: You can declare the namespace for the incoming order and the namespace for the conceptual model entity in the XQuery prolog. For example:

```
declare namespace im="http://xmlns.oracle.com/InputMessage";
declare namespace otm="CommonModelBroadbandCart/1.0.0.0.0";
```

- Body: The body of the XQuery will return a node set of elements that correspond to the conceptual model entity data elements. Since you can have as many separate bindings between the entities as you like, this can return anything from one data element to all of them.

The following example shows an XQuery expression that returns an UploadSpeed and a DownloadSpeed parameter from two name-value pairs where the names are Upload Speed and Download Speed.

```
declare namespace fulfillord="http://xmlns.oracle.com/InputMessage";
declare namespace otm="OSMCom_3Play/1.0.0.0.0";
```

```
<otm:UploadSpeed>{fn:normalize-space(fulfillord:itemReference/
fulfillord:specificationGroup/fulfillord:specification[fulfillord:name='Upload Speed']/
fulfillord:value)}</otm:UploadSpeed>
```

```
<otm:DownloadSpeed>{fn:normalize-space(fulfillord:itemReference/
fulfillord:specificationGroup/fulfillord:specification[fulfillord:name='DownloadSpeed']/
fulfillord:value)}</otm:DownloadSpeed>
```

About Transformed Order Item Fulfillment State XQuery Expressions

This topic describes how to use the Transformed Order Item Fulfillment State Composition Rule Set editor, **Composition Rules** tab, **Source Order Item** subtab, **XQuery** field to write an expression that defines the conceptual model entities that should be present if the condition is to be evaluated. This field is only available when you have a condition selected in the tree in the tab, and you have selected the **Advanced** option on the subtab.

- Context: The input document is the order.
- Prolog: You can declare **\$orderItemIndex** to access the index of the order item being considered.
- Body: The body of the XQuery will return a Boolean value indicating whether the current rule should be used to calculate the fulfillment state.

The following example shows an XQuery expression that returns true if a particular order item property has a specific value.

```
declare variable $orderItemIndex external;
```

```
let $orderData := fn:root(.) / GetOrder.Response
let $orderItem := $orderData / _root / ControlData / OrderItem[@index=$orderItemIndex]
return
  if (fn:exists($orderItem) and fn:data($orderItem/AnyProperties) = 'ABC')
```

```
then fn:true()  
else fn:false()
```