Oracle® Communications Service Catalog and Design

Developer's Guide

Release 8.0
F78560-02
November 2023

Oracle Communications Service Catalog and Design Developer's Guide, Release 8.0

F78560-02

Contents

3 Working with Guided Assistance

12 Design Studio Platform Tools

13 Extending Solution Designer

Preface

This guide explains how to work with Oracle Communications Service Catalog and Design - Design Studio design patterns, guided assistance, and externally created schemas. It explains how to automate builds and provides information about implementing continuous integration to design, create, and deliver operations support system (OSS) solutions across Oracle Communications products.

This guide assumes that you have a conceptual understanding of Design Studio and have read the *Concepts* guide.

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 work with Service Catalog and Design to build the code to support the metadata-driven components of Service Catalog and Design projects. You should have a good working knowledge of development languages such as Java, XPath, XQuery, or SQLcreate service fulfillment solutions.

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.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

1

Creating, Packaging, and Distributing Plug-in Projects

This chapter provides information about how to work with plug-in projects in Oracle Communications Service Catalog and Design - Design Studio. It provides a short overview of plug-in projects and provides instructions for creating, packaging, and distributing plug-in projects.

About Plug-in Projects

A plug-in is a modular, extendable, and sharable unit of code that enables integration of tools within Eclipse. Plug-ins are saved in Eclipse plug-in projects. You use plug-in projects to package and deliver guided assistance, design patterns, and reports to other design studio users.

Plug-in projects include a **MANIFEST.MF** file that describes the plug-in, the dependencies, a **plugin.xml** file to identify the extension points, and a set of Java classes to implement the extension points.

Plug-in projects are grouped into feature projects. Feature projects are groups of plug-ins packaged together into a single unit. You use update sites to organize feature projects and to make the feature projects available to your team members for installation. In this manner, team members can install and update the groups of plug-ins as a single unit.

For example, a plug-in project can include any number of design patterns or report designs, and a feature project can be associated with any number of plug-in projects. Feature projects are associated with update site projects, and update sites are distributed to solution designers to enable access to the design patterns.

See the *Eclipse Plug-in Development Environment Guide* for more information.

Creating Plug-in Projects

To create a new plug-in project:

1. In Design Studio, from the **File** menu, select **New**, and then select **Project**.

   The New Project dialog box appears.
2. Expand the **Plug-in Development** folder and then select **Plug-in Project**.
3. Click **Next**.
4. In the **Project Name**, enter a name for the plug-in project.
5. Set the values for the remaining fields with information specific to your installation.

   For more information about the Plug-in Project wizard fields, see the Eclipse *Plug-in Development Environment Guide*.
6. Click **Next**.

   The Content page appears, which enables you to customize the plug-in data.

7. In the **Options** area, deselect the **Generate an activator** option.
8. Set the values for the remaining fields with information specific to your installation.

   For more information about the Plug-in Project wizard fields, see the Eclipse *Plug-in Development Environment Guide*.
9. Click **Finish**.

   Design Studio prompts you to open the Plug-in Development perspective.
10. Click **Yes**.

    Design Studio switches to the Plug-in Development perspective and displays the project manifest (MANIFEST.MF) in the Plug-in Project editor.

Packaging Plug-in Projects

To package a plug-in project:

- Create a feature project to contain the plug-in project. See "Creating Feature Projects" for more information.
- Create an update site project in which you bundle the feature projects. See "Creating Update Site Projects" for more information.

Creating Feature Projects

To create a feature project:

1. From the **File** menu, select **New** and then select **Other**.

   The New Project wizard appears.
2. Expand the **Plug-in Development** folder.
3. Select **Feature Project** and click **Next**.
4. In **Project name** field, enter a name for the feature project.
5. Accept the defaults for the remaining fields, or enter values specific to your feature project.
6. Click **Next**.

   The Referenced Plug-ins and Fragments page appears.
7. Select the plug-in projects to include in the feature.
8. Click **Finish**.

Creating Update Site Projects

To create an update site project:

1. From the **File** menu, select **New** and then select **Other**.

   The New Project wizard appears.
2. Expand the **Plug-in Development** folder.
3. Select **Update Site Project** and click **Next**.
4. In **Project name** field, enter a name for the update site project.

5. Click **Finish**.

   The new project opens in the Update Site Map editor.

6. Click **New Category**.

   The Category Properties area appears.

7. In the **ID** field, enter a unique ID for the new category.
8. In the **Name** field, enter a name that will appear for the category in the update site.
9. In the Managing the Site area, select the new category and click **Add Feature**.

   The Feature Select dialog box appears.

10. Select the feature project that contains your plug-in projects and click **OK**.

    If you do not see your feature listed, begin typing the name of the feature in the Feature Selection dialog box.

11. In the Managing the Site area, click the **Build All** button.

Distributing Plug-in Projects

Plug-in projects are grouped into features, and features are made available to users through update sites. To distribute custom functionality, such as report designs and design patterns, you create a feature project using the Feature Project Creation wizard and add the plug-in project to the feature project. When you are finished, contact your system administrator to request that the new feature be added to the Design Studio update site.

See *Eclipse Plug-in Development Environment Guide* for information about using the Feature Project Creation wizards. Design Studio report design examples demonstrate how to configure a feature project.

2

Working with Design Patterns

This chapter provides information about design patterns, how to create design patterns in Oracle Communications Service Catalog and Design - Design Studio, and how to distribute design patterns.

About Design Patterns

Design Studio design patterns are wizards that automate complex, repeatable tasks, and that enable team members with varying levels of skill to complete those tasks. When extending solutions, you may be required to repeat design activities multiple times and in a specific order. Design patterns enable you to define a generic pattern that, when run, automates the creation of model objects and their relationships in a user's workspace. Your teams can use design patterns to reduce errors, simplify modeling, and increase productivity.

Typically, designers create design patterns using information identified from existing reference implementations and sample solutions. For example, a designer can identify common modeling tasks and the key resources included in those tasks, then create a design pattern to formalize those tasks into a reusable modeling pattern (one that is not specific to any domain). The designer then distributes the design pattern to solution design teams.

Solution design teams install design patterns as Design Studio features and, using wizards, apply the patterns to their workspace. These wizards ensure compliance with the best practices and reduce the need for coding and complex configuration.

When a user runs a design pattern, a domain-specific implementation of the design pattern is run in the user workspace. For example, you might create a design pattern that creates customer edges or provider edges in a VPN. Or, a user might run a design pattern to set up a Technical Order Management layer or a Service Order Management layer for a solution.

About the Design Pattern Framework

The design pattern framework includes:

- A Data Model

  The data model includes all of the entities and data elements that realize the pattern, how that data is organized in the workspace, the expected user input and how the input is applied to the workspace, and the embedded Help available when the user runs the design pattern.

- A User Interface

  Users interact with design patterns using the Design Pattern wizard. The wizard collects information from the user. The information that the design pattern requires can be organized onto different wizard pages, can be augmented with hints or embedded help, and can be validated when entered by the user.

- An Implementation Processor

  When a user completes the Design Pattern wizard and clicks the **Finish** button, the design pattern applies the user input against the entities and data elements defined in the

design pattern data. The design pattern generates and organizes the entities and data elements in the user workspace.

About the Design Pattern Development Life Cycle

The life cycle of a design pattern begins with the identification and isolation of the pattern itself. Working from a reference implementation, designers identify the repeatable pattern, which comprises the resources and the relationships of the resources to the workspace.

The tasks in the life cycle of a design pattern are completed by two different actors, a designer who creates and distributes design patterns, and a user (solution designer) who installs the design patterns and then runs the patterns to facilitate solution development.

Designers do the following:

1. Evaluate common modeling tasks and key resources in reference implementations, sample solutions, and best practices. then identify which repeatable tasks can be automated in a design pattern.
2. Develop design patterns using the identified resources as key components of the design patterns.
3. Test design patterns by running them in the Design Studio environment.
4. Include design patterns in plug-in projects, associate the plug-in project to a feature project, and associate the feature project to an update site.
5. Distribute update sites to Design Studio users.

Design pattern users do the following:

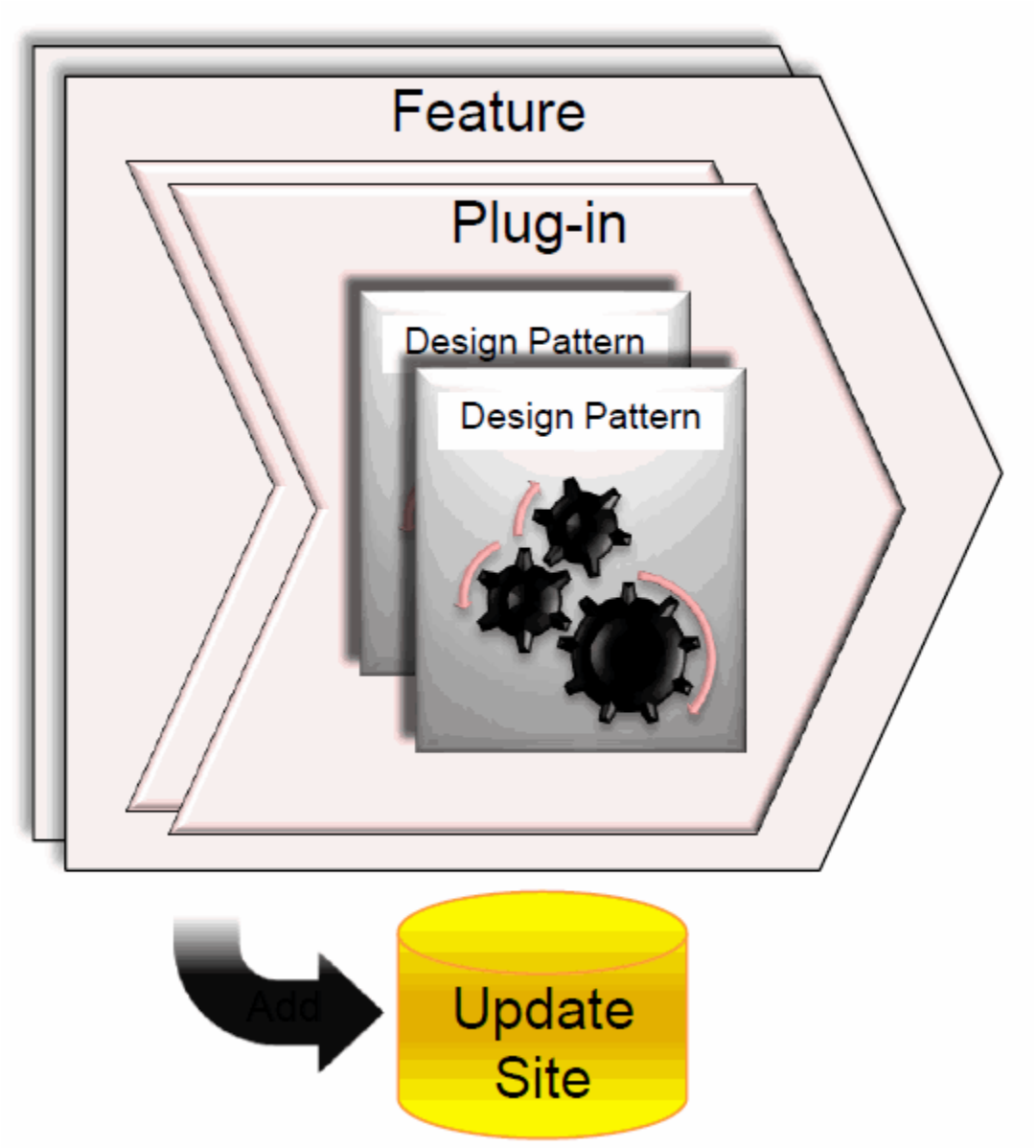
1. Install features based on their role and objectives.
2. Run design patterns to assist with solution design.

About the Design Pattern Development Environment

Design patterns are built into Eclipse plug-in projects, and plug-in projects are associated with feature projects. See "About Plug-in Projects" for more information.

When creating feature projects, include all logically related design patterns. For example, you might include in a single feature project all design patterns applicable to OSM-IPSA integrations to ensure that the patterns are always installed together. Oracle recommends that you create separate features for design patterns that target different audiences.

Figure 2-1 Design Pattern Development Environment



The number of plug-in projects that you use depends on the number of design pattern designers and whether the team members prefer to own their own plug-in projects. For example, team members may prefer to manage their own plug-in projects during the development cycle to avoid difficult source code merges.

To facilitate ease of distribution and maintenance, partition the design patterns across plug-ins so that common design patterns can be distributed to distinct audiences.

About Design Pattern Folder Structure

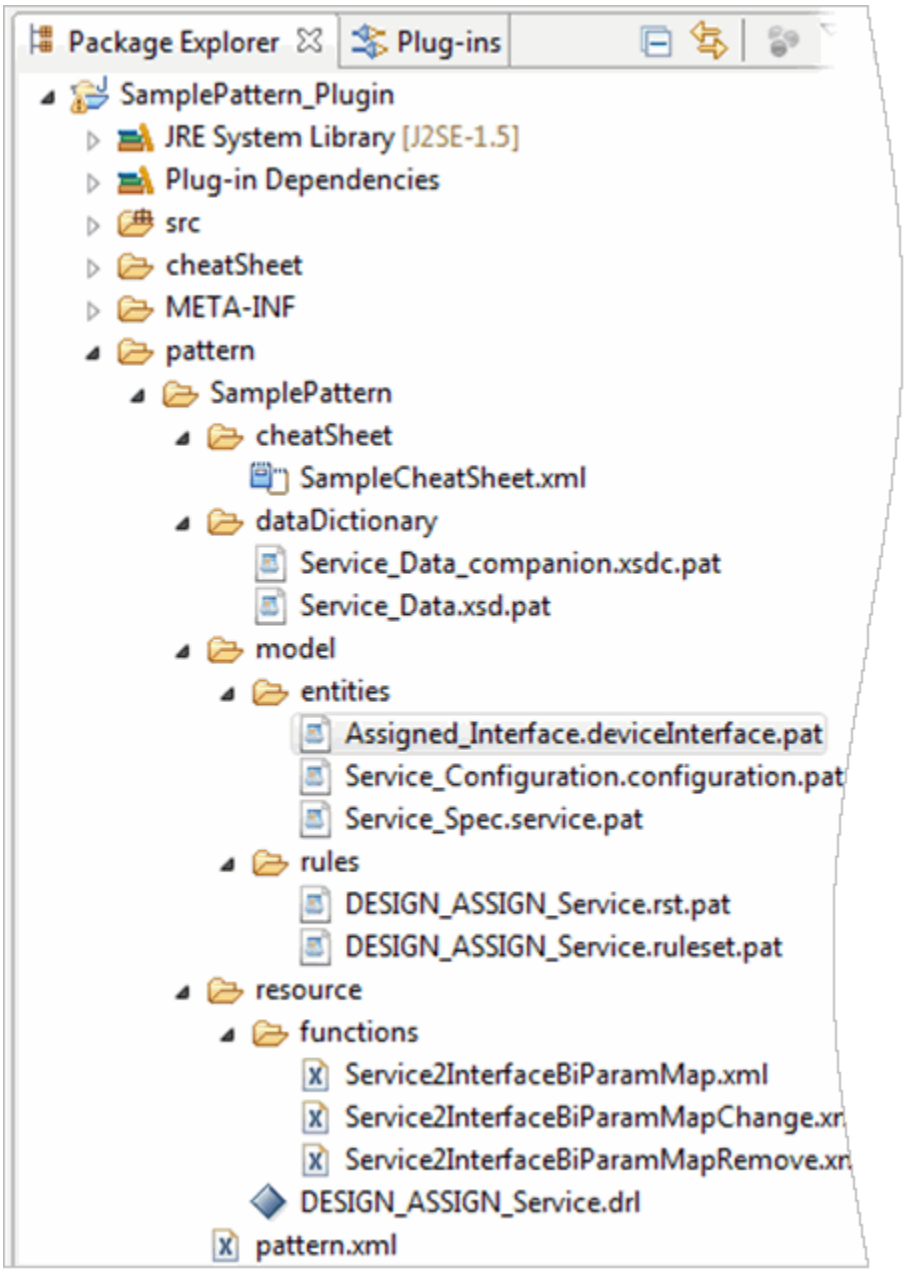
In a design pattern plug-in project, each pattern is assigned a folder:

plug-inID/**pattern**/*patternID*

where *plug-inID* is the name of the plug-in project and *patternID* is the name of a design pattern in the **pattern** folder.

Designers develop all of the design pattern resources in a folder structure. Figure 2-2 illustrates an example of the folder structure as viewed in the Package Explorer view when you create a plug-in project using the Design Pattern Development design pattern.

Figure 2-2 Design Pattern Folder Structure Example



In your plug-in project **pattern** folder, there exists a folder for each design pattern. For example, in Figure 2-2, the **pattern** folder contains a single design pattern called **SamplePattern**.

Each design pattern folder contains subfolders and a **pattern.xml** file:

- **cheatSheet**: include one or multiple cheat sheets. For example, you might include a cheat sheet that the design pattern initiates to provide additional information to the user who runs the design pattern.
- **dataDictionary**: include all data schemas and all data dictionary companion files relevant to the entities generated by the design pattern.
- **model**: include the definitions of the entities generated by the design pattern. For example, you include all of the resources (suffixed with **.pat** extension) generated by the design pattern.
- **resource**: include the design pattern resources that are not Design Studio model entities. For example, this folder can contain Java or XML files.
- **pattern.xml**: define the contents of the design pattern model.

About the pattern.xml File

Each design pattern has a **pattern.xml** file that defines the contents of the design pattern model. In addition to the high-level attributes defined for the **pattern.xml** file (for example, the **namespace**, **pattern ID**, and **name** attributes), the file also includes the sections in Table 2-1.

For detailed descriptions of the elements and attributes in the **pattern.xml** file, see the Design Pattern XML Schema, which is named **DesignPattern.xsd** in the **schema** folder, available in the Design Studio software package, which is available from the Oracle software delivery website:

https://edelivery.oracle.com

Table 2-1 pattern.xml Sections

Element	Description
version	The version number of the design pattern. For example, you can specify whether the design pattern is a major or minor release, or part of a service pack release.
description	The information that describes how the design pattern can be used. This information is displayed to users on the Design Pattern wizard Introduction page. See "Defining a Description for a Design Pattern" for more information.
includes	The list of existing design patterns in a workspace from which a design pattern leverages logic. See "Leveraging Logic from Existing Design Patterns" for more information.
contexts	The places in the Design Studio user interface where the design pattern is accessible. You can define multiple contexts for a design pattern to enable users to access the design pattern from multiple places in the application. See "Defining Context for Design Patterns" for more information.
projects	The valid target projects into which resources are saved when the design pattern is applied. See "Defining Target Projects for Design Patterns" for more information.

Table 2-1 (Cont.) pattern.xml Sections

Element	Description
tokens	Placeholders that represent information to be entered by the user applying the design pattern. The information entered by the user customizes the resources in the manifest when the design pattern is applied. Tokens can be embedded in the target locations of resources in the manifest, in text documents that will be copied to a user's workspace, and in other tokens. See "Working with Tokens" for more information.
tokenGroups	The pages that appear in the Design Pattern wizard. Design pattern tokens are organized as pages in the Design Pattern wizard, where each page is a token group. You can define any number of token groups in a design pattern, and each token is associated with a single token group. See "Defining Token Groups for Design Patterns" for more information.
manifest	The list of resources included in the design pattern. The design pattern copies these resources to a user's workspace when the design pattern is applied. You can include in a manifest any type of resource that is valid in an Eclipse workspace. See "Defining the Manifest for Design Patterns" for more information.
actions	The actions that a user can perform on resources or inputs, such as creating relationships between or adding data elements to inputs or resources. See "Working with Design Pattern Actions" for more information.
inputs	The default information provided to the Design Pattern wizard to automatically populate token names. For example, if a design pattern creates actions for services, when a user runs the design pattern they can specify an existing service entity as input, and the design pattern uses that entity to generate the action. See "Defining Inputs for Design Patterns" for more information.
customActions	The custom actions that you define to call custom Java code from a design pattern. See "Defining Custom Actions" for more information.

