Oracle® Communications Unified Inventory Management

Developer's Guide

Release 7.4

E88058-01

December 2017

ORACLE®

Oracle Communications Unified Inventory Management Developer's Guide, Release 7.4

E88058-01

Contents

3 Using the Persistence Framework

4 Extending the Data Model

5 Extending Life Cycles

6 Extending the Topology

7 Extending Security

8 Extending UIM Through Rulesets

12 Localizing UIM

13 Optimizing Concurrent Resource Allocation

14 Using the Federation Framework

15 Integrating UIM Using UIM-Formatted URLs

A Federation Data Domain Cartridges

B Federation Protocol Cartridges

C Base Rulesets

Preface

This guide explains how to extend Oracle Communications Unified Inventory Management (UIM) through standard Java practices using Oracle Communications Design Studio, which is an Eclipse-based integrated development environment. This guide includes references to both applications, and often directs the reader to see the Design Studio Help and the UIM Help for instructions on how to perform specific tasks.

This guide should be read after reading *UIM Concepts*, because this guide assumes that the reader has a conceptual understanding of UIM. This guide should be read from start to finish because the information presented in a chapter often builds upon information presented in a preceding chapter.

This guide includes examples of typical development code used in given situations. The guidelines and examples may not be applicable in every situation.

Audience

This guide is intended for developers who implement code to extend UIM. The developers should have a good working knowledge of XML and Java development and, in particular, JPA, standard Java practices, and J2EE principles.

Related Documentation

For more information, see the following documents in the Oracle Communications Unified Inventory Management documentation set:

- *UIM Installation Guide*: Describes the requirements for installing UIM, installation procedures, and post-installation tasks.
- *UIM System Administrator's Guide*: Describes administrative tasks such as working with cartridges and cartridge packs, maintaining security, managing the database, configuring Oracle Map Viewer, and troubleshooting.
- *UIM Security Guide*: Provides guidelines and recommendations for setting up UIM in a secure configuration.
- *UIM Concepts*: Provides an overview of important concepts and an introduction to using both UIM and Design Studio.
- *UIM Web Services Developer's Guide*: Describes the UIM Service Fulfillment Web Service operations and how to use them, and describes how to create custom web services.

- *UIM API Overview*: Provides detailed information and code examples of numerous APIs presented within the context of a generic service fulfillment scenario, and within the context of a channelized connectivity enablement scenario.
- *UIM Information Model Reference*: Describes the UIM information model entities and data attributes, and explains patterns that are common across all entities. This is available on the Oracle Software Delivery Cloud under "Oracle Communications Unified Inventory Management Developer Documentation."
- *Oracle Communications Information Model Reference*: Describes the Oracle Communications information model entities and data attributes, and explains patterns that are common across all entities. The information described in this reference is common across all Oracle Communications products. This is available on the Oracle Software Delivery Cloud under "Oracle Communications Unified Inventory Management Developer Documentation."
- *UIM Cartridge Guide*: Provides information about how you use cartridges and cartridge packs with UIM. Describes the content of the base cartridges.
- *UIM NFV Orchestration Implementation Guide*: Provides information about how you use NFV Orchestration components and the NFV Orchestration RESTful API resources.

For step-by-step instructions for performing tasks, log in to each application to see the following:

- Design Studio Help: Provides step-by-step instructions for tasks you perform in Design Studio.
- UIM Help: Provides step-by-step instructions for tasks you perform in UIM.

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.

1

Overview

This chapter provides an overview of extending Oracle Communications Unified Inventory Management (UIM).

> **Note:** Throughout this guide, the *UIM_Home* placeholder is used to represent the directory where you installed UIM. For a typical UIM installation, *UIM_Home* is **opt/Oracle/Middleware/user_projects/domains/***domain_name***/UIM**, where *domain_name* is the domain name you supplied when installing UIM.

Extending UIM

UIM extensions can be categorized as static or dynamic:

- Static extensions are changes made prior to rebuilding the application, which results in the changes becoming a part of the application deployment. For example, extending the data model involves adding content to the existing metadata files, which are contained within the **inventory.ear** file. So, you must rebuild the **inventory.ear** file to include the changed metadata files, and then redeploy the application for the changes to affect.
- Dynamic extensions are made anytime, applied at run time, and do not require rebuilding the application for the changes to take effect. For example, a cartridge containing specifications can be deployed into UIM, making the specifications available within the application without rebuilding the application.

Creating Cartridges

Cartridges can contain specifications, characteristics, rulesets, and extended code. You can create cartridges to meet specific business needs in Oracle Communications Design Studio. For example, if your equipment requires specific logic not provided by the EquipmentManager class, you can create your own class, inherited from the EquipmentManager class, and write a new method to address the specific equipment logic. The new method can then be called from within a ruleset.

The extensions defined within a cartridge may be static or dynamic. Cartridges are further explored in Chapter 2, "Using Design Studio to Extend UIM".

Extending the Data Model

You can statically extend the data model by adding new columns to existing tables, or by adding new tables. For example, your business requirements may dictate that you save particular information regarding a telephone number that the existing UIM data

model does not save. You can extend the UIM data model to include this piece of information. Your business requirements may dictate that you save information unrelated to any data that the existing UIM data model saves. You can extend the UIM data model to include a new table to retain this information.

This is done through additions to the metadata. The UIM installation provides tools that enable you to automatically regenerate the data model based on the metadata, and to update the application (**inventory.ear**) to reflect the additions. This topic is further explored in Chapter 4, "Extending the Data Model".

You can dynamically extend the data model through characteristics. For example, you can define a specification for a telephone number and add characteristics that further describe the telephone number. When you create entities in UIM based on a specification that includes characteristics, the characteristics are automatically included in the entities. This topic is further explored in *UIM Concepts*. For instructions on how to define characteristics in Design Studio, see the Design Studio Help.

Extending Life Cycles, Topology, and Security

An entity is a Java representation of UIM data, and an entity can be defined as life-cycle-managed in the metadata. Life cycle refers to an entity having a start to its life, an end to its life, and a defined state at any given point during its life. Life-cycle transition definitions are part of the UIM metadata, and you can extend these definitions to solve specific business requirements.

An entity can also be defined as topology-managed in the metadata. Topology is a graphical representation of the spatial relationships and connectivity among your inventory entities. Topology-managed entities map to topology entities, which are used in the graphical representation. UIM defines several entities as topology-managed, and you can extend topology by defining additional entities to be topology-managed.

Information on security is provided in *UIM System Administrator's Guide*. However, this guide (*UIM Developer's Guide*) provides additional security information specific to securing UIM APIs and UIM entities.

These topics are further explored in:

- Chapter 5, "Extending Life Cycles"
- Chapter 6, "Extending the Topology"
- Chapter 7, "Extending Security"

For information specific to securing web services, see *UIM Web Services Developer's Guide*.

Creating Rulesets

A ruleset is custom code that extends existing logic at a specified point. You can dynamically extend UIM by creating rulesets to meet specific business needs. For example, if the default telephone number format does not match the telephone number format used by the country in which you are implementing UIM, you can use a ruleset to reformat the telephone number.

This topic is further explored in Chapter 8, "Extending UIM Through Rulesets".

Creating Web Services

Web services are APIs that can be accessed over a network, such as the Internet, and run on a remote system hosting the requested services. UIM provides web services that are used for service fulfillment and for cartridge management. You can statically extend UIM by creating custom web services. For example, you can write a web service that performs a search for a specified entity, such as a pipe, a party, or a telephone number.

This topic is further explored in *UIM Web Services Developer's Guide*.

Customizing the User Interface

You can customize the user interface by adding fields or functionality to existing pages, or by adding new pages. For example, you may want to add a field named **Type** to the Equipment Maintenance page and populate it with your equipment type. Customizing the user interface statically extends UIM.

This topic is further explored in Chapter 11, "Customizing the User Interface".

Localizing UIM

Localizing UIM is the process of changing the user interface and the online Help from the language in which it was written to another language. This process involves modifying files that contain text that displays in the user interface and the online Help.

This topic is further explored in Chapter 12, "Localizing UIM".

Optimizing Concurrent Resource Allocation in UIM

You can optimize UIM performance by extending entity types that are heavily used in your UIM environment to implement the rowLock pattern.

This topic is further explored in Chapter 13, "Optimizing Concurrent Resource Allocation".

Federation with UIM

You can extend UIM to interface with other external systems through federation, leasing in data, leasing out data, viewing data, or sharing data. UIM provides sample cartridges that you can configure and use, or extend and use as a starting point in creating a custom federation solution.

This topic is further explored in Chapter 14, "Using the Federation Framework", Appendix A, "Federation Data Domain Cartridges" and Appendix B, "Federation Protocol Cartridges".

Requirements for Extending UIM

Extending UIM requires the installation of Design Studio, Oracle WebLogic Server, and UIM. Extensions are developed in Design Studio, but you also need access to a UIM development environment into which you can deploy cartridges and run unit tests.

Tools for Extending UIM

Several tools are available for extending UIM and are described in Chapter 2, "Using Design Studio to Extend UIM".

Design Studio

Design Studio is an Eclipse-based integrated development environment. Design Studio is not part of UIM, but it does come with features specific to UIM that enable you to extend UIM. Information on using Design Studio to extend UIM is in Chapter 2, "Using Design Studio to Extend UIM".

Additional Tools

Additional tools such as Ant, Drools, and Groovy are available to you when extending UIM. The UIM installation includes a collection of Apache Ant executable targets that are used to extend the data model. These targets automate entity regeneration, entity recompilation, and repackaging the application EAR file to include the recompiled entities. The Drools and Groovy plug-ins can be used to edit ruleset syntax within Design Studio. Information on these tools, how to install them, and how to use them is in Chapter 2, "Using Design Studio to Extend UIM".

Documentation for Extending UIM

Additional information needed to extend UIM is described in the following sections. The resources described here are intended to be used together. For example, the Javadoc provides specific information on methods that are available per entity, and method signatures may define specific entity attributes. However, the Javadoc does not get into details regarding the entity itself or any of the attributes it defines; this type of information is covered elsewhere. See "Information Model Documentation" for more information.

Information Model Documentation

Entities are Java representations of UIM data. The entities that comprise UIM are detailed in *Oracle Communications Information Model Reference* and *UIM Information Model Reference*. The documents describe each entity, lists the entity attributes, provides examples, and includes information on patterns that are common across all entities.

Oracle Communications Information Model Reference and *UIM Information Model Reference* are located under “Oracle Communications Unified Inventory Management Developer Documentation” on the Oracle Software Delivery Cloud.

API Documentation

Information on UIM APIs is detailed in *UIM API Overview*. The document provides detailed information and code examples of numerous APIs presented within the context of a generic service fulfillment scenario, and within the context of a channelized connectivity enablement scenario. The document also provides information about transactions, exceptions, and logging when working the APIs. It also provides a complete listing of the UIM entity manager classes and common utility code examples.

Javadoc Documentation

The classes that comprise UIM, and the Platform classes upon which UIM is built, contain Javadoc. The Javadoc that comes with the UIM installation includes both UIM and Platform Javadoc.

To access the Javadoc:

1. Start the application server.

For instructions on how to start the application server, see *UIM System Administrator's Guide*.

2. From the application server console, deploy the *UIM_Home*/**app/inventory.ear** file, which automatically deploys the *UIM_Home*/**doc/ora_uim_javadoc.war** file.

   For instructions on how to deploy a file from the application server console, see *UIM System Administrator's Guide*.

3. In your Web browser, do one of the following:

   - If UIM was installed with SSL, enter:

     https://*server*:*port*/**ora_uim_javadoc**

   - If UIM was installed without SSL, enter:

     http://*server*:*port*/**ora_uim_javadoc**

   where *server* is the specific server on which the application is deployed and *port* is the port on which the application listens.

Guidelines for Extending UIM

You should be aware of backward compatibility guidelines when extending UIM.

Backward Compatibility

Before you extend UIM, understand the implications of backward compatibility and the effects on future upgrades.

UIM maintains backward compatibility for one release for all published external interfaces:

- Manager interfaces and method signatures
- Published extension points
- Web service interfaces

UIM does not maintain backward compatibility for:

- Metadata and physical data model
- User interface
- Localization

Detecting Code Changes Between Releases

The *UIM_Home*/**doc/ora_uim_delta.war** file contains information regarding changes between releases. Oracle recommends that you review the WAR file content when upgrading UIM to determine if any of the upgrades affect your current extensions.

To read about code changes between releases:

1. Start the application server.

   For instructions on how to start the application server, see *UIM System Administrator's Guide*.

2. From the application server console, deploy the *UIM_Home*/**doc/ora_uim_delta.war** file.

For instructions on how to deploy a file from the application server console, see *UIM System Administrator's Guide*.

3. In your Web browser, do one of the following:

   - If UIM was installed with SSL, enter:

     https://*server*:*port*/ora_uim_delta**

   - If UIM was installed without SSL, enter:

     http://*server*:*port*/ora_uim_delta**

   where *server* is the specific server on which the application is deployed and *port* is the port on which the application listens.

Software Requirements

For developers, the list of software tools expands beyond installing and running the UIM application. For instance, Design Studio, Java, and Eclipse plug-ins have specific version requirements with each UIM release.

Table 1–1 lists the developer-related software and the required versions.

Table 1–1 Software Versions for Developer Tools

Software	Version
Design Studio	7.3.5
Java JDK	Java 8 (with the latest critical patches applied)
Groovy	2.4.6
Drools	6.5.0
Eclipse	4.6 (Neon)
Oracle WebLogic Server Enterprise Edition (included with the Oracle Fusion Middleware WebLogic Server)	12*c* (12.2.1.2)
Apache log4j	2.7
Ant	1.9.1

For more information on software requirements, see *UIM Installation Guide*.

2

Using Design Studio to Extend UIM

This chapter provides information on Oracle Communications Design Studio, an Eclipse-based integration development environment. Design Studio comes with features specific to Oracle Communications Unified Inventory Management (UIM) that enable you to extend UIM.

Installing Design Studio

Design Studio is used to extend Oracle products. Different features are available for the different Oracle products, and each feature provides JAR files that are unique to the product.

For directions on how to install Design Studio, see *Design Studio Installation Guide*. The instructions describe how to install all available Oracle Communications features with a single installation. Of the features installed, UIM requires:

- Oracle Communications Design Studio Platform
- Oracle Communications Design Studio Domain Modelling
- Oracle Communications Design Studio for Inventory

Configuring Design Studio

To do development work in Design Studio, you must configure the Design Studio environment. This requires:

- Setting System Variables
- Setting the Compiler Compliance Level
- Configuring the eclipse.ini File
- Importing the Model Projects
- Configuring the Project Library List

Setting System Variables

After installing Eclipse, you must set system variables to point to the correct version of the JDK.

To set the system variables:

1. From the Windows **Start** menu, select **Control Panel**, then select **System**.

   The System Properties window appears.

2. Click the **Advanced** tab.

3. Click **Environment Variables**.

   The Environment Variables window appears.

4. Define a new system variable named JAVA_HOME:

   a. In the System Variables section, click **New**.

      The New System Variables window appears.

   b. In the **Variable name** field, enter **JAVA_HOME**.

   c. In the **Variable value** field, enter the path to the **jdk** directory for your installation. For example:

      C:/Java/*jdkVersionDir***/bin**

      where *jdkVersionDir* is the supported JDK version directory.

   d. Click **OK**.

   For information on the JDK version, see "Software Requirements".

5. Update the existing system variable Path:

   a. In the System Variables section, select **Path**, and click **Edit**.

      The Edit System Variables window appears.

   b. In the **Variable value** field, add the path to the **bin** directory for your **jdk** installation. This should be added at the beginning of the Path to take precedence on other possible system path values. For example:

      C:/Java/*jdkVersionDir***/bin**

      where *jdkVersionDir* is the supported JDK version directory.

   c. Click **OK**.

      The Environment Variables window appears.

   For information on the JDK version, see "Software Requirements".

6. Click **OK**.

   The System Properties window appears.

7. Click **OK**.

Setting the Compiler Compliance Level

When you install Eclipse, the compiler compliance level is set to a default value. This compliance value must reflect the correct version of the JDK for UIM.

See "Software Requirements" for information on the JDK version.

To set the compiler compliance level in Design Studio:

1. From the menu, select **Window**, then select **Preferences**.

   The Preferences window appears.

2. In the navigation panel, expand **Java**, and click **Compiler**.

3. Verify that the **Compiler compliance level** is set to the correct Java version.

   If it is not, from the **Compiler compliance level** list, select the correct Java version.

4. Click **Apply**, then click **OK**.

Configuring the eclipse.ini File

You must configure the **eclipse.ini** file to include the -vm run-time option, which is used to locate the Java VM to use to compile projects within Eclipse. If not specified, Eclipse uses a search sequence to locate a suitable VM. If an incorrect VM is used to compile a project, you may encounter issues when deploying the resultant cartridge into UIM.

To configure the **eclipse.ini** file:

1. Navigate to your *Eclipse_Home* directory.
2. Open the **eclipse.ini** file.
3. Add the -vm run-time option and its value (the path to your Java executable) before the -vmargs run-time option.

   When adding the -vm option, follow these guidelines:

   - The -vm option and its value (the path) must be on separate lines.
   - The -vm option value must be the full absolute path to the Java executable, not just to the directory that contains the Java executable.
   - The -vm option value cannot contain spaces. If any of the directories in the path to your Java executable contain spaces, you must rename the directories.
   - The -vm option must be placed before the -vmargs option, which is a default option that is present in the **eclipse.ini** file.

   The following are examples; your exact path to the Java executable may be different:

   ```
   -vm
   C:/Java/jdkVersionDir/bin/javaw.exe
   ```

   Or:

   ```
   -vm
   C:/jdkVersionDir/bin/javaw.exe
   ```

   where *jdkVersionDir* is the JDK version directory where you installed Java.

4. Save and close the **eclipse.ini** file.

See "Software Requirements" for information on the JDK version.

Importing the Model Projects

The following model projects must be imported into your workspace before modeling any UIM entities in Design Studio. The successful compilation of an Inventory project is dependent upon the model projects; however, the model projects are not compiled in Design Studio, nor are they deployed into UIM. The model projects are located in the *UIM_Home*/**cartridges/required** directory, and are also delivered as part of the UIM SDK. See "About the UIM SDK" for more information.

- **ora_uim_mds**
- **ora_uim_model**

For instructions on how to import projects into Design Studio, see the Design Studio Help.

Note: The model projects are installed with UIM. As a result, the model projects may change with each new UIM patchset or maintenance release. Contact your System Administrator to get the latest version of the model projects.

Configuring the Project Library List

Depending on the contents of your project, you may or may not need to configure the project library list. For example, if you are extending a UIM class, the project library list must be configured to point to the location of the UIM JAR file that contains the UIM class you are extending. In this example, the UIM JAR file is required to compile the project. The UIM JAR files, and other files, are provided in the UIM Software Developer's Kit (UIM SDK). See "About the UIM SDK" for more information, including a listing of the UIM SDK contents, and instructions on how to build an Inventory project using the UIM SDK.

Imported projects include a library list of the files needed to compile the project, and the project library list must be configured to point to a location to pick up the cited files.

For instructions on how to configure the project library list, see the Design Studio Help.

Note: Project library lists include JAR files that are installed with UIM. As a result, these JAR files may change with each new UIM patchset or maintenance release. Contact your System Administrator to get the latest version of these JAR files.

About Design Studio Perspectives

Perspectives define your Workbench layout and provide different functionality for working with different types of resources. Several perspectives are available within Design Studio.

When extending UIM, commonly used perspectives include:

- Java
- Studio Design
- Studio Environment

For instructions on how to open a perspective, see the Design Studio Help.

About Design Studio Views

Within a given perspective, views further define your Workbench layout and provide different presentations of resources. Several views are available within Design Studio, and the available views are dependent upon the perspective.

When extending UIM, commonly used views include:

- Java perspective views:
  - Ant
  - Navigator
  - Package Explorer

  - Problems
- Studio Design perspective views:
  - Cartridge
  - Package Explorer
  - Problems
- Studio Environment perspective views:
  - Cartridge Management
  - Environment
  - Problems

For instructions on how to open a view, see the Design Studio Help.

About Cartridges and Cartridge Packs

A cartridge is collection of entity specifications, characteristics, rulesets, and extended code defined in Design Studio. Cartridges are built in Design Studio from projects. When a project is compiled, the result is a JAR file (the cartridge) that you can deploy into UIM. The name you choose for the project becomes the name of the cartridge, and everything you create within that project is automatically part of the cartridge.

A cartridge pack is one or more cartridges that collectively address a particular business need or technology. Oracle offers cartridge packs that extend UIM for a particular technology, such as Cable TV or GSM 3GPP. Cartridge packs can also be created by customers and third parties. Cartridge packs can be deployed as downloaded, or they can be imported into Design Studio and extended before deployment.

You can create your own custom cartridges to extend UIM and to organize the extensions. For example, you could create a cartridge that contains all characteristics, another that contains all specifications, and so forth. Or you could create one cartridge per business area, such as telephone numbers or equipment, where each cartridge contains characteristics, specifications, and so forth, that are specific to the business area.

See *UIM Cartridge Guide* for additional information.

Working with Cartridges in Design Studio

This section includes a brief overview of how you work with projects and cartridges in Design Studio. For more information see Design Studio Help.

Working With Cartridge Dependencies

A cartridge can be dependent on other cartridges. These dependencies are specified on the Project editor **Dependency** tab. For example, all Inventory cartridges are dependent upon the **ora_uim_model** project. So, when you create a new Inventory project, the **Dependency** tab automatically includes **ora_uim_model** in the list of project names. Projects that are listed on the **Dependency** tab indicate that the project can be referenced by the Inventory project at design time.

The projects listed on the **Dependency** tab do not indicate project compilation dependencies, which simply require that dependent projects be present in the workspace. For example, to compile an Inventory project, both the **ora_uim_model** and **ora_uim_mds** projects must be present in your workspace. However, only the

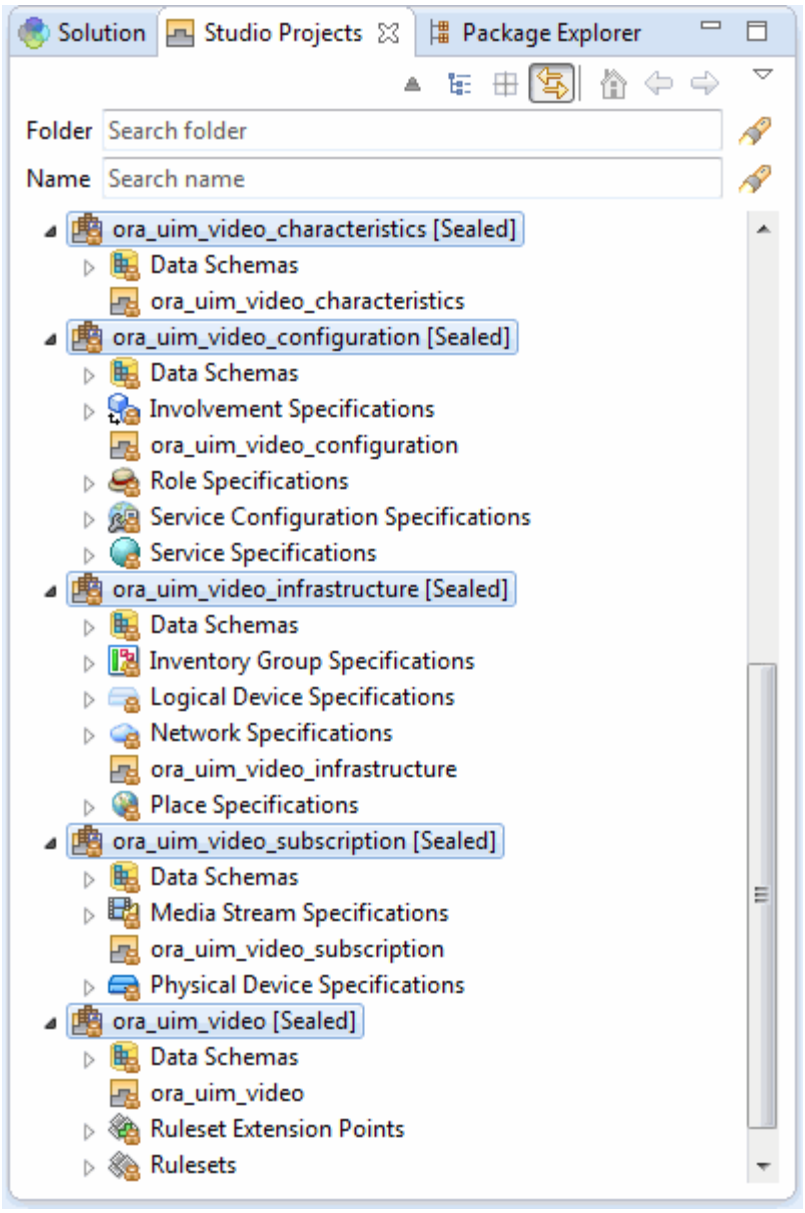
ora_uim_model project is listed on the **Dependency** tab. The **ora_uim_mds** project is not listed on the **Dependency** tab because designing the Inventory project content is not dependent upon referencing anything in the **ora_uim_mds** project; but, the **ora_uim_mds** project must be present in the workspace to compile run-time artifacts that are used by the UIM UI.

You can define additional cartridge dependencies on the Project editor **Dependency** tab, and specify the order of compilation by moving the project names up or down within the list. For instructions on how to define cartridge dependencies, see the Design Studio Help.

About Imported Cartridge Packs

When you import a cartridge pack into Design Studio, its contents are sealed, meaning that you cannot modify them. Figure 2–1 shows the Studio Design perspective Studio Projects view of the imported projects from the Cable TV cartridge pack.

Figure 2–1 Imported Cartridge Packs



Imported projects include a library list of the files needed to compile the project, and the project library list must be configured to point to a location to pick up the cited files. See "Configuring the Project Library List" for more information.

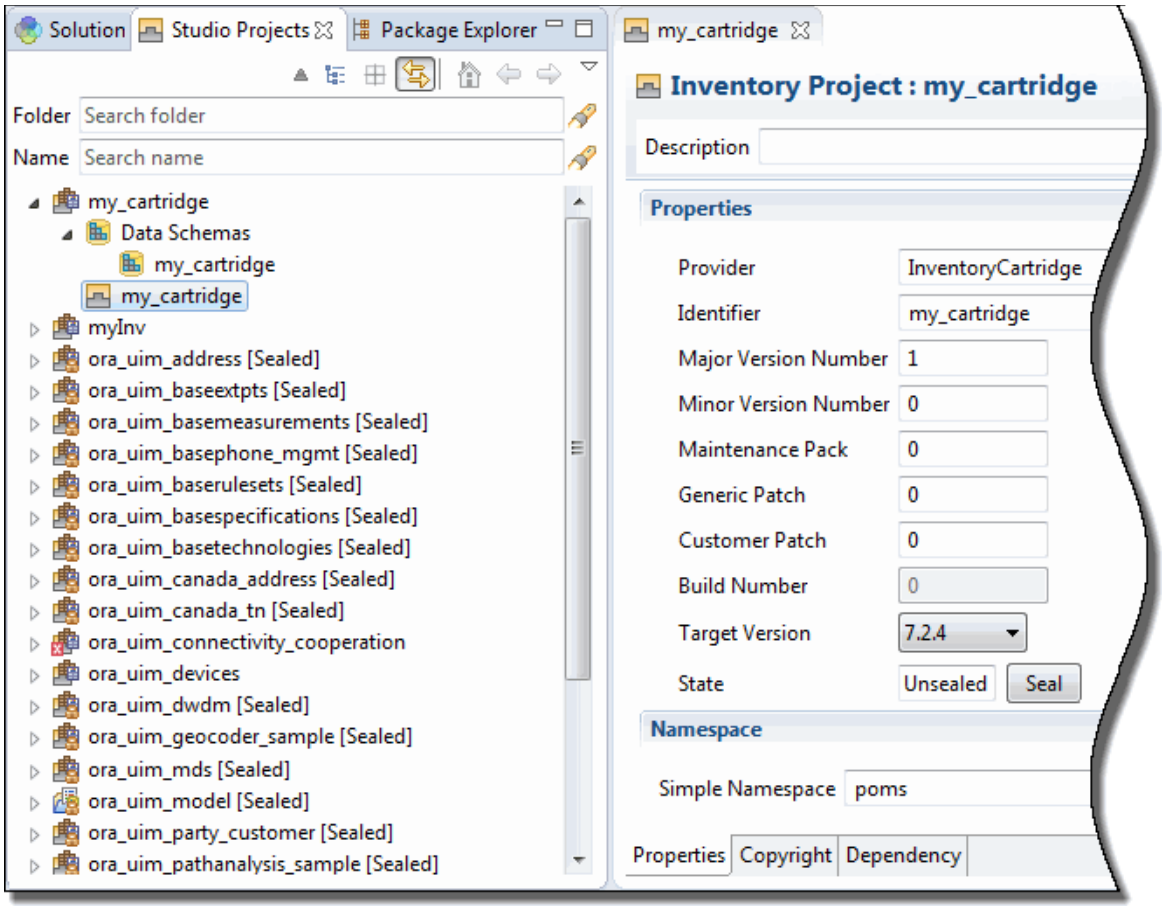
For instructions on how to import a cartridge pack into Design Studio, see the Design Studio Help.

Viewing Cartridges in Design Studio

There are several ways to view the content of cartridges in Design Studio.

Figure 2–2 shows a cartridge project called **my_cartridge** as it appears in the Studio Projects view. The corresponding Inventory Project editor is also shown.

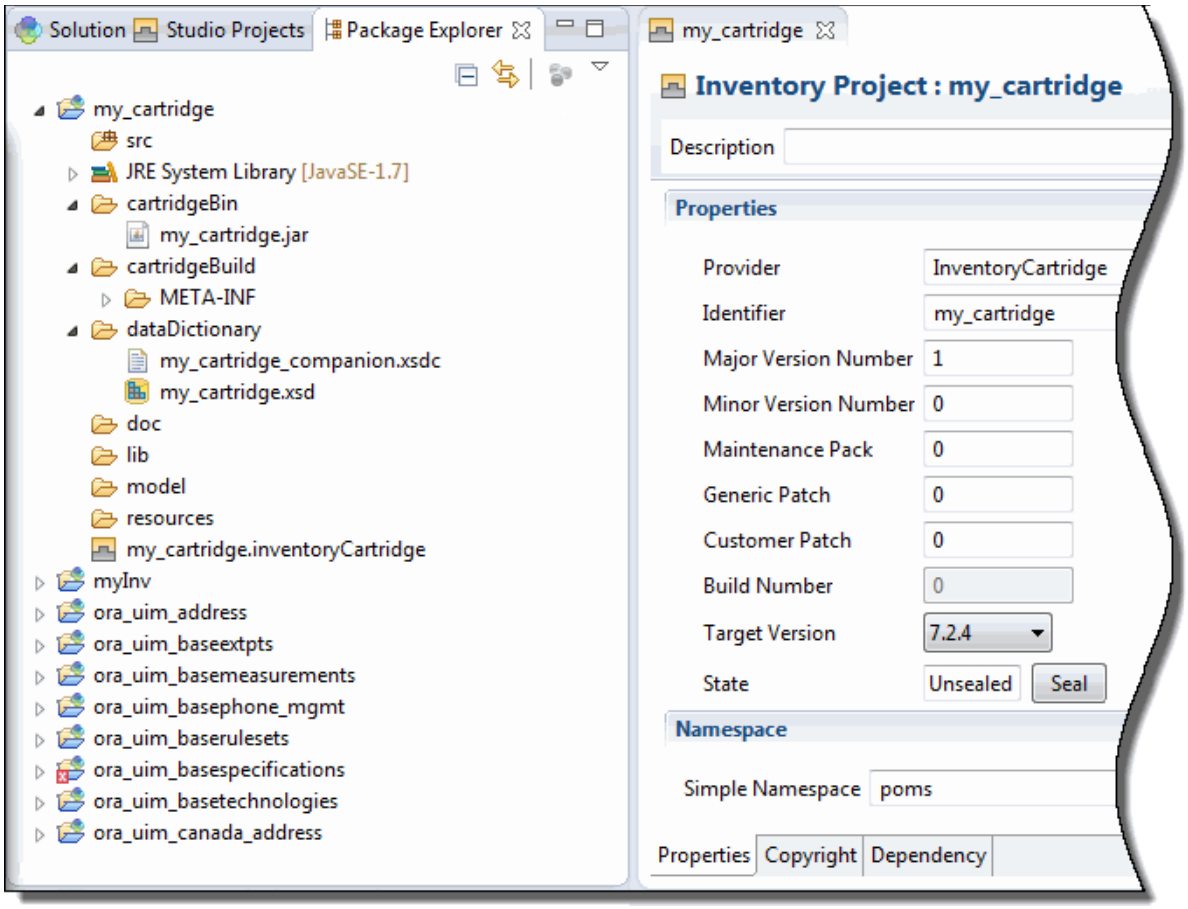
Figure 2–2 Studio Projects View of a Project



By expanding the cartridge in the Studio Projects view, you can see the contents created with each cartridge.

By switching to the Package Explorer view and expanding the cartridge, you can see the file types of the contents created with each cartridge. Figure 2–3 shows **my_cartridge** as it appears in the Package Explorer view. The resultant JAR file resides in the **cartridgeBin** directory. The corresponding Inventory Project editor is also shown.

Figure 2–3 Package Explorer View of Inventory Cartridge



The Package Explorer view shows four files named **my_cartridge**. They are:

- **my_cartridge:** The Studio Inventory project.
- **my_cartridge.jar:** The JAR file that is deployed into UIM.
- **my_cartridge.xsd** and **my_cartridge_companion.xsdc:** Design Studio core files that are used to store characteristics as data elements within a schema entity (data dictionary).
- **my_cartridge.inventoryCartridge:** The Inventory Project editor, shown on the right side of Figure 2–3.

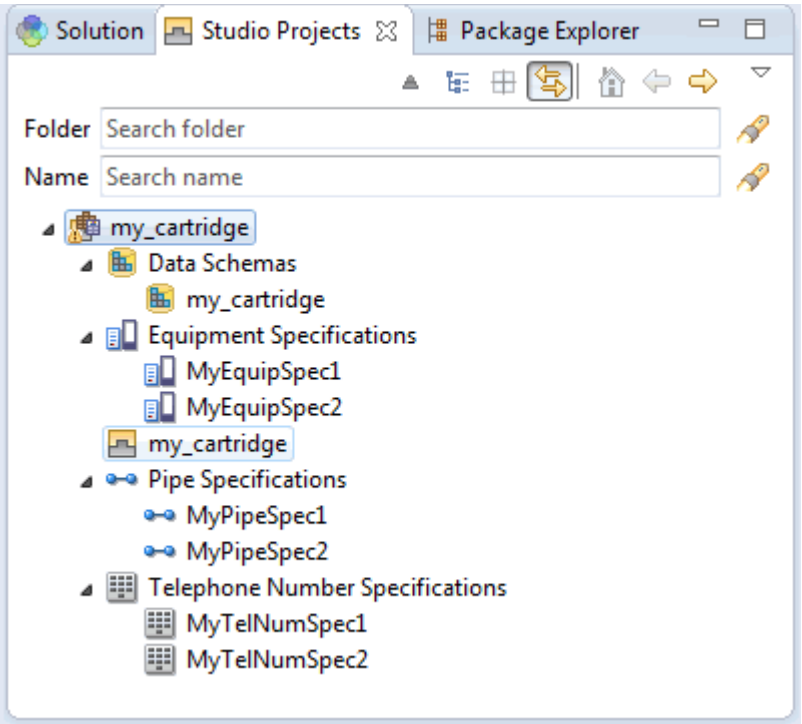
The **cartridgeBin** and **cartridgeBuild** directories do not exist until you build the project, which also creates the **my_cartridge.jar** file.

How Content Is Displayed

The specifications and other content of a project are grouped based on type in the Studio Projects view of the Studio Design perspective. For example, when an Equipment specification is created, it is grouped under **Equipment Specifications**. When a Pipe specification is created, it is grouped under **Pipe Specifications**.

These groupings are purely organizational; they do not represent physical directories. Figure 2–4 shows cartridge content that includes six entities created from three different specifications.

Figure 2–4 Cartridge Content



About Building Cartridges

When building cartridges in Design Studio, it is important that you configure your environment correctly to avoid errors later in the process.

See "Configuring Design Studio" for more information.

About Deploying Cartridges and Cartridge Packs

You can deploy cartridges and cartridge packs into UIM using the following methods:

- Deploy directly from Design Studio.

  You can deploy cartridges and cartridge packs interactively from Design Studio to test environments. Design Studio enables you to manage cartridges in the test environment consistently, manage common test environment connection parameters across the design team, and compare cartridge version and build numbers in the development environment with those of the cartridges deployed in the test environment. See the Design Studio Help for more information.

- Deploy using the Design Studio Cartridge Management Tool.

  The Cartridge Management Tool (CMT) enables you to automate cartridge deployment. You can use the CMT to deploy cartridges into both test and production environments. See the *Design Studio Developer's Guide* for more information.

- Deploy using the UIM Cartridge Deployer Tool.

  The UIM Cartridge Deployer Tool (CDT) is a GUI-based tool that enables you to deploy to UIM run-time environments. The Oracle Universal Installer installs the CDT as part of the UIM installation process. See the *UIM Cartridge Guide* for more information.

See "Troubleshooting Cartridge Deployment" for information on working around possible cartridge deployment issues.

About Cartridge Upgrades

Cartridges can be upgraded. For example, the cartridges in a cartridge pack might be upgraded for a new release. The upgrade process occurs in Design Studio and begins automatically when you open a cartridge that was built in a previous release.

When upgrading a cartridge that is dependent on another cartridge, you must upgrade the dependent cartridge first. During the upgrade process, all dependent cartridges must exist in the workspace to ensure that the upgrade process can convert all cartridges in the correct order.

For instructions on how to upgrade a cartridge, see the Design Studio Help.

About the UIM SDK

The UIM Software Developer's Kit (UIM SDK) provides the resources required to build an Inventory cartridge in Design Studio.

Any custom code that extends UIM can be written in Design Studio by creating an Inventory cartridge and adding custom Java code to the cartridge. In the Java perspective Package Explorer view, you can create package structures and Java source files as needed. Depending on what your custom code references, you may need the SDK to successfully build your Inventory cartridge. For example, you need the SDK if you do one of the following tasks:

- Call a UIM API Entity Manager method.
- Use the utility methods provided in the **ora_uim_common** cartridge.
- Build a new UIM web service.
- Install the TOSCA Parser (an OpenStack project) for use with NFV Orchestration.
- Use the metadata files that define the UIM data model.

This section describes the UIM SDK contents and provides instructions for setting up your Design Studio workspace to include the UIM SDK-provided resources required to build an Inventory cartridge.

Note: Compiled code becomes part of the Inventory cartridge JAR file that can be deployed into UIM. See "About Deploying Cartridges and Cartridge Packs" for more information.

UIM SDK Contents

The UIM SDK contains the directories, subdirectories, and artifacts for the support of Design Studio cartridges. Figure 2–5 shows the following contents:

- **UIM_SDK/cartridges** directory and its following subdirectories
  - **base**: Contains the UIM base cartridges.
  - **required**: Contains the UIM required cartridges for any inventory cartridge.
  - **sample**: Contains some sample cartridges.
  - **tools.studioProjects**: Contains the ora_uim_entity_sdk cartridge ZIP file. See "About the Metadata Files" for more information about this cartridge.

- **UIM_SDK/lib** directory with the list of UIM libraries. See Figure 2–6 for the file content list.
- **UIM_SDK/NSO tools** directory and subdirectory containing artifacts for NFV Orchestration support.
  - **ToscaTranslator**: Contains the Python setup files for Tosca parser support.
- **UIM_SDK/webservices** directory containing cartridge project ZIP files that you import into Design Studio.

For more details on the contents of the **UIM_SDK/webservices** directory and the UIM Reference Web Service, see *UIM Web Services Developer's Guide*.

For more information on the NFV Orchestration RESTful API resources, refer to *UIM NFV Orchestration Implementation Guide*.

Figure 2–5 shows contents of the **UIM_SDK** directory.

Figure 2–5 UIM SDK High Level Contents

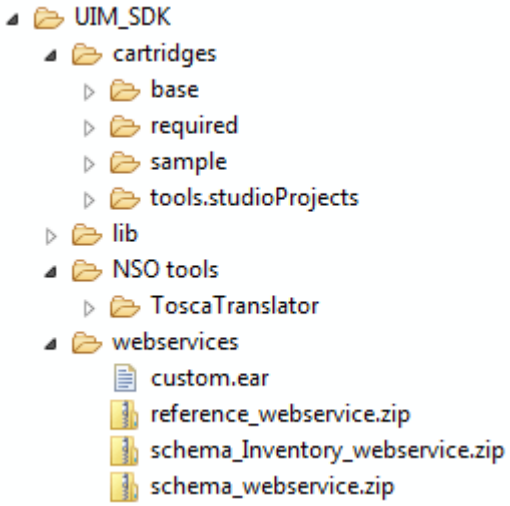
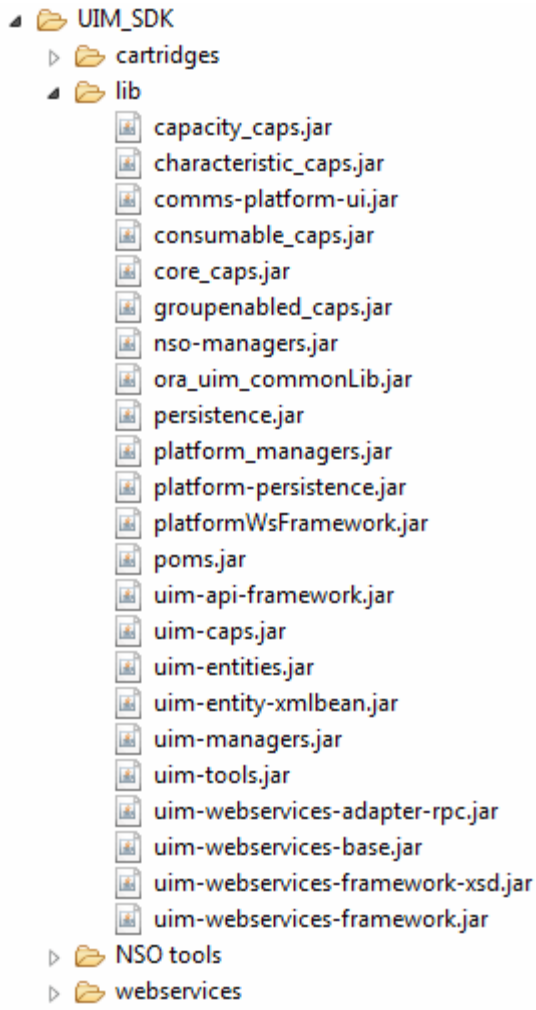


Figure 2–6 shows the **UIM_SDK/lib** directory, which contains UIM JAR files that you use to configure your project library list in Design Studio.

Figure 2–6 UIM SDK lib Directory Contents



See "Building an Inventory Cartridge Using the UIM SDK" for instructions on importing the ZIP files into Design Studio, and configuring your project library list with the JAR files in Design Studio.

Building an Inventory Cartridge Using the UIM SDK

This section assumes you have already installed the following software for use with the UIM SDK for UIM:

- Design Studio

  If not, see *Design Studio Installation Guide* for information on installing Design Studio.

- JDK (with the latest critical patch)

  If not, see *UIM Installation Guide* for information on installing the JDK.

To build an Inventory cartridge using the UIM SDK:

1. Create a local directory, such as *UIM_SDK_Home*.
2. From the Oracle Software Delivery Cloud, download the UIM SDK into the *UIM_SDK_Home* local directory.

3. Open the downloaded **UIM_SDK.zip** file and extract the contents into the *UIM_SDK_Home* local directory.
4. Create another local directory named **OTHER_LIB**.
5. Copy the following WebLogic libraries from your WebLogic Server installation into the **OTHER_LIB** local directory:
   - *WL_Home*/**oracle_common/modules/javax.ejb**_*version*.**jar**
   - *WL_Home*/**oracle_common/modules/javax.jms**_*version*.**jar**
   - *WL_Home*/**oracle_common/modules/javax.persistence**_*version*.**jar**
   - *WL_Home*/**oracle_common/modules/jersey-core-***version*.**jar**
   - *WL_Home*/**wlserver/modules/com.bea.core.xml.xmlbeans**_*version*.**jar**
   - *WL_Home*/**wlserver/modules/javax.transaction**_*version*.**jar**
   - *WL_Home*/**wlserver/server/lib/weblogic.jar**
   - *WL_Home*/**wlserver/server/lib/wlclient.jar**
   - *WL_Home*/**wlserver/server/lib/wsee-client-ext.jar**
   - *WL_Home*/**wlserver/modules/com.bea.core.xml.beaxmlbeans**_*version*.**jar**

   where
   - *WL_Home* is the WebLogic Server installation home directory.
   - *version* is the version number in each filename located in the WebLogic Server installation.

   See "Software Requirements" for version information on the WebLogic Server installation.
6. Copy the **log4j-***version*.**jar** file into the **OTHER_LIB** local directory where *version* is the recommended log4j version.

   Note: If you do not have this file, go to the following website:

   `http://archive.apache.org/dist/logging/log4j`

   Select the appropriate software version directory. Download the **apache-log4j-***version*.**zip** file and extract the **log4j-***version*.**jar** file into the **OTHER_LIB** local directory.

   See "Software Requirements" for version information on log4j utility.
7. Open Design Studio.
8. Configure your Design Studio environment for development work. See "Configuring Design Studio" for more information.
9. Open a new workspace.
10. Import the required cartridges into your workspace from the *UIM_SDK_Home*/**UIM_SDK/cartridges/required** directory.

    Note: See the Design Studio Help for information on how to import projects using archive files related to steps 10-14.

11. Import any needed base cartridges into your workspace from the *UIM_SDK_Home*/**UIM_SDK/cartridges/base** directory.
12. Import any needed sample cartridges into your workspace from the *UIM_SDK_Home*/**UIM_SDK/cartridges/sample** directory.
13. Import any needed tools into your workspace from the *UIM_SDK_Home*/**UIM_SDK/cartridges/tools** directory.
14. Import any needed web service cartridges into your workspace from the *UIM_SDK_Home*/**UIM_SDK/webservices** directory.
15. Create a new Inventory project.

    See the Design Studio Help for information on creating new cartridge projects.
16. Configure the Inventory project library list with the UIM libraries:
    a. Select the Inventory project.
    b. From the project menu, select **Properties**.
    c. Select **Java Build Path**.
    d. Click the **Libraries** tab.
    e. Click **Add Variable**.

       The New Variable Classpath Entry dialog box appears.
    f. Click **Configure Variables**.

       The Preferences dialog box appears.
    g. Click **New**.

       The New Variable Entry dialog box appears.
    h. In the **Name** field, enter **UIM_LIB**.
    i. Click **Folder**, and browse to and select *UIM_SDK_Home*/**UIM_SDK/lib**.
    j. Click **OK**.

       The Preferences dialog box appears.
    k. Click **OK**.

       The New Variable Classpath Entry dialog box appears.
    l. Select **UIM_LIB**.
    m. Click **Extend**.

       The Variable Extension dialog box appears.
    n. Select all libraries.
    o. Click **OK**.

       The Inventory project Java Build Path dialog box appears with the selected libraries added.
    p. Click **OK**.
17. Configure the Inventory project library list with the other libraries from WebLogic and Log4j. To do so, repeat step 16, but:
    a. For step h: In the **Name** field, enter **OTHER_LIB**.
    b. For step i: Click **Folder**, and browse to and select **OTHER_LIB**.

c. For step l: Select **OTHER_LIB**.

18. Build the Inventory project.

    After you successfully build the project, add any custom artifacts and rebuild the project.

About the Developer-Facing Inventory Menu Options

From the **Studio** menu, select **New**, then select **Inventory**, then select **Administration** to see the developer-facing options that are available in Design Studio with the installation of the Inventory feature. The options are:

- Sequence Specification
- Extension Point
- Enabled Extension Point
- Ruleset
- Ruleset Extension
- Ruleset Extension - Global
- Inventory Group Specification

The Sequence Specification option is described in the following sections. The remaining developer-facing options, with the exception of Inventory Group Specification, are described in Chapter 8, "Extending UIM Through Rulesets". For information on Inventory Group Specification, see *UIM Concepts* and the Design Studio Help.

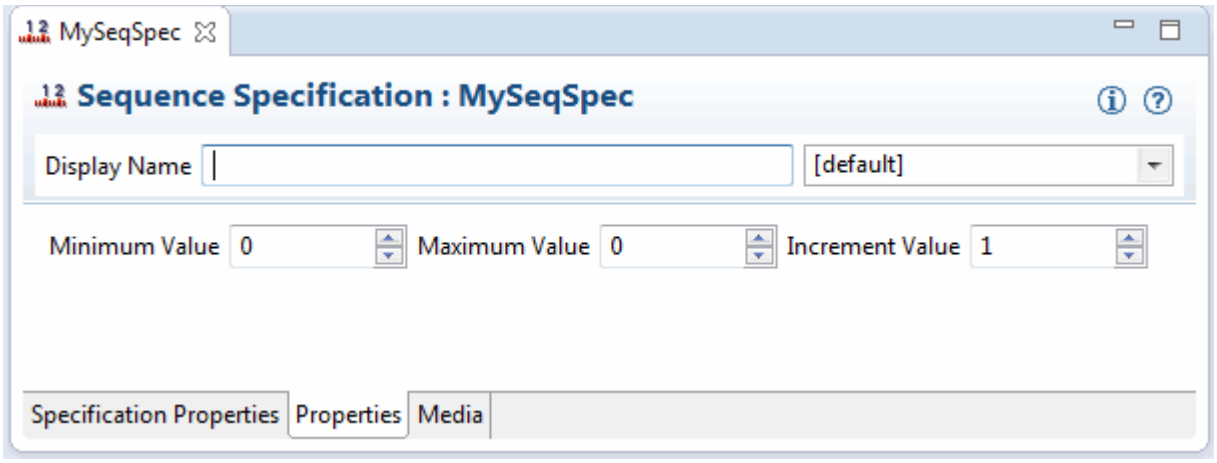
Understanding the Sequence Specification

A sequence is a unique, generated number that is used as an identifier. Sequences can be used alone, or concatenated with other attributes to create a larger identifier, such as a connection ID. For example, in the following connection ID, a sequence can be generated to represent the facility designator in bold: **101**/T1/PLANTXXAK01/IRVGTXXAK1.

The Sequence specification defines criteria for a sequence. Figure 2–7 shows the Sequence Specification editor **Properties** tab where the criteria of **Minimum Value**, **Maximum Value**, and **Increment Value** are defined.

For instructions on how to create a Sequence Specification, see the Design Studio Help.

Figure 2–7 Sequence Specification Editor



The Sequence specification can be used:

- In custom code to set an identifier
- With the Entity Identifier specification to set the entity identifier

Using the Sequence Specification in Custom Code

UIM provides the SequenceGenerator interface, which is a mechanism for generating sequences. The interface exposes methods that generate three types of sequences:

- Global Sequence
- Context-Based Sequence
- Specification-Based Sequence

Global Sequence

A global sequence is a generated number that starts at 1, is incremented by 1, and is unique. When writing custom code, you can obtain a global sequence by calling the following method on SequenceGenerator:

```
public long next()
```

This method returns a global sequence (a number that is unique across all calls to the method).

A global sequence does not use the Sequence specification.

Context-Based Sequence

A context-based sequence is a generated number that starts at 1, is incremented by 1, and is unique within a given context. When writing custom code, you can obtain a context-based sequence by calling the following method on SequenceGenerator:

```
public long next(String context)
```

This method returns a context-based sequence (a number that is unique across all calls to the method that supply the same context). A context-based sequence does not use the Sequence specification.

The request for a context-based sequence results in the creation of an Oracle native sequence, created with a name that equals the context value. This Oracle native sequence is used to generate subsequent sequence values for the context. The

maximum length of an Oracle sequence name is 30 characters. Therefore, the context value for a context-based sequence cannot exceed 30 characters.

Specification-Based Sequence

A specification-based sequence is a generated number that starts at 1, is incremented by 1, and is unique within a given context. Additionally, the number is based on criteria that is defined by a Sequence specification (minimum value, maximum value, and increment value). When writing custom code, you can obtain a specification-based sequence by calling the following method on SequenceGenerator:

```
public long next(String sequenceSpecName, String context)
```

This method returns a specification-based sequence, which is the next sequence value for the combination of the context and sequence specification. (This is a number that is unique across all calls to the method that supply the same context, and that is based on the sequence criteria as defined by the supplied Sequence specification.)

The request for a specification-based sequence results in the creation of an Oracle native sequence, created with a name that follows the naming convention:

```
<CONTEXT>_<SequenceSpecification ENTITYID>
```

where <SequenceSpecification ENTITYID> is the internal primary key ENTITYID value on the SEQUENCESPECIFICATION row for the given sequence specification, and <CONTEXT> is the given context value. This Oracle sequence is used to generate subsequent sequence values for the combination of sequence specification and context.

The maximum length for the CONTEXT portion of the name for a specification-based Oracle native sequence is 10 characters. This constraint is due to the fact that the maximum length of an Oracle native sequence name is 30 characters, and the ENTITYID, (defined as NUMBER(19)) and the underscore take up 20 of the 30 characters.

When calling the next method to get a specification-based sequence, your custom code must be coded with the Sequence specification name. Also, the custom cartridge that defines the Sequence Specification must be deployed before running the custom code.

Note: SequenceGenerator operates outside of a transaction. So, if the transaction gets rolled back, any IDs created are not rolled back. (Oracle native sequences work the same way.)

Using the Sequence Specification with the Entity Identification Specification

The Sequence specification can also be used with the Entity Identification specification to obtain a specification-based sequence to set the entity identifier.

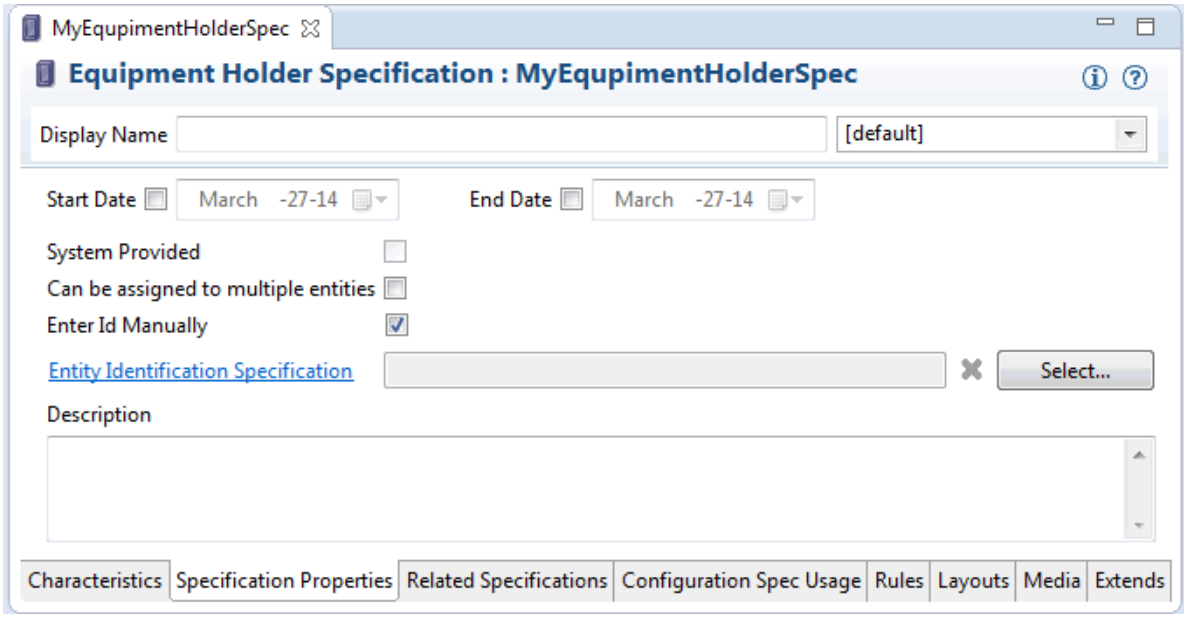
In the metadata, an entity can be defined to have the Entity Identification pattern, as described in *Oracle Communications Information Model Reference*. When an entity defines this pattern, the entity defines the **id** attribute (entity identifier), which is unique across a specific entity type. For example, EquipmentHolder is defined with the Entity Identification pattern. So, each equipment holder defines the **id** attribute, and the **id** attribute value is unique across all equipment holders.

Note: The **id** attribute differs from the **entityId** attribute: Only entities that are defined with the **id** attribute can be defined with the Entity Identification pattern. All entities define the **entityId** attribute, which is always unique across the entire database.

For entities that define the Entity Identification pattern, the specification editor includes the **Enter Id Manually** check box, as shown in Figure 2–8. When the check box is selected, it indicates that when creating an instance of the specification in UIM, you must manually enter the **id** attribute value through the UIM UI.

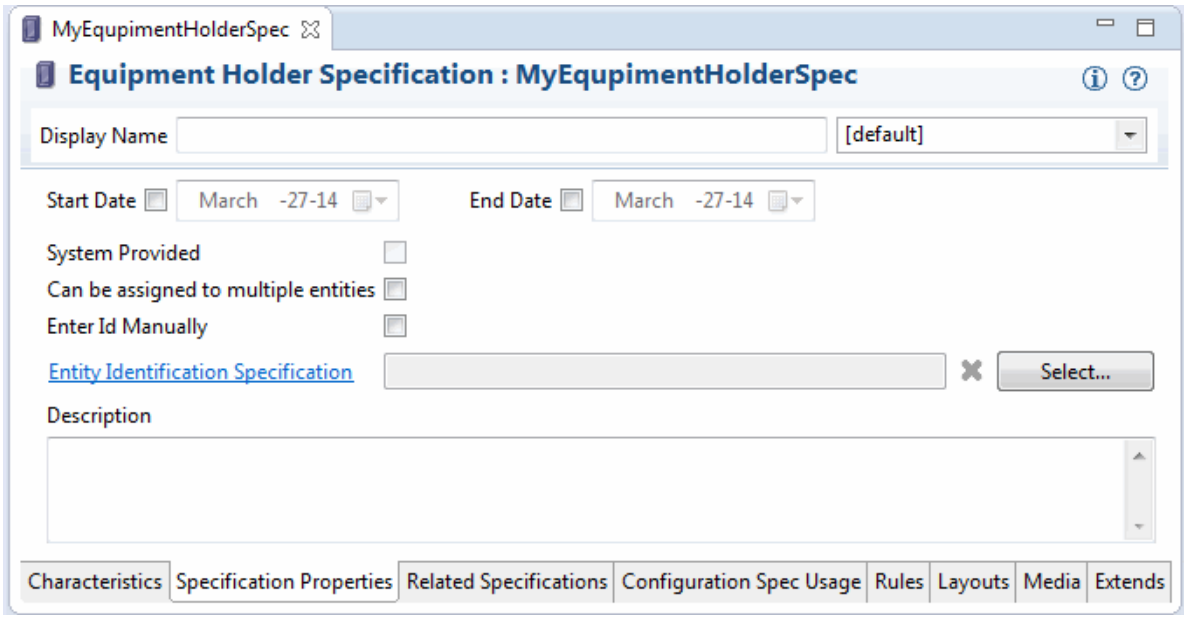
In Figure 2–8, the **Enter Id Manually** check box is selected, and the option to select an Entity Identification Specification is disabled.

Figure 2–8 Specification Editor: Enter Id Manually Selected



When the **Enter Id Manually** check box is deselected, it indicates that when creating an instance of the specification in UIM, the **id** attribute value is automatically generated. In Figure 2–9, the **Enter Id Manually** check box is deselected, and the option to select an Entity Identification Specification is now enabled.

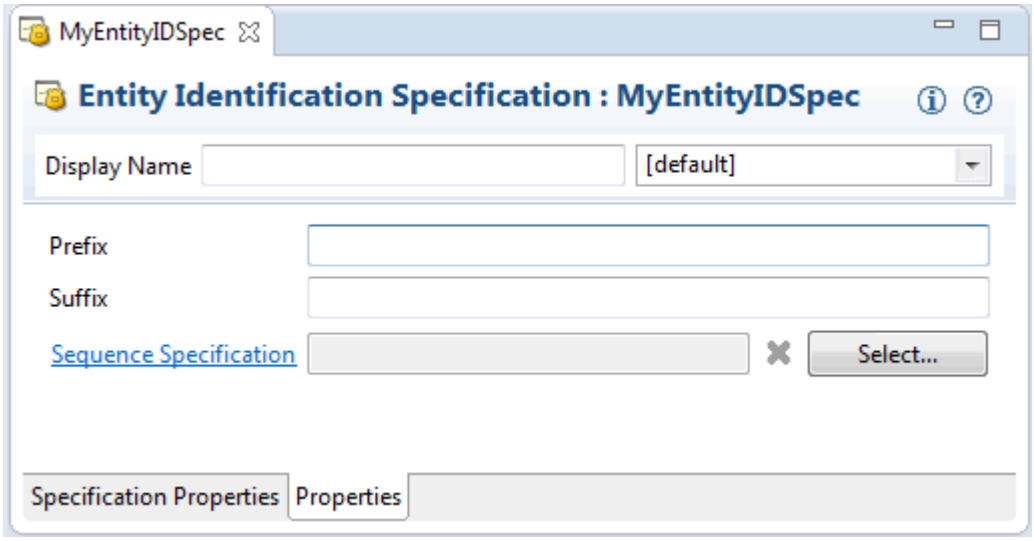
Figure 2–9 Specification Editor: Enter Id Manually Deselected



When the **id** attribute value is to be automatically generated (**Enter Id Manually** is deselected, which is the default), UIM uses the SequenceGenerator interface to obtain a sequence that is used to set the **id** attribute value. You can optionally format the **id** attribute value by selecting an Entity Identification specification. When you click **Select**, a list of all previously defined Entity Identification specifications displays. If no Entity Identification specification is selected, a context-based sequence is generated and used to set the **id** attribute value. For example, in this scenario the context is the Equipment Holder entity type, resulting in the sequence being unique across all equipment holders.

Figure 2–10 shows the Entity Identification Specification **Properties** tab, where you can define the sequence format. For instructions on how to create an Entity Identification Specification, see the Design Studio Help.

Figure 2–10 Entity Identification Specification Editor



Based on the Entity Identification specification, the **id** attribute value is generated as:

```
prefix + sequence + suffix
```

where:

- prefix is the **Prefix** value specified by the Entity Identification specification. Specifying a prefix is optional.
- sequence is a unique sequence value based on Sequence specification criteria. Specifying a Sequence Specification is required.
- suffix is the **Suffix** value specified by the Entity Identification specification. Specifying a suffix is optional.

> **Note:** You can choose to not specify the prefix or suffix. For example, to have your **id** attribute value incremented by 100, define an Entity Identification specification with no prefix or suffix, and specify a Sequence specification that defines an increment value of 100.

Additional Tools

Third-party tools such as Ant, Drools, and Groovy are used to extend UIM:

- **Ant** is used to extend the UIM data model, web services, and user interface.

  Ant is an open source software tool for automating a build process. Ant uses XML to describe a build process and its dependencies. When extending the UIM data model, web services, or user interface, Ant targets are run from within Design Studio. See "Installing, Configuring, and Using Ant" for more information.

- **Drools** is used to extend UIM through rulesets.

  Drools is an open source project that enables accessing, changing, and managing business rules. When extending UIM using rulesets with Drools, Oracle recommends that you install the Drools Eclipse plug-ins in Design Studio. The plug-ins provide a Drools editor, and Drools-specific menu options. See "Installing, Configuring, and Using the Drools Eclipse Plug-ins" for more information.

- **Groovy** is also used to extend UIM through rulesets.

  Groovy is an open source project that enables accessing, changing, and managing business policies. When extending UIM using rulesets with Groovy, Oracle recommends that you install the Groovy Eclipse plug-ins in Design Studio. The plug-ins provide a Groovy editor. See "Installing, Configuring, and Using the Groovy Eclipse Plug-ins" for more information.

Installing, Configuring, and Using Ant

This section provides information on:

- Downloading Ant
- Installing Ant
- Configuring Ant
- Running Ant Targets

Downloading Ant

To download Ant:

1. Go to the following website:

   `http://archive.apache.org/dist/ant/binaries`

2. Scroll down and click the **apache-ant-***releaseNumber***-bin.zip** link, where *releaseNumber* is the latest patch of the recommended Ant software release version. See "Software Requirements" for information on the Ant version.

   The File Download window appears.

3. Click **Save**.

   The Save As window appears.

4. Navigate to a local directory such as **tempDir** and click **Save**.

Installing Ant

To install Ant:

1. In your local hard drive root directory, or in your **Program Files** directory, create a new directory named **ant**.
2. Open the **tempDir/apache-ant-***releaseNumber***-bin.zip** file where *releaseNumber* is the latest patch recommended Ant software release version. See "Software Requirements" for information on the Ant version.
3. Extract the file contents to the **ant** directory you created in step 1.

Configuring Ant

To configure Ant:

1. From the Windows **Start** menu, select **Control Panel**, then select **System**.

   The System Properties window appears.

2. Click the **Advanced** tab.
3. Click **Environment Variables**.

   The Environment Variables window appears.

4. Define a new system variable named ANT_HOME:
   a. In the System Variables section, click **New**.

      The New System Variables window appears.

   b. In the **Variable name** field, enter ANT_HOME.
   c. In the **Variable value** field, enter the path to the extracted directory. For example, **C:/ant/apache-ant** or **C:/Program Files/ant/apache-ant**.
   d. Click **OK**.
5. Update the existing system variable Path:
   a. In the System Variables section, select **Path**, then click **Edit**.

      The Edit System Variables window appears.

   b. In the **Variable value** field, add the path to the **bin** directory of the extracted Ant directory.

      For example, **C:/ant/apache-ant/bin** or **C:/Program Files/ant/ apache-ant/bin**.

c. Click **OK**.

The Environment Variables window appears.

6. Click **OK**.

The System Properties window appears.

7. Click **OK**.

Running Ant Targets

This procedure is applicable only if the cartridge you imported contains a **build.xml** file that defines Ant targets.

To run an Ant target within Design Studio:

1. Open the Java perspective.

For instructions on how to open the Java perspective, see the Design Studio Help.

2. From the **Window** menu, select **Show View**, then select **Ant**.

The Ant window appears.

3. Within the Ant window, right-click and select **Add Buildfiles**.

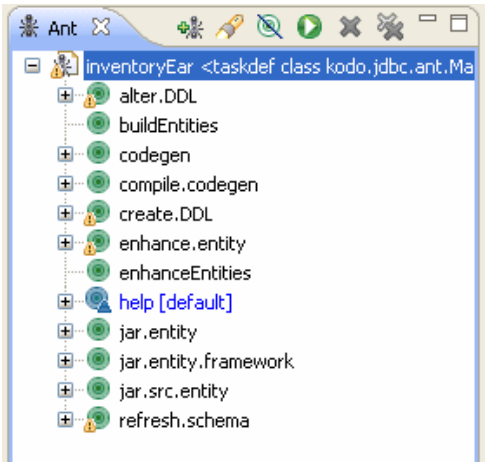
The Buildfile Selection window appears.

4. Navigate to and select the file that contains the Ant target you plan to run.

5. Click **OK**.

The Ant targets defined in the selected file appear in the Ant window, as shown in Figure 2–11.

Figure 2–11 Design Studio Ant Window



6. Double-click a target.

The Ant target runs.

3

Using the Persistence Framework

This chapter provides information on using the persistence framework, which moves program data (in memory objects) to and from a permanent data store (the database). The persistence framework also manages the database and manages the mapping between the database and the objects.

You use the persistence framework when extending Oracle Communications Unified Inventory Management (UIM). For example, custom rulesets or custom web services typically have code that reads or updates the database, which is done using the persistence framework. So, UIIM custom code developers need to be familiar with the contents of this chapter.

About the Persistence Framework Foundation

The persistence framework is built on top of EclipseLink, which implements Java Persistence API (JPA) technology. Functional extensions employ standard Java practices.

This chapter does not replace the EclipseLink or JPA development guides. Both technologies are covered in greater detail at the following websites:

JPA Specifications

http://wiki.eclipse.org/EclipseLink/Specs

EclipseLink

http://wiki.eclipse.org/EclipseLink

JPA

http://www.eclipse.org/eclipselink/#jpa

> **Note:** Documentation on third-party software products is limited to the information needed to use the UIM persistence framework. If you need additional information on a third-party software application, consult the documentation provided by the product's manufacturer.

Understanding Persistence Framework Concepts

The persistence framework employs the concepts of eager and lazy fetching and of managed and non-managed entities, as described in the following sections.

Eager and Lazy Fetching

Note: Information on eager and lazy fetching can be found on the Oracle Technology Network website at:

http://www.oracle.com/technetwork/articles/javase/index-138213.html

A **fetchType** of **eager** means that a persistence provider loads the attribute of an entity along with the entity, while a **fetchType** of **lazy** is a hint to the provider that the attribute need not be fetched along with the entity. This means that even though you may specify the **fetchType** as **lazy**, the persistence provider may choose to load the attribute eagerly.

By default, all relationships are configured as lazy loading in the metadata, and all basic attributes are configured as eager fetched in the metadata. To configure an attribute as lazy loading in the metadata, set the <lazy> attribute to **true**. For example:

```
<entity type="cim:DataTypes"
    interface="oracle.communications.platform.entity.DataTypes">
    <attribute name="clobString" lazy="true"/>
</entity>
```

The result is a generated annotation within the DataTypesDAO.java class. For example:

```
@Basic(fetch=FetchType.LAZY) private java.lang.String clobString;
```

If a field is configured as lazy loading, and you want to eager fetch it when the entity is retrieved from the database, use the Finder.addEagerFetchField() method. For example:

```
finder = PersistenceHelper.makeFinder();
    finder.setResultClass(TelephoneNumber.class);
finder.setJPQLFilter("o.id = :tnId");
finder.addParameter("tnId","88888888");
finder.addEagerFetchField(EagerFetch.LEFT_FETCH,"o.specification");
Collection<TelephoneNumber> tns = finder.findMatches();
finder.close();
```

The previous example shows a fetch mode of LEFT_FETCH. The persistence framework supports the following fetch modes for an eager fetch:

- BATCH: Batch reading may require more than one trip to the database but is usually more efficient than a join fetch, especially join fetches that involve collection relationships. Batch reading configures the query to optimize the retrieval of the related objects, and the related objects for all the resulting objects are read in a single query (instead of multiple queries).
- FETCH: This fetch mode uses an inner join.
- LEFT_FETCH: This fetch mode uses an outer join.

Managed and Non-Managed Entities

A persistence context is a set of entities such that for any persistent identity there is a unique entity instance. Within a persistence context, entities are managed. An entity manager controls life cycles and accesses data store resources.

When a persistence context ends, previously managed entities become non-managed. A non-managed entity is no longer under the control of the entity manager and no longer has access to data store resources. The major difference between an entity that is managed and an entity that is non-managed is:

- When an entity is managed, the object is connected to the database and changes made to the object are reflected in the database when committed, or flushed in a transaction.
- When an entity is non-managed, the object is not connected to the database, so changes are never applied to the database.

The non-managed object can be stale, which can cause you to receive the OptimisticLockingException when calling the EclipseLink attach() method. In this case, discard the stale non-managed object, retrieve the new object from the data store, and perform any update operation against the new version.

When a transaction already exists and an entity is explicitly retrieved from the database, the entity is managed. There is no need to eager fetch the entity attributes because the attributes and relationships can be lazy-loaded while the transaction is still active.

When no transaction exists, the entity becomes non-managed. EclipseLink supports the lazy loading of relationships of a non-managed entity. EclipseLink also supports the lazy loading of primitive attributes such as String.

Note: If an entity is serialized and then deserialized, such as sent through a remote EJB interface or web service, the relationships cannot be lazy loaded, and an eager fetch must be used to access the relationships. Alternatively, the application can start a transaction to make the entity managed. When the entity is managed, the attributes and relationships can be accessed directly, and the eager fetch is not required.

When an entity is created, it is neither managed nor non-managed. The entity status is Transient because there is no representation of the entity in the database yet. All relational and collection attributes on a transient entity are available (that is, if it is null, then it is really null). When the transient entity is passed to a ManagerX.createX(..) method, the entity is persisted into the database. The entity is persisted by reference. The entity life-cycle status is changed from Transient to Persistent-New. There is a copy of the entity created. When the transaction is committed, the entity becomes non-managed.

Usually, the UI code keeps the non-managed entity in the session so that it can be updated. Sometimes, the UI code does an explicit Find to retrieve and store a non-managed entity in a session. You can store the non-managed entity in a session to avoid table locks on the database. The client code performs a set on the detached entity. When the updated entity is passed into the ManagerX.updateX(..) method, the manager persists the changes by using the EclipseLink attach() method. The current implementation of EclipseLink attach makes a copy of the entity and persists the changes. The new copy represents the managed entity, making the non-managed entity obsolete. For example, the createEntity() and updateEntity() methods each return a new copy of the entity. The returned copies are managed entities, making the original non-managed entity obsolete.

```
A = createEntity(A)
A = updateEntity(A)
```

Persistence Framework Classes and API Methods

All of the persistence framework classes covered in this section expose API methods that you can use when extending UIM. For example, you may want to add additional validations to existing UIM functionality, or add additional processing.

Note: For information on the classes described in this section, including a listing of method names, arguments, and returns, see the Javadoc. For instructions on how to access the Javadoc, see "Javadoc Documentation".

PersistenceManager

Package: oracle.communications.platform.persistence

PersistenceManager generically manages entities defined in the ***-entities.xml** files. This manager provides methods to:

- Create, update, and delete an entity or entities
- Check whether an EntityManager is invoked from a Java Transaction API (JTA) context

  (JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: The resource manager, the application server, and the transactional applications. A JEE application may use JTA, but a standalone JSE application does not.)
- Set and get the logging level

This class is a wrapper for the methods defined in the standard javax.persistence.EntityManager class.

TypeRegistry

Package: oracle.communications.platform.persistence

TypeRegistry is a generated class that extends TypeRegistryBase. As part of the entity code generation process, each entity is added to a class list managed by TypeRegistry. TypeRegistry provides convenient methods to get a data access object (DAO) implementation class for each entity. Table 3–1 contains a list of the methods defined in the TypeRegistryBase class.

Table 3–1 TypeRegistryBase APIs

API	Description
classFor(Class)	Gets the concrete class, which implements the given inventory entity class.
interfaceFor(Class)	Gets the inventory entity interface, based on the concrete class.
classForDiscriminator(String discriminator)	Gets the entity implementation class, based on the discriminator.
discriminatorForClass(Class)	Gets the discriminator, based on the implementation class.

Finder

Package: oracle.communications.platform.persistence

Finder provides methods for querying entities based on simple or complex search criteria. It has convenience methods that set up query parameters and fetch properties. Convenient find methods are provided; however, a complex Java persistence query language (JPQL) query can also be built iteratively.

Finder provides the most frequently used query mechanism. Additional query complexity that can be reused should be incorporated into the entity managers instead. The entity managers should then use Finder for building the queries, or use JPA directly. See "Entity Managers" for more information.

Finder defines methods that enable you to:

- Get an entity based on the entity key
- Refresh an entity or a collection of entities
- Find an entity or entities based on various options, such as name, entity, or ID
- Define a JPQL statement
- Run the defined JPQL statement
- Reset the Finder, which resets all query parameters to null

These methods are further explored in the following sections.

Defining JPQL Statement Methods

The Finder class provides numerous methods that you can use to define a JPQL statement. By using these methods you can:

- Set the result class to query
- Add a join expression
- Set filters, such as a where clause or min/max
- Add an attribute to specify the result set be returned in ascending or descending order
- Set a range to filter the result set
- Add and set parameters
- Declare variables
- Add and set variables
- Add hints, which are JPA-specific
- Add eager fetch fields
- Clear eager fetch fields

Finder.find() and Finder.findMatches() Methods

The Finder.find(Class<E> candidateType, String filter) method is a convenient method to use because it does not consider any parameters you set on Finder before you call that method. To include parameters, use the Finder.find(Class< E > candidateType, String filter, String [] paramNames, Object [] params) method.

Alternatively, you can build the parameters list using Finder beforehand, then use the findMatches() method. The findMatches() method uses the parameters you set.

Table 3–2 lists some of the commonly used methods defined in the Finder class.

Table 3–2 Finder APIs

API	Description
find	Overloaded method that finds an entity or entities based on various arguments, such as entity type, the current filter setting on Finder, a list of the current parameters set on Finder, etc.
findByName, findById, findByEntity	Various methods that find an entity or entities based on name, ID, or entity type.
findMin and findMax	Finds the minimum or maximum value based on entity type and the value of min or max, which is used by the method to call Finder.setJPQLFilter().
findMatches	Overloaded methods that finds an entity or entities based on various arguments, such as an Oracle Text search String, and other arguments that you set.
findByJPQL	Finds a result set based on a String argument representing a JPQL statement that you define.
executeUpdateJPQL	Executes an update based on a String argument representing a JPQL statement that you define. This method returns the number of updated entities.
findByNativeSQL	Finds a result set based on a String argument representing a native SQL statement that you define.
executeUpdateNativeSQL	Executes an update based on a String argument representing a native SQL statement that you define. This method returns the number of updated entities.
get	Overloaded method that gets an entity based on entity type and entity key, or based on entity type, entity key, and whether or not the entity is a valid entity.
refresh	Overloaded method that refreshes the given entity, or the given collection of entities.

PersistenceManager refresh(), attach(), and connect() Methods

The basic differences between PersistenceManager.refresh(), PersistenceManager.attach(), and PersistenceManager.connect() are:

- Refresh() refreshes the entity content back to the state of the database, and discards any changes made to the entity. If the entity is managed, the refresh API retrieves a copy from the database to refresh the managed entity. If the entity is non-managed, the refresh API makes the entity managed. Any changes previously made to the managed or non-managed entity are discarded. The refresh API returns the reference to the managed entity.
- Attach() makes the non-managed entity managed, and retains any changes made to the entity. If the entity is already managed, the attach API does nothing in terms of attaching the entity to the database. If the entity is non-managed, the attach API makes the entity managed. Any changes previously made to the managed or non-managed entity are sent to the database by EclipseLink when the transaction is committed or flushed. The attach API returns the reference to the managed entity.
- Connect() makes the non-managed entity managed, and discards any changes made to the entity. If the entity is already managed, the connect API does nothing in terms of connecting the entity to the database. If the entity is non-managed, the connect API makes the entity managed. Any changes previously made to the

managed or non-managed entity are discarded. The connect API returns the reference to the managed entity.

Refresh() does a get from the database. refresh() takes a detached entity, connects it to the database, but does not merge the entity attribute into the database. Refresh() re-retrieves the entity even when it is already attached.

Attach() takes a detached entity and merges its data into the database. The operation fails if the detached entity is stale. When attach attaches the detached entity to the database, it also merges the entity attribute values into the database. Attach() ignores the entity if it is already attached.

If you do not intend to merge the entity attributes of an entity in the database, do not use attach(). If you do, you may be updating an attribute in the database. Also, the last modified fields for the entity are updated, and the entity version is updated.

Note: Using attach() may cause an OptimisticLockVerificationException because it tries to merge values in the database. If the detached entity is a stale entity (some other code thread has modified the same entity and has incremented the entity version), using attach again causes this exception.

InventoryFinder

Package: oracle.communications.inventory.api.framework.persistence

InventoryFinder extends Finder and provides a few additional methods, as described in Table 3–3.

Table 3–3 InventoryFinder APIs

API	Description
find(String queryExpression, Object... parms);	This method finds and returns the result of executing a JPQL search using the passed expression.
findTotalCounts();	This method returns the total number of records found for a given JPQL.

PersistenceHelper

Package: oracle.communications.platform.persistence

PersistenceHelper is a generated class that provides factory methods to get an instance of an entity manager, the TypeRegistry, a Finder, an InventoryFinder, or the PersistenceManager.

Persistent

Package: oracle.communications.platform.persistence

All persistent entities implement the Persistent API. It provides convenience methods for determining the state of an entity. These methods are all read-only; so, the methods can run whether or not there is an active transaction. The following methods are defined in the Persistent API and are available on all entities.

```
public Class getEntityType();
public String getOid();
public long getEntityId();
public String getEntityClass();
```

```
public int getEntityVersion();
public boolean isEntityIdValid();
public Identifier makeIdentitifer();
public void makeTransient();
public boolean isPopulated(String fieldName);
public void unpopulate(String fieldName);
public boolean isTransient();
public boolean isPersistent();
public boolean isTransactional();
public boolean isNew();
public boolean isDirty();
public boolean isDeleted();
public boolean isDetached();
public <E extends Persistent> E connect();
public <E extends Persistent> E refresh();
public <E extends Persistent> E attach();
public String getEntityDescription();
```

Entity Managers

Entity managers are not part of the persistence framework: they are additional managers that use the persistence framework to support the overall application logic. An entity manager manages the database tables for a specific functional area. For example, EquipmentManager manages the Equipment table, but it also manages EquipmentHolder, PhysicalPort, PhysicalConnector, PhysicalDevice, and so forth.

Defining Entity Managers

Entity managers are defined in the metadata by the <manager> element and <interface> attribute. Example 3–1 is an excerpt from the **uim-equipment-entities.xml** file:

Example 3–1 uim-equipment-entities.xml

```
<manager
interface="oracle.communications.inventory.api.equipment.EquipmentManager"
class="oracle.communications.inventory.api.equipment.impl.EquipmentManagerImpl"/>
```

Every entity manager defined in the metadata has a corresponding entity manager and implementation of the manager. So, based on Example 3–1, the following classes exist:

- EquipmentManager
- EquipmentManagerImpl

Entity managers are not generated classes; however, the factory methods in PersistenceHelper that allow for the instantiation of the managers are generated. These factory methods are generated based on the metadata definition.

> **Note:** Entity managers are provided for all entities defined in the metadata. If the database is extended to define new entities, the entity manager and implementation of the manager must be written.

The relationship of entity to entity manager is not one-to-one. For example, in **ocim-equipment-entities.xml** file, there are a number of entities defined, and each entity defines its own entity interface (which differs from a manager interface). An entity interface defines the getter and setter methods for data defined for the entity. Example 3–2 is an excerpt from the **ocim-equipment-entities.xml** file that shows two

entity definitions. The definitions include the interface that is defined for an entity (not for a manager).

Example 3–2 ocim-equipment-entities.xml

```
<entity type="ocim:Equipment"
interface="oracle.communications.inventory.api.entity.Equipment"
accessControlled="true">
.
.
.
<entity type="ocim:EquipmentHolder"
interface="oracle.communications.inventory.api.entity.EquipmentHolder"
accessControlled="true">
```

Entity Manager Implementation Inheritance Structure

The PersistenceManagerBean class is the common base class for all entity manager implementations, and all entity manager implementations extend BaseInvManager. TransitionManagerBean is another layer of inheritance. The inheritance structure of all entity manager implementations is shown below. The following sections discuss each of these classes.

```
PersistenceManagerBean
|
TransitionManagerImpl
|
BaseInvManager
|
EntityNameManagerImpl
```

> **Note:** In some cases, there are additional layers between BaseInvManager and *EntityName*ManagerImpl, but these four layers of inheritance are always present. An example that has additional layers is LogicalDeviceManagerImpl.

Specifying the <managedBy> attribute for an entity in the metadata allows the entity manager to override the default behavior of the following methods:

- TransitionManager.transition(LifeCycleManaged, Object)
- PersistenceManagerBean.completeCreate(Persistent)
- PersistenceManagerBean.completeUpdate(Persistent)
- PersistenceManagerBean.completeDelete(Persistent)

PersistenceManagerBean

Package: oracle.communications.platform.persistence.impl

PersistenceManagerBean is the common base class for all entity managers. It provides convenient create, read, update, and delete (CRUD) methods for managing entity persistence. It also provides methods to attach an object to the persistence engine, and methods to test for object equality. Developing entity managers requires the use of the PersistenceManagerBean class. It defines all the persistence-related methods used by entity managers, it hides the JPA standard PersistenceManager, and it wraps the persistence logic required.

TransitionManagerImpl

Package: oracle.communications.inventory.api.common.impl

TransitionManagerImpl transitions an entity's business and object states, which is only applicable for entities defined as life-cycle managed in the metadata. This layer of inheritance is always in place, but it is used only by life-cycle managed entities. See Chapter 5, "Extending Life Cycles" for more information.

BaseInvManager

Package: oracle.communications.inventory.api.common

BaseInvManager extends PersistenceManagerBean and provides application-specific logic to the PersistenceManagerBean methods. All entity manager classes must extend this class.

JPQL Query Examples

This section provides some JPQL query examples that show common UIM search scenarios.

Example 3–3 Custom Object Search

```
// This example shows a search for a custom object based on the name of the custom
// object.

SELECT COUNT(DISTINCT o) FROM CustomObject o
WHERE UPPER(o.name) LIKE UPPER(:nameParam) escape '\'
AND o.specification = :specParam
AND (o.objectState = oracle.communications.inventory.api.ObjectState.ACTIVE
OR o.objectState = oracle.communications.inventory.api.ObjectState.INACTIVE)
```

Example 3–4 Physical Port Search

```
// This example shows a search for a physical port based on physical device name
// and physical port specification name.

SELECT COUNT(DISTINCT o) FROM PhysicalPort o
WHERE o.id is not null
AND o.specification = :specParam
AND UPPER(o.physicalDevice.name) LIKE UPPER(:pdNameParam) escape '\'
AND (o.objectState = oracle.communications.inventory.api.ObjectState.ACTIVE
OR o.objectState = oracle.communications.inventory.api.ObjectState.INACTIVE)
```

Example 3–5 Physical Device Search

```
// This example shows a search for a physical device based on a characteristic.

SELECT COUNT(DISTINCT o) FROM PhysicalDevice o
JOIN o.characteristics chars0var
WHERE o.id is not null
AND (chars0var.name = :pCharName0Param
AND  UPPER(chars0var.value) LIKE :pCharValue0Param  escape '\' )
AND o.specification = :specParam
AND (o.objectState = oracle.communications.inventory.api.ObjectState.ACTIVE OR
o.objectState = oracle.communications.inventory.api.ObjectState.INACTIVE)
```

4

Extending the Data Model

This chapter provides information on how to extend the Oracle Communications Unified Inventory Management (UIM) data model through additions to the metadata. The information describes statically extending the UIM data model, which can result in backward compatibility issues. See "Backward Compatibility" for the implications regarding this type of extension. Another option is to dynamically extend UIM through characteristics. For information about characteristics, see *UIM Concepts*.

About the UIM Data Model

The UIM data model extends the Oracle Communications Information Model (Information Model). The Information Model is shared by several Oracle Communications products, including UIM. The data model for each product is defined by a collection of XML and XSD files called metadata. Some metadata files are defined by the Information Model, and some metadata files are defined by the product. Regardless of their origin, all metadata files are part of the product installation.

Metadata files define:

- Tables and columns that comprise the UIM database
- Entities and attributes that correspond to the tables and columns
- Enumerated data
- Life-cycle state transition data
- Native sequences
- Tags that govern the definition of an entity, entity manager, enumeration, and native sequence

The metadata files, in conjunction with UIM-provided Ant targets, are used to regenerate the database tables and the corresponding entity Java source files. Another UIM-provided Ant target compiles the entity source files into entity class files and rebuilds the **inventory.ear** file to include the entity class files.

> **Note:** The generated entities and compiled source files reside in the **inventory.ear** file upon installation. The Ant targets for regeneration and compilation are needed only if you statically extend the data model.

About Entities

Entities are Java representations of UIM data and are used to persist data in the database. For example, in the UIM database, the TelephoneNumber table defines several columns of data including ID, name, and description, each of which are defined with a data type of String. For each table, there is corresponding Java entity class, such as TelephoneNumber.class, that is compiled from a Java source file, such as TelephoneNumber.java. Each source file defines data attributes with the same names and same data types as the data columns defined for the corresponding table. Each row in the TelephoneNumber table is persisted by an instance of TelephoneNumber.class. The TelephoneNumber table name, column names, and data types correspond directly to TelephoneNumber.java attributes and data types because both are generated from the same entity definition in the metadata.

About Entity Capabilities

A capability is a design pattern that is applied to an entity, such as enabling an entity to be life- cycle managed. For example, an entity that is life-cycle managed progresses through a succession of states during the course of its life. For life-cycle managed entities, UIM tracks two states: administrative state and object state. To support this capability, an entity must define the **adminState** and **objectState** attributes. Rather than define these attributes for every entity that supports this pattern, the capability is declared as part of the entity definition. As a result of this declaration, the **adminState** and **objectState** attributes are generated for the entity. So, a capability that is declared in an entity definition can result in the generation of attributes, and also the generation of any related entities that support the capability, neither of which are explicitly defined in metadata.

When extending the data model, you can extend existing entities to declare capabilities, or you can create new entities that declare capabilities. See "Understanding Entity Capability Definitions" for more information.

About Entity Relationships

Entity relationships describe how an entity relates to other entities. Entity relationships can be defined as one-to-one, one-to-many, many-to-one, or many-to-many. An entity definition can specify a relationship to an explicitly-defined entity, or to a capability-generated entity.

When extending the data model, you can extend existing entities to define additional relationships, or you can create new entities that define relationships to other entities. See "Understanding Entity Relationship and Collection Definitions" for more information.

About Entity Managers

Entity managers are Java classes that manage a specified set of database tables for a specific functional area. For example, the EquipmentManager class manages the Equipment table, but it also manages other tables in the equipment functional area such as EquipmentHolder, PhysicalPort, PhysicalConnector, and PhysicalDevice.

Entity manager class files are part of UIM and work with the Persistence Framework in managing the UIM database. The metadata defines entity manager interfaces, citing existing entity manager classes in the definition.

When extending the data model, you can extend existing entities to be managed by a specific entity manager, or you can create new entities which requires the creation of new entity managers. See "Understanding Entity Manager Definitions" for more information.

About Entity ID Sequencing

The Oracle database provides a mechanism for obtaining a generated unique number known as a sequence. This mechanism is called an Oracle native sequence. Each UIM entity defines the **entityId** attribute, which the persistence framework uses to uniquely identify an object. In previous releases, the **entityId** attribute value was set using just one Oracle native sequence, resulting in the value being a unique number across the entire database. However, this scenario does not provide for optimal processing performance.

To improve processing performance, UIM now defines several additional Oracle native sequences. Each native sequence is given a sequence generator name that is based on a functional area, such as ConnectivitySeqGen, EquipmentSeqGen, and TelephoneNumberSeqGen. The native sequences and corresponding sequence generator names are defined in the metadata, and an entity definition may specify a sequence generator, indicating the native sequence that the entity is to use when setting the **entityId** attribute value for the entity. For example, the Pipe, PipeTerminiationPoint, and PipeRel entity definitions specify the ConnectivitySeqGen sequence generator. In this scenario, the entityId values for the entities that use a specific native sequence are unique; entityId values are not unique across the entire database.

To keep this information centrally located, all native sequences and their corresponding sequence generator names are defined in the same type of file. Additionally, all entity definitions that specify a sequence generator are extended to specify it in the same type of file. Entity definitions that do not specify a sequence generator name use the default native sequence provided by the database.

Depending on your implementation of UIM, you may determine that you have heavily-used entities that, upon installation, use the default database native sequence. You can extend the data model by extending your heavily-used entity definitions to specify one of the UIM-defined sequence generators. You can also define your own native sequences and corresponding sequence generator names in the metadata, and extend your heavily-used entity definitions to specify one of your new sequence generators. See "Understanding Native Sequence Definitions" for more information.

> **Note:** The remainder of this chapter refers only to entities and attributes, rather than to tables and columns and corresponding entities and attributes, all of which are generated from entity definitions in the metadata.

About the Metadata Files

The metadata files are contained in the *UIM_Home*/**cartridges/tools/ora_uim_entity_sdk_cartproj.zip** file. This ZIP file is also located in the UIM SDK. Within the ZIP file, the metadata files are located in the **src/uim_poms_lib.jar** file unless otherwise noted.

The metadata files include:

ocim-*.*

File names that start with **ocim-** indicate that the file is part of the Information Model. These files are common to several Oracle Communications products, including UIM.

uim-*.*

File names that start with **uim-** indicate that the file defines UIM-specific entities, entity attributes, entity managers, enumerations, and transitions.

*-entities.xml

File names that end with **-entities.xml** indicate that the file defines entities for a specific area, such as service, equipment, or connectivity. Entities are defined through XML tags that are governed by the **package.xsd** file and the ***-plugin.xsd** files, which are described below. Any tags used in an entity definition are a subset of the tags defined in the **package.xsd** file and the ***-plugin.xsd** files.

In addition to defining entities, ***-entities.xml** files also define entity managers, enumerations, and native sequences. The ***-entities.xml** file content is further explored in "Understanding Entity Definitions".

*-types.xsd

File names that end with **-types.xsd** indicate that the file defines entity attributes (name and data type), or inherits entity attributes from a specified entity. For example, the Equipment entity defines several attributes including **id**, **name**, and **description**, all of which are defined as String. In another example, the EquipmentRole entity does not define any attributes; rather, it inherits all of InventoryRole entity attributes. The ***-types.xsd** file content is further explored in "Understanding Entity Attribute Definitions".

-enum-.*

File names that contain **-enum-** indicate that the file defines either enumeration types or enumeration values. The ***-enum-*.*** file content is further explored in "Understanding Enumeration Definitions".

*-entityidsequenceextension-entities.xml

File names that end with **-entityidsequenceextension-enitities.xml** indicate that the file defines native sequences and corresponding sequence generator names, and that it extends both explicitly-defined and capability-generated entities by specifying a sequence generator for the entity to use. The **-entityidsequenceextension-enitities.xml** file content is further explored in "Understanding Native Sequence Definitions".

*-transitions.xml

File names that end with **-transitions.xml** indicate that the file defines life-cycle state transitions. The ***-transitions.xml** file content is further explored in Chapter 5, "Extending Life Cycles".

*-plugin.xsd

File names that end with **-plugin.xsd** indicate that the file defines XML tags that govern definitions in the ***-entities.xml** files. The ***-plugin.xsd** files include:

- **uim-plugin.xsd**
- **core-plugin.xsd**
- *capability*-**plugin.xsd** files, where *capability* represents a specific capability such as capacity, characteristic, or consumable

The file content of the ***-plugin.xsd** files is similar to the **package.xsd** file content. See "package.xsd" for more information.

The **uim-plugin.xsd** resides in the **src/ora_uim_poms.jar** file. The **core-plugin.xsd** file resides in the **src/platformFiles/poms/core_poms_lib.jar** file.

The *capability*-**plugin.xsd** files reside in the correspondingly named **src/platformFiles/poms/***capability*_**poms_lib.jar** files, where *capability* represents a specific capability such as capacity, characteristic, or consumable.

*-libs.xml

The *capacity*-**caps-libs.xml** and *capacity*-**model-libs.xml** files are internal files that support the modularity of the *capability*-**plugin.xsd** files.

The ***-libs.xml** files reside in the correspondingly named **src/platformFiles/poms/***capability*_**poms_lib.jar** files, where *capability* represents a specific capability such as capacity, characteristic, or consumable.

package.xsd

The **package.xsd** file, and the ***-plugin.xsd** files, defines XML tags that govern definitions in the ***-entities.xml** files. For example, <entity>, <implements>, and <relationship> are XML tags used to define an entity, to specify an interface that the entity implements, and to define the entity's relationship to other entities.

The XML tags defined in these files are enforced by the build that generates the database and entities. If an XML tag is added to an ***-entities.xml** file that is not defined in the **package.xsd** file or in a ***-plugin.xsd** file, the build fails with an error citing the invalid XML tag.

There is only **package.xsd** file and copy of the file resides in each of the **src/platformFiles/poms/***capability*_**poms_lib.jar** files, where *capability* represents a specific capability such as capacity, characteristic, or consumable.

Note: Platform is the base code upon which all Oracle Communications products are built. Platform provides common code used by all Oracle Communications products, including the code that generates and builds the each product's database and entities using the metadata files.

XMLSchema.xsd

The **XMLSchema.xsd** file defines all the XML tags for the World Wide Web Consortium (W3C). The W3C is the main international standards organization for the World Wide Web.

The **XMLSchema.xsd** file is industry-standard specific.

Understanding Metadata File Content

The metadata file content defines:

- Entities
- Entity attributes
- Enumerations
- Native sequences
- XML tags that govern the definition of an entity, entity manager, enumeration, and native sequence

Understanding Entity Definitions

Entity definitions result in the creation of database tables and corresponding entity source files, which are compiled into entity class files. Entity classes are used to persist data in the database, and each entity class instance mirrors a unique database record in a table.

*-entities.xml Files

Entities are defined by XML elements and attributes that identify various properties of the entity. An entity definition can reside in an **ocim-*-entities.xml** file, in a **uim-*-entities.xml** file, or in both files. When an entity definition resides in both files, the UIM portion of the definition extends from the Information Model portion of the definition. For the UIM data model, most entities are defined by both files. There are a handful of entities that are UIM-specific, in which case the entity definition resides only in the **uim-*-entities.xml** file.

Example of an Entity Defined by Both Files

In the UIM data model, most entity definitions reside in an **ocim-*-entities.xml** file, with the entity definition extended in a **uim-*-entities.xml** file, as shown in the following examples. Example 4–1 is an excerpt from the **ocim-number-entities.xml** file that shows the TelephoneNumber entity definition. The entity definition includes any interfaces the entity implements, any capabilities for which the entity is enabled, and any relationships that the entity has to other entities.

Example 4–1 Entity Definition

```
<entity type="ocim:TelephoneNumber"
    interface="oracle.communications.inventory.api.entity.TelephoneNumber"
    accessControlled="true" entityIdSequenceGenerator="TelephoneNumberSeqGen">
    <implements interface=
        "oracle.communications.inventory.api.entity.common.NetworkAddress"/>
    <attribute name="id" index="true"/>
    <attribute name="name" index="true"/>
    <!-- ********** Capabilities **********-->
    <lifecycle stateType="ocim:InventoryState"/>
    <consumable prefix="TN" attribute="telephoneNumber"
        assignmentStateType="ocim:AssignmentState">
        <consumer name="ocim:Service" ConfigurationItemEnabled="true"/>
    </consumable>
    <referenceEnabled prefix="TelephoneNumber" attribute="telephoneNumber"/>
    <characteristic spec="ocim:CharacteristicSpecification">
    <characteristicName name="ocim:TNCharacteristic"
        interface=
            "oracle.communications.inventory.api.entity.TNCharacteristic"
            table="TN_CHAR"/>
    </characteristic>
    <businessInteractionEnabled history="true" visibilityState="SHOW"/>
    <groupEnabled/>
    <!-- ********** Relationships **********-->
    <!-- One-Sided Many-to-One TelephoneNumber to TelephoneNumberSpec -->
    <relationship name="specification">
        <otherSide type="ocim:TelephoneNumberSpecification"/>
    </relationship>
</entity>
```

Example 4–2 is an excerpt from the **uim-number-entities.xml** file. The example shows the TelephoneNumber entity definition that extends the TelephoneNumber entity definition from the **ocim-number-entites.xml** file. In this example, the entity declares an entity manager through the **managedBy** tag because the entity is business-interaction enabled, as defined in the **ocim-number-entities.xml** file. Any methods that an entity needs to implement are also defined in the UIM portion of the entity definition.

Example 4–2 Extended Entity Definition

```
<entity type="ocim:TelephoneNumber"
    managedBy="oracle.communications.inventory.api.number.TelephoneNumberManager">
    <method name="getDisplayInfo">
        <signature><![CDATA[String getDisplayInfo()]]></signature>
        <body><![CDATA[
            return getName();]]>
        </body>
        <javadoc>Return an identifiable info String for this resource.</javadoc>
    </method>
</entity>
```

Example of a UIM-Specific Entity Definition

Very few entities are defined by a **uim-*-entities.xml** file only. One such example is the **uim-rule-entities.xml** file. Rulesets are UIM-specific functionality, so the Information Model does not define any ruleset entities.

Example 4–3 an is an excerpt from the **uim-rule-entities.xml** file. The file defines four entities that deal with rulesets. The excerpt shows the ExtensionPoint entity definition, which includes any interfaces the entity implements, any capabilities for which the entity is enabled, and any relationships that the entity has to other entities.

Example 4–3 Entity Definition

```
<entity type="ocim:ExtensionPoint"
    interface="oracle.communications.inventory.api.entity.ExtensionPoint">
    <!-- Two-Sided One-to-Many ExtensionPoint to EnabledExtensionPoint -->
    <relationship name="enabledExtensionPoints">
        <thisSide inverse="true" collection="java.util.HashSet"/>
        <otherSide dependent="true" type="ocim:EnabledExtensionPoint"/>
    </relationship>
    <!-- Two-Sided One-to-Many ExtensionPoint to ExtensionPointRuleSet -->
    <relationship name="extensionPointRuleSets">
        <thisSide inverse="true" collection="java.util.HashSet"/>
        <otherSide dependent="true" type="ocim:ExtensionPointRuleSet"/>
    </relationship>
</entity>
```

More on Entity Definitions

Entity definitions include capability and relationship declarations. See "Understanding Entity Capability Definitions" and "Understanding Entity Relationship and Collection Definitions" for more information.

The ***-entities.xml** files also define entity managers. See "Understanding Entity Manager Definitions" for more information.

Understanding Entity Attribute Definitions

Each entity defines a set of attributes in which to store data. Entity attribute definitions result in the creation of database table columns and corresponding entity attributes in source files, which are compiled into entity class files. Entity classes are used to persist data in the database, and each entity class instance mirrors a unique database record in a table.

*-types.xsd Files

Entity attributes are defined by XML elements and attributes that define an attribute's name and data type. Entity attribute definitions reside in ***-types.xsd** files. Each

***-entities.xml** file has a corresponding ***-types.xsd** file. For example, **ocim-number-entities.xml** and **ocim-number-types.xsd**, or **uim-rule-entities.xml** and **uim-rule-types.xsd**. For each set of corresponding files:

- The ***-entities.xml** files define entities and entity managers
- The ***-types.xsd** files define entity attributes (name and data type)

Example 4–4 is an excerpt from the **ocim-number-types.xsd** file that defines the entity attributes (name and data type) for the TelephoneNumber entity. The attribute names defined in the excerpt are **id**, **name**, and **description**, and all of the attribute data types are defined as string.

Example 4–4 Entity Attributes Definition

```
<xs:complexType name="TelephoneNumber">
    <xs:sequence>
        <xs:element name="id" type="xs:string">
        </xs:element>
        <xs:element name="name" type="xs:string">
        </xs:element>
        <xs:element name="description" type="xs:string">
        </xs:element>
        .
        .
        .
    </xs:sequence>
</xs:complexType>
```

Understanding Enumeration Definitions

Enumeration definitions result in attributes that are defined with an enumeration type being limited to storing only previously defined data values. Enumerations are used to regulate data upon which code is based. For example, code can be written to handle a finite number of scenarios based on a finite number of defined enumeration values.

The following ***-enum-*.*** files are in the metadata:

- ocim-enum-entities.xml
- uim-enum-entities.xml
- ocim-enum-types.xsd
- uim-enum-types.xsd

*-enum-entities.xml Files

Enumeration types are defined in the **ocim-enum-entities.xml** and **uim-enum-entities.xml** files. Example 4–5 is an excerpt from the **uim-enum-entities.xml** file that shows the definition of two enumeration types: BusinessInteractionState and BusinessInteractionAction.

Example 4–5 Enumeration Type Definition

```
<enum type="ocim:BusinessInteractionState"
enumType="oracle.communications.inventory.api.entity.BusinessInteractionState"
adminState="true"/>
<enum type="ocim:BusinessInteractionAction"
enumType="oracle.communications.inventory.api.entity.BusinessInteractionAction"/>
```

*-enum-types.xsd Files

Enumeration values are defined in the **ocim-enum-types.xsd** and **uim-enum-types.xsd** files. Example 4–6 is an excerpt from the **ocim-enum-types.xsd** file that shows the definition of two sets of enumeration values. The **ocim-enum-entities.xml** file and the **ocim-enum-types.xsd** file both show excerpts of BusinessInteractionState and BusinessInteractionAction. The **ocim-enum-entites.xml** file defines the enumeration type, and the **ocim-enum-types.xsd** file defines the enumeration values that are valid for each enumeration type.

Example 4–6 Enumeration Value Definition

```
<xs:simpleType name="BusinessInteractionState">
    <xs:restriction base="xs:string">
        <xs:enumeration value="CANCELLED"/>
        <xs:enumeration value="COMPLETED"/>
        <xs:enumeration value="CREATED"/>
        <xs:enumeration value="IN_PROGRESS"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="BusinessInteractionAction">
    <xs:restriction base="xs:string">
        <xs:enumeration value="CANCEL"/>
        <xs:enumeration value="COMPLETE"/>
        <xs:enumeration value="PROCESS"/>
        <xs:enumeration value="TRANSFER"/>
    </xs:restriction>
</xs:simpleType>
```

Understanding Native Sequence Definitions

Several native sequences and their corresponding sequence generator names are defined in the metadata. Native sequence definitions result in the creation of the native sequence in the database. Several entity definitions are extended to specify a sequence generator, which results in the corresponding native sequence being used to set the **entityId** attribute value for the entity. The native sequence definitions, their corresponding sequence generator names, and the entity definitions that are extended to specify a sequence generator, reside in the **ocim-entityidsequenceextension-entities.xml** file.

> **Note:** Entity definitions that do not specify a native sequence use the default native sequence provided by Platform.

ocim-entityidsequenceextension-entities.xml File

Example 4–7 is an excerpt from the **ocim-entityidsequenceextension-entities.xml** file that shows the definition of three native sequences:

- ENTITYID_CONNECTIVITY_SEQ
- ENTITYID_EQUIPMENT_SEQ
- ENTITYID_TN_SEQ

The corresponding sequence generator names given to these native sequences are:

- ConnectivitySeqGen
- EquipmentSeqGen
- TelephoneNumberSeqGen

Example 4–7 Native Sequence Definition

```
<entityIdSequenceGenerator name="ConnectivitySeqGen"
    sequence="ENTITYID_CONNECTIVITY_SEQ" />
<entityIdSequenceGenerator name="EquipmentSeqGen"
    sequence="ENTITYID_EQUIPMENT_SEQ"  />
<entityIdSequenceGenerator name="TelephoneNumberSeqGen"
    sequence="ENTITYID_TN_SEQ" />
```

Example 4–8 is an excerpt from the **ocim-entityidsequenceextension-entities.xml** file that shows the extended definition of three entities to include a sequence generator. In this example, the Pipe, PipeTerminationPoint, and PipeRel entities all specify the ConnectivitySeqGen sequence generator, which results in these entities using the corresponding ENTITYID_CONNECTIVITY_SEQ native sequence to set their respective **entityId** attribute values.

Example 4–8 Extended Entity Definition

```
<entity type="ocim:Pipe"
    entityIdSequenceGenerator="ConnectivitySeqGen" extension="true"/>
<entity type="ocim:PipeTerminationPoint"
    entityIdSequenceGenerator="ConnectivitySeqGen" extension="true"/>
<entity type="ocim:PipeRel"
    entityIdSequenceGenerator="ConnectivitySeqGen" extension="true"/>
```

Note: See the Persistent Pattern chapter in *Oracle Communications Information Model Reference* for more information, including a list of entity-to-sequence mappings.

Understanding the Tags that Govern Definitions

The **package.xsd** and ***-plugin.xsd** files define the tags that govern definitions of entities, entity managers, enumerations, and native sequences in ***-entities.xml** files.

This section introduces the **package.xsd** and ***-plugin.xsd** file content to help you better understand the content when you are viewing it. This section also explains the entity definition's use of the tags to help you better understand how the tags correlate to an entity definition.

Note: This section does not explain the functionality of the governing tags; it explains where the governing tags are defined and how they are used within the ***-entities.xml** files.

For information on the functionality that the tags provide, see:

- The documentation for each tag within the **package.xsd** and ***-plugin.xsd** files
- *Oracle Communications Information Model Reference*

The **package.xsd** defines the following complexType elements:

- entity
- manager
- enum
- entityIdSequenceGenerator

The <entity> element defines several elements, and some of the elements define attributes. For example, the <entity> element defines the <import>, <implements>, <attribute>, <relationship>, and <method> elements, as well as several other elements. The <relationship> element defines the **join**, **thisSide**, and **otherSide** attributes. The <entity> element also defines several attributes directly (as opposed to the attributes being defined for an element). For example, the <entity> element defines the **interface** and **managedBy** attributes. Within the file, each complexType, element, and attribute is described.

In a similar fashion, the ***-plugin.xsd** files also define tags that are used in the ***-entities.xml** files. For example, the *capability*-***-plugin.xsd** files (where *capability* represents a specific capability such as capacity, characteristic, and consumable) only define tags that are used in an entity definition to declare a particular capability for an entity.

Example 4–9 shows the TelephoneNumber entity definition. The example is numbered so that the information describing the example can be referenced.

Example 4–9 Entity Definition

```
01  <entity type="ocim:TelephoneNumber"
02      interface="oracle.communications.inventory.api.entity.TelephoneNumber"
03      accessControlled="true" >
04      <implements interface=
05      "oracle.communications.inventory.api.entity.common.NetworkAddress"/>
06      <attribute name="id" index="true"/>
07      <attribute name="name" index="true"/>
08      <!-- ***************** Capabilities ******************-->
09      <lifecycle stateType="ocim:InventoryState"/>
10      <consumable prefix="TN" attribute="telephoneNumber"
11          assignmentStateType="ocim:AssignmentState">
12          <consumer name="ocim:Service" ConfigurationItemEnabled="true"/>
13      </consumable>
14      <referenceEnabled prefix="TelephoneNumber" attribute="telephoneNumber"/>
15      <characteristic spec="ocim:CharacteristicSpecification">
16          <characteristicName name="ocim:TNCharacteristic"
17          interface=
18          "oracle.communications.inventory.api.entity.TNCharacteristic"
19          table="TN_CHAR"/>
20      </characteristic>
21      <businessInteractionEnabled history="true" visibilityState="SHOW"/>
22      <groupEnabled/>
23      <!-- ***************** Relationships ******************-->
24      <!-- One-Sided Many-to-One TelephoneNumber to TelephoneNumberSpec -->
25      <relationship name="specification">
26          <otherSide type="ocim:TelephoneNumberSpecification"/>
27      </relationship>
28   </entity>
```

Lines 01 through 07 define the entity with various tags. For example:

- Line 01 uses <entity> and **type**; <entity> is defined as a complexType in **package.xsd**, and **type** is defined as an attribute of the <entity> element in **package.xsd**.
- Line 02 uses **interface**, which is defined as an attribute of the <entity> element in **package.xsd**.
- Line 03 uses **accessControlled**, which is defined in the **core-plugin.xsd** file.

- Lines 04 and 05 use <implements> and **interface**; <implements> is defined as an element of the <entity> element in **package.xsd**, and **interface** is defined as an attribute of the <implements> element in **package.xsd**.
- Lines 06 and 07 use <attribute> and **name**; <attribute> is defined as a complexType in **package.xsd**, and **name** is defined as an attribute of the <attribute> element in **package.xsd**.

Lines 08 through 22 continue the entity definition by defining the entity's capabilities. For example:

- Line 09 uses <lifecycle> and **stateType**, both of which are defined in **uim-plugin.xsd**.
- Lines 10 through 13 uses several tags that are defined in **consumable-plugin.xsd**.
- Line 14 uses <referenceEnabled>, which is defined in **uim-plugin.xsd**.
- Lines 15 through 20 uses several tags that are defined in **characteristic-pluginx.xsd**.
- Line 21 uses <businessInteractionEnabled> and **visibilityState**, both of which are defined in **uim-plugin.xsd**.
- Line 22 uses <groupEnabled>, which is defined in **groupenabled-plugin.xsd**.

Lines 23 through 27 continue the entity definition by defining the entity's relationships to other entities. For example, <relationship>, **name**, <otherSide>, and **type** are all defined in **package.xsd**.

Extending the Data Model Through the Metadata Files

You extend the data model by creating new metadata files.

Caution: Do not make modifications to the existing metadata files. See "Backward Compatibility" for the issues involved with modifying the existing metadata files.

You use Oracle Communications Design Studio to create new metadata files by importing the **ora_uim_entity_sdk_cartproj.zip** file, and creating new XML or XSD files within the imported project. Any new metadata files you create must reside in the **ora_uim_entity_sdk** project, within the **src** directory.

When you define new entities and attributes, or extend existing entities and attributes, the changes are picked up by the Ant target that generates the database and the corresponding entity Java source files. For example, if you add the new entity **myNewEntity**, **myNewEntity** is generated as a new table in the database, and **MyNewEntity** is generated as an entity Java source file. If you add **myNewAttribute** to an existing entity, **myNewAttribute** is generated as a new column on the existing table in the database, and **myNewAttribute** is generated as an attribute within the generated entity Java source file.

The following sections describe extending the data model through the creation of new metadata files.

Defining New Entities

When defining new entities, look at existing ***-entities.xml** files for examples of how to define various entity properties. The XML tags you use to define a new entity are

governed by the **package.xsd** and ***-plugin.xsd** files. Be sure to include any referenced schemas in the package statement.

To define a new entity:

1. Create a new XML file.

   The file name must end with **-entities.xml**. For example, **myNewFile-entities.xml**.

2. Open an existing ***-entities.xml** file.
3. Copy and paste an entity definition from the existing file to your new file.
4. Modify the copied entity definition as needed:
   a. Change the name of the entity to reflect the name of your new entity.
   b. Remove or update the tags to reflect the definition of your new entity.
5. Write an entity manager that defines the interfaces to manage the new entity. See "Creating New Entity Managers".
6. Write an entity manager implementation that inherits from BaseInvManager and defines the methods to manage the new entity. See "Creating New Entity Managers".
7. Include the entity manager interface definition in your new ***-entities.xml** file.

Creating New Entity Managers

When creating new entity managers, the entity managers:

- Should provide coarse-grained methods that may involve other entity managers
- Assume that the caller is managing transaction boundaries
- Must be developed so that they are stateless and thread safe so they can be exposed to web service calls
- Should avoid creating duplicate records using makePersistent(). To avoid this, call the connect() method before you set entities to the transient state. See "PersistenceManager refresh(), attach(), and connect() Methods" for more information.

See Chapter 3, "Using the Persistence Framework" for more information on entity managers.

Defining New Entity Attributes

When defining new entity attributes, look at the existing ***-types.xsd** files for examples of how to define various attributes. The XML tags you use to define new entity attributes are governed by the **XMLSchema.xsd** file. Be sure to include any referenced schemas in the package statement.

XSD is an industry standard. For information about writing XSD, see the W3C website:

`http://www.w3.org/XML/Schema.html`

To define attributes for a new entity, or to add new attributes to an existing entity:

1. Create a new XSD file.

   The file name must end with **-types.xsd**. For example, **myNewFile-types.xsd**.

2. Open an existing ***-types.xsd** file.
3. Copy and paste an entity attribute definition from the existing file to your new file.

4. If defining attributes for a new entity, modify the copied entity attributes definition as needed:
   a. Change the entity name to reflect the new entity name.
   b. Change the attribute names to reflect the new attribute names.
   c. Change the attribute types to reflect the new attribute types.
5. If adding new attributes to an existing entity, modify the copied entity attributes definition as needed:
   a. Change the entity name to reflect the entity name that defines the attributes to which you are adding new attributes.
   b. Change the attribute names to reflect the new attribute names you are adding.
   c. Change the attribute types to reflect the new attribute types you are adding.

Defining New Enumerations

When defining new enumeration types, look at the existing ***-enum-entities.xml** files for examples of how to define them; when defining new enumeration values, look at the existing ***-enum-types.xsd** files for examples of how to define them. The XML tags you use to define new enumerations are governed by the **XMLSchema.xsd** file. Be sure to include any referenced schemas in the package statement.

Enumerations are an industry standard. For information about writing enumerations, see the W3C Web Services Enumeration website at:

http://www.w3.org/Submission/WS-Enumeration/

You can place all new enumeration types in one new file, and all new sets of enumeration values in another new file.

To define a new enumeration type:

1. Create a new XML file.

   The file name must contain **-enum-** and end with **-entities.xml**. For example, **myNew-enum-entities.xml**.
2. Open an existing ***-enum-entities.xml** file.
3. Copy and paste an enumeration type definition from the existing file to your new file.
4. Modify the copied enumeration type definition as needed.

To define enumeration values for a new enumeration type, or to add new enumeration values to an existing enumeration type:

1. Create a new XSD file.

   The file name must contain **-enum-** and end with **-types.xsd**. For example, **myNewFile-types.xsd**.
2. Open an existing ***-enum-types.xsd** file.
3. Copy and paste a set of enumeration values from the existing file to your new file.
4. If defining new enumeration values for a new enumeration type, modify the copied enumeration values as needed:
   a. Change the enumeration type to reflect your new enumeration type.
   b. Change the data type to reflect the data type of your new enumeration values.

c. Change the enumeration values to reflect your new enumeration values.

5. If adding new enumeration values to an existing enumeration type, modify the copied enumeration values as needed:

   a. Change the enumeration type to reflect the name of the existing enumeration type to which you are adding the new enumeration values.

   b. Change the data type to reflect the same data type as defined by the enumeration type to which you are adding the new enumeration values.

   c. Change the enumeration values to reflect the new enumeration values you are adding to the enumeration type.

Defining New Native Sequences

When defining new native sequences, look at the existing **ocim-entityidsequenceextension-entities.xml** file for examples of how to define a native sequence and corresponding sequence generator name. Be sure to include any referenced schemas in the package statement.

To define a new native sequence, and specify an entity to use it:

1. Create a new XML file.

   The file name must end with **-entities.xml**. The file name should also contain a meaningful reference so you can readily recognize the file content, such as **-seqext-**. For example, **myNewSeqExts-entities.xml**.

2. Open the existing **ocim-entityidsequenceextension-entities.xml** file.

3. Copy and paste a native sequence definition from the existing file to your new file.

4. Modify the copied native sequence definition as needed:

   a. Change the name of the native sequence to reflect the name of your new native sequence.

   b. Change the corresponding sequence generator name to reflect a functional name for your new native sequence.

5. Copy and paste an extended entity definition from the existing file to your new file.

6. Modify the copied extended entity definition as needed:

   a. Change the entity name.

   b. Change the specified sequence generator to your new sequence generator.

Extending Existing Entities

You can extend an existing entity using the XML tags defined in the **package.xsd** and ***-plugin.xsd** files that enable an entity's use of framework functionality. For example, you can extend an entity to be life-cycle enabled, capacity enabled, business-interaction enabled, place enabled, group enabled, and so forth. Each of these are functional areas of UIM that become available to an entity through the entity's original definition, or through an extension and custom code.

There are two kinds of existing entities: Explicitly-defined entities and capability-generated entities. For example, the Equipment entity is explicitly defined in the metadata, and includes the declaration of the consumable capability. As a result of this declaration, the EquipmentConsumer entity is generated, even though the EquipmentConsumer entity is not explicitly defined in the metadata.

Explicitly-defined entities and capability-generated entities are extended the same way, with one slight difference: When extending a capability-generated entity, you must include the **extension** tag.

Understanding the Extension Tag

The **package.xsd** file defines tags that are used to govern entity definitions. One such tag is the **extension** tag. Since entities are defined in ***-entities.xml** files, the **extension** tag may only be used within an ***-entities.xml** file.

You can use the **extension** tag to extend:

- Capability-generated entities
- Generated attributes of an explicitly-defined entity
- Generated relationships of an explicitly-defined entity

For example, the following is an excerpt from the **ocim-entityidsequenceextension-entities.xml** file that extends the capability-generated EquipmentConsumer entity to use the ConsumerSeqGen native sequence:

```
<entity type="ocim:EquipmentConsumer"
    entityIdSequenceGenerator="ConsumerSeqGen" extension="true"/>
```

A similar extension in the same file for the explicitly-defined Equipment entity does not require the **extension** tag because the entity is not capability-generated:

```
 <entity type="ocim:Equipment"
    entityIdSequenceGenerator="EquipmentSeqGen"/>
```

In another example, the TelephoneNumber entity is explicitly defined in the metadata, and by default has the Trackable Pattern as described in *Oracle Communications Information Model Reference*. The Trackable Pattern generates the **createdDate**, **createdUser**, **lastModifiedDate**, and **lastModifiedUser** attributes. You can extend any of these generated attributes through the use of the **extension** tag. For example, the following adds an index to the **createdUser** generated attribute:

```
<entity type="ocim:TelephoneNumber" extension="true">
   <attribute name="createdUser" index="true"/>
</entity>
```

The TelephoneNumber entity explicitly defines several attributes, such as **id**, **name**, and **description**. So, a similar extension for an explicitly-defined attribute of TelephoneNumber does not require the **extension** tag because the attribute is not generated:

```
<entity type="ocim:TelephoneNumber">
   <attribute name="id" index="true"/>
</entity>
```

> **Note:** You cannot add the **extension** tag to the original entity definition; the tag must be placed in a separate entity definition that reflects the same name as the entity you are extending. The separate entity definition can reside in the same file as the original entity definition or in a separate file, but because Oracle recommends that you do not modify the metadata files directly, the separate entity definition should reside in a separate file.

Note: The **extension** tag that is defined by the **package.xsd** file should not be confused with the standard XSD **extension** tag, which is used to define entity attributes in the ***-types.xsd** files, or to define enumeration values in the ***-enum-types.xsd** files. For example, the **ocim-equipment-types.xsd** file defines the PipeRole entity to define the same attributes that the InventoryRole entity defines through the use of the standard XSD **extension** tag:

```
<xs:complexType name="PipeRole">
    <xs:complexContent>
        <xs:extension base="ocim:InventoryRole"/>
    </xs:complexContent>
</xs:complexType>
```

Extending Existing Entities

You can extend an existing entity through a new ***-entities.xml** file that defines the same entity name and includes additional properties for the entity. When regenerating the entities:

- For explicitly-defined entities, the properties defined for the entity in both files are merged together, resulting in the entity possessing the original properties and the extended properties.
- For capability-generated entities, the properties of the generated entities and any extended definitions are merged together, resulting in the entity possessing the original generated properties and the extended properties.

To extend an existing entity:

1. Create a new XML file.

   The file name must end with **-entities.xml**. For example, **myNewFile-entities.xml**.
2. Open an existing ***-entities.xml** file.
3. Copy and paste an entity definition from the existing file to your new file.
4. Modify the copied entity definition as needed:
   a. The entity name must be the same name as the entity you are extending.
   b. Add any tags to reflect the extension.
   c. If you are extending a capability-generated entity, include the **extension** tag.

Extending Existing Entity Attributes

Extending an existing entity attribute is done in an ***-entities.xml** file, not in the ***-types.xsd** file.

You can extend an existing attribute using the XML tags defined for attributes in the **package.xsd** file. For example, you can extend an attribute definition to have an index, to be encrypted, or to have a maximum length that the database stores for an attribute. See "Understanding the Tags that Govern Definitions" for more information.

There are two kinds of existing attributes: Explicitly-defined attributes and capability-generated attributes. For example, the Equipment entity and its attributes are explicitly defined in the metadata. The Equipment entity definition includes the declaration of the life cycle management capability. As a result of this declaration, the **adminState** and **objectState** attributes are generated for the Equipment entity, even though they are not explicitly defined in the metadata.

Explicitly-defined attributes and capability-generated attributes are extended the same way, with one slight difference: When extending a capability-generated attribute, you must include the **extension** tag. See "Understanding the Extension Tag" for more information.

You can extend an existing attribute through a new ***-entities.xml** file that defines the same entity name, and includes the attribute extension. When regenerating the entities:

- For explicitly-defined attributes, the properties defined for the attribute in both files are merged together, resulting in the attribute possessing the original properties and the extended properties.
- For capability-generated attributes, the properties of the generated attributes and any extended definitions are merged together, resulting in the attribute possessing the original generated properties and the extended properties.

To extend an existing attribute:

1. Create a new XML file.

   The file name must end with **-entities.xml**. For example, **myNewFile-entities.xml**.

2. Open an existing ***-entities.xml** file.
3. Copy and paste an entity definition from the existing file to your new file.
4. Modify the copied entity definition as needed:
   a. The entity name must be the same name as the entity that defines or generates the attribute you are extending.
   b. Update the tags to reflect the extension.
   c. If you are extending a capability-generated entity, include the **extension** tag.

Extending Existing Enumerations

You cannot extend existing enumerations. To clarify, you can add new enumeration values to an existing set of enumeration values. See "Defining New Enumerations" for more information.

Extending Existing Native Sequences

You can extend the use of any UIM-defined native sequences to include additional entities.

To extend the use of UIM-defined native sequences:

1. Create a new XML file.

   The file name must end with **-entities.xml**. The file name should also contain a meaningful reference so you can readily recognize the file content, such as **-seqext-**. For example, **myNewSeqExts-entities.xml**.

2. Open the existing **ocim-entityidsequenceextension-entities.xml** file.
3. Copy and paste an extended entity definition from the existing file to your new file.
4. Modify the copied extended entity definition as needed:
   a. Change the entity name.

b. Change the specified sequence generator to a UIM-defined sequence generator that you want the entity to use.

c. If the entity is a capability-generated entity, include the **extension** tag.

You can also modify existing UIM-defined native sequences.

Note: Oracle recommends that modifications to existing sequences be made before using UIM (before any sequence numbers have been generated).

To modify existing UIM-defined native sequences:

1. Create a new XML file.

   The file name must end with **-entities.xml**. The file name should also contain a meaningful reference so you can readily recognize the file content, such as **-seqext-**. For example, **myNewSeqExts-entities.xml**.

2. Open the existing **ocim-entityidsequenceextension-entities.xml** file.

3. Copy and paste an extended entity definition from the existing file to your new file.

4. In the copied file, modify the sequence definition as needed.

5. In the copied file, set the **entityIdSequenceGeneratorPriority** attribute to a value higher than the **entityIdSequenceGeneratorPriority** value in the original file so that the modified sequence overrides the original sequence.

6. If modifying the existing sequence after using UIM (after sequence numbers have been generated):

   a. Determine which entities use the modified sequence.

   b. For each entity using the sequence, determine the current maximum entityId value.

   c. For each entity using the sequence, extend the entity definition to set the **initialValue** attribute to a value 50 to 100 higher than the current maximum entityId value.

      See "Extending Existing Entities" and "Extending Existing Entity Attributes" for information on how to do this.

Applying Metadata Static Extensions

Statically extending the data model involves manually extending the metadata files. To apply the metadata static extensions:

- Generate the entity source files from the metadata files
- Compile the generated entity source files
- Package the compiled entity source files in the **inventory.ear, inventory-adapter.ear**, and **custom.ear** files.
- Deploy the **inventory.ear**, **inventory-adapter.ear**, and **custom.ear** files

You generate, compile, and package the entity source files using Ant targets that are provided in the **ora_uim_entity_sdk_cartproj.zip/src/build.xml** file.

About the build.xml File

The **build.xml** file defines several Ant targets that you can run to manage the database. An Ant target is a set of executable tasks that can be run using Ant. See "Running Ant Targets" for more information.

Note: If you are running these Ant targets on Linux/Solaris, set the following variables:

```
export COMPUTERNAME=hostName of Linux/Solaris machine
export ANT_HOME=path to Ant installation
```

Table 4–1 describes the Ant targets defined in the **build.xml** file.

Table 4–1 build.xml Ant Targets

Ant Target	Description
entities	From the metadata, this target generates entity source files and compiles them into entity Java classes. As a result, this target creates: ▪ The **api/build** directory, where all of the generated source files and compiled classes reside. This directory is placed in the **ora_uim_entity_sdk/src** directory. ▪ The **uim-entities.jar** file, which contains all of the entity classes. This file is placed in the **ora_uim_entity_sdk/src/generated/inventory/entities/APP-INF/lib** directory.
clean	This target deletes the **api/build** and **generated** directories created by the **entities** Ant target. Oracle recommends that you always run the **clean** target prior to running the **entities** target.
create.tables	This target creates new database tables based on any new entities defined in the metadata; this target does not alter existing tables that were created as part of the UIM installation. To connect to the database in which you are creating tables, set the database credentials in step 8 of the procedure to generate, compile, and package the entity source files. See "Generating, Compiling, and Packaging the Entity Source Files" for more information.
alter.tables	This target creates new database tables based on any new entities defined in the metadata.This target also alters existing database tables based on any new attributes defined for existing entities in the metadata. To connect to the database in which you are creating or altering tables, set database credentials in step 8 of the procedure to generate, compile, and package the entity source files. See "Generating, Compiling, and Packaging the Entity Source Files" for more information.
create.DDL	This target creates the **createDDL.jdbc** and **alterDDL.jdbc** scripts based on any new entities defined in the metadata, or any new attributes defined for existing entities in the metadata. These files are placed in the **ora_uim_entity_sdk/src/generated/entities/scripts** directory. Run these scripts to create new database tables or alter existing database tables. The **create.DDL** target is used in place of the **create.tables** and **alter.tables** targets.

Table 4–1 (Cont.) build.xml Ant Targets

Ant Target	Description
update.earwithEntities	This target updates the **inventory-adapter.ear** and **uim_core_lib.ear** files with the generated **uim-entities.jar** file.

Generating, Compiling, and Packaging the Entity Source Files

To generate, compile, and package the entity source files for deployment:

1. Configure your Design Studio environment. Define the following system variables:
   - JAVA_HOME

     See "Configuring Design Studio" for more information.
   - ANT_HOME

     See "Configuring Ant" for more information.
2. In Design Studio, open the Studio Design perspective.