Figure 2-3 displays part of an example **pattern.xml** file in the Design Studio default XML editor:

Figure 2-3 pattern.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<DesignPattern xmlns="http://oracle.communications/cgbu/sce/pattern/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="oracle.communications/cgbu/sce/pattern/DesignPatte
    id="reserveEquipment"
    name="Reserve Equipment">
    <version>
        <major>1</major>
        <minor>0</minor>
        <maintenance>0</maintenance>
    </version>
    <description>&lt;b&gt;Reserve Equipment&lt;/b&gt;&lt;br&gt;&lt;br&gt;
    Explanation of the design pattern goes here.&lt;br&gt;&lt;br&gt;
    Remember to escape special characters if using html tags.</description
    <contexts>
        <context>
            <scope>project</scope>
            <id>OSM</id>
            <id>UIM</id>
        </context>
        <cheatSheet>cheatSheet</cheatSheet>
    </contexts>
    <projects>
        <project id="reserveEquipmentOSMProject" name="Reserve Equipmen
            <description>Reserve Equipment Function Project</descriptio
            <typeId>OSM</typeId>
        </project>
        <project id="reserveEquipmentUIMProject" name="Reserve Equipmen
            <description>Reserve Equipment Inventory Project</descripti
            <typeId>UIM</typeId>
        </project>
    </projects>

    <tokens>
        <token id="equipment" name="Equipment" tokenGroup="reserveEquipme
```

Developing Custom Design Patterns

You create custom design patterns using groups of resources identified from an existing reference implementation or from a sample solution. After you create a design pattern, you package and distribute it to solution design teams. Solution design teams can install a design pattern as a Design Studio feature and, using a wizard, apply the pattern to their workspace.

To develop a custom design pattern:

1. Create and switch to a clean workspace.

   See the Design Studio Help for information about switching workspaces.

2. Create a design pattern plug-in project.

   See "Creating Design Pattern Plug-in Projects" for more information.

3. Model the design pattern.

   You model design patterns in the **pattern.xml** file. See "Modeling Design Patterns" for more information.

4. Create custom Java code.

   See "Invoking Custom Java Code from Design Patterns" for more information.

5. Build the design pattern plug-in project.

   A successful project build indicates that the design pattern is built and ready for testing. See the Design Studio Help for more information about building projects.

6. Test the design pattern.

   See "Testing Design Patterns" for more information.

7. Package the design pattern project.

   See "Packaging Plug-in Projects" for more information.

8. Distribute the design pattern project.

   See "Distributing Plug-in Projects" for more information.

9. Instruct team members to apply the design pattern.

   See "Applying Design Patterns" for more information.

Creating Design Pattern Plug-in Projects

You can create design pattern plug-in projects using the **Design Pattern Development** design pattern. This design pattern creates the basic structure for design pattern model files.

To create new design pattern plug-in projects:

1. Verify that the **Oracle Communications Design Studio Design Pattern Feature** is installed.

   Contact your system administrator if this feature is not available from your organization's update site.

2. In Design Studio, from the **Studio** menu, select **Design Pattern**.

   The Design Pattern dialog box appears.

3. Expand the **Others** folder and then expand the **Design Pattern Development** folder.

4. Select **Design Pattern Development** and then click **Next**.

   The Design Pattern wizard Introduction page appears.

5. Read the information on the Information page, and then click **Next**.

   The Select Project page appears.

6. Select an existing plug-in project to be used for the design pattern development or create a new design pattern plug-in project.

   To create a new project:

   a. Click **New**.

      The New Project page appears.

b. Expand the **Plug-in Development** folder and select **Plug-in Project**.

c. Click **Next**.

   The Plug-in Project page appears.

d. In the **Project Name**, enter a name for the design pattern plug-in project.

e. Accept the default values for the remaining fields or replace the default values with information specific to your installation.

f. Click **Next**.

   The Content page appears.

g. Accept the default values for the remaining fields or replace the default values with information specific to your installation.

h. Click **Finish**.

   Design Studio prompts you to open the Plug-in Development perspective. Oracle recommends that you develop design patterns using this perspective. Click **Yes** to switch to the Plug-in Development perspective.

   Design Studio populates the **Design Pattern Plug-in Project** field with the project you created.

7. Click **Next.**

   The Plug-in Information page appears.

8. Enter all required information, and then click **Next**.

   The Summary page appears.

9. Review the summary information, and then click **Finish**.

   Design Studio populates the project with information necessary to build a design pattern. The information includes a manifest, a resource directory, and all plug-in-related configuration for the packaging of the design pattern. Design Studio opens the Design Pattern Development cheat sheet in the Help view.

10. In the Design Pattern Development cheat sheet, click the **Click to Begin** link and complete the steps in the cheat sheet.

    For example, the cheat sheet steps help you with tasks such as copying resources to the project, populating the design pattern manifest, building the design pattern, testing, and distributing the design pattern.

Modeling Design Patterns

Before you can begin to model the design pattern content, identify required resources from a reference implementation or a sample solution and copy the required resources to the location of your design pattern in your plug-in project.

To model design patterns:

1. Review the design pattern configuration data.

   See "Reviewing Design Pattern Configuration Data" for more information.

2. Open the design pattern **pattern.xml** file.

> **Note:**
>
> Use the Design Studio default XML editor to edit the **pattern.xml** file. This default editor includes a **Design** tab with right-click context menu options and a **Source** tab to view the XML. Additionally, the default XML editor includes content assistance to help with tag completion and tag documentation.
>
> For detailed information about the elements and attributes in the **pattern.xml** file, see the Design Pattern XML schema, which is named **DesignPattern.xsd** and is located in the Design Studio software package.

To open the **pattern.xml** file:

a. Open the Plug-in Development perspective. See the Design Studio Help for information about opening perspectives.

b. Click the **Package Explorer** tab.

c. Navigate to the location of your design pattern.

d. In the design pattern root directory, double-click the **pattern.xml** file.

   The **pattern.xml** file opens in an XML editor.

3. Add a description of the design pattern that displays on the design pattern Introduction page.

   See "Defining a Description for a Design Pattern" for more information.

4. (Optional) Enable designs patterns to leverage the logic defined in other design patterns.

   See "Leveraging Logic from Existing Design Patterns" for more information.

5. Define the context for the design pattern.

   See "Defining Context for Design Patterns" for more information.

6. Define the types of projects into which the design pattern resources can be saved.

   See "Defining Target Projects for Design Patterns" for more information.

7. Define the tokens for the design pattern.

   See "Defining Tokens for Design Patterns" for more information.

8. Define the token groups for the design pattern.

   See "Defining Token Groups for Design Patterns" for more information.

9. Define the manifest for the design pattern.

   See "Defining the Manifest for Design Patterns" for more information.

10. Define actions for the design pattern.

    See "Defining Actions for Design Patterns" for more information.

11. Define custom actions for the design pattern.

    See "Defining Custom Actions" for more information.

12. Define the inputs for the design pattern.

See "Defining Inputs for Design Patterns" for more information.

13. Define a description for the design pattern.

    See "Defining a Description for a Design Pattern" for more information.

14. (Optional) Write a new cheat sheet for the design pattern.

    See "Working with Cheat Sheets" for more information.

Reviewing Design Pattern Configuration Data

Before configuring design patterns, review the **MANIFEST.MF**, **build.properties**, and **plugin.xml** files. The **MANIFEST.MF** file is in the **META-INF** directory. The **build.properties** and **plugin.xml** files can be found at the root of the project. These files contain dependencies to the required packages needed for design pattern development. They also include the registration and specific build properties required to build and package your design pattern.

> **Note:**
>
> When you are working on design patterns, many of the artifacts are not visible in standard Design Studio views. Use the Plug-in Development perspective and the Package Explorer view when working with design patterns. See the Design Studio Help for information.

If you are configuring the plug-in project manually (and not creating the design pattern plug-in project using the **Design Pattern Development** design pattern), ensure that you configure the data noted in the following files:

- **MANIFEST.MF**

  The name and version of the project and the required plug-ins. The **oracle.communications.sce.pattern.core** plug-in must be named as a dependency or your design patterns will not be visible when your plug-in is installed.

- **build.properties**

  The content that will be visible to other plug-ins after you have packaged your project. If you intend to locate design patterns in other folders, you must manually configure the location in the **bin.includes** section of the document.

- **plugin.xml**

  The design patterns contained by this plug-in. The **plugin.xml** includes an extension point named **oracle.communications.sce.pattern.core.designPattern**. This extension point has one entry in the section that points to the location of the pattern. You can add additional entries to package multiple design patterns within a single plug-in project. The **Name** and **ID** fields of the registration of a design pattern are not visible to Design Studio users. Rather, they are used only to add clarity to the registration. The **ID** field must be unique.

You can edit the content of these files using the Plug-in Manifest editor. See the *Eclipse Plug-in Development Environment Guide* for more information about using the Plug-in Manifest editor.

Defining a Description for a Design Pattern

You can provide a description of a design pattern to describe what the design pattern does when users apply the pattern, information that users require before running a design pattern, and what is required from users after applying the pattern. The description appears on the first page of the Design Pattern wizard.

To add a description to a design pattern:

1. With the **pattern.xml** file open in the Design Studio default XML editor, click the **Source** tab.
2. In the **description** element, provide a description of the design pattern.

   Escape special characters if you use HTML tags.
3. (Optional) Insert images into the description.

The following is an example of the **description** element:

```
<description>
   &lt;b&gt;dp1title&lt;/b&gt;&lt;br&gt;&lt;br&gt;Enter explanation of the
     design pattern here.&lt;br&gt;&lt;br&gt;Escape special characters if
     using HTML tags.
   &lt;img
      src="images\sampleImage.jpg"
      alt="This is a sample image"
      style="width:128px;height:128px;display: block;margin-left:
             auto; margin-right: auto;"
      align="middle"&gt;&lt;
   /img&gt;
</description>
```

Inserting Images into Description Elements

You can include images on a design pattern Introduction page by inserting the image in the **description** element of the **pattern.xml** file.

To insert images into the Introduction page, you define an **img** attribute with the following child attributes:

- Use the **src** attribute to specify the location of the image.
- Use the **alt** attribute to specify alternate text for an image. For example, you can use this attribute to insert text if the original image cannot be displayed because of slow connections or errors in the **src** attribute. This attribute is optional.
- Use the **style** attribute to specify the size of the image. Also, you use this attribute in conjunction with the **align** attribute if you want more control over the image alignment. This attribute is optional.
- Use the **align** to specify the alignment of an image on the page. You can align the image with the left, middle or right of the page. If you don't define an alignment, the image is left-aligned. This attribute is optional.

> **Note:**
>
> Inserting a high-resolution image may produce a horizontal scroll in a browser. An HTML limitation prevents the appearance of the horizontal scroll if the image is right-aligned.

You can also add images on the design pattern Introduction page by inserting a CDATA section:

```
<description>
   &lt;b&gt;dp1title&lt;/b&gt;&lt;br&gt;&lt;br&gt;Enter explanation of the
   design pattern here.&lt;br&gt;&lt;br&gt;Escape special characters if using
   HTML tags.

   <![CDATA[
        <img src="images\HomePhoneService.jpg" alt="Home Phone Service
            image is not available" style="width:128px;height:128px;display:
            block;margin-left: auto; margin-right: auto;" align="middle"></img>
   ]]>
</description>
```

Leveraging Logic from Existing Design Patterns

You can create a design pattern, called a composite design pattern, that leverages the logic of existing design patterns and combines that logic with its own configuration. The ability to share logic among design patterns enables you to define common logic in a single design pattern and leverage that logic, as required. When users run a composite design pattern, the design pattern presents all of the fields, pages, and custom logic defined in all of the leveraged design patterns.

About Composite Design Patterns

When creating and running composite design patterns, consider the following:

- You can leverage logic from any valid design pattern installed in the workspace.
- You can leverage logic only from the following design pattern elements: **manifest**, **projects**, **tokenGroups**, **tokens**, **actions**, **inputs**, and **customActions**.
- When an element (such as a **project** element, **token** element, and so forth) defined in a composite design pattern is defined with an **id** attribute that is identical to that defined for an element in a leveraged design pattern, the element defined in the composite design pattern is used.
- If identical element **id** attribute values exist in multiple leveraged design patterns, the value defined in the design pattern that appears first in composite design pattern **includes** element list is used.
- When a user runs a composite design pattern, pages (**tokenGroup** elements) defined in the composite design pattern appear before all pages defined in the leveraged design patterns. When the composite design pattern leverages logic from multiple existing design patterns, the pages defined in the leveraged design patterns appear in the order defined by **includes** element list in the composite design pattern.
- When a user runs a composite design pattern, the tokens defined in the composite design pattern appear before all tokens defined in a leveraged design pattern. When the composite design pattern leverages logic from multiple existing design patterns, the

tokens from the leveraged design patterns appear in the order defined by **includes** element list in the composite design pattern.

Defining the <includes> Element List:

When creating a composite design pattern, you define an **includes** element in the **pattern.xml** file to which you add the list of design patterns that contribute logic.

To define an **includes** element in the **pattern.xml** file:

1. With the **pattern.xml** file open in the Design Studio default XML editor, click the **Source** tab.
2. Add an **includes** element, and then add one or more child **include** elements.
3. Define the following attributes for each **include** element:
   - For the **id** attribute, enter a value that uniquely identifies the leveraged design pattern.
   - For the **pattern** attribute, enter the unique identifier of the design pattern from which this design pattern leverages logic.

The following is an example of the **includes** element:

```
<includes>
   <include id="Add_New_RFS_to_Tech_Domain" pattern="addRfstoExistingRfs" />
   <include id="Add_New_Resource_to_Tech_Domain" pattern="addResourceToRfs" />
</includes>
```

Defining Context for Design Patterns

You define the context for a design pattern to specify where in the user interface the design pattern will be accessible and to group design patterns in the Design Pattern Selection dialog box.

Because there are a large number of design patterns delivered with Design Studio, and because individual users can add to their environments additional custom design patterns developed post-market, defining context for your design patterns provides guidance to users by narrowing the scope of the design patterns that are available for a task.

To define context for design patterns:

1. With the **pattern.xml** file open in the Design Studio default XML editor, click the **Source** tab.
2. Add a **contexts** element and then add a **context** child element.
3. In the **context** child element, add a **scope** element.

   The **scope** identifies where in Design Studio the design pattern will be accessible.

   Enter one of the following values:

   - **project**: use this value to make the design pattern accessible when a project is selected in the Studio Projects view.
   - **entity**: use this value to make the design pattern accessible when an entity is selected in the Studio Projects view or Solution view.
   - **folder**: use this value to make a design pattern accessible when a folder is selected.

- **category**: use this value to create an arbitrary grouping of design patterns that appear in the Design Pattern wizard (in the **other** folder).

4. Add an **id** element and enter a value to filter the scope of the context.

   For example, if you defined the **scope** as **entity**, then you specify the types of entities in the **id** element.

   Use the following values in the **id** element:

If you defined scope as:	Use the following values:
project	• **OSM**, to filter for Order and Service Management projects. • **OSM_COMP**, to filter for Order and Service Management Composite projects. • **UIM**, to filter for Inventory projects. • **ACT**, to filter for Activation projects. • **ACT_IPSA**, to filter for Activation IPSA projects. • **ACT_SRT**, to filter for Activation SRT projects. • **NI**, to filter for Network Integrity projects. • **MODEL**, to filter for Model projects.
entity	Use the file extension defined for the type of Design Studio entity for which you want to filter. For example, to filter for an OSM manual task, define the **id** element as **manualTask**.
folder	Define the **id** element with the file extension of the entity and the folder name, separated by a period. For example: **productClass.fulfillmentPlan**
category	Define the **id** element with the text to be used as a category heading in the Design Pattern wizard.

5. (Optional) Add a **cheatSheet** element and define the resource ID to identify the cheat sheet in the manifest.

   The cheat sheet must be defined as a **resource** in the manifest element and the resource **id** element must be defined. The **cheatSheet** element must reference the resource **id** element. Also, you can define the cheat sheet element with a token.

   After the design pattern completes, design studio launches the cheat sheet in the Cheat Sheets view. You can use cheat sheets to assist users with manual configuration required after the design pattern completes, or to display the user-specific resources that were created by the design pattern.

   See "Working with Cheat Sheets" for more information.

The following is an example of the **contexts** element:

```
<contexts>
   <context>
      <scope>project</scope>
      <id>OSM</id>
      <id>UIM</id>
   </context>
   <cheatSheet>cheatSheet</cheatSheet>
</contexts>
```

Defining Target Projects for Design Patterns

When a design pattern is applied, each resource in the design pattern is copied to a single target project in a workspace. The **projects** element enables you to specify the number and

type of target projects required by your design pattern. The user applying the design pattern can select an existing target project or create a new one. The type of project the user can select or create is limited to the types that you specify in the design pattern.

To define target projects for design patterns:

1. With the **pattern.xml** file open in the Design Studio default XML editor, click the **Source** tab.
2. Add a **projects** element, and then add a child **project** element.
3. Define the following attributes for the **project** element:
   - For the **id** attribute, enter a value that uniquely identifies the project in a design pattern. You use this value if you need to make a reference to the project.
   - For the **name** attribute, enter the value that appears in the Design Pattern wizard for the project field.
4. (Optional) Add a **tokenGroup** attribute and enter the name of the token group to which this project belongs.

   A token group represents a page in the design pattern. You can group project elements in a group so that a project field appears on a specified page in the wizard.

   If you do not define a token group, project tokens appear on the default project token page, which appears to the user following the Introduction page.
5. (Optional) Add the **Sealed** attribute.

   Add the **Sealed** attribute to determine whether the design pattern creates a sealed or unsealed project, or whether to enable the user to determine the status after applying the design pattern. Define the **Seal** attribute with one of the following values:
   - **ALWAYS**, to indicate that the design pattern creates a sealed project.
   - **NEVER**, to indicate that the design pattern creates an unsealed project. This is the default value.
   - **OPTIONAL**, to add a check box to the Design Pattern wizard that enables the user to seal the project after the design pattern completes.
6. (Optional) Add a **defaultValue** element and enter the information that appears initially for the project field in the Design Pattern wizard.

   Users applying the design pattern can override default values. You can embed tokens in the **defaultValue** element, except for conditional tokens. See "Example: Defining Tokens as Default Values" for more information.

   If the user running the design pattern does not have a Project entity with the default name defined in the workspace, then the user can click the **Next** button to create a project with the default value name.

   If you are defining a default value for an Activation Service cartridge, you must also add the **domain** child element and the **service** child element. If you are defining a default value for an Activation Network cartridge, you must also add the **vendor**, the **technology**, and the **softwareLoad** child elements.
7. (Optional) Add a **regularExpression** element.

   You can use regular expressions to enforce naming conventions. The value that you define in the **message** element is displayed to the user if the condition that

you define in the **expression** element is not satisfied. See "About Regular Expressions" for more information.

8. Add a **typeID** element and enter a value.

   The **typeID** element defines a project type to which this project belongs. Define **typeID** with one of the following values:

   - **OSM**, to associate the design pattern with Order and Service Management projects.
   - **OSM_COMP**, to associate the design pattern with Order and Service Management Composite projects.
   - **UIM**, to associate the design pattern with Inventory projects.
   - **ACT**, to associate the design pattern with Activation projects.
   - **ACT_IPSA**, to associate the design pattern with Activation IPSA projects.
   - **ACT_SRT**, to associate the design pattern with Activation SRT projects.
   - **NI**, to associate the design pattern with Network Integrity projects.
   - **MODEL**, to associate the design pattern with Model projects.
   - **OTHER**, to associate the design pattern with a project type not listed above (for example, with a Java or plug-in project).

9. Add a **description** element and enter a description of the project.

   The description appears in the Design Pattern wizard as embedded Help.

10. (Optional) Add a **condition** element and define a condition to dynamically control the visibility of the project field in the Design Pattern wizard.

    See "About Token Conditions" for more information.

11. (Optional) Add a **modelVersion** element.

    Defining a model version assures compatibility with the solution loaded in the user workspace by limiting the projects to only those with a specific set of target version values. A project target version specifies the version of the server to which a cartridge project is deployed. For example, if you define the **modelVersion** as **7.3.0**, then a user running the design pattern can select only projects with target versions defined as **7.3.0**.

12. (Optional) Add a **projectVersion** element.

    Defining a project version assures compatibility of the design pattern with a user development environment, ensuring that users cannot run the design pattern in an incompatible version of Design Studio. The version number of a project is defined in the **.studio** file, located at the root of a project.

See "Example: Adding Project Dependencies" for an example of a design pattern that creates a dependency between a new project and an existing base project. The following is an example of the **projects** element:

```
<projects>
    <project name="Activation Network Cartridge"
             id="DesignPatternProject"
             tokenGroup="ProjectPage"
             seal="ALWAYS">
        <description>Create a first project</description>
        <condition>
           <equals v2="@@SelectProject@@" v1="true" />
        </condition>
        <typeId>ACT</typeId>
```

```
    <defaultValue>COMVERSE_MMS</defaultValue>
    <vendor>CMVT</vendor>
    <technology>@@technology@@</technology>
    <softwareLoad>3-5-X</softwareLoad>
</project>
```

Working with Tokens

The outcome of a design pattern is affected by the data that a user enters into the Design Pattern wizard. When creating design patterns, you use a token to represent each piece of data that you expect a user to add to the wizard. Tokens are placeholders that represent information to be collected by the Design Pattern wizard from a user applying a design pattern. You use tokens to ensure that the resources a design pattern copies to a workspace are based on information supplied by the user who applies the design pattern. You define tokens for resources in **pattern.xml** files.

You can use tokens to:

- Append the value of a token to the name of a Design Studio entity.
- Influence where in a target project an entity will be created by including the value of a token in the location path.
- Influence the actions that create relationships, populate data elements, or otherwise modify existing Design Studio entities.
- Automate design pattern decisions with the use of conditions.
- Provide default values to other tokens. You can use tokens in the **defaultValue** element of other tokens to provide intelligent defaults based on previously collected information.
- Customize the content of files.

About Tokens

Token elements include an **id** attribute that uniquely identifies the token in a design pattern. To reference a token **id** in another element in a design pattern, you use the following syntax:

@@id@@

where *id* is the value that you defined for the token **id** attribute.

Token elements also include a **name** attribute, which appears in the Design Pattern wizard as the field name, and a **description** attribute, which appears in the Design Pattern wizard as embedded Help under the field.

The following example includes a token called **service** in a resource **targetLocation** element:

```
targetLocation="/src/oracle/communications/services/@@service@@/
@@service@@Interface.java"
```

If the user applying the design pattern enters the value **VOIP** for the **service** token, the **targetLocation** element would appear as:

```
targetLocation="/src/oracle/communications/services/VOIP/VOIPInterface.java"
```

> **Note:**
>
> Design Studio cannot detect embedded token name spelling errors. Check for consistent token usage to ensure tokens are replaced properly. Design Studio detects invalid tokens when a user attempts to run design pattern.

For a more detailed description of token types and their configuration, see the Design Pattern XML schema (**schema/DesignPattern.xsd**).

About Token Types

Token types represent the type of data that you expect a design pattern user to supply in the Design Pattern wizard. For example, a user may be expected to select from a list of valid projects, to select or deselect a check box, or to specify an entity name with a valid string of characters. When you create design patterns, the type of token that you specify determines what child elements are available to define for the token element.

Define tokens with one of the following token types:

- **StringToken**, if you want the user to enter a string of text. The text accepted by the token can be constrained by configuring regular expressions and by defining a **maxLength** value to limit the string to a maximum number of characters. The Design Pattern wizard displays each **StringToken** as a single-line text input field.
- **NumericToken**, if you want the user to enter numeric input. The number accepted by the token can be constrained by configuring regular expressions and by defining a **minValue** and a **maxValue**. The Design Pattern wizard displays each **NumericToken** as an input field with up and down arrows that can be used to increase or decrease the value.

  > **Note:**
  >
  > When defining numeric tokens, Oracle recommends that you define the minimum and maximum values for the token.

- **BooleanToken**, if you want the user to select a **true** or **false** value. Boolean tokens are displayed in the Design Pattern wizard as check boxes. When a token type is defined as a Boolean, the design pattern ignores any values defined in the **regularExpression** element. You can define the initial state of the check box by defining the **defaultValue** element as **true** or **false**.
- **EnumerationToken**, if you want the user to select from a set of preconfigured choices that you define using the **values** tag. Enumeration tokens appear in the Design Pattern wizard as lists, from which the user must select a value. You can define a default value from the list of values.
- **EntityRefToken**, if you want the user to select a Design Studio entity in a workspace. The Design Pattern wizard displays a **Select** button next to entity reference token fields, which enables a user to select an existing entity of a specific type from the workspace. See "About Entity Reference Tokens" for more information.
- **ElementRefToken**, if you want the user to select an element, such as a Data Dictionary element, in a workspace. Design patterns can use element reference tokens to associate entities created by a design pattern with elements that exist in the workspace. The

Design Pattern wizard displays a **Select** button next to element reference token fields, which enables the user to select an existing element from the workspace. See "About Element Reference Tokens" for more information.

About Entity Reference Tokens

You can use entity reference tokens to represent the name of a Design Studio entity in a workspace. An entity reference token enables a user to select an existing entity of a specific type from the workspace. **EntityRefToken** types include the following child elements:

- **entityType**: Use to restrict the type of entity that a user can select. You specify the type by defining this element with the file extension for an entity type.
- **projectFilter**: Use to restrict the type of entities that a user can select. This element includes two mandatory child elements, **filterType** and **id**. For the **filterType** element, select the value **project** or **entityRef**. For the **id** child element, enter the unique identifier of the **project** element or of the **EntityRefToken** as defined in the **pattern.xml** file. Depending on the mandatory element values, the project is derived at run time and the user is restricted to select entities from this derived project.
- **entityFilter**: Use to further restrict the entities available for selection. This element includes one mandatory attribute, **filterType**, for which you can define one of the following properties: **entityName**, **entityReference**, or **referencedByEntity**. Depending on the property that you specified, you can define additional child element criteria. See "About EntityRefToken Filtering" for more information.
- **relationship**: Use to construct the reference to the entity. You can determine the relationship type by viewing the XML file of an entity where the reference is embedded.
- **allowMultiple**: Use to enable design pattern users to select multiple entities from a list. When you define this child element with the value **true**, design pattern users can select multiple existing entities of the specified **entityType** from the workspace. This element is optional. The default value of this element is **false**, which limits a selection to one existing entity of the specified entity type from the workspace. See "Example: Supporting Multiple Selections for Entity Reference Tokens" for more information.

Restrictions that you define using regular expressions apply to the entity name.

You can use entity properties to enable a design pattern access to information about the entity that is bound to an **EntityRefToken**. See "About Entity Properties" for more information.

You can define a default value for an **EntityRefToken**. When a design pattern is run, the pattern validates the default value to ensure that the value exists in the workspace. If an entity with the default value name does not exist in the workspace, the Design Pattern wizard displays a validation error and disables the **Next** button. In this scenario, the user must select a different value.

Example 2-1 illustrates a token defined with a type of **EntityRefToken**.

Example 2-1 EntityRefToken