   For instructions on how to open perspectives in Design Studio, see the Design Studio Help.
3. Import the *UIM_Home*/**cartridges/tools/ora_uim_entity_sdk_cartproj.zip** file.

   For instructions on how to import a project into Design Studio using archive files, see the Design Studio Help.
4. Change to the Java perspective.

   For instructions on how to change perspectives in Design Studio, see the Design Studio Help.
5. Copy and rename the **ora_uim_entity_sdk/etc/COMPUTERNAME.properties** file to *HOSTNAME*.**properties**, where *HOSTNAME* is the name of the computer where Design Studio is installed.

   You can determine the computer name by running the following DOS command:

   ```
   echo %COMPUTERNAME%
   ```
6. In the *HOSTNAME*.**properties** file, set the following properties to reflect your configuration:
   - UIM_HOME
   - DB_HOME
   - PROJECT_HOME
7. From a command line, navigate to your workspace **ora_uim_entity_sdk/src** directory, and run the following Ant command:

   ```
   ant -f build_extract_poms_zip.xml
   ```

   This command extracts the POMS SDK into the **ora_uim_entity_sdk/src/platformFiles/extract** directory.
8. In the **ora_uim_entity_sdk/src/platformFiles/extract/objectmgmt/poms/config/poms.properties** file, set the ConnectionUserName, ConnectionPassword, and ConnectionURL database credentials to reflect your database:

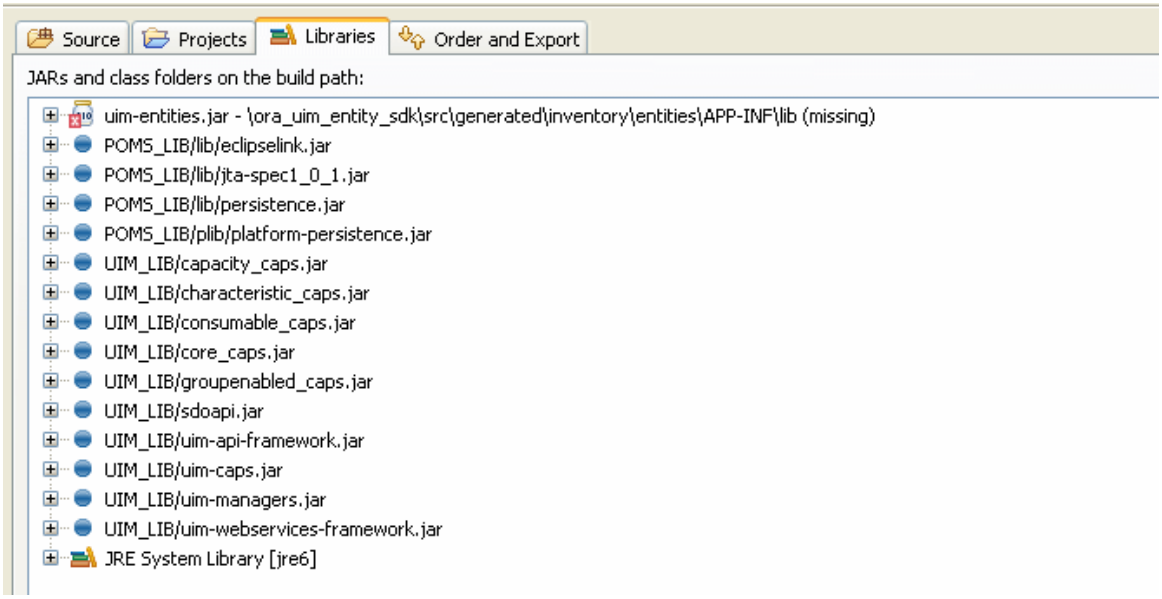
```
#Set datastore connection information for offline utilities.
poms.ConnectionDriverName = oracle.jdbc.OracleDriver
poms.ConnectionUserName = uimuser
poms.ConnectionPassword = welcome@123
poms.ConnectionURL = jdbc:oracle:thin:@localhost:1521:XE
```

9. Configure the project library list.

   For instructions on how configure the project library list, see the Design Studio Help.

   Figure 4–1 shows the imported project library list, which includes the JAR files needed to compile the project.

Figure 4–1 Project Library List Before Configuring



Note: The **uim-entities.jar** file gets created and placed in the specified directory by a later step in this procedure.

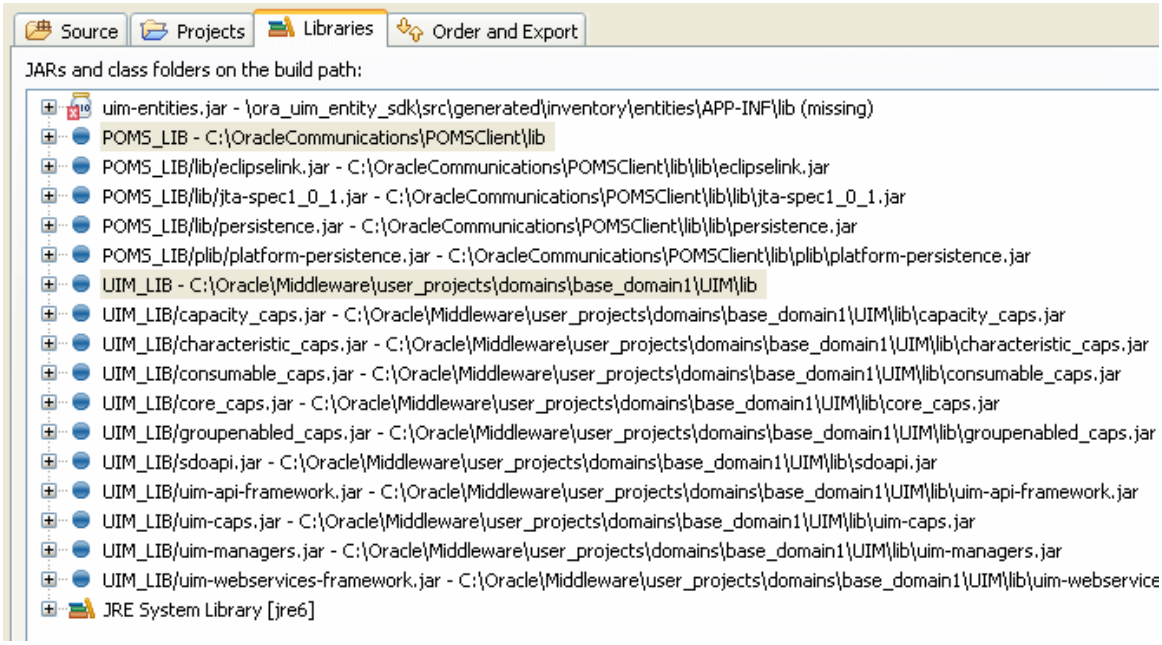
The project library list of JAR files does not indicate the location of the files, so you must configure the project library list to point to the location of the JAR files. To do this, you need to add new variables named POMS_LIB and UIM_LIB that point the specified directory, as listed in Table 4–2.

Table 4–2 Location of JAR Files

Variable Name	Directory Name
POMS_LIB	*Oracle_Home*/**POMSClient/lib**
UIM_LIB	*UIM_Home*/**lib**

Figure 4–2 shows the project library list after the variables are added. Notice that the library list now includes the location of the JAR files, not just the JAR file names.

Figure 4–2 Project Library List After Configuring



10. Add any new metadata files to the **ora_uim_src_entity/src/api** directory. See "Extending the Data Model Through the Metadata Files" for more information.

11. Modify the **ora_uim_src_entity/src/api/custom-model-lib.xml** file to include any new metadata files in the build. For example, if you have created new files named **my-entities.xml** and **my-entities.xsd**, you need to add these file names in the **custom-model-lib.xml** as follows:

Example 4–10 custom-model-lib.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<libs>
    <modellib>
        <id>
          <name>http://xmlns.oracle.com/communications/persistence/UimModel</name>
          <version>1.0</version>
        </id>
        <art>my-entities.xml</art>
        <art>my-types.xsd</art>
    </modellib>
</libs>
```

12. Add any custom Java code that supports new entities to the **ora_uim_src_entity/src/api/src-man** directory. For example, defining new entities in the metadata requires creating new entity managers. See "Creating New Entity Managers" for more information.

13. From a command line, navigate to your workspace **ora_uim_src_entity/src** directory, and run the following Ant command:

    ```
    ant entities
    ```

    This command creates the **uim-entities.jar** file and places it in the **ora_uim_src_entity/src/generated/inventory/APP_INF/lib** directory. The **uim-entities.jar** file contains the entity Java classes for all UIM entities and all custom entities. See "About the build.xml File" for more information.

14. Make a backup copy of the following files:
    - *UIM_Home*/**app/inventory-adapter.ear**
    - *UIM_Home*/**app/7_2_x/uim_core_lib.ear**
15. From a command line, navigate to your workspace **ora_uim_src_entity/src** directory and run the following Ant command:

    ```
    ant update.earwithEntities
    ```

    This command updates the **inventory-adapter.ear** and **uim_core_lib.ear** files with the **uim_entities.jar** file created in step 13. See "About the build.xml File" for more information.
16. If your *Domain_Home*/**servers/***serverName*/**upload/oracle.communications.inventory.corelib/7_2_x/app** directory contains a **uim_core_lib.ear** file:
    a. Make a backup copy of this **uim_core_lib.ear** file.
    b. Copy *Domain_Home*/**UIM/app/7_2_x/uim_core_lib.ear** to *Domain_Home*/**servers/***serverName*/**upload/oracle.communications.inventory.corelib/7_2_x/app**.
17. From a command line, navigate to your workspace **ora_uim_src_entity/src** directory and run one of the following set of Ant commands:

    ```
    ant create.tables
    ant alter.tables
    ```

    or

    ```
    ant create.DDL
    ```

    These commands update the database either directly (create.tables, alter.tables), or indirectly through a script (create.DDL). See "About the build.xml File" for more information.
18. Deploy the updated **inventory-adapter.ear** file.
19. Deploy the updated **uim_core_lib.ear** file.
20. If you have custom Java code in the **ora_uim_entity_sdk** project, deploy the resultant **ora_uim_entity_sdk.jar** file (cartridge) from the Studio environment into UIM.

    Deploying the cartridge adds your custom Java code to the *UIM_Home*/**app/7_2_x/uim_custom_lib.ear** file.
21. Deploy the **inventory.ear** file. At the time of deploying this EAR file, you must name this application **oracle.communications.inventory**.
22. If you added custom code, deploy the **custom.ear** file.

More on Entity Definitions

This section further describes entity definitions, focusing on:

- Understanding Entity Capability Definitions
- Understanding Entity Relationship and Collection Definitions
- Understanding Entity Manager Definitions

Understanding Entity Capability Definitions

A capability is a design pattern that is applied to an entity, such as enabling an entity to be life-cycle managed. A capability is declared in the metadata using tags, and results in the generation of attributes and related entities that are not explicitly defined in the ***-entities.xml** or ***-types.xsd** files.

For example, an entity that is life-cycle-managed progresses through a succession of states during the course of its life. For life-cycle-managed entities, UIM tracks two states: administrative state and object state. To support this capability, an entity must define the **adminState** and **objectState** attributes. Rather than declare these attributes in the ***-types.xsd** of every entity that supports this pattern, the capability is declared for the entity in the ***-entities.xml** file using the <lifeCycle> element **stateType** tag. As a result of this tag, the **adminState** and **objectState** attributes are generated on the entity, and the corresponding columns are generated on the database table.

The **package.xsd** and ***-plugin.xsd** files defines the tags that are available to declare a capability. Some capability definitions are modularized, such as the capacity, characteristic, consumable, and group-enabled capabilities, as defined in the following ***-plugin.xsd** files:

- capacity-plugin.xsd
- characteristic-plugin.xsd
- consumable-plugin.xsd
- groupenabled-plugin.xsd

Other ***-plugin.xsd** files include:

- core-plugin.xsd
- uim-plugin.xsd

The design patterns that are declared as capabilities are documented in *Oracle Communications Information Model Reference*.

Understanding Entity Relationship and Collection Definitions

Note: For information on all possible elements and attributes that can be used to define entity relationships and collections, see the **package.xsd** file.

The ***.entities.xml** files define entities and their relationships to other entities. Relationships between entities can be categorized into two types: Uni-directional and bi-directional. A uni-directional relationship only allows one-way traversal from one entity to another; a bi-directional relationship allows traversal both ways. The relationships can also be separated into three cardinalities: One-to-one, one-to-many, and many-to-many. Types and cardinalities of relationships result in any given entity relationship falling into one of six different combinations. Examples of these six different combinations are described in the following sections.

The direction of the relationships is not the determining factor of how primary and foreign keys are defined in the physical model. It affects only how the logical object model is defined. However, the relationship cardinality and ownership dictates the primary-foreign keys and join table definitions.

Relational and collection-type attributes can also be dependent. The entity or collection of entities referenced by a dependent attribute is deleted when the owning entity is deleted.

Uni-Directional, One-to-One Relationship

In Example 4–11, TopologyProfileEdge has a reference to TopologyEdge, but TopologyEdge does not have a reference to TopologyProfileEdge.

Example 4–11 Uni-Directional, One-to-One Relationship

```
<entity type="ocim:TopologyProfileEdge"
interface="oracle.communications.inventory.api.entity.TopologyProfileEdge"
accessControlled="true" entityIdSequenceGenerator="TopologySeqGen">
    .
    .
    .
    <!--  One-Sided One-to-One TopologyProfileEdge to TopologyEdge  -->
    <relationship name="topologyEdge">
        <otherSide type="ocim:TopologyEdge"/>
        <javadoc>
            The TopologyEdge that contains the TopologyProfileEdge.
        </javadoc>
    </relationship>
</entity>
```

Uni-Directional, One-to-Many Relationship

In Example 4–12, CharactersticSpecification has a reference to a collection of CharacteristicSpecValue entities, but a CharacteristicSpecValue does not have a reference back to the CharacteristicSpecification. The relationship can be omitted only on the many side. The CharacteristicSpecification ENTITYID foreign key is still realized physically as a column in the CharacteristicSpecValue table. However, a CharacteristicSpecValue entity does not have a Java attribute generated that allows the traversal back to the CharacteristicSpecification.

Example 4–12 Uni-Directional, One-to-Many Relationship

```
<entity type="ocim:CharacteristicSpecification"
interface="oracle.communications.inventory.api.entity.CharacteristicSpecification"
timeBound="true">
<implements interface="java.lang.Cloneable"/>
    .
    .
    .
    <!-- One-Sided One-to-Many
    CharacteristicSpecification to CharacteristicSpecValue-->
    <relationship name="values">
        <thisSide collection="java.util.HashSet"/>
        <otherSide dependent="true" type="ocim:CharacteristicSpecValue"/>
    </relationship>
</entity>
```

The collection data type is defined as a java.util.ArrayList. EclipseLink suggests that java.util.HashSet be used whenever possible to achieve better performance on their smart proxies logic because list-type collections such as ArrayList require sequential ordering for the elements for indexed access and allow for duplicate values of elements. Therefore, if the usage pattern of the collection attributes does not involve

direct indexed access to a specific element, and the elements are unique within the collection, set-type collections should be used instead.

Uni-Directional, Many-to-Many Relationship

There is no example in UIM of a uni-directional, many-to-many relationship; however, it is a valid relationship. Using a scenario of entity1 and entity2, entity1 is applicable to multiple entity2s, and each entity2 has access to multiple entity1s. However, only entity1 has a collection of entity2s. A join table is required for the many-to-many relationship.

In this relationship, the logical object model does not provide immediate insight that the relationship is many-to-many. From the entity1 point of view, it is one-entity1-to-many-entity2s. The logical object model does not show the many-to-many cardinality because there is no relationship back to entity1. However, the physical model exhibits the many-to-many relationship through the use of the join table.

Bi-Directional, One-to-One Relationship

In Example 4–13, Equipment has a reference to EquipmentEquipmentRel, and EquipmentEquipmentRel has a reference back to its sole Equipment. The relationship is owned by the Equipment and the EquipmentEquipmentRel is dependent on the Equipment.

The relationship name is used for generating the attribute name in the entity.

Example 4–13 Bi-Directional, One-to-One Relationship

```
<entity type="ocim:Equipment"
interface="oracle.communications.inventory.api.entity.Equipment"
accessControlled="true" entityIdSequenceGenerator="EquipmentSeqGen">
.
.
.
    <!-- Two-Sided One-to-One Equipment to EquipmentEquipmentRel (B) -->
        <relationship name="parentEquipment">
            <thisSide inverse="true"/>
            <otherSide type="ocim:EquipmentEquipmentRel"
            attribute="childEquipment"/>
            <javadoc>
                The holding parent equipment.
            </javadoc>
        </relationship>
</entity>
.
.
.
<entity type="ocim:EquipmentEquipmentRel"
interface="oracle.communications.inventory.api.entity.EquipmentEquipmentRel"
table="Eq_EqRel" accessControlled="true"
entityIdSequenceGenerator="EquipmentSeqGen">
.
.
.
    <!-- Two-Sided One-to-One EquipmentEquipmentRel to Equipment -->
        <relationship name="childEquipment">
            <otherSide type="ocim:Equipment" attribute="parentEquipment"/>
        </relationship>
</entity>
```

Bi-Directional, One-to-Many Relationship

In Example 4–14, Equipment has a reference to a collection of EquipmentHolderEquipmentRel entities, and EquipmentHolderEquipmentRel has a reference back to Equipment.

The inverse relationship is always on the one side of the relationship because the foreign key is on the many side. The relationship is owned by the Equipment. The collection of EquipmentHolderEquipmentRel entities is defined as dependent. As a dependent collection, the entities in the collection are deleted automatically when the owner entity is deleted. A dependent property is also applicable to simple no-collection type attributes.

Example 4–14 Bi-Directional, One-to-Many Relationship

```
<entity type="ocim:Equipment"
interface="oracle.communications.inventory.api.entity.Equipment"
accessControlled="true" entityIdSequenceGenerator="EquipmentSeqGen">
.
.
.
    <!-- Two-Sided One-to-Many Equipment to EquipmentEquipmentHolderRel -->
    <relationship name="parentEquipmentHolders">
        <thisSide inverse="true" collection="java.util.HashSet"/>
        <otherSide dependent="true" type="ocim:EquipmentHolderEquipmentRel"
            attribute="equipment"/>
        <javadoc>
            Set of parent equipment holders the equipment is held by.
        </javadoc>
    </relationship>
</entity>
.
.
.
<entity type="ocim:EquipmentHolderEquipmentRel"
interface="oracle.communications.inventory.api.entity.EquipmentHolderEquipmentRel"
table="EqHolder_EqRel" accessControlled="true"
entityIdSequenceGenerator="EquipmentSeqGen">
.
.
.
    <!-- Two-Sided Many-to-One EquipmentHolderEquipmentRel to Equipment -->
    <relationship name="equipment">
        <otherSide type="ocim:Equipment" attribute="parentEquipmentHolders"/>
            <javadoc>
                The child equipment.
            </javadoc>
    </relationship>
</entity>
```

Bi-Directional, Many-to-Many Relationship

In Example 4–15, Equipment can have multiple DeviceInterface entities, and each DeviceInterface entity can be applicable to many Equipment entities. A value of inverse="true" means that the other side of a two-way relationship owns the foreign key. In a many-to-many relationship, the inverse is arbitrary because there should be a join table created. In this case, the inverse="true" defines the other side as the owner of the relationship. Collection types include ArrayList and HashMap. See the EclipseLink documentation for all supported collection types.

Example 4–15 Bi-Directional, Many-To-Many Relationship

```
<entity type="ocim:Equipment"
interface="oracle.communications.inventory.api.entity.Equipment"
accessControlled="true" entityIdSequenceGenerator="EquipmentSeqGen">
.
.
.
    <!-- Two-Sided Many-to-Many Equipment to DeviceInterface-->
    <relationship name="supportedDeviceInterfaces">
        <join table="equipment_deviceinterface"/>
        <thisSide inverse="true" collection="java.util.HashSet"/>
        <otherSide type="ocim:DeviceInterface"/>
        <javadoc>
            The list of mapped device interaces supported by the equipment.
        </javadoc>
    </relationship>
</entity>
.
.
.
<entity type="ocim:DeviceInterface"
interface="oracle.communications.inventory.api.entity.DeviceInterface"
accessControlled="true" entityIdSequenceGenerator="LogicalDeviceSeqGen">
.
.
.
    <!-- Two-Sided Many-to-Many DeviceInterface to Equipment-->
    <relationship name="supportingEquipment">
        <join table="equipment_deviceinterface"/>
        <thisSide collection="java.util.HashSet"/>
        <otherSide type="ocim:Equipment"/>
        <javadoc>
             The list of equipment up the hierarchy that supports the device
             interface mapping. This will only be populated if the device
             interface is mapped.
        </javadoc>
    </relationship>
</entity>
```

Relationship Definition Affect on Generated Entities

Each relationship definition adds an attribute to the generated entity for which it is defined. For a uni-directional relationship, an attribute is generated for the owning entity. For a bi-directional relationship, an attribute is generated for the owning entity and for the dependent entity.

Example 4–16 is an excerpt from the **uim-rule-entities.xml** file that defines the **extensionPoint**, **ruleSetEntity**, and **specification** relationships for the ExtensionPointRuleSet entity.

Example 4–16 Entity Definition

```
<entity type="ocim:ExtensionPointRuleSet"
interface="oracle.communications.inventory.api.entity.ExtensionPointRuleSet">
    <identifier>
        <attribute>extensionPoint</attribute>
        <attribute>ruleSetEntity</attribute>
        <attribute>specification</attribute>
    </identifier>
    <!-- Two-Sided Many-to-One ExtensionPointRuleSet to ExtensionPoint -->
```

```
    <relationship name="extensionPoint">
        <otherSide type="ocim:ExtensionPoint" attribute="extensionPointRuleSets"/>
    </relationship>
    <!-- Two-Sided Many-to-One ExtensionPointRuleSet to RuleSetEntity  -->
    <relationship name="ruleSetEntity">
        <otherSide type="ocim:RuleSetEntity" attribute="extensionPointRuleSets"/>
    </relationship>
    <!-- Two-Sided Many-to-One ExtensionPointRuleSet to Specification  -->
        <relationship name="specification">
            <otherSide type="ocim:Specification"
            attribute="extensionPointRuleSets"/>
    </relationship>
</entity>
```

Example 4–17 is an excerpt from the **uim-rule-types.xsd** file that defines the **type** attribute and the **sequence** attribute for the ExtensionPointRuleSet entity.

Example 4–17 Entity Attributes Definition

```
<xs:complexType name="ExtensionPointRuleSet">
    <xs:annotation>
        <xs:documentation>
            Associates extension points and rulesets.
        </xs:documentation>
    </xs:annotation>
    <xs:sequence>
        <xs:element name="type" type="ocim:ExtensionPointRuleSetType">
            </xs:annotation>
        </xs:element>
        <xs:element name="sequence" type="integer">
            </xs:annotation>
        </xs:element>
    </xs:sequence>
</xs:complexType>
```

Example 4–18 is a code excerpt from the ExtensionPointRuleSet generated source file that defines the attributes for ExtensionPointRuleSet entity. The **type** attribute and the **sequence** attribute are generated based on the attributes defined in the **uim-rule-types.xsd** file for the ExtensionPointRuleSet entity. The fields extensionPoint, ruleSetEntity, and specification are generated based on the relationships defined in the **uim-rule-entities.xml** file for the ExtensionPointRuleSet entity.

Example 4–18 Generated Source File

```
/*
 * ExtensionPointRuleSet.java
 * [CODE-GENERATED]
 */
package oracle.communications.inventory.api.entity;
/**
 *         Associates extension points and rulesets.
 */
public interface ExtensionPointRuleSet
    extends java.io.Serializable,
        oracle.communications.platform.persistence.Persistent,
        oracle.communications.inventory.api.Trackable
{
public static final
oracle.communications.platform.persistence.impl.EntityField _type =  new
oracle.communications.platform.persistence.impl.EntityField(ExtensionPointRuleSet.
```

```
class, "type");

public static final
oracle.communications.platform.persistence.impl.EntityField _sequence =  new
oracle.communications.platform.persistence.impl.EntityField(ExtensionPointRuleSet.
class, "sequence");

public static final
oracle.communications.platform.persistence.impl.EntityField _extensionPoint =  new
oracle.communications.platform.persistence.impl.EntityField(ExtensionPointRuleSet.
class, "extensionPoint");

public static final
oracle.communications.platform.persistence.impl.EntityField _ruleSetEntity =  new
oracle.communications.platform.persistence.impl.EntityField(ExtensionPointRuleSet.
class, "ruleSetEntity");

public static final
oracle.communications.platform.persistence.impl.EntityField _specification =  new
oracle.communications.platform.persistence.impl.EntityField(ExtensionPointRuleSet.
class, "specification");
```

Taking the example one step further, you can look at the generated source code for the ExtensionPoint, RuleSetEntity, and Specification entities. These entities are defined as the other side of the bi-directional relationships in Example 4–16, "Entity Definition". All three generated source files define the **extensionPointRuleSets** attribute, as defined by the **otherSide** attribute for each.

Understanding Entity Manager Definitions

The persistence framework manages the database and the mapping between the database and the entity classes. An entity manager manages the database tables for a specific functional area. For example, EquipmentManager manages the Equipment table, but it also manages EquipmentHolder, PhysicalPort, PhysicalConnector, PhysicalDevice, and so forth.

Defining Entity Managers

Entity managers are UIM classes. As a result, entity managers are defined in **uim-*-entities.xml** files, and not in **ocim-*-entities.xml** files. Entity managers are defined using the <manager> element and **interface** attribute that are defined in the **package.xsd** file. Example 4–19 is an excerpt from the **uim-equipment-entities.xml** file that shows the EquipmentManager entity manager definition.

Example 4–19 Entity Manager Definition

```
<manager
interface="oracle.communications.inventory.api.equipment.EquipmentManager"
class="oracle.communications.inventory.api.equipment.impl.EquipmentManagerImpl"/>
```

Upon installation of UIM, every entity manager that is defined in the metadata has a corresponding entity manager and implementation of the manager. For example, based on the excerpt shown in Example 4–19, the following classes exist:

- EquipmentManager
- EquipmentManagerImpl

If you extend the data model by creating a new entity, you must also create a new entity manager, and implementation of the manager, to manage the entity data. See

"Creating New Entity Managers" for more information.

The relationship of entity to entity manager is not one-to-one. For example, **ocim-equipment-entities.xml** defines several entities, each of which defines its own entity interface. An entity interface differs from a manager interface; an entity interface defines the getter and setter methods for entity attributes, while a manager interface defines methods for the entity, such as the createEquipment(), getEquipment(), or updateEquipment() methods. Example 4–20 is an excerpt from the **ocim-equipment-entities.xml** file that shows a portion of the Equipment and EquipmentHolder entity definitions.

Example 4–20 Entity Definition

```
<entity type="ocim:Equipment"
    interface="oracle.communications.inventory.api.entity.Equipment"
    accessControlled="true" entityIdSequenceGenerator="EquipmentSeqGen">

<entity type="ocim:EquipmentHolder"
    interface="oracle.communications.inventory.api.entity.EquipmentHolder"
    accessControlled="true" entityIdSequenceGenerator="EquipmentSeqGen">
```

Both of these entities are extended in the **uim-equipment-entities.xml** file, as shown in Example 4–21.

Example 4–21 ManagedBy Declarations

```
<entity type="ocim:Equipment"
    managedBy="oracle.communications.inventory.api.equipment.EquipmentManager">

<entity type="ocim:EquipmentHolder"
    managedBy="oracle.communications.inventory.api.equipment.EquipmentManager">
```

The **managedBy** tag is only present on entities that are business-interaction enabled. For business-interaction enabled entities, the **managedBy** tag specifies which entity manager manages the entity.

5

Extending Life Cycles

This chapter provides information on extending Oracle Communications Unified Inventory Management (UIM) entity life cycles. An entity life cycle refers to an entity having a start to its life, an end to its life, and a defined state at any given point during its life. Life-cycle state transition definitions are part of the UIM metadata, and these definitions can be extended to solve specific business requirements.

An entity can be defined as life-cycle managed in the metadata. Life-cycle managed entities transition through various states throughout the life cycle. The states are determined by the transition definition specified for the entity in the metadata.

The information presented in this chapter describes statically extending UIM, which can result in backward compatibility issues. See "Backward Compatibility" for the implications regarding this type of extension.

Note: Before you begin reading about extending life cycles, you should have an understanding of the following concepts described in *UIM Concepts*:

- Business Interactions
- Life Cycles

About Business Interactions

Business interactions represent business transactions or events that affect products, services, and resources in inventory. They include service requests, sales orders, and network planning projects. Business interactions are modeled in inventory to facilitate change in the inventory, provide traceability, and enable transaction cancellations and changes. They can involve current business transactions, such as service orders, or future planned events, such as grooming projects.

In the UIM user interface (UI), you can switch between business interactions and current inventory by choosing **Current** on the menu bar. The **Current** menu has the following options:

- **Current:** Switches from a business interaction to current inventory.
- **Recent BIs:** Lists the five most recently accessed business interactions.
- **Search:** Opens the Business Interaction Search page. Accessing a business interaction from the Search page switches the current business interaction to the selected business interaction, and also adds the selected business interaction as an option on the **Current** menu.

Business interactions tie in with transition definitions because the business states through which an entity transitions depend on whether the entity is within the context of a business interaction or current inventory. Each transition definition can define different <from> and <to> business states for business interaction versus current inventory. See Example 5–4, "Create Transition".

Understanding Metadata File Content

Extending an entity to be life-cycle managed, and extending life-cycle state transitions, is done through the metadata files and involves the definitions of:

- Entities
- Enumerations
- Transitions

Understanding Life-Cycle Managed Entity Definitions

This section builds upon the information presented in "Understanding Entity Definitions".

An entity can be defined as life-cycle managed and business-interaction enabled in the metadata. A business-interaction enabled entity is, by inheritance, automatically a life-cycle managed entity. Conversely, an entity can be defined as life-cycle managed in the metadata without being a business-interaction enabled entity. The elements and attributes used to define an entity as life-cycle managed and business-interaction enabled are defined in the **uim-plugin.xsd** file. For example, the <lifecycle> element is used to define an entity as life-cycle managed, and the <businessInteractionEnabled> element is used to define an entity as business-interaction enabled.

Example 5–1 is an excerpt from the **ocim-number-entites.xml** file that shows the TelephoneNumber entity definition. The definition includes the declaration of the life-cycle managed and business-interaction enabled capabilities, which are bolded in the example. The <lifecycle> element defines the **stateType** attribute, which defines a value of **InventoryState**. **InventoryState** is an enumeration and is described in "Understanding Life-Cycle Managed Enumeration Definitions".

Example 5–1 Entity Definition

```
<entity type="ocim:TelephoneNumber"
interface="oracle.communications.inventory.api.entity.TelephoneNumber" accessControlled="true">
.
.
.
    <!-- ***************** Capabilities ******************-->
    <lifecycle stateType="ocim:InventoryState"/>
    <consumable prefix="TN" attribute="telephoneNumber"
    assignmentStateType="ocim:AssignmentState">
        <consumer name="ocim:Service" ConfigurationItemEnabled="true"/>
    </consumable>
    <referenceEnabled prefix="TelephoneNumber" attribute="telephoneNumber"/>
    <characteristic spec="ocim:CharacteristicSpecification">
        <characteristicName name="ocim:TNCharacteristic"
        interface="oracle.communications.inventory.api.entity.TNCharacteristic"
        table="TN_CHAR"/>
    </characteristic>
    <businessInteractionEnabled history="true" visibilityState="SHOW"/>
    <groupEnabled/>
```

```
.
.
.
</entity>
```

Understanding Life-Cycle Managed Enumeration Definitions

This section builds upon the information presented in "Understanding Enumeration Definitions".

About Life-Cycle States

Life-cycle managed entities transition through various states throughout the life cycle. These life-cycle states are defined as enumerations. There are two types of life-cycle states that an entity transitions through: Business states and object states.

- A **business state** represents the current state as a result of a business action such as validate, approve, issue, complete, or cancel.
- An **object state** represents the current state as a result of an object activity such as create, update, or delete.

Business state enumerations are defined in the ***-enum-entities.xml** and ***-enum-types.xsd** metadata files. Numerous business state enumerations are defined in the metadata upon installation of UIM, and you can extend the business state enumerations to solve business requirements.

Object state enumerations are defined in a Java class and cannot be extended. The object state enumerations are:

- PLANNED
- QUEUED
- ACTIVE
- INACTIVE
- CANCELLED
- DELETED

Understanding Business State Enumerations

Example 5–2 is an excerpt from the **ocim-enum-entities.xml** file, which defines the InventoryState enumeration type.

Example 5–2 Enumeration Type Definition

```
<enum type="ocim:InventoryState"
enumType="oracle.communications.inventory.api.entity.InventoryState"
adminState="true"/>
```

Example 5–1, "Entity Definition" defined the TelephoneNumber entity to be life-cycle managed, and the definition included the **stateType** attribute value of **InventoryState**, which is an enumeration.

Example 5–3 is an excerpt from the **ocim-enum-types.xsd** file, which defines the enumeration values for the InventoryState enumeration type. The enumeration type and enumeration values indicate that the TelephoneNumber entity may transition through up to eight business states during its life cycle.

Example 5–3 Enumeration Values Definition

```
<xs:simpleType name="InventoryState">
    <xs:annotation>
        <xs:documentation>Inventory Status</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
        <xs:enumeration value="PLANNED"/>
        <xs:enumeration value="PENDING_INSTALL"/>
        <xs:enumeration value="INSTALLED"/>
        <xs:enumeration value="PENDING_UNAVAILABLE"/>
        <xs:enumeration value="UNAVAILABLE"/>
        <xs:enumeration value="PENDING_REMOVE"/>
        <xs:enumeration value="END_OF_LIFE"/>
        <xs:enumeration value="PENDING_AVAILABLE"/>
    </xs:restriction>
</xs:simpleType>
```

Understanding Transition Definitions

> **Note:** Transition definitions for current inventory are defined within the <live> element; <live> displays as **current** in the UIM UI.

A transition defines the intermediate step from one business state to another business state, or from one object state to another object state. For example, within the context of a business interaction, the create transition moves an entity from inception to the initial PENDING_INSTALL business state, and the createComplete transition moves an entity from the PENDING_INSTALL business state to the INSTALLED business state. Similarly, the create transition moves an entity from inception to the initial QUEUED object state, and the createComplete transition moves an entity from the QUEUED object state to the ACTIVE object state.

Transition definitions are defined in files that start with **uim-** and end with **-transitions.xml**. For example, **uim-default-transitions.xml**. The transition definition files are located in the *UIM_Home***/cartridges/tools/ora_uim_entity_sdk.zip/src/uim_poms_lib.jar** file.

You can extend business state enumerations, but you cannot extend object state enumerations. For transitions, you can extend both business state and object state transitions.

Example 5–4 is an excerpt from the **uim-default-transitions.xml** file, which defines the **Create** transition for the business state:

- From inception to PENDING_INSTALL within the context of a business interaction
- From inception to INSTALLED within the context of current inventory

The example also defines the **Create** transition for the object state:

- From inception to QUEUED within the context of a business interaction
- From inception to ACTIVE within the context of current inventory

Example 5–4 Create Transition

```
<transition name="Create" priority="0" default="true">
    <objectActivity value="CREATE"/>
    <businessState type="ocim:InventoryState">
```

```
        <attribute name="adminState" isCharacteristic="false"/>
        <businessInteraction>
            <from/>
            <to>PENDING_INSTALL</to>
        </businessInteraction>
        <live>
            <from/>
            <to>INSTALLED</to>
        </live>
    </businessState>
    <objectState>
        <businessInteraction>
            <from/>
            <to>QUEUED</to>
        </businessInteraction>
        <live>
            <from/>
            <to>ACTIVE</to>
        </live>
    </objectState>
</transition>
```

Understanding How Transitions Are Triggered

Transitions can be triggered automatically from within custom code or manually from within the UIM user interface.

For information on the life cycle management interfaces that are available when writing custom code to automatically transition an entity's life-cycle state, Custom code can be called from:

- Customized user interface
- Rulesets
- Web services

You can manually transition an entity's life-cycle state from the **Actions** menu on the Summary page of any entity that is defined as life-cycle managed. The **Actions** menu options reflect the applicable transitions defined for the entity, based on the entity's current state.

> **Note:** Manually transitioning through an entity's life cycle by selecting the options on the **Actions** menu implies that the correct life-cycle state is dependent on user interaction to initiate the transition.

Example 5–5 is an excerpt from the **uim-default-transitions.xml** file, which defines the **Activate** and **Deactivate** transitions. The **Activate** and **Deactivate** transitions are shown in Figure 5–1, "Summary Page Actions Menu".

The example defines the **Activate** transition for the business state:

- From UNAVAILABLE to PENDING_AVAILABLE within the context of a business interaction
- From UNAVAILABLE to INSTALLED within the context of current inventory

The example defines the **Activate** transition for the object state:

- Nothing is defined within the context of a business interaction

- From INACTIVE to ACTIVE within the context of current inventory

The example defines the **Deactivate** transition for the business state:

- From INSTALLED to PENDING_UNAVAILABLE within the context of a business interaction
- From INSTALLED to UNAVAILABLE within the context of current inventory

The example defines the **Deactivate** transition for the object state:

- Nothing is defined within the context of a business interaction
- From ACTIVE TO INACTIVE within the context of current inventory

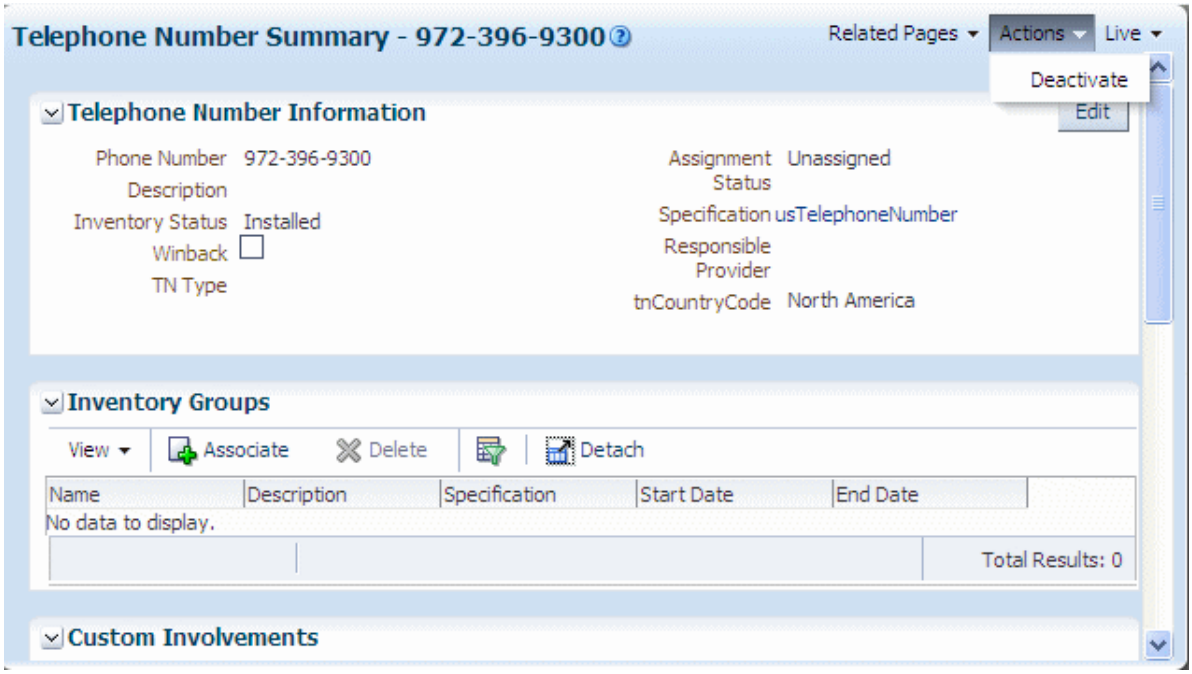
Example 5–5 Activate and Deactivate Transitions

```
<transition name="Activate" priority="0" default="true">
    <businessAction type="ocim:ResourceAction" value="ACTIVATE"/>
    <businessState type="ocim:InventoryState">
        <attribute name="adminState" isCharacteristic="false"/>
        <businessInteraction>
            <from>UNAVAILABLE</from>
            <to>PENDING_AVAILABLE</to>
        </businessInteraction>
        <live>
            <from>UNAVAILABLE</from>
            <to>INSTALLED</to>
        </live>
    </businessState>
    <objectState>
        <live>
            <from>INACTIVE</from>
            <to>ACTIVE</to>
        </live>
    </objectState>
    .
    .
    .
</transition>
<transition name="Deactivate" priority="0" default="true">
    <businessAction type="ocim:ResourceAction" value="DEACTIVATE"/>
    <businessState type="ocim:InventoryState">
        <attribute name="adminState" isCharacteristic="false"/>
        <businessInteraction>
            <from>INSTALLED</from>
            <to>PENDING_UNAVAILABLE</to>
        </businessInteraction>
        <live>
            <from>INSTALLED</from>
            <to>UNAVAILABLE</to>
        </live>
    </businessState>
    <objectState>
        <live>
            <from>ACTIVE</from>
            <to>INACTIVE</to>
        </live>
    </objectState>
    .
    .
    .
```

```
</transition>
```

Figure 5–1 shows the Telephone Number Summary page **Actions** menu, which reflects the applicable transitions defined for the TelephoneNumber entity based on its current state. The telephone number, shown in the context of current inventory, has an inventory status of **Installed**, so **Deactivate** is the only available transition option. If you select **Deactivate**, the inventory status changes to **Unavailable**, and **Activate** becomes the only available transition option. When in the context of a business interaction, the value of the **Inventory Status** field reflects the states defined for <businessInteraction> based on the entity's current state. When in the context of current inventory, the value of the **Inventory Status** field reflects the status defined for current (live) based on the entity's current state.

Figure 5–1 Summary Page Actions Menu



About Transition Groups

A transition group provides the ability to associate a group of transition definitions with a specification. A transition group requires a name, which is used to associate it with a specification. A transition group can be associated with multiple specifications. By default, transition definitions defined within a transition group are templates. Templates are transition definitions that are not active/searchable until the group in which they are defined is associated with a specification. Within a **uim-*-transitions.xml** file, the <transitionGroup> element can define the **templateOnly** optional attribute, which defaults to **true**. If set to **true**, the transition definitions in the group are active/searchable, even though they are not yet associated with a specification.

Example 5–6, "Transitions Definition" shows an example of a transition group.

Extending Life Cycles through the Metadata Files

You extend life cycles by creating new metadata files.

Caution: Do not make modifications to the existing metadata files. See "Backward Compatibility" for the issues involved with modifying the existing metadata files.

The metadata files are contained in the *UIM_Home*/**cartridges/tools/ora_uim_entity_sdk.zip/src/uim_poms_lib.jar** file.

You can use Oracle Communications Design Studio to create new metadata files. For example, you can import the **ora_uim_entity_sdk.zip** file and create any new XML or XSD files within the imported project.

This section builds upon information presented in Chapter 4, "Extending the Data Model". Any new metadata files you create must reside in **ora_uim_entity_sdk** project to be picked up by the generator. See "Applying Metadata Static Extensions" for more information.

Extending Entity Definitions

You can extend an entity definition to be business-interaction enabled, life-cycle managed, or both.

Defining an Entity as Life-Cycle Managed

The presence of the <lifecycle> element in the entity definition defines an entity as life-cycle managed. **stateType** is a required attribute of the <lifecycle> element. The value of **stateType** is an enumeration type that is defined in the **ocim-enum-entities.xml** file. This file defines several enumeration types that can be specified for **stateType**. Each enumeration type defines a set of enumeration values that represent the states of a specific life cycle. **initialState** is an optional attribute of the <lifecycle> element. **initialState** defines an enumeration that represents the default initial life-cycle state.

To define a new entity as life-cycle managed, add the <lifecycle> element to the entity definition in the new ***-entities.xml** file. See "Defining New Entities" for more information.

To define an existing entity as life-cycle managed, add the <lifecycle> element to the existing entity by extending the entity definition in the new ***-entities.xml** file. See "Extending Existing Entities" for more information.

Any new entity files you create must end with **-entities.xml** and reside in the **ora_uim_entity_sdk/src/api** directory, to be picked up by the entity generator.

Defining an Entity as Business-Interaction Enabled

The presence of the <businessInteractionEnabled> element in the entity definition defines an entity as business-interaction enabled. **history** is an optional attribute of the <businessInteractionEnabled> element. The **history** attribute is a boolean: if it is set to **true**, the version object is kept in the data store; if it is set to **false,** the version object is deleted. (Versioning is not covered in this guide. For information about versioning, see *UIM Concepts*.) **visibilityState** is also an optional attribute of the <businessInteractionEnabled> element. **visibilityState** defines an enumeration that is the default initial display level for the business-interaction enabled entity.

To define a new entity as business-interaction enabled, add the <businessInteractionEnabled> element to the new entity definition in the new ***-entities.xml** file. See "Defining New Entities" for more information.

To define an existing entity as business-interaction enabled, add the <businessInteractionEnabled> element to the existing entity by extending the entity definition in the new ***-entities.xml** file. See "Extending Existing Entities" for more information.

Any new entity files you create must end with **-entities.xml** and reside in the **ora_uim_entity_sdk/src/api** directory to be picked up by the entity generator.

> **Note:** If an entity is inherited from a business-interaction enabled entity, the entity cannot be defined as business-interaction enabled.

Defining an Entity as Life-Cycle Managed and Business-Interaction Enabled

A business-interaction enabled entity is, by inheritance, automatically a life-cycle managed entity. The presence of the <businessInteractionEnabled> element in the entity definition defines an entity as business-interaction enabled and as life-cycle managed. However, the presence of the <lifecycle> element in the entity definition is still required to specify **stateType**. Example 5–1, "Entity Definition" showed both the <lifecycle> and <businessInteractionEnabled> elements in the entity definition.

Extending Enumeration Definitions

You can create new enumeration files to address business requirements. New files you create must end with **-entities.xml** or **-types.xsd**. New files that follow this naming convention, and that reside the **ora_uim_entity_sdk/src/api** directory, are picked up by the entity generator.

See "Defining New Enumerations" and "Extending Existing Enumerations" for more information.

Extending Transition Definitions

When extending transitions by either creating new transitions or extending existing transitions, look at the existing **uim-*-transitions.xml** files for examples. Any new entity files you create that end with **-transitionss.xml** and that reside in the **ora_uim_entity_sdk/src/api** directory are picked up by the entity generator

The following transition procedures state how to create a new file, but you do not need to create a new transitions file for each new transition. For example, you can optionally define all new transitions and extending existing transitions in the same file.

Defining New Transitions

To add a new transition to a new ***-transitions.xml** file:

1. Create a new XML file.

   The file name must end with **-transitions.xml**. For example, **myNewFile-transitions.xml**.

2. Open an existing **uim-*-transitions.xml** file.
3. Copy and paste a transition definition from the existing file to your new file.
4. Modify the copied transition definition as needed:
   a. Change the name of the transition to reflect the name of your new transition.
   b. Remove or update the tags to reflect the definition of your new transition.

Extending Existing Transitions

To extend an existing transition in a new ***-transitions.xml** file:

1. Create a new XML file.

   The file name must end with **-transitions.xml**. For example, **myNewFile-transitions.xml**.

2. Open the existing **uim*-transitions.xml** file that you plan to extend.
3. Copy and paste the transition definition from the existing file to your new file.
4. Modify the copied transition definition as needed:
   a. Add additional <businessState> elements as needed.
   b. Do not change the transition name.
   c. If the copied transition does not define the **priority** attribute, add it and set the value to 1 (the default is 0). If the copied transition already defines the **priority** attribute, increase the value. The **priority** attribute value is used when the transition name is not unique. The higher the value, the higher the priority.

Updating Properties Files

If you extend life cycles, you need to update some properties files that are used to display life-cycle statuses. The following properties files are located in the *UIM_Home*/**config/resources/logging** directory:

- **status.properties**

  This file defines statuses that are referenced by the UI. If life cycles are extended by introducing new statuses through the metadata transition files, and the statuses are referenced by the UI, the **status.properties** file must be updated to reflect the new statuses.

- **enum.properties**

  This file defines enumerations that are referenced by the UI. If life cycles are extended by introducing new enumerations through the metadata enumeration files, and the enumerations are referenced by the UI, the **enum.properties** file must be updated to reflect the new enumerations.

Updating Security

If you extend life cycles, you need to update security for any new actions to display in the UIM UI.

To update security:

1. Log in to the Enterprise Manager Console.
2. In the navigation panel, expand **Application Deployments** and click the **oracle.communications.inventory (AdminServer)** link.

   The **oracle.communications.inventory** page appears.

3. From the **Application Deployment** list menu, select **Security**, then select **Application Policies**.

   The Application Policies page appears.

4. Expand the **Search** page.
5. From the **Principle Type** list, select **Application Role** and click the search icon.

The search results display.

6. Select the **uimuser** row and click **Edit**.

   The Edit Application Grant page appears.

7. Under Permissions, click **Add**.

   The Add Permission dialog box appears.

8. Expand the **Search** page, and choose **Permissions**.

9. From the **Permission Class** list, select **oracle.security.jps.ResourcePermission**.

10. From the **Resource Name** list, select **Starts With**.

11. In **Resource Name**, enter the following text where *Entity* is any entity such as Service, CustomObject, and so forth, and where *BusinessAction* is your custom business action:

    resourceType=PAGE_ACTION,resourceName=*Entity.BusinessAction*

12. Click the search icon.

    The search results display.

13. Select the applicable resource name, and click **Continue**.

14. In **Permission Actions**, enter **view**.

15. Click **Select**.

    The Add Permission dialog box closes.

16. Click **OK** to grant the permission.

More on Transition Definitions

The following information is provided to help you define ***-transitions.xml** files. Each transition file can define multiple transition definitions, and each transition definition can define multiple states. Example 5–6, "Transitions Definition" includes all the possible elements and attributes described below.

- <transition> can be defined multiple times within the same file.
  - **name** is required and should be unique. If duplicate transition names are found, the one with the higher **priority** attribute value is used.
  - **entityType** is optional. If it is not specified, the transition definition is available for all entity types.
  - **priority** is optional, and has a default value of 0. The higher the value, the higher the priority. The value is used when name is not unique. If the same name and same priority are specified, an error occurs.
- <specification> is optional. If it is not specified, the transition definition is available for entities with any specification.
- <businessAction> and <objectActivity> are optional, but one of them must be specified. These values are used by the lookup process to determine the transition definition.
- <businessState> can be defined multiple times within a transition. This defines the business states that the entity transitions through during its life cycle.
  - <businessState> can be set on an entity's attribute or a custom attribute.
  - **type** must be a valid enumeration.

  - **isCharacteristic** indicates whether <businessState> is an attribute or a custom attribute.
  - **name** is either the attribute name or the custom attribute name.
  - If **isCharacteristic** is set to **true**, you can specify the **characteristicSpecName** attribute. If this attribute is not set, the system uses the **name** attribute value as the characteristic-specific name.
  - <businessState> can optionally define zero, one, or many <businessInteraction> blocks, or zero or one <live> block, or both.
  - If <businessInteraction> is defined, its <from> state is used to match the entity's current business state if the transition happens within the context of a business interaction. If <from> is not specified, it is considered a wild card and can be matched with any entity's current state.
  - If <live> is defined, its <from> state is used to match the entity's current business state if the transition happens within the context of current inventory.
  - If the transition happens within the context of a business interaction and <businessInteraction> is not defined, the search for a match continues. Similarly, if the transition happens within the context of current inventory and <live> is not defined, the search for a match continues.

- The <businessInteraction> block and <live> block can define multiple <from> states. This allows matching multiple <from> states without defining them separately in each <businessState> block. If <from> is not specified, it is considered a wild card and can be matched with any entity's current state.
- There can be only one <to> state defined in the <businessInteraction> and <live> blocks. The value is used to set the entity's business state. If <to> is not specified, the entity's current state is not changed.
- Only one <objectState> block can be defined for the transition definition.
- <objectState> can define zero, one, or many <businessInteraction> blocks, and zero or one <live> block, and each can define multiple <from> states.
- The <dependants> block defines the methods to retrieve the dependent entities and how to transition them. Multiple <dependants> blocks can be defined in a transition definition.
  - **attribute** is the attribute name of the parent entity and is used to hold the dependent entities by the parent entity.
  - **isCollection** is a boolean that indicates whether attribute holds a collection (**true**) or a single dependent entity (**false**).
  - If the dependent entity is accessed indirectly through the weak reference of the attribute, then **weakReference** is the name of the access method to resolve the weak reference. For example, the BusinessInteraction entity has an **items** attribute that holds a collection of BusinessInteractionItem entities, but the BusinessInteractionItem entity has a weak reference that refers to the real dependent entity (TelephoneNumber). The **toEntity** attribute is specified to resolve the TelephoneNumber entity from the BusinessInteractionItem entity.
  - After the dependent entities have been resolved, the system is ready to transition the dependent entities with the parent's business action and object activity. However, if **useDependentObjectActivity** is **true**, the system uses the dependent entity's object activity and parent entity's business action to transition the dependents.

– If the parent's business action is not valid for transitioning the dependents, you have the option to specify one or more <transitionName> elements in the <dependants> block. TransitionNames retrieves the transitions in sequence, then uses the transition's business action and object activity to look up the matching transition for each dependent. If there is no matching transition by using transitionNames, the action described in the previous bullet is performed.

Example 5–6 Transitions Definition

```
<transitionGroup name="defaultBusinessInteractionGroup" templateOnly="false">
    <transition name="BusinessInteractionCreate"
    entityType="ocim:BusinessInteraction" assignable="true" priority="0">
        <objectActivity value="CREATE"/>
        <businessState type="ocim:BusinessInteractionState">
            <attribute name="adminState" isCharacteristic="false"/>
            <businessInteraction>
                <from/>
                <to>CREATED</to>
            </businessInteraction>
            <live>
                <from/>
                <to>CREATED</to>
            </live>
        </businessState>
        <objectState>
            <businessInteraction>
                <from/>
                <to>ACTIVE</to>
            </businessInteraction>
            <live>
                <from/>
                <to>ACTIVE</to>
            </live>
        </objectState>
    </transition>
    <transition name="BusinessInteractionComplete"
    entityType="ocim:BusinessInteraction" assignable="true" priority="0">
        <businessAction type="ocim:BusinessInteractionAction" value="COMPLETE"/>
        <businessState type="ocim:BusinessInteractionState">
            <attribute name="adminState" isCharacteristic="false"/>
            <businessInteraction>
                <from>CREATED</from>
                <from>IN_PROGRESS</from>
                <to>COMPLETED</to>
            </businessInteraction>
        <businessState>
        <objectState>
            <businessInteraction>
                <from>ACTIVE</from>
                <to>INACTIVE</to>
            </businessInteraction>
        </objectState>
        <dependants attribute="items" isCollection="true" weakReference="toEntity"
             useDependentObjectActivity="true">
        </dependants>
    </transition>
</transitionGroup>
```

About Life Cycle Management Interfaces

The following sections describe life cycle management interfaces. For information on the methods defined by any of these interfaces, see the Javadoc. For instructions on how to access the Javadoc, see "Javadoc Documentation".

LifeCycleManaged

An entity that is defined as life-cycle managed in the metadata automatically implements the LifeCycleManaged interface. It is not necessary to include the tag:

```
<implements interface="oracle.communications.inventory.api.LifeCycleManaged"/>
```

The LifeCycleManaged interface:

- Defines a business state for the entity

  A business state represents the current state as a result of a business action such as validate, approve, issue, complete, or cancel.

- Defines an object state for the entity

  An object state represents the current state as a result of an object activity such as create, update, or delete.

TransitionManager

An entity that is defined as life-cycle managed in the metadata automatically implements the LifeCycleManaged interface. This enables you to call methods on the oracle.communications.inventory.api.common.TransitionManager interface, which takes in a LifeCycleManaged entity as an input parameter.

The TransitionManager interface:

- Defines methods that take in a business action and appropriately transition the business and object states
- Automatically updates the business state and object state of any life-cycle managed dependent entities when the parent life-cycle managed entity business state or object state is updated
- Provides the ability to associate or disassociate a specification with a transition group

Transition Definition Search

The transition() method provides the ability to transition through the defined business states and object states. To do this, it must first determine the transition definitions for business state and object state that apply to the entity. This is accomplished through a search that takes place within the transition() method.

The transition() method input parameters are the life-cycle managed entity, business action, and object activity. The life-cycle managed entity parameter contains entity type and specification, which are used in the transition definition search. If no match is found, a less relevant search is performed until a transition definition is found. The following lists the search criteria in the most-significant to least-significant order. The least-relevant transition definition returned would be the default transition definition.

1. Business action, object activity, entity type, specification
2. Business action, object activity, entity type
3. Business action, object activity

At this point, the search has returned one or more transition definitions that matched the criteria. This list of transition definitions is now interrogated to find one that defines a <from> business state that matches the entity's current business state. Whether the entity is within the context of a business interaction determines which <from> business state is interrogated: Business interaction or current (live).

6

Extending the Topology

This chapter provides information on extending the topology in Oracle Communications Unified Inventory Management (UIM). The topology is a graphical representation of the spatial relationships and connectivity among your inventory entities.

The topology uses a specific set of entities and a specific algorithm to determine the path between any two entities. This algorithm is called the path analysis. You can extend the topology to include additional entities in the topology, and you can modify the path analysis to suit your business needs.

The information presented in this chapter describes statically extending UIM, which can result in backward compatibility issues. See "Backward Compatibility" for the implications regarding this type of extension.

Note: Before you begin reading about extending topology, it is important that you have an understanding of the following subjects described in *UIM Concepts*:

- Connectivity
- Topology

About Topology Entities and Topology-Managed Entities

Topology entities are defined in the metadata and are used to display the topology. Topology-managed entities are also defined in the metadata and are indirectly used to display the topology. UIM maps topology-managed entities to one of two topology entities, and, as a result of the mapping, topology-managed entities indirectly display in the topology.

Topology Entities

The metadata defines the following topology entities:

- TopologyEdge
- TopologyNode

TopologyNode entities represent locations, network nodes, or devices, and TopologyEdge entities represent pipes or network edges.

The metadata defines the topology entities in the **topology-entities.xml** file. Example 6–1 is an excerpt from this file that shows the definition of the TopologyNode entity.

Example 6–1 topology-entities.xml

```
<entity type="ocim:TopologyNode"
interface="oracle.communications.inventory.api.entity.TopologyNode"
accessControlled="true" entityIdSequenceGenerator="TopologySeqGen">
    <implements interface="java.lang.Cloneable"/>
    <implements interface=
        "oracle.communications.inventory.api.entity.common.TopologyObject"/>
    <attribute name="isTopLevelNode" index="true"/>
    <attribute name="geometry" spatial="true"/>
    <relationship name="businessObject">
        <thisSide inverse="true"/>
        <otherSide dependent="true" type="ocim:TopNodeAssociation"
            attribute="topologyNode"/>
    </relationship>
</entity>
```

The TopologyEdge entity is also defined in the **topology-entities.xml** file in the same manner.

> **Note:** There are actually several topology entities defined in the **topology-entities.xml** file that support topology. However, within the context of extending topology, this chapter focuses solely on the TopologyEdge and TopologyNode entities.

Topology-Managed Entities

The metadata defines the following entities as topology-managed:

- Equipment
- GeographicPlace
- LogicalDevice
- Network
- NetworkEdge
- NetworkNode
- PhysicalDevice
- Pipe

The metadata defines these entities as topology-managed throughout the various ***-entities.xml** files. Example 6–2 is an excerpt from the **equipment-entities.xml** file. The example shows the entity definition for PhysicalDevice, which includes the implementation of the TopologyObject interface. Implementing the TopologyObject interface in the entity definition is what defines an entity as topology-managed.

Example 6–2 Topology-Managed Entity Definition

```
<entity type="ocim:PhysicalDevice"
interface="oracle.communications.inventory.api.entity.PhysicalDevice" accessControlled="true"
entityIdSequenceGenerator="PhyDeviceSeqGen">
    <implements interface="oracle.communications.inventory.api.entity.common.PhysicalResource"/>
    <implements interface="java.lang.Cloneable"/>
    <implements interface="oracle.communications.inventory.api.entity.common.TopologyObject"/>
    <implements interface=
        "oracle.communications.inventory.api.entity.common.PhysicalMappingObject"/>
    <implements interface="oracle.communications.inventory.api.entity.common.NetworkNodeEnabled"/>
```

```
    .
    .
    .
</entity>
```

About Topology Mapping

Entities defined as topology-managed in the metadata are mapped to either TopologyEdge or TopologyNode by the UIM-provided TopologyMapperImpl class.

TopologyEdge

The following topology-managed entities are mapped to TopologyEdge:

- NetworkEdge
- Pipe

TopologyNode

The following topology-managed entities are mapped to TopologyNode:

- Equipment
- GeographicPlace
- LogicalDevice
- Network
- NetworkNode
- PhysicalDevice

> **Note:** The GeographicPlace entity is defined as topology-managed in the UIM metadata, and the UIM mapping logic indirectly maps this entity to TopologyNode. The mapping logic actually checks for GeographicLocation and GeographicSite, not GeographicPlace. GeographicPlace is a parent to GeographicLocation and GeographicSite. A place becomes a topology object when it is associated to a resource such as Logical Device or Physical Device.

Extending the Topology

To extend the topology:

1. Determine entities that you plan to define as topology-managed. (This step is performed by the business analyst, who relays the information to the developer.)
2. Determine the mapping of each topology-managed entity to TopologyEdge or TopologyNode. (This step is performed by the business analyst, who relays the information to the developer.)
3. Define identified entities as topology-managed in the metadata by creating new **ext-*-entities.xml** files. See "Defining an Entity as Topology-Managed" for more information.
4. Regenerate the entities to pick up the new **ext-*-entities.xml** files. See "Applying Metadata Static Extensions" for more information.

5. Extend the mapping logic to include the mapping of any additional entities defined as topology-managed in the metadata. See "Extending the Mapping" for more information.

Defining an Entity as Topology-Managed

An entity can be defined as topology-managed through a new file in the metadata.

Caution: Do not modify existing metadata files. See "Backward Compatibility" for the issues involved with making additions to the existing metadata files.

To define a new entity as topology-managed, add the <implements> element to the entity definition in the new ***-entities.xml** file to implement the TopologyObject interface. See "Defining New Entities" for more information.

To define an existing entity as topology-managed, add the <implements> element to the entity by extending the entity definition in the new ***-entities.xml** file to implement the TopologyObject interface. See "Extending Existing Entities".

Extending the BusinessObjectType.java File

If you define an entity as topology-managed in the metadata, you must also extend the BusinessObjectType class by modifying it to include an enumerated value for that entity. This provides the ability to keep a weak reference between the topology entity and the business object.

For example, the BusinessObjectType class defines the BusinessObjectType enumeration, and you must assign an enumerated value to any entities you define as topology-managed:

```
/**
 * This class defines the business IDs for mapping Business objects to
 * TopologyEdges and TopologyNodes in the topology model.
 * Every different business entity must have a unique ID.
 * Once a value has been set it cannot be changed.
 */
public enum BusinessObjectType {
  LogicalDeviceDao(1), GeographicPlaceDao(2), PipeDao(3),
  PhysicalDeviceDao(4), NetworkDao(5), NetworkNodeDao(6),
  NetworkEdgeDao(7), EquipmentDao(8), PhysicalConnectorDao(9),
  PhysicalPortDao(10), EquipmentHolderDao(11), CustomObjectDao(12),
  ServiceDao(13), GeographicSiteDao(14), ServiceConfigurationVersionDao(15),
  TopologyOnly(9999);
```

Extending the Mapping

Entities defined as topology-managed in the metadata must be mapped to TopologyEdge or TopologyNode by extending the TopologyMapperImpl class.

This class is located in the oracle.communications.inventory.api.topology package.

Configuring the topologyProcess.properties file

If you extend the mapping, you must also configure the **topologyProcess.properties** file to point to your new mapper class.

For example, the file includes the following upon installation, and you must configure it to point to your new mapper class instead:

```
# mapperClass - The Class Object that maps the business model to Topology
mapperClass=oracle.communications.api.topology.mapper.impl.TopologyMapperImpl
```

About Path Analysis

Path analysis is an automated process in UIM that helps you locate and assign pipes for enablement. You specify a starting point (the source), an ending point (the target), and a variety of optional criteria. Path analysis evaluates possible paths based on the criteria you provide and returns paths from which you can select. See *UIM Concepts* for more information.

Path analysis uses the topology to find paths.

Configuring and Customizing Path Analysis

Path analysis evaluates connections based on topology-managed entity data. Only entities in the topology are included in path analysis. You can configure and customize path analysis, as described in the following sections.

Configuring the Path Analysis Mode

Path analysis can use two different algorithms to determine paths:

- The Complex algorithm (the default) considers all possible paths between end points, which means evaluating a large number of permutations. You can use filtering to limit the amount of data to be processed. This mode of path analysis is suitable for complex networks with many possible connections.
- The Simple Linear algorithm works by iteratively analyzing paths working from the end points toward a common node. This mode of analysis is suited to relatively simple scenarios where paths are inherently linear and include 10 or fewer hops, such as POTS. The Simple Linear algorithm has less impact on system performance than the Complex algorithm.

You can use the **topologyProcess.properties** file to configure path analysis. For example, the properties file includes the following upon installation:

```
# Path Analysis Properties
simpleLinearMode=false
simpleLinearModeMaxCycles=5
continueProcessingIndicator=true
```

- The **simpleLinearMode** parameter is used to denote the path analysis mode. The default value is **false**, indicating that Complex mode is the default path analysis mode.

  Note: Before changing the value of this parameter, you need to be certain that the Simple Linear mode is appropriate for your needs. Path analysis will not find some kinds of paths in this mode.

  You can extend path analysis so that Simple Linear mode is used when analyzing paths for particular pipe specifications, even when Complex mode is used for the application in general. See "Customizing Path Analysis" for more information.

- The **simpleLinearModeMaxCycles** parameter denotes the number of connected neighbors that a Simple Linear path analysis finds before determining that a path

cannot be found. The default value is **5**. You can increase the value if path analysis fails to find paths.

- The **continueProcessingIndicator** parameter denotes whether UIM will try to find a path with the Complex mode if no path can be found by using Simple Linear mode. The default value is **true**, indicating that if no path is found using Simple Linear mode, path analysis continues by attempting to find a path using Complex mode. Setting the value to **false** indicates that if no path is found using Simple Linear mode, path analysis stops.

Customizing Path Analysis

You can use rulesets to customize path analysis. By associating rulesets to individual Pipe specifications, you can tailor path analysis to meet various business scenarios.

A sample ruleset is provided with UIM to serve as a starting place for three types of customization:

- Adding additional filter criteria to the analysis. See "Adding Filtering Criteria" for more information.
- Setting Simple Linear mode for path analysis involving a particular Pipe specification. See "Setting the Analysis Mode" for more information.
- Specifying that only pipes based on particular specifications be included in a path analysis. See "Limiting the Analysis by Pipe Specification" for more information.

The PATHANALYSIS_FINDPATHS_SETCUSTOMCRITERIA sample ruleset is included in the *UIM_Home*/**cartridges/sample/ora_uim_pathanalysis_sample** cartridge.

You can customize path analysis by appending code to the body of the ruleset. The sample ruleset includes examples of each of the three types of customizations mentioned in this section.

The ruleset is applicable to Pipe specifications and must be associated with the PathAnalysisManager_findPaths base extension point and the oracle.communications.inventory.api.entity.PipeSpecification enabled extension point. The placement of the ruleset extension point must be BEFORE.

Adding Filtering Criteria

You can add filtering criteria to a path analysis. Filtering criteria restrict the amount of data that UIM considers when locating paths, reducing the amount of processing required.

> **Note:** Because the additional criteria are defined using standard JPAQL syntax, knowledge of JPAQL is required to implement this feature.

For example, you can limit the analysis to consider only nodes or edges that include particular characters in their names or only pipes in a particular status. Including the following code in the PATHANALYSIS_FINDPATHS_SETCUSTOMCRITERIA ruleset limits the path analysis to pipes in the Installed state.

```
filterStr.append("businessObject.referenceId == vPipe.ext:getColumn('ENTITYID')
");
filterStr.append(" && vPipe.adminState == pStatus ");
params.add("pStatus");
```

```
values.add(InventoryState.INSTALLED);
criteria.setAppendQuery (params, values, filterStr.toString());
```

Setting the Analysis Mode

You can configure path analysis to use Simple Linear mode when enabling pipes based on a particular specification. Including the following code in the PATHANALYSIS_FINDPATHS_SETCUSTOMCRITERIA ruleset sets the mode to Simple Linear when the ruleset runs. It also sets values for the **SimpleLinearModeMaxCycles** and **ContinueProcessingIndicator** parameters.

```
criteria.setSimpleLinearMode(true);
criteria.setSimpleLinearModeMaxCycles(10);
criteria.setContinueProcessingIndicator(true);
```

Limiting the Analysis by Pipe Specification

You can limit the pipe analysis so that it considers only transport pipes based on a particular specification. For example, you can filter out trunk and ISDN lines that are not valid connections for POTS. Similarly, if there are cables between a switch and an MDF that are not used for POTS, you can exclude them from the pipe analysis.

> **Note:** You can also limit path analysis to particular Pipe specifications by including a specification in the **Transport** configuration item of a Pipe configuration.

For example, including the follow code in the PATHANALYSIS_FINDPATHS_SETCUSTOMCRITERIA ruleset limits the path analysis to pipes based on the Sample Terminated Pipe specification:

```
SpecManager sm = InventoryHelper.makeSpecManager();
SpecSearchCriteria specCriteria = sm.makeSpecSearchCriteria();
CriteriaItem critSpecName = specCriteria.makeCriteriaItem();
critSpecName.setValue("SampleTerminatedPipe");
critSpecName.setOperator(CriteriaOperator.EQUALS_IGNORE_CASE);

specCriteria.setName(critSpecName);
List<Specification> specs = sm.findSpecifications(specCriteria);
ArrayList includeSpecs = new ArrayList();
for (Specification pipespec : specs){
     includeSpecs.add(new Long(pipespec.getEntityId()));
}
criteria.setIncludeSpecifications(includeSpecs);
```

About Topology Interfaces

You can use the topology interfaces when writing rulesets or web services to meet business requirements that involve extending the topology or customizing path analysis.

The following sections describe the available topology interfaces. For information on the methods defined by any of these interfaces, see the Javadoc. For instructions on how to access the Javadoc, see "Javadoc Documentation".

TopologyObject is the only topology interface described in this section that is available to all entities. Defining an entity to implement this interface makes the entity topology-managed. Topology-managed entities must be mapped to TopologyEdge or TopologyNode.

The remaining interfaces described in this section are available to TopologyEdge and TopologyNode entities. Example 6–3 is an excerpt from the **uim-common-entities.xml** file showing the common manager interfaces defined for the entities, including TopologyEdge and TopologyNode.

Example 6–3 uim-common-entities.xml Manager Interfaces

```
<manager interface="oracle.communications.inventory.api.framework.policy.SearchPolicy"
    class="oracle.communications.inventory.api.framework.policy.impl.SearchPolicyImpl"/>
<manager interface="oracle.communications.inventory.api.common.TransitionManager"
    class="oracle.communications.inventory.api.common.impl.TransitionManagerImpl"/>
<manager interface="oracle.communications.inventory.api.common.AttachmentManager"
    class="oracle.communications.inventory.api.common.impl.AttachmentManagerImpl"/>
<manager interface="oracle.communications.inventory.api.common.SequenceGenerator"
    class="oracle.communications.inventory.api.common.impl.SequenceGeneratorImpl"/>
<manager interface="oracle.communications.inventory.api.consumer.ConsumerManager"
    class="oracle.communications.inventory.api.consumer.impl.ConsumerManagerImpl"/>
<manager interface="oracle.communications.inventory.api.consumer.AssignmentManager"
    class="oracle.communications.inventory.api.consumer.impl.AssignmentManagerImpl"/>
<manager interface="oracle.communications.inventory.api.common.ConfigurationInputManager"
    class="oracle.communications.inventory.api.common.impl.ConfigurationInputManagerImpl"/>
<manager interface="oracle.communications.inventory.api.consumer.ConditionManager"
    class="oracle.communications.inventory.api.consumer.impl.ConditionManagerImpl"/>
<manager interface="oracle.communications.inventory.api.consumer.ReservationManager"
    class="oracle.communications.inventory.api.consumer.impl.ReservationManagerImpl"/>
<manager interface="oracle.communications.inventory.api.common.FederationManager"
    class="oracle.communications.inventory.api.common.impl.FederationManagerImpl"/>
<manager interface="oracle.communications.inventory.api.common.EntityIdGenerator"
    class="oracle.communications.inventory.api.common.impl.EntityIdGeneratorImpl"/>
<manager interface="oracle.communications.inventory.api.admin.SecurityManager"
    class="oracle.communications.inventory.api.admin.impl.SecurityManagerImpl"/>
<manager interface="oracle.communications.inventory.api.topology.TopologyManager"
    class="oracle.communications.inventory.api.topology.impl.TopologyManagerImpl"/>
<manager interface="oracle.communications.inventory.api.topology.mapper.TopologyMapper"
    class="oracle.communications.inventory.api.topology.mapper.impl.TopologyMapperImpl"/>
<manager interface="oracle.communications.inventory.api.topology.PathAnalysisManager"
    class="oracle.communications.inventory.api.topology.impl.PathAnalysisManagerImpl"/>
<manager interface="oracle.communications.inventory.api.topology.mapper.PathAnalysisMapper"
    class="oracle.communications.inventory.api.topology.mapper.impl.PathAnalysisMapperImpl"/>
<manager interface="oracle.communications.inventory.api.topology.mapper.TopologyProfileMapper"
    class="oracle.communications.inventory.api.topology.mapper.impl.TopologyProfileMapperImpl"/>
<manager interface="oracle.communications.inventory.api.capacity.CapacityManager"
    class="oracle.communications.inventory.api.capacity.impl.CapacityManagerImpl"/>
<manager interface="oracle.communications.inventory.api.characteristic.CharacteristicManager"
    class="oracle.communications.inventory.api.characteristic.impl.CharacteristicManagerImpl"/>
<manager interface="oracle.communications.inventory.api.role.RoleManager"
    class="oracle.communications.inventory.api.role.impl.RoleManagerImpl"/>
<manager interface="oracle.communications.inventory.api.common.RowLockManager"
    class="oracle.communications.inventory.api.common.impl.RowLockManagerImpl"/>
<manager interface="oracle.communications.inventory.api.framework.policy.LockPolicy"
    class="oracle.communications.inventory.api.framework.policy.impl.LockPolicyImpl"/>
```

TopologyObject

Package: oracle.communications.api.inventory.entity.common

This interface defines getter methods for the object's IDs: ID, ENTITYID, and OID. There are no setter methods because these IDs are generated for the object, not set for the object.

TopologyManager

Package: oracle.communications.inventory.api.topology

This interface defines methods for finding and maintaining TopologyEdge and ToplogyNode entity objects.

TopologyMapper

Package: oracle.communications.inventory.api.topology.mapper

This interface defines the business rules for mapping topology-managed entity objects to a TopologyEdge entity object or a TopologyNode entity object.

PathAnalysisManager

Package: oracle.communications.inventory.api.topology

This interface defines methods for finding paths (edges and nodes) through the topology network based on specified criteria.

PathAnalysisMapper

Package: oracle.communications.inventory.api.topology.mapper

This interface defines the business rules for mapping business object path analysis criteria to values used in the topology model. This object provides a mapping layer between the business model and the topology model for cases where the data in the topology model must be converted from a value in the business model.

TopologyProfileMapper

Package: oracle.communications.inventory.api.topology.mapper

This interface defines mapping for service topology. While topology is extended through the metadata, service topology is extended through characteristics, specifications, extension points, and rulesets, all of which can be defined in Oracle Communications Design Studio. UIM provides a service topology sample cartridge that is a working example of how you could extend service topology. See *UIM Cartridge Guide* for more information on the service topology sample cartridge.

TopologyEdgeSearchCriteria

Package: oracle.communications.inventory.api.topology

This interface defines the available search criteria for the TopologyEdge entity object and is an input parameter to topology manager and topology mapper interface methods.

TopologyNodeSearchCriteria

Package: oracle.communications.inventory.api.topology

This interface defines the available search criteria for the TopologyNode entity object and is an input parameter to topology manager and topology mapper interface methods.

About the topologyProcess.properties File

Topology logic references the *UIM_Home*/**config/resources/event/topologyProcess.properties** file for specifying the mapper class and for configuring path analysis. You can also use this file to:

- Turn off topology updates. If you turn off topology updates, you can rebuild the topology if you need to use a topology-related feature. See *UIM System Administrator's Guide* for more information.

  For example, the file includes the following upon installation:

  ```
  # disableTopology - turns Topology Refresh On or Off
  disableTopology=false
  ```

- Opt whether to update the topology synchronously or asynchronously with business model updates. See *UIM System Administrator's Guide* for more information.

  For example, the file includes the following upon installation:

  ```
  # processSynchronous - Topology is refreshed as part of the transaction (true)
  # or asynchronoulsy in a seperate transaction (false)
  processSynchronous=true
  ```

7

Extending Security

This chapter provides information on extending Oracle Communications Unified Inventory Management (UIM) security to include APIs and entity data.

Security for other parts of UIM is handled by external systems, such as the Oracle WebLogic Server Administration Console and Oracle Enterprise Manager. See *UIM System Administrator's Guide* for more information.

> **Note:** For information on securing web services, see *UIM Web Services Developer's Guide*.

Securing APIs

By default, UIM APIs are not secured. To secure an API, you must extend UIM security to include the APIs. This can be done by:

- Securing APIs through the SecurityValidation Aspect
- Securing APIs through Rulesets and Extension Points

Securing APIs through the SecurityValidation Aspect

You can secure access to an API by adding the API method to the UIM-provided security extension point (securityExtensionPoint) definition, which is defined within the SecurityValidation aspect in the **aop.xml** file. See Chapter 8, "Extending UIM Through Rulesets" for more information about aspects and the **aop.xml** file.

At the framework level, security is automatically enforced at the security extension point for any methods that the extension point defines. For example, if no API methods are defined for the security extension point within the SecurityValidation aspect, then no APIs are secured. If 20 API methods are defined for the security extension point within the SecurityValidation aspect, then those 20 API methods are validated/secured.

Example 7–1 shows API security definitions that are provided as a comment in the **aop.xml** file. If uncommented, these definitions would secure the createConditions, updateConditions, and deleteConditions APIs using the SecurityValidation aspect through the specified extension point (securityExtensionPoint). The result of this entry in the **aop.xml** file is that security validations are run prior to every call to the createConditions, updateConditions, and deleteConditions APIs.

You can use this example as a starting point by modifying it and uncommenting it in the **aop.xml** file to secure any API.

Example 7–1 SecurityValidation Aspect

```
<concrete-aspect
    name="oracle.communications.extensibility.extension.SecurityValidation"
    extends=
    "oracle.communications.extensibility.extension.SecurityValidationExtension" >
    <pointcut name="securityExtensionPoint" expression="
          call(public *
               oracle.communications.inventory.api.consumer.ConditionManager.
                   createConditions(java.util.Collection))
          call(public *
               oracle.communications.inventory.api.consumer.ConditionManager.
                   updateConditions(java.util.Collection))
          call(public *
               oracle.communications.inventory.api.consumer.ConditionManager.
                   deleteConditions(java.util.Collection))"/>
</concrete-aspect>
```

Creating the Global Extension Point

Global extension points are created in Oracle Communications Design Studio. For information on global extension points, see Chapter 8, "Extending UIM Through Rulesets". For instructions on how to create a global extension point, see the Design Studio Help.

When using this approach to secure APIs, you must also create one global extension point that defines the handleSecurityViolation API, which enables the rulesets to generate errors. The handleSecurityViolation API is located in the oracle.communications.inventory.api.admin.SecurityManager package. Example 7–2 shows the API method signature to use when defining the global extension point for the handleSecurityViolation API.

Example 7–2 Custom Global Extension Point Signature

```
public void oracle.communications.inventory.api.admin.SecurityManager.
handleSecurityViolation([])
```

Creating the Global Ruleset Extension Point

Global ruleset extension points are created in Design Studio. For information on global ruleset extension points, see Chapter 8, "Extending UIM Through Rulesets". For instructions on how to create a global ruleset extension point, see the Design Studio Help.

After you have created the ruleset and global extension point in Design Studio, you must also create the corresponding global ruleset extension point in Design Studio. A global ruleset extension point associates a ruleset with a global extension point, so the global extension point knows which ruleset to run.

Securing APIs through Rulesets and Extension Points

You can also secure access to an API by creating custom rulesets that run at specified extension points. The custom rulesets set permissions for an API, enforces any permissions that are set for an API, and logs error messages whenever a security violation is detected.

Setting and enforcing API permissions through rulesets is done in the same manner as setting and enforcing entity data permissions. See "Securing Entity Data through Rulesets and Extension Points" for more information.

Securing Entity Data

By default, UIM entity data is not secured. To secure entity data, you must extend UIM security to control data access to individual entities. This is done by creating custom rulesets that run at specified extension points. The custom rulesets set permissions or partitions for an entity, enforces any permissions or partitions that are set for an entity, and logs error messages whenever a security violation is detected.

About Entity Access Control

To configure access control for an entity, the entity must be declared as access-controlled in the metadata. For example, the following is an excerpt from the metadata that shows the Equipment entity definition, which is declared as access-controlled:

```
<entity type="ocim:Equipment"
interface="oracle.communications.inventory.api.entity.Equipment"
accessControlled="true">
```

Most, but not all, entities are declared as access-controlled. If you want to configure access control for an entity that is not declared as access-controlled in the metadata, you must first extend the data model to declare the entity as access-controlled. See Chapter 4, "Extending the Data Model" for more information.

Access-controlled entities define additional attributes that contain security-specific data. For example, access-controlled entities define the **owner**, **permissions**, and **partition** attributes. Access-controlled entities also extend the AccessControlled class, so each entity class has access to the setOwner(), setPermissions(), and setPartition() methods defined in the AccessControlled parent class. The value of these attributes can be set by custom rulesets that call these methods.

> **Note:** When controlling access to a range of entities, the ruleset custom code must iterate through the range and call the method for each entity in the range. See "Securing Entity Data for a Range of Entities Example" for more information.

Securing Entity Data through Rulesets and Extension Points

You can secure entity data through rulesets and extension points by:

- Setting Permissions in a Custom Ruleset
- Setting Partitions in a Custom Ruleset
- Enforcing Security in a Custom Ruleset

Setting Permissions in a Custom Ruleset

> **Note:** This section also applies to securing APIs through permissions.

To control data access to an entity through permissions, set the **permissions** attribute for the entity through custom code that calls the setPermissions() method, which is defined as:

```
public void setPermissions(String acl);
```

This method is defined in the oracle.communications.inventory.api.AccessControlled class, which is the parent class of all entities that are declared as access-controlled in the metadata. In the custom ruleset, you can call this method on the parent class (AccessControlled) or on the child class (*EntityName*, such as TelephoneNumber, Equipment, and so forth).

See "Creating Custom Rulesets and Extension Points" for examples of setPermissions() method calls.

Understanding ACL

The permissions are defined as an access control list (ACL). The ACL is a Java string that specifies who is allowed to access an object and what operations they can perform on an object.

An ACL consists of one or more entry statements separated by semicolons. Each statement includes the type of permission (**allow** or **deny**), the permission (**r** for read or **w** for write), and a principal or role to whom the permission is granted. (A principal is a user or group. It is easier to manage permissions at the level of roles, however.)

The syntax is as follows:

```
allow|deny r|w = principal|roles[role1,role2,role3...];
```

where *principal* is the name of any user or group and *role* is the name of any role.

Example 7–3 shows the ACL syntax in Extended Backus-Naur Format (EBNF).

Example 7–3 ACL Syntax

```
acl:= acl_entry (';'acl_entry)*
acl_entry:=('allow'|'deny')permission? target_list
permission:= ('r'|'w')'='
target_list:= target (','target)*
target:= principal|'roles' '['role_list']'
role_list:= role(','role)*
```

Note the following about the ACL:

- The ACL is evaluated left to right until a security decision of **allow** or **deny** is enforced.
- If no permission is stated, **allow** is implied.

  Allowing write access implies allowing read access.
- Denying read access also implies denying write access.
- Any user having the **uimuser** role is permitted full access to an entity, regardless of the permissions set for the entity. This role exists by default and is defined as a superuser.

Table 7–1 lists examples of permissions and how they work together.

Table 7–1 Examples of Permissions

Permissions	Explanation
Allow roles[billing_admin]; deny all	Anyone assigned to the billing_admin role can read or write the entity, but no one else.
Allow all	Everyone can read or write the entity. The same can be achieved by simply not defining permissions for the entity.

Table 7–1 (Cont.) Examples of Permissions

Permissions	Explanation
Allow r=all,w=roles[location_admin]	Everyone can read the entity; anyone having the location_admin role can write the entity.
Deny all	No one may can the entity except superusers.
Deny w=all	No one can write the entity except superusers, but everyone may read the entity.
Deny roles[OrderEntryUser,GeoMapAdmin User]	Anyone having either the OrderEntryUser or the GeoMapAdminUser role is denied access. Everyone else has full access.

Setting Partitions in a Custom Ruleset

To control data access to an entity through partitions, set the **partition** attribute for the entity through custom code that calls the setPartition() method, which is defined as:

```
public void setPartition(String partition);
```

This method is defined in the oracle.communications.inventory.api.AccessControlled class, which is the parent class of all entities that are declared as access-controlled in the metadata. In the custom ruleset, you can call this method on the parent class (AccessControlled) or on the child class (*EntityName*, such as TelephoneNumber, Equipment, and so forth).

See "Creating Custom Rulesets and Extension Points" for examples of setPartition() method calls.

Configuring Partitions

To control data access to an entity through partitions, some additional configuration is required:

1. In the WebLogic Server Administration Console, you must define a user group within a security realm. The group you define represents a data partition in UIM. For instructions on how to do this, see *UIM System Administrator's Guide*.

   Caution: The group name must begin with **ora_uim_partition#** to be recognized by UIM. For example, if you define a group name of **ora_uim_partition#myPartition**, then the custom ruleset would set the partition to **/myPartition**.

2. In the *UIM_Home*/**config/system-config.properties** file, set the **uim.security.filter.enabled** property to **true**, as shown here:

   ```
   uim.security.filter.enabled=true
   ```

Enforcing Security in a Custom Ruleset

Note: This section also applies to enforcing security permissions set for APIs.

API access that is controlled through set permissions, and entity data access that is controlled through set permissions and partitions is enforced through custom code that calls the checkPermissions() method, which is defined as:

```
public void checkPermissions(String perm, AccessControlled instance);
```

This method is defined in the oracle.communications.inventory.api.framework.security.UserEnvironment class. The checkPermissions() method calls the hasAccessToPartition() method, so the checkPermissions verifies access for both permissions and partitions.

If a security violation is detected, the application throws a java.security.AccessControlException. The custom code catches and logs the AccessControlException by calling the error() method, which is defined as:

```
public void error(String s, Throwable t);
```

This method is defined in the oracle.communications.inventory.api.framework.logging.Log class.

See "Creating Custom Rulesets and Extension Points" for examples of error() method calls.

Creating Custom Rulesets and Extension Points

When using custom rulesets to secure an API or entity data, you must also create an extension point or global extension point to run the ruleset. The following sections provide additional information and examples for creating the ruleset and extension point. If creating a global extension point, see "Creating the Global Extension Point" for more information.

Creating Custom Rulesets

Rulesets are created in Oracle Communications Design Studio. Rulesets can be written using Drools or Groovy. This section provides several custom ruleset examples, and each example is shown twice; once using Drools and once using Groovy. For information on rulesets, and using Drools and Groovy, see Chapter 8, "Extending UIM Through Rulesets". For instructions on how to create a ruleset, see the Design Studio Help.

Note: In the following custom ruleset examples, all import statements are omitted.

Securing APIs Example

Example 7–4 shows a custom ruleset that secures access to the createConditions, updateConditions, and deleteConditions APIs by setting permissions. The ruleset defines four rules:

- Default Condition Validation Rule

  This rule always runs and calls the validate() method, which simply logs the method name and logs the user that is calling the method.

- Create Condition Validation Rule

  This rule runs only when the ruleset is called from an extension point that defines the createConditions API. This rule calls the setConditionsOwner() method, which sets permissions.

- Update Condition Validation Rule

This rule runs only when the ruleset is called from an extension point that defines the updateConditions API. This rule calls the validateConditionsOwner() method, which enforces security and logs an error if a security violation is detected.

- Delete Condition Validation Rule

  This rule runs only when the ruleset is called from an extension point that defines the deleteConditions API. This rule also calls the validateConditionsOwner() method, which enforces security and logs an error if a security violation is detected.

Note: Example 7–4 uses the context.getArguments() method. However, depending on how you configure your custom ruleset to run (**before**, **after**, or **instead** of the method your extension point defines), you may need to use the context.getReturnValue() method instead.

For example, when the ruleset runs **before** the method the extension point defines, use context.getArguments() because the return value is always empty in this scenario. When the ruleset runs **after** the method the extension point defines, use context.getReturnValue() because the data in the context argument that was passed to the ruleset may have changed through the use of context.setArguments().

See "ExtensionPointRuleContext.returnValue" for more information.

Example 7–4 Custom Ruleset Using Drools

```
package oracle.communications.rules;
  .
  .
  .
global Log log;

//--------------------------------------------------------------------
// FUNCTIONS
//--------------------------------------------------------------------
function void validate(ExtensionPointRuleContext context, Log logger,
UserEnvironment env) {
    logger.info("", new String[]{"********"});
    logger.info("", new String[]{"method: ", context.getMethodName()});
    logger.info("", new String[]{"user: ", env.getUserName()});
    logger.info("", new String[]{"********"});
}

function void setConditionsOwner(ExtensionPointRuleContext context, Log logger,
UserEnvironment env) {
    logger.info("", new String[]{"********"});
    logger.info("", new String[]{"setConditionsOwner"});
    Collection conditions = (Collection) context.getArguments()[0];
    if (conditions != null && !conditions.isEmpty())
    {
        String owner = env.getUserName();
        for (Iterator itr = conditions.iterator(); itr.hasNext();) {
            Condition cond = (Condition) itr.next();
            if (cond instanceof AccessControlled) {
                ((AccessControlled)cond).setOwner( owner );
                ((AccessControlled)cond).setPermissions("deny contractEmployees");
            }
```

```
        }
    }
    logger.info("", new String[]{"********"});
}

function void validateConditionsOwner(ExtensionPointRuleContext context, Log
logger, UserEnvironment env) {
    logger.info("", new String[]{"********"});
    logger.info("", new String[]{"validateConditionsOwner"});
    Collection conditions = (Collection) context.getArguments()[0];

    String methodName = context.getMethodName();
    String targetName = context.getDeclaringTargetType().getSimpleName();
    String policyName = targetName + "." + methodName;
    logger.info("", new String[]{"policyName: ", policyName});

    if (conditions != null && !conditions.isEmpty()) {
        for (Iterator itr = conditions.iterator(); itr.hasNext();) {
            Condition cond = (Condition) itr.next();
            if (cond instanceof AccessControlled) {
                try {
                    env.checkPermissions( policyName, (AccessControlled) cond );
                }
                catch (java.security.AccessControlException ace) {
                    logger.error("", new String[] {ace.getMessage()});
                    logger.error("", new String[] {"My error message for: " +
cond.toString()});
                }
            }
        }
    }
    logger.info("", new String[]{"********"});
}

//-------------------------------------------------------------------------
// RULES
//-------------------------------------------------------------------------
rule "Default Condition Validation Rule"
    salience 10
    when
        context: ExtensionPointRuleContext()
    then
        UserEnvironment env = UserEnvironmentFactory.getUserEnvironment();
        RuleDebug.breakPoint(context);
        RuleDebug.breakPoint(env);
        validate(context, log, env);
end

rule "Create Condition Validation Rule"
    salience 1
    when
        context: ExtensionPointRuleContext(methodName == "createConditions")
    then
        UserEnvironment env = UserEnvironmentFactory.getUserEnvironment();
        setConditionsOwner(context, log, env);
end

rule "Update Condition Validation Rule"
    salience 1
    when
```

```
        context: ExtensionPointRuleContext(methodName == "updateConditions")
    then
        UserEnvironment env = UserEnvironmentFactory.getUserEnvironment();
        validateConditionsOwner(context, log, env);
end


rule "Delete Condition Validation Rule"
    salience 1
    when
        context: ExtensionPointRuleContext(methodName == "deleteConditions")
    then
        UserEnvironment env = UserEnvironmentFactory.getUserEnvironment();
        validateConditionsOwner(context, log, env);
end
```

Example 7–5 shows the same custom ruleset content-wise, but using Groovy instead of Drools. Only the rules section is shown. (The functions section content is the same; the only difference between Drools and Groovy is the syntax of the function definition itself: Drools uses the key word **function** and Groovy uses the key word **def**.)

Example 7–5 Custom Ruleset Using Groovy

```
//--------------------------------------------------------------------
// RULES
//--------------------------------------------------------------------

UserEnvironment env = UserEnvironmentFactory.getUserEnvironment();
String methodName = ExtensionPointRuleContext.getMethodName();

if (methodName == "createConditions") {
    setConditionsOwner(context, log, env); }
else {
    if (methodName == "updateConditions") {
        validateConditionsOwner(context, log, env); }
    else {
        if (methodName == "deleteConditions") {
            validateConditionsOwner(context, log, env); }
        else {
            validate(context, log, env);
        }
    }
}
```

Securing Entity Data through Permissions Example

Example 7–6 shows a custom ruleset that secures access to party entities by setting permissions. The ruleset name implies that it is intended to run when a party is created.

Example 7–6 Custom Ruleset Using Drools

```
package oracle.communications.inventory.rules;
  .
  .
  .
global Log log;

rule "Create Party with Permissions"
salience 2
```

```
when
    partyList : Collection()
then
    UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
    if ((partyList != null) && !(partyList.isEmpty())) {
        for (Object partyO : partyList ) {
            Party party = (Party)partyO;
            party.setOwner("inv");
            party.setPermissions("allow inv; deny all");
        }
    }
end
```

Example 7–7 shows the same custom ruleset content-wise, but using Groovy instead of Drools. Only the rules section is different, so that is all that is shown.

Example 7–7 Custom Ruleset Using Groovy

```
UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
if ((partyList != null) && !(partyList.isEmpty())) {
    for (Object partyO : partyList ) {
        Party party = (Party)partyO;
        party.setOwner("inv");
        party.setPermissions("allow inv; deny all");
    }
}
```

Retrieving Permissions Information Example

Example 7–8 shows a custom ruleset that retrieves user and role information so you can view the permissions that are set for a user through roles.

Example 7–8 Custom Ruleset Using Drools

```
package oracle.communications.inventory.rules;
  .
  .
  .
rule "Get Permissions Info"
salience 2
when
    true
then
    UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
    String user = env.getUser();
    String userName = env.getUserName();
    Collection roles = env.getRoles();
end
```

Example 7–9 shows the same custom ruleset content-wise, but using Groovy instead of Drools. Only the rules section is different, so that is all that is shown.

Example 7–9 Custom Ruleset Using Groovy

```
UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
String user = env.getUser();
String userName = env.getUserName();
Collection roles = env.getRoles();
```

Securing Entity Data through Partitions Example

Example 7–10 shows a custom ruleset that secures access to logical device entities by setting a partition. The ruleset name implies that it is intended to run when a logical device is created.

Example 7–10 Custom Ruleset Using Drools

```
package oracle.communications.inventory.rules;
  .
  .
  .
global Log log;
rule "Create LogicalDevice with Partitions"
salience 2
when
    ldList : Collection()
then
    UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
    if ((ldList != null) && !(ldList.isEmpty())) {
        for (Object ld : ldList ) {
            ((LogicalDevice)ld).setPartition("/US_PARTITION/NY_PARTITION");
        }
    }
end
```

Example 7–11 shows the same custom ruleset content-wise, but using Groovy instead of Drools. Only the rules section is different, so that is all that is shown.

Example 7–11 Custom Ruleset Using Groovy

```
UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
if ((ldList != null) && !(ldList.isEmpty())) {
    for (Object ld : ldList ) {
            ((LogicalDevice)ld).setPartition("/US_PARTITION/NY_PARTITION");
    }
}
```

Securing Entity Data for a Range of Entities Example

When securing entity data for a range of entities, the ruleset custom code must iterate through the range and call the access control method for each entity in the range. To do this, you must configure your custom ruleset to run **After** the API call.

Example 7–12 shows a custom ruleset that secures access to a range of logical devices by iterating through the range of logical devices, and setting a partition for each logical device in the range. The ruleset name implies that it is intended to run when a range of logical devices are created.

> **Note:** Example 7–12 shows the use of the setPartition() method to secure entity data for a range, but the same concept applies when using the setOwner() or setPermissions() methods to secure entity data for a range.

Example 7–12 Custom Ruleset Using Drools

```
package oracle.communications.inventory.rules;
  .
  .
  .
```

```
global Log log;

rule "Create Range of LogicalDevices with Partitions"
salience 2
when
    ldList : Collection()
then
    UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
    if ((ldList != null) && !(ldList.isEmpty())) {
        for (Object obj : ldList ) {
            LogicalDevice ld = (LogicalDevice)obj;
            ld.setPartition("/US_PARTITION/NY_PARTITION");
        }
    }
end
```

Example 7–13 shows the same custom ruleset content-wise, but using Groovy instead of Drools. Only the rules section is different, so that is all that is shown.

Example 7–13 Custom Ruleset Using Groovy

```
UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
if ((ldList != null) && !(ldList.isEmpty())) {
    for (Object obj : ldList ) {
        LogicalDevice ld = (LogicalDevice)obj;
        ld.setPartition("/US_PARTITION/NY_PARTITION");
    }
}
```

Enforcing Security Example

Example 7–14 shows a custom ruleset that enforces security access to a party. The ruleset name implies that it is intended to run when a party is updated.

Example 7–14 Custom Ruleset Using Drools

```
package oracle.communications.inventory.rules;
  .
  .
  .
global Log log;

rule "Secure Update Party"
salience 2
when
    partyList : Collection()
then
    UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
    if ((partyList != null) && !(partyList.isEmpty()))
    {
        for (Object party0 : partyList ) {
            Party party = (Party)party0;
            try {
                environment.checkPermissions
                    (WritePermission.getInstance().toString(), party);
            }
            catch(Throwable t){
                log.error("", t);
            }
        }
```

```
    }
end
```

Example 7–15 shows the same custom ruleset content-wise, but using Groovy instead of Drools. Only the rules section is different, so that is all that is shown.

Example 7–15 Custom Ruleset Using Groovy

```
UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
if ((partyList != null) && !(partyList.isEmpty()))
{
    for (Object party0 : partyList ) {
        Party party = (Party)party0;
        try {
           environment.checkPermissions
              (WritePermission.getInstance().toString(), party);
        }
        catch(Throwable t){
            log.error("", t);
        }
    }
}
```

Creating Extension Points

> **Note:** Check the **ora_uim_baseextpts** cartridge to determine if any extension points you may need are already defined. Depending on what you are securing, you may or may not need to create new extension points.

Extension points are created in Design Studio. For information on extension points, see Chapter 8, "Extending UIM Through Rulesets". For instructions on how to create an extension point, see the Design Studio Help.

When securing APIs, you must create one extension point per API to secure, where each extension point defines the specific API method to secure. In the same vein, when securing entity data, you must create one extension point per entity to secure, where each extension point defines the specific entity method to secure.The same ruleset can be called from multiple extension points. Example 7–16 shows the API method signatures to use when defining the extension point for each API secured by the custom ruleset shown in Example 7–4.

Example 7–16 Custom Extension Point Signatures

```
public void
oracle.communications.inventory.api.consumer.ConditionManager.createConditions
(java.util.Collection)

public void
oracle.communications.inventory.api.consumer.ConditionManager.updateConditons
(java.util.Collection)

public void
oracle.communications.inventory.api.consumer.ConditionManager.deleteConditions
(java.util.Collection)
```

Creating the Ruleset Extension Point

Ruleset extension points are created in Design Studio. For information on ruleset extension points, see Chapter 8, "Extending UIM Through Rulesets". For instructions on how to create a ruleset extension point, see the Design Studio Help.

After you have created the ruleset and extension point in Design Studio, you must also create the corresponding ruleset extension point in Design Studio. A ruleset extension point associates a ruleset with an extension point, so the extension point knows which ruleset to run.

8

Extending UIM Through Rulesets

This chapter provides information about extending Oracle Communications Unified Inventory Management (UIM) by using rulesets.

About Using Rulesets to Extend UIM

A ruleset is a file containing custom code that extends existing UIM code at a specified point. The UIM extensibility framework supports the use of rulesets and is built upon the open source project of AspectJ. Ruleset code is written using either Drools or Groovy, both of which are also open source projects.

AspectJ is an Eclipse open source project that enables aspect-oriented programming (AOP). AOP provides the ability to insert code at various points across a code base. For example, when UIM is started, the AspectJ engine weaves (interlaces) custom extension points into the UIM code stream. The AspectJ engine is called the Weaver.

Drools and Groovy are open source projects that enable accessing, changing, and managing business rules. Both enable you to view business rules because they are completely decoupled from the code. This allows for greater flexibility in changing, adding, or removing rules as business needs change.

About Rules

Rules are written using either Drools or Groovy. A ruleset contains one or more rules. The following sections describe rules in terms of Drools and Groovy.

Using Drools to Define Rules

A Drools rule is a two-part structure that defines a condition and an action. When the condition evaluates to true, the action occurs. While the entire rule is custom code, the action is the business-specific custom code. This chapter uses the term *custom code* to refer to the business-specific custom code.

Example 8–1 shows the structure of a Drools rule:

Example 8–1 Structure of a Drools Rule

```
rule "RuleName"
    salience 0
when
    condition
then
    action
end
```

> **Note:** You do not need to specify a condition in the rule. If no condition is specified, the condition is assumed to be true and the action occurs.

The custom code may reside:

- In the action
- In a function within the ruleset that the action calls
- In a separate Java class that the action calls

If the custom code is short and simple, you can place it in the action, or within a function in the ruleset. If the custom code is even slightly complex, Oracle recommends that you place it in a separate Java class. The advantage of placing custom code in a separate Java class is that you can use the use of the Java features. For example, the Java editor catches syntax errors, creates import statements, and provides a list of method names when you type a class name.

> **Note:** A ruleset always runs, but this does not mean that the custom code always runs. Whether or not the custom code runs depends on the outcome of the rule conditions.

A rule can optionally define soft keywords, one of which is **salience**. Example 8–1, "Structure of a Drools Rule" includes the salience soft keyword, which is commonly used.

Salience defines the priority of when a rule runs, which is necessary when multiple rules run at the same time. Salience is a numeric value that defaults to zero; the higher the number, the higher the priority. The rule with the highest priority runs first.

For more information about soft keywords, see the topic of keywords on the *Drools Documentation* website:

`http://docs.jboss.org/drools/release/6.5.0.Final/drools-docs/html_single/`

Using Groovy to Define Rules

Unlike Drools rules that define a condition and an action, a Groovy rule defines only an action (the business-specific custom code). You define Groovy rules in Groovy scripts.

Example 8–2 shows the structure of a Groovy script:

Example 8–2 Structure of a Groovy Script

```
def x = "World"
println "Hello, $x"
if (condition) action
```

The custom code may reside:

- In the rule (action)
- In a function within the ruleset that the rule (action) calls
- In a separate Groovy script that the rule (action) calls

If the custom code is short and simple, you can place it in the rule, or within a function in the ruleset. If the custom code is even slightly complex, Oracle recommends that you place it in a separate Groovy script. The advantage of placing custom code in a separate Groovy script is having the full use of the Eclipse debugging functionality. For example, the debugger may not stop on a breakpoint in the rule Groovy code itself, but will stop on a breakpoint within a Groovy script that the parent script calls. See "Debugging Custom Groovy Rulesets" for more information.

About Rulesets and Extension Points

A ruleset is a file that contains one or more rules. UIM provides several base rulesets in the **ora_uim_baserulesets** cartridge and also provides a framework that enables you to create custom rulesets. Each base ruleset provides a Drools version and a Groovy version of the ruleset. See "About Base Rulesets" for more information.

In addition to rules, a ruleset may also contain:

- A package statement
- Import statements
- Global and local variables
- Functions (similar to Java methods)

The content of a ruleset file is similar to the content of a Java source file (**.java**), but a ruleset is either a Drools file (**.drl**) or a Groovy file (**.groovy**).

Example 8–3 shows a ruleset that contains a package statement, import statements, a global variable, a function, and a Drools rule. Within a ruleset, functions must be defined prior to the rule so the rule can recognize the function when compiling. When a ruleset runs, execution begins at the rule; it does not begin at any functions that may be defined prior to the rule.

Example 8–3 Drools Ruleset

```
package oracle.communications.inventory.rules

import oracle.communications.inventory.api.entity.TelephoneNumberSpecification;
import oracle.communications.inventory.extensibility.extension.util.
ExtensionPointRuleContext;
import oracle.communications.inventory.api.framework.logging.Log;

global Log log;

//--------------------------------------------------------------------
// FUNCTIONS
//--------------------------------------------------------------------
function String getEditMask(TelephoneNumberSpecification tnSpec)
{
    // Set the default edit mask.
    String editMask = "##########";
    if ( tnSpec == null )
        return editMask;

    // Set the edit mask based on specification name
    if(tnSpec.getName().equals("US TN Spec NPA-NXX"))
        editMask = "###-###-####";
    return editMask;
}
```

```
//-------------------------------------------------------------------
// RULES
//-------------------------------------------------------------------
rule "Get TN Edit Mask"
    salience 0
when
    telephoneNumberSpecification : TelephoneNumberSpecification()
    context : ExtensionPointRuleContext()
then
    String editMask = getEditMask(telephoneNumberSpecification);
    context.setReturnValue(editMask);
end
```

Example 8–4 shows a ruleset that contains a package statement, import statements, a global variable, a function, and a Groovy rule. Example 8–3 and Example 8–4 provide the same functional result, but one was written using Drools and the other using Groovy.

Example 8–4 Groovy Ruleset

```
package oracle.communications.inventory.rules

import oracle.communications.inventory.api.entity.TelephoneNumberSpecification;
import oracle.communications.inventory.extensibility.extension.util.
ExtensionPointRuleContext;
import oracle.communications.inventory.api.framework.logging.Log;

global Log log;

//-------------------------------------------------------------------
// RULE CODE
//-------------------------------------------------------------------
String editMask = getEditMask(telephoneNumberSpecification);
context.setReturnValue(editMask);

//-------------------------------------------------------------------
// FUNCTION
//-------------------------------------------------------------------
def String getEditMask(TelephoneNumberSpecification tnSpec)
{
    // Set the default edit mask.
    String editMask = "##########";
    if ( tnSpec == null )
        return editMask;

    // Set the edit mask based on specification name
    if(tnSpec.getName().equals("US TN Spec NPA-NXX"))
        editMask = "###-###-####";
    return editMask;
}
```

Rulesets enable you to run custom code that extends UIM code at specified points called extension points.

Rulesets:

- Use Drools or Groovy rules and are enabled by AspectJ
- Functionally extend UIM through custom code

- Dynamically extend UIM through additions, changes, or deletions to rulesets without rebuilding or restarting UIM
- Are provided in the **ora_uim_baserulesets** cartridge (base rulesets)
- Can be created in Oracle Communications Design Studio within an Inventory project (custom rulesets)
- Are deployed into UIM as part of a cartridge

To understand how custom rulesets work, you must understand the following concepts:

- Extension Points
- Ruleset Extension Points
- Enabled Extension Points

Extension Points

An extension point defines a UIM API method signature to establish a specific point in the code at which to call a ruleset. UIM provides several base extension points in the **ora_uim_baseextpts** cartridge and also provides a framework that enables you to create custom extension points.

Custom extension points are created in Design Studio in the Extension Point editor and in a corresponding custom **aop.xml** file. See "Creating Extension Points" for more information.

An extension point is defined as specification-based or global, based on the Design Studio Extension Point editor **Global** check box. If the **Global** check box is not selected, the extension point is specification-based. If the **Global** check box is selected, the extension point is global.

Specification-based extension points pertain to a particular specification, and global extension points do not.

Specification-Based Extension Points

The signature argument for specification-based extension points must define a specific UIM entity object, such as TelephoneNumber, Equipment, or Pipe, or define a generic object, such as a java.util.Collection that can contain specific UIM entity objects. For example, the TelephoneNumberManager_createTelephoneNumbers base extension point defines the following signature:

```
public abstract interface java.util.List
oracle.communications.inventory.api.number.TelephoneNumberManager.
createTelephoneNumbers(java.lang.String, java.lang.String,
                       oracle.communications.inventory.api.entity.TelephoneNumber)
```

The signature defines TelephoneNumber as the method argument. This indicates that the extension point is intended to be used with the Telephone Number specification.

Global Extension Points

The signature argument for global extension points is not restricted; it may define any type of argument, or no argument at all. For example, the TimeoutEventListener_timerExpired base global extension point defines the following signature, which includes no argument:

```
public void
oracle.communications.inventory.api.common.TimeoutEventListener.timerExpired()
```

Extension Point Types

Extension points also define a type, which dictates how the extension point is weaved into the UIM code stream. There are two extension point types:

- Execution
- Call

To understand extension point types, you must first understand ruleset extension points. See "Ruleset Extension Points".

Ruleset Extension Points

A ruleset extension point configures a ruleset to run at an extension point and configures the placement of the ruleset with respect to the method signature defined by the extension point. UIM provides no ruleset extension points, but it does provide a framework that enables you to create ruleset extension points.

Through a ruleset extension point, you can configure:

- A base ruleset to run at a base extension point
- A base ruleset to run at a custom extension point
- A custom ruleset to run at a base extension point
- A custom ruleset to run at a custom extension point
- Whether to run the rule set:
- Before the method
- After the method
- Instead of the method

Ruleset extension points are created in Design Studio in the Ruleset Extension Point and Ruleset Extension Point - Global editors. See "Creating Ruleset Extension Points" for more information.

Understanding Extension Point Type and Ruleset Placement

An extension point defines a type of execution or call, and a ruleset extension point defines where and when the ruleset is run (before, after, or instead of the method defined by the extension point). Together, this information dictates how the extension point is weaved into the UIM code stream, as explained using the following figures.

Figure 8–1 represents the UIM code stream. Within the UIM code stream, method a() is shown, as well as calls to UIM method a() from various places within the UIM code stream. The dots within method a() represent executable lines of code.

Figure 8–1 UIM Code Stream

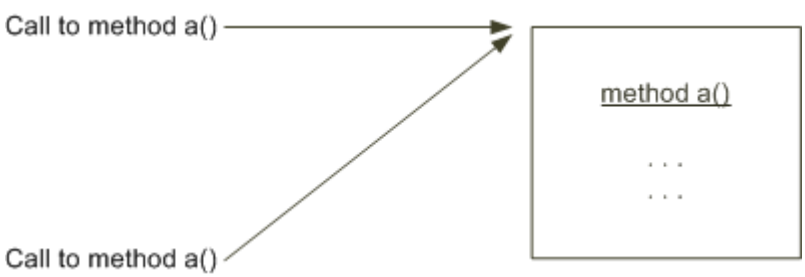


Figure 8–2, Figure 8–3, and Figure 8–4 show an extension point type of execution, which dictates the extension point is weaved within the method defined by the extension point. For this type, the extension point is weaved in only one place: within the method itself.

Figure 8–2 represents a ruleset configured to run before the method. For this type of configuration, the extension point is weaved into the method, immediately prior to the first line of the method's executable code. The result is that the ruleset custom code runs before the method runs.

Figure 8–2 Type Execution, with Placement Before

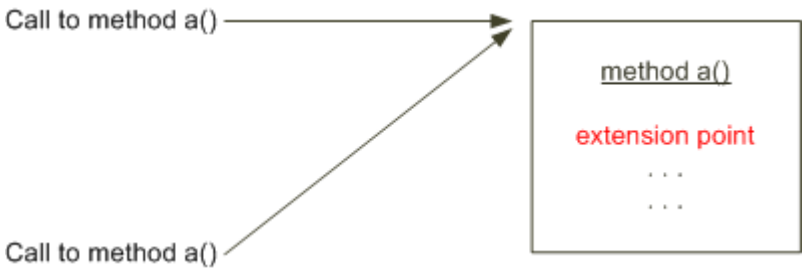


Figure 8–3 represents a ruleset configured to run after the method. For this type of configuration, the extension point is weaved into the method, immediately following the last line of the method's executable code. The result is the ruleset custom code runs after the method runs.

Figure 8–3 Type Execution, with Placement After

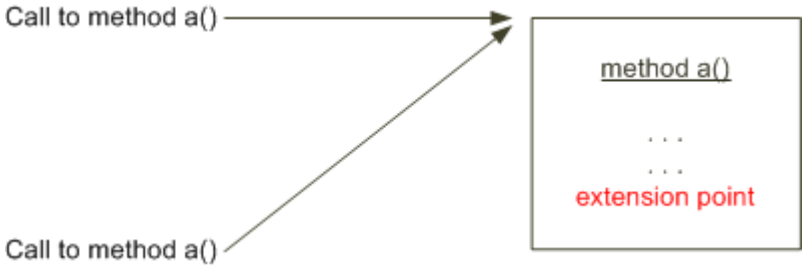


Figure 8–4 represents a ruleset configured to run instead of the method. For this type of configuration, the extension point is weaved into the method, and the method's executable code does not run. The result is the ruleset custom code runs instead of the method.

Figure 8–4 Type Execution, with Placement Instead

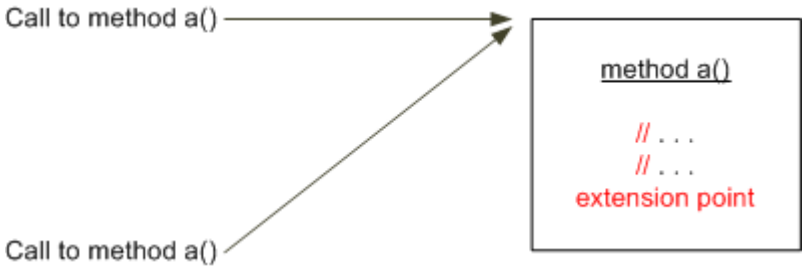


Figure 8–5, Figure 8–6, and Figure 8–7 show an extension point type of call, which dictates the extension point is weaved at the call to the method defined by the extension point. For this type, the extension point may be weaved in multiple places: at each place from where the method is called.

Figure 8–5 represents a ruleset configured to run before the method. For this type of configuration, the extension point is weaved into the UIM code stream, immediately prior to the method call. The result is the ruleset custom code runs before the method runs.

Figure 8–5 Type Call, with Placement Before

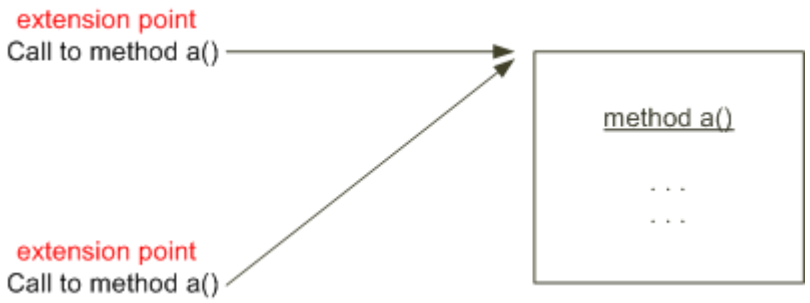


Figure 8–6 represents a ruleset configured to run after the method. For this type of configuration, the extension point is weaved into the UIM code stream, immediately after the method call. The result is the ruleset custom code runs after the method runs.

Figure 8–6 Type Call, with Placement After

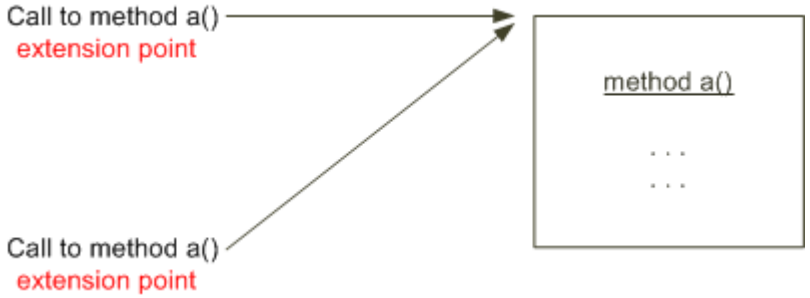
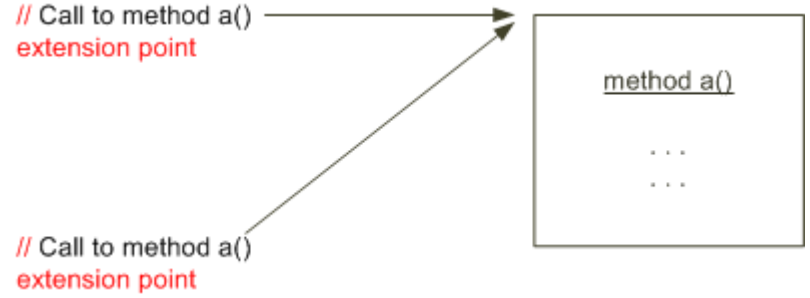


Figure 8–7 represents a ruleset configured to run instead of the method. For this type of configuration, the extension point is weaved into the UIM code stream, and the method is not called. The result is the ruleset custom code runs instead of the method.

Figure 8–7 Type Call, with Placement Instead



Runtime performance is not affected by extension point type; however, server startup performance is affected because that is when custom extension points are weaved, and there is more to weave for type call. For this reason, Oracle recommends that extension points be defined as type execution.

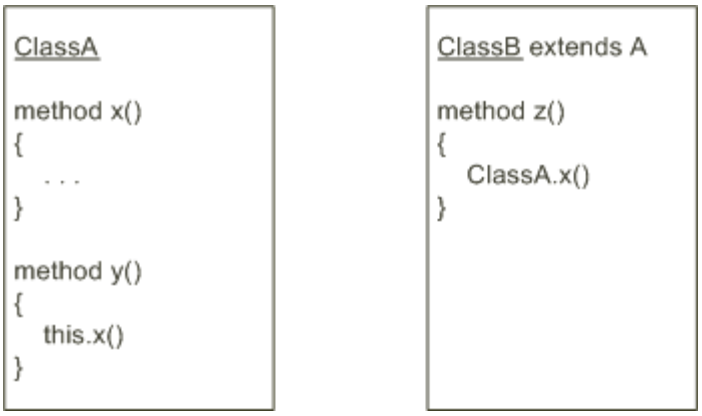
Based on this recommendation, you cannot specify extension point type in the Design Studio Extension Point editor; all extension points default to type execution. However, there are cases when you may need to use type call. For example, if your custom code needs to know the calling class for processing reasons, or needs to know if the call originated from a web service for processing reasons. In such cases, you must define type call, and there is a way to do this: see "Creating Extension Points" for more information.

Note: Base extension points are all defined as type execution. Base extension points are part of the UIM code base, so they are not weaved into the UIM code stream when UIM is started. Only custom extension points are weaved into the UIM code stream when UIM is started.

The benefit of specifying type call is the ruleset can retrieve the caller through the ExtensionPointRuleContext.getCaller() method. The drawback of specifying type call is the ruleset does not run if the method the extension point defines is called by a method defined in the same class or subclass.

For example, Figure 8–8 shows ClassA, which defines methods x() and y(), and y() calls x(). ClassB extends ClassA and defines method z(), and z() also calls x(). The **aop.xml** file defines an extension point for method x() of type call. The ruleset runs when method x() is called from anywhere outside ClassA or ClassB, but the ruleset does not run when method x() is called from y() or z() because x() is called from within the same class or subclass.

Figure 8–8 Type Call Drawback



Enabled Extension Points

Note: Enabled extension points are used only with specification-based extension points; they are not used with global extension points.

An enabled extension point enables a specification-based extension point for a particular specification. UIM provides several base enabled extension points in the **ora_uim_baseextpts** cartridge and also provides a framework that enables you to create enabled extension points.

Enabling a specification-based extension point for a particular specification is accomplished by associating an entity specification Java class to a specification-based extension point. To understand an enabled extension point, you must first understand that, for specification-based extension points, you must configure the specification for a ruleset extension point. This configuration is done on the **Rules** tab of any specification editor, where you select a ruleset extension point from a list. The list is populated for the specification, based on extension points that are enabled for the specification. If no extension points are enabled for the specification, no ruleset extension points are available for selection on the **Rules** tab of the specification.

For example, if 10 extension points are defined, along with 10 ruleset extension points, and no enabled extension points are defined, the Equipment Specification editor **Rules** tab lists no ruleset extension points from which to choose. However, if 10 extension points are defined, along with 3 ruleset extension points, and 3 of these extension points are enabled for the EquipmentSpecification Java class through enabled extension points, the Equipment Specification editor **Rules** tab lists 3 ruleset extension points from which to choose.

Enabled extension points are created in Design Studio in the Enabled Extension Point editor. See "Creating Enabled Extension Points" and "Configuring a Specification for a Ruleset Extension Point" for more information.

About the UIM Extensibility Framework

The extensibility framework supports the functionality that rulesets and extension points provide. The following sections describe various parts of the extensibility framework that are critical to understanding how rulesets work.

RulesExecutor Class

Package: oracle.communications.inventory.extensibility.rules

This class defines the following methods:

- load()
- execute()
- unload()

You use these methods to load, execute, and unload rulesets. The extensibility framework enables you to automatically run rulesets at extension points. However, you can also write custom code that directly runs a ruleset by calling the execute() method on the RulesExecutor class. See the Javadoc for information about this class. For instructions on how to access the Javadoc, see "Javadoc Documentation".

ExtensionPointContext and ExtensionPointRuleContext Class

Package: oracle.communications.extensibility.extension.util

ExtensionPointRuleContext extends ExtensionPointContext.

For any given extension point, ExtensionPointRuleContext is constructed and made available to the ruleset as an argument. The extensibility framework adds the ExtensionPointContext as an argument, following any arguments defined by the extension point signature.

For extension points of type call, the context contains the calling class. This is provided so custom code can process differently based on the caller. For example, the custom code may need to perform a different process if called from the UI, versus being called from a web service. In this scenario, the custom code can use the context's getCaller() method to make the determination. For extension points of type execution, the context does not contain the calling class. So, the getCaller() method should not be used for extension points of type execution because the return is always null.

Regardless of type, the context contains the target class and method arguments. Method arguments are placed into the argument collection in left-to-right parameter order. Integral types are placed in the corresponding wrapper object. For example, int arguments are passed by reference using an Integer.

ExtensionPointRuleContext.returnValue

Data is returned to a ruleset in the **returnValue** attribute defined in the ExtensionPointRuleContext class. For example, you use the ExtensionPointRuleContext.setReturnValue(Object) method to set the **returnValue** attribute. The placement of the ruleset affects the use of the **returnValue** attribute as follows:

- Before

  If the ruleset populates the **returnValue** attribute, the intercepted method removes any **returnValue** set by the ruleset.

- After

  Data in ExtensionPointRuleContext is available to the ruleset to manipulate. The ruleset can change the **returnValue** attribute either by setting a new return object in the context or by changing attribute values of the return object already in the context. For this scenario, the return value type must match the value type that is normally returned by the intercepted method or an exception is thrown.

- Instead

  The ruleset completely controls what is returned to the caller by setting the **returnValue** attribute. For this scenario, the return value type must match what is normally returned by the intercepted method or an exception is thrown.

For an example of the use of ExtensionPointRuleContext, view the TELEPHONE_NUMBER_FORMATTING ruleset that is provided in the **ora_uim_baserulesets** cartridge.

aop.xml File

The *UIM_Home***/config/extensibility/META-INF/aop.xml** file is provided as an example to follow when creating custom extension points, which is a two-part process: Creating the extension point in the Design Studio Extension Point editor, and creating a custom **aop.xml** file. Both are deployed into UIM as part of a cartridge.

The custom **aop.xml** file must reside in the cartridge's **model/aspects** directory. When UIM is started, the **aop.xml** file is used to weave custom extension points into the UIM code stream.

Example 8–5 is an excerpt from the provided **aop.xml** file, and shows all of the XML elements that the file defines, as well as several of the example extension points that the file defines. (Many of the extension point definitions were removed for readability.)

Example 8–5 aop.xml File

```
<aspectj>
```

```
<!--
  <aspects>
    <concrete-aspect name=
"oracle.communications.inventory.extensibility.extension.SpecAsTarget"
    extends=
"oracle.communications.inventory.extensibility.extension.SpecTargetExtension">
      <pointcut name="ruleExtensionPoint" expression="execution(public *
oracle.communications.inventory.api.impl.entity.SpecificationDAO.getName(..) )"/>
    </concrete-aspect>
    <concrete-aspect name=
"oracle.communications.inventory.extensibility.extension.SpecBasedAsArgument"
    extends=
"oracle.communications.inventory.extensibility.extension.SpecBasedArgumentExtensio
n" >
      <pointcut name="ruleExtensionPoint" expression="
        call(public *
oracle.communications.inventory.api.number.TelephoneNumberManager.createTelephoneN
umbers(String, String, oracle.communications.inventory.api.entity.TelephoneNumber,
java.util.Set, java.util.List))
        || call(public *
oracle.communications.inventory.api.number.TelephoneNumberManager.deleteTelephoneN
umbers(oracle.communications.inventory.api.entity.TelephoneNumber...))
        || call(public *
oracle.communications.inventory.api.number.TelephoneNumberManager.updateTelephoneN
umbers(java.util.List, java.util.Set, java.util.List))"/>
    </concrete-aspect>

    <concrete-aspect name=
"oracle.communications.inventory.extensibility.extension.SpecAsArgument"
    extends=
"oracle.communications.inventory.extensibility.extension.SpecArgumentExtension">
      <pointcut name="ruleExtensionPoint" expression="
        call(public *
oracle.communications.inventory.api.consumer.ReservationManager.extendReservation(
oracle.communications.inventory.api.entity.ServiceSpecification, java.util.List,
java.lang.String, oracle.communications.inventory.api.entity.ReservedForType))
        || call(public *
oracle.communications.inventory.api.consumer.ReservationManager.reserveResource(or
acle.communications.inventory.api.entity.ServiceSpecification,
java.util.Collection,
oracle.communications.inventory.api.entity.common.Reservation))
        || call(public *
oracle.communications.inventory.api.service.ServiceManager.createService(oracle.co
mmunications.inventory.api.entity.Service,
oracle.communications.inventory.api.entity.ServiceSpecification))"/>
    </concrete-aspect>

    <concrete-aspect name=
"oracle.communications.inventory.extensibility.extension.GlobalRule"
    extends=
"oracle.communications.inventory.extensibility.extension.GlobalRuleExtension" >
      <pointcut name="ruleExtensionPoint" expression="
          call(public *
oracle.communications.inventory.api.common.TimeoutEventListener.timerExpired())
        || call(public *
oracle.communications.inventory.api.admin.SecurityManager.handleSecurityViolation(
..))
        || call(public *
oracle.communications.inventory.api.common.AttachmentManager.createAttachment(orac
le.communications.inventory.api.entity.common.Attachment...))
```

```
        || call(public *
oracle.communications.inventory.api.common.AttachmentManager.updateAttachment(orac
le.communications.inventory.api.entity.common.Attachment...))
        || call(public *
oracle.communications.inventory.api.common.AttachmentManager.deleteAttachment(orac
le.communications.inventory.api.entity.common.Attachment...))"/>
    </concrete-aspect>

    <concrete-aspect
name="oracle.communications.inventory.extensibility.extension.SecurityValidation"

extends="oracle.communications.inventory.extensibility.extension.SecurityValidatio
nExtension" >
      <pointcut name="securityExtensionPoint" expression="
          call(public *
oracle.communications.inventory.api.group.InventoryGroupManager.createInventoryGro
up(oracle.communications.inventory.api.entity.InventoryGroup))
        || call(public *
oracle.communications.inventory.api.group.InventoryGroupManager.deleteInventoryGro
up(oracle.communications.inventory.api.entity.InventoryGroup))
        || call(public *
oracle.communications.inventory.api.group.InventoryGroupManager.updateInventoryGro
up(oracle.communications.inventory.api.entity.InventoryGroup))"/>
    </concrete-aspect>
  </aspects>

  <weaver>
    <include within=
"oracle.communications.inventory.api.number.impl.TelephoneNumberManagerImpl"/>
    <include within=
"oracle.communications.inventory.api..*"/>
  </weaver>

-->
</aspectj>
```

The provided **aop.xml** file defines the following:

- aspects

  This element defines the concrete extensions through the <concrete-aspect> element. The implemented aspects are:

  - SpecAsTarget

    Defines extension points for method signatures defined on a specification object. An example of a specification object is a specification itself, such as EquipmentSpecification. For example, when the EquipmentSpecification.setModelNumber(String modelNbr) method is called, EquipmentSpecification is the target of the invocation.

    Oracle recommends that you not use this type of aspect.

  - SpecBasedAsArgument

    Defines extension points for method signatures that define specification-based arguments. An example of a specification-based argument is an instance of an entity, such as Equipment. For example, the Equipment.createPhysicalPorts(Equipment equip, List physPorts) method defines an argument of Equipment.

  - SpecAsArgument

Defines extension points for method signatures that define specification arguments. An example of a specification argument is a specification itself, such as TelephoneNumberSpecification. For example, the SpecManager.getEditMask(TelephoneNumberSpecification) method defines an argument of TelephoneNumberSpecification.

- GlobalRule

  Defines global extension points for method signatures that define arguments that are neither a specification nor specification-based. For example, ReservationManager.expireReservation(boolean) is a method defined as a global extension point.

- SecurityValidation

  Defines extension points for APIs that require authorization to access. Extension points defined in this element are neither specification-based nor global because they are not part of the extensibility framework. Rather, they are part of the Security Validation Extension framework, and the execution of the APIs defined for these extensions go through UIM logic authorization.

- weaver

  Defines the Java packages the extensibility framework is to search for classes in which to weave any custom extension points. The weaving of custom extension point is done when UIM is started.

- There are five aspects defined, and each aspect defines several extension points.
- Extension points are defined, including type of call or execution.
- The argument for the deleteTelephoneNumbers() method uses a notation of "..." to represent an array of TelephoneNumber objects. This notation is used in the **aop.xml** method signature, but not in the Design Studio method signature. See "Creating Extension Points" for more information.
- All of the method signatures in the SpecBasedAsArgument aspect define at least one argument that is specification-based (TelephoneNumber, an Array of TelephoneNumber objects, and a java.util.Collection of TelephoneNumber objects).
- All of the method signatures in the SpecAsArgument aspect define at least one argument that is a specification (ServiceSpecification).
- The method signatures in the GlobalRule aspect may define specification-based arguments, specification arguments, generic arguments, or no arguments at all. There are no restrictions on these arguments. So, regardless of the arguments defined, a ruleset configured to run at a global extension point always runs because it is defined within the <GlobalRule> element.
- The method signatures defined in the SecurityValidation aspect control access to the API methods of createInventoryGroup, deleteInventoryGroup, and updateInventoryGroup.
- The <weaver> element tells the extensibility framework what packages to search when looking to weave any custom extension points.

Turning the Weaver On

When UIM is started, the custom **aop.xml** file is used to weave custom extension points into the UIM code stream. To initiate this weaving of custom extension points, the Weaver must be turned on. *UIM_Home/Domain_Home***/bin/startUIM.cmd** is the script that starts UIM, and it calls *UIM_Home/Domain_Home***/bin/setUIMEnv.cmd**. The **setUIMEnv.cmd** script sets the UIM environment, and includes the following lines. To

turn the Weaver on or off, uncomment or comment the following lines in the **setUIMEnv.cmd** file.