```
<token name="Service Order" tokenGroup="HelloWorldInfo"
   id="Order" xsi:type="EntityRefToken">
   <description>Select the service order defined for
               your solution.</description>
```

```
   <allowMultiple>true</allowMultiple>
   <entityType>order</entityType>
   <relationship>com.mslv.studio.provisioning.task.manual.orderType</relationship>
   <defaultValue>VoiceMailService</defaultValue>
</token>
```

About EntityRefToken Filtering

When defining entity reference tokens, you can specify an entity type to limit the available selections to the user running a design pattern. You can define additional filtering criteria by using the **entityFilter** child element.

The **entityFilter** element includes the property **filterType**, which you can define using the following values:

- Use **entityName** to limit the selection based on an entity name. This filter option supports additional child elements, such as **equals**, **notEquals**, **beginsWith**, **endsWith**, and **contains**.
- Use **entityReference** to limit the selection to entities with a reference to the entity type defined in the **entityType** child element and to those entities that meet the **entityName** criteria.
- Use **referencedByEntity** to limit the selection to entities that are referenced by the entity specified in the **entityType** element and to those that meet the **entityName** criteria.

Example 2-2 demonstrates how to limit the selection to all Domain entities with name **Line**.

Example 2-2 Filtering Entity Reference Tokens Using the entityName Property

```
<token id="domain1" name="Domain Entity Name Line"
          tokenGroup="PSRModelInfo"
          xsi:type="EntityRefToken">
     <description>Choose a Domain entity of Name Line</description>
     <entityType>cmnDomain</entityType>
     <entityFilter filterType="entityName">
          <entityName>
               <equals>Line</equals>
          </entityName>
     </entityFilter>
</token>
```

Example 2-3 demonstrates how to limit the selection to all Domain entities that contain references to Functional Area entities that are named **Service**.

Example 2-3 Filtering Entity Reference Tokens Using the entityReference Property

```
<token id="domain1" name="Service Domain"
          tokenGroup="PSRModelInfo"
          xsi:type="EntityRefToken">
    <entityType>cmnDomain</entityType>
    <entityFilter filterType="entityReference">
         <entityType>funcArea</entityType>
         <entityName>
              <equals>Service</equals>
         </entityName>
    </entityFilter>
</token>
```

Example 2-4 demonstrates how to limit a selection to all the Customer Facing Service entities that are referenced by Domain entities with a name that begins with **Line**.

Example 2-4 Filtering Entity Reference Tokens Using the referencedByEntity Property

```
<token id="domain1" name="Service Domain"
           tokenGroup="PSRModelInfo"
           xsi:type="EntityRefToken">
     <entityType>custSrvc</entityType>
     <entityFilter filterType="referencedByEntity">
          <entityType>cmnDomain</entityType>
          <entityName>
               <beginsWith>Line</beginsWith>
          </entityName>
     </entityFilter>
</token>
```

About Entity Properties

You use entity properties to enable the design pattern to acquire information from conceptual model entities that are bound to an entity token. Using a token substitution string, you can instruct the design pattern to acquire the value entered for the entity token.

Use the following format:

@@tokenid.propertyName@@

where *tokenid* is the **id** attribute defined for the token, and *propertyName* is the supported property.

Design Studio supports the use of entity properties for **project** and **resourceType** properties for all entity types, and for **implementationMethod** and **implementationSystem** properties for conceptual model entities that support realization.

In Example 2-5, the design pattern queries the **Implementation Method** and the **Implementation System** properties of the entities that are selected for the tokens with IDs defined as **resourceEntity**. The design pattern compares the properties to the value **DEVICE**. If both properties evaluate to true, the design pattern adds the resource to the workspace.

Example 2-5 Entity Properties

```
<resource id="resImpl" overwritePolicy="NEVER">
   <condition>
      <All>
         <equals v1="@@resourceEntity.implementationMethod@@" v2="DEVICE"/>
         <equals v1="@@resourceEntity.implementationSystem@@" v2="UIM"/>
      </All>
   </condition>
```

About Element Reference Tokens

You can use element reference tokens to represent the name of a data element defined in a workspace. An element reference token enables a user to select an existing data element from the workspace.

You can use element reference tokens to embed a reference to the select element in the other Design Studio entities.

ElementRefToken types include the following child elements:

- **entityType**: Use to restrict the type of entity for data element selection. You specify the type by defining this element with the file extension for an entity type.
- **elementType**: Use to restrict the type of elements for selection. An **elementType** is defined with an ID, for example **com.mslv.studio.provisioning.order.node** or **com.mslv.studio.core.data.dictionary.node**. You can determine the element type by viewing the XML file of an entity where the reference is embedded.
- **projectFilter**: Use to restrict the type of entities that a user can select. This element includes two mandatory child elements, **filterType** and **id**. For the **filterType** element, select the value **project** or **entityRef**. For the **id** child element, enter the unique identifier of the **project** element or of the **ElementRefToken** as defined in the **pattern.xml** file. Depending on the mandatory element values, the project is derived at run time and the user is restricted to select elements from this derived project.
- **relationship**: Use to construct the reference to the entity. You can determine the relationship type by viewing the XML file of an entity where the reference is embedded.

Example 2-6 illustrates a token defined with a type of **ElementRefToken**.

Example 2-6 ElementRefToken

```
<token name="Data Element for Order Template" tokenGroup="HelloWorldInfo"
   id="OTDataRef" xsi:type="ElementRefToken">
   <description>Select a data element for the order template.</description>
   <entityType>order</entityType>
   <elementType>com.mslv.studio.provisioning.order.node</elementType>
   <relationship>ora.task.orderTemplateRef</relationship>
</token>
```

Element reference tokens return XML instead of a simple value (and therefore, you cannot embed element references in default values or in target locations). Design Studio returns XML to ensure that the token can be embedded in a Design Studio entity document, where it replaces a reference to an element.

The format of the return XML is:

```
<com:entity>--from selection--</com:entity>
<com:entityType>--from selection--</com:entityType>
<com:relationship>--from token configuration--</com:relationship>
<com:element>--from selection--</com:element>
<com:elementType>--from token configuration--</com:elementType>
```

Use the token to replace a section of XML in a Design Studio entity document with the corresponding structure.

About Regular Expressions

When configuring design patterns, you can use regular expressions to ensure that the information a user enters for a token is valid. When using String tokens, Oracle recommends that you restrict the valid input using regular expressions, because some character-based input is not valid for use in a token definition. For example, if you use a token as a file name, then it must contain only characters that are valid for a file name. If you use a token as a Design Studio entity name or in an XML document, you must restrict the use of XML special characters. Embedded tokens are the most common place where errors are introduced into a design pattern. If a design pattern is not working properly, first ensure that all tokens are properly replaced when the pattern was applied.

Example 2-7 illustrates a how a regular expression can be used to ensure that a user enters a valid entity name. The example also includes a message that will appear to the user if they enter invalid values.

Example 2-7 Regular Expression

```
<regularExpression>
     <expression>^[a-zA-Z_][a-zA-Z0-9_-[.]]{0,255}$</expression>
     <message>Invalid value for Name. The first character should be
              an alphabetic character or underscore. The second and
              following characters can be alphanumeric characters,
              underscores, hyphens, and periods. The length should not
              exceed 255 characters.
     </message>
</regularExpression>
```

About Token Functions

You can use the following functions with any token (except for element reference tokens) in a design pattern:

- toUpper
- toLower

These functions enable design patterns to force token values to upper case or to lower case when required by naming conventions. Use the following format:

@@*tokenId*.toUpper@@

@@*tokenId*.toLower@@

where *tokenid* is the **id** attribute defined for the token.

> **Note:**
>
> You can not use entity properties and entity functions in the same token expression. See "About Entity Properties" for more information.

About Token Conditions

Conditions enable you to dynamically control the visibility of a token that appears in the Design Pattern wizard. You use conditions with tokens to optimize the user experience by displaying only the tokens relevant to the specific task and context from which the pattern is initiated.

When a design pattern is run, the Design Pattern wizard evaluates the conditions defined for the tokens. Additionally, the Design Pattern wizard evaluates conditions before each page in the wizard is displayed. This behavior enables you to control whether a token appears on a page based on user input values and entity selections made on previous pages.

Example 2-8 is a small piece of XML from a design pattern that creates a new Service specification entity in an Inventory project. The design pattern also optionally creates a reference to an existing Resource Facing Service (RFS) entity in a model project.

When initiated, the design pattern displays a check box (in this example, represented by the Boolean token **isRFS**) that a user selects if they want to add the optional RFS reference. If the user selects the check box (and the Boolean token evaluates to true), the design pattern displays a list of available RFS entities (in this example, represented by the entityRef token **rfsToken**). If the user does not select the check box, the design pattern does not display the list of RFS entities.

Example 2-8 Conditional Token

```
<token id="isRFS" name="Create RFS" tokenGroup="rfsDetails"
   xsi:type="BooleanToken">
        <description>Add a reference to an existing
                     Resource Facing Service entity.
        </description>
      <defaultValue>false</defaultValue>
</token>

<token id="rfsToken" name="Resource Facing Service" tokenGroup="taDetails"
   xsi:type="EntityRefToken">
       <description>The Resource Facing Service</description>
       <condition>
          <equals v1="@@isRFS@@" v2="true"/>
       </condition>
       <entityType>rsrcSrvc</entityType>
</token
```

Tokens referenced by conditions that evaluate to false do not appear in the Design Pattern wizard (and a user, therefore, has no opportunity to enter or select a value). A design pattern will display an error if a condition that references an EntityRefToken or ElementRefToken type evaluates to false and if the same EntityRefToken or ElementRefToken type is included in a subsequent condition definition of another token. Therefore, conditional tokens that support default values must specify a default value.

Conditions are optional for tokens, and condition statements can be arbitrarily complex.

Defining Tokens for Design Patterns

You define tokens to ensure that the resources a design pattern copies to a workspace are based on information supplied by the user who applies the design pattern. See "Working with Tokens" for more information.

To define tokens in a design pattern:

1. With the **pattern.xml** file open in the Design Studio default XML editor, click the **Design** tab.
2. Add a **tokens** element, and then add a child **token** element for each piece of information that the design pattern must obtain from a user.
3. Add and define values for the following token attributes:
   a. For the **name** attribute, enter the name of the field to appear in the Design Pattern wizard.
   b. For the **tokenGroup** attribute, enter the name of the token group to which this token belongs.

      A token group represents a page in the design pattern. You can group tokens in a group so that the tokens all appear on the same page in the wizard.

c. For the **id** attribute, enter a unique value to represent the token. You use the value in this attribute to reference the token in the design pattern. See "About Tokens" for more information.

4. In the **token** element, add a **type** attribute.

   See "About Token Types" for more information.

5. Add child elements to all token elements.

   Some elements are common to all tokens, and some are available only to specific token types.

   - For the **description** element, enter a description of the information being requested from the user.

     The description appears in the Design Pattern wizard and provides information or instructions to the user about the values that they must provide.

   - For the **defaultValue** element, enter the information that appears initially for the token in the Design Pattern wizard.

     You can embed other tokens in the **defaultValue** element, except for conditional tokens. Default values are optional. Users applying the design pattern can override default values. See "Example: Defining Tokens as Default Values" for more information.

     When applying design patterns, users can click the **Reset Page** button to reset all of a page's token values to their initial default values. When a user clicks the **Reset Page** button, all token values on the current page are reset to the default value. All token values on previous pages remain as defined by the user.

   - For the **regularExpression** element, define an expression to validate or restrict the token values entered by users in the Design Pattern wizard. Regular expressions are optional and may not apply to all token types. You can also define the message that appears to the user if the input text fails to conform to the regular expression. See "About Regular Expressions" for more information.

   - For the **value** element, define one of the enumerated values that appears in a list of enumeration tokens. This element appears for **EnumerationToken** types only.

   - For the **entityType** element, specify the type of Design Studio entity that a user must select. The value that you define here is determined by the file extension of the Design Studio entity. For example, to indicate that a user must select an OSM manual task, you define this element as **manualTask**. The **entityType** element appears for entity reference tokens only. See "About Entity Reference Tokens" for more information.

   - For the **elementType** element, specify the element ID for the type of element that the user must select. The **elementType** element appears for element reference tokens only. See "About Element Reference Tokens" for more information.

   - For the **relationship** element, specify the relationship type used in a reference.

   - For the **condition** element, define a condition to dynamically control the visibility of a token that appears in the Design Pattern wizard. See "About Token Conditions" for more information.

6. Add token functions to ensure that all characters in a token value are forced to upper case or to lower case.

   See "About Token Functions" for more information.

7. Click **Save**.

The following is an example of the **tokens** element:

```
<tokens>
   <token id="equipment" name="Equipment" tokenGroup="reserveEquipmentInfo"
       xsi:type="StringToken">
      <description>Name of the equipment you want to reserve.rn.</description>
      <regularExpression>
         <expression>[a-zA-Z0-9_]+</expression>
         <message>Reserve Equipment IDs should only contain letters, numbers and
                  underscores.</message>
      </regularExpression>
      <regularExpression>
         <expression>[a-zA-Z][a-zA-Z0-9_]*</expression>
         <message>Reserve Equipment IDs should start with a letter.</message>
      </regularExpression>
      <regularExpression>
         <expression>[a-zA-Z0-9_]{0,20}$</expression>
         <message>Reserve Equipment IDs should be 20 characters or less.</message>
      </regularExpression>
   </token>
</tokens>
```

See the following topics for examples that demonstrate token use:

- Example: Defining Tokens for Resources
- Example: Defining Tokens as Default Values
- Example: Defining Action Subjects or Participants With Values External to Design Patterns

Defining Token Groups for Design Patterns

A token group represents a page in the Design Pattern wizard. You organize design pattern tokens into pages to control the manner in which information is collected by the Design Pattern wizard. You can define any number of token groups in a design pattern, and each token is associated with a single token group.

You can use token groups, for example, to group related input fields together on a single Design Pattern wizard page or to limit the number of input fields on each page to improve usability.

To define token groups for design patterns:

1. With the **pattern.xml** file open in the Design Studio default XML editor, click the **Source** tab.
2. Add a **tokenGroups** element and then add a child **tokenGroup** element for each page that you want to appear in the Design Pattern wizard.
3. Add and define values for the following **tokenGroup** attributes:
   a. For the **name** attribute, enter the name of the page that will appear at the top of the Design Pattern wizard.

b. For the **id** attribute, enter a unique value to represent the token group. The **token** elements included in this group reference the value that you enter here.

4. In the **tokenGroup** element, add a child **description** element.
5. In the **description** element, enter the description of the page as it should appear in the Design Pattern wizard.
6. Click **Save**.

The following is an example of the **tokenGroups** element:

```
<tokenGroups>
   <tokenGroup name="Plug-in Information" id="reserveEquipmentInfo">
      <description>Define the information on this page to create reserved
                   equipment.</description>
   </tokenGroup>
</tokenGroups>
```

Defining the Manifest for Design Patterns

A design pattern manifest is the list of resources included in a design pattern. You define the manifest for a design pattern to determine how resources are generated and where they generated when the pattern is applied. A design pattern can generate Design Studio entities, cheat sheets, Java files, XML files, and so forth, in specified target projects.

When defining resources that you copied from an existing reference implementation, you can change the names of those resources to be less specific, which enables you to use the resources across different service domains. For example, if you copy a specification named **ADSL_Port** from a Broadband Internet reference implementation, you can rename the resource to **Assigned_Interface** for use in a design pattern that applies across service domains.To define the manifest in design patterns:

1. Identify all required resources from a reference implementation or a sample solution.
2. Open the **pattern.xml** file in the Design Studio default XML editor and click the **Source** tab.
3. Add a **manifest** element, and then add child **resource** elements.

   When adding child **resource** elements, you can:

   - Copy existing resources to your design pattern. You can copy resources from a reference implementation or from a sample solution or your can create new resource elements. You can copy any source file that you require in your solution, such as Design Studio entities, XML content, rules, and so forth.
   - Define new child **resource** elements for additional required resources.
4. In the **location** child element, enter the path and file name from the pattern root to the copied resource.

   You must retain the original name of all Design Studio entities in the source location of the resource (do not embed tokens). The location is the relative path of the resource from the location of the design pattern. For example, if the location of the design pattern is:

   /pattern/myPattern

   and one of the resources for the pattern from the project root is:

/pattern/myPattern/dataDictionary/pattern.xsd

then you define the value for the **location** element for the corresponding resource as:

dataDictionary/pattern.xsd

5. Append the extension **.pat** to all Design Studio entity file names in the **resources** element **location** element.

> **Note:**
>
> You must append the extension **.pat** to the Design Studio entity file names when working with Design Studio entity resources in a plug-in project. Resources that are not Design Studio entities, such as Java or XML files, do not need the **.pat** extension.
>
> For example, if you add to the resource list an Order and Service Management order called **myOrder.order**, you must rename that order entity to **myOrder.order.pat**.
>
> Design Studio refers to the original entity name while processing the design pattern to update references to entities in the pattern. Failure to append the **.pat** extension to the resource may result in problem markers and entries in your error log. When working with Data Dictionary companion files, Design Studio automatically deletes the resource from the workspace when it cannot resolve correctly to its associated schema file.

6. In the **resource** element, add an **id** attribute and enter an ID that is unique among all resources in the manifest.

   Components in the design pattern use the **id** attribute when referencing the resource.

7. In the **projectId** element, enter the ID of the project into which the design pattern should save the resource when the design pattern is applied.

> **Note:**
>
> The ID that you define here must also be defined in the **projects** element of the **pattern.xml** file, which describes the projects that the design pattern can place resources into when the design pattern is applied. See "Defining Target Projects for Design Patterns" for more information.

8. (Optional) In the **resource** element, add a **condition** child element.

   You use conditions to include simple or complex conditional logic when determining whether a design pattern adds the resource the workspace. See "About Conditions" for more information.

9. In the **resource** element, add a **targetLocation** child element.

10. In the **targetLocation** element, enter a path and file name of the location (in the user's workspace) where the design pattern-generated artifacts will be saved when the design pattern is applied.

    The path is relative to the target project.

The **targetLocation** element can contain embedded tokens. Embedding tokens enables the user applying the design pattern to influence the name and location of the generated resource, implement naming conventions, and maintain referential integrity across resources that are included in a design pattern. When run, the design pattern substitutes the token references in the path with the values entered by the user.

For example, the following **targetLocation** includes a declared token called **service** to be used in the path and name of a Java class resource:

```
targetLocation="/src/oracle/communications/services/@@service@@/
@@service@@Interface.java"
```

If the user applying the design pattern entered **VOIP** as the value for the **service** token, **targetLocation** would expand to:

```
targetLocation="/src/oracle/communications/services/VOIP/VOIPInterface.java"
```

and the design pattern copies the file into the workspace using this location and name.

11. In the **resource** element, add a **type** attribute.
12. In the **type** attribute, specify the type of resource to be copied to the user's workspace.

    Use one of the following values:

    - **TEXT**: use to identify the resource as a text file. You can use tokens in the content of text files. If you specify no resource type, the design pattern uses **TEXT** as the default value. For example, you can identify Design Studio entities, XML, XQuery, XSLT, and Java resources as **TEXT** types.
    - **DIRECTORY**: use to identify the resource as a directory that will be created in the user's workspace.
    - **BINARY**: use to identify the resource as a binary file (for example, a JPEG file). You cannot use tokens in binary resources.
    - **LIBRARY**: use to identify the resource as a Java library (for example, a JAR file). After the design pattern copies the library to the user's workspace, the design pattern adds the library to the classpath of the appropriate project.
13. In **resource** element, add an **overwritePolicy** attribute.
14. In the **overwritePolicy** attribute, specify the design pattern response if a resource with the same name and location exists in the target project.

    Use one of the following values:

    - **ALWAYS**: use to specify that the existing file is to be overwritten by the resource contained in the design pattern. However, if the resource in the workspace is read-only, the design pattern cannot overwrite the existing resource. The log produced after the design pattern is completed lists all resources that cannot be overwritten. **ALWAYS** is the default value.
    - **NEVER**: use to specify that the existing file is not to be overwritten by the resource contained in the design pattern. The design pattern performs no token substitutions for the resource when the **overwritePolicy** is defined with this value.
15. In **resource** element, add an **overrideDisplayName** attribute.

By default, the display name of a resource is generated from the resource name if the resource is Design Studio entity.

16. If you are using a token to define the display name of a resource, change the value of the **overrideDisplayName** attribute to **true**.

    Changing the value of the **overrideDisplayName** attribute to **true** prevents Design Studio from overriding the display name value with the Design Studio entity name.

17. Click **Save**.

The following is an example of the **manifest** element:

```
<manifest>

<resource id="cheatSheet">
   <location>cheatSheet/reserveEquipmentCheatSheet.xml</location>
   <targetLocation>cheatSheet/reserveEquipment/DesignPatternCreation.xml
   </targetLocation>
   <projectId>reserveEquipmentOSMProject</projectId>
</resource>

<resource id="dd_companion" >
   <location>dataDictionary/reserveHardware_companion.xsdc.pat</location>
   <targetLocation>
      dataDictionary/@@equipment@@/reserve@@equipment@@_ companion.xsdc
   </targetLocation>
   <projectId>reserveEquipmentOSMProject</projectId>
</resource>

<resource id="dd" >
   <location>dataDictionary/reserveHardware.xsd.pat</location>
   <targetLocation>dataDictionary/@@equipment@@/reserve@@equipment@@.xsd
   </targetLocation>
   <projectId>reserveEquipmentOSMProject</projectId>
</resource>

<resource id="automation" >
   <location>model/OSM/reserveHardware.automationTask.pat</location>
   <targetLocation>model/@@equipment@@/reserve@@equipment@@.automationTask
   </targetLocation>
   <projectId>reserveEquipmentOSMProject</projectId>
</resource>

<resource id="baseTask" >
   <location>model/OSM/reserveHardwareBaseTask.manualTask.pat</location>
   <targetLocation>model/@@equipment@@/reserve@@equipment@@BaseTask.manualTask
   </targetLocation>
   <projectId>reserveEquipmentOSMProject</projectId>
</resource>

<resource id="falloutTask" >
   <location>model/OSM/reserveHardwareFallout.manualTask.pat</location>
   <targetLocation>model/@@equipment@@/reserve@@equipment@@FalloutTask.manualTask
   </targetLocation>
   <projectId>reserveEquipmentOSMProject</projectId>
</resource>

<resource id="function" >
   <location>model/OSM/reserveHardwareFunction.orderComponentSpec.pat</location>
   <targetLocation>
      model/@@equipment@@/reserve@@equipment@@Function.orderComponentSpec
```

```
   </targetLocation>
   <projectId>reserveEquipmentOSMProject</projectId>
</resource>

<resource id="process" >
   <location>model/OSM/reserveHardwareProcess.process.pat</location>
   <targetLocation>model/@@equipment@@/reserve@@equipment@@Process.process
   </targetLocation>
   <projectId>reserveEquipmentOSMProject</projectId>
</resource>

<resource id="query" >
   <location>model/OSM/reserveHardwareQueryTask.manualTask.pat</location>
   <targetLocation>model/@@equipment@@/reserve@@equipment@@QueryTask.manualTask
   </targetLocation>
   <projectId>reserveEquipmentOSMProject</projectId>
</resource>

<resource id="role" >
   <location>model/OSM/reserveHardwareRole.rolePermissions.pat</location>
   <targetLocation>model/@@equipment@@/reserve@@equipment@@Role.rolePermissions
   </targetLocation>
   <projectId>reserveEquipmentOSMProject</projectId>
</resource>

<resource id="summary" >
   <location>model/OSM/reserveHardwareSummaryTask.manualTask.pat</location>
   <targetLocation>model/@@equipment@@/reserve@@equipment@@SummaryTask.manualTask
   </targetLocation>
   <projectId>reserveEquipmentOSMProject</projectId>
</resource>

<resource id="compositeView" >
   <location>model/OSM/reserveHardwareView.compositeCartridgeView.pat</location>
   <targetLocation>
      model/@@equipment@@/reserve@@equipment@@View.compositeCartridgeView
   </targetLocation>
   <projectId>reserveEquipmentOSMProject</projectId>
</resource>

<resource id="java" >
   <location>resource/reserveHardware.java</location>
   <targetLocation>src/oracle/communications/services/reservation/@@equipment@@/
                   Reserve@@equipment@@.java</targetLocation>
   <projectId>reserveEquipmentOSMProject</projectId>
</resource>

<resource id="equip" >
   <location>model/UIM/reservable.equipment.pat</location>
   <targetLocation>model/@@equipment@@/@@equipment@@.equipment</targetLocation>
   <projectId>reserveEquipmentUIMProject</projectId>
</resource>

</manifest>
```

Working with Design Pattern Actions

You define actions to enable design patterns to perform actions on entities, such as creating relationships between or adding data elements to inputs or resources.

Actions affect entities that exist in the target workspace, but the actions do not cause the existing entities to be replaced in the target workspace. This enables users who run design patterns and subsequently enrich the model to re-run patterns without losing the changes they make during an iterative design cycle.

Actions have two main components:

- A subject: An action subject is the entity or element that is affected by the action. The subject is the entity or element to which, for example, a reference is added or data elements are added.
- A participant: An action participant is the entity or element that provides the information for the action. For example, the participant is the entity providing the data elements to the subject.

About Action Elements

You include **action** elements in the **pattern.xml** file to define the action attributes. You define the following elements:

- **condition**: Add simple or complex conditional logic to enable the design pattern to perform the action only under specific conditions. See "About Conditions" for more information.
- **actionType**: Specify the type of action to be performed. See "About Design Pattern Action Types" for more information.
- **subject** and **participant**: Specify the entity or data element that is affected by the action, and the entity or data element that provides the information for the action, respectively. The **subject** and **participant** elements require you to define a:
  - **participantType**: Specify whether the subject or participant entity is a **resource** that is internal to the design pattern, or whether the subject or participant entity is **input** that the design pattern selects when the pattern is run. See "Defining Inputs for Design Patterns" for more information.
  - **id**: specify a unique identification for the subject and participant entities.
- **name**: Specify the name of the action.
- **actionKey**: Specify the type of relationship that the action creates between the subject and participant. See "About the Design Pattern Action Reference Table" for more information.
- **executeOnExistingEntity**: Specify whether to run actions on entities that already exist in the workspace. The default is **true**. Define this value as **false** to prevent actions on entities that exist in the workspace.

> **Note:**
>
> Resource definitions include an attribute called **OverwritePolicy** that specifies a design pattern response if a resource with the same name and the same location exists in the target project.
>
> If the **OverwritePolicy** attribute for a resource is defined as **Always**, the existing resource is overwritten by the resource contained in the design pattern. In this scenario, a design pattern runs actions on the resource even when the **executeOnExistingEntity** attribute is defined as **false**. See "Defining the Manifest for Design Patterns" for more information.

About Design Pattern Action Types

Action types represent a category of action to be performed. For example, an action type can represent a relationship that is created between entities and elements, a copy action that copies data between entities, or an extend action that extends source entities.

You can define design pattern action types with one of the following values:

- **relationship**: Use to create a relation between entities, between elements, or between elements and entities. For example, you can use this action type to establish a relationship between a conceptual model entity in the conceptual model and a resource in an application model.
- **interface**: Use to copy data elements from a participant entity to a subject entity. For example, the data configured in a customer facing service in a conceptual model can be copied to an Inventory Service specification in an application model.
- **parameter**: Use to pass a participant entity as an input to a subject entity. For example, a customer facing service in a conceptual model can be passed as an input to an Action entity that is supported on an Inventory Service specification in an application model.

  When using the **parameter** action type, the data elements in the subject entity reference the participant entity (whereas in the **interface** action type, the data elements are copied to the subject entity).
- **extension**: Use to define an extension relationship between a participant and a subject entity (the subject entity extends from the participant entity).

  When applying design patterns that include this action type, existing subject entities must be writable and must not be included in a sealed project.

About Actions Used in Conceptual Modeling

You can use actions to realize entities in a conceptual model. For example, you can use actions to realize customer and resource facing services, resources, locations, and conceptual model actions.

You can use action type and action key combinations that are specific to the following conceptual modeling tasks.

For more information about action types and action keys, see the Design Pattern Action Reference table, which is available on the Oracle Help Center. Click the following link and then click the link for the current Design Studio version:

http://docs.oracle.com/en/industries/communications/design-studio/index.html

Copying Changeable Elements from Participant to Subject

You use the **interface** action type with the action key noted below if you want to copy from the participant to the subject configuration only those data elements that are tagged with **Changeable** tag. Data elements are that are not defined with the **Characteristic** tag or with the **Changeable** tag are copied to the Service specification but not to the corresponding Service Configuration.

- **Action Key**: oracle.communications.common.configuration
- **Action Type**: interface
- **Subject**: configuration
- **Participant**: customer facing service, location, product, resource, or resource facing service

Adding Components as Configuration Items to Service Configuration Specifications

You use the **relationship** action type with the action key noted below if you want to realize a conceptual model entity that includes components and add those components as configuration items to the Service Configuration specification.

- **Action Key**: oracle.communications.common.configuration
- **Action Type**: relationship
- **Subject**: configuration
- **Participant**: customer facing service, location, product, resource, or resource facing service

Copying Data Elements as Virtual Root Nodes to Conceptual Model Actions

You use the **parameter** action type with the action key noted below if you want to copy data elements from a conceptual model entity to a conceptual model action. The data elements are copied to the action, in this scenario, as virtual root nodes.

- **Action Key**: oracle.communications.sce.common.entity.entities.action.actionclassification.action.parameter.helper
- **Action Type**: parameter
- **Subject**: action
- **Participant**: customer facing service, location, product, resource, or resource facing service

Realizing Technical Actions for Inventory Entities

When working with Inventory entities, you can realize technical actions but maintain (rather than override) the data elements previously saved on the technical action by using the **interface** action type and action key noted below. Using this combination, you can keep existing data elements and save new data elements to the Data Dictionary.

- **Action Key**: oracle.communications.sce.common.entity.action.relalization.ta.helper
- **Action Type**: interface
- **Subject**: technical action generation helper

- **Participant**: service action

Copying Changeable Data Elements from a CFS to an RFS

You use the **interface** action type with the action key noted below if you want to copy data elements tagged as **Changeable** from a customer facing service to a resource facing service. This functionality is reserved for customer facing services and resource facing services only.

- **Action Key**: oracle.communications.sce.common.entity.rfs.interface.helper
- **Action Type**: interface
- **Subject**: resource facing service
- **Participant**: customer facing service

Copying Changeable Elements from a Service to a Service Specification

You use the **interface** action type with the action key noted below if you want to copy data elements tagged as **Changeable** from a service to a service configuration specification. Data elements not tagged as **Changeable** are copied from the service to the service specification.

- **Action Key**: oracle.communications.common.configuration.enabled
- **Action Type**: interface
- **Subject**: service specification
- **Participant**: customer facing service, location, product, resource, or resource facing service

Extending Actions

You use the **extension** action type with the action key noted below to extend one action from another (the subject action extends from the participant action). In this scenario, the data elements are copied from the participant to the subject only if the **Extends** option is not selected for the participant on the Conceptual Model Editor **Properties** tab.

- **Action Key**: oracle.communications.sce.common.entity.entities.action.extends
- **Action Type**: extension
- **Subject**: action
- **Participant**: action

Using Order Item Parameter Bindings to Link Conceptual Model Entities to OSM Projects

You use the **relationship** action type with the action key noted below to link conceptual model entities with an OSM project through an order item parameter binding. In this scenario, when a user applies a design pattern using a conceptual model entity as input and associates to that entity an order item parameter binding, the design pattern creates an XML file containing the attributes of the conceptual model entity, as well as an XQuery file.

- **Action Key**: oracle.communications.studio.osm.transformation.entities.binding.psrentity.handler
- **Action Type**: relationship

- **Subject**: order item parameter binding
- **Participant**: conceptual model entities

Filtering Action Codes

You use the **relationship** action type with the action key noted below to filter action codes when creating a relationship between an action and an action code. In this scenario, only those action codes defined in the associated functional area are added to the action.

- **Action Key**: oracle.communications.sce.common.entity.entities.action.actioncodes.actionTypeRef
- **Action Type**: relationship
- **Subject**: action
- **Participant**: action code

About the Design Pattern Action Reference Table

You use the Design Pattern Action Reference table to review all of the information required to define an action in a design pattern for Design Studio entities, including the valid relationship types and target entities for the actions that can affect each entity type.

For each entity type, the table lists the:

- relationship name
- action key
- subject
- participant
- action type

The Design Pattern Action Reference table is available on the Oracle Help Center. Click the following link and then click the link for the current Design Studio version:

http://docs.oracle.com/en/industries/communications/design-studio/index.html

Defining Actions for Design Patterns

You include actions in a design pattern to define which actions to perform on entities in a workspace. See "Working with Design Pattern Actions" for more information.

To define actions for design patterns:

1. With the **pattern.xml** file open in the Design Studio default XML editor, click the **Source** tab.
2. Add an **actions** element, and then add an **action** element for each action that you want the design pattern to perform.
3. In the **action** element, add and define values for the following action attributes.
   a. For the **name** attribute, enter the name of the field to appear in the design pattern.
   b. For the **id** attribute, enter a unique value to represent the action.
4. Add and define the action child elements.

Some elements are common to all actions, and some are available only to specific action types. See "About Action Elements" for more information.

5. Click **Save**.

Defining Custom Actions

You define custom actions to call custom Java code from a design pattern.

Design patterns invoke custom actions last because custom actions can have dependencies on other design pattern artifacts. Design Studio performs a build before running the custom actions to ensure that the custom actions receive up-to-date Exchange Format model information. See "Invoking Custom Java Code from Design Patterns" for more information.

To define custom actions for design patterns:

1. With the **pattern.xml** file open in the Design Studio default XML editor, click the **Source** tab.
2. Add a **customActions** element, and then add a **customAction** element.
3. In the **customAction** element, add and define values for the following action attributes:
   a. For the **name** attribute, enter the name of the field to appear in the design pattern.
   b. For the **id** attribute, enter a unique value to represent the action.
4. Add and define the **customAction** child elements.
   a. In the **condition** element, add simple or complex conditional logic to enable the design pattern to perform the custom action only under specific conditions. See "About Conditions" for more information.
   b. In the **exchangeFormat** element, add **model** child elements for each studio model entity required for the custom action. A **model** element can be of type **entityRef**, **project**, or **resource**.
   c. In the **parameters** element, add **parameter** child elements required by the custom action. A **parameter** element consists of name and value pairs.
   d. In the **classId** element, enter a registered Java class ID.
5. Click **Save**.

About Conditions

You use conditions to build design patterns that can produce variable outcomes, depending on user input and the existing state of the solution. Conditions enable the design pattern user to define how resources are created and how actions are performed on those resources.

When creating design patterns, you can include simple conditional logic using common String operations, and you can include complex conditions by combining simple conditions using the Any and All operators and by nesting conditions. A condition must evaluate to true for Design Studio to add the associated resource to the workspace or for Design Studio to perform the associated action.

To define design pattern conditions:

1. With the **pattern.xml** file open in the Design Studio default XML editor, click the **Source** tab.
2. (Optional) In the **resource** element or in the **action** element, add a **condition** child element.
3. Expand the **condition** element, then select the **All** child element.
4. Add logical expressions for simple conditions.

   Logical expressions include v1 and v2 string operands. You can embed tokens and token functions in the operand values. Token types are resolved to a string representation for evaluation.

   The following simple conditions are supported:

   - **Equals**: the expression evaluates to true when v1 and v2 are identical.
   - **Not Equals**: the expression evaluates to true when v1 and v2 are different.
   - **Begins With**: the expression evaluates to true when v1 begins with v2.
   - **Ends With**: the expression evaluates to true when v1 ends with v2.
   - **Contains**: the expression evaluates to true when v1 contains v2.
5. Add logical expressions for complex conditions.

   The following complex conditions are supported:

   - **Any**: the expression evaluates to true when any of the component expressions are true.
   - **All**: the expression evaluates to true when all of the component expressions are true.

In the following example, an All condition is defined to ensure that a design pattern adds a resource to a workspace only when the implementation method of the resource is defined as a device and when the implementation system is UIM.

```
<resource id="resImpl" overwritePolicy="NEVER">
   <condition>
      <All>
         <equals v1="@@resource.implementationMethod@@" v2="DEVICE"/>
         <equals v1="@@resource.implementationSystem@@" v2="UIM"/>
      </All>
   </condition>
```

Defining Inputs for Design Patterns

You can use design pattern inputs to create design patterns that enable users to select and perform actions against existing entities in the workspace. The design pattern can use the selected entities or elements as the subject of an action if the design pattern includes an input defined for the selected entity or element type.

For example, if a design pattern is launched from a selected customer facing service (CFS), and in the design pattern there is an input defined for a CFS type, the design pattern can use the selected CFS as the subject of an action defined in the design pattern. The design pattern uses the selected CFS entity name as a default value in the appropriate fields in the Design Pattern wizard.

Design pattern input types can be defined as **entityRefToken** or as **elementRefToken**. See "Working with Tokens" for more information.

To define inputs for design patterns:

1. With the **pattern.xml** file open in the Design Studio default XML editor, click the **Source** tab.
2. Add an **inputs** element, and then add an **input** element for each type of element or entity for which you want to automatically populate token names in a design pattern.
3. In the **input** element, add and define values for the following attributes.
   a. For the **name** attribute, enter the name of the field to appear in the design pattern.
   b. For the **id** attribute, enter a unique value to represent the input.
4. Add and define values for the following **input** child elements.
   - For the **entityRefToken** element, specify an entity for which to create an input.
   - For the **elementRefToken** element, specify a data element for which to create an input.

> **Note:**
>
> You can define entity and element reference tokens for the same input element. When a user selects multiple entities or elements in a Design Studio view and launches a design pattern, the pattern randomly picks one of the entities or elements to display in token fields if multiple selected entities or elements match the design pattern input definitions.

5. Click **Save**.

Securing Design Pattern Information

Design patterns have no security for the information they include, and any user can apply design patterns. Do not include sensitive information in design patterns. If sensitive information is needed to complete the configuration of a design pattern, include a cheat sheet to run after the pattern is applied to manually configure the sensitive information.

Oracle recommends that you use model variables in design pattern templates to replace any sensitive information. Design Studio users can be prompted to configure a sensitive model variable with the name used in the template. See the Design Studio Help for information about working with model variables.

Invoking Custom Java Code from Design Patterns

You can create a design pattern that invokes custom Java code. For example, you can create a design pattern that accesses the Design Studio Exchange Format and generates artifacts based on the Exchange Format model information that is passed to the design pattern.

To invoke custom Java code from a design pattern:

1. Create a Java class that implements the **IDesignPatternCustomAction** Java interface.

   See "About the IDesignPatternCustomAction Java Interface" for more information.

2. Register your Java class.

   When you register your Java class, you define parameter attributes and Exchange Format model entity attributes. See "About Registering Your Java Class" for more information.

3. Call the Java code.

   The code must pass values to the parameters and to the Exchange Format model entities in the design pattern. See "About Calling Your Custom Java Code" for more information.

About the IDesignPatternCustomAction Java Interface

Design Studio provides a Java interface, named **IDesignPatternCustomAction**, to facilitate the use of custom Java code. The interface contains one method, named **performCustomAction**, which provides parameters and Design Studio Exchange Format model entity information. Write your custom Java class to implement this interface and to use the values from the parameters and from the Exchange Format model entities to perform your business logic.

Example 2-9 demonstrates how to write a Java class that implements the **IDesignPatternCustomAction** Java interface.

Example 2-9 IDesignPatternCustomAction Java Interface Example

```
/**
 * DesignPatternCustomActionRITest demonstrates how to write a custom action class and
how to access the Design Studio Exchange Format APIs.
 *
 */
public class DesignPatternCustomActionRITest implements IDesignPatternCustomAction {

    ModelLocator modelLocator = new ModelLocator(true);

    @Override
    public void performCustomAction(final Map<String, Object> parameters, final
Map<String, Model> models) {

        CustomerFacingService cfs = null;
        Project project = null;

        // Output parameter details
        Set<String> keys = parameters.keySet();
        for (String key : keys) {
            StudioLog.logInfo("Parameter Key: " + key + "\t\tParameter Value: " +
parameters.get(key));
        }

        // Get models: CFS and project
        Set<String> modelKeys = models.keySet();
        for (String key : modelKeys) {
            Model model = models.get(key);
            Entity entity = model.getEntity().get(0);
            StudioLog.logInfo("Model Key: " + key + "\t\tModel Entity Name: " +
entity.getName());
            if (entity instanceof CustomerFacingService) {
                cfs = (CustomerFacingService) entity;
            } else if (entity instanceof Project) {
                project = (Project) entity;
            }
```

```
        }

        // Output CFS name and components
        if (cfs != null) {
            StudioLog.logInfo("CFS name = " + cfs.getName());
            List<ResourceFacingService> rfsList =
getAllRFSComponentsFromCFS(cfs);
            for (ResourceFacingService rfs : rfsList) {
                StudioLog.logInfo("RFS name = " + rfs.getName());
            }

            List<CGBUResource> resourceList =
getAllResourceComponentsFromCFS(cfs);
            for (CGBUResource resource : resourceList) {
                StudioLog.logInfo("Resource name = " + resource.getName());
            }
        }

        // Output project details
        if (project != null) {
            StudioLog.logInfo("Project name = " + project.getName());
        }
    }
```

About Registering Your Java Class

You register your Java class to enable Design Studio to validate the input from the design pattern.

You define the parameters with the following attributes:

- **name**, which is the key used by the **performCustomAction** method.
- **type**, which you define with one of the following values: **boolean**, **string**, **date**, **dateTime**, **int**, **float**, **double**, **long**.
- **optional**, which is a boolean that indicates whether the parameter is optional (if omitted, the parameter is mandatory).

You define the Exchange Format model information with the following attributes:

- **name**, which is the key used by the **performCustomAction** method.
- **entityType,** which specifies the entity file extension. For example, you can define an **entityType** for actions (**cmnAction**), resources (**cmnResource**), locations (**cmnLocation**), customer facing services (**custSrvc**), resource facing services (**rsrcSrvc**), and model projects (**ddCartridge**).
- **optional**, which is a boolean that indicates whether the parameter is optional (if omitted, the parameter is mandatory).

> **Note:**
>
> **oracle.communications.sce.integration.test.design.pattern.apply.DesignPatternCustomActionRITest** must implement **IDesignPatternCustomAction**.

Example 2-10 shows an example of how to register your Java class.

Example 2-10 Registering a Java Class

```
<extension point="oracle.communications.sce.pattern.core.customActionClass">
     <customActionClass
         id="oracle.communications.sce.integration.test.design.
             pattern.apply.DesignPatternCustomActionTest.ri"
         class="oracle.communications.sce.integration.test.design.
                pattern.apply.DesignPatternCustomActionRITest">
         <!-- Parameter hasPrefix is a boolean, and it is mandatory -->
         <parameter name="usePrefix" type= "boolean"/>
         <!-- Parameter prefix is a string, and it is optional -->
         <parameter name="prefix" type="string" optional = "true"/>
         <!-- Exchange Format model cfs is a customer facing service defined
              with an extension custSrvc, and it is mandatory -->
         <model name="cfs" entityType="custSrvc"/>
         <!-- Exchange format model uimProject is an inventory cartridge defined
              with an extension inventoryCartridge, and it is mandatory -->
         <model name="uimProject" entityType="inventoryCartridge"/>
     </customActionClass>
  </extension>
```

About Calling Your Custom Java Code

To call your custom Java code, you add a **customActions** element to your design pattern. Because custom actions can have dependencies on other design pattern artifacts, the design pattern invokes custom actions last. Design Studio builds the projects before running the custom actions to ensure that the custom actions receive up-to-date Exchange Format model information.

The **customAction** element requires:

- A **condition** child element.
- A list of Design Studio model entities, which can be **entityRef**, **project**, or **resource**. The design pattern generates an error if a model entity is not in the workspace and if a model entity in the design pattern does not match a model entity registered in the Exchange Format.
- A list of parameters, which are name and value pairs.

  The design pattern generates a validation error if the parameters in the design pattern do not match the registered parameters.
- A registered Java class ID.

  oracle.communications.sce.integration.test.design.pattern.apply.DesignPatternCustomActionTest.ri is the ID of the registered Java class **oracle.communications.sce.integration.test.design.pattern.apply.DesignPatternCustomActionTest**

Example 2-11 displays an example of the **customActions** element in the DesignPattern.xsd file.

Example 2-11 customActions Element

```
<customActions>
   <customAction id="riCustomAction" name="RI Custom Action">
      <condition>
         <equals v1="@@usePrefix@@" v2="true"/>
      </condition>
      <exchangeFormat>
         <model>
```

```
            <name>cfs</name>
            <id>cfs</id>
            <type>entityRef</type>
         </model>
         <model>
            <name>uimProject</name>
            <id>uimCfsProject</id>
            <type>project</type>
         </model>
      </exchangeFormat>
      <parameters>
         <parameter>
            <name>usePrefix</name>
            <value>@@usePrefix@@</value>
         </parameter>
         <parameter>
            <name>prefix</name>
            <value>@@prefix@@</value>
         </parameter>
      </parameters>
      <classId>oracle.communications.sce.integration.test.design.
               pattern.apply.DesignPatternCustomActionTest.ri</classId>
   </customAction>

   <customAction id="riCustomActionNoPrefix"
                 name="RI Custom Action With No Prefix">
      <condition>
         <equals v1="@@usePrefix@@" v2="false"/>
      </condition>
      <exchangeFormat>
         <model>
            <name>cfs</name>
            <id>cfs</id>
            <type>entityRef</type>
         </model>
         <model>
            <name>uimProject</name>
            <id>uimCfsProject</id>
            <type>project</type>
         </model>
      </exchangeFormat>
      <parameters>
         <parameter>
            <name>usePrefix</name>
            <value>@@usePrefix@@</value>
         </parameter>
      </parameters>
      <classId>oracle.communications.sce.integration.test.design.
               pattern.apply.DesignPatternCustomActionTest.ri</classId>
   </customAction>