```
set JAVA_OPTIONS=%JAVA_OPTIONS% -javaagent:%UIM_HOME%\lib\aspectjweaver.jar
set JAVA_OPTIONS=%JAVA_OPTIONS% -Daj.weaving.verbose=false
```

About Base Rulesets

This section provides information on base rulesets, and on base extension points and base enabled extension points that can be used call a base ruleset or custom ruleset.

> **Note:** See Appendix C, "Base Rulesets" for detailed information about each base ruleset, including a description of what the base ruleset does and step-by-step instructions for running it.

UIM provides several base rulesets in the **ora_uim_baserulesets** cartridge. Base rulesets are called by UIM code and provide examples for creating custom rulesets. Each base ruleset provides a Drools version and a Groovy version of the ruleset. You can find the base rulesets in the *UIM_Home***/cartridges/base/ora_uim_baserulesets.jar** file.

Base rulesets can be viewed in Design Studio or in UIM:

- To view base rulesets in Design Studio, import the **ora_uim_baserulesets** cartridge into Design Studio. After you import the base cartridge, you can view the base rulesets in the Design perspective Studio Projects view.

  For instructions on how to import a cartridge into Design Studio, see the Design Studio Help.

- To view base rulesets in UIM, deploy the **ora_uim_baserulesets** cartridge into UIM. After you deploy the base cartridge, you can view the rulesets in UIM by clicking the **Rulesets** link in the Tasks panel of the UIM Home page.

  See *UIM Cartridge Guide* for information about deploying cartridges and cartridge packs.

About Base Extension Points and Base Enabled Extension Points

UIM provides numerous base extension points and base enabled extension points in the **ora_uim_baseextpts** cartridge. You can use the base extension points to call base rulesets or custom rulesets. You can find the base extension points and base enabled extension points in the *UIM_Home***/cartridges/base/ora_uim_baseextpts.jar** file.

Base extension points and base enabled extension points can be viewed in Design Studio. To do so, import the **ora_uim_baseextpts** cartridge into Design Studio. After you import the base cartridge, you can view the base extension points and base enabled extension points in the Design perspective Studio Projects view.

For instructions on how to import a cartridge into Design Studio, see the Design Studio Help.

See *UIM Cartridge and Technical Pack Guide* for more information on the **ora_uim_base_extpts** cartridge.

About Naming Conventions

Understanding the naming convention used for the base extension points and base enabled extension points helps you readily locate them within the **ora_uim_baseextpts**

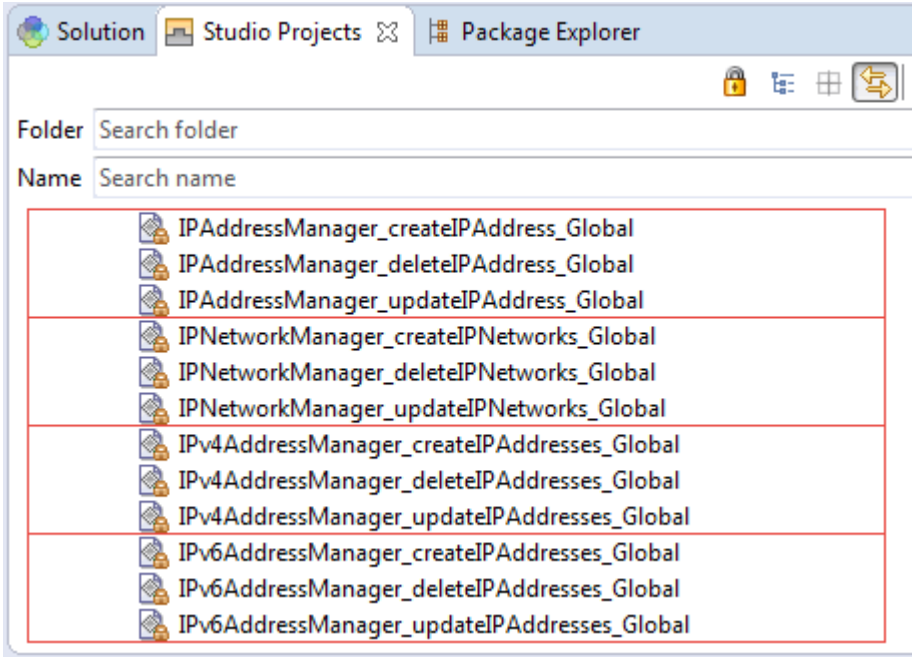
cartridge, which is important because there are hundreds of them. Within the base cartridge, the base extension points are grouped together alphabetically, and the base enabled extension points are grouped together alphabetically.

The naming convention for base extension points is *ClassName_methodName*. For example, the TelephoneNumberManager_createTelephoneNumbers.

The naming convention for base enabled extension points is *SpecificationName_ClassName_methodName*. For example, the TelephoneNumberSpecification_TelephoneNumberManager_createTelephoneNumbers.

There are often multiple base extension points defined per class. For example, Figure 8–9 is an excerpt of the **ora_uim_baseextpts** cartridge expanded in Design Studio that shows several base extension points defined for several entity manager classes. Specifically, IPAddressManager, IPNetworkManager, IPv4AddressManager, and IPv6AddressManager each define extension points for their respective create(), delete(), and update() methods.

Figure 8–9 IP Address Entity Manager Classes



Note: The naming convention used for these extension points indicates that these are global extension points. However, this naming convention is not always employed. So, when working with base extension points, be sure to look at the extension point definition to determine whether the **Global** check box is selected, indicating that the extension point is a global extension point.

Working with Rulesets

Before reading this section:

- Read the preceding sections of this chapter and have an understanding of rules, rulesets, extension points, and ruleset extension points.

- You should understand what a cartridge is, how to create one in Design Studio, and how to deploy one into UIM.
- You should understand Design Studio perspectives and views, and how to switch between them.

When working with rulesets to extend UIM:

1. Determine the functionality that you plan to extend and how you plan to extend it.

   See *UIM Concepts* to learn about existing UIM functionality. See the UIM Javadoc to learn about specific classes or methods you plan to extend. For information on accessing the Javadoc, see "Javadoc Documentation".

2. Configure Design Studio. When working with rulesets, it is important that you configure your environment correctly to avoid errors later in the process.

   See "Configuring Design Studio".

3. In Design Studio, install, configure, and learn how to use one of the following:
   - Drools Eclipse plug-ins

     See "Installing, Configuring, and Using the Drools Eclipse Plug-ins".
   - Groovy Eclipse plug-ins

     See "Installing, Configuring, and Using the Groovy Eclipse Plug-ins".

4. Create an Inventory project.

   See the Design Studio Help.

5. Create a ruleset.

   See "Creating Rulesets".

6. Create an extension point, or use an existing base extension point.

   See "Creating Extension Points".

7. Create a ruleset extension point to configure the ruleset to run at the extension point.

   See "Creating Ruleset Extension Points".

8. For specification-based extension points, create an enabled extension point to enable the extension point for the specification.

   See "Creating Enabled Extension Points".

9. For specification-based extension points, configure the specification for the ruleset extension point.

   See "Configuring a Specification for a Ruleset Extension Point".

10. Validate and compile the ruleset, and build the project to create the cartridge.

    See "Validating and Compiling Rulesets" and the Design Studio Help.

    If your ruleset custom code is dependent upon third-party code for successful compilation, see "Compiling Rulesets with Third-Party Dependencies".

11. Deploy the cartridge into UIM.

    See "Deploying Cartridges Containing Rulesets".

12. If you created an extension point (as opposed to using a base extension point), make sure the Weaver is turned on and restart UIM.

See "Turning the Weaver On".

13. Run the ruleset.

    See "Running Rulesets".

14. If problems are encountered, debug and troubleshoot.

    See "Debugging Custom Groovy Rulesets".

Installing, Configuring, and Using the Drools Eclipse Plug-ins

Installing, configuring, and using the Drools Eclipse plug-ins are described in the following sections:

- Installing the Drools Eclipse Plug-ins
- Configuring the Drools Eclipse Plug-ins
- Configuring the Project Builders
- Using the Drools Eclipse Plug-ins

Installing the Drools Eclipse Plug-ins

To install the Drools Eclipse plug-ins:

1. From the Design Studio **Help** menu, select **Install New Software**.

   The Install window appears.

2. Click **Add**.

   The Add Repository window appears.

3. In the **Name** field, enter an arbitrary name, such as “Drools.”

4. Copy the following URL and paste it into the **Location** field:

   ```
   http://downloads.jboss.org/drools/release/6.5.0.Final/org.drools.updatesite/
   ```

5. Click **OK**.

   The Add Repository window closes.

6. From the **Work with** list, select the name of the repository you just added.

7. Select the **Group items by category** check box, if it is not already selected.

8. Expand **Drools and jBPM**.

9. Select **Drools and jBPM**, which automatically selects following plug-ins to install:

   - JBoss Drools Core
   - JBoss Drools Guvnor
   - JBoss iBPM Core

10. Click **Next** twice, accept the license agreement, and click **Finish**.

    A security warning response window appears.

11. Click **OK**.

    A prompt to restart Eclipse appears.

12. Click **Yes**.

    You must restart Eclipse for the installed plug-ins to work.

Configuring the Drools Eclipse Plug-ins

After you install the Drools Eclipse plug-ins in Design Studio, you must configure Design Studio to recognize them. Configuring the Drools Eclipse plug-ins is described in the following sections:

- Configuring the Drools Runtime Preference
- Configuring the File Associations Preference
- Configuring the Cartridge for Drools Files

Configuring the Drools Runtime Preference

To configure the Drools runtime preference:

1. Within your *Eclipse_Home* directory, create a new folder and name it **myDroolsRuntime**.
2. In Design Studio, from the **Window** menu, select **Preferences**.

   The Preferences window appears.
3. In the navigation panel, expand **Drools** and select **Installed Drools Runtimes**.
4. Click **Add**.

   The Drools Runtime window appears.
5. Click **Create a new Drools Runtime**.

   The Browse For Folder window appears.
6. Navigate to the *Eclipse_Home*/**myDroolsRuntime** folder and click **OK**.

   The Browse For Folder window closes.

   The **Name** and **Path** fields are now populated on the Drools Runtime window.
7. Click **OK**.

   The Drools Runtime window closes.
8. Select the check box located to the left of the Drools Runtime you just added, and click **OK**.

   The Preferences window closes.

Configuring the File Associations Preference

To configure the File Associations preference:

1. In Design Studio, from the **Window** menu, select **Preferences**.

   The Preferences window appears.
2. In the navigation panel, expand **General**, then **Editors**, and then select **File Associations**.
3. In the File types section, select **.drl**.
4. In the Associated editors section, select **Rule Editor** and click **Default**.
5. Click **OK**.

   The Preferences window closes.

Configuring the Cartridge for Drools Files

To configure the **.project** and **.classpath** files, perform the following steps for each cartridge in which you plan to create rulesets:

1. In Design Studio, open the Java perspective, and open the Navigator view.
2. Expand the cartridge in which you will create rulesets with Drools files.
3. Open the cartridge's **.project** file by double-clicking the file name.
4. In the **.project** file editor, click the **Source** tab located at the bottom of the editor.
5. Within the XML, create a new <buildCommand> element by copying and pasting one of the existing <buildCommand> elements.
6. Change the copied <buildCommand> element to the following value:

```
<buildCommand>
    <name>org.drools.eclipse.droolsbuilder</name>
    <arguments>
    </arguments>
</buildCommand>
```

7. Save the **.project** file.
8. In the Navigator view, open the **.classpath** file by double-clicking the file name.
9. In the **.classpath** file editor, click the **Source** tab located at the bottom of the file editor.
10. Within the XML, create a new <classpathentry> element by copying and pasting one of the existing <classpathentry> elements.
11. Change the copied <classpathentry> element to the following value:

```
<classpathentry kind="con" path="DROOLS/Drools"/>
```

> **Note:** When specifying the path, the case matters: it must be as shown above.

12. Save the **.classpath** file.

Configuring the Project Builders

To configure the project builders:

1. In Design Studio, open the Java perspective, and open the Navigator or Package Explorer view.
2. Select the cartridge.
3. From the **Project** menu, select **Properties**.
4. In the navigation panel, select **Builders**.
5. Select **Drools Builder** and click **Down** as needed to place **Drools Builder** at the end of the list.

Using the Drools Eclipse Plug-ins

The Drools Eclipse plug-ins provide a Drools rule editor and several Drools-specific menu options.

Drools Rule Editor

After you install and configure the plug-ins, the Drools rule editor works in a Drools file opened within Design Studio. The editor catches syntax errors when you are writing a ruleset. The rule editor also provides the ability to compile a ruleset prior to deploying the cartridge containing the ruleset.

Drools-Specific Menu Options

To access the Drools-specific menu options:

1. In Design Studio, from **File** menu, select **New**, then select **Other**.
2. Expand **Drools**.

   The Drools-specific menu options are:

   - Decision Table
   - Domain Specific Language
   - Drools Project
   - Flow File
   - Knowledge Base
   - Rule Resource

For information on using the Drools-specific menu options, see the *Drools Documentation* website:

`http://docs.jboss.org/drools/release/6.5.0.Final/drools-docs/html_single/`

Installing, Configuring, and Using the Groovy Eclipse Plug-ins

Installing, configuring, and using the Groovy Eclipse plug-ins is described in following sections:

- Installing the Groovy Eclipse Plug-ins
- Configuring the Groovy Eclipse Plug-ins
- Using the Groovy Eclipse Plug-ins

For more information on installing Groovy plug-ins, see the *Groovy Eclipse Wiki* website:

`https://github.com/groovy/groovy-eclipse/wiki`

Installing the Groovy Eclipse Plug-ins

To install the Groovy Eclipse plug-ins:

1. From the Design Studio **Help** menu, select **Install New Software**.

   The Install window appears.
2. Click **Add**.

   The Add Repository window appears.
3. In the **Name** field, enter an arbitrary name, such as “Groovy.”
4. Determine the appropriate update site URL from the list provided in the *Groovy Eclipse Wiki* website.

   See "Software Requirements" for information on the required Groovy version.

5. Paste the update site URL into the **Location** field.
6. Click **OK**.

   The Add Repository window closes.
7. From the **Work with** list, select the name of the repository you just added.
8. Select the **Group items by category** check box, if it is not already selected.
9. Expand and select the Groovy-Eclipse plug-in feature, and follow the prompts to install.
10. Accept the license agreement, and click **Finish** and **OK**.
11. Click **Yes** to agree to restart Eclipse.

    You must restart Eclipse for the installed plug-ins to work.

Configuring the Groovy Eclipse Plug-ins

After you install the Groovy Eclipse plug-ins in Design Studio, you must configure Design Studio to recognize them. Configuring the Groovy Eclipse plug-ins is described in the following sections:

- Configuring the Groovy Compiler Version
- Configuring File Associations for Groovy
- Configuring the Cartridge for Groovy Files

Configuring the Groovy Compiler Version

To configure the Groovy compiler version:

1. In Design Studio, from the **Window** menu, select **Preferences**.

   The Preferences window appears.
2. In the navigation panel, select **Groovy** to expand the list and select **Compiler**. By default the highest compiler version is selected.
3. Click the appropriate Groovy version for UIM.
4. Click **Yes** to confirm the change.
5. Click **OK**.

   The Preferences window closes.
6. Restart Eclipse for the compiler setting to take effect.
7. Verify that the Groovy compiler version is set correctly:
   a. From the Design Studio **Window** menu, select **Preferences**.
   b. In the navigation panel, select **Groovy** to expand the list and then select **Compiler**.
   c. Verify that the compiler version is set to the appropriate version.

See "Software Requirements" for the Groovy version information.

Configuring File Associations for Groovy

To configure File Associations for Groovy:

1. In Design Studio, from the **Window** menu, select **Preferences**.

   The Preferences window appears.

2. In the navigation panel, expand **General**, then **Editors**, and then select **File Associations**.
3. In the File types section, select **.groovy**.
4. In the Associated editors section, select **Groovy Editor** and click **Default**.
5. Click **OK**.

   The Preferences window closes.

Configuring the Cartridge for Groovy Files

To configure the **.project** and**.classpath** files, perform the following steps for each cartridge in which you plan to create rulesets:

1. In Design Studio, open the Java perspective, and open the Navigator view.
2. Expand the cartridge in which you will create rulesets.
3. Open the cartridge's **.project** file by double-clicking the file name.
4. In the **.project** file editor, click the **Source** tab located at the bottom of the editor.
5. Within the XML, create a new <nature> element by copying and pasting one of the existing <nature> elements.
6. Change the copied <nature> element to the following value:

   ```
   <nature>org.eclipse.jdt.groovy.core.groovyNature</nature>
   ```

7. Save the **.project** file.
8. In the Navigator view, open the cartridge's **.classpath** file by double-clicking the file name.
9. In the **.classpath** file editor, click the **Source** tab located at the bottom of the editor.
10. Within the XML, create a new <classpathentry> element by copying and pasting one of the existing <classpathentry> elements.
11. Change the copied <classpathentry> element to the following value:

    ```
    <classpathentry exported="true" kind="con" path="GROOVY_SUPPORT"/>
    ```

    Note: The path value is case-sensitive and must be entered as shown above.

12. Save the **.classpath** file.

Using the Groovy Eclipse Plug-ins

The Groovy Eclipse plug-ins provide a Groovy editor and several Groovy-specific menu options.

Groovy Editor

After you install and configure the plug-ins, the Groovy editor works in a Groovy file opened within Design Studio. The editor catches syntax errors when you are writing ruleset code. The Groovy editor also enables you to compile a ruleset prior to deploying the cartridge containing the ruleset.

Groovy-Specific Menu Options

To access the Groovy-specific menu options:

1. In Design Studio, from the **File** menu, select **New**, and then **Other**.
2. Select **Groovy**.

   The Groovy-specific menu options are:

   - Groovy Class
   - Groovy DSL Descriptor
   - Groovy Project
   - Groovy Test Case

For information about using Groovy, see the Groovy documentation at the following website:

http://www.groovy-lang.org/documentation.html

Creating Rulesets

For instructions on how to create a ruleset in Design Studio, see the Design Studio Help. When creating a ruleset, use the following information.

Name Field

When entering the name of the ruleset in the **Name** field, the text editor forces you to enter all capitals. Oracle recommends that you use underscores for readability, such as MY_RULE_SET.

DRL File or Groovy File

A ruleset resides in a **.drl** file or in a **.groovy** file, both of which you access from the Ruleset editor. These files are saved in the inventory project's **model** directory.

When writing a custom ruleset, Oracle recommends that you:

- Read the preceding conceptual information in this chapter.

  See "About Rules" and "About Rulesets and Extension Points".
- Install one of the following in Design Studio:
  - Drools Eclipse plug-ins

    See "Installing, Configuring, and Using the Drools Eclipse Plug-ins".
  - Groovy Eclipse plug-in

    See "Installing, Configuring, and Using the Groovy Eclipse Plug-ins".
- Determine the base extension point or custom extension point that is to run your ruleset. The extension point dictates the ruleset input parameters to code, which, in turn, dictates the data made available to the ruleset.

  See "Creating Extension Points".
- Review some of the base rulesets, which provide examples that can help you gain a better understanding of rulesets.

  See "About Base Rulesets".
- Start with a base ruleset **.drl** or **.groovy** file and modify it as needed. To do this, copy any base ruleset **.drl** or **.groovy** file to your custom ruleset **.drl** or **.groovy** file.

  See "About Base Rulesets".
- Review examples of custom code that call various UIM API methods.

See *UIM API Overview*.

- Reference the Drools documentation as needed.

  See the *Drools Documentation* website:

  http://docs.jboss.org/drools/release/6.5.0.Final/drools-docs/html_single/

Creating Extension Points

This section applies to specification-based and global extension points.

> **Note:** Before creating a custom extension point, check the *UIM_Home*/**cartridges/base/ora_uim_baseextpts** cartridge to see if a base extension point already exists that defines the UIM API method you need to use.

Before you create an extension point, you must first determine the UIM API method signature that you want the extension point to define. For example, you may want to create an extension point that deals with disassociating a telephone number from an inventory group. To determine the UIM API method signatures you need to use the Javadoc and search for the manager class using ***Manager.class** as search criterion to return a list of all manager classes such as TelephoneNumberManager, EquipmentManager, and PipeManager. After locating the appropriate manager class, search the list of methods for the most likely method, such as the TelephoneNumberManager.disassociateTN() method.

For instructions on how to access the Javadoc, see "Javadoc Documentation".

Creating an extension point is a two-part process:

- Creating the Extension Point in Design Studio
- Creating the aop.xml File

Both are deployed into UIM as part of a cartridge.

Creating the Extension Point in Design Studio

For instructions on how to create an extension point in Design Studio, see the Design Studio Help. When creating an extension point, use the following information.

Name Field

When entering a name, Oracle recommends that you follow the same naming convention used in the **ora_uim_baseextpts** cartridge, which is *ClassName_methodName*. For example TelephoneNumberManager_disassociateTN.

Point Name Field

When entering a point name, Oracle recommends that you follow the same naming convention used in the **ora_uim_baseextpts** cartridge, which is *ClassName.methodName*. For example, TelephoneNumberManager.disassociateTN.

Signature Field

Correctly entering the method signature in the **Signature** field is critical for the ruleset to run. An exact signature match is required. For example, spacing errors result in the ruleset not executing, as well as using a class interface hierarchy supertype. Oracle

recommends that you copy the signature from the *UIM_Home*/**lib/uim-core-interfaces.txt** file and paste it into the **Signature** field. The **uim-core-interfaces.txt** file provides a generated listing of all API method signatures. Copy the text from **public abstract interface** through the end of the line.

Example 8–6 shows a typical signature definition:

Example 8–6 Signature

```
public abstract interface java.lang.String
oracle.communications.inventory.api.businessinteraction.BusinessInteractionManager.getEntityAction(
oracle.communications.inventory.api.entity.common.RootEntity,
oracle.communications.inventory.api.entity.BusinessInteraction, java.lang.String)
```

A signature requires the following:

- Visibility modifier (public, private, protected)

  The visibility modifier must be defined as **public abstract interface**, as shown in Example 8–6. The existence of the Javadoc for the method indicates that the method is a public interface. AspectJ requires that all methods defined for an extension point be declared as abstract, even if the method in the Java code is not defined as abstract.

- Return

  This part of the signature defines the return values. In Example 8–6, getEntityAction() returns java.lang.String.

- Fully qualified method call, which includes:
  - Package

    In Example 8–6, the package that contains the BusinessInteractionManager class is oracle.communications.inventory.api.businessinteraction.

  - Class

    In Example 8–6, the class is BusinessInteractionManager.

  - Method

    In Example 8–6, the method is getEntityAction().

  - Arguments

    In Example 8–6, the arguments are the fully qualified objects of RootEntity, BusinessInteraction, and String.

Putting all parts together results in the signature being defined as shown in Example 8–6.

When entering the signature:

- Characters that define the signature are case sensitive.
- No extra spaces can exist within the signature.
- If the signature defines multiple arguments, the arguments are separated by a comma followed by a space.
- Signatures that define arrays must use [] to represent an array of objects.
- Signatures that define an array of objects as a parameter must contain the **transient** keyword. The AspectJ framework requires this keyword to retrieve the extension point, as shown in Example 8–7.

Example 8–7 Signature with Transient and Array

```
public abstract transient interface oracle.communications.inventory.api.entity.BusinessInteraction
oracle.communications.inventory.api.businessinteraction.BusinessInteractionManager.transferItems(or
acle.communications.inventory.api.entity.BusinessInteraction,
oracle.communications.inventory.api.entity.BusinessInteraction,
oracle.communications.inventory.api.entity.BusinessInteractionItem[])
```

> **Caution:** Do not copy the signature from the Javadoc and paste it into the **Signature** field: Copying from an HTML file results in spacing errors. Avoiding these spacing errors is critical because the ruleset does not run and you do not get an error, so it is difficult to determine the problem.

Creating the aop.xml File

To create an **aop.xml** file:

1. In Design Studio, create an Inventory project.
2. Switch to the Java perspective.
3. Expand the inventory project, and expand the **model** directory. (The **model** directory gets created when the inventory project is created.)
4. Create a new directory named **aspects** within the **model** directory.
5. Create a new file named **aop.xml** in the **model/aspects** directory.
6. Model the contents of the custom **aop.xml** file after the *UIM_Home*/**config/extensibility/ META-INF/aop.xml** file. See "aop.xml File" for more information on the **aop.xml** file.
7. Determine the aspect of the extension point, which is based on the method signature defined by the extension point. See "aop.xml File" for more information on aspects.
8. Define the extension point within the determined aspect. See "aop.xml File" for more information.
9. Delete any <concrete-aspect> elements that do not define any extension points.
10. Ensure that the <weaver> element is not commented out.
11. Update the <weaver> include elements to reflect the correct package or packages that are applicable to your custom extension point. See "aop.xml File" for more information.
12. Save the **aop.xml** file.
13. Open the *UIM_Home/Domain_Home*/**bin/startUIM.cmd** file.
14. Verify that the following lines are uncommented:

    ```
    set JAVA_OPTIONS=%JAVA_OPTIONS% -javaagent:%UIM_HOME%\lib\aspectjweaver.jar
    set JAVA_OPTIONS=%JAVA_OPTIONS% -Daj.weaving.verbose=false
    ```

    See "Turning the Weaver On" for more information.
15. Save the **startUIM.cmd** file.

Caution: The concrete-aspect name must be unique across all extension points installed on your application server or the following error appears on server startup: "Error Attempt to concretize but chosen aspect name already defined: *name* in **aop.xml** warning register definition failed."

To fix this error, change the concrete-aspect name. Do not change the extends portion of the concrete-aspect. Example 8–8 shows the before and after reflecting this change.

Example 8–8 Concrete-Aspect Name

```
// Before
<concrete-aspect name="oracle.communications.extensibility.extension.SpecBasedAsArgument"
        extends="oracle.communications.extensibility.extension.SpecBasedArgumentExtension">
<pointcut name="ruleExtensionPoint" expression="
call (public abstract interface java.util.List
oracle.communications.inventory.api.connectivity.PipeManager.updatePipes(java.util.Collection))"/>

// After
<concrete-aspect name="oracle.communications.extensibility.extension.SpecBasedAsArgumentXyz"
        extends="oracle.communications.extensibility.extension.SpecBasedArgumentExtension">
<pointcut name="ruleExtensionPoint" expression="
call (public abstract interface java.util.List
oracle.communications.inventory.api.connectivity.PipeManager.updatePipes(java.util.Collection))"/>
```

Creating Ruleset Extension Points

Note: This section applies to ruleset extension points and global ruleset extension points.

Note: No base ruleset extension points are provided. If you want to use a base ruleset and a base extension point, you must create a ruleset extension point to configure the base ruleset to run at the base extension point.

The ruleset extension point configures a ruleset to run at a specification-based extension point and configures the placement of the ruleset.

For instructions on how to define a ruleset extension point, see the Design Studio Help.

Creating Enabled Extension Points

Note: This section applies only to specification-based extension points.

Note: Before creating a custom enabled extension point, check the **ora_uim_baseextpts** cartridge to see if a base enabled extension point already exists for the specification and extension point that you need to enable.

Create an enabled extension point for every specification-based extension point you create. For instructions on how to create an enabled extension point, see the Design Studio Help. When creating an enabled extension point, use the following information.

Name Field

When entering a name, Oracle recommends that you follow the same naming convention used in the **ora_uim_baseextpts** cartridge, which is *SpecificationName_ClassName_methodName*. For example, usTelephoneNumber_TelephoneNumberManager_disassociateTN.

Specification Class Name Field

You must select a value from the **Class Specification Name** list, which is preloaded with the fully qualified entity specification Java class names.

Nearly all of the Studio Inventory entities are recognizable by their Java class name. For example, the Telephone Number Specification entity appears in the list as oracle.communications.inventory.api.entity.TelephoneNumberSpecification, and the Equipment Specification entity appears in the list as oracle.communcations.inventory.api.entity.EquipmentSpecfication.

The only exceptions to this recognizable naming convention are the configuration specifications:

- Logical Device Configuration Specification
- Network Configuration Specification
- Pipe Configuration Specification
- Place Configuration Specification
- Service Configuration Specification

To enable an extension point for a configuration specification, you must select the oracle.communications.inventory.api.entity.InventoryConfigurationSpec class, which enables the **Configuration Version Instance Type** field. See "Configuration Version Instance Type Field" for more information.

Configuration Version Instance Type Field

The **Configuration Version Instance Type** field is enabled only when you select oracle.communications.inventory.api.entity.InventoryConfigurationSpec for the **Specification Class Name** field.

When enabled, you may select a value for **Configuration Version Instance Type**, which is preloaded with the available entity configuration specification Java class names. The selection list displays the following fully qualified Java class names:

- oracle.communications.platform.entity.impl.LogicalDeviceConfigurationVersionDAO
- oracle.communications.platform.entity.impl.NetworkConfigurationVersionDAO
- oracle.communications.platform.entity.impl.PipeConfigurationVersionDAO

- oracle.communications.platform.entity.impl.PlaceConfigurationVersionDAO
- oracle.communications.platform.entity.impl.ServiceConfigurationVersionDAO

If you select a value for **Configuration Version Instance Type**, the extension point is enabled for the selected entity configuration specification. If you do not select a value for **Configuration Version Instance Type**, the extension point is enabled for all of the entity configuration specifications.

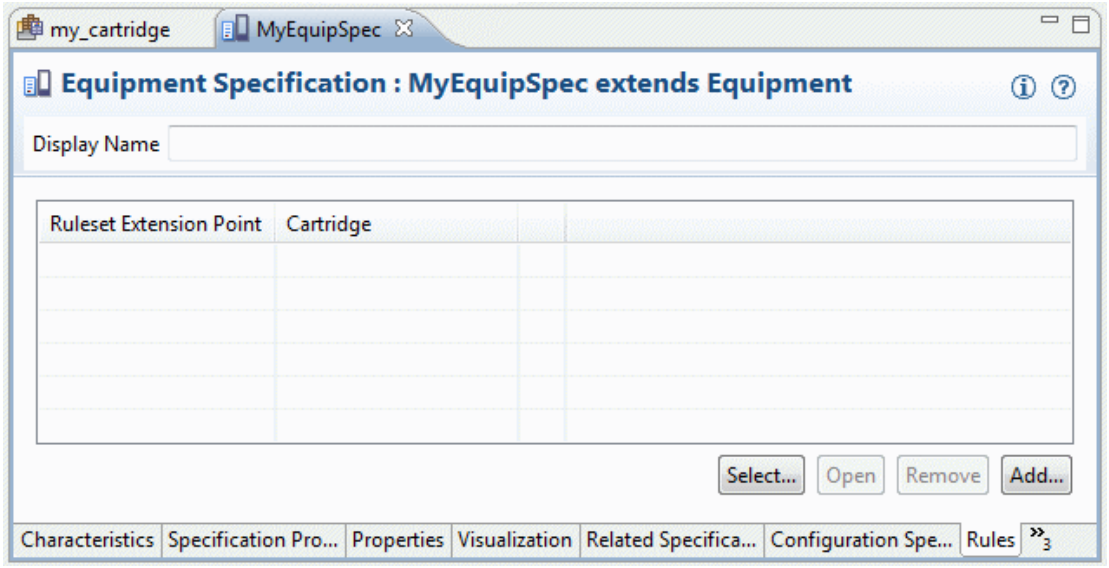
Configuring a Specification for a Ruleset Extension Point

Note: This section applies only to a ruleset extension points; it does not apply to global ruleset extension points.

To run a ruleset that is configured to run at a specification-based extension point, you must also configure the specification for the ruleset extension point. This configuration is done in Design Studio, on the **Rules** tab of any Specification editor. For example, Figure 8–10 shows that, when you click **Select** to select a ruleset extension point, only the ruleset extension points that are enabled for the Equipment Specification appear.

For instructions on how to configure a specification for a ruleset extension point, see the Design Studio Help.

Figure 8–10 Specification Editor Rules Tab



Validating and Compiling Rulesets

Rulesets are validated:

- As you write a ruleset. The respective rule editor, either Drools or Groovy, validates syntax to prevent compilation errors. These rule editors are not part of UIM or Design Studio, but can readily be installed. See "Installing, Configuring, and Using the Drools Eclipse Plug-ins" or "Installing, Configuring, and Using the Groovy Eclipse Plug-ins" for more information.

- When you build the project. Validations are performed to ensure that required values are supplied and that the specification configured for a ruleset extension point complies with the definitions in the enabled extension points.

Rulesets are compiled:

- When UIM is started and there are uncompiled rulesets (such as after an upgrade). The serialized compilation is stored in the database. If compilation errors are encountered, the startup fails and the errors are cited.
- When a cartridge that contains rulesets is deployed. The serialized compilation is stored in the database. If the compilation errors are encountered, the deployment fails and the errors are cited.

> **Note:** When compiling rulesets or building cartridges that contain rulesets, it is important that you configure your environment correctly to avoid errors later in the process.
>
> See "Configuring Design Studio" for more information.

Compiling Rulesets with Third-Party Dependencies

If your ruleset custom code is dependent upon third-party code for successful compilation, you must add the third-party JAR files containing the code upon which your custom code is dependent to the Inventory project. The third-party JAR files must be included in the Inventory project so when the resultant cartridge is deployed into UIM, the third-party code is available to the ruleset at runtime.

> **Note:** Adding third party JAR files to the Eclipse project library list successfully compiles dependent custom code, but if compiled in this manner, third party code is not part of the Inventory project and is not available at runtime.

To add third-party JAR files to your Inventory project:

1. In Design Studio, within the Studio Design perspective, open the Package Explorer view.
2. In the Package Explorer view, expand your Inventory project containing your ruleset and third party-dependent custom code.
3. Under the **model** directory, create the following directory structure: **content/inventory.ear/APP-INF/lib**.
4. Copy any required third-party JAR files into the **model/content/inventory.ear/APP-INF/lib** directory.
5. Build the project.

Deploying Cartridges Containing Rulesets

Deploying cartridges containing rulesets and extension points is no different than deploying other cartridges. See *UIM Cartridge Guide* for information about deploying cartridges and cartridge packs.

Running Rulesets

Rulesets can be run manually or automatically.

Manually Running Rulesets

Rulesets can be run manually from within UIM by clicking the **Execute Rule** link in the Tasks panel. Manually running rulesets is commonly used to manage UIM data. For example, you can manually run the SYSTEM_EXPORT base ruleset in one environment to export data, and manually run the SYSTEM_IMPORT base ruleset in another environment to load the exported data. See "About Base Rulesets" for more information.

Automatically Running Rulesets

Rulesets can be run automatically after they are deployed into UIM: When an event occurs that runs an existing UIM method that was defined as an extension point, the extensibility framework calls the RulesExecutor.execute() method, which runs the ruleset associated with the extension point.

Debugging Custom Drools Rulesets

For information on debugging custom rulesets (DRL files), see the *Drools Documentation* website:

`http://docs.jboss.org/drools/release/6.5.0.Final/drools-docs/html_single/`

Note: The Drools documentation describes debugging rulesets within Eclipse, within the context of a Drools project, not within the context of a UIM server.

Note: The Drools documentation on debugging states you must install the Eclipse Graphical Editing Framework (GEF) to debug rulesets. However, the Design Studio Inventory feature plug-in contains the GEF, so it is already installed.

For information on turning on debugging in UIM (to debug anything other than DRL files), see *UIM System Administrator's Guide*.

Debugging Custom Groovy Rulesets

This section provides information on debugging custom Groovy rulesets and extension points. Ensure that the steps for installing and configuring the Groovy Eclipse plug-ins are completed prior to setting up debug. See "Installing, Configuring, and Using the Groovy Eclipse Plug-ins" for more information.

Converting Inventory Projects to Groovy Projects

To convert an Inventory project to a Groovy project, perform the following steps before the project is deployed to the UIM server for the first time:

1. In the Design Studio Package Explorer view, right-click the desired Inventory project, and select **Configure**.

   A submenu for the **Configure** action appears.

2. Select **Convert to Groovy Project**.

   The project is converted.

Setting Up Debug Configurations

To set up a debug configuration to allow debugging of the project:

1. Ensure the UIM server is running so that a debugger can be connected. See the *UIM System Administrator's Guide* for more information.
2. In the Design Studio Project Explorer view, select your project.
3. From the **Run** menu, select **Debug Configurations**.

   The Debug Configurations window appears.
4. Click **Remote Java Application**, and then the **New** icon.

   A new remote java application is created with prompts for the settings.
5. Click **Browse** and select your project.
6. Click the **Connect** tab and enter the host and port information for the UIM server on which the cartridge with the custom rulesets is located.
7. On the **Common** tab, under the **Display in favorites menu,** select the **Debug** check box.
8. Click **Apply** to save the configuration.
9. Click **Close**.

   The Debug Configurations window closes.

Debugging Groovy Rules

To debug your Groovy rule code:

1. Set a breakpoint on at least one line of your code that is enabled. You can set a breakpoint by a double-click on a line of code, or right-click on the line of code and select **Toggle Breakpoint**.
2. In the Design Studio Project Explorer view, select your project.
3. From the **Run** menu, select **Debug Configurations**.

   The Debug Configurations window appears.
4. Under **Remote Java Application**, select your application.
5. Click **Debug**.

   The application is launched in debug mode and the debug perspective is displayed. The Eclipse debugger stops on the line of code where the breakpoint is set.

> **Note:** Only the secondary level of scripts can be debugged. The parent-level rule code does not recognize breakpoints; and only a breakpoint in a script that is called from the parent is recognized.

Troubleshooting Rulesets and Cartridge Deployment

This section provides information on troubleshooting problems you may encounter when working with custom rulesets, extension points, and cartridge deployment.

Troubleshooting Custom Rulesets

When troubleshooting custom rulesets, check the following:

- Does the ruleset compile?
  - Use the Drools or Groovy Eclipse plug-in editor.
  - Check the import statements.
  - Check the project library list.
- Does the cartridge build and deploy successfully?
  - Check the UIM application server log.
- If using Drools, does the ruleset condition ever evaluate to true?
  - Debug to find out.
- Are the ruleset argument values and return values correct?
  - Debug to find out.

Troubleshooting Custom Extension Points

When troubleshooting custom extension points, check the following:

- Is the extension point defined in both Design Studio and in a custom **aop.xml** file?
- Is the signature defined correctly in both places?
  - Check spacing.
  - Check spelling of package and class names.
- Regarding the weaver section in the custom **aop.xml** file:
  - Did you include it?
  - Are the package names correct?
- Did the cartridge build and deploy successfully?
  - Check the UIM application server log.
- Is the Weaver turned on?
  - Check the *UIM_Home/Domain_Home***/bin/setUIMEnv.cmd** file.
- After deploying a cartridge containing custom extension points, did you restart the UIM application server?
  - Check the UIM application server log to see if the custom extension point was successfully weaved into the UIM code stream.

Troubleshooting Configuring a Ruleset to Run at an Extension Point

When using base rulesets and base extension points, base ruleset extension points are not provided. You must configure base rulesets, and custom rulesets, to run at extension points.

- Did you create a ruleset extension point to configure the ruleset to run?
- Did you select the correct ruleset?
- Did you select the correct extension point?

- Did you select the correct placement of the rule to run before, after, or instead of the method?

  Be mindful of the rule placement. For example, if you are expecting your ruleset custom code to perform a process based on something the extension point method does, and you configure the ruleset to run before or instead of the method, you will not get the results you are expecting.

Troubleshooting Using Timing Events

If you set up rulesets based on timing events, be sure the *UIM_Home*/**config/timers.properties** file has the timing event you are using turned on. For example, if you configure a ruleset to run based on the timing of telephone number aging, and the timing event for this is not turned on, your ruleset will never run. For more information on the **timers.properties** file, see *UIM System Administrator's Guide*.

Troubleshooting Cartridge Deployment

When deploying a cartridge from Design Studio, you check the following items to ensure you can successfully deploy a cartridge.

Base Cartridges are Deployed

All required base cartridges are deployed into UIM before deploying other cartridges. Oracle recommends that you deploy all base cartridges even if they are not immediately required. See the *UIM Cartridge Guide* for more information on base cartridges.

Java JDK Version

If you encounter the Design Studio error shown in Example 8–9, the build of your project may be picking up an incorrect version of the Java. (In Example 8–9, *cartridgeName* is the name of the cartridge you are attempting to deploy, and *releaseNumber_buildNumber* is the release number and build number of the cartridge you are attempting to deploy.)

Example 8–9 Design Studio Error

```
Error installing cartridges: {<Cartridge cartridgeName releaseNumber_
buildNumber>=java.io.IOException: The model data cannot be imported. Please verify
logs for more information.}
```

In the server log, this error may appear as shown in Example 8–10. In this example, **MySampleManager** is a Java file in a cartridge that is trying to import **MyHelper**, which is another file in the same cartridge. However, **MySampleManager** cannot find **MyHelper** because **MyHelper.java** and the other files in the cartridge have inadvertently been compiled with the incorrect version.

Example 8–10 Server Log

```
2013-09-25 11:30:45,910 ERROR [] [[ACTIVE] ExecuteThread: '11' for queue:
'weblogic.kernel.Default (self-tuning)'] [RulesExecutor] [INV-180014]
A JBoss Rules compilation error occurred in: SAMPLE_DEVICE_ASSIGN
Error importing :
'oracle.communications.inventory.techpack.sample.MyHelper'
java.lang.Exception
at oracle.communications.inventory.extensibility.rules.impl.RulesExecutorImpl.
    getRuleBase(RulesExecutorImpl.java:151)
at oracle.communications.inventory.extensibility.rules.impl.RulesExecutorImpl.
```

```
    compileRules(RulesExecutorImpl.java:200)
at oracle.communications.inventory.extensibility.rules.RuleCompiler.run
    (RuleCompiler.java:128)
at weblogic.work.j2ee.J2EEWorkManager$WorkWithListener.run
    (J2EEWorkManager.java:184)
at weblogic.work.ExecuteThread.execute(ExecuteThread.java:256)
at weblogic.work.ExecuteThread.run(ExecuteThread.java:221)
```

To resolve this error, verify that you have configured Design Studio correctly, as described in "Configuring Design Studio". In particular, pay close attention to "Configuring the eclipse.ini File" because even though you may have your Eclipse compiler compliance level set correctly, Eclipse can still pick up an incorrect Java JDK if you have more than one Java version installed on your machine. You can also use a decompiler on your class files to verify the correct version of the Java JDK is being used.

Maximum Characteristics for a Table and Required Privileges

While deploying a cartridge, you can need **create table** privileges on the UIM database schema. Characteristics within a cartridge can require additional database tables by UIM core. This user privilege is required when the number of characteristics per entity exceeds the maximum number of columns (1000) in the table.

The error returned for this situation has the following text:

```
Error installing cartridges: ...
org.eclipse.persistence.exceptions.DatabaseException
Internal Exception: java.sql.SQLSyntaxErrorException:
ORA-01031: insufficient privileges
ORA-06512: at "U734B383.CREATE_CHAR_EXT_TABLE", ...
ORA-06512: at "U734B383.ADD_CHAR_COLUMN_TRIGGER", ...
ORA-04088: error during execution of trigger 'U734B383.ADD_CHAR_COLUMN_TRIGGER'
```

If you encounter this error, ensure the user deploying the cartridge has **create table** privileges for the database.

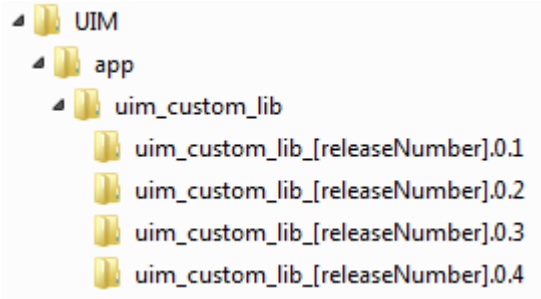
Existing Custom Extensions Overwritten

This section describes options to resolve the issue when a cartridge deployment is overwriting existing custom extensions. For a cartridge that has a custom **aop.xml** file, it is possible to overwrite another cartridge's **aop.xml** file. Try one of the following resolution options:

- Post-deployment option: Rename the **mslv-aop-filter-aop.jar** in the custom library before deploying another cartridge having a custom **aop.xml** file. This custom library resides in the **uim_custom_lib.ear** file in the *UIM_Home*/*Domain_Home*/**UIM/app/uim_custom_lib/uim_custom_lib**_*releaseNumber.version* directory, where *releaseNumber* is the UIM software release and *version* is the sequential number of changes to the library. You must select the highest existing version. Figure 8–11 shows a sample directory with four versions of the custom library. For this example, the correct directory to select is **uim_custom_lib**_*releaseNumber***.0.4** directory.

  Within the EAR file, the JAR file is located in the **APP-INF/lib** directory.

Figure 8–11 Sample Directories of Custom Library



- Pre-deployment option: When you can make the change before deployment, this is a more desirable option that the post-deployment option. Before deploying the cartridge, rename the **aop.xml** for every custom cartridge that has the XML file in the **model/aspects** folder to the filename:

  *prefix***Aop.xml**

  where *prefix* is any string value that makes the filename unique. This generates a **mslv-aop-filter-***prefix***Aop.jar** when the cartridge is built successfully.

- Clean-up option: To delete or disable a previously deployed ruleset, redeploy the same cartridge with an empty **aop.xml** file.

See "Creating the aop.xml File" for more information on creating custom **aop.xml** files.

Upgrading or Converting Rulesets

Rulesets may be written using either Drools or Groovy. The following sections further describe upgrading or converting rulesets in terms of Drools and Groovy.

Note: This section assumes that you have already upgraded UIM, so have already deployed the base cartridges for the current release, and deployed any applicable cartridge packs for the current release. See *UIM Installation Guide* for ore information on upgrading UIM.

Upgrading Drools Rulesets

Note: This section is only applicable if you:

- Plan to use Drools in this release
- Are upgrading from a previous release of UIM to UIM 7.2.3 or later
- Have existing custom rulesets written using Drools

If you plan to use Drools in this release and are new to UIM, this section is not applicable. If you plan to use Groovy in this release, see "Converting Drools Rulesets to Groovy Rulesets".

Previous versions of UIM (before 7.2.3) used Drools 3.0.4, and UIM 7.2.3 and forward uses a later Drools release. The Drools upgrade requires that you upgrade custom rulesets, which may reside in custom cartridges or in extended cartridge packs. See *UIM Cartridge Guide* for more information on extended cartridge packs.

See "Software Requirements" for information on the Drools version for UIM.

To upgrade custom rulesets:

1. Configure Design Studio. When upgrading custom rulesets, it is important that you configure your environment correctly to avoid errors later in the process.

   See "Configuring Design Studio".

2. In Design Studio, recompile all custom rulesets and all custom code your rulesets call.

   See "Validating and Compiling Rulesets" for more information.

3. If compilation errors are encountered, update custom rulesets and custom code as follows:

   a. Refer to Table 8–1 and update all occurrences of these commonly used 3.0.4 Drools methods to the newer Drools methods.

Table 8–1 Upgraded Drools Methods

3.0.4 Drools Method	Updated Drools Method
assert()	insert()
assertObject()	insert()
assertLogical()	insertLogical()
assertLogicalObject()	insertLogical()
modify()	update()
modifyObject()	update()

   b. If compilation errors remain, refer to the Drools Knowledge API Javadoc to determine if your custom code calls any other Drools methods that have changed. You can access the Drools Knowledge API Javadoc at the website:

      `http://docs.jboss.org/drools/release/5.5.0.Final/knowledge-api-javadoc/index.html`

      Then select the Drools version, and **knowledge-api-javadoc** directory.

4. Deploy all cartridges containing recompiled custom rulesets and custom code.

   See "Deploying Cartridges Containing Rulesets" for more information.

5. Test your changes by running the rulesets.

   See "Running Rulesets" for more information.

Converting Drools Rulesets to Groovy Rulesets

Note: This section is only applicable if you:

- Plan to use Groovy in this release
- Are upgrading from a previous release of UIM to UIM 7.3.0 or later
- Have existing custom rulesets written using Drools that you want to convert to Groovy

If you plan to use Groovy in this release and are new to UIM, this section is not applicable. If you plan to use Drools in this release, see "Upgrading Drools Rulesets".

To convert existing custom rulesets from Drools to Groovy:

1. In Design Studio, open the Studio Design perspective and the Studio Projects view.
2. Import the following base cartridges:
   - ora_uim_baserulesets
   - ora_uim_mds
   - ora_uim_model
3. Import your custom cartridge containing the custom ruleset you are converting.
4. Expand your custom cartridge that contains the custom ruleset you are converting, and open the custom ruleset.

   The Ruleset editor opens.
5. Click **Browse Drools Code**.

   The DRL file opens.
6. Copy all of the content and close the DRL file.
7. In the Ruleset editor, click **Browse Groovy Code**.

   The GROOVY file opens.
8. Update the rule section of the code.

   Note: All of the base rulesets provide both a DRL file and a GROOVY file. Look at the base rulesets for examples of how the code needs to change.
9. Save and close the GROOVY file.

   When the cartridge is deployed, both the DRL and the GROOVY files are updated in the RULESET database table in UIM.
10. In the Ruleset editor, click the **Run Extension Language** list arrow and select **GROOVY**.

    When the cartridge is deployed, this indicator is updated in the RULESET database table in UIM. At runtime, this indicator tells UIM which file to run; the DRL file or the GROOVY file.

11. Save and close the Ruleset.
12. Rebuild the cartridge that contains the ruleset.
13. Redeploy the cartridge that contains the ruleset into UIM.

Handling Concurrent Scenarios

Sometimes you may be required to concurrently update the entity relationships in the ruleset. For example, for an existing network, there may be a need to add nodes and edges concurrently. Typically, when you concurrently add nodes and edges in a network, there is a possibility of the network getting updated resulting in an Optimistic Locking exception.

To avoid this situation, add the following code in the ruleset to not update the network when nodes and edges are concurrently added to the network:

```
(NetworkNodeBaseDAO)ntwkNode1).setNetwork(net,false)
```

The above code adds the network node to the network without updating the network; however, it is possible that the network in the Eclipselink cache may get out of synch and not contain the newly added node/edge.

To clear the Eclipselink cache, add the following code at the end of the ruleset:

```
EntityManager em =
PersistenceHelper.makePersistenceManager().getPersistenceManager();
EntityManagerImpl emImpl = (EntityManagerImpl)  JpaHelper.getEntityManager(em);
Cache cache = ((EntityManagerFactoryDelegate)
emImpl.getEntityManagerFactory()).getCache();
cache.evictAll();
```

9

Using Rulesets for Bills of Materials

This chapter provides information about using rulesets to extend Oracle Communications Unified Inventory Management (UIM) Bills of Materials (BOMs).

See *UIM Concepts* for more information about BOMs.

About Cost Information for Bills of Materials

BOMs can be generated for engineering work orders, business interactions, and projects. Retrieving the BOM cost information is based on the resource and task specification tags. A database table is provided for storing the cost information. You must either write a SQL script that populates this table or provide your own method of storing and retrieving the cost information to fit your requirements.

Extending BOM Manager Methods

You can customize BOM data by using extension points in the BOMManager API interface, which generates BOMs. This interface includes methods that you can use or override to perform the following customizations:

- Provide your own cost retrieval functionality. You can extend the getCostFromReference() method with custom logic replacing this functionality. The getCostFromReference() method retrieves the cost for a specified cost reference ID. A string is the single input parameter to this method and it returns the cost value as a string. The input parameter is the cost reference tag provided in Design Studio. You can override this method and provide your own cost retrieval functionality. By default, this method retrieves and returns the COST field from the COSTREFERENCE table. The default SQL query is the following:

  ```
  SELECT COST FROM COSTREFERENCE WHERE COSTREFERENCEID = input_value
  ```

  where *input_value* is the cost reference ID key for the table row. See "Cost References" for more information.

- Provide additional information about BOM activities by using the populateAdditionInfoOnActivity() method. This method populates the Activity class attributes with information. The method takes the following input parameters:

  – BillOfMaterialActivity item

  – Activity object

  You can provide functionality that populates the Activity object attributes as needed. By default, this method is empty.

- Provide additional information about BOM resources by using the populateAdditionalInfoOnResource() method. This method populates the inventory resource class attributes with information. The method takes the following input parameters:
  - BillOfMaterialResource item
  - Inventory object (resource)

  You can provide functionality that populates the resource object attributes as needed. By default, this method is empty.
- Customize the format of BOMs by using the toXML() method on the BillOfMaterial object that is generated by the getBOM() method. The getBOM() method generates and returns a BillOfMaterial object. The method takes the following input parameters:
  - BOMEntityType (project or business interaction)
  - String for the entity identifier (the project name or business interaction ID)
  - BOMType (activity, quote, resource, or engineering type)

  You can use the toXML() method to format the output: for example, as a report or spreadsheet.

Cost References

BOMs can include cost information. Cost information for a resources or task is generated when the cost reference tag is associated with the entity specification. The tag includes a cost reference ID. By default, UIM uses the cost reference ID to retrieve the cost from a database table named COSTREFERENCE. You can populate this table by using a SQL script. Table 9–1 describes the required columns for the COSTREFERENCE table.

Table 9–1 COSTREFERENCE Table Columns

Column Name	Type
COSTREFERENCEID	String
COST	String

You can modify this default behavior. For example, you can call a third-party system or access a custom database table.

A cost reference tag can be associated with a specification in Design Studio. A cost reference ID must begin with the string **costreferenceid-** as a prefix. The private getCostReference() method finds the tag matching this prefix. This method uses the **costreferenceid** field as the key to look up the cost value.

Example 9–1 shows a snippet of the getCostReference() private method that shows how the tag is retrieved from the TagSpecificationRel object.

Example 9–1 getCostReference() Method Snippet

```
private String getCostReference(Specification spec) {
.
.
.
       List<TagSpecificationRel> Tags = spec.getTags();
       for (TagSpecificationRel tag : Tags) {
```

```
            Tag itemTag = tag.getTag();
            if (itemTag.getName().startsWith("costreferenceid-")) {
                return itemTag.getOtherInformation();
            }
        }
.
.
.
}
```

10

Extending Notifications

This chapter provides information about extending notifications in Oracle Communications Unified Inventory Management (UIM).

About Notifications

Notifications in UIM are messages that can optionally be sent to a user or a user group when a qualifying event occurs. By default, qualifying events relate to workflows and their activities, but notifications can be extended to other events. Notification messages inform the recipient that an event requires attention.

By default, UIM sends notifications when the following events occur:

- An activity in a workflow is changed to the Ready state, which indicates a user or group can work on the activity.
- An activity in a workflow is assigned to a user or a user group.

These events cause a notification to be sent to the user assigned to the activity. (For more information about activities and workflows, see *UIM Concepts*.)

Notifications are email messages by default. You can extend notifications to include other types of messages, such as Short Message Service (SMS) text messages. You can also customize message content. Additionally, you can extend notifications for other UIM events by creating custom events and handling these events using rulesets.

Note: Notifications are sent only if the system administrator has configured UIM to send notifications.

About Extending Notification Functionality

You can customize the UIM notification functionality in the following ways:

- By customizing the content and format of notification messages. You can specify the message text or template that is sent to users. You can format messages by using HTML markup, or you can use simple text format. You can use variables in message template strings. The variables are replaced in notification messages by values that are retrieved at the time the messages are sent. See "Understanding Notification Message Content" for more information about the structure and syntax of templates and variables.
- By changing the type of notification message sent or sending multiple types of messages. You customize the type of message sent by using different Java handler classes. UIM includes a handler for email messages by default. To send other types of messages, such as SMS text messages, you create a custom handler. See

"Changing the Type of Notification Messages Sent" for more information.

- By setting up notifications for additional types of events. For example, you might want to send an email when a service moves to a pending disconnect status. See "Adding Notifications for Additional Events" for more information.

When you customize UIM notification functionality, you supply the following information in the **system-config.properties** file:

- The names of the handler classes used for different message types
- The message template string
- The name of the resolver class used to resolve variables in the message template string

See "System Configuration Properties for Notifications" for a list of the notification properties you set in the **system-config.properties** file.

Understanding Notification Message Content

The content of a notification message consists of a message template and variables within that template. When a qualifying event occurs, the notification framework resolves the variables and sends the message.

The following sections describe the structure of variables and templates.

Understanding Message Variables

A notification message variable is a value that can be different in each notification. When generating an event, the Java notification resolver classes substitute values retrieved from the event for the variables in the template.

For example, you might use a variable to represent the sender of the message. When the notification message is generated, the sender's name or address is retrieved and is inserted into the message content.

Variables have the following syntax:

${*variableName*}

where *variableName* is the name of the variable.

For example, the variable ${activityName} in a message template represents the name of an activity. When a notification message is generated, the activity name (for example, "Print Reports") replaces the variable in the notification message.

If you want to use custom variables, you can create a custom Java resolver class that looks up and resolves each variable. For example, the resolver class can look up the following:

- A value from the activity event object
- A value from an API call
- A value from a customized property file

See "Customizing Message Content and Format" for additional examples of variable references in message templates.

Understanding Message Templates

Message templates provide the text that is included in every notification for a particular event. UIM includes a default message template for email notification messages. You can define your own template to modify the message content for email

messages or to define content for other types of messages. The message string is the value of a configuration property in the **system-config.properties** file.

You define message templates in the inventory.*EventClassName*.message.template property in the **system-config.properties** configuration file. *EventClassName* represents the name of your Java event class. You enter the full text and formatting marks of the message template in the inventory.*EventClassName*.message.template property.

See Table 10–1 " System Configuration Properties for Notifications" for additional information on this and other notification properties.

Notification message templates can have these formats:

- Rich text format using HTML tags. You use HTML tags to apply formatting to the text, such as bold, color, fonts, and so on.
- Simple text format that includes line feed characters (\n).

Both rich text and simple text templates can include variables.

Message Templates in HTML Format

You can use HTML tags to apply formatting to the message templates. Example 10–1 shows an email message template in HTML format that includes several different variables. Some variables are enclosed with quotation characters (\") when the quotes are desired in the output text. This example template is also the UIM default template.

> **Note:** This example includes extra white space for readability. You do not include extra white space in the template property value.

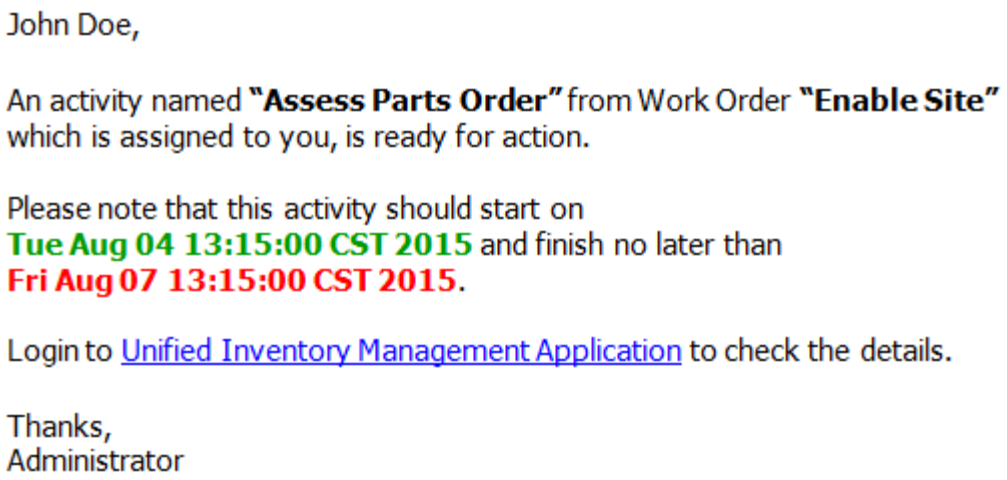
Example 10–1 Example Email Message Template in HTML Format

```
<html>
    <body>
        <pre>
            <span style=\"font-family: tahoma, arial, helvetica,
                    sans-serif; font-size: small;\">${notificationReceiver},
                <br />
                <br /> An activity named
                <strong> \"${activityName}\"</strong> from Work Order
                <strong>\"${workOrderName}\"</strong> which is assigned to you, is
                    ready for action.
                <br />
                <br /> Please note that this activity should start on
                <span style=\"color: #008000;\">
                    <strong>${activityStartDate}</strong>
                </span> and finish no later than
                <strong>
                    <span style=\"color: #ff0000;\">${activityEndDate}</span>
                </strong>.
            </span>
            <br />
            <br />
            <span style=\"font-family: tahoma, arial, helvetica, sans-serif;
                font-size: small;\"> Login to
                <a href=\"${uimURL}\">Unified Inventory Management Application</a>
                to check the details.
            </span>
            <br />
            <br />
            <br />
```

```
            <span style=\"font-family: tahoma, arial, helvetica, sans-serif;
                font-size: small;\"> Thanks,</span>
        </pre>
        <pre>
            <span style=\"font-family: tahoma, arial, helvetica, sans-serif;
                font-size: small;\"> ${notificationOriginator}</span>
        </pre>
    </body>
</html>
```

Figure 10–1 shows the email text that is the result of the template in HTML format in Example 10–1.

Figure 10–1 Email Message from an HTML Template



Example 10–2 shows how the example email template in HTML format is set as the value of the inventory.ActivityAssignmentEvent.message.template property.

Example 10–2 Example Email Message Template (HTML) as a Property Value

```
inventory.ActivityAssignmentEvent.message.template = <html><body><pre><span
style=\"font-family: tahoma, arial, helvetica, sans-serif; font-size:
small;\">${notificationReceiver},<br /><br /> An activity named<strong>
\"${activityName}\"</strong> activity from Work Order
<strong>\"${workOrderName}\"</strong> has been assigned to you. <br /> <br />
Please note that this activity should start on <span style=\"color:
#008000;\"><strong>${activityStartDate}</strong></span> and finish no later than
<strong><span style=\"color: #ff0000;\">${activityEndDate}</span></strong>.
</span><br /> <br /><span style=\"font-family: tahoma, arial, helvetica,
sans-serif; font-size: small;\"> Login to <a href=\"${uimURL}\">Unified Inventory
Management Application</a> to check the details. </span><br /><br /><br /><span
style=\"font-family: tahoma, arial, helvetica, sans-serif; font-size: small;\">
Thanks,</span></pre><pre><span style=\"font-family: tahoma, arial, helvetica,
sans-serif; font-size: small;\">
${notificationOriginator}</span></pre></body></html>
```

Message Templates in Simple Text Format

You can use minimal text formatting in message templates. Example 10–3 shows an email message template in simple text format.

Note: This example includes extra white space for readability. You do not include extra white space in the template property value.

Example 10–3 Example Email Message Template in Simple Text Format