</customActions>
```

Testing Design Patterns

You test design patterns by running them in the Design Pattern wizard. Testing design patterns directly in Design Studio shortens the design-test cycles by eliminating the need to build and deploy design patterns as features.

To test design patterns:

1. In Design Studio, from the **Studio** menu, select **Design Pattern**.

   The Design Pattern wizard appears.

2. Select the **Select a Design Pattern from File** option.

3. Click **Browse** and locate and select the design pattern that you want to test.

   Select the design pattern root folder for the design pattern project. The design pattern root folder contains the **pattern.xml** file.

4. Click **Next**.

   Design Studio validates the integrity of the design pattern. If the selected design pattern contains structural problems (such as schema validation errors or invalid XML), an error message appears. The error message displays a specific schema validation error and the line and column in the document where the error exists. You must fix all errors before testing a design pattern.

   If there are no errors, the Introduction page appears.

5. Navigate through the wizard to ensure that the design pattern works as intended.

6. Review the summary of changes on the Summary page.

   Ensure that the entities created by the design pattern meet your expectations.

7. Finish running the design pattern and review the resources added to the workspace.

   After the design pattern completes, examine the files that have been added to the workspace to ensure they are named correctly and that they contain the expected configuration. If they are configured incorrectly, review the token substitution configuration.

8. When the wizard completes, verify that all token replacements work as intended.

   For example, you may need to correct any misspelled tokens in the **pattern.xml** file and all other documents.

9. (Optional) If the design pattern doesn't meet with your expectations, roll back to a previous version.

   Oracle recommends that you use a version control system and employ a backup strategy to roll back to previous versions and to ensure against data loss if the design pattern creates unexpected results.

Applying Design Patterns

Design Studio users apply design patterns by accessing the Design Pattern wizard in the Design Studio user interface. Design patterns do not require reinstallation when changing workspaces or when importing or deleting projects.

> **Note:**
>
> Design patterns may overwrite resources or skip existing resources. Re-running a design pattern with the same input may result in a different output depending on the current state of the workspace and the configuration details of the design pattern.
>
> Some design patterns can overwrite existing resources. Oracle recommends that you use a version control system and employ a backup strategy to ensure against data loss. See *Design Studio System Administrator's Guide* for more information about backing up and restoring Design Studio data.

See Design Studio Help for information about applying design patterns.

About the Design Pattern Summary Page

When design pattern users navigate through the Design Pattern wizard to completion, Design Studio displays the Design Pattern Summary page.

Users can review the Summary page to ensure that the pattern is applying the correct resources to the workspace, performing the correct actions, and recognizing appropriate inputs. The Summary page also displays expanded condition evaluations, enabling users to debug condition behaviors. Design Studio saves the Summary page in the root of the workspace.

The Summary page displays the following:

- All field values that you provided.
- All resources to be copied to the workspace. The original name, new name, resource type, and target project displays for each resource. This section also indicates whether any resources with identical names exist in the workspace and whether the design pattern will overwrite the existing values. For example, if a resource with the same name and type exist in the workspace and the resource override value is defined as true, when the design pattern is applied the local resource file will be overwritten.
- All actions. The subject entity, the participant entity, and action type appear for each action. This information includes whether the relationships and parameters in existing entities can be overwritten by the design pattern. Restricted actions that cannot be performed because of the configuration appear in the Restricted Actions section.
- All inputs. The input entity or element appears for each input.
- All conditions, entity properties, and token functions used in a design pattern.

Design Pattern Examples

Use the following examples to help you create your design patterns.

- Example: Adding Project Dependencies
- Example: Defining Tokens for Resources
- Example: Defining Tokens as Default Values

- Example: Defining Action Subjects or Participants With Values External to Design Patterns
- Example: Supporting Multiple Selections for Entity Reference Tokens

Example: Adding Project Dependencies

Example 2-12 demonstrates how to make a new project created in a design pattern dependent on an existing base cartridge.

In the example, the design pattern creates a dependency from a newly created project to the existing base project **OracleComms_Model_base**. The user running the design pattern is not required to specify the name of the base project to create the dependency.

When a user runs the design pattern, and if the cartridge entity specified in the **entityreftoken** element (**OracleComms_Model_base.ddCartridge**) does not exist in the workspace, Design Studio displays the following message for the Action **Add Dependency** on the summary page:

```
This Action cannot be executed

Reason : Either subject or participant or both is/are evaluated as NULL/do(es)
not exist.
```

Example 2-12 Adding Project Dependencies

```
<input id="DPProject1" name="input1">
     <entityRefToken>ref1</entityRefToken>
</input>
<input id="DPProject2" name="input2">
     <entityRefToken>ref2</entityRefToken>
</input>
<project id="DPProject" name="DS Project" tokenGroup="ProjectCreation">
     <description>Model Project</description>
     <typeId>MODEL</typeId>
     <defaultValue>Test_CFS</defaultValue>
</project>
<tokenGroup name="Project Creation" id="ProjectCreation">
     <description>Create project silently using default value.</description>
</tokenGroup>
<tokenGroup name="Project Creation" id="DependencyCreation">
     <description>Create project silently using default value.</description>
</tokenGroup>
<token name="EntityName1" tokenGroup="DependencyCreation"
             id="ref1" xsi:type="EntityRefToken">
     <condition>
          <equals v2="1" v1="2"/>
     </condition>
     <defaultValue>@@DPProject@@</defaultValue>
     <entityType>ddCartridge</entityType>
     <relationship>xxx</relationship>
</token>
<token name="EntityName2" tokenGroup="DependencyCreation"
             id="ref2" xsi:type="EntityRefToken">
     <condition>
          <equals v2="1" v1="3"/>
     </condition>
     <defaultValue>OracleComms_Model_base</defaultValue>
     <entityType>ddCartridge</entityType>
     <relationship>xxx</relationship>
</token>
```

```
<action name="Add dependency" id="dependendyaction">
     <actionType>relationship</actionType>
     <subject>
          <participantType>input</participantType>
          <id>DPProject1</id>
     </subject>
     <participant>
          <participantType>input</participantType>
          <id>DPProject2</id>
     </participant>
     <actionKey>unknown</actionKey>
 </action>
```

Example: Defining Tokens for Resources

Consider that you want to define a token called **equipmentName** and use it to give an equipment specification in a design pattern a different name.

```
<token name="Equipment ID" tokenGroup="Resources" id="equipmentName">
     <description> The Equipment specification</description>
</token>
```

In this example, you would define the **resource** element **targetLocation** element for the specification as:

```
<targetLocation>model/equipment/@@equipmentName@@.equipment</targetLocation>
```

If the user (who runs the design pattern) enters the value **opticalDevice** for the **equipmentName** token, the **targetLocation** value in this example expands to the following when the pattern is applied:

```
<targetLocation>model/equipment/opticalDevice.equipment</targetLocation>
```

You can also define tokens in other locations of the path in the **targetLocation**. For example, you might define a token named **deviceVendor** and use it to expand the previous example:

```
<targetLocation>model/equipment/@@deviceVendor@@/@@equipmentName@@.equipment</
targetLocation>
```

If the user enters the value **oracle** for the **deviceVendor** token, the **targetLocation** value in this example expands to:

```
<targetLocation>model/equipment/oracle/opticalDevice.equipment</targetLocation>
```

> **Note:**
>
> Design Studio automatically creates directories as needed when copying resources into a workspace.

Example: Defining Tokens as Default Values

You can define tokens as default values of other tokens. For example, you might define a token called **deviceGateway** and define it with the following default value:

```
<defaultValue>@@equipmentName@@_gateway</defaultValue>
```

If the user enters the value **opticalDevice** for the **equipmentName** token, the default value in this example expands to:

```
<defaultValue>opticalDevice_gateway</defaultValue>
```

> **Note:**
>
> When using embedded tokens as default values for other tokens, ensure that the embedded token appears in an earlier token group than where it is used. If a value has not been assigned before the token is displayed to a user, the Design Pattern wizard displays the embedded token ID in the **defaultValue** element.

Example: Defining Action Subjects or Participants With Values External to Design Patterns

In design patterns, action subjects and participants can be populated with values provided by the user or populated with resources generated by the design pattern. If you have a project or entity to which you need to make a reference or with which you need to create a dependency, and that entity or project exists in the workspace and is not a resource generated by the design pattern or the result of user input, you can use a default value of a hidden token to populate a field that references the entity.

Example 2-13 demonstrates how you can define a token element default value with a pre-populated value (rather than with inputs captured when running the design pattern). By defining a condition that always evaluates to false, the token does not appear as a field in the design pattern, but the token value can be used by the design pattern to reference an existing entity in the workspace.

Example 2-13 Defining Action Subjects or Participants With Values External to Design Patterns

```
<token name="projectNameToken1" tokenGroup="PDHiddenInfo" id="projectName1"
      xsi:type="EntityRefToken">
      <condition>
         <equals v1="test" v2="test1"/>
      </condition>
      <defaultValue>testProject1</defaultValue>
      <entityType>ddCartridge</entityType>
      <relationship>unknown</relationship>
</token>

<token name="projectCommon" tokenGroup="PDHiddenInfo" id="projectCommon"
      xsi:type="EntityRefToken">
      <condition>
         <equals v1="test" v2="test2"/>
      </condition>
      <defaultValue>OracleComms_Model_Base</defaultValue>
      <entityType>ddCartridge</entityType>
      <relationship>unknown</relationship>
</token>
<input id="projectName1_input" name="Project">
      <entityRefToken>projectName1</entityRefToken>
</input>
```

```
<input id="projectOracleComm_input" name="Project">
      <entityRefToken>projectCommon</entityRefToken>
</input>

<action id="AssociateDependencyProject" name="Associate Dependency Project">
      <actionType>relationship</actionType>
      <subject>
           <participantType>input</participantType>
           <id>projectName1_input</id>
      </subject>
      <participant>
           <participantType>input</participantType>
           <id>projectOracleComm_input</id>
      </participant>
      <actionKey>unknown</actionKey>
      <executeOnExistingEntity>true</executeOnExistingEntity>
</action>
```

Example: Supporting Multiple Selections for Entity Reference Tokens

Example 2-14 illustrates how to use inputs, entity reference tokens, and actions in the **pattern.xml file** to enable a user to select multiple entities when defining a project dependency.

Example 2-14 defines two entity reference tokens that are used in two different inputs. When the user runs this design pattern, the **Add dependency** action creates a dependency from the Cartridge Project entity the user selects in the **Conceptual Model Project** field (specified in input1) to each Cartridge Project entity they select in the **Conceptual Model Project Dependencies** field (specified in input2).

Example 2-14 Supporting Multiple Selections for Entity Reference Tokens

```
<inputs>
   <input id="input1" name="input1">
      <entityRefToken>modelEntity</entityRefToken>
   </input>
   <input id="input2" name="input2">
      <entityRefToken>entityContainer</entityRefToken>
   </input>
</inputs>
<tokens>
   <token name="Conceptual Model Project"
          tokenGroup="ProjectCreation"
          id="modelEntity"
          xsi:type="EntityRefToken">
          <entityType>ddCartridge</entityType>
   </token>
   <token name="Conceptual Model Project Dependencies"
          tokenGroup="ProjectCreation"
          id="entityContainer" xsi:type="EntityRefToken">
          <allowMultiple>true</allowMultiple>
          <entityType>ddCartridge</entityType>
   </token>
</tokens>
<actions>
   <action name="Add dependency" id="dependencyaction">
      <actionType>relationship</actionType>
      <subject>
         <participantType>input</participantType>
```

```
            <id>input1</id>
         </subject>
         <participant>
            <participantType>input</participantType>
            <id>input2</id>
         </participant>
         <actionKey>unknown</actionKey>
      </action>
</actions>
```

Working with Cheat Sheets

Design Studio supports cheat sheets, which refers to the integration of documented procedures with wizards in the application. Cheat sheets are XML documents that can be interpreted by the Eclipse Cheat Sheet framework, and developers can map cheat sheets to specific points in the Design Studio user interface (for example, in editors and views). You access the cheat sheets that are relevant to current tasks, and complete those tasks using the included instructions. Cheat sheets enable you to find documentation for relevant solution design procedures and facilitate the learning of those procedures.

For example, you can use cheat sheets with design patterns to describe the resources added to a workspace and to assist users with any manual steps required after a design pattern is applied. Cheat sheets are not mandatory for design patterns, but they are recommended.

You can develop and edit cheat sheets using the Eclipse Cheat Sheet editor.

For information about creating and developing cheat sheets, see "Building Cheat Sheets in Eclipse" on the Oracle Technology Network:

http://www.oracle.com/technetwork/articles/entarch/eclipse-cheat-sheets-092351.html

3

Working with Guided Assistance

This chapter provides information about guided assistance, how to create guided assistance in Oracle Communications Service Catalog and Design - Design Studio, and how to distribute guided assistance.

Working with Guided Assistance

Design Studio guided assistance is a range of context-sensitive learning aides mapped to specific editors and views in the user interface. For example, when working in editors, you can open the Guided Assistance dialog box for Help topics, cheat sheets, and recorded presentations that are applicable to that editor.

When working with guided assistance, you can review the learning aids delivered with Design Studio, and you can create your own and map them to projects and entities by using design patterns or by defining values for attributes directly in the guided assistance extension point.

About the Guided Assistance Dialog Box

You can access learning aids delivered in Design Studio by opening the Guided Assistance dialog box, which is available from the **Studio** menu, the main tool bar, and from the Studio Projects view context menu.

The learning aids included in the Guided Assistance dialog box are organized into categories that reflect a specific domain. For example, the **Order and Service Management Project** directory includes a category called **Order**, which includes learning aids that help you define and configure orders.

The Guided Assistance dialog box is organized in the following hierarchy:

- The root level contains global guided assistance that is not specific to any cartridge project.
- The second level is organized by project type and contains guided assistance for specific cartridge projects.
- The third level contains entity-specific guided assistance for each project type.
- The fourth level contains folders that include learning aids that are specific to functionality within an entity.

Folders appear only when there is guided assistance available at the corresponding folder level. When you first open the Guided Assistance dialog box, the hierarchy expands to the folder relevant to the context in focus. If no guided assistance is mapped to the active context, the hierarchy will appear fully collapsed. You can navigate to any folder level in the hierarchy, regardless of the context in focus.

Working with Guided Assistance Design Patterns

You can create and implement new guided assistance by using design patterns delivered with Design Studio.

These design patterns are preconfigured for guided assistance development and can help you create a directory structure where Design Studio users can save guided assistance learning aids. After applying a guided assistance design pattern, this directory structure appears in the **plugin.xml** file.

Creating Guided Assistance Using Design Patterns

The following procedure describes how to create your own guided assistance using design patterns.

To create new guided assistance using design patterns:

1. In Design Studio, verify that the **Oracle Communications Design Studio Design Pattern Feature** is installed.

   Contact your system administrator if this feature is not available.

2. From the **Studio** menu, select **Design Pattern**.

   The Design Pattern dialog box appears.

3. Expand the **Others** folder.

   A list of guided assistance folders appears, one for each Design Studio project type.

4. Expand the folder for the project type for which you want to add guided assistance.

   For example, if you are adding guided assistance for model projects, expand the **Guided Assistance Design Pattern for Model** folder.

5. Select the design pattern and click **Next**.

   The Design Pattern wizard Introduction page appears.

6. Read the information on the Information page, and then click **Next**.

   The Select Project page appears.

7. Do one of the following:

   - To use an existing project for guided assistance development, select a plug-in project in the **Guided Assistance Plug-in Project** field.
   - To create a new project for guided assistance development, click **New**.

8. Click **Next**.

   The Plug-in Information page appears.

9. Enter all required information, and then click **Next**.

   The Summary page appears.

10. Review the summary information, and then click **Finish**.

    Design Studio populates the project with information necessary to build the guided assistance for the designated project type. The information includes a manifest and a **guidedAssistance** directory, which contains the folder structure for the project type and all plug-in related configuration for the packaging of the guided assistance. When you finish the wizard, Design Studio opens the Guided Assistance cheat sheet in the Help view.

11. (Optional) Modify the cheat sheets in the **oracle.communications.sce.guided.assistance.feature**.

The cheat sheets should provide help to users who apply the guided assistance design pattern. The information should describe how to copy guided assistance learning aids to the folder structure created by applying the design pattern.

Working with the Guided Assistance Extension Point

You use the guided assistance extension point (**com.mslv.studio.core.ui.studioGuidedAssistance**) to register the learning aid content locations.

In the **plugin.xml** file (located at the root of the plug-in project), you define attributes for the extension point to register guided assistance (for example, cheat sheets, HTML files, and Help documents) content locations that are applicable to:

- All project types (**globalGuidedAssistance**)
- A specific project type (**cartridgeGuidedAssistance**)
- A specific entity type (**entityGuidedAssistance**)
- A specific functionality in an entity type (**guidedAssistanceContent**, which is a child element of **entityGuidedAssistance**)

Table 3-1 Guided Assistance Extension Point Attributes

Attribute	Element Used In	Use
guideName	globalGuidedAssistance cartridgeGuidedAssistance guidedAssistanceContent	Enter the display name that appears in the Guided Assistance dialog box for the learning aid. If you define no value for **helpContextId**, the name you define here is displayed for the **contentLocation**, but only when the location refers to a single file; otherwise this value is ignored.
contentLocation	globalGuidedAssistance cartridgeGuidedAssistance guidedAssistanceContent	Enter the folder location for the learning aid.
helpContextId	globalGuidedAssistance cartridgeGuidedAssistance guidedAssistanceContent	Enter a unique ID that represents the learning aid. Design Studio uses this value to display the appropriate Help page. Optionally, you can specify the Help URL. Use the following format to define a Help context ID: *pluginId.contextId* For example: **com.company.product.help.myContextId**
projectTypeId	cartridgeGuidedAssistance (optional) entityGuidedAssistance	Enter a fully qualified ID of the project type to which the learning aid is related. The projectType extension can be defined in any plug-in (that is, it does not need to be defined in the same plug-in as cartridgeGuidedAssistance).

Table 3-1 (Cont.) Guided Assistance Extension Point Attributes

Attribute	Element Used In	Use
entityId	entityGuidedAssistance	Enter the fully qualified name of a modelType extension. The extension can be defined in any plug-in (that is, it does not need to be defined in the same plug-in as entityGuidedAssistance). The Package Explorer view **model** folder includes an entityId for each entity in a project.
folderId	guidedAssistanceContent	Enter the name of the folder in which the learning aid should appear in the Guided Assistance dialog box.

Guided Assistance Extension Point Example

Example 3-1 shows how you can configure the attributes for the guided assistance extension point:

Example 3-1 Example: Guided Assistance Extension Point

```
<extension
  point="com.mslv.studio.core.ui.studioGuidedAssistance">
   <globalGuidedAssistance
         contentLocation="guidedassistances/"
         guideName="Global Guides Location">
   </globalGuidedAssistance>

     <cartridgeGuidedAssistance
         contentLocation="guidedassistances/ModelProject/"
         guideName="Data Dictionary Wizard Help"
         helpContextId="com.mslv.studio.core.help.DataDictionaryWizard"
         projectTypeId="com.mslv.studio.core.datadictionary.project">
     </cartridgeGuidedAssistance>
      <entityGuidedAssistance
            entityId="com.mslv.studio.core.dataDictionary">
            projectTypeId="com.mslv.studio.core.datadictionary.project">
        <guidedAssistanceContent
            contentLocation="guidedassistances/ModelProject/DataSchema/"
            guideName="Data Schema Guides"></guidedAssistanceContent>
      </entityGuidedAssistance>

      <cartridgeGuidedAssistance
           contentLocation="guidedassistances/OSM/"
           guideName="OSM Guides"
           projectTypeId="com.mslv.studio.provisioning.project">
      </cartridgeGuidedAssistance>

      <entityGuidedAssistance
           entityId="com.mslv.studio.provisioning.process">
        <guidedAssistanceContent
            contentLocation="guidedassistances/OSM/order/"
            folderId="com.mslv.studio.provisioning.order">
        </guidedAssistanceContent>
        <guidedAssistanceContent
            contentLocation="guidedassistances/OSM/process/creationTask/"
            folderId="orderCreationTask">
        </guidedAssistanceContent>
```