```
${notificationReceiver}, \n\n

An activity named \"${activityName}\" from Work Order \"${workOrderName}\" has
been assigned to you. \n\n

Please note that this activity should start on ${activityStartDate} and finish no
later than ${activityEndDate}. \n\n

Login to ${uimURL} to check the details. \n\n

Thanks,\n\n
${notificationOriginator}
```

Example 10–4 shows how the example email template in simple text format is set as the value of the inventory.ActivityAssignmentEvent.message.template property.

Example 10–4 Example Email Message Template (Simple Text) as a Property Value

```
inventory.ActivityAssignmentEvent.message.template = ${notificationReceiver}, \n
\n  An activity named \"${activityName}\" from Work Order \"${workOrderName}\" has
been assigned to you. \n \n  Please note that this activity should start on
${activityStartDate} and finish no later than ${activityEndDate}. \n \n  Login to
${uimURL} to check the details. \n\n Thanks,\n\n ${notificationOriginator}
```

Extending Notifications

This section explains how to extend the notification functionality. You can extend the functionality to alter message content, notification type, or behavior. You can:

- Alter the message content of the message using property settings without making additional changes. For this extension, you only need to add the new template definition in the system configuration property file.
- Alter the content and the variables of the message. For this extension, you add the template text and provide a resolver class. The resolver class is specified with a configuration property.
- Alter the type of notification messages that UIM sends or alter the behavior of the default email message. For this extension, you specify one or more handler classes in the configuration properties. For example, the handler classes can change the functionality by:
  - Retrieving the recipients of the message from a third-party system.
  - Determining the type of notification to send, such as email, SMS or another type.
  - Sending multiple types of notification messages for one event.
  - Tailoring the subject of the message depending on some other criteria.

- Alter the notifications for additional custom events. For this extension, you specify one or more handler classes in the configuration properties.

Customizing Message Content and Format

You can customize the content and format of notification messages by defining a message template. The message content in your template can include variables.

To customize notification message content and format, you perform the following tasks:

- In the *UIM_Home*/**config/system-config.properties** file, where *UIM_Home* represents the directory into which UIM was installed, define the message template. You define the message template as the value of the inventory.*EventClassName*.message.template property, where *EventClassName* represents the name of your Java event class. Add a property entry for each event type to which the template applies, or to the InventoryEvent event class if the generic class applies.

  See "Understanding Notification Message Content" for information about the syntax of the template value.

- If your message template includes variables, do the following:
  - Create a custom resolver class to resolve the variables and replace them with values in the notification messages. The resolver class must extend the StrLookup class in the Apache Commons project.

    For information about the StrLookup class, refer to the Apache Commons project website:

    `https://commons.apache.org/`

  - In the **system-config.properties** file, specify your custom resolver class as the value of the inventory.*EventClassName*.variable.resolver property, where *EventClassName* represents the name of your Java event class. Add a property entry for each event type to which the template applies.

For more information about the notification properties in the **system-config.properties** file, see "System Configuration Properties for Notifications".

Changing the Type of Notification Messages Sent

You can change the type of notification messages that UIM sends. You can also send additional types of messages, such as SMS text messages.

To change the type of notification message sent, you perform the following tasks:

- Create a custom Java handler class that handles the new message type. For simplicity, create a handler class for each type of message that you want to send. Your custom handler classes must extend the NotificationHandler class.
- In the *UIM_Home*/**config/system-config.properties** file, specify your custom handler class in the value of the inventory.event.*EventClassName*.handler.list property. To specify multiple handler classes, separate the class names with a comma. UIM invokes the handlers in the order you list them.
- (Optional) If you want to define a new notification message template or modify an existing template for the new message type, perform the tasks described in "Customizing Message Content and Format".

See "System Configuration Properties for Notifications" for more information about the notification properties in the **system-config.properties** file.

Adding Notifications for Additional Events

You can extend UIM notifications to include additional events. To add a notification for a new event class, you perform the following tasks:

- Create a custom Java event class that extends InventoryEvent class.
- Create a custom Java handler class that handles the new event class. Your custom handler classes must extend the NotificationHandler class.
- In the *UIM_Home*/**config/system-config.properties** file, specify your custom handler class in the value of the inventory.event.*EventClassName*.handler.list property. To specify multiple handler classes, separate the class names with a comma. UIM invokes the handlers in the order you list them.
- Define a new notification message template for this new event class. Refer to the tasks described in "Customizing Message Content and Format" for more information on this topic.
- Create ruleset logic to inject code for any API manager method with an extension point. You include the handling of the new custom event in the rule logic.

  Refer to Chapter 8, "Extending UIM Through Rulesets" for more information on rulesets and extension points.

Your new custom event class includes the following:

- A constructor where you set the message information for this event, such as the subject and handler classes.
- A populateValueMap() method that returns a HashMap of the variable values.
- (Optional) A getUIMUrl() method if you need direction navigation from the email template to a specific UIM web page.

Example 10–5 gives an example of a constructor for a custom event class.

Example 10–5 Code Example for a Custom Event Class Constructor

```
public CustomEvent() {

String propertyName = "inventory." + this.getClass().getSimpleName() +
   ".message.template";
// get the template from the property file
setNotificationMessage(SystemConfig.getInstance().getProperty(propertyName,
    DEFAULT_MESSAGE_TEMPLATE));
List<NotificationHandler> notificationHandlerList =
    new ArrayList<NotificationHandler>();
// get the list of handlers from the property file
notificationHandlerList.add(NotificationHandlerFactory.getNotificationHandler(
    NotificationType.Email));
setNotificationHandlers(notificationHandlerList);
String subject = MessageResource.getMessage("custom.event",null);
setNotificationSubject(subject);
}
```

Example 10–6 gives an example of a populateValueMap() method for a custom event class.

Example 10–6 Code Example of a populateValueMap() method for a Custom Event

```
protected Map<String, String> populateValueMap() {
        HashMap<String, String> valuesMap = new HashMap<String, String>();
        valuesMap.put("notificationReceiver",
this.getUserDisplayName(this.getNotificationReceiver().get(0)));
        valuesMap.put("activityName", this.getActivityName());
        valuesMap.put("workOrderName", this.getWorkOrderName());
        valuesMap.put("activityStartDate", this.getActivityStartDate());
        valuesMap.put("activityEndDate", this.getActivityDueDate());
        valuesMap.put("uimURL", this.getUIMUrl());
        valuesMap.put("notificationOriginator",
this.getUserDisplayName(this.getNotificationOriginator()));
        return valuesMap;
    }
```

Example 10–7 gives an example of a getUIMUrl() method for a custom event class.

Example 10–7 Code Example of a getUIMUrl() method for a Custom Event

```
private static final String ACTIVITY_URL =
"http://${SERVER}:${PORT}/Inventory/faces/adf.task-flow?adf.tfId=MasterFlow&adf.tf
Doc=/WEB-INF/MasterFlow.xml&objectId=${ActivityOid}&entity=Activity&workOrder=${Wo
rkOrderOid}";
.
.
.
public String getUIMUrl() {
    String serverDetails = super.getUIMUrl();
    String[] serverDetailsArray = serverDetails.split(":");
    String serverName = serverDetailsArray[0];
    String serverPort = serverDetailsArray[1];
    HashMap<String, String> valuesMap = new HashMap<String, String>();
    valuesMap.put("SERVER", serverName);
    valuesMap.put("PORT", serverPort);
    valuesMap.put("ActivityOid", this.getActivityOid());
    valuesMap.put("WorkOrderOid", this.getWorkOrderOid());
    StrSubstitutor substitutor = new StrSubstitutor(valuesMap);
    String url = substitutor.replace(ACTIVITY_URL);
    return url;
}
```

Overview of Notification Java Classes

This section provides an overview of the Java classes that support UIM notification functionality. The Java classes are grouped into the following categories:

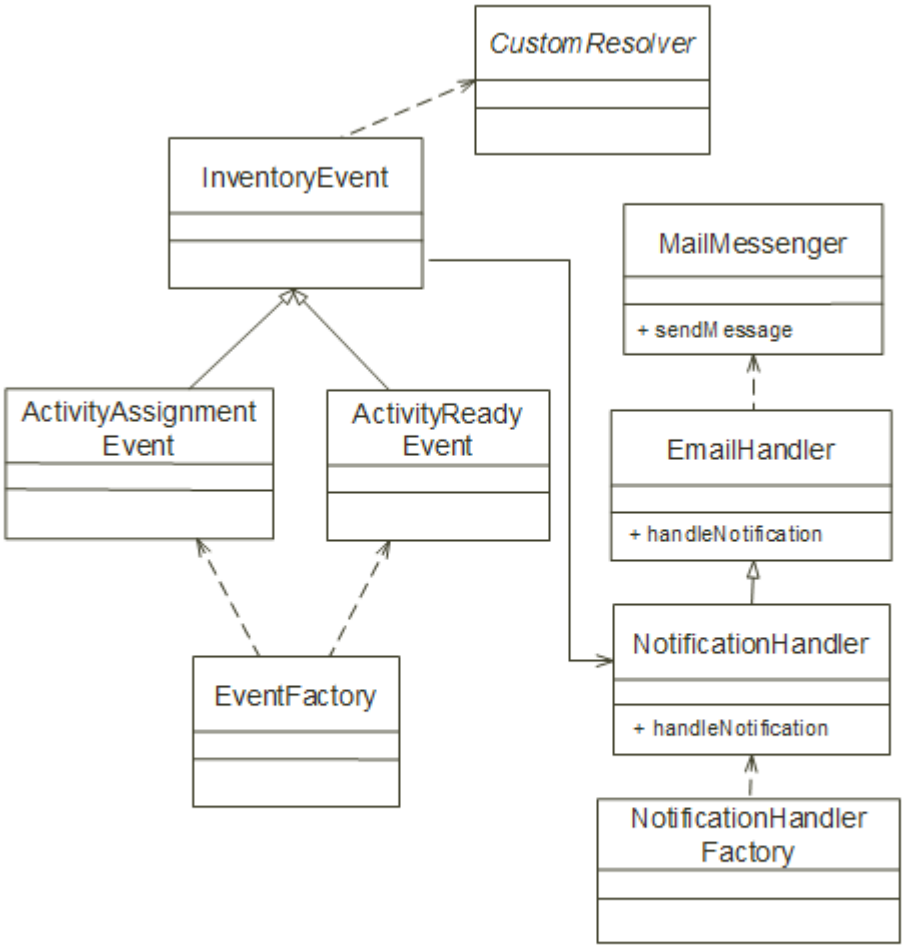
- Container or data classes that are the event classes
- Behavioral classes that are the handler and utility classes
- Internal UIM classes

Notification Functionality Class Diagram

The primary Java notification classes and their relationships are represented in Figure 10–2. The factory classes (EventFactory and NotificationHandlerFactory) create the specific instances of event or handler classes. ActivityAssignmentEvent and ActivityReadyEvent extend the abstract InventoryEvent class. EmailHandler, the UIM handler class, extends the abstract NotificationHandler class. MailMessenger is called

by EmailHandler. The InventoryEvent class calls the custom resolver class, when it is provided, in an extension.

Figure 10–2 Notification Functionality Class Diagram



About Event Java Classes

This section describes the classes used to manage the event data. These classes include getter and setter methods for the event data. The constructor sets many of the member values.

InventoryEvent Java Class

The InventoryEvent Java class is an abstract base class for ActivityAssignmentEvent and ActivityReadyEvent. You can create your own custom events by extending InventoryEvent. This base class maintains the generic notification data. The InventoryEvent class has getter and setter methods for the following attributes:

- The notificationOriginator string indicates the originating email of this event, such as the user or user group source of the email.
- The notificationMessage string is the content of the notification message.

- The notificationSubject string is the subject of the notification.
- The notificationReceiver is a Java list of strings indicating who is being notified.
- The notificationHandler is a Java list of NotificationHandler objects that represents the Java handler classes. The handler classes determine the notification behavior of event instances that extend InventoryEvent. See "Handler Classes" for definitions of the Java handler classes used for notifications.
- The copyNotificationTo is a Java list of strings indicating who is being copied on the notification. The strings specify the recipients' email addresses. This attribute is used only in extensions of the notification functionality.

Example 10–8 illustrates the package location and the declaration of the InventoryEvent class.

Example 10–8 InventoryEvent Class Declaration

```
package oracle.communications.inventory.api.framework.event;
public abstract class InventoryEvent
```

Activity Event Java Classes

The ActivityAssignmentEvent and ActivityReadyEvent classes manage activity event notification data. ActivityAssignmentEvent encapsulates the data required to notify users when they are assigned activities. ActivityReadyEvent encapsulates the data required to send a notification when an activity is in the Ready state. These classes have getter and setter methods for attributes, such as activity start date, activity due date, and activity name. The activity event classes extend the InventoryEvent class.

Example 10–9 illustrates the package location and the declaration of the ActivityReadyEvent class.

Example 10–9 ActivityReadyEvent Class Declaration

```
package oracle.communications.inventory.api.framework.event;
public class ActivityReadyEvent extends InventoryEvent
```

Example 10–10 illustrates the package location and the declaration of the ActivityAssignmentEvent class.

Example 10–10 ActivityAssignmentEvent Class Declaration

```
package oracle.communications.inventory.api.framework.event;
public class ActivityAssignmentEvent extends InventoryEvent
```

About Notification Behavior Java Classes

This section describes the classes used to manage the notification behavior and message variables. The handler and resolver classes do not have getter and setter methods.

Handler Classes

When an event occurs, the Java notification handler classes determine the behavior of the notification for the following:

- The recipients for the message
- The mechanisms to look up the message addresses
- The subject of the message

- The originator of the message
- The type of notification message to send, such as sending an email or a text

UIM includes the default EmailHandler notification handler class. A handler class must implement the interface NotificationHandler.

Resolver Classes

The Java resolver classes resolve the variables in the message templates for an event class. You can use the same resolver class for multiple types of notification messages. The resolver class looks up a variable string key and provides a string value result. You must define this class to extend the class StrLookup in the Apache Commons project. See "Understanding Message Variables" for more information on variables.

Note: For information about the StrLookup class, refer to the Apache Commons website:

`https://commons.apache.org/`

Overview of Internal Notification Java Classes

This section describes the Java classes that are part of the notification functionality but are internal to UIM. They are included in the documentation to provide clarity on the usage of all classes. The internal classes do not have getter and setter methods.

Factory Classes

The Java notification factory classes provide instances of the notification classes. The EventFactory class creates the event classes, and the NotificationHandlerFactory class creates a list of NotificationHandler classes.

NotificationType Class

The NotificationType class is an enumeration class that identifies the type of notification message to send. Example 10–11 show the class declaration, which enumerates the **Email** and **SMS** message types.

Example 10–11 NotificationType Enum Class

```
public enum NotificationType {
    Email,
    SMS
}
```

MailMessenger Class

The Java MailMessenger class is a utility class that sends email messages. The MailMessenger class does not have getter and setter methods. This class holds a private constant that identifies the mail session name.

Note: The system administrator must configure the server to set up the mail session as a postinstallation step after UIM is installed. For more information about this topic, see *UIM Installation Guide*. For more information about managing the email addresses for users and user groups, see *UIM System Administrator's Guide*.

System Configuration Properties for Notifications

This section describes the system configuration properties that you set when you customize UIM notification functionality. You add the properties to the **system-config.properties** file in the *UIM_Home*/**config** directory, where *UIM_Home* represents the directory into which UIM was installed. By default, these properties are not in the properties file; you add them only to extend the notification functionality.

When specifying the property name that includes an event name, you include the name of the event Java class for which to send notifications. For instance:

```
inventory.event.EventClassName.list
```

The property applies to that event class and to all its subclasses. The general syntax of the property entries is a name and value pair:

```
property_name=value
```

For properties that can have multiple values, such as the handler list properties, values must be separated by a comma. For example:

```
inventory.event.InventoryEvent.handler.list=CustomEmailHandler,CustomSMSHandler
```

The syntax of the message template property values depends on how you want to format the notification message. See "Understanding Notification Message Content" for information about how to specify the template value.

Table 10–1 describes the notification properties you set in the **system-config.properties** file.

Table 10–1 System Configuration Properties for Notifications

Property	Description
inventory.event.ActivityAssignment Event.handler.list	A comma-delimited list of Java handler classes that handle notifications when an activity event is assigned to a user. This handler list takes precedence over the generic event handlers defined in the inventory.event.InventoryEvent.handler.list property.
inventory.event.ActivityReadyEvent .handler.list	A comma-delimited list of Java handler classes that handle notifications when an activity event is moved to the Ready state. This handler list takes precedence over the generic event handlers defined in the inventory.event.InventoryEvent.handler.list property.
inventory.event.InventoryEvent.han dler.list	A comma-delimited list of Java handler classes that handle notifications for all events. This handler list takes lower precedence than the specific event handlers defined in the ActivityAssignmentEvent and ActivityReadyEvent properties.
inventory.event.*EventClassName*.han dler.list	A comma-delimited list of Java handler classes that handle notifications for a custom event where *EventClassName* is the name of the custom event class.
inventory.ActivityAssignmentEvent. message.template	Defines the notification message template. The template provides the static text of the notification message for the ActivityAssignmentEvent and can include variables that are resolved by the Java resolver class.
inventory.ActivityReadyEvent.mess age.template	Defines the notification message template. The template provides the static text of the notification message for the ActivityReadyEvent and can include variables that are resolved by the Java resolver class.

Table 10–1 (Cont.) System Configuration Properties for Notifications

Property	Description
inventory.*EventClassName*.message.template	Defines the notification message template. The template provides the static text of the notification message for a custom event where *EventClassName* is the name of the custom event class.
inventory.ActivityAssignmentEvent.variable.resolver	Specifies the Java resolver class name. This class determines the variable values in the message template for the ActivityAssignmentEvent event class. This resolver class looks up a variable string key and provides a string value result.
inventory.ActivityReadyEvent.variable.resolver	Specifies the Java resolver class name. This class determines the variable values in the message template for the ActivityReadyEvent event class. This resolver class looks up a variable string key and provides a string value result.
inventory.*EventClassName*.variable.resolver	Specifies the Java resolver class name. This class determines the variable values in the message template for a custom event where *EventClassName* is the name of the custom event class. This class looks up a variable string key and provides a string value result.
uim.host.name	Specifies the server host name. This is the Oracle WebLogic Server host name where UIM runs. This property is used in building the URL in the default email template text.
uim.host.port	Specifies the server port. This is the Oracle WebLogic Server port where UIM runs. This property is used in building the URL in the default email template text.

11

Customizing the User Interface

This chapter provides information on customizing the Oracle Communications Unified Inventory Management (UIM) user interface (UI), which is written using Oracle Application Development Framework (ADF) and Platform Common User Interface (CUI). The information in this chapter describes statically customizing the UI, which can result in backward compatibility issues. See "Backward Compatibility" for the implications regarding this type of extension.

UIM UI customizations are made in JDeveloper. After installing JDeveloper, you customize the UIM UI by importing the *UIM_Home*/**app/inventory.ear** file into JDeveloper and making the desired customizations. You then update the **inventory.ear** file with the customizations and redeploy it for testing.

Installing JDeveloper

> **Note:** Before installing JDeveloper, you must install the Java Development Kit (JDK). For information on installing JDK, see *UIM Installation Guide*.

JDeveloper is included in the Unified Inventory Management software distribution on the Oracle Software Delivery Cloud. To install JDeveloper, contact your system administrator for the location of downloaded the JDeveloper JAR file.

1. Copy the JDeveloper JAR file named jdev_suite_*version*.**jar** to a local directory, such as *JDev_Home* where *version* is a version number in the filename.
2. From a command line, navigate to *JDev_Home* and run the following command:

   ```
   java -jar jdev_suite_version.jar
   ```

   This initiates the JDeveloper installer.
3. On the Welcome window, click **Next**.

   The Choose Middleware Home Directory window appears.
4. Select **Create a new Middleware Home**, enter a middleware home directory name, and click **Next**.

   > **Note:** The remainder of this chapter refers to the middleware home directory you entered as *JDev_Home*.

   The Choose Install Type window appears.

5. Select **Complete** and click **Next**.

   The JDK Selection window appears.

6. Click **Browse** and navigate to your local installation of the JDK, and click **Next**.

   The Confirm Product Installation Directories appears.

7. Take the defaults and click **Next**.

   The Choose Shortcut Location window appears.

8. Take the defaults and click **Next**.

   The Installation Summary window appears.

9. Take the defaults and click **Next**.

   The installation begins.

10. When the installation completes, click **Done**.

Extracting the inventory.ear File into JDeveloper

To extract the **inventory.ear** file into JDeveloper:

1. Create a local directory, such as *tempEar_Home*.
2. Copy the *UIM_Home***/app/inventory.ear** file to *tempEar_Home*.
3. Double-click the *JDev_Home***/jdeveloper/jdeveloper.exe** file.
4. Set the role to **Studio Developer** by doing one of the following:
   - If the Select Role window appears when you launch JDeveloper:

     Select **Studio Developer (All Features)** and click **OK**.
   - If the Select Role window does not appear when you launch JDeveloper:

     From the JDeveloper menu, select **Tools**, then select **Switch Roles**, then select **Studio Developer (All Features)**.
5. From the JDeveloper menu, select **File**, then select **New**, then select **From Gallery**.

   The New Gallery window appears.
6. Under **Categories**, expand **General**, and select **Applications**.
7. Under **Items**, select **Application from EAR File**, and click **OK**.

   The Create Application from EAR File window appears. This window has three parts: **Location**, **Ear Modules**, and **Finish**. **Location** appears first.
8. Next to the **EAR File** field, click **Browse** and navigate to the *tempEar_Home***/inventory.ear** file.

   Selecting the **inventory.ear** file automatically populates the fields on this window as follows:
   - **EAR File** defaults to *tempEar_Home***/inventory.ear**, based on the selected EAR file.
   - **Application File** defaults to **inventory**, based on the name of the selected EAR file.
   - **Directory** defaults to **C:/JDeveloper/mywork/inventory**. You can change the defaulted directory to any directory you prefer. The directory specified gets created by the process you are about to initiate.

- **Source Roots** defaults to *tempEar_Home*.

9. Leave the **Copy Files to Application** check box deselected, and click **Next**.

   Ear Modules appears.

10. Accept the default module names and project names, and click **Next**.

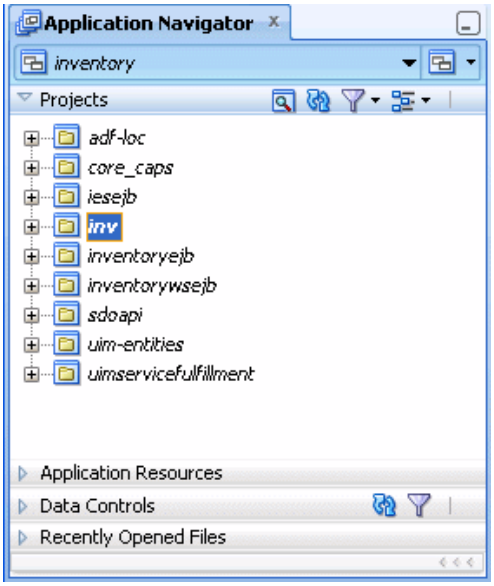
    Finish appears, showing the **inventory.ear** file location, and the location of the projects that JDeveloper is about to build based on the modules in the **inventory.ear** file.

11. Click **Finish**.

    JDeveloper does the following:

    - Creates a workspace. The workspace directory name and location are based on the directory name and location specified in the **Directory** field on the Location window.
    - Creates an application in the workspace. The application name (inventory) is based on the imported EAR file name.
    - Creates several projects within the inventory application, as shown in Figure 11–1. Each project name is based on a module name from the selected **inventory.ear** file.

Figure 11–1 Inventory Application Projects



12. Delete all of the projects except the **inv** project by doing the following:

    a. Select all of the projects except the **inv** project.

    b. Right-click on the group of selected projects and select **Delete Project**.

       The Confirm Delete Project window appears.

    c. Select **Remove projects from application**, and click **Yes**.

Configuring the JDeveloper Project

You must configure the JDeveloper project to successfully compile the project.

To configure the JDeveloper project:

1. Outside of JDeveloper, copy the following JAR files from the *UIM_Home*/**lib** directory to a temporary directory used for the duration of extending the UI:
   - capacity_caps.jar
   - characteristic_caps.jar
   - comms-platform-ui.jar

     The **comms-platform-ui.jar** file is located in the *UIM_Home*/**lib/comms-platform-webapp.war/WEB-INF/lib** directory.
   - consumable_caps.jar
   - core_caps.jar
   - groupenabled-caps.jar
   - ies.jar
   - ojdbc*version*.jar
   - poms.jar
   - sdoapi.jar
   - stringtemplate-*version*.jar
   - uim-api-framework.jar
   - uim-caps.jar
   - uim-entities.jar
   - uim-entity-xmlbean.jar
   - uim-managers.jar
   - uim-webservices-framework.jar
   - uim-webservices-framework-xsd.jar

   where *version* is the version number in the filename.
2. In addition, copy the following JAR files from the cited directory to the same temporary directory:
   - adf_richclient-api-*version*.jar

     The **adf_richclient-api-*version*.jar** file is located in the *DOMAIN_NAME*/**servers/AdminServer/tmp/_WL_user/adf.oracle.domain.webapp/***directory_name*/**WEB-INF/lib** directory, where *version* is the version in the filename and *directory_name* is an auto-generated name that varies per installation.
   - platform-managers.jar

     The **platform-managers.jar** file is located in the *UIM_Home*/**app/***version*/**uim_core_lib.ear/APP-INF/lib** directory, where *version* is the UIM version number.
   - adfsharembean.jar

     The **adfsharembean.jar** file is located in the *Oracle_Home*/**oracle_common/modules/oracle.adf.share**_*version* where *version* is the version number in the directory name.
3. In JDeveloper, select the **inv** project, right-click, and select **Project Properties**.

The Project Properties window appears.

4. In the navigation panel, click **Libraries and Classpath**.

   The Project Properties Libraries and Classpath window appears.

5. Click **Add JAR/Directory**.
6. Navigate to the temporary directory that contains the copied JAR files.
7. Select all of the copied JAR files and click **OK**.

   The JAR files are added to the Project Properties Libraries and Classpath window.

8. Click **OK** again to close the Project Properties window.

Customizing the User Interface

Customizations can be in the form of new files or additions to existing files. If you are deleting files or modifying existing files with changes or deletions, be aware of the errors this may cause. These types of errors are logged by Oracle WebLogic Server when you deploy the updated **inventory.ear** file.

Note: You cannot customize the UIM home page.

About the UI Files

UIM UI customizations involve several types of files, such as JSFF, XML, Java, and XLF files, as described in the following sections.

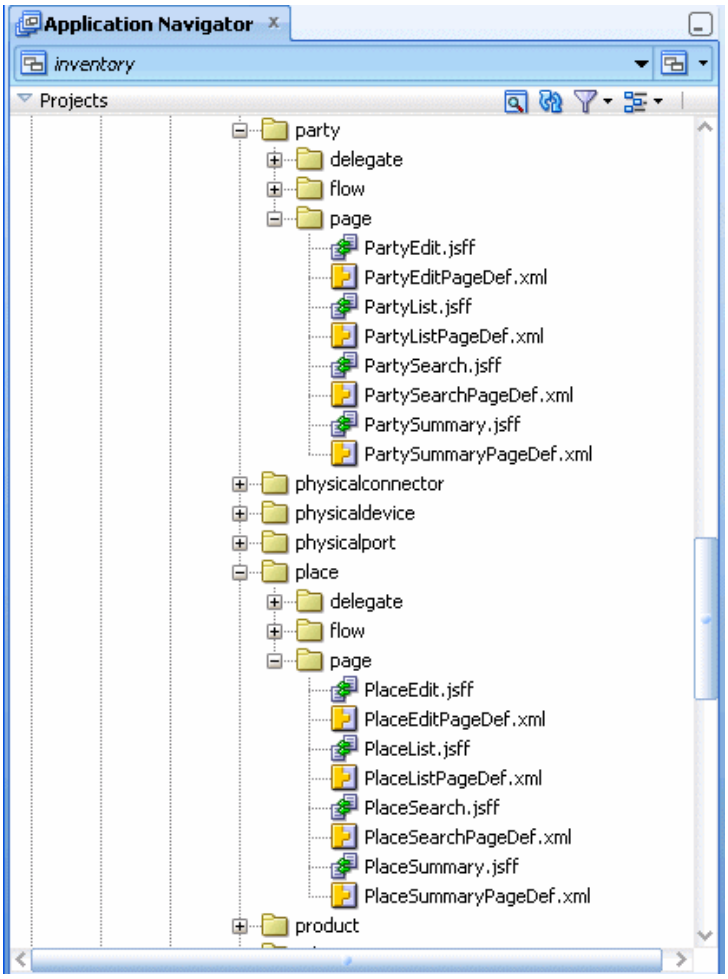
JSFF and XML Files

Each page in the UIM UI is defined by a JSFF and XML file. For example, the UIM Party Summary page is defined by the **PartySummary.jsff** and **PartySummaryPageDef.xml**, and the UIM Party Maintenance page is defined by the **PartyEdit.jsff** and **PartyEditPageDef.xml** files.

These files are located within the inventory application inv project, in the **Web Content/oracle/communications/inventory/ui/***functionalArea***/page** directory, where *functionalArea* is a UIM functional area such as equipment, number, service, and so forth.

Within each *functionalArea***/page** directory, the JSFF and XML file names follow the naming convention shown in Figure 11–2. For example, each file name contains the entity name (Place, Party, and so forth), and the Web page (Search, List, Summary, Edit, and so forth). The XML page file names end with **PageDef**.

Figure 11–2 Page File Naming Conventions



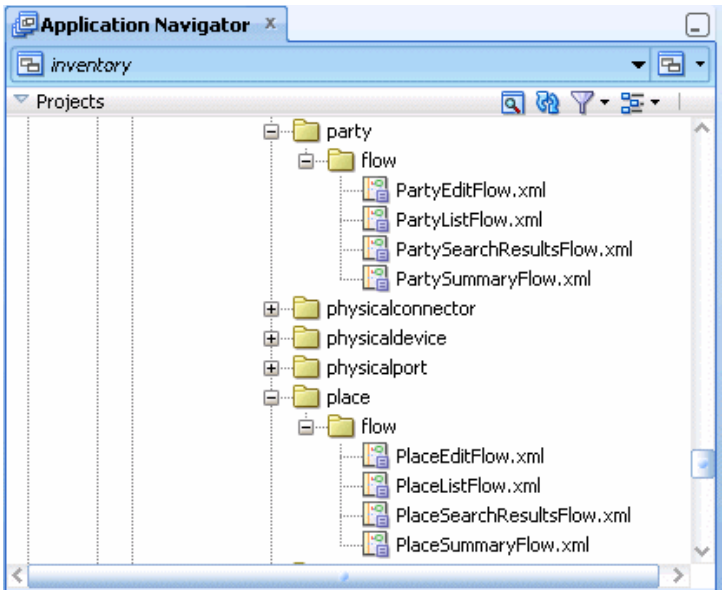
XML Files

Each page in the UIM UI has a specific task flow defined by an XML file. For example, the UIM Party Summary page task flow is defined by the **PartySummaryFlow.xml** file, and the UIM Party Maintenance page task flow is defined by the **PartyEditFlow.xml** file.

These files are located within the inventory application inv project, in the **Web Content/WEB-INF/oracle/communications/inventory/ui/***functionalArea***/flow** directory, where *functionalArea* is a UIM functional area such as equipment, number, service, and so forth.

Within each *functionalArea***/flow** directory, the XML file names follow the naming convention shown in Figure 11–3. For example, each file name contains the entity name (Place, Party, and so forth), and the Web page (Search, List, Summary, Edit, and so forth). The XML task flow file names end with **Flow**.

Figure 11–3 Task Flow File Naming Conventions



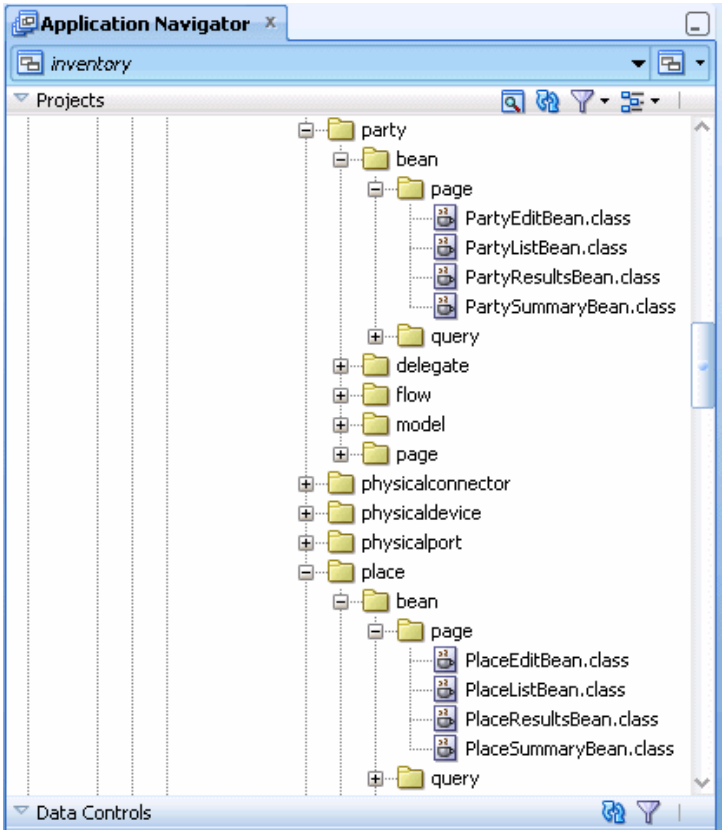
Java Files

The functionality of each page in the UIM UI is driven by logic in a Java source file that is compiled into a Java class file. For example, the UIM Equipment Summary page is driven by the **EquipmentSummaryBean.class** file, and the Equipment Maintenance page is driven by the **EquipmentEditBean.class** file.

These files are located within the inventory application inv project, in the **Web Content/WEB-INF/classes/oracle/communications/inventory/ui/***functionalArea* **/bean/page** directory, where *functionalArea* is a UIM functional area such as equipment, number, service, and so forth.

Within each *functionalArea***/bean/page** directory, the Java file names follow the naming convention shown in Figure 11–4. For example, each file name contains the entity name (Place, Party, and so forth), and the Web page (Search, List, Summary, Edit, and so forth). The Java file names end with **Bean**.

Figure 11–4 Java File Naming Conventions



XLF Files

XLF files define text values that display throughout the UIM UI. XLF files also define formats that are used to display the text values in a specific way.

The **InventoryUIBundle.xlf** file, which defines text values, is located within the inventory application inv project, in the **Web Content/WEB-INF/classes/oracle/communications/inventory/ui/common/bundle** directory.

The **Preferences.xlf** file, which defines the DATE_FORMAT, is located within the inventory application inv project, in the **Web Content/WEB-INF/classes/oracle/communications/platform/ui** directory.

Note: If you customize the DATE_FORMAT in the **Preferences.xlf** file, you must also change the system.dateFormat specified in the *UIM_Home***/config/resources/logging/system.properties** file.

When entity managers throw informational, warning, or error messages that contain a date, the message date is not formatted using the DATE_FORMAT specified in the XLF file. Rather, the message date is formatted using the system.dateFormat specified in the **system.properties** file. So, if you customize the date format, you must change it in both files.

DCX File

The **DataControls.dcx** file defines the registry for all the delegates, which are defined as data controls. If your customizations require a new delegate, this file needs to be updated to include the new delegate.

This file is located within the inventory application inv project, in the **Web Content/oracle/communications/inventory/ui** directory.

The following websites are useful when working with DCX files to customize the UIM UI:

- The *ADF Tasks* virtual book provides ADF information and is available at:

  http://www.oracle.com/pls/as111130/vbook_subject?subject=adf

- The *Oracle Fusion Middleware Web User Interface with Oracle ADF Faces* online documentation is available at:

  http://docs.oracle.com/middleware/12212/adf/develop-faces/toc.htm

- Additional information can be found at:

  http://docs.oracle.com/middleware/12212/adf/docs.htm

Displaying Custom Attributes on a Web Page

Custom attributes are any attributes that you have added to an existing entity. You can display custom attributes by editing the JSFF files for the entity's functional area. For example, if you add the **subscriberId** attribute to the Service entity, you can display **subscriberId** on the UIM Service Summary page by editing the **inv/oracle/communications/inventory/ui/service/page/ServiceSummary.jsff** file.

To display the value of the **subscriberId** attribute, add the following component to the JSFF file:

```
<af:outputText
value="#{pageFlowScope.ServiceSummaryBean.entityObject.subscriberId}"/>
```

Adding Custom Input Fields to a Web Page

You can edit the value of custom attributes in the UIM UI by adding an input field to the JSFF file for the entity's maintenance page.

By convention, maintenance page file names end with **Edit**. For example, **EquipmentEdit.jsff** maintains an equipment entity, and **ServiceEdit.jsff** maintains a service entity. Maintenance pages operate in two modes:

- **New:** For creating a new instance of the entity
- **Edit:** For modifying an existing instance of the entity

A managed bean exists for every entity, and the bean contains all of the attributes defined for the entity. For example, for an equipment entity, the page is **EquipmentEdit.jsff** and the Java class is **EquipmentEditBean.class**. Similarly, for a service entity, the page is **ServiceEdit.jsff** and the Java class is **ServiceEditBean.class**.

If the **type** attribute is added to the Equipment entity, **type** can be displayed on the UIM Equipment Summary page and edited on the UIM Equipment Maintenance page.

To do this, edit the following files:

- In the **InventoryUIBundle.xlf** file, add the following to define the text for the **type** attribute as it displays in the UI:

```
<trans-unit id="TYPE">
        <source>Type</source>
    <target/>
</trans-unit>
```

- In the **EquipmentSummary.jsff** file, add the following ADF component:

```
<af:panelLabelAndMessage label="#{inventoryUIBundle.Type}" id="plam2">
    <af:outputText
    value="#{pageFlowScope.EquipmentSummaryBean.entityObject.type}" id="ot3"/>
</af:panellabelAndMessage>
```

- In the **EquipmentEdit.jsff** file, add the following ADF component to edit the field:

```
<af:inputText value="#{pageFlowScope.EquipmentEditBean.entityObject.type}"
label="#{inventoryUIBundle.Type}" id="it1"></af:inputText>
```

Adding Conditional Components to a Web Page

Components are Web page building blocks. For example, the OutputText component is used for displaying entity attribute values on a Web page, and the InputText component is used for editing entity attribute values on a Web page. Conditional components are components that may or may not be rendered on a Web page, depending upon the outcome of an expression that can be evaluated. You can make a component a conditional component with custom logic.

To make a component a conditional component, edit the JSFF page file and map the component to a Java method, which evaluates an expression. The expression can be implemented with a custom logic class that extends the original bean class. For example, to make the **Activate/Deactivate** check box attribute on the UIM Equipment Maintenance page conditional upon an active condition being true, make the following changes:

- In the **EquipmentEdit.jsff** file:

```
<af:selectBooleanCheckbox value="#{pageFlowScope.EquipmentEditBean.active}"
text="#{inventoryUIBundle.Active}"
disabled="#{!(pageFlowScope.EquipmentEditBean.active)}" id="it7">
```

- Create a new Java class that extends **EquipmentEditBean.java**, and have the class define the following method:

```
public boolean getActive()
{
    if(this.getEquipment() != null && this.getEquipment() instanceof Equipment)
    {
        Equipment equipment = (Equipment)this.getEquipment();
        InventoryState  inventoryState = equipment.getAdminState;
        return(inventoryState != null
        !InventoryState.equals(InventoryState.END_OF_LIFE):true);
    }
    return false;
}
```

Disabling an Input Field on a Web Page

You can disable InputText components based on a condition. For example, to make the **type** attribute on the UIM Equipment Maintenance page conditional upon an active condition being true, make the following changes to the **EquipmentEdit.jsff** page file:

```
<af:inputText value="#{pageFlowScope.EquipmentEditBean.entityObject.type}"
```

```
label="#{inventoryUIBundle.Type}" id="it1"
disabled="#{!(pageFlowScope.EquipmentEditBean.active)}">
</af:inputText>
```

Adding a Custom Action to a Web Page

You can add a custom action to a Web page by editing the JSFF page file to include a link or a button to call a custom listener method on the page. For example, to add a button to the UIM Service Summary page, which calls the generateReport() method, make the following changes to the **ServiceSummary.jsff** file:

```
<af:commandButton actionListener=#{pageFlowScope.ServiceSummary.generateReport}">
    <af:outputTextvalue="Text value of the link"/>
</af:commandButton>
```

A custom class needs to implement this method, and the custom class needs to run in place of the original. This is done using rulesets and extension points. For information on rulesets and extension points, see Chapter 8, "Extending UIM Through Rulesets".

If the custom class is **com.foo.ServiceSummary.class**, the **ServiceSummary.java** source file would reside in the **inv/Web Content/WEB-INF/src/com/foo** directory and would contain the following:

```
package com.foo;
public abstract class ServiceSummary
extends oracle.communications.inventory.ui.service.bean.page.ServiceSummaryBean
{
    public void generatReport(ActionEvent event)
    {
        // perfrom the custom logic here
        System.out.println(getService().toString());
    }
}
```

Next, the **inv/Web Content/WEB-INF/oracle/communications/inventory/ui/service/flow/ServiceSummaryFlow.xml** task flow must be edited to add the new **com.foo.ServiceSummary.class** to the pageFlowScope.

Adding a Custom Search Field

You can add a custom search field to existing search criteria. For example, you can add the **Grade** field to the Telephone Number Search criteria. To do this, you must extend the API that the UI calls, as well as the UI.

Extending the API

To add a custom search field to existing search criteria and extend the API to take this new field into account:

1. Write a ruleset to extend the TelehoneNumberManager.findTelephoneNumber() method. Set the rule to have placement before the findTelephoneNumber() method.
2. In the extension code, add your business-specific code to include new search criteria to restrict the result set. In the extension, you set the custom query with the setCustomJPQL() method on the existing UIM search criteria class. You also need to set the attributes and values for the criteria with the setCustomParameters() method.

See Chapter 8, "Extending UIM Through Rulesets" for information on how to write a ruleset.

Example 11–1 is an example code section of a business interaction query and using the setCustomJPQL() and setCustomParameters() methods.

Example 11–1 Using the setCustomJPQL() and setCustomParameters() Methods

```
BusinessInteractionSearchCriteria criteria =
        biMgr.makeBusinessInteractionSearchCriteria();

criteria.setCustomJPQL(" AND o.id LIKE :id AND o.name LIKE :name");

String[] attributes = {"id", "name"};
Object[] values = new Object[] {"225005", "CREATE SERVICE" };

criteria.setCustomParameters(attributes, values);

Collection<BusinessInteraction> bis = biMgr.findBusinessInteraction(criteria);
```

See Chapter 8, "Extending UIM Through Rulesets" for information on how to write a ruleset.

Extending the UI

TelephoneNumberSearch.jsff renders **InventoryQuery.jsff** to build the query criteria on the page. To add the **Grade** field:

1. Create a new custom class, such as **TNQueryBean.java**, that extends the **oracle.communications.inventory.ui.number.bean.query.TelephoneNumberQueryBean** class. To add a new field, the getAttributeDescriptors() method needs to be overridden.
2. Your custom class, **TNQueryBean.java**, needs to have something as shown in the following example. (In the getAttributeDescriptors() method, the fields in the queryAttributes String Array are rendered on the UI as search fields.)

```
package oracle.communications.inventory.ui.number.bean.query;

import java.util.List;
import oracle.adf.view.rich.model.AttributeDescriptor;
import oracle.communications.platform.ui.bean.query.AttributeDescriptorImpl;
import oracle.communications.inventory.ui.common.utils.Constants;
import oracle.adf.view.rich.model.AttributeDescriptor.ComponentType;

public class TNQueryBean extends TelephoneNumberQueryBean
{
    public static final String GRADE = "GRADE";

    public TNQueryBean() { super(); }

    protected List<AttributeDescriptor> getAttributeDescriptors()
    {
        List<AttributeDescriptor> attributeDescriptors =
            super.getAttributeDescriptors();

        AttributeDescriptorImpl attributeDescriptor = null;
        attributeDescriptor =
            this.createAttributeDescriptorImpl
               ("GRADE", "GRADE", Constants.STRING_TYPE,null,
```

```
                ComponentType.inputText);
        attributeDescriptors.add(attributeDescriptor);

        String[] queryAttributes =
            {TELEPHONE_NUMBER, RANGE_FROM, Constants.SPECIFICATION, RANGE_TO,
           SERVICE_SPECIFICATION, INVENTORY_GROUP, Constants.INVENTORY_STATUS,
            CONDITION_TYPE, Constants.RESOURCE_ASSIGNMENT_STATUS, GRADE};
        this.setQueryAttributes(queryAttributes);

        return attributeDescriptors;
    }
}
```

3. Change the **TelephoneNumberSearchResultsFlow.xml** file to add your custom class (**TNQueryBean**) in pageFlowScope, as shown below:

```
<managed-bean>
    <managed-bean-name>InventoryQueryBean</managed-bean-name>
    <managed-bean-class>
        oracle.communications.inventory.ui.number.bean.query.TNQueryBean
    </managed-bean-class>
    <managed-bean-scope>pageFlow</managed-bean-scope>
    <managed-property>
        <property-name>beanClass</property-name>
        <property-class>java.lang.String</property-class>
        <value>
            oracle.communications.inventory.api.entity.TelephoneNumber
        </value>
    </managed-property>
    <managed-property>
        <property-name>searchName</property-name>
        <property-class>java.lang.String</property-class>
        <value>Telephone Number Search</value>
    </managed-property>
</managed-bean>
```

4. Create a new custom class, such as **TNDelegate.java**, that extends the **TelephoneNumberDelegate** class. In the custom class, override the getSearchCriteria() method to pass the **Grade** field to API. On the API side, extend **oracle.communications.inventory.api.number.TelephoneNumberSearchCriteriaImpl**, and define the **Grade** field as a member. This is shown in the following example:

```
import oracle.communications.inventory.ui.common.utils.CriteriaContainer;
import
oracle.communications.inventory.api.number.TelephoneNumberSearchCriteria;

public class TNDelegate extends TelephoneNumberDelegate
{
    public TNDelegate() { super(); }

    protected CriteriaContainer getSearchCriteria()
    {
        CriteriaContainer container = super.getSearchCriteria();

/* Get the criteria from the container and cast it to the
CustomTNSearchCriteriaImpl custom telephone number search criteria */
CustomTNSearchCriteriaImpl criteriaTNObj =
(CustomTNSearchCriteriaImpl)container.getCriteria();
```

```
/* Create a CriteriaItem for the newly added Grade field and set the item in
the criteriaTNObj custom telephone number search criteria */
if (searchFieldDefs.containsKey("GRADE") &&
       criteriaTNObj != null) {
       CriteriaItem item = searchFieldDefs.get("GRADE");
       item.setValue((item.getValue().toString()));
       criteriaTNObj.setGrade(item);
}
/* Pass criteriaTNObj to the finder method and return the container */
container = new CriteriaContainer(tnManager, "findTelephoneNumbers",
criteriaTNObj);

        return container;
    }
}
```

5. Update the **DataControls.dcx** file to include the new **TNDelegate**, as shown below:

```
<AdapterDataControl id="TelephoneNumberDelegate"
FactoryClass="oracle.communications.inventory.ui.framework.datacontrol.Inventor
yDataControlFactoryImpl"
ImplDef="oracle.adf.model.adapter.beanBeanDefinition"
SupportTransactions="false"
SupportsSortCollection="true"
SupportsResetState="false"
SupportsRangesize="false"
SupportsFindMode="false
SupportsUpdates="true"
Definition="oracle.communications.inventory.ui.number.delegate.TNDelegate"
BeanClass="oracle.communications.inventory.ui.number.delegate.TNDelegate"
xmlns="http://xmlns.oracle.com/adfm/datacontrol">
```

6. Copy **TelephoneNumberDelegate.xml**, paste it in the same directory, and rename it **TNDelegate.xml**. Afterward, open **TNDelegate.xml** and change all occurrences of **TelephoneNumberDelegate** to **TNDelegate**.

Deploying User Interface Customizations

To deploy your UIM UI customizations:

1. In JDeveloper, create the **inv.war** file:
   a. In the Application Navigator, right-click on the **inv** project, select **Deploy**, then select **inv**.

      The Deploy inv window appears. This window has two parts: **Deployment Action**, and **Summary**. **Deployment Action** appears first.

   b. Select **Deploy to WAR** and click **Next**.

      Summary appears, showing the location of **inv.war** file after JDeveloper builds it.

   c. Click **Finish**.

Note: Neither the created **inv.war** file nor the created **deploy** directory in which **inv.war** resides displays in JDeveloper, even after a refresh. To see the **inv.war** file, navigate to your JDeveloper workspace outside of the JDeveloper application.

---

2. Update the **inventory.ear** file to include the updated version of the **inv.war** file you just created:
   a. Outside of JDeveloper, navigate to *tempEar_Home*.
   b. Open the **inventory.ear** file.
   c. Add the **inv.war** file to the **inventory.ear** file, replacing the existing **inv.war** file with the **inv.war** file that contains your UI customizations.
   d. Save and close the **inventory.ear** file.
   e. Copy the updated **inventory.ear** file from *tempEar_Home* to the *UIM_Home*/**app** directory.
3. Deploy the updated **inventory.ear** file.

   For instructions on how to deploy the **inventory.ear** file, see *UIM System Administrator's Guide*.

Customizing Logos

When customizing the UI, you can also customize logos. Customizing logos involves a different set of files, so there is a separate procedure for customizing them.

To customize logos:

1. Open the *UIM_Home*/**lib/comms-platform-webapp.war** file and extract the **WEB-INF/lib/comms-platform-ui.jar** file to a local directory, such as *tempDir*.
2. Open the *tempDir*/**comms-platform-ui.jar/images** directory and add your custom logo file.

   Note: Custom logo files are images, which are typically GIF, JPG, or PNG file types.

3. Open the *tempDir*/**comms-platform-ui.jar/oracle/communications/platform/templates/CommsUIShell.jspx** file, and modify the file as follows:
   a. Locate the text:

```
<af:image id="oracleImage"
    source="/afr/logo-oracle-red.png"
    clientComponent="true" shortDesc="Oracle"/>
```

   b. Change the text that defines the source to:

```
<af:image id="oracleImage"
    source="/images/customLogoFileName"
    clientComponent="true" shortDesc="Oracle"/>
```

   where *customLogoFileName* is the name of your custom logo file that you previously added to the *tempDir*/**comms-platform-ui.jar/images** directory. The *customLogoFileName* includes the file type extension, such as **.gif**, **.jpg**, or **.png**.
4. Save and close the *tempDir*/**comms-platform-ui.jar** file.
5. Repackage the WAR file by doing the following:
   a. Open the *UIM_Home*/**lib/comms-platform-webapp.war/WEB-INF/lib** directory.

b. Replace the **comms-platform-ui.jar** file with the *tempDir*/**comms-platform-ui.jar** file that contains your customizations.

c. Save and close the *UIM_Home*/**lib/comms-platform-webapp.war** file.

6. Log in to the WebLogic Server Administration Console.
7. Stop the UIM application:
   a. In the left panel, under Domain Structure, click the **Deployments** link.

      The Summary of Deployments page appears.

   b. Select the check box for **oracle.communications.inventory**, and click **Stop**.
   c. Choose **Force Stop Now**, and click **Yes**.

      The UIM application stops.
8. To delete the UI library, on the Summary of Deployments page, select the check box for **oracle.communications.platform.cui.webapp**, and click **Delete**.

   The library is deleted.
9. Open a command line.
10. Navigate to the *UIM_Home*/**servers/***serverName*/**tmp/_WL_user** directory:

    ```
    cd UIM_Home/servers/serverName/tmp/_WL_user
    ```

11. Delete the **oracle.communications.inventory** directory.

    ```
    rm -rf oracle.communications.inventory
    ```

    Note: If working in a clustered environment, delete the **oracle.communications.inventory** directory from the **tmp/_WL_user** directory for each of the servers.

12. Close the command line and return to the WebLogic Server Administration Console, Summary of Deployments page.
13. To install the UI library, click **Install**, and select **comms-platform-ui.jar** located in *UIM_Home*/**lib**.

    The library is installed.
14. Select the check box for **oracle.communications.inventory**, and click **Update**.

    This redeploys the **inventory.ear** file and starts the UIM application.

Testing User Interface Customizations

You can test your UIM UI customizations by running UIM and navigating to the customized pages or new pages to validate that the customizations are working correctly. If customizations included changes or deletions to existing files, regression testing is required to ensure the customizations did not break existing UIM UI functionality.

12

Localizing UIM

This chapter provides information on localizing the Oracle Communications Unified Inventory Management (UIM) user interface (UI), and on localizing the UIM Help. Localization is the process of translating a UI or Help system from the original language in which it was written into a different language for use in a specific country or region. For example, the UIM UI and UIM Help are written in English. If your company is based in France and you purchase UIM, you may want to localize UIM to display the UI and Help in French.

Localizing UIM involves modifying a specific set of files that UIM uses to display text in the UI and in the Help.

> **Note:** Before localizing your UIM environment, you must identify a strategy for maintaining future localizations. Oracle does not provide a file that lists the details of what changed between releases.

Setting the Language Preference in Internet Explorer

For a localized version of UIM to display correctly in Internet Explorer, users need to configure language preferences.

To configure language preferences in Internet Explorer:

1. From the **Tools** menu, select **Internet Options**.

   The Internet Options window appears.

2. Click **Languages**.

   The Language Preference window appears.

3. The language you plan to use must display at the top of the list to have priority.

   If the language you plan to use is listed:

   a. Select the language.

   b. Click **Move Up** to move the language you plan to use to the top of the list.

   If the language you plan to use is not listed:

   a. Click **Add**.

      The Add Language window appears.

   b. Select a language.

   c. Click **OK**.

      The Language Preference window returns.

d. Select the language you have added, and click **Move Up** to move it to the top of the list.

4. Click **OK**.

Determining the Locale ID

A locale ID is a standardized ID that represents a language and region in which the language is spoken. For example, **fr-CA** is the locale ID for French spoken in Canada, and **es-MX** is the locale ID for Spanish spoken in Mexico.

Localizing UIM involves copying and renaming existing files to include a locale ID. The renamed files that include a locale ID become the translated version of the original files.

To determine the locale ID in Internet Explorer:

1. From **Tools** menu, then select **Internet Options**.

   The Internet Options window appears.

2. Click **Languages**.

   The Language Preference window appears.

3. Click **Add**.

   The Add Language window appears.

   Languages are listed alphabetically. Several languages are spoken in more than one country, so the locale ID reflects the language and the country in which the language is spoken.

4. Locate the language to which you are localizing and note the locale ID.

5. Close the Add Language, Language Preference, and Internet Option windows.

Localizing UIM

Localizing the UIM UI involves working with a UIM-provided cartridge that you import into Oracle Communications Design Studio, modify, and deploy. Design Studio also provides various editors, such as an XML editor and an HTML editor, that you can use to translate files for localization.

The following sections describe localizing UIM:

- About the UI-Specific Files
- Localizing the UI-Specific Files
- Deploying the Cartridge Containing the Localized Files
- Testing the UIM UI Localization

About the UI-Specific Files

The UI-specific files are a set of **.xlf** and **.properties** files that contain localizable text strings that define labels and messages. You modify the text string within these files to localize UIM.

- **.xlf** files

The UIM UI was written using Application Development Framework (ADF). ADF-specific files use the **.xlf** file extension. XLF files contain localizable text strings for labels that display in the UI.

- **.properties** files

  The UIM UI calls UIM API methods, which may result in an information, warning, or error message displaying in the UI. Properties files contain localizable text strings for API messages that display in the UI.

Localizing the UI-Specific Files

Localizing the UI is accomplished by modifying the text strings in XLF and properties files that display in the UI.

To localize the UI-specific files, perform the work described in the following sections:

- Importing the Localization Archive File into Design Studio
- Locating the UI-Specific Files within the Project
- Copying and Renaming the UI-Specific Files
- Editing the UI-Specific Files

Importing the Localization Archive File into Design Studio

Note: Within Design Studio, you must be in the Studio Design perspective Studio Projects view.

The *UIM_Home*/**cartridges/sample/ora_uim_localization_reference_cartproj.zip** file contains an Inventory project with all of the UI-specific files that you can import into Design Studio to localize.

For instructions on how to import projects using archive files, see the Design Studio Help.

Locating the UI-Specific Files within the Project

Note: Within Design Studio, you must be in the Java perspective Package Explorer view.

The localization archive file that you imported into Design Studio contains the **ora_uim_localization_reference** project. The UI-specific files are contained within the project.

XLF Files

The UI-specific XLF files are located in the **ora_uim_localization_reference** project, within the **model/content/inventory.ear/inv.war/WEB-INF/classes/oracle/communications** directory. The **communications** directory contains the following subdirectories, which contain the UI-specific XLF files:

- **inventory/ui/common/bundle/InventoryUIBundle.xlf**
- **inventory/ui/framework/bundle/InventoryOHWBundle.xlf**

- **platform/ui/CommsUIShell.xlf**
- **platform/ui/Preferences.xlf**

Properties Files

The UI-specific properties files are located in the **ora_uim_localization_reference** project, within the **model/content/product_home/config/resources/logging** directory.

Copying and Renaming the UI-Specific Files

Copying and renaming the UI-specific files ensures that the default file is always in place to use for display if needed. Adding the locale to the file name differentiates your localized files from the default files, which simplifies upgrades. If files are edited for localization without being renamed to reflect the locale, all localization efforts are lost when you upgrade because the files are overwritten.

To copy and rename the files within the Design Studio Java perspective Package Explorer view:

1. Right-click on the file and select **Copy**.
2. Right-click on the parent directory of the copied file and select **Paste**.

   The Name Conflict dialog box appears.
3. Modify the file name to include the appropriate locale ID.

   For example, rename **InventoryUIBundle.xlf** to **InventoryUIBundle_fr_ca.xlf** and rename **equipment.properties** to **equipment_fr_ca.properties** for French spoken in Canada.

   See "Determining the Locale ID" for more information.

   Note: On the Add Language window shown in "Determining the Locale ID", the locale ID is separated by a dash. When renaming the XLF and properties files, use an underscore in place of the dash.
4. Click **OK**.

   Note: If you copy and paste the file, and then try to rename it, the **Rename** menu option is not available when right-clicking on the file in the Java perspective. You can, however, copy and paste the file and rename by selecting **File** from the menu, and then selecting **Rename**.

Editing the UI-Specific Files

To edit the UI-specific files, perform the work described in the following sections:

- Editing the XLF Files
- Editing the Properties Files

Editing the XLF Files

To edit the XLF files within Design Studio:

1. Open the Java perspective.
2. Open the Package Explorer view.
3. Within the imported project, locate the XLF files.

See "Locating the UI-Specific Files within the Project" for more information.

4. Right-click on the file and select **Open With**, then select **Text Editor**.

> **Caution:** If you double-click on the file, Design Studio may open the file for editing outside of Design Studio.

5. Edit the value of the <source> elements, which define text that displays in the UI.

   Example 12–1 is an excerpt from the **InventoryUIBundle.xlf** file that shows numerous <source> elements. Edit only the value of the <source> elements: for example **UIM Home Page**, **Inventory**, **Home**, and **Products**.

Example 12–1 InventoryUIBundle.xlf

```
<trans-unit id="LANDING_PAGE_TITLE">
    <source>UIM Home Page</source>
    <target/>
</trans-unit>
<trans-unit id="MENU_INVENTORY">
    <source>Inventory</source>
    <target/>
</trans-unit>
<trans-unit id="MENU_HOME">
    <source>Home</source>
    <target/>
</trans-unit>
<trans-unit id="MENU_PRODUCT">
    <source>Products</source>
    <target/>
</trans-unit>
```

> **Note:** The **Preferences.xlf** file defines a date format. If you want to localize the date format, see "XLF Files" for more information.

Editing the Properties Files

To edit the properties files within Design Studio:

1. Open the Java perspective.
2. Open the Package Explorer view.
3. Within the imported project, locate the properties files.

   See "Locating the UI-Specific Files within the Project" for more information.

4. Right-click on the file and select **Open With**, then select **Text Editor**.

> **Caution:** If you double-click on the file, Design Studio may open the file for editing outside of Design Studio.

5. Edit the text strings that define API messages that display in the UI.

   Example 12–2 is an excerpt from the **party.properties** file that shows two messages. Each message is defined by two lines: the first line defines the message ID, and the second line defines the message text that displays in the UI. Edit only

the message text: for example, **Party Id {0} already exists** and **The party with Id {0} was successfully deleted**.

Example 12–2 also shows that messages are not necessarily error messages; the partyDeleted message in is an informational message.

Example 12–2 party.properties

```
party.alreadyExists.id=230002
party.alreadyExists=Party Id {0} already exists.
party.partyDeleted.id=230009
party.partyDeleted=The party with Id {0} was successfully deleted.
```

For languages that have a heavy usage of single quotes like the French language, you use two single quotes in order to have the single quote show in the message. Example 12–3 has an example of using the two single quotes. This example results in the following message:

- En attente d'attribution

Example 12–3 Languages with Use of Single Quotes Need Two Single Quotes

```
status.PENDING_ASSIGN=En attente d''attribution
```

Note: UIM uses Java's MessageFormat class, which use a single quote to represent a pattern within the string. To have a single quote visible, you must use two single quotes to represent a single quote in the string.

Most of the entries in the properties files are informational and error messages. Example 12–4 is a date format example where the value exists with a single line definition. This date format property is defined in the **system.properties** file.

Example 12–4 system.properties Entry with a Date Format

```
system.dateFormat=MM/dd/yyyy
```

For the single-line property values that are not messages, the symbols must contain English abbreviations. For instance, French abbreviations for the date values

```
MM/jj/aaaa
```

create an error situation. Example 12–5 shows a property setting to accommodate a modified date display.

Example 12–5 system.properties Entry with a Modified Date Format

```
system.dateFormat=dd/MM/yyyy
```

Deploying the Cartridge Containing the Localized Files

Localized files are modified as part of a project. After the modifications are complete, build the project to create the cartridge that can be deployed into UIM. Every cartridge should be cleaned and rebuilt prior to deploying.

See *UIM Cartridge Guide* for information about deploying cartridges and cartridge packs.

Note: When a cartridge containing localizable XLF files is deployed into UIM, the **inventory.ear** file is automatically redeployed, resulting in the localization changes being applied to the UI.

Testing the UIM UI Localization

You can test your UIM UI localization by running UIM and navigating from page to page to validate that the pages are displaying the localized text.

Localizing UIM Help

The following sections describe localizing UIM Help:

About UIM Help

The UIM Help uses Oracle Help for the Web. Oracle Help is a browser-based Help system that runs as a Web application based on a Java servlet. You do not need specialized knowledge of Oracle Help to localize UIM Help; you can use the information in this chapter, supplemented by the Oracle Help documentation. See *Oracle Fusion Middleware Developing Help Systems with Oracle Help* for more information:

http://docs.oracle.com/middleware/1213/jdev/develop-help/toc.htm

About the Oracle Help Configuration File

The Oracle Help configuration file, **ohwconfig.xml**, is located in the *UIM_Home*/**app/inventory.ear/inv.war/WEB-INF/help** directory. The **ohwconfig.xml** file contains references to each Help system deployed into an application. Upon installation, **ohwconfig.xml** references the default UIM Help system (English) deployed into UIM. This file requires configuration for localization.

About the UIM Help Files

The UIM Help files are located in the *UIM_Home*/**app/inventory.ear/inv.war/WEB-INF/help/helpsets/uimoh_help.jar** file, which contains the following Help files:

- **.htm files**: Each HTML file is a separate Help topic. The text in all of the HTML files requires translation.
- **uimoh.hs**: This file describes the Help system. When UIM Help is initiated through the UIM user interface, **uimoh.hs** is the starting point. This file does not require translation.
- **toc.xml**: This file defines the Table of Contents that appears in the left pane of the Oracle Help window. The text in this file requires translation.
- **map.xml**: This file associates Help IDs with the HTML file names. The **toc.xml** file uses the IDs to link entries to Help topics. This file does not require translation.

- **search.idx**: This file is used when you perform a text search of the Help content. The file defines a search index that searches the Help content in the HTML files. After the HTML files are translated, the search index must be regenerated using the Java-based Help Indexer. For more information, see "Regenerating the Search Index File".
- **target.db**: This file contains cross-reference information used for navigating between Help topic headings. This file does not require translation.
- **dcommon/html/cpyr.htm**: This file defines the Help copyright page and requires translation. (The **dcommon** directory contains standard Oracle support files, including a CSS file, several graphics files, and the Help copyright page, but only the Help copyright page requires translation.)

Localizing the UIM Help Files

To localize UIM Help, perform the work described in the following sections:

- Extracting the Help Files
- Translating the Help Files
- Regenerating the Search Index File
- Creating the Localized Help JAR File
- Configuring the Oracle Help File

Extracting the Help Files

Use the default Help files installed with UIM as the starting point for your localization.

To extract the Help files:

1. Copy the *UIM_Home*/**app/inventory.ear/inv.war/WEB-INF/help/helpsets/uimoh_help.jar** file to *tempDir*, where *tempDir* is a local directory.
2. Open the *tempDir*/**uimoh_help.jar** file.
3. Extract all the objects in the **uimoh_help.jar** file into *tempDir*.
4. Click the **File** column heading in *tempDir*, which sorts the objects by file type.

   You should see the following directories and files in *tempDir*:

   - **dcommmon** directory
   - **img** directory
   - **META-INF** directory
   - **target.db**
   - **uimoh_help.jar**
   - **uimoh.hs**
   - numerous **.htm** files
   - **search.idx**
   - **map.xml**
   - **toc.xml**

Translating the Help Files

To translate the Help files, perform the work described in the following sections:

Translating the Copyright Page

To translate the copyright page:

1. Navigate to the *tempDir*/**dcommon/html** directory.
2. Open the **cpyr.htm** file.
3. Translate the content of the <title>, <h1> through <h6>, and <p> elements to the local language.

   For example, translate the bolded content in Example 12–6:

Example 12–6 Excerpt from cpyr.htm

```
<title>Oracle Legal Notices</title>
<link rel="stylesheet" href="../css/blafdoc.css" type="text/css" />
</head>
<body>
<h1>Oracle Legal Notices</h1>

<h2>Copyright Notice</h2>
<p>Copyright &copy; 1994-2012, Oracle and/or its affiliates. All rights
reserved.</p>
```

Translating the Help Topics

To translate the Help topics:

1. Navigate to the *tempDir* directory.

   The Help topics text is defined in the numerous **.htm** files within this directory. Each **.htm** file must be translated.
2. Open an **.htm** file.
3. Translate the content of the <title>, <h1> through <h6>, <p>, and <td> elements to the local language.

   For example, translate the bolded content in Example 12–7. Elements that are not text, such as the HTML tags themselves, should not be changed.

Example 12–7 Excerpt from tel_nbr_info_work_area.htm

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta name="OAC_IGNORE_SKIP_NAV" content="true" />
<meta http-equiv="Content-Type" content="text/html; charset=us-ascii" />
<meta http-equiv="Content-Style-Type" content="text/css" />
<meta http-equiv="Content-Script-Type" content="text/javascript" />
<title>Telephone Number - Information Work Area</title>
<meta name="generator" content="Oracle DARB XHTML Converter (Mode = ohj/ohw) -
Version 5.1.2 Build 073" />
<meta name="date" content="2012-09-17T22:25:55Z" />
<meta name="robots" content="noarchive" />
<meta name="doctitle" content="Telephone Number - Information Work Area" />
<meta name="relnum" content="Release 7" />
```

```
<meta name="partnum" content="E36042-0" />
<meta name="topic-id" content="telephoneInfo" />
<link rel="copyright" href="./dcommon/html/cpyr.htm" title="Copyright"
type="text/html" />
<link rel="stylesheet" href="./dcommon/css/blafdoc.css" title="Oracle BLAFDoc"
type="text/css" />
<link rel="contents" href="toc.htm" title="Contents" type="text/html" />
</head>
<body>
<p><a id="CHDCJEIG" name="CHDCJEIG"></a><a id="telephoneInfo"
name="telephoneInfo"></a></p>
<div class="sect1"><!-- infolevel="all" infotype="General" -->
<h1>Telephone Number - Information Work Area</h1>
<p>You use the <span class="gui-object-title">Telephone Number -
Information</span> work area to edit the information that appears in the Summary
work area <span class="gui-object-title">Information</span> panel. Some data
elements, such as the ID, cannot be changed after the entity is created.</p>
<div align="center">
<div class="inftblnote"><br />
<table class="Note oac_no_warn" summary="" cellpadding="3" cellspacing="0">
<tbody>
<tr>
<td align="left">
<p class="notep1">Note:</p>
The fields that appear in this work area are determined by the entity
specification definition used to create the entity. The specification is created
in Design Studio. The fields defined below for this entity are common among most
specifications.</td>
</tr>
```

4. Repeat steps 2 and 3 for each **.htm** file in the *tempDir* directory.

Translating the Table of Contents

To translate the Table of Contents:

1. Navigate to the *tempDir* directory.
2. Open the **toc.xml** file.

   Each item in the Table of Contents is defined by a <tocItem> element.
3. Translate the content of each <tocItem> to the local language.