```
      </entityGuidedAssistance>

      <entityGuidedAssistance
            entityId="com.mslv.studio.provisioning.order">
        <guidedAssistanceContent
            contentLocation="guidedassistances/OSM/order/">
        </guidedAssistanceContent>
      <guidedAssistanceContent
        contentLocation="guidedassistance/OSM/video/722_whats_new.htm"
        guideName="Design Studio 7.2.2 What's New Video">
      </guidedAssistanceContent>
      </entityGuidedAssistance>
</extension>
```

Distributing Guided Assistance

You save guided assistance in Eclipse plug-in projects. Plug-in projects are grouped into features, and your system administrator can make these features available to other users by adding the feature to your Design Studio update site.

To distribute guided assistance:

1. From the **File** menu, select **New** and then select **Project**.

   The New wizard appears.
2. Expand the **Plug-in Development** folder.
3. Select **Feature Project**, and then click **Next**.

   The New Feature wizard appears.
4. Enter the information required by the wizard, and then click **Finish**.

   The new feature appears in the Feature editor.
5. Click the **Plug-ins** tab.
6. Click **Add**.
7. Add the plug-in project in which you saved the guided assistance.

   You can add any number of projects to the feature.
8. Contact your system administrator to request that the new feature be added to the Design Studio update site.

See *Eclipse Plug-in Development Environment Guide* for information about using feature projects. The samples included in the Design Studio software package demonstrate how your system administrator can configure feature projects.

About the Design Pattern and Guided Assistance SDK Folder

The design pattern and guided assistance **SDK** folder is a root-level folder included in the Design Studio software package available on the Oracle software delivery website:

https://edelivery.oracle.com

It includes the following:

- A **samples** folder

  This folder contains the **Pattern** folder and a **Guided Assistance** folder.

The **Pattern** folder contains the **oracle.communications.sce.pattern.sample.zip** archive file, which includes the following projects:

- **oracle.communications.sce.pattern.sample** contains the plug-in project (**plug-in.xml**) that includes a single design pattern.
- **oracle.communications.sce.pattern.sample.feature** contains the feature project (**feature.xml**) that you can use for building and distributing the plug-in project.
- **oracle.communications.sce.pattern.update.site** contains an update site project (**site.xml**) that illustrates how your system administrator can build an update site for delivering your feature to end users.

The **Guided Assistance** folder contains the **oracle.communications.sce.guidedassistance.sample.zip** archive file, which includes the following projects:

- **oracle.communications.sce. guidedassistance.sample** contains the plug-in project (**plug-in.xml**) that includes guided assistance mappings.
- **oracle.communications.sce. guidedassistance.sample.feature** contains the feature project (**feature.xml**) that you can use for building and distributing the plug-in project.
- **oracle.communications.sce. guidedassistance.update.site** contains an update site project **(site.xml**) that illustrates how your system administrator can build an update site for delivering your feature to end users.

See the Design Studio Help for information about importing projects into your workspace.

- A **schema** folder

  This folder contains the Design Pattern XML schema (**DesignPattern.xsd**), which is a standard XML Schema document. You can review the contents of this document using any schema or XML editor.

4

Working with the Design Studio Exchange Format

This chapter describes the Oracle Communications Service Catalog and Design - Design Studio Exchange Format, the Exchange Format data schemas, and the modeling patterns that facilitate custom extensions when working with the Design Studio Exchange Format.

About the Design Studio Exchange Format

The Design Studio Exchange Format is an XML document based on the data model defined for Design Studio projects. The XML document is generated by a project build.

The Exchange Format represents the output of Design Studio configuration in a published XML format, facilitates the exchange of solution modeling information between Design Studio and other systems or applications, and enables you to extend Design Studio functionality.

For example, you can use the Exchange Format when:

- Driving a run-time engine, such as for the Calculate Technical Action or for the Design and Assign provider functions.
- Driving a third-party application, such as when generating reports from a solution.
- Synchronizing with another catalog, such as when refining a catalog external to Design Studio.
- Generating Java code in a reference implementation, especially for an implementation that is repetitive or pattern-based.

The Exchange Format represents all entities, elements, and relationships in Design Studio, and is used to produce an XML file (ending in **.studioModel**) for every Design Studio entity type in a solution. Each entity type, such as Product, Customer Facing Service, and Order entities, has an XML schema that describes the content of the XML file that is produced for the entity type. The XML files are generated every time you run a full or incremental build. Design Studio saves the XML files in a project **generated** folder, which you can access from the Project Explorer view.

> **Note:**
>
> Design Studio generates Exchange Format XML files for a sealed project during the initial import if the sealed project directory does not contain any **.studioModel** XML files in the project **generated** folder. Design Studio does not update the **generated** folder XML files for sealed projects during subsequent builds.
>
> Before distributing sealed projects, Oracle recommends that you generate the project Exchange Format XML files to reduce initial build times when team members import the sealed projects.

The XML output that is generated from the Exchange Format is the same as the XML input used in the Design Studio reporting framework. Report designers can use the Exchange Format XML files to design custom reports. See "Working with Reports" for more information.

You can leverage the Exchange Format information by referencing the published XML format and data schemas, or you can use the Design Studio Model Java API to access the information in the Exchange Format. The Design Studio Model Java API is a wrapper that reads the Exchange Format XML files produced by Design Studio. When combined with other public Eclipse APIs, the Design Studio Model Java API enables you to extend Design Studio with custom functionality and features. For example, you can use the Design Studio Model Java API to add a new action to a Design Studio menu. See "Working with Design Studio Model Java API" for more information.

You can leverage the Exchange Format information when working with:

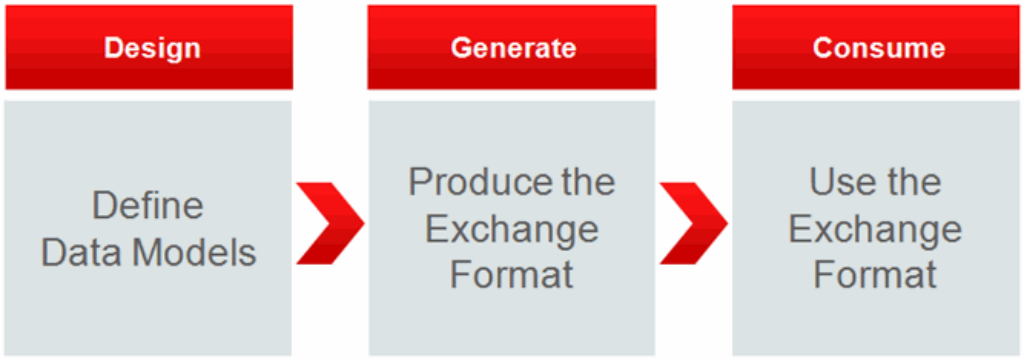
- XML technologies, such as XQuery, XSLT, JavaScript, Java, and so forth
- The Java model API
- Action command extensions
- Eclipse Builder and Packager extensions
- Other Eclipse extensions

About the Exchange Format Model Lifecycle

Figure 4-1 illustrates the Exchange Format model lifecycle, which includes the following phases:

- Design: You define data models in a project in Design Studio.
- Generate: You produce the Exchange Format by building the project.
- Consume: You use the Exchange Format to extend Design Studio or to integrate with external systems.

Figure 4-1 Exchange Format Lifecycle



When integrating with third-party applications, you can:

- Add commands to Design Studio menus, which can be invoked to read the Exchange Format and run the integration logic necessary to propagate the model to an external system. Using action commands enables you to interactively invoke

custom logic while designing a solution, such as validating an XQuery path provided in the solution design.

- Add Model Processors to Design Studio, which run as background processes when you build a project. Model Processors can consume the Exchange Format and integrate with external systems. Using Model Processors enables you to embed custom logic in the solution design and to invoke that logic while building or compiling the solution. For example, you can generate custom artifacts or validate third-party components.

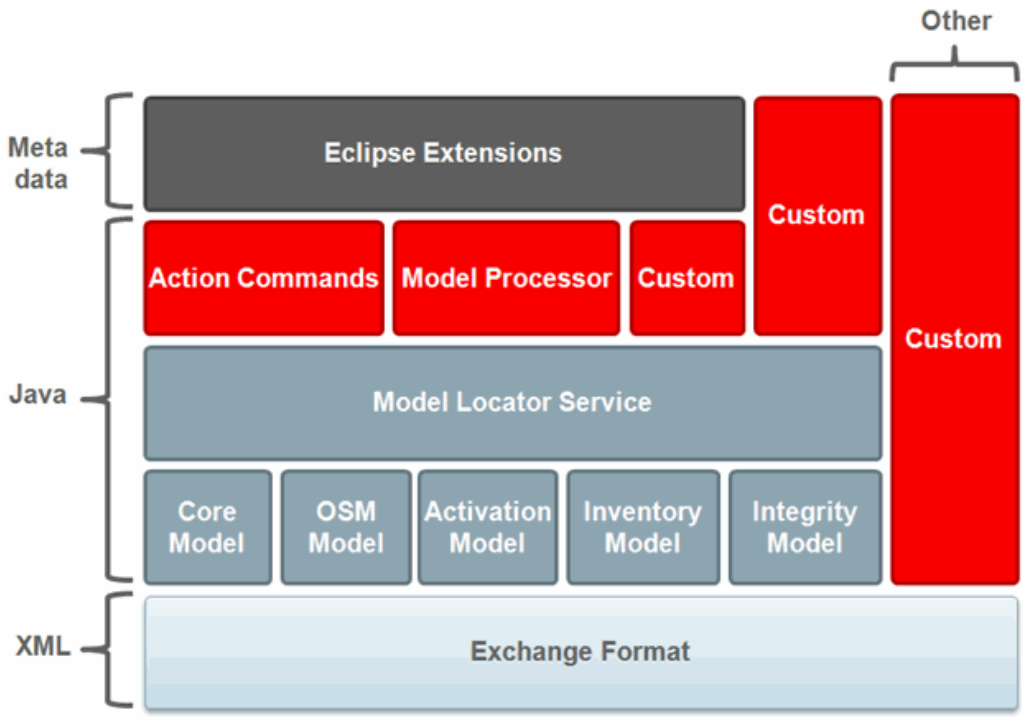
Additionally, external systems can interact with custom Design Studio extensions to access the Java Model API, from which the external system can consume the Exchange Format. Also, external systems can use the Java Model API or the XML files directly to consume the Exchange Format.

About the Exchange Format Architecture

Figure 4-2 illustrates the Exchange Format architecture:

- The Action Commands, Model Processor, and three Custom blocks represent architecturally distinct components that provide custom logic using the Exchange Format XML or using the Design Studio Model Java API.
- The Exchange Format block represents the fundamental definition of the Exchange Format (using XML technologies). The Model Locator Service block and the set of Model blocks directly under the Model Locator Service block represent the parts of the Java API which provide simplified access to the Exchange Format XML.
- The Eclipse Extensions block represents the metadata that describes extensions to the Eclipse platform and to Design Studio. Typically, the extensions are supported by Java implementations. The Action Commands, Model Processor, and Custom Java code represent the Java implementation supporting those extensions.
- The Custom block built on the Design Studio Java API (represented by the Model Locator Service block and the set of Model blocks directly under the Model Locator Service block) represents a custom Java implementation built independently of the Eclipse platform and of the Design Studio features.
- The Custom block noted as Other and built directly on the Exchange Format represents an implementation that utilizes the Exchange Format XML directly. This type of custom implementation may use XML transformation technologies (such as XQuery and XSLT) and represents any integration with the Design Studio platform for which you elect to use the Exchange Format.

Figure 4-2 Exchange Format Architecture



About the Design Studio Model Schemas

Design Studio includes schemas that describe the XML files generated from the published Exchange Format. These schemas are bundled in a **schemas** folder in the **Design Studio Report Design** example and in the **Design Studio Action Command** example.

If, after you add the examples to your workspace, you move these schemas to a new location, ensure that you copy and move the full set of schemas in the **schemas** folder, as these schemas have dependencies defined among them.

You can review the details of the full Design Studio model by opening and reviewing the Design Studio schemas (you can also review the model by browsing the Design Studio Model Java API). For example, you can browse the schemas to review details about named attributes, named lists, and cardinality.

Viewing the Design Studio Schemas

You can review the details of the full Design Studio model by viewing the Design Studio schemas.

> **Note:**
>
> To view the Design Studio schemas, you must first add the **Design Studio Report Design** example or the **Design Studio Action Command** example to your workspace. See "Adding the Report Design Example to the Workspace" or "Adding the Design Studio Action Command Example to a Workspace" for more information.

To view the Design Studio schemas:

1. In Design Studio, switch to the Java perspective.
2. Click the **Package Explorer** tab.

   The Package Explorer view becomes active.
3. Do one of the following:
   - Expand the **design.studio.example.action.command** folder.
   - Expand the **design.studio.example.report.designs** folder.
4. Expand the **schemas** folder.
5. Double-click one of the schema files to open the file in the Data Schema editor.

About the Design Studio Exchange Format Model

The Design Studio Exchange Format represents all entities, elements, and relationships in the Design Studio model. The information in the following sections describe model patterns that are useful when designing reports and when extending Design Studio with custom functionality.

Element Attributes and Children

The Design Studio Exchange Format includes XML elements named **<element>**, and these elements are called base elements.

All base elements in the Exchange Format model include the following base attributes:

- @id (unique locator)
- @name
- @type
- @typeName
- @kind (Project, Entity, Element)
- @path

Base elements also define the following child elements:

- displayName (localizedString)
- Note (localizedString)

A **localizedString** is a string value with an **@lang** attribute. The **@lang** attribute defines the related language. You use the **@lang** value **default** to define the default string value.

Example 4-1 Element Base Attributes and Children

```
<element
    id=""
    name=""
    type=""
    typeName=""
    kind=""
    path="">

    <note
       lang="en-ca">
    </note>

    <displayName
       lang="default">
    </displayName>

</element>
```

Entity Attributes and Children

You can extend entities to add additional children. Entity base attributes include all base element attributes, and the following:

- @resource
- extends (relationSingle)
- project (relationSingle)

Element Lists

Child elements are contained in a list even when there is only one occurrence. The content of an element list is always an element. The elements in the list are usually of the same type. Mixing element types in a list is supported as well, and the element type attribute can be used to filter for specific element types when the list includes mixed types.

Nested entity and elements have the following modeling pattern:

Example 4-2 Element Lists

```
<element…>
   <elementList1 …>
      <element…>
         <elementList2 …>
            <element…/>
         </elementList2>
         <elementList3 …>
            <element…/>
         </elementList3>
      </element>
   </elementList1>
</element>
```

For nested elements, parent element details exist two levels up from the current element.

Element list cardinality indicates whether an element list is mandatory or optional. For zero or more elements, the element container is defined with a **0..1** cardinality. For zero or one elements, the element container element is defined with a **1..1** cardinality. Lists will always contain at least one element.

In Table 4-1:

- Overall cardinality is the effective combined cardinality of the list element and its child elements.
- The list element cardinality represents the first level of containment and indicates whether the list element is required or optional. List elements are elements with a specific type of child element. There is, at most, one occurrence of a list element in a parent element.
- The child element cardinality indicates the number of instances of a child element that can appear in the list element. The child element is an instance of a standard element and its type is appropriate for the list element in which it is contained. An instance of a list element will always contain at least one child element. If the list is empty, the list element is not present.

Table 4-1 Element List Cardinality

Overall Cardinality	List Element Cardinality	Child Element Cardinality
0..1	0..1 (Optional)	1..1
1..1	1..1 (Required)	1..1
0..*	0..1 (Optional)	1..*
1..*	1..1 (Required)	1..*

Relation Attributes

The **target** and **inEntity** attributes provide contextual information from the target entity or element.

The **inEntity** attribute is optional. This element appears only when the relation is to an element (the **kind** attribute is defined as **Element**).

Relation attributes include the following:

- @ref (represents an entity or element ID)

  @type

  @name

  @kind (Project, Entity, Element)
- target

  @name

  @typeName
- inEntity

  @name

  @typeName

Example 4-3 Relation Attributes

```
<relation
    ref=""
    type=""
    name=""
    kind="">

    <target
        name=""
        typeName=""/>

    <inEntity
        name=""
        typeName=""/>

</relation>
```

Named Relation Lists

Lists of named relations can appear in elements and entities. A list can be optional or required and can contain one or more relations. The objects that can be referenced by a list are projects, entities, and elements.

Each relation list is typed as:

- **relationSingle**: This list has one child defined as type **relation**.
- **relationMultiple**: This list has multiple children defined as type **relation**.

The relation list cardinality defines whether the relation list is mandatory or optional. For zero or more relations, the relation container element is defined with **0..1** cardinality and with the type **relationMultiple**. For zero or one relations, the relation container element is defined with a **0..1** cardinality and with a type of **relationSingle**. Lists always contain at least one element.

Example 4-4 Named Relation Lists

```
<entity …>
   <relationList1 …>
      <relation…/>
   </relationList1>

   <relationList2 …>
      <relation…/>
      <relation…/>
   </relationList2>

   <elementList1 …>
      <element…>
         <relationList3 …>
            <relation…/>
         </relationList3>
      </element>
   </elementList1>
</entity>
```

5

Extending Design Studio

This chapter describes how to extend Oracle Communications Service Catalog and Design - Design Studio using action commands. It provides the identifiers necessary to make additions to components in the Design Studio user interface. Also, this chapter describes how to extend Design Studio by adding custom logic to builds.

About Extending Design Studio

You can extend the functionality of Design Studio. For example, you create custom report designs that are included in the Design Studio installation and you can create custom action commands that you add to a Design Studio menu. You can extend Design Studio functionality by working directly with the Design Studio Exchange Format published XML files or by using the Design Studio Model Java API to access the information in the Exchange Format.

Extending Design Studio with Action Commands

Action commands are custom menu commands that you can add to Design Studio menus and toolbars. When extending Design Studio with action commands, you use the following extension points:

- org.eclipse.ui.commands
- org.eclipse.ui.menus
- eclipse.core.extensions.propertyTester

See the Eclipse *Platform Developer Guide* for more information about these extension points.

The **Design Studio Action Command** example includes details about these extension points. See "About the design.studio.example.action.command Example Project" for more information about the Action Command Examples project.

Adding the Design Studio Action Command Example to a Workspace

Design Studio includes the **Design Studio Action Command** example, which includes example projects that demonstrate how to extend Design Studio with action commands. These example projects are included in the Design Studio installation and can be added to your workspace.

To add the **Design Studio Action Command** example projects to a workspace:

1. From the Design Studio **File** menu, select **New**, and then select **Example**.

   The New Example wizard appears.

2. Expand the **Design Studio Examples** folder and select **Design Studio Action Command Examples**.
3. Click **Next**.

The Example Projects page appears. The **Design Studio Action Command** example includes three example projects.

4. Click each of the following example projects to read a summary of the example project:
   - The **design.studio.example.action. command.update.site** project creates a project that demonstrates how to export installable features into an update site.
   - The **design.studio.example.action.command.feature** project creates a project that demonstrates how action commands can be packaged into a feature for installation into Design Studio.
   - The **design.studio.example.action.command** project creates a project that contains sample action commands that you can add to Design Studio.
5. Click **Finish**.

   The example projects are added to the current workspace.

About the design.studio.example.action.command Example Project

The **design.studio.example.action.command** example project includes a **plug.xml** file that illustrates how to create actions that appear in the user interface and that leverage the information published by the Design Studio Exchange Format.

> **Note:**
>
> The examples presented in this chapter are displayed in text form, such as that displayed on the **plugin.xml** tab of the Plug-in Manifest editor. You can configure extensions in the Plug-in Manifest editor using the form-based representation that appears on the **Extensions** tab as well. The **plugin.xml** tab and the **Extensions** tab display two views of the same information.

The **design.studio.example.action.command** example project illustrates how to complete the following tasks:

- Adding Commands to the Studio Menu
- Adding Commands to the Design Studio Toolbar
- Adding Commands to the Solution View Context Menu
- Adding Commands to the Studio Projects View Context Menu
- Adding Commands to the Package Explorer View Context Menu
- Adding Commands to the Project Explorer View Context Menu
- Configuring the Visibility of Commands Using the Property Tester
- Configuring the Visibility of Commands Using the File Extension of Resources
- Obtaining the Model From a Resource Using the Design Studio Model Java API
- Obtaining the Model From an Entity Relation Using the Design Studio Model Java API

- Obtaining the Model From an Element Relation Using the Design Studio Model Java API

Adding Commands to the Studio Menu

The **Design Studio Action Command Examples** project includes an example that demonstrates how to add a command to the **Studio** menu.

Adding Commands to the Studio Menu displays an example of the configuration of the extensions for the following command class:

```
design.studio.example.action.command.handler.StudioMenuCommandHandler
```

In Example 5-1, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Action Command Examples** project for more information.

Example 5-1 Adding Commands to the Studio Menu

```
<extension
   point="org.eclipse.ui.commands">
   <command
      defaultHandler=
       "design.studio.example.action.command.handler.StudioMenuCommandHandler"
      id="design.studio.example.action.command.studioMenuCommand.command"
      name="Studio menu command">
    </command>
</extension>

<extension
   point="org.eclipse.ui.menus">
    <menuContribution
      locationURI="menu:studioMenu?after=perspective">
      <command
        commandId="design.studio.example.action.command.studioMenuCommand.command"
        mnemonic="%contributions.menu.studioMenuCommand.mnemonic"
        icon="icons/sample.gif"
        id="design.studio.example.action.command.studioMenuCommand.command">
      </command>
    </menuContribution>
</extension>
```

Adding Commands to the Design Studio Toolbar

The **Design Studio Action Command Examples** project includes an example that demonstrates how to add a command to the Design Studio toolbar.

Example 5-2 displays an example of the configuration of the extensions for the command class:

```
design.studio.example.action.command.handler.StudioMenuCommandHandler.
```

In Example 5-2, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Action Command Examples** project for more information.

Example 5-2 Adding Commands to Design Studio Toolbar

```
<extension
   point="org.eclipse.ui.commands">
   <command defaultHandler=
```

```
     "design.studio.example.action.command.handler.StudioMenuCommandHandler"
      id="design.studio.example.action.command.studioMenuCommand.command"
      name="Studio Menu Command">
   </command>
</extension>

<extension
   point="org.eclipse.ui.menus">
    <menuContribution
     locationURI="toolbar:org.eclipse.ui.main.toolbar">
     <toolbar
      id="design.studio.example.action.command.toolbar">
      <command
        commandId=
        "design.studio.example.action.command.studioMenuCommand.command"
         id="design.studio.example.action.command.menu.studioMenuCommand.command"
         mnemonic="%contributions.menu.studioMenuCommand.mnemonic"
         icon="icons/sample.gif"
         tooltip="Studio menu command">
       </command>
      </toolbar>
    </menuContribution>
</extension>
```

Adding Commands to the Solution View Context Menu

The **Design Studio Action Command Examples** project includes an example that demonstrates how to add a command to the Solution view context menu. In the example, the command appears when a Resource, Location, or Product entity is selected in the Solution view.

Example 5-3 displays an example of the configuration of the extensions for the command class:

```
design.studio.example.action.command.handler.GetModelFromResourceCommandHa
ndler
```

and for the property tester class:

```
design.studio.example.action.command.propertytesters.ResourceTester
```

In Example 5-3, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Action Command Examples** project for more information.

Example 5-3 Adding Commands to the Solution View Context Menu

```
<extension
  point="org.eclipse.ui.commands">
   <command defaultHandler=
      "design.studio.example.action.command.handler.
       GetModelFromResourceCommandHandler"
     id="design.studio.example.action.command.getModelFromResource.command"
     name="Get Conceptual Model">
   </command>
</extension>

<extension
  point="org.eclipse.ui.menus">
   <menuContribution
        locationURI="popup:oracle.communications.sce.ui.solution.view?
```

```
after=additions">
<command
  commandId=
   "design.studio.example.action.command.getModelFromResource.command"
  mnemonic="%contributions.menu.getModelFromResource.mnemonic"
  tooltip="Get conceptual model from resource"
  icon="icons/sample.gif"
  id=
    "design.studio.example.action.command.menu.getModelFromResource.command">
      <visibleWhen
        checkEnabled="false">
          <iterate
            operator="or">
              <adapt
                  type="org.eclipse.core.resources.IResource">
                <or>
                  <test
                    forcePluginActivation="true"
                    property=
                      "design.studio.example.action.command.
                       propertytesters.isProduct">
                  </test>
                  <test
                     forcePluginActivation="true"
                     property=
                       "design.studio.example.action.command.
                        propertytesters.isResource">
                  </test>
                  <test
                     forcePluginActivation="true"
                     property=
                       "design.studio.example.action.command.
                        propertytesters.isLocation">
                  </test>
                </or>
              </adapt>
          </iterate>
      </visibleWhen>
    </command>
  </menuContribution>
</extension>

<extension
   point="org.eclipse.core.expressions.propertyTesters">
   <propertyTester
     class=
      "design.studio.example.action.command.propertytesters.ResourceTester"
     id="design.studio.example.action.command.propertytesters.resourceTester"
     namespace="design.studio.example.action.command.propertytesters"
     properties="isResource,isProduct,isLocation"
     type="org.eclipse.core.resources.IResource">
   </propertyTester>
</extension>
```

Adding Commands to the Studio Projects View Context Menu

The **Design Studio Action Command Examples** project includes an example that demonstrates how to add a command to the Studio Projects view context menu. In the example, the command appears when a Product entity is selected in the Studio Projects view.

Example 5-4 displays an example of the configuration of the extensions for the following command class:

```
design.studio.example.action.command.handler.GetProductModelFromResourceCo
mmandHandler
```

and for the property tester class:

```
design.studio.example.action.command.propertytesters.ResourceTester
```

In Example 5-4, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Action Command Examples** project for more information.

Example 5-4 Adding Commands to the Studio Projects View Context Menu

```
<extension
  point="org.eclipse.ui.commands">
   <command
     defaultHandler=
       "design.studio.example.action.command.handler.
        GetProductModelFromResourceCommandHandler"
     id="design.studio.example.action.command.
         getProductModelFromResource.command"
     name="Get Product Model">
   </command>
</extension>

<extension
  point="org.eclipse.ui.menus">
   <menuContribution
     locationURI=
     "popup:com.mslv.studio.view.StudioView?before=common-additions">
   <command
      commandId=
        "design.studio.example.action.command.
         getProductModelFromResource.command"
      mnemonic="%contributions.menu.getProductModelFromResource.mnemonic"
      tooltip="Get product model from resource"
      icon="icons/sample.gif"
      id=
        "design.studio.example.action.command.
         menu.getProductModelFromResource.command">
      <visibleWhen
         checkEnabled="false">
           <iterate
              operator="or">
             <adapt
                type="org.eclipse.core.resources.IResource">
               <test
                  forcePluginActivation="true"
                       property=
                         "design.studio.example.action.command.
                          propertytesters.isProduct">
               </test>
             </adapt>
           </iterate>
         </visibleWhen>
       </command>
    </menuContribution>
</extension>
```

```
<extension
   point="org.eclipse.core.expressions.propertyTesters">
   <propertyTester
   class=
     "design.studio.example.action.command.propertytesters.ResourceTester"
   id="design.studio.example.action.command.propertytesters.resourceTester"
   namespace=
     "design.studio.example.action.command.propertytesters"
   properties="isResource,isProduct,isLocation"
   type="org.eclipse.core.resources.IResource">
 </propertyTester>
</extension>
```

Adding Commands to the Package Explorer View Context Menu

The **Design Studio Action Command Examples** project includes an example that demonstrates how to add a command to the Package Explorer view context menu.

Example 5-5 displays an example of the configuration of the extensions for the following command class:

```
design.studio.example.action.command.handler.StudioMenuCommandHandler
```

In Example 5-5, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Action Command Examples** project for more information.

Example 5-5 Adding Commands to the Package Explorer View Context Menu

```
<extension
  point="org.eclipse.ui.commands">
  <command
    defaultHandler=
      "design.studio.example.action.command.handler.StudioMenuCommandHandler"
    id="design.studio.example.action.command.studioMenuCommand.command"
    name="Studio Menu Command">
  </command>
</extension>

<extension
  point="org.eclipse.ui.menus">
  <menuContribution
     locationURI="popup:org.eclipse.jdt.ui.PackageExplorer">
     <command
        commandId=
          "design.studio.example.action.command.studioMenuCommand.command"
        mnemonic="%contributions.menu.studioMenuCommand.mnemonic"
        tooltip="Studio menu command"
        icon="icons/sample.gif"
        id=
          "design.studio.example.action.command.menu.studioMenuCommand.command">
     </command>
  </menuContribution>
</extension>
```

Adding Commands to the Project Explorer View Context Menu

The **Design Studio Action Command Examples** project includes an example that demonstrates how to add a command to the Project Explorer view context menu.

Example 5-6 displays an example of the configuration of the extensions for the following command class:

```
design.studio.example.action.command.handler.StudioMenuCommandHandler
```

In Example 5-6, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Action Command Examples** project for more information.

Example 5-6 Adding Commands to the Project Explorer View Context Menu

```
<extension
  point="org.eclipse.ui.commands">
  <command
    defaultHandler=
      "design.studio.example.action.command.handler.StudioMenuCommandHandler"
    id="design.studio.example.action.command.studioMenuCommand.command"
    name="Studio Menu Command">
  </command>
</extension>

<extension
  point="org.eclipse.ui.menus">
  <menuContribution
     locationURI="popup:org.eclipse.ui.navigator.ProjectExplorer#PopupMenu">
     <command
        commandId=
          "design.studio.example.action.command.studioMenuCommand.command"
        mnemonic="%contributions.menu.studioMenuCommand.mnemonic"
        tooltip="Studio menu command"
        icon="icons/sample.gif"
        id=
          "design.studio.example.action.command.menu.studioMenuCommand.command">
     </command>
  </menuContribution>
</extension>
```

Configuring the Visibility of Commands Using the Property Tester

The **Design Studio Action Command Examples** project includes an example that demonstrates how to configure the visibility of commands using the property tester. In the example, the command appears when a Resource entity is selected in the Studio Projects view.

Example 5-7 displays an example of the configuration of the extensions for the following command class:

```
design.studio.example.action.command.handler.GetResourceModelFromResourceC
ommandHandler
```

and for the property tester class:

```
design.studio.example.action.command.propertytesters.ResourceTester
```

In Example 5-7, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Action Command Examples** project for more information.

Example 5-7 Configuring the Visibility of Commands Using the Property Tester

```
<extension
  point="org.eclipse.ui.commands">
  <command
     defaultHandler=
       "design.studio.example.action.command.handler.
        GetResourceModelFromResourceCommandHandler"
     id=
       "design.studio.example.action.command.
        getResourceModelFromResource.command"
     name="Get Resource Model">
  </command>
</extension>

<extension
   point="org.eclipse.ui.menus">
   <menuContribution
      locationURI=
        "popup:com.mslv.studio.view.StudioView?before=common-additions">
      <command
         commandId=
           "design.studio.example.action.command.
            getResourceModelFromResource.command"
         mnemonic="%contributions.menu.getResourceModelFromResource.mnemonic"
         tooltip="Get resource model from resource"
         icon="icons/sample.gif"
         id=
           "design.studio.example.action.command.menu.
            getResourceModelFromResource.command">
         <visibleWhen
            checkEnabled="false">
            <iterate
                operator="or">
              <adapt
                  type="org.eclipse.core.resources.IResource">
                  <test
                     forcePluginActivation="true"
                     property=
                       "design.studio.example.action.command.
                        propertytesters.isResource">
                  </test>
              </adapt>
            </iterate>
          </visibleWhen>
        </command>
     </menuContribution>
</extension>

<extension
   point="org.eclipse.core.expressions.propertyTesters">
   <propertyTester
      class=
        "design.studio.example.action.command.
         propertytesters.ResourceTester"
      id="design.studio.example.action.command.propertytesters.resourceTester"
      namespace="design.studio.example.action.command.propertytesters"
      properties="isResource,isProduct,isLocation"
      type="org.eclipse.core.resources.IResource">
   </propertyTester>
</extension>
```

Configuring the Visibility of Commands Using the File Extension of Resources

The **Design Studio Action Command Examples** project includes an example that demonstrates how to configure the visibility of commands using the file extension of a resource. In the example, the command appears when a Location entity is selected in the Studio Projects view.

Example 5-8 displays an example of the configuration of the extensions for the following command class:

```
design.studio.example.action.command.handler.GetLocationModelFromResourceC
ommandHandler
```

In Example 5-8, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Action Command Examples** project for more information.

Example 5-8 Configuring the Visibility of Commands Using the File Extension of Resources

```
<extension
   point="org.eclipse.ui.commands">
   <command
      defaultHandler=
        "design.studio.example.action.command.handler.
         GetLocationModelFromResourceCommandHandler"
      id=
        "design.studio.example.action.command.
         getLocationModelFromResource.command"
      name="Get Location Model">
   </command>
</extension>

<extension
   point="org.eclipse.ui.menus">
   <menuContribution
     locationURI=
       "popup:com.mslv.studio.view.StudioView?before=common-additions">
     <command
        commandId=
           "design.studio.example.action.command.
            getLocationModelFromResource.command"
        mnemonic="%contributions.menu.getLocationModelFromResource.mnemonic"
        tooltip="Get location model from resource"
        icon="icons/sample.gif"
        id=
           "design.studio.example.action.command.menu.
            getLocationModelFromResource.command">
        <visibleWhen
              checkEnabled="false">
           <iterate
                 operator="or">
              <adapt
                 type="org.eclipse.core.resources.IResource">
                 <test
                    property="org.eclipse.core.resources.name"
                    value="*.cmnLocation">
                 </test>
              </adapt>
```

```
          </iterate>
        </visibleWhen>
      </command>
    </menuContribution>
</extension>
```

Obtaining the Model From a Resource Using the Design Studio Model Java API

The **Design Studio Action Command Examples** project includes an example that demonstrates how to obtain a model configuration from a resource using the Design Studio Model Java API.

Example 5-9 displays an example of the configuration of the extensions for the following command class:

```
design.studio.example.action.command.handler.GetLocationModelFromResourceCommand
Handler
```

In Example 5-9, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Action Command Examples** project for more information.

Example 5-9 Obtaining the Model From a Resource Using the Design Studio Model Java API

```
<extension
   point="org.eclipse.ui.commands">
   <command
      defaultHandler=
         "design.studio.example.action.command.handler.
          GetLocationModelFromResourceCommandHandler"
      id=
         "design.studio.example.action.command.
          getLocationModelFromResource.command"
      name="Get Location Model">
   </command>
</extension>

<extension
   point="org.eclipse.ui.menus">
     <menuContribution
        locationURI=
           "popup:com.mslv.studio.view.StudioView?before=common-additions">
        <command
           commandId=
              "design.studio.example.action.command.
               getLocationModelFromResource.command"
           mnemonic="%contributions.menu.getLocationModelFromResource.mnemonic"
           tooltip="Get location model from resource"
           icon="icons/sample.gif"
           id=
              "design.studio.example.action.command.menu.
               getLocationModelFromResource.command">
           <visibleWhen
                 checkEnabled="false">
              <iterate
                    operator="or">
                 <adapt
                       type="org.eclipse.core.resources.IResource">
                       <test
                             property="org.eclipse.core.resources.name"
```

```
                                value="*.cmnLocation">
                            </test>
                        </adapt>
                    </iterate>
                </visibleWhen>
            </command>
        </menuContribution>
</extension>
```

Obtaining the Model From an Entity Relation Using the Design Studio Model Java API

The **Design Studio Action Command Examples** project includes an example that demonstrates how to obtain a model configuration from an entity relation using the Design Studio Model Java API.

Example 5-10 displays an example of the configuration of the extensions for the following command class:

```
design.studio.example.action.command.handler.GetCreationTaskModelFromRelat
ionCommandHandler
```

In Example 5-10, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Action Command Examples** project for more information.

Example 5-10 Obtaining the Model From an Entity Relation Using the Design Studio Model Java API

```
<extension
    point="org.eclipse.ui.commands">
    <command
        defaultHandler=
           "design.studio.example.action.command.handler.
            GetCreationTaskModelFromRelationCommandHandler"
        id=
           "design.studio.example.action.command.
            getCreationTaskModelFromOrder.command"
        name="Get Creation Task Model From Order">
    </command>
</extension>

<extension
    point="org.eclipse.ui.menus">
       <menuContribution
           locationURI=
             "popup:com.mslv.studio.view.StudioView?before=common-additions">
          <command
              commandId=
                "design.studio.example.action.command.
                 getCreationTaskModelFromOrder.command"
              mnemonic=
               "%contributions.menu.
                 getCreationTaskModelFromOrderRelation.mnemonic"
              tooltip="Get Creation Task Model From Order"
              icon="icons/sample.gif"
              id=
                "design.studio.example.action.command.
                 getCreationTaskModelFromOrder.command">
              <visibleWhen
```

```
                    checkEnabled="false">
                  <iterate
                        operator="or">
                     <adapt
                         type="org.eclipse.core.resources.IResource">
                         <test
                              property="org.eclipse.core.resources.name"
                              value="*.order">
                         </test>
                     </adapt>
                  </iterate>
               </visibleWhen>
            </command>
         </menuContribution>
</extension>
```

Obtaining the Model From an Element Relation Using the Design Studio Model Java API

The **Design Studio Action Command Examples** project includes an example that demonstrates how to obtain a model configuration from an element relation using the Design Studio Model Java API.

Example 5-11 displays an example of the configuration of the extensions for the following command class:

```
design.studio.example.action.command.handler.GetOrderItemActionPropertyFromRelat
ionCommandHandler
```

In Example 5-11, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Action Command Examples** project for more information.

Example 5-11 Obtaining the Model From an Element Relation Using the Design Studio Model Java API

```
<extension
    point="org.eclipse.ui.commands">
    <command
        defaultHandler=
          "design.studio.example.action.command.handler.
           GetOrderItemActionPropertyFromRelationCommandHandler"
        id=
           "design.studio.example.action.command.
            getOrderItemActionPropertyFromOrderItemSpec.command"
        name="Get Order Item Action Property From Order Item Specification">
    </command>
</extension>

<extension
    point="org.eclipse.ui.menus">
    <menuContribution
        locationURI=
           "popup:com.mslv.studio.view.StudioView?before=common-additions">
        <command
            commandId=
              "design.studio.example.action.command.
               getOrderItemActionPropertyFromOrderItemSpec.command"
            mnemonic=
               "%contributions.menu.
```