   For example, translate the bolded content of the text attribute in Example 12–8. Do not change the content of the target attribute.

Example 12–8 Excerpt from toc.xml

```
<tocitem target="uim_help_interface.htm" text="Getting Started with Unified
Inventory Management">
```

Note: Oracle Help automatically translates the Help window menu options; field names; and informational, warning, and error messages. The translation is based on the locale defined in the **ohwconfig.xml** file.

For example, if the only language preference specified is English, and the **ohwconfig.xml** file defines a single locale of French, Oracle Help translates the Help window menu options, field names, and messages to French.

Oracle recommends that the language preference with the highest priority be the same language defined as the locale in the **ohwconfig.xml** file.

Regenerating the Search Index File

After translating the Help files, regenerate the search index file to reflect the content of the translated files. This is accomplished using Oracle Help Indexer.

Note: Using Oracle Help Indexer requires that you have Java installed.

To install Oracle Help Indexer:

1. Go to the Oracle Technology Network website:

   http://www.oracle.com/technetwork/topics/utilsoft-085729.html

2. Download the **help-indexer.jar** file to *tempDir*, where *tempDir* is a local directory.

To regenerate the search index file using Oracle Help Indexer:

1. Open a Windows command prompt.
2. Change the directory to *tempDir* by entering the following command:

   cd *tempDir*

3. Enter the following command, which creates a new **search.idx** file, and overwrites the existing **search.idx** file:

   java -mx64m -classpath *pathToJarFile*/**help-indexer.jar oracle.help.tools.index.Indexer -l=***locale* **-e=***charSet pathToHelpFiles* **search.idx**

   where:

   - *pathToJarFile* is the directory path to the **help-indexer.jar** file.
   - *locale* is the standardized locale ID that represents the localized language. See "Determining the Locale ID" for more information.

     If you do not specify a locale, the system's default locale is used.
   - *charSet* is the Java-supported character set encoding.
   - *pathToHelpFiles* is the directory path to the Help files.

   For example:

   ```
   java -mx64m -classpath C:\tempDir\help-indexer.jar
   oracle.help.tools.index.Indexer -l=fr_CA -e=8859_1 C:\tempDir search.idx
   ```

See *Oracle Fusion Middleware Developing Help Systems with Oracle Help* for more information on using the text search indexer:

http://docs.oracle.com/middleware/1213/jdev/develop-help/toc.htm

Creating the Localized Help JAR File

After translating the Help files and regenerating the search index, create a new JAR file containing the localized Help files.

To create the new JAR file:

1. Navigate to the *tempDir* directory.

   The *tempDir* directory contains the **uimoh_help.jar** file, the translated Help files, and the regenerated search index file.

2. Copy the **uimoh_help.jar** file to *tempDir* to create a second copy of the **uimoh_help.jar** file in *tempDir*.

3. Select the copied version of the **uimoh_help.jar** file and rename it **uimoh_help.jar**_*locale*.**jar**, where *locale* is the standardized ID that represents a language and region in which the language is spoken. For example, **fr-CA** is the locale for French spoken in Canada, and **es-MX** is the locale for Spanish spoken in Mexico.

   For more information, see "Determining the Locale ID".

   Note: On the Add Language window shown in "Determining the Locale ID", the locale ID is separated by a dash. When renaming the JAR file, use an underscore in place of the dash.

4. Open the **uimoh_help**_*locale*.**jar** file.

5. Delete all of the objects in the JAR file.

6. Add the localized Help files to the **uimoh_help**_*locale*.**jar** file. (This includes all of the directories and all of the files in *tempDir*, with the exception of **uimoh_help.jar** and **uimoh_help**_*locale*.**jar**.

7. Save and close the **uimoh_help**_*locale*.**jar** file.

You can verify that you included all of the directories and files by checking the number of objects in the **uimoh_help.jar** file and in the **uimoh_help**_*locale*.**jar** file; the two JAR files should contain the same number of objects. To determine the number of objects in each JAR file, select all of the objects in each JAR file; this provides a count of all objects selected.

Configuring the Oracle Help File

After translating the Help files, regenerating the search index, and creating a localized Help JAR file, configure the **ohwconfig.xml** file to reflect the localized Help JAR file.

To configure the **ohwconfig.xml** file:

1. Open the *UIM_Home*/**app/inventory.ear/inventory.war/WEB-INF/help/ohwconfig.xml** file.

   The file defines the default Help system (English):

```
<locales>
    <!-- English: -->
    <locale language="en">
        <books>
```

```
            <helpSet id="uimoh_help"
                     jar="/helpsets/uimoh_help.jar"
                     location="uimoh.hs"/>
        </books>
    </locale>
</locales>
```

2. Update the <locale> element to reflect the localized Help system:

```
<locales>
    <!-- French Canadian: -->
    <locale language="fr">
        <books>
            <helpSet id="uimoh_help_fr_ca"
                     jar="/helpsets/uimoh_help_fr_ca.jar"
                     location="uimoh.hs"/>
        </books>
    </locale>
</locales>
```

You do not need to change the location attribute value, which is the name of the file that resides in the specified JAR file.

About Multiple Locales

Oracle Help can support multiple locales. For multiple locales, each localized Help system is configured with a <locale> element in the **ohwconfig.xml** file. For example, the following results in both French and Spanish Help systems being available in UIM upon redeployment:

```
<locales>
    <!-- French: -->
    <locale language="fr">
        <books>
            <helpSet id="uimoh_help_fr_ca"
                     jar="/helpsets/uim_oh_help_fr_ca.jar"
                     location="uimoh.hs"/>
        </books>
    </locale>
</locales>
<locales>
    <!-- Spanish: -->
    <locale language="es">
        <books>
            <helpSet id="uimoh_help_es_mx"
                     jar="/helpsets/uimoh_help_es_mx.jar"
                     location="uimoh.hs"/>
        </books>
    </locale>
</locales>
<parameters>
    <combineBooks>false</combineBooks>
    <useLabelInfo>true</useLabelInfo>
    <cacheSize>3</cacheSize>
</parameters>
```

When multiple locales are defined, the language preference for all locales must be set. If not set, only the first locale defined in the **ohwconfig.xml** file displays in UIM Help. See "Setting the Language Preference in Internet Explorer" for more information.

When multiple locales are defined, the <parameters> element configuration values are applied:

- <combineBooks>

  To merge Help systems, set the value of <combineBooks> to **true**. The Help navigational views behave as a single, integrated Help system.

  To use separate Help systems, set the value of <combineBooks> to **false**. The separate Help navigational views are accessed based on the language preference with the higher priority.

  Regardless of the <combineBooks> value, each locale that is defined in the **ohwconfig.xml** file must be specified as a language preference. See "Setting the Language Preference in Internet Explorer" for more information.

  Note: Oracle Help automatically translates the Help window menu options; field names; and informational, warning, and error messages. The translation is based on the first locale defined in the **ohwconfig.xml** file.

  For example, if the only language preference specified is English, and the **ohwconfig.xml** file defines the locales of French and Spanish, Oracle Help translates the Help window menu options, field names, and messages to French.

  However, when multiple locales are defined, the language preference for all locales must be specified. Otherwise, only the first locale defined in the **ohwconfig.xml** file displays in UIM Help. So, when the language preferences are set, Oracle Help translates the Help window menu options, field names, and messages to the language preference with the highest priority.

- <useLabelInfo>

  If <useLabelInfo> is set to **true**, author-defined labels are used for the navigators of merged Help systems.

  If <useLabelInfo> is set to **false**, default labels such as Contents, Index, and Search are used for the navigators of merged Help systems.

- <cacheSize>

  <cacheSize> indicates the number Help systems kept in memory at one time. The default value is 3.

For more information on the web configuration file, see *Oracle Fusion Middleware Developing Help Systems with Oracle Help* which you can find here:

`http://docs.oracle.com/middleware/1213/jdev/develop-help/toc.htm`

Deploying the Localized Help System

The default Help system is deployed when you deploy the **inventory.ear** file.

To deploy the localized Help system:

1. Repackage the *UIM_Home*/**app/inventory.ear** file to include the localized Help files by doing the following:

a. Delete the *UIM_Home*/**app/inventory.ear/inv.war/WEB-INF/help/helpsets/uimoh_help.jar** file.

b. Copy the *tempDir*/**uimoh_help**_*locale*.**jar** file to the *UIM_Home*/**app/inventory.ear/inv.war/WEB-INF/help/helpsets** directory.

Note: If your UIM Help is supporting multiple locales, each JAR file defined by each <locale> element in the **ohwconfig.xml** file must be present in the *UIM_Home*/**app/inventory.ear/inv.war/WEB-INF/help/helpsets** directory.

---

2. Deploy the repackaged **inventory.ear** file.

   For instructions on how to deploy the **inventory.ear** file, see *UIM System Administrator's Guide*.

Testing the UIM Help Localization

After you deploy the localized Help system, test your UIM environment to verify that the localized Help system is working correctly.

In UIM, open the Help and do the following:

- Navigate to several topics from links in the Table of Contents to ensure that the correct topics appear and display correctly.
- Test several links within Help topics to ensure they are working.
- Search for several terms and verify that you get the expected results.
- If testing multiple locales that function as a single Help system, verify translations for all locales.
- If testing multiple locales that function as separate Help systems, change the language preference priority to verify translations for each locale.

13

Optimizing Concurrent Resource Allocation

This chapter provides information about optimizing Oracle Communications Unified Inventory Management (UIM) concurrent resource allocation. The information describes the use of row locking to optimize concurrent resource allocation for consumable entities, and how you can extend additional entities to use row locking in your UIM environment. For example, if your UIM environment heavily uses party entities, you can extend UIM to have party entities use row locking to optimize concurrent resource allocation of party entities.

About Concurrent Resource Allocation

Concurrent resource allocation occurs when multiple clients simultaneously select the same resource from the same resource pool and then try to reserve or assign that resource.

The inherent problem with concurrent resource allocation is that only one of the clients can assign the resource; the other clients fail and must re-try the operation. When the client re-tries the operation, the resources are queried a second time, and again only one client succeeds. As this scenario unfolds, new clients may query the same resource pool. Multiple clients simultaneously selecting from the same resource pool, failing to reserve or assign a resource, and re-trying the operation can continue indefinitely, causing performance issues.

For example, when two clients simultaneously search by ZIP code for an available telephone number to assign, the search results can return the same telephone number to both clients. If both clients try to assign the same telephone number, one client succeeds, and one client fails. The client that fails must re-run the search to get a new set of available telephone numbers from which to assign.

In most UIM environments, telephone number entities are heavily used. Due to this, telephone number entities use row locking to optimize concurrent resource allocation for assigning and reserving telephone numbers. See "About Row Locking" for more information.

About Row Locking

This section builds upon the information presented in Chapter 4, "Extending the Data Model".

Row locking is a capability, which is a design pattern that is applied to an entity. A capability is declared in the metadata using tags, and results in the generation of attributes and related entities that are not explicitly defined in the ***-entities.xml** or ***-types.xsd** metadata files. See "About Entity Capabilities" and "Understanding Entity Capability Definitions" for more information.

The row locking capability is declared for a UIM entity in the metadata files using the **RowLockEnabled** tag. The row locking capability declaration results in the generation of the *EntityName*RowLock entity, where *EntityName* is the name of an entity. Row locking is enabled and used through UIM functionality, which processes entities with row locking capabilities differently than entities without row locking capabilities.

The following example shows the **RowLockEnabled** tag as declared for the TelephoneNumber entity:

```
<rowLockEnabled prefix="TelephoneNumber"/>
```

In this example, the **RowLockEnabled** tag as declared for the TelephoneNumber entity results in the generation of the TelephoneNumberRowLock entity.

Upon UIM installation, all consumable entities have row locking capability and use row locking through UIM. For example, TelephoneNumber, Equipment, IPAddress, and so forth. (Being a consumable entity is also a capability that is declared in the metadata files.)

You can statically extend additional entities to declare the row locking capability through the metadata files, and enable and use row locking on these entities through custom code.

When row locking is in place for an entity and an entity finder API called, lock policy details may be provided to the entity finder API. If lock policy details are provided, selected entity rows are locked. If no lock policy details are provided, selected entities are returned to the client with no row locks on them.

For example, when two clients simultaneously search by ZIP code for an available telephone number to reserve or assign, the search results do not return the same telephone number to both clients because each telephone number entity that meets the search criteria is locked prior to returning the search results. The locking is achieved by updating the TelephoneNumberRowLock table with the telephone numbers returned in the search. If a telephone number is already present in the TelephoneNumberRowLock table, the telephone number is filtered from the search results prior to returning the search results to the client. When telephone number locks are released, the telephone numbers are removed from the TelephoneNumberRowLock table.

Row locking removes the potential of the same telephone number being returned to both clients. If the same number is not returned to both clients, the same number cannot be selected by both clients, which eliminates the possibility of one client failing to assign the telephone number.

Row locking is applied by setting the appropriate LockPolicy values during the search. See "About the LockPolicy Object" for more information.

Note: Implementing row locking as described here achieves the same result as database row-locking. However, traditional database row-locking is not used.

Understanding How Row Locking Works

This section uses the TelephoneNumber entity as an example to provide an understanding of how row locking works.

In UIM, TelephoneNumber entities are used for resource reservation and resource assignment. Upon UIM installation, the TelephoneNumber entity declares the row locking capability through the entity definition in the metadata. UIM search logic

performs row-locking operations based on the lock policy details provided to the search.

For example, when the entity finder API is called to find telephone numbers for resource reservation, the entity finder API logic row-locks the telephone numbers.If the row-lock operation is successful, telephone numbers are reserved and the row-locks are released. If the row-lock operation fails (row-locks for all the telephone numbers are not obtained), an error message displays and no attempt is made to reserve the telephone numbers.

Similarly, when the entity finder API is called to find telephone numbers for resource assignment, the entity finder API logic row-locks the telephone numbers. If the row-lock operation is successful, telephone numbers are assigned to a configuration item and the row-locks are released. If the row-lock operation fails (row-locks for all the telephone numbers are not obtained), an error message displays and no attempt is made to assign the telephone numbers.

The RowLock capability is statically declared for an entity in the metadata. Additionally, a LockPolicy object is used in the entity search criteria. When searching for entities, such as telephone numbers or logical devices, the entities returned in the search results are locked based on the details specified in the LockPolicy object. In this manner, concurrent resource allocation attempts return different sets of entities.

About Releasing Locked Rows

Locked rows are released in the following ways:

- When an entity finder API is called with a LockPolicy, the entity finder API logic calls a method at the end of the operation that releases the locks.
- A timer listener automatically releases row-locked entities at specified intervals.
- A database administrator can manually release row-locked entities.

About the LockPolicy Object

The LockPolicy object defines the attributes listed in Table 13–1. LockPolicy attribute values are set as part of the search criteria, which is passed to the entity finder API, and which affects the search behavior.

Table 13–1 LockPolicy Attributes

Attribute	Data Type	Default Value
numberOfResources	long	0
expirationTimeStamp	Date or int	Value specified for the lockPolicy.defaultRowLockExpirationDuration property in the **system-config.properties** file.
filterExistingLocks	boolean	false

numberOfResources

The numberOfResources value tells the entity finder API the number of entity rows to lock. For example, when a value of zero is specified, no entity rows are locked; when a value greater than zero is specified, the number of entity rows specified by the value are locked.

If the search criteria specifies a range, and the numberOfResources value is greater than zero, the entity finder API ignores the range and returns the number of entities specified by the numberOfResources value. For example, if the range specifies 0-20

and the numberOfResources values specifies 50, 50 entities are returned. This occurs because the find-by-range feature is disabled when row locking is used when calling and entity finder API.

expirationTimeStamp

The expirationTimeStamp value is used by the entity finder API when creating the locked row. The expirationTimeStamp value can be set as a date or as a duration. When the value is a date, the value is used to set the lock expiration date and time. When the value is a duration, the value is added to the current timestamp to calculate the lock expiration date and time.

filterExistingLocks

The filterExistingLocks value indicates whether or not the entity finder API filters out existing locked entities from the search result. When the value is true, existing locks are not included in the search results. When the value is false, existing locks are included in the search results. When a client search results are for view-only purposes, the value must be false.

Example LockPolicy Attribute Combinations

The entity finder API finds entities based on the specified search criteria and lock policy. Table 13–2 summarizes the different LockPolicy attribute value combinations and the affect each has on the entity finder API search results.

When numberOfResources is zero, the expirationTimeStamp value is not applicable because if no locks are applied, there is no need to set when the locks expire. When numberOfResources is greater than zero, the expirationTimeStamp value does not affect the outcome. As a result, the expirationTimeStamp attribute is not included in the table.

Table 13–2 LockPolicy Attribute Combination Outcomes

numberOf Resources	filterExisting Locks	Entity Finder API Search Results
0	true	The search results exclude row-locked entities. From the search results, no entities are locked. The search results are returned to the client.
0	false	The search results include row-locked entities. From the search results, no entities are locked. The search results are returned to the client.
n	true	The search results exclude row-locked entities. From the search results, *n* entities are locked, time stamped, and returned to the client.
n	false	The search results include row-locked entities. From the results, *n* entities are locked, time stamped, and returned to the client.

About the Lock Strategies

The LockStrategy object utilizes the LockPolicy object when a lock of resources is requested. UIM provides the following lock strategies:

- The *Random Range Locking Strategy* selects the range of resources to be locked randomly. The start range is determined using a thread-based local random value. This allows the concurrent threads to look at different resources so that collisions are avoided.

- The *Extended Range Locking Strategy* executes the concurrent threads with the same range; however, the range is widened by a multiplier so that collisions are avoided. For example, if the client code requires 10 resources, this strategy looks at 1-100 resources assuming the range multiplier is 10.
- The *No Range Locking Strategy* is a strategy where no range is applied. This strategy continues until the required number of locks are obtained. You can use this in cases where the ordering or resources to be consumed is critical to the scenario.

You choose the lock strategy by specifying the value in the locking policy properties file named **locking-policy.properties**. You can also define a custom lock strategy. The valid values are:

- **RandomRangeLocking**
- **ExtendedRangeLocking**
- **NoRangeLocking**
- **Custom**

For example:

```
lockpolicy.processing.strategy=RandomRangeLocking
```

Table 13–3 lists the locking properties.

Table 13–3 Locking Property Settings

Property	Default Value	Description
lockpolicy.processing.strategy	NoRangeLocking	This property defines the desired locking strategy. The valid UIM strategy values are **RandomRangeLocking**, **ExtendedRangeLocking**, and **NoRangeLocking**. The value of **Custom** is also valid. If **Custom** is given then you must provide the class name in the **lockpolicy.custom.strategy.class** property.
lockpolicy.custom.strategy.class	n/a	This property defines the custom Java class name of the strategy. You only give this property a value if the **lockpolicy.processing.strategy** property is set to the **Custom**.

In addition to setting the locking strategy in the property file, the following method can be used for a particular scenario.

```
lockpolicy.setLockingStrategy();
```

Use this method only if you need a locking strategy other than the one defined in the property file.

Extending UIM Entities to Use Row Locking

Extending UIM entities to use row locking involves:

- Statically Extending the Data Model
- Enabling Row Locking
- Using Row Locking with Entity Finder APIs

Statically Extending the Data Model

This section builds upon the information presented in Chapter 4, "Extending the Data Model".

To statically extend the data model.

1. Open Oracle Communications Design Studio.
2. Import the *UIM_Home*/**cartridges/tools/ora_uim_entity_sdk_cartproj.zip** file.
3. Create a new **uim-*-entities.xml** file in the **ora_uim_entity_sdk/src** directory.
4. Open the existing ***-entities.xml** file that contains the entity definition you plan to extend.
5. Copy and paste the entity definition you plan to extend from the existing file to your new file.
6. Add the **EntityRowLock** tag to the entity definition.

   Example 13–1 is an excerpt from the **uim-number-entities.xml** metadata file that shows the TelephoneNumber entity definition, which declares the row locking capability.

Example 13–1 uim-number-entities.xml

```
<entity type="ocim:TelephoneNumber"
managedBy="oracle.communications.inventory.api.number.TelephoneNumberManager">
    .
    .
    .
    <!-- ***************** Capabilities ******************-->
    <rowLockEnabled prefix="TelephoneNumber"/>
</entity>
```

   Example 13–2 shows what you would need to add to the **uim-party-entities.xml** metadata file to extend the Party entity to declare the row locking capability.

Example 13–2 uim-party-entities.xml

```
<entity type="ocim:Party"
managedBy="oracle.communications.inventory.api.number.PartyManager">
    .
    .
    .
    <!-- ***************** Capabilities ******************-->
    <rowLockEnabled prefix="Party"/>
</entity>
```

7. Generate the data model to include any newly declared entity row lock capabilities. See "Generating, Compiling, and Packaging the Entity Source Files" for more information.

Enabling Row Locking

To enable row locking:

1. Configure the **timer.properties** file so the default RowLockExpiryTimerListener class is called at regular intervals by defining the **firstTime** and **period** properties, as shown in Example 13–3.

Example 13–3 timer.properties File

```
# Timer to cleanup the expired entity row locks
rowLockExpiration.firstTime=120
rowLockExpiration.period=600
rowLockExpiration.listener=oracle.communications.inventory.api.common.impl.RowLock
ExpiryTimerListener
```

> **Note:** Do not change the rowLockExpiration.listener property. The default RowLockExpiryTimerListener class clears out expired locks for all entities with row locking capability, including any entities you extend to have the capability.

2. Configure the **system-config.properties** file to set the default values for the **defaultRowLockExpirationDuration** and **maxSupportedRowLocks** properties, as shown in Example 13–4.

Example 13–4 system.config.properties File

```
# The default row locks expiration duration in milliseconds for the entity.
# This value should be defined to be less than the transaction time out.
lockPolicy.defaultRowLockExpirationDuration=30000

# Default maximum number of entities to be row locked.
# This should be in sync with the maximum number or range.
lockPolicy.MaxSupportedRowLocks=100
```

Using Row Locking with Entity Finder APIs

You can write custom code to use row locking with entity finder APIs for any heavily-used entities in your UIM environment.

Understanding How UIM Uses Row Locking

Figure 13–1 shows the flow of an entity finder API call for UIM entities that do not use row locking. When a UIM user initiates an entity search, an entity finder API is called, and the search results are returned to the client.

Figure 13–1 Flow of Entity Finder API without Row Locking

Figure 13–2 shows the flow of an entity finder API call for UIM entities that use row locking. When a UIM user initiates an entity search, the LockPolicy attributes are set before calling the entity finder API, and the locked search results are returned to the client.

Figure 13–2 Flow of Entity Finder API with Row Locking

Writing Custom Code to Use Row Locking

You can write custom code to use row locking with entity finder APIs for any heavily-used entities in your UIM environment. The custom code must set the LockPolicy attributes and call the entity finder API. This can be accomplished through:

- Custom Rulesets
- Custom Web Services

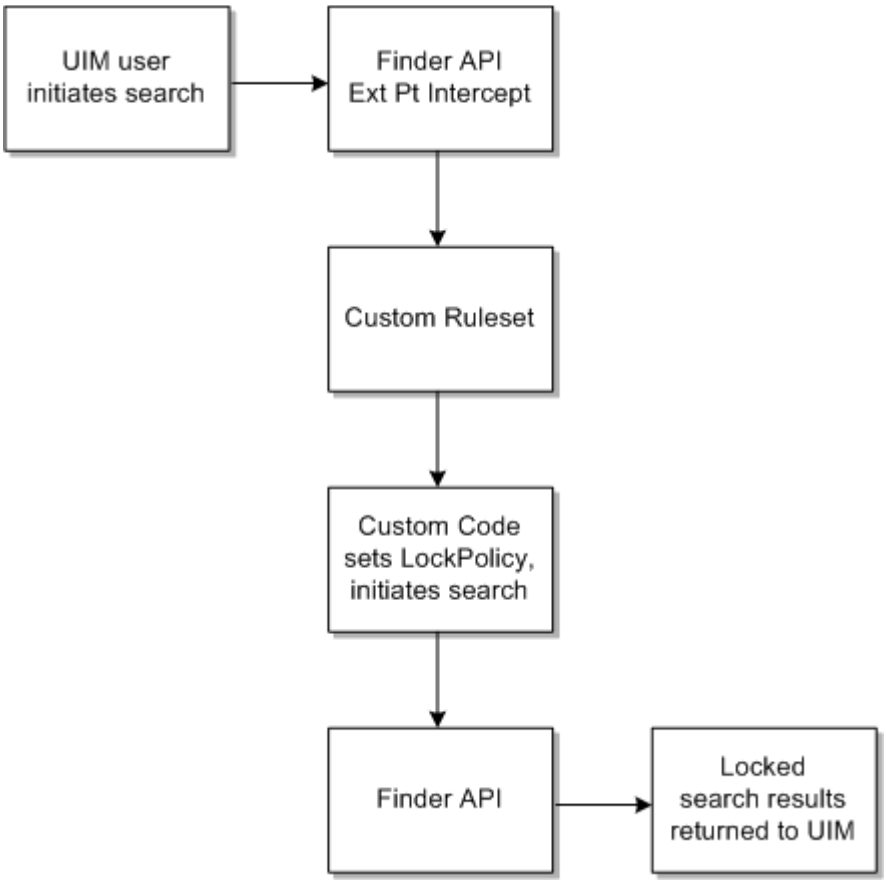
Custom Rulesets

> **Note:** This section builds upon the information presented in Chapter 8, "Extending UIM Through Rulesets", and assumes you have an understanding of rulesets, extension points, and ruleset extension points.

In the following scenario, the custom code resides in an Inventory cartridge, within a custom Java class that is called by a custom ruleset.

Figure 13–3 shows the flow of an entity finder API call for rowlock-enabled entities that you use with row locking through custom code. After the customizations are in place, when a UIM user initiates an entity search, an entity finder API is called. However, before the entity finder API runs, the method is intercepted by the custom extension point, which is configured to run the custom ruleset instead of the entity finder API. The custom ruleset calls the custom code, which sets the LockPolicy attributes and calls the same entity finder API. Based on the LockPolicy attributes specified, the search results are locked and returned to the client.

Figure 13–3 Flow of Entity Finder API Using a Custom Ruleset



The following procedure provides detailed steps and example custom code that you can use to create the customized flow of an entity finder API call that uses row locking.

To use row locking with entity finder APIs:

1. Create a custom Java class.

   Example 13–5 shows a custom Java class that finds party entities using row locking, but you can write similar logic for any entity. To accomplish this, the custom logic must:

   - Create a LockPolicy object
   - Set the LockPolicy attributes
   - Set the entity-specific search criteria object with the LockPolicy object
   - Call the appropriate entity finder API method, passing in the appropriate entity search criteria object that is populated with the LockPolicy
   - Set the ruleset return value to the row-locked results from the entity finder API call

Example 13–5 Custom Java Class

```
package oracle.communications.custom;

import java.util.*;
import oracle.communications.platform.persistence.PersistenceHelper;
import oracle.communications.inventory.api.entity.Party;
```

```
import oracle.communications.inventory.api.party.PartyManager;
import oracle.communications.inventory.api.party.PartySearchCriteria;
import oracle.communications.inventory.api.framework.LockPolicy;
import oracle.communications.inventory.extensibility.extension.util.
ExtensionPointRuleContext;

public class CustomPartySearch
{
    public void main(PartySearchCriteria criteria ExtensionPointRuleContext
    context) throws Exception
    {
        LockPolicy lockPolicy = PersistenceHelper.makeLockPolicy();

        lockPolicy.setNumberOfResources(20);
        lockPolicy.setExpiration(5000);
        lockPolicy.setFilterExistingLocks(true);

        criteria.setLockPolicy(lockPolicy);

        PartyManager partyMgr = PersistenceHelper.makePartyManager();
        List<Party> partyObjs = partyMgr.findParty(criteria);

        context.setReturnValue(partyObjs);
    }
}
```

2. Create a custom ruleset.

   Example 13–6 shows a custom ruleset that call the custom Java class shown in Example 13–5. You can write a similar custom ruleset to call any custom Java class.

Example 13–6 Custom Ruleset Using Drools

```
package oracle.communications.inventory.rules

import oracle.communications.custom.CustomPartySearch;
import oracle.communications.inventory.api.entity.party.PartySearchCriteria;
import racle.communications.inventory.extensibility.extension.util.
ExtensionPointRuleContext;

rule "PartySearch"
    salience 0
when
    criteria : PartySearchCriteria()
    context : ExtensionPointRuleContext()
then
    CustomPartySearch customClass = new CustomPartySearch();
    customClass.main(criteria, context);
end
```

Example 13–7 shows the same custom ruleset content-wise, but using Groovy instead of Drools. For more information on writing custom rulesets, and on the use of Drools and Groovy to do so, see Chapter 8, "Extending UIM Through Rulesets".

Example 13–7 Custom Ruleset Using Groovy

```
package oracle.communications.inventory.rules

import oracle.communications.custom.CustomPartySearch;
import oracle.communications.inventory.api.entity.party.PartySearchCriteria;
import racle.communications.inventory.extensibility.extension.util.
```

```
ExtensionPointRuleContext;

CustomPartySearch customClass = new CustomPartySearch();
customClass.main(criteria, context);
```

> **Note:** The criteria local variable is based on the argument that the custom extension point defines, which is PartySearchCriteria, as shown in Example 13–8. The context local variable of ExtensionPointRulesetContext is made available to all rulesets by the extensibility framework, which appends this argument to the list of arguments defined by the custom extension point. See "ExtensionPointContext and ExtensionPointRuleContext Class" for more information.

3. Create a custom extension point.

   Example 13–8 shows the custom extension point method signature for the findParty API method. You can define a similar extension point signature for any entity finder API.

Example 13–8 Custom Extension Point Signature

```
public abstract interface java.lang.String
oracle.communications.inventory.api.party.PartyManager.findParty(oracle.communicat
ions.inventory.api.entity.party.PartySearchCriteria)
```

4. Create a custom ruleset extension point.

   The custom ruleset extension point configures the custom ruleset to run at the custom extension point. In this scenario, the placement of the custom ruleset must be **Instead** of the method defined by the custom extension point. In this manner, the custom ruleset calls the entity finder API and returns the row-locked entities to UIM.

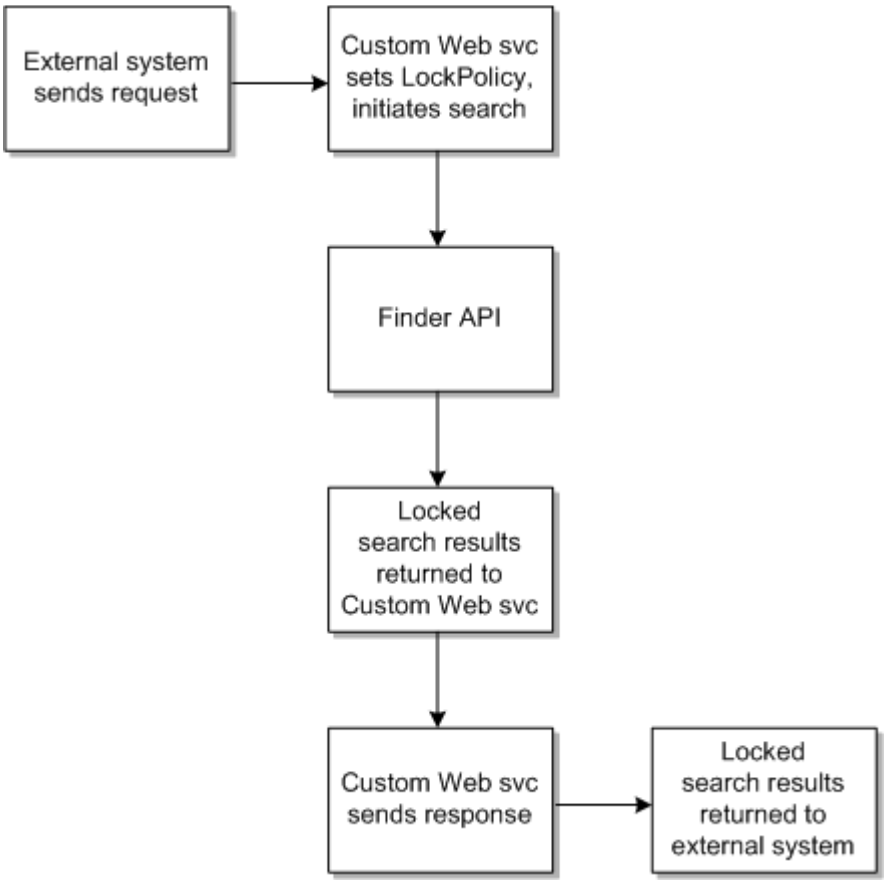
Custom Web Services

> **Note:** This section builds upon information presented in *UIM Web Services Developer's Guide*, and assumes you have an understanding of web services and how to develop them.

In this scenario, the custom code resides in a custom web service.

Figure 13–4 shows the flow of an entity finder API call for rowlock-enabled entities that you use with row locking through custom code. After the customizations are in place, the web service is initiated by an external system through a request. The web service custom code sets the LockPolicy attributes and calls the entity finder API. Based on the LockPolicy attributes specified, the search results are locked and returned to the web service. The web service then sends the locked search results back to the external system through a response.

Figure 13–4 Flow of Entity Finder API Using a Custom Web Service



To use row locking with entity finder APIs through a custom web service, the web service must contain a Java class similar to the one shown in Figure 13–5.

Using Row Locking Without Entity Finder APIs

You can also use row locking without using entity finder APIs. Example 13–9 shows a custom Java class that locks a Collection of entities. To accomplish this, the custom logic must:

- Create a Collection of entities
- Create a LockPolicy object
- Set the LockPolicy attributes
- Call the RowLockManager.lock method, passing in the entity rows to be locked and the LockPolicy used to lock them

Example 13–9 Custom Java Class

```
package oracle.communications.custom;

import java.util.*;
import oracle.communications.platform.persistence.PersistenceHelper;
import oracle.communications.inventory.api.common.RowLockManager;
import oracle.communications.inventory.api.framework.LockPolicy;

public class CustomClass
```

```
{
    public void main() throws Exception
    {
        Collection myCollection = new Collection();

        // Poulate myCollection with like entities, such as Party entities,
        // Role entities, etc.
        .
        .
        .
        // Create a LockPolicy and populate the attributes
        LockPolicy lockPolicy = PersistenceHelper.makeLockPolicy();
        lockPolicy.setNumberOfResources(20);
        lockPolicy.setExpiration(5000);
        lockPolicy.setFilterExistingLocks(true);

        // Call RowLockManager.lock to lock entities in myCollection
        RowLockManager rowLockMgr = PersistenceHelper.makeRowLockMgr();
        Collection myLockedCollection =
            rowLockMgr.lock(myCollection, lockPolicy);
    }
}
```

Note: You can use rulesets and extension points to run the custom code shown in Example 13–9. See "Using Row Locking with Entity Finder APIs" for an example.

14

Using the Federation Framework

This chapter provides information about the Oracle Communications Unified Inventory Management (UIM) federation framework. The federation framework enables UIM to work with other systems, such as Oracle Communications Internet Name and Address Management (INAM), and Oracle Communications MetaSolv Solution (MSS). The federation framework also enables communication with other external systems via different communication protocols.

About the Federation Cartridge Packs

Federation is ability of different software systems to communicate, cooperate, and exchange information. Federation can present a common user experience or simply convey information about a data object from an external system.

Each federation cartridge pack is a set of sample cartridges and artifacts that provides:

- Extension functionality for data domains like IP Address and VLANs interfacing with external systems, such as INAM and MSS. See "About the Federation Data Domain Cartridges" for more information.
- Extension functionality providing a database connection as well as JMS and SOAP communication. See "About the Federation Protocol Cartridges" for more information.

About the Federation Data Domain Cartridges

The federation data domain cartridges focus on the exchange of different types of data, such as IP Addresses and VLANs. You use these cartridges when you design a federation solution utilizing a specific type of data while leveraging externally enabled entities.

These are the federation data domain ZIP files that contain the cartridges:

- **IP Address Federation** - OracleComms_UIM_IPAddress_Federation
- **VLAN ID Federation** - OracleComms_UIM_VLAN_ID_Federation
- **Connectivity Federation** - OracleComms_UIM_Connectivity_Federation
- **Objectel Federation** - OracleComms_UIM_Objectel_Federation

Note: The federation protocol cartridges are not utilized in these cartridges. The federation protocol cartridges use an alternative communication infrastructure to interface with the external systems.

See Appendix A, "Federation Data Domain Cartridges" for more information on these federation data domain cartridges.

About the Federation Protocol Cartridges

The federation protocol cartridges focus on enabling communication protocols. They provide a database connection as well as JMS and SOAP communication. You use these cartridges when you design a federation solution utilizing a specific type of communication protocol optionally using externally enabled entities.

These are the federation protocol cartridges and artifacts:

- **Protocol Cartridge** - OracleComms_UIM_FederationProtocols
- **Properties Cartridge** - OracleComms_UIM_FederationProperties
- **Message Driven Bean** - UIMFederationResponseListenerMBD
- **Response Queue Script** - UIMFederationResponseQueue

See Appendix B, "Federation Protocol Cartridges" for more information on the federation protocol artifacts.

About External Arrangements

In a federation arrangement, specific data access, data management tasks, and processes are transparently delegated to other systems. For example, UIM manages services, and INAM manages IP addresses. The two systems federate through the use of the **ora_uim_ipaddress_cooperation** cartridge. When this federation is in place, it is transparent to you that UIM is communicating with INAM to supply the IP address.

The different ways in which systems federate are called external arrangements. The federation framework supports four types of external arrangements:

- Federated
- Leased In
- Leased Out
- Shared

Table 14–1 lists the external arrangements used by the IP address, VLAN ID, and connectivity cartridges, and also the federation protocol cartridges.

Table 14–1 External Arrangements and the Federation Cartridges

External Arrangement Enum	External Arrangement Display in UI	IP Address Federation	VLAN ID Federation	Connectivity Federation	Federation Protocol
FEDERATED	Viewed From	UIM views IP addresses from INAM.	UIM views network system and product catalog from MSS.	Not applicable.	Not applicable.
LEASED_IN	Leased From	UIM leases in IP address from INAM for service assignment.	Not applicable.	UIM leases in a connection from an external system for service trail enablement.	UIM leases in a Local Loop (Pipe) entity reference from an external system.
LEASED_OUT	Leased To	Not applicable.	UIM leases out VLAN ID to MSS for service assignment.	Not applicable.	Not applicable.
SHARED	Shared With	Not applicable.	MSS shares service catalog and network system entities to relate to UIM VLAN domains.	Not applicable.	Not applicable.

About Transaction-Based and Order-Based Federation

The federation cartridges fall into one of the following categories:

- Transaction-Based Federation
- Order-Based Federation

Transaction-Based Federation

Transaction-based federation is a point-to-point integrations between UIM business logic and an external system. Transaction-based federation typically revolves around a simple resource, such as a telephone number or IP address.

For transaction-based federation to work, the external system must support a synchronous API that UIM can call. The synchronous transaction has a beginning and an end through the use of the startTransaction method and the endTransaction method.

The IP address federation cartridge and the VLAN ID federation cartridge are examples of transaction-based federation. These solutions use the Custom Object and Custom Network Address entities, which have a generic nature that can model virtually any resource from a foreign system. They are simple and avoid complex requests in UIM.

Order-Based Federation

Order-based federation is a schema-based integration between UIM and an external order management system, such as Oracle Communications Order and Service Management (OSM). Order-based federation involves order requests from UIM to an external system that creates, designs, assigns, activates, and tests resources. The

external system then provides a response back to UIM. Order-based federation typically revolves around a multi-phased design and delivery process, such as an OSM order flow. Within the multi-phased process, the external order management system may send requests to UIM to lease data, such as pipe-related data for connectivity.

For order-based federation to work, UIM must support the ability to create an order request and send it asynchronously to an external system. Additionally, UIM must support the ability to listen for, and handle, the asynchronous order response from the external system.

The connectivity federation cartridge is an order-based interface that calls asynchronous APIs provided by external systems to lease connectivity resources. This solution uses the Connectivity (Pipe), Business Interaction, and Service entities. A complex resource such as a circuit can be federated, but UIM does not have a native connectivity understanding of the resource; so, in this type of scenario, it is better to relay the work to the external system through a work order.

Note: The following subsections build upon information presented in *UIM Web Services Developer's Guide*, which describes a Service Order, and how the Service Order is saved as a business interaction attachment.

Work Order

A Service Order is a type of Business Interaction request from an external system for UIM to perform various actions on a Service entity. The actions can affect a service, service configuration, and the life cycles of supporting service configuration item resources. Similarly, a Work Order is a type of Business Interaction request from UIM to an external system to perform various actions on inventory entities in external systems, such as network resources, connections, devices, or services. The work order is used within the context of order-based federation.

The schema used for external systems to communicate with UIM is consistent and extensible. For example, the schema:

- Provides a consistent way to organize and group the items and entities related to the order
- Supports actions with corresponding parameters or properties at the order, item, and entity level
- Defines one structure that is used for both requests and responses, regardless of which system is requesting or responding
- Defines the <parameter> element, which makes the schema readily extensible through custom parameter names and corresponding custom parameter values

Business Interaction Attachment

When a Service Order request is received by UIM from an external system, the XML is saved as business interaction attachment. Similarly, when a Work Order request is sent by UIM to an external system, the XML is saved as a business interaction attachment.

The BusinessInteractionAttachment entity defines the following attributes:

- name

  The name attribute is used for identifying the entity in UIM.

- content

The content attribute supports any generic content for a request or response. The content attribute data type is a BLOB, so the entity attachments can contain formats other than XML requests and responses.

- category

  The category attribute in an enumeration that distinguishes the different attachment categories. The enumeration values are REQUEST and RESPONSE.

- parentAttachment and childAttachments

  The parentAttachment and childAttachments attributes make it possible to receive multiple responses per request, such as relating the request to its responses in a hierarchical relationship. As a result, the parentAttachment and childAttachments attributes create a parent-child relationship for the attachments. The childAttachments attribute is an ordered list.

About Externally Enabled Entities

This section builds upon the information presented in Chapter 4, "Extending the Data Model".

Externally enabled entities are entities that are part of a federation solution. To support federation in UIM, several entities are defined as externally enabled in the metadata through the use of the <externalEnabled> element. The externally enabled entities are:

- BusinessInteraction
- CustomNetworkAddress
- CustomObject
- IPAddress
- IPSubnet
- Pipe
- Service

Note: The federation protocol cartridges do not require using externally enabled entities. Using externally enabled entities is recommended, but is optional depending on your solution requirements.

External Identification

UIM requires a consistent way to identify external entities, so their external system identities need to be maintained. The entity external identity may or may not have similar properties to UIM entity identity.

External entity identities and internal UIM entity identities must be correlated for both systems to operate on the same intended entity. In addition, the same entity may have other types of identity. For example, the NativeEMS domain presents another identity that is typically found for network-facing entities.

So, it is possible for an entity to have multiple identities, depending on the perspective used to refer to the entity. This perspective is known as the entity management domain. The entity management domain is the context in which the entity identity is commonly known and used, which is typically the owner of the entity identity.

It is also possible to have a one-to-many relationship from the entity to multiple identities. However, some of the more commonly used identities are defined as attributes on the main entity to improve performance and to support application logic. For example, the application logic that supports federated inventory is dependent on these identities.

Externally enabled entities have the following generated attributes:

- externalObjectId

  The externalObjectId attribute provides a public unique identity for a business entity within the context of the domain specified by externalManagementDomain.

- externalName

  The externalName attribute provides a business-meaningful name of the business entity (identified by externalObjectID) within the context of the domain specified by externalManagementDomain.

- externalManagementDomain

  The externalManagementDomain attribute identifies an external system, domain name, party, or participant in a federation solution.

  Note: externalManagementDomain is not the entity owner. Entity ownership can refer to technical or system ownership, such as MSS or INAM. Entity ownership can also refer to business or ownership, such as AT&T or East Region. These two types of entity ownership are independent of each other, as is the type of entity ownership that refers to entity identification management. An ownership attribute is not supported.

- externalArrangement

  The externalArrangement attribute is an enumeration that identifies the federation model between UIM and the external party for the given entity. The valid enumerated values are:

  – FEDERATED

    Used when the resource is temporarily retrieved from an external system into UIM views. For example, Network System, Product Catalog, and IP Address; before Network System, Product Catalog, and IP Address are shadowed into UIM.

  – LEASED_IN

    Used when data is leased by UIM from an external system, such as an IP address or a connection.

  – LEASED_OUT

    Used when data is leased by UIM to an external system, such as VLAN ID.

  – SHARED

    Used when data is managed cooperatively between UIM and an external system. For example, Network System and Product Catalog data are shadowed into UIM. That is, the data is stored in both the Network System and in UIM.

Note: In UIM, the availability of a leased resource, such as connectivity (Pipe) or VLAN ID (Custom Network Address), is based on the entity's **Inventory State** attribute value of INSTALLED or UNAVAILABLE. The leasing terms for the resources, such as effective dates, are not managed using additional entity attributes. Leasing terms are instead managed by communication with the external system. For example, UIM is responsible for initiating or terminating the lease of the resources, along with a corresponding update to the resource inventory state; updates to the entity start and end date are not necessary.

Federation Solution Considerations

When planning federation with an external system, you need to consider the following actions.

Determining the Solution Type

When planning a federated solution, one of the first decisions you need to make is determining which type solution best suits your needs:

- Transaction-based solution
- Order-based solution

For the federation data domain cartridges, all of the entities that are used in the provided transaction-based and order-based solutions are defined in the metadata as externally-enabled entities. The federation protocol cartridges use a combination of utilizing characteristics and utilizing externally-enabled entities for persisting data from the external system.

When planning a federation solution, you can use any of these entities in your solution, or you can extend the data model by defining any entity to be externally enabled. See Chapter 4, "Extending the Data Model" for more information.

For additional information on planning a federation solution, see the federation technical white papers, which provide detailed information about these solution approaches, including examples of good approaches, as well as approaches to avoid.

Avoiding Federation Cartridge Conflicts

Oracle recommends that you deploy only one version of each of the cartridges into any given UIM environment. All three of the federation data domain cartridges, and all the federation protocol cartridges can be deployed into one UIM environment, but not multiple versions of the same cartridge.

See *UIM Cartridge Guide* for more information on upgrading and extending cartridges and cartridge packs.

Managing External Identifiers

Your federation solution must manage external identifiers (IDs). The IDs in UIM, and the IDs in the federated external system, must be evaluated and included in the solution planning process. During the planning process, consider the following regarding managing external IDs:

- If the external system is represented in UIM as a Custom Object, the deployed federation cartridge logic must maintain the UIM ID, ensuring the uniqueness across all Custom Objects. The same principle holds true for all the external enabled entities.
- The external system has its own ID for the object. This ID can be used to set the UIM ID for the object. For example, the VLAN ID federation cartridge, which interfaces with MSS, sets UIM IDs to *system-component-externalSystemID*, where *system* is MSS, *component* is an MSS component, and *externalSystemID* is the MSS native ID for the object. This results in UIM IDs such as MSS-NS-1234 or MSS-PC-1234.

  Setting UIM IDs using this type of pattern ensures Custom Objects are unique in UIM. Similarly, the IP Address federation cartridge, which cooperates with INAM, sets the UIM ID to include the unique IP address from INAM. This ensures that the Custom Network Addresses are unique in UIM.
- The UIM externalObjectId attribute stores the external system's unique ID for an object. The externalObjectId value must be set for UIM logic, or any ruleset logic, to correctly determine which objects are external, and which are native, to UIM.

Creating Externally Enabled Entities in UIM

When a deployed federation cartridge creates an externally enabled entity in UIM, and wants to utilize the externally enabled entity features, the logic that creates the entity must also set the entity attributes. This includes the externally-enabled entity attributes of:

externalObjectId

externalName (optional)

externalArrangement

externalManagementDomain

The methods to set these attributes are defined on the entity. For example, *EntityName*.setExternalObjectId(), where *EntityName* is any externally-enabled entity such as CustomObject, BusinessInteraction, Service, and so forth. See "External Identification" for more information about these attributes.

In addition to setting the attributes, the cartridge logic must declare the entity as external by calling the setExternal(true) method. The setExternal() method then calls the setTemporaryEntityId() method to generate a temporary ID for the UIM internal entity ID. Whenever you persist the external entity in UIM, you must first call the unsetTemporaryEntityId() method to remove the temporary ID for the UIM internal entity ID. You can then safely persist the entity in UIM. These methods are also defined on the entity. For example, *EntityName*.setExternal(), where *EntityName* is any externally-enabled entity such as CustomObject, BusinessInteraction, Service, and so forth.

Creating Custom Web Services

You can extend a federation cartridge by adding custom code in a ruleset, or adding custom Java code that a ruleset calls. The custom logic can call a custom web service, making custom web services part of a new federation solution. For example, the VLAN ID federation cartridge that enables communication between UIM and MSS includes a web service called by MSS to update the status of UIM objects.

You can also extend a federation cartridge by using JMS queues to invoke external web services. Refer to the connectivity federation cartridge ora_uim_connectivity_cooperation and the federation protocol cartridge pack for a sample of this scenario.

See *UIM Web Services Developer's Guide* for information on creating custom web services.

15

Integrating UIM Using UIM-Formatted URLs

This chapter provides information about integrating Oracle Communications Unified Inventory Management (UIM) with an external application using a UIM-formatted Uniform Resource Locators (URL), which provides the ability for an external application to access a UIM page.

For example, the OSM-UIM Reference Implementation uses a UIM-formatted URL to access a UIM page from within OSM. In the implementation, the OSM task flow defines numerous tasks, one of which calls the UIM Service Fulfillment Web Service operation of ProcessInteraction. If the operation fails, an OSM fallout task provides a URL link that the user can click to access the UIM Business Interaction Summary page for the business interaction that failed to process.

In such scenarios when a UIM-formatted URL is used to access a UIM page, UIM redirects to the UIM Login page so you can enter your security credentials before continuing to the page specified by the URL. After the specified UIM page is accessed, you have the ability to navigate freely in UIM. For example, from a summary page, you can click **Edit** and update UIM data from a maintenance page.

About UIM-Formatted URLs

UIM-formatted URLs are used to access UIM pages from an external application.

Table 15–1 lists the supported UIM pages you can access using UIM-formatted URLs. The table also lists the corresponding entity type that you specify as part of the UIM-formatted URL, as described in "About the URL Format".

If you need to access UIM pages other than those listed in Table 15–1, you can do so by extending the UIM-formatted URL functionality. See "Extending UIM-Formatted URL Functionality" for more information.

Table 15–1 Supported Page Names and Corresponding Entity Type

Page Name	Entity Type
Business Interaction Summary	BusinessInteraction
Connectivity Details	Connectivity
Device Interface Summary	DeviceInterface
Equipment Summary	Equipment
Inventory Group Summary	InventoryGroup
Logical Device Summary	LogicalDevice
Logical Device Account Summary	LogicalDeviceAccount

Table 15–1 (Cont.) Supported Page Names and Corresponding Entity Type

Page Name	Entity Type
Network Summary	Network
Physical Device Summary	PhysicalDevice
Pipe Summary	Pipe
Pipe Configuration Summary	PipeConfiguration
Property Location Details	PropertyLocation
Service Summary	Service
Service Configuration Summary	ServiceConfiguration

About the URL Format

The URL format for invoking the UIM master flow, which is defined by the **MasterFlow.xml** file, is:

```
http://server:port/Inventory/faces/adf.task-flow?adf.tfId=MasterFlow&adf.tfDoc=/WE
B-INF/MasterFlow.xml&id=id&entity=entity
```

where:

- *server:port* is the server and port on which UIM is running
- *id* is the id number of the entity you want to access
- *entity* is the entity type for the summary page you want to access

For example, the following URL invokes the UIM master flow on myServer:7001, and accesses the Business Interaction Summary page for the business interaction with id 456:

```
http://myServer:7001/Inventory/faces/adf.task-flow?adf.tfId=MasterFlow&adf.tfDoc=/
WEB-INF/MasterFlow.xml&id=456&entity=BusinessInteraction
```

> **Note:** The UIM **Help** > **Link To Page** menu option displays the UIM-formatted URL for the current page, which you can view for additional examples.

About id

The id that you specify in the UIM-formatted URL is the sequentially-generated **id** attribute that uniquely identifies a UIM entity. This is not to be confused with the persistent pattern-generated **entityId** attribute.

> **Note:** For more information about the sequentially-generated **id** and persistent pattern-generated **entityId**, see *Oracle Communications Information Model Reference*.

About entity

The entity that you specify in the UIM-formatted URL is the Java class name, so it must be spelled correctly and have no spaces.

For example, this is a valid URL:

```
http://jsmith:7001/Inventory/faces/adf.task-flow?adf.tfId=MasterFlow&adf.tfDoc=/WE
```

```
B-INF/MasterFlow.xml&id=102&entity=BusinessInteraction
```

And this is an invalid URL:

```
http://jsmith:7001/Inventory/faces/adf.task-flow?adf.tfId=MasterFlow&adf.tfDoc=/WE
B-INF/MasterFlow.xml&id=102&entity=Business Interaction
```

About the InventoryGroup Entity

If you specify InventoryGroup as the entity in the UIM-formatted URL, you must replace **id** with **name** and provide the inventory group name because for an inventory group, the name is the unique identifier.

For example:

```
http://jsmith:7001/Inventory/faces/adf.task-flow?adf.tfId=MasterFlow&adf.tfDoc=/WE
B-INF/MasterFlow.xml&name=MyInvGrpName&entity=InventoryGroup
```

Using UIM-Formatted URLs

You can use UIM-formatted URLs to access any of the supported UIM pages from an external application. This is done by writing custom code that constructs a UIM-formatted URL, which is then used as the input argument to the following Java method:

```
javax.faces.context.ExternalContext.redirect(java.lang.String url)
```

The redirect() method takes in a URL and opens a page based on the input URL. The page is opened in a new browser. To return to the external application that called the redirect() method, close the new browser.

To learn more about this method, see the following website:

http://docs.oracle.com/javaee/7/api/javax/faces/context/ExternalContext.html

How you call the custom code depends on the external application and how it supports extending the application through custom code. For example, when extending UIM, custom code is called through the use of rulesets and extension points.

Example 15–1 shows custom code that constructs the UIM-formatted URL and calls the redirect() method.

Example 15–1 Custom Code

```
// This example code assumes that server, port, id, and entity are either input
// arguments, or readily retrievable/available to the custom code.
// It also assumes that id is an int, and the rest are Strings.

String url = "http://" + server + ":" + port +
"/Inventory/faces/adf.task-flow?adf.tfId=MasterFlow&adf.tfDoc=/WEB-INF/MasterFlow.
xml&id=" + Integer.toString(id) + "&entity=" + entity;

FacesContext facesContext = FacesContext.getCurrentInstance();
ExternalContext externalContext = facesContext.getExternalContext();
externalContext.redirect(url);
```

Extending UIM-Formatted URL Functionality

If you need to access UIM pages other than those listed in Table 15–1, you must extend the UIM-formatted URL functionality through the following:

- MasterFlow.xml
- MasterBean.class
- TaskFlowModel.class

MasterFlow.xml

UIM_Home/**inventory.ear/inv.war/WEB-INF/MasterFlow.xml** defines the master task flow for UIM and is used to invoke specific task flows, such as BusinessInteractionSummaryFlow, EquipmentSummaryFlow, and so forth.

Extending MasterFlow.xml

To extend **MasterFlow.xml**:

1. Ensure the <visibility> element defines the <url-invoke-allowed> element. For example:

```
<visibility id="__7">
    <url-invoke-allowed/>
</visibility>
```

2. Ensure the <data-control-scope> element value is set to **isolated**. For example:

```
<data-control-scope>isolated</data-control-scope>
```

MasterBean.class

The *UIM_Home*/**inventory.ear/inv.war/WEB-INF/classes/oracle/communications/inventory/ui/common/bean/MasterBean.class** logic supports the UIM pages listed in Table 15–1.

MasterBean runs when an ADF standard-formatted URL that invokes the UIM MasterFlow is entered in a browser. **MasterBean** uses the input ADF standard-formatted URL to construct the UIM-formatted URL by doing the following:

- Defines a static Map that houses taskflow data
- Defines an init() method that:
  - Extracts the entity type and entity id from the ADF standard-formatted URL
  - Uses the entity type to get the entity taskflowMap and load it into the static Map that houses the taskflow data
  - Uses the entity id to get the entity and stores some of the entity data in local variables
  - Calls the MasterBean.getCurrentUrl() method, passing in various taskflow data and entity data to construct and return the UIM-formatted URL
  - Calls the javax.faces.context.ExternalContext.redirect() method, passing in the UIM-formatted URL to access the page
- Defines the getCurrentUrl() method, which takes in various taskflow data and entity data and uses it to construct and return the UIM-formatted URL

Extending MasterBean

If you need to access UIM pages other than those listed in Table 15–1, you must extend **MasterBean** to handle any additional taskflow data or entity data that may be needed to access other UIM pages, construct the UIM-formatted URL accordingly using the additional data, and call the redirect() method with the customized UIM-formatted URL.

To extend **MasterBean**:

1. View the UIM-formatted URL for the page you want to access to determine what data the custom URL requires:
   a. Log in to UIM.
   b. Navigate to the UIM page you want to access from the external application.
   c. Click **Help** > **Link To** to display the UIM-formatted URL.
   d. Study the UIM-formatted URL to determine what data it uses.
2. Create a new custom Java class that extends **oracle.communications.inventory.ui.common.bean.MasterBean**.
3. Define a static Map named **taskFlowMap** to store taskflow-related data.

   For example:

```
private static final Map<String, TaskFlowModel> taskFlowMap =
    new HashMap<String, TaskFlowModel>();
```

4. In a static block, define the taskflow for the UIM page you need to access.

   For example, the existing **MasterBean** logic defines the taskflows for all of the UIM pages listed in Table 15–1, of which Business Interaction is shown here:

```
static {
TaskFlowModel taskFlowModelObj =
    new TaskFlowModel(BusinessInteraction.class,
        InventoryUIBundleManager.getLabel("BUSINESS_INTERACTION_SUMMARY"));

taskFlowModelObj.setTaskFlowId("/WEB-INF/oracle/communications/inventory/ui/bus
inessinteraction/flow/BusinessInteractionSummaryFlow.xml#BusinessInteractionSum
maryFlow");

taskFlowMap.put("BusinessInteraction", taskFlowModelObj);
}
```

5. Define an init() method that does the following:

   Note: These steps assume your custom code knows the entity type you want to access, and has the id of the entity you want to access.

   a. Use the entity type to get the entity taskflowMap from the static Map that is defined and loaded when the class is created. For example:

```
taskFlowModelObj = taskFlowMap.get(entity);
```

   b. Use the entity id to get the entity. For example:

```
finder = PersistenceHelper.makeFinder();
Collection result = null;
result = finder.findById(taskFlowModelObj.getEntityClass(),id);
```

c. Depending on what you determined in step 1, create local variables and populate them with the retrieved taskflow data and retrieved entity data.

The variables you create depend on the data required for the UIM-formatted URL for the page you want to access. These variables will be used as method input parameters in the next step.

d. Call your custom getURL() method (as described in step 6, below).

Pass in the appropriate taskflow data and entity data to construct and return the UIM-formatted URL. For example, the following shows a call to the getURL() method using local variables of taskFlowId, taskKeyList, taskParameterList, taskLabel, and taskType:

```
String url = getURL(taskFlowId,taskKeyList,taskParametersList,
                    taskLabel,taskType);
```

e. Call the **javax.faces.context.ExternalContext.redirect()** method, passing in the UIM-formatted URL to access the page. For example:

```
FacesContext facesContext = FacesContext.getCurrentInstance();
ExternalContext externalContext = facesContext.getExternalContext();
externalContext.redirect(url);
```

6. Define a custom getURL() method that:

a. Defines input arguments that reflect the taskflow-related data and entity data needed to construct the custom UIM-formatted URL.

b. Constructs the custom UIM-formatted URL using the input argument data.

See "About the URL Format" for detailed information on the URL format.

c. Returns a String that is the custom UIM-formatted URL.

Example 15–2 shows the MasterBean.getCurrentURL() method, which you can use in writing a similar custom getURL() method.

Example 15–2 MasterBean.getCurrentURL()

```
public String getCurrentURL(String navTaskFlowId, String navTaskKeyList,
                            String navTaskParametersList, String navTaskLabel,
                            String navTaskType) throws Exception
{
    try {
        String homePageWebApp = null;
        String homePageViewId = null;
        boolean contextualAreaCollapsed =
            PatternsConstants.DEFAULT_CONTEXTUAL_AREA_COLLAPSED;
        int contextualAreaWidth =
            PatternsConstants.DEFAULT_CONTEXTUAL_AREA_WIDTH;
        StringBuffer url = null;
        FacesContext facesContext = FacesContext.getCurrentInstance();
        String urlStr = null;
        urlStr = ControllerContext.getInstance().getGlobalViewActivityURL
                 ("/InventoryUIShell");

        int qIndex = urlStr.indexOf('?');
        if (qIndex > -1) {
            urlStr = urlStr.substring(0, qIndex);
        }
```

```
url = new StringBuffer(urlStr);
if (urlStr.indexOf('?') > -1) {
    url.append('&');
} else {
    url.append('?');
}
url.append(MainAreaHandler.FND).append('=');

StringBuffer fndBuffer = new StringBuffer();
if (navTaskFlowId != null) {
    fndBuffer.append
        (URLEncoder.encode(navTaskFlowId, MainAreaHandler.UTF_8));
}
fndBuffer.append(';');
if (navTaskParametersList != null) {
    fndBuffer.append
        (URLEncoder.encode(navTaskParametersList, MainAreaHandler.UTF_8));
}
fndBuffer.append(';');
if (navTaskKeyList != null) {
    fndBuffer.append
        (URLEncoder.encode(navTaskKeyList, MainAreaHandler.UTF_8));
}
fndBuffer.append(';');
if (navTaskLabel != null) {
    fndBuffer.append
        (URLEncoder.encode(navTaskLabel, MainAreaHandler.UTF_8));
}
fndBuffer.append(';');
if (navTaskType != null) {
  fndBuffer.append
      (URLEncoder.encode(navTaskType, MainAreaHandler.UTF_8));
}
fndBuffer.append(';');
fndBuffer.append
    (URLEncoder.encode(String.valueOf(contextualAreaCollapsed),
     MainAreaHandler.UTF_8));
fndBuffer.append(';');
fndBuffer.append
    (URLEncoder.encode(String.valueOf(contextualAreaWidth),
     MainAreaHandler.UTF_8));
url.append
    (URLEncoder.encode(fndBuffer.toString(), MainAreaHandler.UTF_8));

// Get from the model when navigating from the Home Page
Map<String, Object> viewScopeMap =
    AdfFacesContext.getCurrentInstance().getViewScope();
DistributedMenuModel homepageMenuModel =
    (DistributedMenuModel)viewScopeMap.get
        (PatternsConstants.HOMEPAGE_MENU_MODEL);
if (homepageMenuModel == null) {
    homepageMenuModel =
       (DistributedMenuModel)PatternsUtil.getCurrentDistributedMenuModel
           (PatternsConstants.HOMEPAGE_MENU_MODEL);
}
if (homepageMenuModel != null) {
    ItemNode focusItemNode =
        PatternsUtil.getFocusItemNode(homepageMenuModel);
    if (focusItemNode != null) {
        homePageViewId = focusItemNode.getFocusViewId();
```

```
                Map<String, String> customPropList =
                    focusItemNode.getCustomPropList();
                if (customPropList != null) {
                    homePageWebApp =
                        customPropList.get(PatternsConstants.WEBAPP_NAME);
                }
            }
        }
        // Model may not be available when avigating in between Apps pages
        if (homePageViewId == null && homePageViewId == null) {
            homePageWebApp =
               (String)viewScopeMap.get(PatternsConstants.WEBAPP_NAME);
            homePageViewId =
               (String)viewScopeMap.get(PatternsConstants.HOMEPAGE_VIEW_ID);
        }
        if (homePageViewId != null && homePageWebApp != null) {
            url.append("&").append(PatternsConstants.WEBAPP_NAME).append
            ("=").append(homePageWebApp).append
            ("&").append(PatternsConstants.HOMEPAGE_VIEW_ID).append("=").append
            (URLEncoder.encode(homePageViewId, MainAreaHandler.UTF_8));
        }
        ControllerContext.getInstance().markScopeDirty(viewScopeMap);
        HttpServletRequest request =
           (HttpServletRequest)facesContext.getExternalContext().getRequest();
        String hostnamePort = request.getScheme() + "://" +
            request.getServerName() + ":" + request.getServerPort();
        String fullUrl = hostnamePort + url.toString();
        return fullUrl;
    }
    finally {
    }
}
```

TaskFlowModel.class

UIM_Home/**inventory.ear/inv.war/WEB-INF/classes/oracle/communications/inventory/ui/common/model/TaskFlowModel.class** holds taskflow-related data. Specifically, **TaskFlowModel** defines the **entityClass**, **displayLabel**, and **taskFlowId** class variables, and defines get() and set() methods for each.

When **MasterBean** is created, an instance of **TaskFlowModel** is created for each of the supported UIM pages listed in Table 15–1. The code example in step 4 of the "Extending MasterBean" shows this.

Extending TaskFlowModel

The **TaskFlowModel** logic supports the UIM pages listed in Table 15–1. If you need to access UIM pages other than those listed in Table 15–1, you may need to extend the **TaskFlowModel** logic to hold any additional taskflow-related data needed to access other UIM pages.

> **Note:** You only need to extend **TaskFlowModel** if additional data is needed to construct the custom UIM-formatted URL, as determined in step 1 of the "Extending MasterBean" section.

To extend **TaskFlowModel**:

1. Create a new custom Java class that extends **oracle.communications.inventory.ui.common.model.TaskFlowModel**.
2. Define additional class variables as needed.
3. For each class variable, define the corresponding get() and set() methods.

A

Federation Data Domain Cartridges

This appendix provides information about using the Oracle Communications Unified Inventory Management (UIM) federation data domain cartridges. These cartridges enable UIM to work cohesively with other external systems, such as Oracle Communications Internet Name and Address Management (INAM) and Oracle Communications MetaSolv Solution (MSS), in handling IP addresses, VLAN IDs, and connectivity.

> **Note:** This appendix builds upon the information presented in Chapter 14, "Using the Federation Framework".

About the Federation Data Domain Cartridges

The federation data domain cartridges focus on the exchange of different types of data, such as IP Addresses and VLANs. You use these cartridges when you design a federation solution utilizing a specific type of data while leveraging externally enabled entities.

Understanding the federation data domain cartridges is necessary when extending UIM to federate data or cooperate with external systems.

> **Note:** In previous releases, UIM referred to federation using the term "cooperation" in describing this type of functionality.

The federation data domain cartridges include the following:

- **IP Address Federation**

  In the IP address federation cartridge, UIM works with INAM. For example, UIM manages services, and INAM manages IP addresses. When a service resource assignment requires an IP address, INAM is the IP address resource repository from which UIM finds a resource to assign. The cartridge ZIP filename is **OracleComms_UIM_IPAddress_Federation.zip**.

- **VLAN ID Federation**

  In the VLAN ID federation cartridge, UIM works with MSS. For example, MSS manages service configurations, and UIM manages VLAN IDs. When a service resource assignment requires a VLAN ID, UIM is the VLAN ID resource repository from which MSS finds a resource to assign. The cartridge ZIP filename is **OracleComms_UIM_VLAN_ID_Federation.zip**.

- **Connectivity Federation**

In the connectivity federation cartridge, UIM works generically with an external system to manage connectivity resources. For example, UIM sends a Work Order to an external system, such as MSS, to request a lease on connections. The cartridge ZIP filename is **OracleComms_UIM_Connectivity_Federation.zip**.

- **Objectel Federation**

  In the objectel federation cartridge set, UIM demonstrates connectivity interfacing with Objectel for MetroEthernet service fulfillment scenarios. The cartridge ZIP filename is **OracleComms_UIM_Objectel_Federation.zip**.

Note: The federation protocol cartridges are not utilized in these federation data domain cartridges. These federation protocol cartridges use an alternative communication infrastructure to interface with the external systems.

Accessing the Federation Data Domain Cartridges

You can download the federation data domain cartridges from the Oracle Software Delivery Cloud. Within the cartridge pack ZIP file, these cartridges will be located in the **Federation/Data Domains** directory.

See *UIM Cartridge Guide* for instructions on how to download the cartridge pack ZIP file.

You can access detailed information about these federation data domain cartridges in the following technical white papers, located within the cartridges:

- UIM_Cooperation_Technical_Spec_Intro.pdf
- UIM_Cooperation_Technical_Spec_VLAN_ID.docx
- UIM_Cooperation_Technical_Spec_Connectivity.docx

Additionally, the connectivity topic contains the following sequence diagrams:

- UIM_Cooperation_Technical_Spec_ConnectivitySeqDiagCreateLease.pdf
- UIM_Cooperation_Technical_Spec_ConnectivitySeqDiagDisconnect.pdf

Using the Federation Data Domain Cartridges

The IP Address and VLAN ID federation cartridges are fully functional and can be deployed with minimal customizations: You must import the cartridge into Oracle Communications Design Studio and:

- Update the properties files to reflect your environment
- Update CooperationConstants class to reflect your environment
- Rebuild the cartridge to include your updates

The connectivity federation cartridge is also functional; however, in addition to minimal customizations listed above, you must also:

- Configure your UIM environment to specify the external system

Creating New or Extending Existing Federation Data Domain Cartridges

You can also use a federation data domain cartridge as an example to follow when creating a new cartridge, or you can modify an existing federation data domain

cartridge to suit your requirements. Whether you are creating a new cartridge or modifying an existing federation cartridge, you must avoid deploying cartridges that contain same-named rulesets, extension points, or ruleset extension points. Doing so results in the last cartridge deployed overriding any same-named cartridge content that was previously deployed. See "Avoiding Federation Cartridge Conflicts".

Note: Follow the guidelines in *UIM Cartridge Guide* when creating or extending federation data domain cartridges.

Federation Solution Considerations

When planning a federation solution, you need to consider the following actions.

Creating New Specifications

If you are extending a federation data domain cartridge by creating a new specification, Oracle recommends that you follow the specification naming convention used in the federation data domain cartridges. The naming convention dictates that specification names end with **(E)** to designate the specifications that represent external objects. For example, if you intend to name your new specification:

- INAMIPAddress, instead name it INAMIPAddress(E)
- VLANIDForM6, instead name it VLANIDForM6(E)
- M6NetworkSystem, instead name it M6NetworkSystem(E)

While this naming convention is not required, it makes it possible for end users to immediately recognize which objects are external.

Accessing a New External System

Accessing a new external system may be part of a new federation solution. For example, your solution may require UIM to have a federated view of System-X. In this scenario, UIM must be able to query SES and include object references to System-X.

When accessing System-X:

- Determine which items to retrieve from System-X. In this example, Component-X has been the determined item to retrieve from System-X.
- Determine how Component-X is to be represented in UIM. For example, Component-X can be represented by any externally-enabled entity such as CustomObject, CustomNetworkAddress, and so forth.
- Determine the mechanism that is to retrieve the information from the System-X. For example, API calls, direct database queries, and so forth.
- Determine the specification names for these objects.
- Create the necessary Custom Object, Custom Network Address, Pipe, Business Interaction, and Service specifications in Design Studio.
- Create a new Java manager interface that contains the method signatures that support the desired behavior.
- Modify the ruleset logic to call the new Java Manager and its methods. Ruleset logic can be added directly in the DRL or GROOVY file, or in a new Java class that the DRL or GROOVY file calls. Keep in mind that you cannot debug a DRL file. You can debug a Groovy file, and you can debug a Java class called by either a

DRL or Groovy file. See Chapter 8, "Extending UIM Through Rulesets" for more information.

When accessing System-X and using the federation data domain cartridges:

- Create a file that contains any constants you may need for accessing a new external system. For example, see the **CooperationConstants.java** source file. This file is located in each of the federation data domain cartridges, in the **src/oracle/communications/inventory/techpack/cooperation/***data***/common** directory, where *data* is either **ipaddress**, **vlanid**, or **connectivity**, depending on the federation data domain cartridge.

  In the constants file:

  - Add new constants for any new specification names.
  - Add new constants for the interface to System-X. For example, host, port, user ID, and password.
  - Add new constants for the external system and the component. For example, System-X and Component-X.

- Create a file that contains a list of all externally-enabled entities, and that includes mapping information of each entity to the external system and component. For example, see the **ExternalEntitiesRegistry.java** source file. This file is located in each of the federation data domain cartridges, in the **src/oracle/communications/inventory/techpack/cooperation/***data***/common** directory, where *data* is either **ipaddress**, **vlanid**, or **connectivity**, depending on the federation data domain cartridge.

- Create a new Java implementation class that contains the logic that supports the desired behavior. Add any new error messages to the *data***Cooperation.properties** file, where *data* is **ipaddress**, **vlanid**, or **connectivity**.

B

Federation Protocol Cartridges

This appendix provides information about using the Oracle Communications Unified Inventory Management (UIM) federation protocol cartridges, which enable UIM to interface with other external systems via different communication protocols, such as a database connection, JMS, and SOAP.

Note: This appendix builds upon the information presented in Chapter 14, "Using the Federation Framework".

About the Federation Protocol Cartridges

The federation protocol cartridges enable communication with external systems using various protocols. They provide functionality for a database connection as well as JMS and SOAP communication. You use these cartridges when you design a federation solution utilizing a specific type of communication protocol optionally using externally enabled entities.

Understanding the federation protocol cartridges is necessary when extending UIM to communicate with external systems. You can utilize these cartridges for the following communication protocol options:

- JMS
- SOAP
- Database connection

The JMS protocol is asynchronous, and the SOAP and database connection protocols are synchronous. The federation protocol cartridges include:

- A set of infrastructure artifacts to use communicating with the external system.
- A set of artifacts providing an example implementation illustrating the usage of the infrastructure artifacts.

About the Federation Protocol Infrastructure Artifacts

You can use the federation protocol infrastructure artifacts to enable JMS, SOAP and database connection communication with external systems. The federation protocol infrastructure artifacts include the following:

- **Protocol Cartridge**

  The protocol cartridge provides abstract adapter classes for leveraging a variety of protocols in federation integrations. You can extend these adapters to provide

your tailored and specific implementation. The cartridge name is **OracleComms_UIM_FederationProtocols**.

- **Properties Cartridge**

  The properties cartridge provides files for managing property settings. For instance, you can configure the following types of information:

  – The JMS request queue

  – The class name that provides the implementation

  The cartridge name is **OracleComms_UIM_FederationProperties**.

- **Message Driven Bean**

  The message driven bean listens for messages on the response queue for the JMS protocol. You must package this MDB file in the **custom.ear** file. The filename is **UIMFederationResponseListenerMBD.jar.**

- **Response Queue Script**

  The response queue Python script creates the response queue for the JMS bridge. You must update this file to reflect the appropriate host name, port, user name, and password before running the script. The Python script file name is **UIMFederationResponseQueue.py**.

About the Federation Protocol Implementation Sample

The implementation example provides sample code utilizing the federation protocol infrastructure. This example interfaces with an additional UIM system as the external system.

> **Note:** This implementation uses UIM as the external system, however using UIM as an external system is not a typical business scenario. UIM is used as the example so you can successfully test the implementation and compare it to your external system selection.

The following set of artifacts provide the implementation example.

- The OracleComms_UIM_FederationProtocolsImpl cartridge is an example of using and implementing the OracleComms_UIM_FederationProtocols cartridge and its adapter classes.
- The ExternalSystem cartridge is used for the SOAP and JMS testing setup on the external system. This cartridge contains the following:
  – Various UIM specifications that the code references
  – A ruleset extension point
  – Ruleset code to complete a business interaction
  – A BIHelper class which has a completeBI() method to look up the response queue and post a reply
- ExternalSystemsMDB is the message driven bean that listens for JMS messages on the request queue of the external system.
- **ExternalSystemQueues.py** is the Python script that creates the request and response queues for the JMS protocol on the external system. You must update this file to reflect the appropriate host name, port, user name, and password values.

- **UIMFederationRequestQueue.py** is the Python script that creates the request on queue for the JMS protocol in UIM.
- The externalsystem_webservice cartridge is a web service that can be invoked on the external system. The WSDL file is located within the cartridge with the filename **ExternalSystemWS.wsdl** in the **wsdl** directory.

The following set of technical documents provide setup instructions and an overview of the implementation:

- **UIM Reference Implementation JMS Federation_Setup_Guide.docx** details the required steps for the JMS protocol setup.
- **UIM Reference Implementation SOAP Federation_Setup_Guide.docx** details the required steps for the SOAP protocol setup.
- **UIM Reference Implementation Federation_Dev_Guide.docx** contains implementation details, such as class level information.

Accessing the Federation Protocol Cartridges

You can access the federation protocol infrastructure artifacts from the UIM Software Development Kit (SDK). Within the UIM SDK, you find these cartridges in the **federation_sdk.zip** file which is located in the **cartridges/sample** directory.

See "About the UIM SDK" for more information on the UIM SDK.

You can download the federation protocol implementation sample from the Oracle Software Delivery Cloud. Within the cartridge pack ZIP file, these artifacts will be located in the **Federation/Communication Protocols** directory.

See *UIM Cartridge Guide* for instructions on how to download the cartridge pack ZIP file.

Using the Federation Protocol Cartridges

After importing the federation protocol cartridge into Design Studio, you use the technical documents provided in the **Federation/Communication Protocols/doc** directory of the cartridge pack ZIP file. Refer to the *UIM Cartridge Guide* for instructions on how to download the cartridge pack ZIP file

These technical documents aid in understanding how to extend, configure and test these cartridges.

Extending the Federation Protocols Cartridge Functionality

You extend the OracleComms_UIM_FederationProtocols cartridge to build your own tailored implementation to federate data in an external system. This cartridge contains the following adapter classes for supporting communication via JMS, SOAP or a database connection. The adapter classes are:

- InventoryFederationJMSAdapter
- InventoryFederationSOAPWSAdapter
- InventoryFederationDBAdapter

These classes all extend from the base class InventoryFederationBaseAdapter.

Configuring the Federation Properties Cartridge

You use the OracleComms_UIM_FederationProperties cartridge to configure property file settings for the external system communication. This cartridge contains the following files that can be configured:

- The **federation-config.xml** file contains information about the external system, such as the connection type the supported entities and the external system access information.
- The **federation-config-schema.xsd** contains the XML schema definition for the federation-config.xml file. Oracle does not recommend altering this file, unless you "clone and own" the cartridges. If you alter this file, you must change the supporting classes to support the changes.

  See *UIM Cartridge Guide* for more information on the "clone and own" topic for cartridges.

In addition to the properties files in this cartridge, you change the UIM **system-config.properties** file to specify if the configuration file reloads. You must add this property and its setting to the **system-config.properties** file.

```
uim.federationProtocols.federationConfigReload=true
```

By default, this setting is not in the system configuration file and must be added. You use the true setting to reload the values in the federation-config.xml file. This setting is recommended for development when values are changing.

Changing the Entity Type

You can set the entity type within the SupportedEntities section of the **federation-config.xml** file. This portion of the file shows UIMEntityType field:

```
<SupportedEntities>
  <EntityInfo>
    <UIMEntityType>Pipe</UIMEntityType>
    <ExternalSystemEntityType>Pipe</ExternalSystemEntityType>
    <UIMEntitySpecification>IH_Local_Loop</UIMEntitySpecification>
    <ExternalSystemSupportedOperationList>Create,Retrieve,Update,Delete
    </ExternalSystemSupportedOperationList>
  </EntityInfo>
</SupportedEntities>
```

You find the valid values for this setting in the OracleComms_UIM_ FederationProtocols cartridge. The Java enum InventoryFederationSupportedEntityTypes in the package:

```
oracle.com.inventory.federationFramework
```

defines the valid entities for the UIMEntityType field.

Changing Operations List

You can set the operations list, such as create, retrieve, update, and delete within the SupportedEntities section of the **federation-config.xml** file. This portion of the file shows the ExternalSystemSupportedOperationList field:

```
<SupportedEntities>
  <EntityInfo>
    <UIMEntityType>Pipe</UIMEntityType>
    <ExternalSystemEntityType>Pipe</ExternalSystemEntityType>
    <UIMEntitySpecification>IH_Local_Loop</UIMEntitySpecification>
```

```
    <ExternalSystemSupportedOperationList>Create,Retrieve,Update,Delete
    </ExternalSystemSupportedOperationList>
  </EntityInfo>
</SupportedEntities>
```

You find the valid values for this setting in the OracleComms_UIM_FederationProtocols cartridge. The Java enum InventoryFederationSupportedOperations in the package:

```
oracle.com.inventory.federationFramework
```

defines the valid entities for the ExternalSystemSupportedOperationList field.

External System Settings

The **federation-config.xml** also contains the settings to login to the external system. This section provides sample XML file segments for the SOAP, JMS and database connection protocol settings.

SOAP Protocol Settings

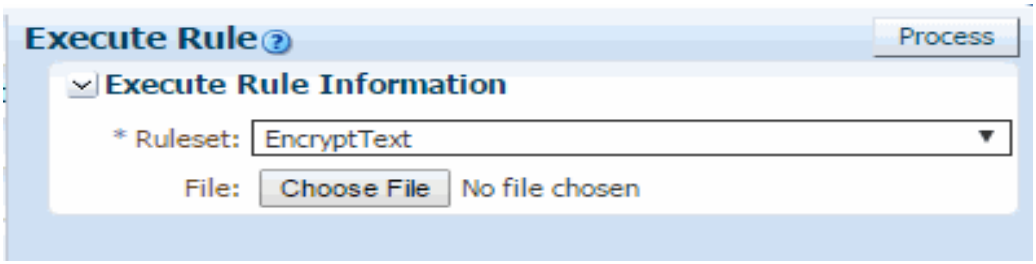
Example B–1 shows a portion of the external system metadata information for the SOAP communication protocol:

Example B–1 SOAP Protocol External System Settings

```
<ExternalSystemMetadata>
  <Property>
    <Name>WS_URL</Name>
    <Value>http://localhost:9001/ExternalSystemWS/ExternalSystemWSHTTP</Value>
  </Property>
  <Property>
    <Name>WS_USERNAME</Name>
    <Value>8A2F9B36DE151F1A12C878EE41681F54</Value>
  </Property>
  <Property>
    <Name>WS_PASSWORD</Name>
    <Value>DE59BC74FC2B56C0FF19E0D7BCA8C016</Value>
  </Property>
</ExternalSystemMetadata>
```

You must update these values in the XML file with the correct values to connect to the external system. To set the user name and password values, you run the rule EncryptText under the Execute Rule option in UIM GUI. Figure B–1 shows the GUI to run a rule and encrypt these values.

Figure B–1 Execute EncryptText Rule UI in UIM



See Appendix C, "Base Rulesets" for examples of running base rulesets.

JMS Protocol Settings

Example B–2 shows a portion of the external system metadata information for the JMS protocol:

Example B–2 JMS Protocol External System Settings

```
<ExternalSystemMetadata>
  <Property>
    <Name>CONNECTION_FACTORY</Name>
    <Value>federationQueueCF</Value>
  </Property>
  <Property>
    <Name>REQUEST_QUEUE</Name>
    <Value>federationRequestQueue</Value>
  </Property>
</ExternalSystemMetadata>
```

You must update the connection factory and the request queue name values in the XML file with your values to connect to the external system.

Database Connection Protocol Settings

Example B–3 shows a portion of the external system metadata information for the database connection protocol:

Example B–3 Database Connection Protocol External System Settings

```
<ExternalSystemMetadata>
  <Property>
    <Name>JDBC_DATA_SOURCE</Name>
    <Value>jdbc/MssTxDataSource</Value>
  </Property>
</ExternalSystemMetadata>
```

You must update the JDBC data source value in the XML file with your value to connect to the external system.

C

Base Rulesets

Note: This appendix assumes that you have read Chapter 8, "Extending UIM Through Rulesets" and have an understanding of rulesets, extension points, ruleset extension points, and enabled extension points.

This appendix provides information about the Oracle Communications Unified Inventory Management (UIM) base rulesets, which are located in the *UIM_Home*/**cartridges/base/ora_uim_baserulesets.jar** file. This appendix provides a description of each base ruleset, and instructions on how to run each base ruleset.

Each base ruleset provides both a DRL file and a GROOVY file.

Address Range Validation

The Address Range Validation base ruleset validates the content of an input GeographicAddressRange object. If a validation error is encountered, the ruleset logs an error. If a validation error is not encountered, processing continues. You can customize the base ruleset to perform custom validations as required by your business needs, and configure the ruleset to run when an address is associated with an address range in UIM.

Running the Base Ruleset

To run the base ruleset, perform the following steps. For instructions on how to perform each step, see the Oracle Communications Design Studio Help and the UIM Help.

1. In Design Studio, import the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges.
2. Modify the ADDRESS_RANGE_VALIDATION base ruleset to reflect your business needs regarding validating address ranges.
3. Save the modified base ruleset.
4. Deploy the extended **ora_uim_baserulesets** cartridge, which now contains the modified ADDRESS_RANGE_VALIDATION ruleset.
5. Create an Inventory project.

   The Inventory Project editor appears.
6. Click the **Dependency** tab.

7. Add the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges as dependencies.
8. Save the Inventory cartridge.
9. Within the cartridge, create a global ruleset extension point.

   The Ruleset Extension Point-Global editor appears.
10. Configure the global ruleset extension point as indicated in Table C–1:

Table C–1 Global Ruleset Extension Point Configuration

Ruleset	Point	Placement
ADDRESS_RANGE_ VALIDATION	AddressRangeManager_ validateAddressForRange	Instead (There are no core validations. The validateAddresForRange() method exists for the purpose of running custom validations. So, whether you configure the base ruleset to run before, after, or instead, the outcome is the same; the custom validations are the only validations that run.)

11. Save the global ruleset extension point.
12. Build the cartridge and deploy it into UIM.
13. In UIM, associate an address to an address range.

    When you associate an address to an address range, the AddressRangeManager.validateAddressForRange() method is called. This results in the base ruleset running instead the method, thereby running your custom validations.

Convert LD SR1 to SR2

Because logical devices were enhanced to include the ability to define rate codes. The Convert LD SR1 to SR2 base ruleset updates existing logical devices that were created in UIM prior to release 7.2.2 to have rate codes.

For example, prior to UIM Release 7.2.2, XYZ Logical Device Specification is created and does not define rate codes; all logical devices created in UIM based on XYZ Logical Device Specification do not have rate codes. In UIM 7.2.2, XYZ Logical Device Specification is updated to define rate codes; from UIM 7.2.2 and later, all logical devices created in UIM based on XYZ Logical Device Specification have rate codes. This base ruleset provides the ability to update the existing pre-7.2.2 logical devices to have rate codes that reflect the rate codes XYZ Logical Device Specification now defined.

Running the Base Ruleset

To run the base ruleset, perform the following steps. For instructions on how to perform each step, see the Design Studio Help and the UIM Help.

1. Create a text file, **input.txt**, and save it in a temporary directory, **tempDir**.

2. Format the **input.txt** file as follows:

   ```
   LogicalDeviceSpecificationName1
   LogicalDeviceSpecificationName2
   LogicalDeviceSpecificationName3
   ```

   where *LogicalDeviceSpecification* is the name of a Logical Device Specification that defines rate codes. The input file can contain multiple Logical Device Specification names, formatted one per line.

3. Save the **input.txt** file.
4. In UIM, in the Tasks panel Administration group, click the **Execute Rule** link.

   The Execute Rule page appears.

5. From the **Ruleset** list, select **CONVERT_LD_SR1_TO_SR2**.
6. Click **Browse**.

   The Choose File to Upload window appears.

7. Navigate to **tempDir**.
8. Select the **input.txt** file.
9. Click **Open**.

   The Choose File to Upload window closes.

10. In the Execute Rule page, in the upper-right corner, click **Process**.

    The base ruleset runs and, based on the input file content, updates any logical device entities that were created from the specified input Logical Device Specifications. The logical device entities are updated with the rate codes defined for the applicable Logical Device Specification.

Create Address Characteristic Map

The Create Address Characteristic Map base ruleset creates a Map object and populates it with characteristic names and corresponding values. The characteristic names and values are mapped from a GeographicAddress object that is input to the ruleset, and the characteristics in the GeographicAddress are derived from the Place specification from which the input GeographicAddress entity was created.

For example, the MyPlace Place specification defines characteristics of address, city, state, and zipCode, as well as a default value for each. In UIM, when a place is created from the MyPlace Place specification, the characteristics are populated with the default values defined in the Place specification. The place is saved as a GeographicPlace object. If the Create Address Characteristic Map base ruleset is called passing in this place, the result is a Map object containing the characteristics of address, city, state, and zipCode, and the corresponding value for each.

Running the Base Ruleset

The Create Address Characteristic Map base ruleset is a supporting ruleset that is called by the Find Address Range and Validate Address For Range base rulesets. The Create Address Characteristic Map ruleset is not intended to be configured to run through a ruleset extension point. See "Find Address Range" and "Validate Address for Range" for more information, and where modifying the Create Address Characteristic Map base ruleset is a step in the instructions for running these base rulesets.

Find Address Range

The Find Address Range base ruleset finds an AddressRange that is valid for the input GeographicAddress, and returns the found AddressRange through the ruleset context. You can customize the base ruleset to find the AddressRange based on customized criteria from the input GeographicAddress, and configure the ruleset to run when you search for an address range for a specific address.

The Find Address Range base ruleset calls the Create Address Characteristic Map base ruleset. See "Create Address Characteristic Map" for more information about this supporting base ruleset.

Running the Base Ruleset

To run the base ruleset, perform the following steps. For instructions on how to perform each step, see the Design Studio Help and the UIM Help.

1. In Design Studio, import the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges.
2. Modify the FIND_ADDRESS_RANGE base ruleset to reflect your business needs regarding finding an address range.
3. Modify the CREATE_ADDRESS_CHARACTERISTIC_MAP base ruleset, which the FIND_ADDRESS_RANGE ruleset calls.

   See "Create Address Characteristic Map" for more information.
4. Save the modified base ruleset.
5. Deploy the extended **ora_uim_baserulesets** cartridge, which now contains the modified FIND_ADDRESS_RANGE ruleset.
6. Create an Inventory project.

   The Inventory Project editor appears.
7. Click the **Dependency** tab.
8. Add the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges as dependencies.
9. Save the Inventory cartridge.
10. Within the cartridge, create a global ruleset extension point.

    The Ruleset Extension Point-Global editor appears.
11. Configure the global ruleset extension point as indicated in Table C–2:

Table C–2 Global Ruleset Extension Point Configuration

Ruleset	Point	Placement
FIND_ADDRESS_RANGE	AddressRangeManager_ findAddressRange	Instead

12. Save the global ruleset extension point.
13. Build the cartridge and deploy it into UIM.
14. In UIM, search for an address range for a specific address.

    When you search for an address range for a specific address, the AddressRangeManager.findAddressRange() method is called. This results in the

base ruleset running instead the method, where your customized search criteria is then used to find the address range.

Import Inventory

The Import Inventory base ruleset does the following, using an input text file that provides a telephone number ID, a logical device account ID, and an equipment ID:

- Creates an instance of a telephone number if it does not exist
- Creates an instance of a logical device account if it does not exist
- Creates an instance of equipment if it does not exist
- Validates and creates a custom involvement between the telephone number and logical device account
- Validates and creates a custom involvement between the logical device account and equipment

Running the Base Ruleset

To run the base ruleset, perform the following steps. For instructions on how to perform each step, see the Design Studio Help and the UIM Help.

1. In Design Studio, create the following specifications, which will be used by the ruleset to create telephone numbers, logical device accounts, and equipment:
   - Telephone Number Specification
   - Logical Device Account Specification
   - Equipment Specification
2. Save the specifications.
3. Deploy the cartridge containing the specifications you just created.
4. Import the **ora_uim_baserulesets** cartridge into Design Studio.
5. Modify the IMPORT_INVENTORY ruleset to reflect the specification names you just created.
6. Save the modified base ruleset.
7. Deploy the extended **ora_uim_baserulesets** cartridge, which now contains the modified Import Inventory ruleset.
8. Create an input text file, **input.txt**, and save it in a temporary directory, **tempDir**.
9. Format the content of the comma delimited **input.txt** file as follows:

   ```
   TN,LDA,EQUIP
   TelephoneNumberId,LogicalDeviceAccountId,EquipmentId
   ```

   where *TN*, *LDA*, and *EQUIP* are the attribute names defined in the ruleset, and where *TelephoneNumberId*, *LogicalDeviceAccountId*, and *EquipmentId* are the corresponding values of the attributes. The file format requires a minimum of one set of attribute values, but multiple sets of attribute values can also be specified.
10. Save the **input.txt** file.
11. In UIM, in the Tasks panel Administration group, click the **Execute Rule** link.

    The Execute Rule page appears.

12. From the **Ruleset** list, select **IMPORT_INVENTORY**.
13. Click **Browse**.

    The Choose File to Upload window appears.
14. Navigate to **tempDir**.
15. Select the **input.txt** file.
16. Click **Open**.

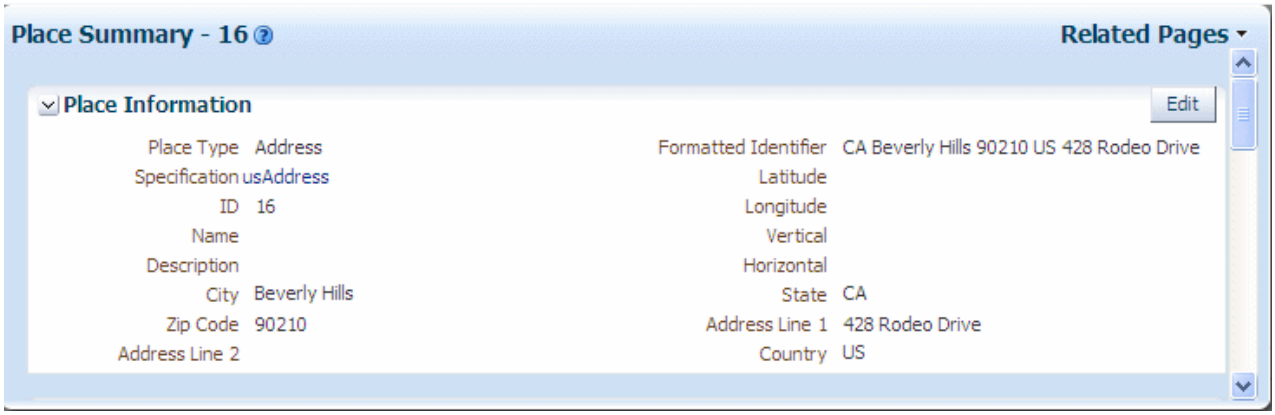
    The Choose File to Upload window closes.
17. In the Execute Rule page, in the upper-right corner, click **Process**.

    The base ruleset runs and, based on the input file content, telephone numbers, logical devices, and equipment are created if the supplied IDs are not found. The ruleset also creates custom involvements between any supplied telephone numbers and logical devices, and creates custom involvements between ant supplied logical device accounts and equipment.

Place Format Identifier

The Place Format Identifier ruleset defines the display format of the **Formatted Identifier** field for places that are associated with a specific Place specification. The field is comprised of the characteristics defined for the Place specification with which the place is associated. This ruleset runs in UIM whenever the **Formatted Identifier** field displays for a place. Figure C–1 shows the Place Summary page for an address where the ruleset concatenates the **State**, **City**, **Zip Code**, **Country**, **Address Line 1**, and **Address Line 2** characteristics to set the **Formatted Identifier** field.

Figure C–1 Formatted Identifier Field



To modify the ruleset, open it in Design Studio and modify the boldface code in Example C–1 if using Drools, or in Example C–2 if using Groovy. The default code randomly appends the characteristics defined for the specification. By modifying the code, you can specify the characteristics to include in the display, and you can specify the order in which the characteristics appear in the display.

Example C–1 Place Format Identifier Rule (Drools)

```
rule "Locations"
        salience 0
    when
        place : GeographicPlace()
```

```
        eval(place instanceof GeographicLocation)
    then
        // execute for location
        StringBuilder formattedIdentifier = new StringBuilder();
        List pc = new ArrayList(place.getCharacteristics());
    if (pc != null) {
        for(int i=0;i<pc.size();i++) {
            String value = ((PlaceCharacteristic)pc.get(i)).getValue();
        if (value != null) formattedIdentifier.append(value).append(" ");
        }
    }
    place.setFormattedIdentifier(formattedIdentifier.toString().trim());
end
rule "Addresses"
        salience 0
    when
        place : GeographicPlace()
        eval(place instanceof GeographicAddress)
    then
        // execute for address
        StringBuilder formattedIdentifier = new StringBuilder();
        List pc = new ArrayList(place.getCharacteristics());
    if (pc != null) {
        for(int i=0;i<pc.size();i++) {
            String value = ((PlaceCharacteristic)pc.get(i)).getValue();
           if (value != null) formattedIdentifier.append(value).append(" ");
        }
    }
    place.setFormattedIdentifier(formattedIdentifier.toString().trim());
end
```

Example C–2 Place Format Identifier Rule (Groovy)

```
if (place instanceof GeographicLocation)
{
    // execute for location
    StringBuilder formattedIdentifier = new StringBuilder();
    List pc = new ArrayList(place.getCharacteristics());
    if (pc != null)
    {
        for(int i=0;i<pc.size();i++)
        {
            String value = ((PlaceCharacteristic)pc.get(i)).getValue();
            if (value != null) formattedIdentifier.append(value).append(" ");
        }
    }
    place.setFormattedIdentifier(formattedIdentifier.toString().trim());
}
if (place instanceof GeographicAddress)
{
    // execute for address
    StringBuilder formattedIdentifier = new StringBuilder();
    List pc = new ArrayList(place.getCharacteristics());
    if (pc != null)
    {
        for(int i=0;i<pc.size();i++)
        {
            String value = ((PlaceCharacteristic)pc.get(i)).getValue();
            if (value != null) formattedIdentifier.append(value).append(" ");
```

```
        }
    }
    place.setFormattedIdentifier(formattedIdentifier.toString().trim());
}
```

Running the Base Ruleset

To run the base ruleset, perform the following steps. For instructions on how to perform each step, see the Design Studio Help and the UIM Help.

1. In Design Studio, import the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges.
2. Modify the PLACE_FORMAT_IDENTIFER ruleset as described above to reflect the needs of your UIM environment.
3. Save the modified base ruleset.
4. Deploy the extended **ora_uim_baserulesets** cartridge, which now contains the modified PLACE_FORMAT_IDENTIFIER ruleset.
5. Create an Inventory project.

   The Inventory Project editor appears.
6. Click the **Dependency** tab.
7. Add the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges as dependencies.
8. Save the Inventory cartridge.
9. Within the cartridge, create a ruleset extension point.

   The Ruleset Extension Point editor appears.
10. Configure the ruleset extension point as indicated in Table C–3:

Table C–3 Ruleset Extension Point Configuration

Ruleset	Point	Placement
PLACE_FORMAT_ IDENTIFIER	PlaceManager_ createGeographicPlace	After

11. Save the ruleset extension point.
12. Within the cartridge, create another ruleset extension point.

    The Ruleset Extension Point editor appears.
13. Configure the ruleset extension point as indicated in Table C–4:

Table C–4 Ruleset Extension Point Configuration

Ruleset	Point	Placement
PLACE_FORMAT_ IDENTIFIER	PlaceManager_ updateGeographicPlace	After

14. Save the ruleset extension point.
15. Because you are working with specification-based extension points, you must also configure any applicable Place Specifications for the ruleset extension points by doing the following for each applicable Place specification:

Note: You do not need to create enabled extension points to configure the Place Specifications; the **ora_uim_baseextpts** cartridge provides the following base enabled extension points, which enable the PlaceManager_createGeographicPlace and PlaceManager_updateGeographicPlace specification-based extension points for the Place Specification:

- PlaceSpecification_PlaceManager_createGeographicPlace
- PlaceSpecification_PlaceManager_updateGeographicPlace

a. Open the Place Specification editor.

b. Click the **Rules** tab.

c. Click **Select**.

The Add Entities window appears.

d. Select the two ruleset extension points that you just created.

e. Click **OK**.

The Add Entities window closes, and the Place Specification is now configured for the two ruleset extension points.

f. Save the Place Specification.

16. Build the cartridge and deploy it into UIM.

17. In UIM, create or update a place.

When a place is created or updated, the PlaceManager.createGeographicPlace() method or PlaceManager.updateGeogrphicPlace() method is called. This results in the base ruleset running after, which uses the customized formatting to display the Place Summary **Formatted Identifier** field.

Reservation Check Redeemer

The Reservation Check Redeemer ruleset enables reservation redemption validations in UIM. By default, reservation redemption validations are disabled in UIM.

In UIM, you can reserve resources to prevent them from being used by other entities or processes. If the reservation is not redeemed by the expiry date, the resource is released back into inventory. You redeem a reserved resource when you assign the resource to a configuration item. By default, UIM does not validate the redemption to ensure that it matches the reservation. So, you may wish to use this ruleset to enable reservation redemption validations.

See *UIM Concepts* for more information about reservations.

Running the Base Ruleset

To run the ruleset, perform the following steps. For instructions on how to perform each step, see the Design Studio Help and the UIM Help.

1. In Design Studio, import the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges.

2. Create an Inventory project.

The Inventory Project editor appears.

3. Click the **Dependency** tab.
4. Add the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges as dependencies.
5. Save the Inventory cartridge.
6. Within the cartridge, create a global ruleset extension point.

   The Ruleset Extension Point-Global editor appears.
7. Configure the global ruleset extension point as indicated in Table C–5:

Table C–5 Global Ruleset Extension Point Configuration

Ruleset	Point	Placement
RESERVATION_CHECK_ REDEEMER	ReservationManager_ checkRedeemer	Instead

8. Save the global ruleset extension point.
9. Build the cartridge and deploy it into UIM.
10. In UIM, assign a reserved resource to a configuration item.

    When you assign a reserved resource to a configuration item, the ReservationManager.checkRedeemer() method is called. This method simply returns false, which results in reservation redemption validations being disabled. Conversely, the base ruleset simply returns true. So, when you configure the base ruleset to run instead of the ReservationManager.checkRedeemer() method, reservation redemption validations are enabled. When enabled, and you assign a reserved resource to a configuration item, you are required to enter valid reservation information to redeem the reservation.

> **Note:** If you enable reservation validations through the base ruleset and then later decide you want to disable reservation validations, modify the ruleset to return **false** and redeploy the cartridge.

Reservation Expiration

The Reservation Expiration base ruleset customizes the reservation expiration process by setting a custom value for the expiry interval when expiring reservations.

When a reservation is created, you can set an expiry date. The expiry date indicates when resource a resource is no longer reserved and the resource returns to the Unassigned state. If no expiry date is set when the reservation is created, the expiry date is calculated and set based on a default interval specified in the *UIM_Home*/**config/consumer.properties** file. The default interval indicates the period of time that must elapse before a resource is no longer reserved and returns to the Unassigned state.

The reservation expiration process runs at timed intervals, as specified in the *UIM_Home*/**config/timers.properties** file. When the reservation expiration process runs, reservations are checked to see if the expiry date has been reached. If the expiry date has been reached, the reservation is deleted and the resource is returned to the Unassigned state.

The Reservation Expiration ruleset is used to customize the telephone number aging process, which is configured to run at timed intervals specified in the *UIM_Home*/**config/timers.properties** file. Reservation expiration is the process of checking

reservations to see if the expiry date has been reached. The Reservation Expiration ruleset provides a way to customize the stateExpiry interval for telephone numbers in the Disconnected state. At the specified timed intervals, the ruleset queries for telephone numbers in the Disconnected or Transitional state. If a telephone number is the Disconnected state, the ruleset transitions the state to Transitional, and if a telephone number is the Transitional state, the ruleset transitions the state to Unassigned.

Running the Base Ruleset

To run the base ruleset, perform the following steps. For instructions on how to perform each step, see the Design Studio Help and the UIM Help.

1. In Design Studio, import the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges.
2. Modify the RESERVATION_EXPIRATION ruleset to reflect the needs of your UIM environment regarding reservation expiration for short term or long term reservations, or both.
3. Save the modified base ruleset.
4. Deploy the extended **ora_uim_baserulesets** cartridge, which now contains the modified RESERVATION_EXPIRATION ruleset.
5. Create an Inventory project.

   The Inventory Project editor appears.
6. Click the **Dependency** tab.
7. Add the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges as dependencies.
8. Save the Inventory cartridge.
9. Within the cartridge, create a global ruleset extension point.

   The Ruleset Extension Point-Global editor appears.
10. Configure the global ruleset extension point as indicated in Table C–6:

Table C–6 Global Ruleset Extension Point Configuration

Ruleset	Point	Placement
RESERVATION_ EXPIRATION	ReservationManager_ expireReservation	Instead

11. Save the global ruleset extension point.
12. Build the cartridge and deploy it into UIM.
13. Modify following property values in the *UIM_Home*/**config/timers.properties** file as needed for your UIM environment configuration:
    - cleanReservation.firstTime
    - cleanReservation.period

    See *UIM System Administrator's Guide* for more information about the **timers.properites** file.
14. Save the modified **timers.properties** file.

ReservationManager.expireReservation() is automatically called at the timer intervals you specified in the **timers.properties** file. When ReservationManager_expireReservation() is called, the ruleset runs instead of the method, as specified by the global ruleset extension point. The ruleset runs your custom code and sets the custom expiry interval for short term or long term reservations, or both.

System Export and System Import

The System Export and System Import rulesets are used to manage UIM data. For example, you may wish to create a new UIM test environment and load the test environment with data from another UIM environment, or you may wish to export UIM data to send the data upstream in the order fulfillment process.

Exporting Data

The System Export ruleset exports database entities into XML-formatted output files, places the XML files in a ZIP file, and returns the ZIP file. Input to the System Export ruleset is a text file that specifies the export configuration. The input text file name is arbitrary, but it is commonly named **config.txt**. The **config.txt** file is also returned in the ZIP file, along with the XML files.

The export configuration specified in the **config.txt** file defines two sections:

- Queries
- Parameters

Queries

Queries must be in the form of:

```
ClassName#QueryString
```

where *ClassName* is the data object representation of the database table you are querying, and *QueryString* is a valid query statement. For example:

```
oracle.communications.inventory.api.entity.TelephoneNumberSpecification#o.name
LIKE 'Sample'
oracle.communications.inventory.api.entity.TelephoneNumberSpecification#o.lastModi
fiedUser='inventory'
```

When defining the queries section in the **config.txt** file:

- You must pre-pend all attribute names with **o** because the table name is hard-coded to have an alias of **o** in the extensibility framework.
- You can specify multiple queries within a single **config.txt** file.

Parameters

Parameters must be in the form of:

```
commitSize=NumericValue
duplicateAction=ActionEnumValue
relationshipsToInclude=IncludeEnumValue
```

Where *NumericValue*, *ActionEnumValue*, and *IncludeEnumValue* are described as values for the commitSize, duplicateAction, and relationshipsToInclude parameters as follows:

- **commitSize:** numeric value greater than 1 and up to a reasonable export size. The default value is 1000.
- **duplicateAction:** action to take when duplicate data is encountered in the target data store. Options are:
  - **Update:** (default) any duplicate record is updated with the newly imported data values.
  - **Ignore:** do not process, skip the duplicate record.
  - **Error:** when a duplicate record exists in the target data store, report it as such and end the transaction without taking any action.
- **relationshipsToInclude:** describes how to process objects that are related to the selected entity. Options are:
  - **Meta:** (default) only include relationships that are metadata type relationships, such as specification relationships.
  - **Data:** only include relationships that are business data in nature, such as characteristics.
  - **All:** include all relationships including metadata and normal business data associations.
  - **None:** do not include any relationships and only process the integral type attributes of the selected entity.

For example:

```
commitSize=1000
duplicateAction=Error
relationshipsToInclude=Meta
```

When defining the parameters section in the **config.txt** file:

- You must place the parameters section directly after the queries section.
- You may specify all, some, or none of the parameters. Any parameters not specified assume the default value.

Caution: The System Export ruleset exports data from the UIM database. Metadata Services (MDS) stores additional data used by the UIM UI in the presentation of specifications and characteristics. The export ruleset does not export this additional data from the MDS.

If you are using System Export to export data from environment A, along with System Import to import data into environment B, you can do the following to work around the issue:

1. Run the System Export ruleset in environment A.
2. Run the System Import ruleset in environment B.
3. Deploy the cartridge or cartridges that define the specifications and characteristics that were previously deployed into environment A (resulting in the additional specification and characteristic data being stored in the MDS) into environment B.

If you are using the System Export ruleset without System Import, for example to send data upstream, this is not an issue.

Importing Data

The System Import ruleset imports previously exported XML-formatted data into the system and returns a count of the records imported. Input to the System Import ruleset is the ZIP file returned by the System Export ruleset; the ZIP file contains the XML files.

Note: To perform the System Import, you must deploy all the same cartridges, that are deployed the System Export environment, to the System Import environment as well.

Caution: Prior to importing data, you must check the data in the XML files for entity IDs that are duplicates of any entity IDs already in the system into which you are importing. If you find duplicate entity IDs, modify the entity IDs in the XML file prior to importing the data.

The way UIM handles duplicate IDs during import depends on the value of the **DuplicateAction** parameter specified in the **config.txt** file. (The **config.txt** file is available to the System Import ruleset because the input is the ZIP file returned from the System Export ruleset; and the ZIP file contains the XML files and the **config.txt** file.)

Running the Base Rulesets

To run the base rulesets, perform the following steps. For instructions on how to perform each step, see the Design Studio Help and the UIM Help.

1. Create a text file, **config.txt**, and save it in a temporary directory, **tempDir**.
2. Format the **config.txt** file as described above in the "Exporting Data" section.
3. Save the **config.txt** file.
4. In the UIM environment from where you plan to export data, in the Tasks panel Administration group, click the **Execute Rule** link.

   The Execute Rule page appears.
5. From the **Ruleset** list, select **SYSTEM_EXPORT**.
6. Click **Browse**.

   The Choose File to Upload window appears.
7. Navigate to **tempDir**.
8. Select the **config.txt** file.
9. Click **Open**.

   The Choose File to Upload window closes.
10. In the Execute Rule page, in the upper-right corner, click **Process**.

    The base ruleset runs and, based on the queries and parameters specified in the **config.txt** file, exports the data to a ZIP file.

    When the process completes, a message displays on the UIM Execute Rule page, informing you of the number of records processed, and the location and name of the created ZIP file. For example:

    ```
    Processed 85 Records Successfully.
    ```

```
Exported Inventory to the location:
/share/uimcluster/domains/clusterUim722b385/UIM/tmp/xmldata/uimuser1/export/xml
export.zip
```

11. Click **Download File** to save **xmlexport.zip** to a temporary directory.

    The File Download window appears.

12. Click **Save**.

    The Save As window appears.

13. Navigate to a temporary directory and click **Save**.

14. If necessary, copy **xmlexport.zip** to a location that can be accessed from the UIM environment where you plan to import the data. (If the UIM environments from which you are exporting and importing are running on the same machine, you do not need to perform this step.)

15. In the UIM environment where you plan to import the data that you just exported, in the Tasks panel Administration group, click the **Execute Rule** link.

    The Execute Rule page appears.

16. From the **Ruleset** list, select **SYSTEM_IMPORT**.

17. Click **Browse**.

    The Choose File to Upload window appears.

18. Navigate to the temporary directory that contains the **xmlexport.zip** file.

19. Select the **xmlexport.zip** file.

20. Click **Open**.

    The Choose File to Upload window closes.

21. In the Execute Rule page, in the upper-right corner, click **Process**.

    The base ruleset runs and, based on the **xmlexport.zip** file content, imports the data.

Telephone Number Formatting

The Telephone Number Formatting base ruleset defines the telephone number length and display format of telephone numbers in UIM. The ruleset defines a default edit mask that is applied to all telephone numbers, unless otherwise specified. The ruleset can also define additional edit masks that apply to specified Telephone Number specifications. The ruleset runs in UIM when working with telephone numbers and applies a default edit mask of 10 digits (##########).

To modify the base ruleset in Design Studio, scroll to the FUNCTIONS section of the code. You can modify the default edit mask, the specification name and corresponding edit mask, or both. You can also replicate the code and define multiple edit masks that are specific to a particular specification. In Example C–3, the boldface code is the code you need to modify.

Example C–3 Telephone Number Formatting Rule (Drools)

```
//---------------------------------------------------------------------
// FUNCTIONS
//---------------------------------------------------------------------
function String getEditMask(TelephoneNumberSpecification tnSpec)
{   // The character # is reserved and represents a required digit.
```

```
    // The default mask is eight required digits.
    String editMask = "##########";
    if ( tnSpec == null )
        return editMask;

    // Define the edit mask based on the spec name
    if(tnSpec.getName().equals("TNspec NPA-NXX"))
        editMask = "###-###-####";

    return editMask;
}
//--------------------------------------------------------------------
// RULES
//--------------------------------------------------------------------
rule "Get TN Edit Mask"
        salience 0
    when
        telephoneNumberSpecification : TelephoneNumberSpecification()
        context : ExtensionPointRuleContext()
    then
        String editMask = getEditMask(telephoneNumberSpecification);
        context.setReturnValue(editMask);
end
```

Example C–4 Telephone Number Formatting Rule (Groovy)

```
//--------------------------------------------------------------------
// FUNCTIONS
//--------------------------------------------------------------------
def String getEditMask(TelephoneNumberSpecification tnSpec)
{
    // The character # is reserved and represents a required digit.
    // The default mask is eight required digits.
    String editMask = "##########";
    if ( tnSpec == null )
        return editMask;

    // Define the edit mask based on the spec name
    if(tnSpec.getName().equals("TNspec NPA-NXX"))
        editMask = "###-###-####";

    return editMask;
}
//--------------------------------------------------------------------
// RULE
//--------------------------------------------------------------------
log.debug ("", "Get TN Edit Mask");
String editMask = getEditMask(telephoneNumberSpecification);
context.setReturnValue(editMask);
```

Example C–5 (Drools) and Example C–6 (Groovy) show the modified portion of the Telephone Formatting rule in boldface. (The only difference between the two examples is that Drools defines a function and Groovy defines a def.)

In these examples, the modified rule redefines the default edit mask length format from 10 digits to 11 digits. The examples also redefine the telephone number display format for telephone numbers created from the NANPA telephone number specification to display as +# (xxx) xxx-xxxx. For example, +1 (972) 555-8495.

Example C–5 Telephone Number Formatting Rule (Drools)

```
function String getEditMask(TelephoneNumberSpecification tnSpec)
 {
    // The character # is reserved and represents a required digit.
    // The default mask is eleven required digits.
    String editMask = "###########";
    if ( tnSpec == null )
        return editMask;

    // Define the edit mask based on the spec name
    if(tnSpec.getName().equals("NANPA"))
        editMask = "+# (###) ###-####";

    return editMask;
}
```

Example C–6 Telephone Number Formatting Rule (Groovy)

```
def String getEditMask(TelephoneNumberSpecification tnSpec)
 {
    // The character # is reserved and represents a required digit.
    // The default mask is eleven required digits.
    String editMask = "###########";
    if ( tnSpec == null )
        return editMask;

    // Define the edit mask based on the spec name
    if(tnSpec.getName().equals("NANPA"))
        editMask = "+# (###) ###-####";

    return editMask;
}
```

From this point forward, all telephone numbers created from the NANPA telephone number specification and the TELEPHONE_NUMBER_FORMATTING ruleset are based on the new default length and display format. Any telephone numbers created from the NANPA telephone number specification prior to this ruleset being deployed do not reflect this new default length and display format, nor will they; the telephone number formatting is not applied retroactively.

You can also modify the telephone number default edit mask in the **number.properties** file. See *UIM System Administrator's Guide* for more information.

Running the Base Ruleset

To run the base ruleset, perform the following steps. For instructions on how to perform each step, see the Design Studio Help and the UIM Help.

1. In Design Studio, import the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges.
2. Modify the TELEPHONE_NUMBER_FORMATTING ruleset as described above to reflect the needs of your UIM environment.
3. Save the modified base ruleset.
4. Deploy the extended **ora_uim_baserulesets** cartridge, which now contains the modified TELEPHONE_NUMBER_FORMATTING ruleset.
5. Create an Inventory project.

The Inventory Project editor appears.

6. Click the **Dependency** tab.
7. Add the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges as dependencies.
8. Save the Inventory cartridge.
9. Within the cartridge, create a ruleset extension point.

   The Ruleset Extension Point editor appears.
10. Configure the ruleset extension point as indicated in Table C–7:

Table C–7 Ruleset Extension Point Configuration

Ruleset	Point	Placement
TELEPHONE_NUMBER_FORMATTING	SpecManager_getEditMask	Instead

11. Save the ruleset extension point.
12. Because you are working with a specification-based extension point, you must also configure any applicable Telephone Specifications for the ruleset extension point by doing the following for each applicable Telephone Number specification:

    Note: You do not need to create an enabled extension point to configure the Telephone Number Specifications; the **ora_uim_baseextpts** cartridge provides the following base enabled extension point, which enables the SpecManager_getEditMask specification-based extension point for the Telephone Number Specification:

    - TelephoneNumberSpecification_SpecManager_getEditMask

    a. Open the Telephone Number Specification editor.
    b. Click the **Rules** tab.
    c. Click **Select**.

       The Add Entities window appears.
    d. Select the ruleset extension point that you just created.
    e. Click **OK**.

       The Add Entities window closes, and the Telephone Number Specification is now configured for the ruleset extension point.
    f. Save the Telephone Number Specification.
13. Build the cartridge and deploy it into UIM.
14. In UIM, create or update a telephone number.

    When you create or update a telephone number, the SpecManager_getEditMask() method is called. This results in the ruleset running instead of the method, which applies the default edit mask or specified edit mask to the telephone number and displays the number accordingly.

Note: Telephone number formatting is not applied retroactively. If you change the formatting for a telephone number specification, the change is not applied to existing telephone numbers created from the specification prior to the edit mask change. However, the change is applied to any new telephone numbers you create from the specification.

Telephone Number Grading

The Telephone Number Grading base ruleset is no longer used. However, the ruleset provides a good example of how to set a characteristic or field, such as country code, when a telephone number is created.

TN Selection

The Telephone Number Grading base ruleset is no longer used, but provides a good ruleset example to follow when creating custom rulesets.

Trail Pipe Topology Edge

Note: The Trail Pipe Topology Edge base ruleset is valid for use with Connectivity - Pipe entities; it is not valid for use with Connectivity - Channelized entities.

The Trail Pipe Topology Edge base ruleset provides a way for a trail pipe to become a topology edge, which makes the trail pipe be included in the topology and available for path analysis.

In pipe connectivity, all trail pipes are topology edged. However, in channelized connectivity, trail pipes that ride channelized pipes are not topology edges and not available for path analysis. So, you may wish to use this ruleset in channelized connectivity to make trail pipes that ride channelized pipes topology edges so they are included in the topology and available for path analysis.

Running the Base Ruleset

To run the ruleset, perform the following steps. For instructions on how to perform each step, see the Design Studio Help and the UIM Help.

1. In Design Studio, import the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges.
2. Create an Inventory project.

   The Inventory Project editor appears.
3. Click the **Dependency** tab.
4. Add the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges as dependencies.
5. Save the Inventory cartridge.
6. Within the cartridge, create a ruleset extension point.

   The Ruleset Extension Point editor appears.

7. Configure the ruleset extension point as indicated in Table C–8:

Table C–8 Ruleset Extension Point Configuration

Ruleset	Point	Placement
TRAIL_PIPE_TOPOLOGY_EDGE	TopologyMapper_createEnabledByPipesTopologyEdge	Instead

8. Save the ruleset extension point.
9. Because you are working with a specification-based extension point, you must also configure any applicable Pipe specifications for the ruleset extension point by doing the following for each applicable Pipe specification:

Note: You do not need to create an enabled extension point to configure the Pipe Specifications; the **ora_uim_baseextpts** cartridge provides the following base enabled extension point, which enables the TopologyMapper_createEnabledByPipesTopologyEdge specification-based extension point for the Pipe Specification:

- PipeSpecification_TopologyMapper_createEnabledByPipesTopologyEdge

---

   a. Open the Pipe Specification editor.
   b. Click the **Rules** tab.
   c. Click **Select**.

      The Add Entities window appears.
   d. Select the ruleset extension point that you just created.
   e. Click **OK**.

      The Add Entities window closes, and the Pipe Specification is now configured for the ruleset extension point.
   f. Save the Pipe Specification.
10. Build the cartridge and deploy it into UIM.
11. In UIM, within a topology model, enable a trail pipe through one or more pipes and save the pipes.

    Saving a pipe calls the processPipe() method, which eventually calls the **TopologyMapper_createEnabledByPipesTopologyEdge** method.

    When the TopologyMapper.createEnabledByPipesTopologyEdge() method is called, the ruleset is called instead.

Validate Address for Range

The Validate Address For Range base ruleset takes in a GeographicAddressRange and a GeographicAddress and determines whether the given address range is valid for the given address. If valid, the ruleset returns the valid geographic address range through the ruleset context. If not valid, the ruleset returns null. You can customize the base ruleset to perform custom validations as required by your business needs, and

configure the ruleset to run when an address is associated with an address range in UIM.

The Validate Address for Range base ruleset calls the Create Address Characteristic Map base ruleset. See "Create Address Characteristic Map" for more information.

Running the Base Ruleset

To run the base ruleset, perform the following steps. For instructions on how to perform each step, see the Design Studio Help and the UIM Help.

1. In Design Studio, import the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges.
2. Modify the VALIDATE_ADDRESS_FOR_RANGE base ruleset to reflect your business needs regarding validating address ranges.
3. Modify the CREATE_ADDRESS_CHARACTERISTIC_MAP base ruleset, which the VALIDATE_ADDRESS_FOR_RANGE ruleset calls.

   See "Create Address Characteristic Map" for more information.
4. Save the modified base ruleset.
5. Deploy the extended **ora_uim_baserulesets** cartridge, which now contains the modified VALIDATE_ADDRESS_FOR_RANGE ruleset.
6. Create an Inventory project.

   The Inventory Project editor appears.
7. Click the **Dependency** tab.
8. Add the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges as dependencies.
9. Save the Inventory cartridge.
10. Within the cartridge, create a global ruleset extension point.

    The Ruleset Extension Point-Global editor appears.
11. Configure the global ruleset extension point as indicated in Table C–9:

Table C–9 Global Ruleset Extension Point Configuration

Ruleset	Point	Placement
VALIDATE_ADDRESS_FOR_RANGE	AddressRangeManager_validateAddressForRange	Instead (There are no core validations. The validateAddresForRange() method exists for the purpose of running custom validations. So, whether you configure the base ruleset to run before, after, or instead, the outcome is the same; the custom validations are the only validations that run.)

12. Save the global ruleset extension point.
13. Build the cartridge and deploy it into UIM.

14. In UIM, associate an address to an address range.

    When you associate an address to an address range, the AddressRangeManager.validateAddressForRange() method is called. This results in the base ruleset running instead the method, thereby running your custom validations.

Validate Relate Places

The Validate Relate Places base ruleset validates the existence of a parent-child relationship between two input GeographicAddress objects by determining if the specifications from which the graphic addresses were created have a parent-child relationship.

Running the Base Ruleset

To run the base ruleset, perform the following steps. For instructions on how to perform each step, see the Design Studio Help and the UIM Help.

1. In Design Studio, import the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges.
2. Create an Inventory project.

   The Inventory Project editor appears.
3. Click the **Dependency** tab.
4. Add the **ora_uim_baseextpts** and **ora_uim_baserulesets** cartridges as dependencies.
5. Save the Inventory cartridge.
6. Within the cartridge, create a ruleset extension point.

   The Ruleset Extension Point editor appears.
7. Configure the ruleset extension point as indicated in Table C–10:

Table C–10 Ruleset Extension Point Configuration

Ruleset	Point	Placement
VALIDATE_RELATE_PLACES	PlaceManager_relatePlaces	Before

8. Save the ruleset extension point.
9. Because you are working with a specification-based extension point, you must also configure any applicable Place specifications for the ruleset extension point by doing the following for each applicable Place specification:

   Note: You do not need to create an enabled extension point to configure the Place Specifications; the **ora_uim_baseextpts** cartridge provides the following base enabled extension point, which enables the PlaceManager_relatePlaces specification-based extension point for the Place Specification:

   - PlaceSpecification_PlaceManager_relatePlaces

   a. Open the Place Specification editor.

b. Click the **Rules** tab.

c. Click **Select**.

   The Add Entities window appears.

d. Select the ruleset extension point that you just created.

e. Click **OK**.

   The Add Entities window closes, and the Place Specification is now configured for the ruleset extension point.

f. Save the Place Specification.

10. Build the cartridge and deploy it into UIM.

11. In UIM, relate two places.

    When you relate two places, the PlaceManager.relatePlaces() method is called. This results in the ruleset running before the method, which validates the relationship between the geographic places before establishing the relationship between them.