```
                getOrderItemPropertyModelFromRelation.mnemonic"
            tooltip="Get Order Item Action Property From Order Item
Specification"
            icon="icons/sample.gif"
            id=
               "design.studio.example.action.command.
                getOrderItemActionPropertyFromOrderItemSpec.command">
            <visibleWhen
                 checkEnabled="false">
              <iterate
                    operator="or">
                 <adapt
                       type="org.eclipse.core.resources.IResource">
                       <test
                             property="org.eclipse.core.resources.name"
                             value="*.orderItemSpec">
                       </test>
                 </adapt>
              </iterate>
           </visibleWhen>
        </command>
    </menuContribution>
</extension>
```

About Design Studio View Identifiers

Each Design Studio component is defined with an identifier that you can use when you want to make additions to that component. For example, you can reference the Solution view identifier in Table 5-1 when adding an action command to the Solution view context menu.

Table 5-1 Design Studio View Identifiers

Component	Identifier
Menu	menu:studioMenu?after=perspective
Toolbar	toolbar:org.eclipse.ui.main.toolbar
Solution View Context Menu	popup:oracle.communications.sce.ui.solution.view?after=additions
Studio Project View Context Menu	popup:com.mslv.studio.view.StudioView?before=common-additions
Project Explorer Context Menu	popup:org.eclipse.ui.navigator.ProjectExplorer#PopupMenu
Package Explorer Context Menu	popup:org.eclipse.jdt.ui.PackageExplorer

Adding Custom Logic to Design Studio Builds

You can add custom logic to Design Studio builds by creating an implementation of the IModelProcessor interface, defined in:

oracle.communications.studio.model.processor

You can use model processors to generate custom artifacts to integrate with external systems, product catalogs, reporting features, and so forth.

> **Note:**
>
> This task is intended for advanced users who are familiar with Eclipse plug-in development and Java coding. The **Design Studio Action Command Example** project includes a simple implementation named **ModelProcessorExample.java** in the **design.studio.example.model.processor** package.
>
> See "About the design.studio.example.action.command Example Project" for more information.

To extend Design Studio by adding custom logic to the build:

1. Create a plug-in project.

   See "Creating Plug-in Projects" for more information. The plug-in project **plugin.xml** file is added to the Package Explorer view.

2. Double-click the plug-in project **plugin.xml** file.

   The file opens in the Plug-in Manifest editor.

3. Click the **Extensions** tab.

4. Click **Add**.

   The New Extension dialog box appears.

5. Locate and select:

   oracle.communications.studio.model.processor

   If you don't see the **oracle.communications.studio.model.processor** value, deselect the **Show only extension points from the required plug-ins** option. If prompted to add dependencies, select **Yes**.

6. Click **Finish**.

   The extension appears in the **Extension** area.

7. Select the extension.

8. In the **Extension Details** area, click **Show extension point description**.

   The Design Studio Model Processor documentation page appears. Use this documentation to finish creating an implementation of the IModelProcessor interface.

6
Working with Reports

This chapter provides information about reports. It provides an overview of Oracle Communications Service Catalog and Design - Design Studio reporting, explains how to create your own report designs in Design Studio, and provides information about packaging, testing, and distributing reports. This chapter also provides information about extending the Design Studio reporting functionality.

About Design Studio Reports

Design Studio enables you to create and generate reports that include information about an implemented solution at a specific point in time. For example, a report can summarize the structure of the solution by listing projects and dependencies, or a report can summarize the composition of a service. Reports can capture the names, types, descriptions, and relationships of projects, entities, and data elements.

You can use reports to facilitate activities in a solution life cycle. For example, you can generate reports to facilitate an approval process or you can use reports as supporting documentation. Report output formats use standard file formats (such as PDF and HTML) familiar to and usable by team members who may not have a Design Studio installation.

Design Studio reports are comprised of data, data transformations, business logic, and data presentation.

- Design Studio reports include data that you model in Design Studio.
- Data transformation features enable you to sort, summarize, and filter Design Studio data. For example, you can perform operations such as grouping sums, calculating percentages of overall totals, and so forth.
- Business logic features enable you to convert raw data to information required by a user.
- Data presentation features enable you to present the data in specific ways, such as in tables, charts, or as text.

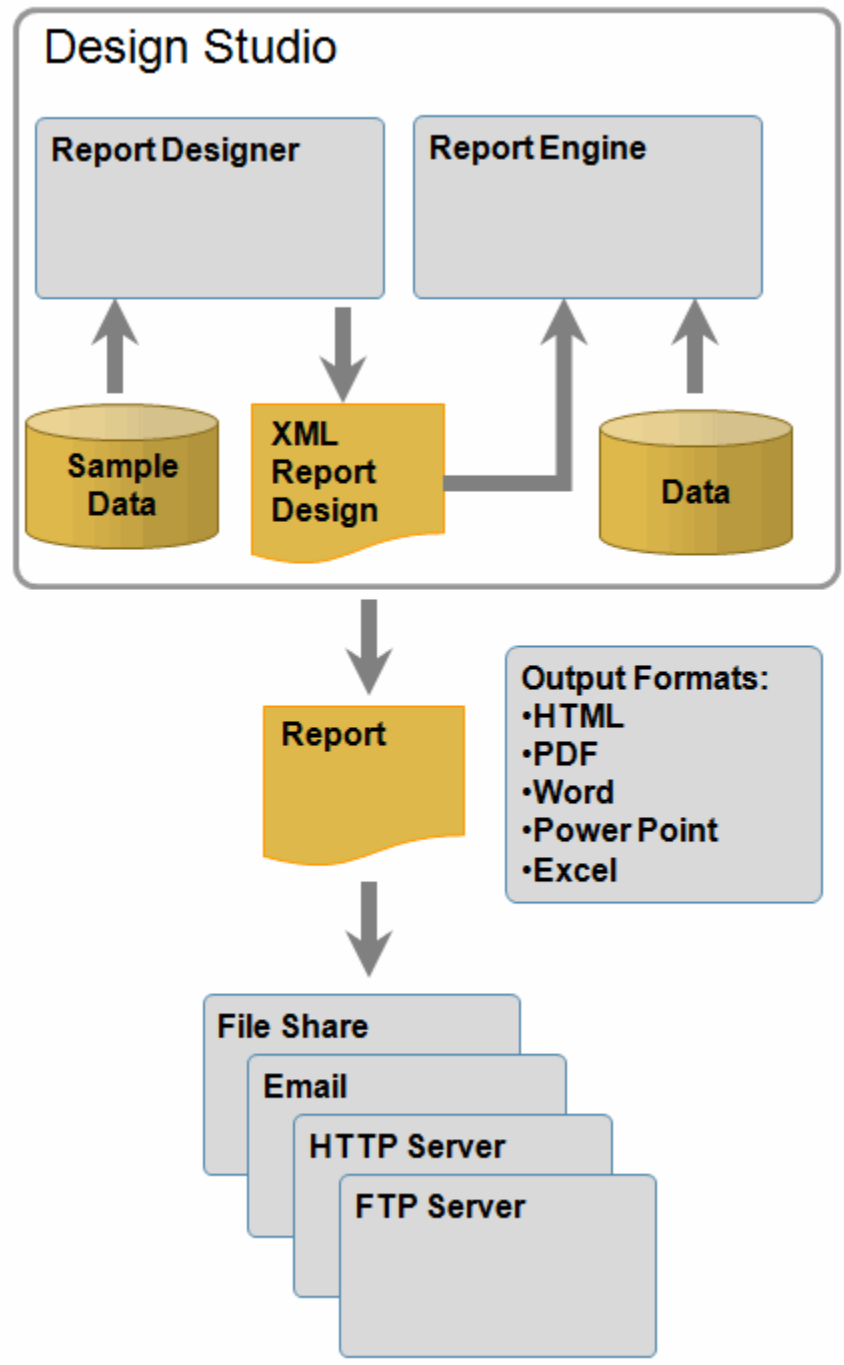
About the Design Studio Reporting Architecture

The Design Studio reporting architecture includes a Report Designer and a Reporting Engine.

- You use the Report Designer to create new reports, customize existing reports, and test reports (using static sample data from an XML file). The Report Designer provides a complete environment for designing and testing reports, and includes a report design perspective and a set of views and editors. The Report Designer is installed with Design Studio but is also available as a stand-alone application. You access the tools included in the Report Designer by switching to the Report Design perspective in Design Studio.
- You use the Report Engine when generating reports. You use a Design Studio wizard that enables you to communicate to the Report Engine which design and layout to use, the data to include in the report, and the format in which to generate the report output.

Figure 6-1 illustrates the Design Studio reporting architecture:

Figure 6-1 Design Studio Reporting Architecture



About the Design Pattern Development Life Cycle

The life cycle of a design pattern begins with the identification and isolation of the pattern itself. Working from a reference implementation, designers identify the repeatable pattern, which comprises the resources and the relationships of the resources to the workspace.

The tasks in the life cycle of a design pattern are completed by two different actors, a designer who creates and distributes design patterns, and a user (solution designer) who installs the design patterns and then runs the patterns to facilitate solution development.

Designers do the following:

1. Evaluate common modeling tasks and key resources in reference implementations, sample solutions, and best practices. then identify which repeatable tasks can be automated in a design pattern.
2. Develop design patterns using the identified resources as key components of the design patterns.
3. Test design patterns by running them in the Design Studio environment.
4. Include design patterns in plug-in projects, associate the plug-in project to a feature project, and associate the feature project to an update site.
5. Distribute update sites to Design Studio users.

Design pattern users do the following:

1. Install features based on their role and objectives.
2. Run design patterns to assist with solution design.

About Report Designs

Each report that you can generate in Design Studio is backed by a report design XML file. A report design file is a template that describes the layout and style of the report. It also includes information about how to obtain the required reporting data and how to filter the available reporting data.

The report design includes information that the Report Engine can use to connect to the source of the report data. The report design also includes information that specifies which subsets of data to include in the report. Queries obtain the data from the data source and the Report Engine maps the data to a table.

When developing report designs, consider that report layouts and styles may not be suitable for some output formats. Also, users generating reports may select an output format that is not best-suited for a particular report design. In these cases, aspects of the report design may be ignored or adapted to fit the output format presentation capabilities. Test the presentation properties of your output formats to ensure that you understand the report design layout nuances.

During report development and testing, you can save report designs to a local directory, and generate reports from a locally saved report design file. Report design files use the file extension **.rptdesign**.

If you require a report design to display in a free-flowing output, such as HTML, and to paginated output, such as PDF, considering creating two distinct report designs. When designing for free-flow output, use percentage-based proportions in the design (for example, for column width). Use specific sizes when designing for paginated output formats.

About the Report Designer

You create new report designs using the Business Intelligence and Reporting Tools (BIRT) Report Designer. BIRT is an open source Eclipse project that provides a complete environment for designing and testing reports. You access the tools included in the BIRT

Report Designer by switching to the Report Design perspective in Design Studio. See Design Studio Help for information about switching perspectives.

The Report Design perspective includes editors, views, and tools to build reports quickly. Some of these are listed below:

- The Layout editor, which enables drag and drop creation of your report presentation.
- The Data Explorer view, which you use to manage information about your report data, such as how to connect to the source and how to filter for only the data that is relevant to your report.
- The Palette view, which includes report elements such as Data, Image, and Table, which you can drag to the Layout editor to design the structure of reports.
- The Resource Explorer view, which you use to view all libraries and shared content, such as images and script files. The libraries store reporting objects and enable you to reuse existing report objects, such as tables and styles.

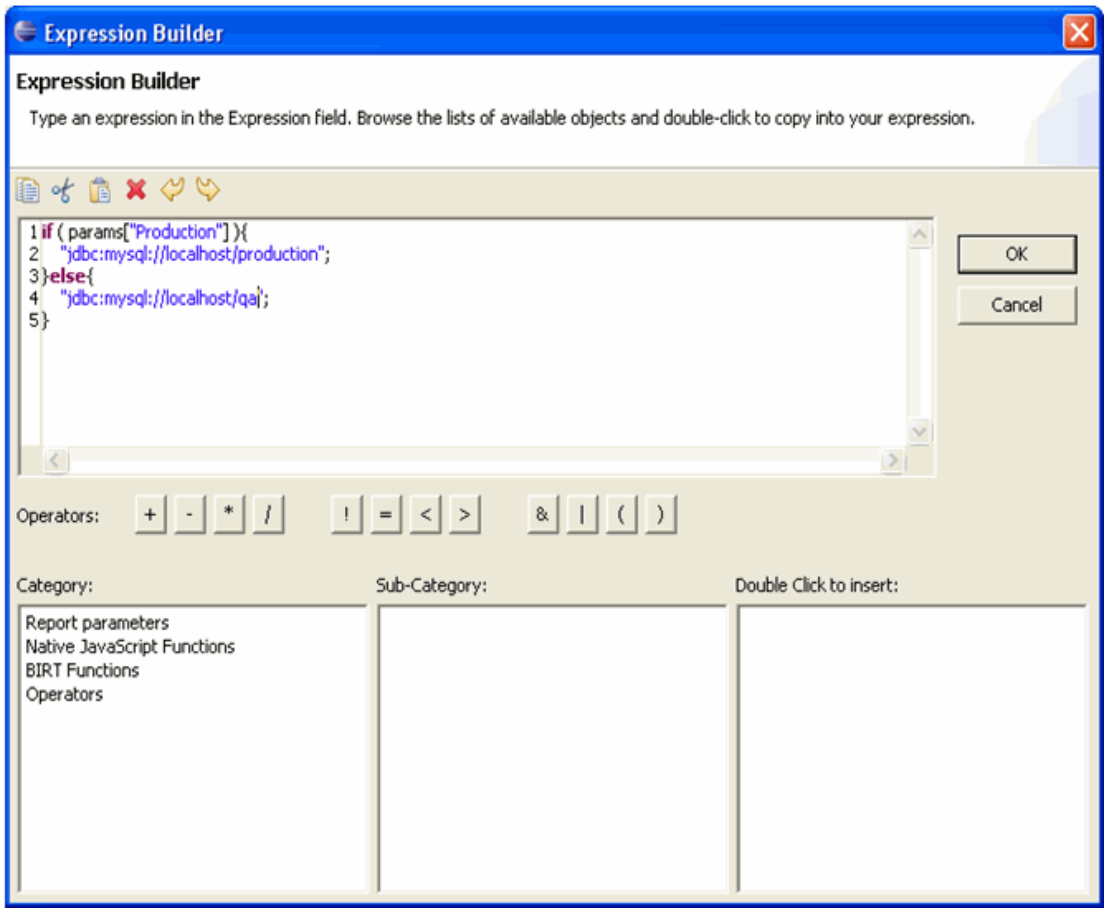
For more information about the Report Design perspective, see the BIRT Documentation page:

http://www.eclipse.org/birt/documentation/

About the Expression Builder

The BIRT Expression Builder is an editor that you can use to create complex expressions. These expressions can include functions, data, conditions, and operations.

Figure 6-2 Expression Builder



You can use the Expression Builder when:

- Creating the display value for a report item
- Creating a computed field in the Data Explorer view
- Specifying a data series for charts
- Specifying filter conditions
- Specifying mapping conditions
- Specifying highlight conditions
- Specifying group keys
- Specifying hyperlinks
- Specifying URIs for images
- Specifying dynamic data in text controls

About Report Generation

You can generate reports on-demand using a Design Studio wizard or you can automate report generation using an automated script in a continuous integration framework.

Report generation is a long-running task, and the time required to compose and render a report can vary from report to report. Processing time is dependent on a number of factors, including the amount of data supporting the report, the complexity of the report, and design of data sets and presentation.

Automated reports can be distributed to web servers, file shares, email, FTP sites, and by other methods by using Ant tasks to perform actions after report generation. These tasks are not explicitly provided by Design Studio but are a supported by Ant. For example, you can publish reports to a shared location or to website; you can email the report to a distribution list, and so forth. See Apache Ant Project website for more information.

See *Design Studio System Administrator's Guide* for more information about using Ant tasks in automated builds.

About Data Sources

A data source is a place from which a report obtains information. The information required to connect to the data source is defined in a Data Source entity. For example, you can create a Data Source entity if your reports use information defined in a database, a text file, an XML file, or a web service.

You can create multiple data sources for a single report. Each type of data source requires different connection information. For example, a report might require data from a database and data from a file. The data source information required to retrieve data from the database is different than the data source information required to retrieve data from a file.

To access the Design Studio data that you require for generating reports, you create a Data Source entity for Design Studio. You create an XML Data Source entity for Design Studio because Design Studio provides the data in XML format.

When defining Design Studio as a data source, you specify:

- An XML data schema that describes the structure of the Design Studio data.
- A sample XML file that the reporting framework can use when you are testing report designs. The sample XML file should include data that is representative of the type of data that you intend to include in finished reports. You may need to create and reference multiple sample XML files, depending on the kinds of reports you are testing and generating.
- A reporting parameter that dynamically identifies the location of the Design Studio XML file that contains the requested reporting data. During report generation, Design Studio generates an XML file and saves it to a temporary location on the file system. This XML file includes all of the requested reporting data. The reporting framework uses this parameter to obtain the location of the XML file and to access the reporting data.

Figure 6-3 shows an XML Data Source entity and a report parameter:

Figure 6-3 XML Data Source and Report Parameter

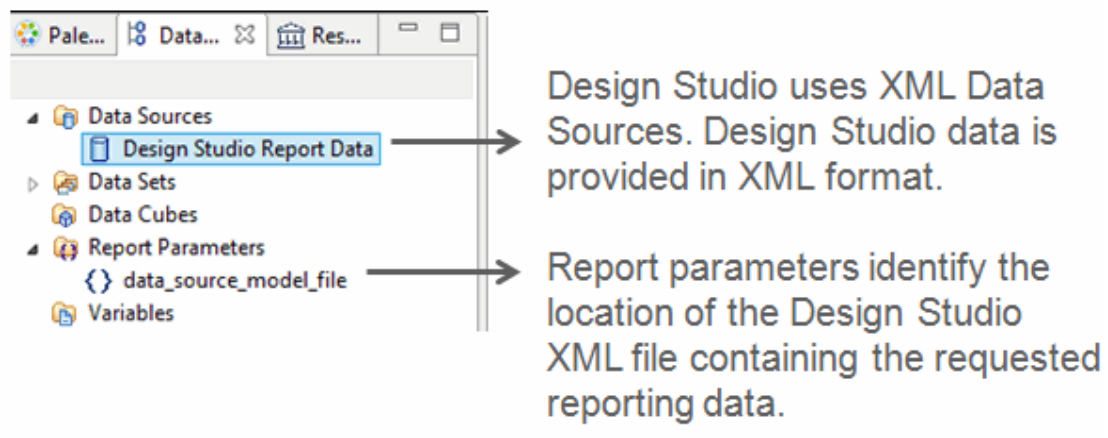
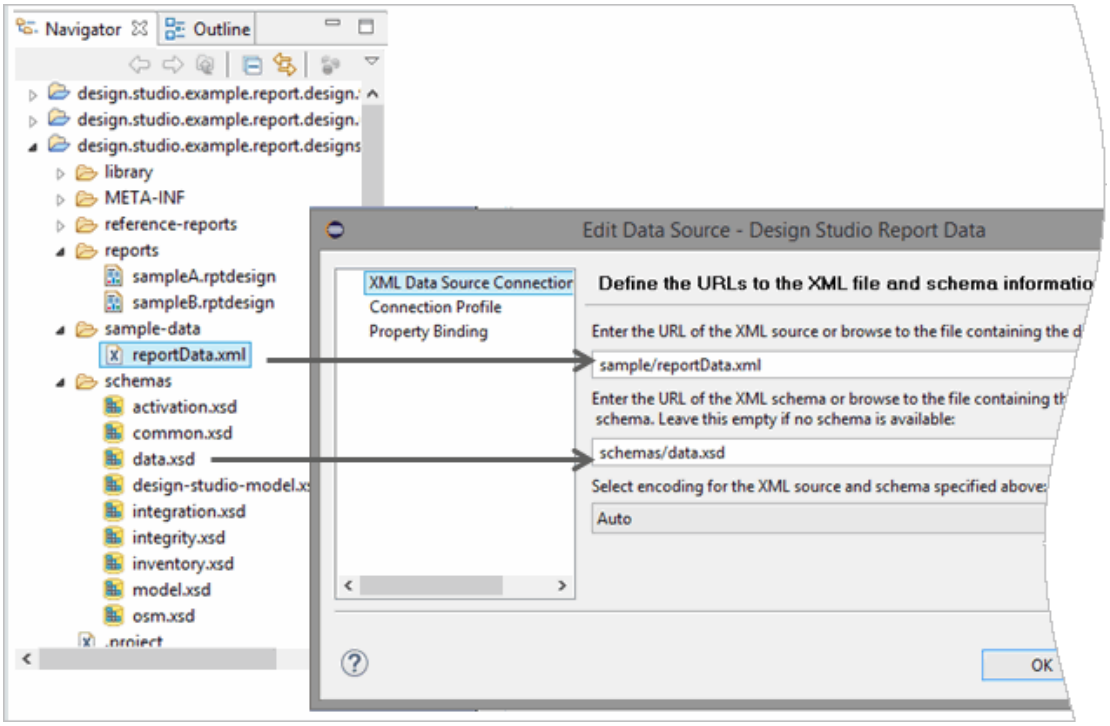


Figure 6-4 shows the Edit Data Source dialog box, where you define a sample XML file that the reporting framework can use when you are testing report designs, and the XML data schema that describes the structure of the Design Studio data:

Figure 6-4 Data Source Sample XML File and Schema



See "Creating the Design Studio Data Source Entity" for more information.

About Data Sets

A data set defines the data that is available to a report. Data sources typically contain more data fields than are needed for a report. When defining a data set, you select the data that

you want to retrieve from the data source and determine how to process that data. For example, you can change column names, create computed columns, and filter the data that appears in the report. You may require multiple data sets for a single report.

Before you can create a data set for Design Studio reports, you must first create the Data Source entity that defines how to obtain reporting information from Design Studio.

See "Creating Data Set Entities" for more information.

Adding the Report Design Example to the Workspace

Design Studio includes a report design example that you can use as a reference or as a starting point for creating your own custom report designs. This example is included in the Design Studio installation and can be added to your workspace. See "Working with the Design Studio Report Examples" for more information.

To add the report design example to the workspace:

1. From the Design Studio **File** menu select **New**, and then select **Example**.

   The New Example wizard appears.

2. Expand the **Design Studio Examples** folder and select **Design Studio Report Design Example**.

3. Click **Next**.

   The Example Projects page appears, listing each of the projects that will be added to the workspace.

4. Click an example project to view its description:

   - **design.studio.example.report.update.site** creates a project to demonstrate how to export installable features into an update site.
   - **design.studio.example.report.design.feature** creates a project to demonstrate how report designs can be packaged into a feature for installation into Design Studio.
   - **design.studio.example.report.designs** creates a project that contains a sample report design, an XML Schema, a report design library, and other supporting content.

5. Click **Finish**.

   The projects are added to the current workspace.

Customizing Existing Design Studio Reports

Design Studio includes reference reports that provide a base set of capabilities. You can use these existing Design Studio reference reports as a starting point for customizing your own reports. Design Studio also includes sample report designs that you can use as a starting point for customization.

When customizing existing reports, you might begin by selecting a report design and determining what changes you require to the presentation of the report data. For example, you may need to customize an existing report design for layout or branding.

You might determine that you also need to change the data captured in the report. For example, you may need to edit report column headings or add additional reporting

fields. You may need to add additional filters so that the report data is more specific to your needs. Changes to the data, of course, impact the presentation of the data in the report.

To customize an existing report:

1. Add the **Design Studio Report Design Example** to your workspace.

   See "Adding the Report Design Example to the Workspace" for more information.

2. In Design Studio, from the Project Explorer view, right-click the **design.studio.example.report.designs** folder and select **Copy**.

3. Right-click in the Project Explorer view white space and select **Paste**.

   The Copy Project dialog box appears.

4. In the **Project Name** field, edit the project name.

   The name must be unique in the workspace.

5. Click **OK**.

   The project appears in the Package Explorer view.

6. Expand the **design.studio.example.report.designs** folder.

7. Select a report design file to customize.

   You can customize:

   - A Design Studio reference report design file, such as the **ProjectSummary.rptdesign** file. The Design Studio reference report design files are located in the **reference-reports** folder.
   - A sample report design. The **sampleA.rptdesign** and **sampleB.rptdesign** files are located in the **reports** folder. The sample report designs are more simplistic than the reference report designs, but include packaging configuration that you can customize to install your reports into Design Studio.

8. Double-click a report design file to open the file in the Report Design editor.

9. Edit the file, as needed.

   See "Defining Data Presentation in Reports" and "Adding Additional Report Design Elements" for more information.

10. Click **Save**.

11. Test the edited report designs.

    See "Testing Report Designs" for more information.

12. Add your edited reports to the Generate Report wizard.

    See "Adding Reports and Report Categories to the Generate Report Wizard" for more information.

13. Package the edited report designs.

    See "Packaging Plug-in Projects" for more information.

14. Distribute the edited report designs.

    See "Distributing Plug-in Projects" for more information.

Developing Custom Report Designs

You develop custom report designs in Eclipse plug-in projects and plug-in projects are associated with feature projects. An Eclipse plug-in project can include any number of custom report designs. For example, you will likely include all of your custom report designs in a single plug-in project.

An Eclipse feature project can be associated with any number of plug-in projects. Feature projects are associated with update site projects. Solution designers install the features from update sites to gain access to the custom report designs that are included in the features.

See "Creating, Packaging, and Distributing Plug-in Projects" for more information about plug-in projects and features.

> **Note:**
>
> Design Studio documentation complements the existing BIRT project documentation and provides guidance for designing reports specific to Design Studio.
>
> Before creating your own custom Design Studio reports, Oracle recommends that you review the following material available from the BIRT project website:
>
> - The tutorials, available on the Tutorial page:
>
>   `http://www.eclipse.org/birt/documentation/tutorial/`
> - The demos, available on the Demo page:
>
>   `http://www.eclipse.org/birt/demos/`
> - The documentation, available on the Documentation page:
>
>   `http://www.eclipse.org/birt/documentation/`

You develop custom report designs by performing the following tasks:

1. Add the **Design Studio Report Design Example** to your workspace.

   See "Adding the Report Design Example to the Workspace" for more information.
2. Create a plug-in project.

   See "Creating Plug-in Projects" for more information.
3. Create a report design file.

   See "Creating Report Design Files" for more information.
4. Create the Design Studio report parameter.

   See "Creating Design Studio Report Parameters" for more information.
5. Create a Design Studio Data Source entity.

   See "Creating the Design Studio Data Source Entity" for more information.
6. Create a data set for the Design Studio Data Source entity.

See "Creating Data Set Entities" for more information.

7. Define the data to be added to the report.

   See "Defining the Data to Add to Reports" for more information.

8. Define the data presentation to be used in the report.

   See "Defining Data Presentation in Reports" for more information.

9. Add additional report design elements.

   See "Adding Additional Report Design Elements" for more information.

10. Test the report designs.

    See "Testing Report Designs" for more information.

11. Add your custom reports to the Generate Report wizard.

    See "Adding Reports and Report Categories to the Generate Report Wizard" for more information.

12. Package the report designs.

    See "Packaging Plug-in Projects" for more information.

13. Distribute the report designs.

    See "Distributing Plug-in Projects" for more information.

> **Note:**
>
> Test custom reports before packaging them and during various stages of report development. When developing custom report designs, test frequently and use iterative implementations to reduce issues.

Creating Report Design Files

A report design is an XML file that defines all of the information required to generate a report.

To create a report design:

1. From the Report Design perspective, select **File**, then select **New**, and then select **Other**.
2. Expand the **Business Intelligence and Reporting Tools** folder, and then select **Report**.
3. Click **Next**.
4. Select the folder in which to save the report design file.

   You must save the report design file in the plug-in project that you created for your reports. For example:

   plug-inProject/**reports**

> **Note:**
>
> Do not save report designs in Java source folders.

5. In the **File Name** field, enter the name of the new report design.

   The file name extension must be **.rptdesign**.

6. Click **Next**.

7. Select a report template.

   You can review selected template layouts in the **Preview** field.

8. (Optional) To get help designing reports, select **Show Report Creation Cheat Sheet**.

   The Cheat Sheet view provides design guidance after the wizard completes. This option is available only for a subset of reports.

9. Click **Finish**.

   Design Studio opens the report in the Layout editor.

Creating Design Studio Report Parameters

Design Studio report designs require at least one report parameter to identify the location of the XML file to be used as an XML data source. You can add additional report parameters to support custom processors.

To create a Design Studio report parameter:

1. In the Report Design perspective, open a report design in the Layout editor.

2. Click the **Data Explorer** tab.

   The Data Explorer view appears.

3. Right click **Report Parameters** and select **New Parameter**.

   The New Parameter dialog box appears.

4. In the **Name** field, enter **data_source_model_file**.

   When using Design Studio as the data source, you must name the parameter **data_source_model_file**.

5. In the **Data Type** field, select **String**.

6. Ensure that the **Is Required** option is selected.

7. Click **OK**.

8. If you intend to create custom report processors, define additional report parameters.

   When defining additional report parameters, you must deselect **Is Required** and you must provide a default value for the custom report processor parameter.

Creating the Design Studio Data Source Entity

You create a Data Source entity to define how the reporting engine obtains reporting data from Design Studio. See "About Data Sources" for more information.

> **Note:**
>
> Add the Design Studio Report Design Example to your workspace before you begin this procedure. See "Adding the Report Design Example to the Workspace" for more information.

To create a Design Studio Data Source entity:

1. In the Design Studio perspective, click **Package Explorer** tab.
2. Expand the **design.studio.example.report.designs** folder.
3. Expand the **sample-data** folder.
4. Copy the **reportData.xml** file to your report plug-in project.

   For example, you might create a sample data folder in the plug-in project:

   plug-inProject/**sampleData**/
5. Expand the **schemas** folder.
6. Copy the schemas folder and all of schema **.xsd** files in the **schemas** folder to your report plug-in project.

   For example, copy the contents of the **schemas** folder to the following location:

   plug-inProject/**schemas**/
7. Switch to the Report Design perspective and open a report design in the Layout editor.
8. Click the **Data Explorer** tab.

   The Data Explorer view appears.
9. Right click **Data Sources** and select **New Data Source**.

   The New Data Source wizard appears.
10. Select **Create from a data source type in the following list**.
11. From the list, select **XML Data Source**.
12. In the **Data Source Name**, enter a name for the data source.
13. Click **Next**.
14. Enter a URL or browse to select a sample XML file.

    You specify a sample XML file that the reporting framework can use when you are testing report designs. The sample XML file should include data that is representative of the type of data that you intend to include in finished reports. You can:

    - Enter the path to the **reportData.xml** file that is included in the **Design Studio Report Design Example**. See "Adding the Report Design Example to the Workspace" for more information.
    - Run the **Design Studio Model in XML** report to create a new sample XML file. See "Working with the Design Studio Report Examples" for more information.
15. Enter a URL to or browse to select an XML schema file.

    This sample schema file, with the sample XML file, provide the reporting engine with the Design Studio model format details that the engine required during testing.

If you haven't yet identified an XML schema, select the **design-studio-model.xsd** file that you copied to your plug-in project.

16. Click **Test Connection**.

    A dialog box appears that displays the results of the test.

17. Click **OK** and then click **Finish**.

18. In the Data Explorer view, right-click the new Data Source entity and select **Edit**.

    The Edit Data Source dialog box appears.

19. Select **Property Binding**.

20. In the **XML Data Source File** field, enter:

    params["data_source_model_file"].value

    This value binds the Design Studio data source to the file that is defined by the Design Studio report parameter. See "Creating Design Studio Report Parameters" for more information.

21. Click **OK**.

Creating Data Set Entities

Data sets specify the data to retrieve from the Design Studio Data Source entity. See "About Data Sets" for more information.

Before you create a Data Set entity, you must first create the Design Studio Data Source entity. See "Creating the Design Studio Data Source Entity" for more information.

To create a Data Set entity:

1. From the Report Design perspective, in the Data Explorer view, right-click **Data Sets** and select **New Data Set**.

   The New Data Set dialog box appears.

2. In the Data Source Selection area, expand **XML Data Source** and select the Design Studio Data Source entity to associate with the data set.

3. In the **Data Set Type** field, ensure that **XML Data Set** is selected.

4. In the **Data Set Name** field, enter a name for the data set.

5. Click **Next**.

   The Sample XML Settings page appears.

6. Select a sample XML source file that contains the type of content you expect from the data set.

   For example, select a file that includes the types of data that you can use to validate query logic for the data set. You can use the **reportData.xml** sample XML file defined in the Data Source entity (and included in the Design Studio report design example) or browse for a different file.

7. Click **Next**.

8. Create data set row mapping by selecting elements in the XML structure and by clicking the arrow button.

Each instance of the selected XML element in the XML document is mapped to a data set row. Use the wizard instructions to define data set row mappings. For more information about the XPath syntax applicable to Design Studio, see "About XPath Expression Patterns for Row Mapping".

The installed Design Studio examples include data set row mapping examples. See "Working with the Design Studio Report Examples" for more information.

> **Note:**
>
> In the **Select or edit the XPath Expression** dialog box, select the **Custom XPath Expression** option and define the path for the XML element.

9. Create data set column mappings by selecting elements and attributes in the XML structure and by clicking the arrow button.

   Use the wizard instructions to define data set column mappings. For more information about the XPath syntax applicable to Design Studio, see "About XPath Expression Patterns for Column Mapping".

   The installed Design Studio examples include data set column mapping examples. See "Working with the Design Studio Report Examples" for more information.

10. (Optional) To test new row and column mappings against the sample XML file, click **Show Sample Data**.

    You can test mappings to ensure that the XPath expressions include no typing or syntax errors.

11. Click **Finish**.

    The Edit Data Set dialog box appears. You can define additional information about the data set, edit row mapping and column mapping, preview the results, and so forth.

12. Click **OK**.

Defining the Data to Add to Reports

The BIRT Report Designer includes tools for designing, debugging, and previewing report designs. This section includes a small subset of procedures that help you create and design reports for use in Design Studio. For additional documentation, tutorials, and examples, see the BIRT project website:

`http://www.eclipse.org/birt/`

Defining Computed Columns for Data Sets

You define computed columns to generate report data from expressions of the data in other columns in the data set (rather than retrieving the data directly from a data source).

To define computed columns for a data set:

1. From the Report Design perspective, in the Data Explorer view, right click a Data Set entity and select **Edit**.

   The Edit Data Set dialog box appears.

2. Select **Computed Columns**.

3. Click **New**.
4. In the **Column Name** field, enter a name for the computed field.
5. In the **Data Type** field, select the type of data to be returned by the computed field.
6. In the **Expression** field, enter an expression that calculates the value.

   You can click the **Fx** button to build the expression in the Expression Builder dialog box.

> **Note:**
>
> Computed column expressions can refer only to column data that you mapped in the data source. Use a naming convention to distinguish columns intended to appear in reports from columns intended for computing values. For example, you can pre-pend column names with an underscore (for example, **_columnX**) to identify columns to be used only for filtering and for supporting computed columns.

7. Click **OK**.

See *BIRT Report Developer Guide* for more information.

Defining Filtering Conditions for Data Sets

You can limit the data included in reports by defining filtering conditions. A filter condition is an expression with a value that resolves to true or false. Design Studio applies the filter to each row of the data set. If the expression resolves to true, the row is included in the report. If you define multiple filters, the expressions of all filters must resolve to true for a row to be included in the report. See "Working with Report Data Filters" for more information.

1. From the Report Design perspective, in the Data Explorer view, right click a data set and select **Edit**.

   The Edit Data Set dialog box appears.
2. Select **Filters**.
3. Click **New**.
4. Enter a filter condition.

> **Note:**
>
> For best performance, order the expressions from most likely to exclude content to least likely to exclude content.

5. (Optional) To test the mappings and computed columns, click **Preview Results**.

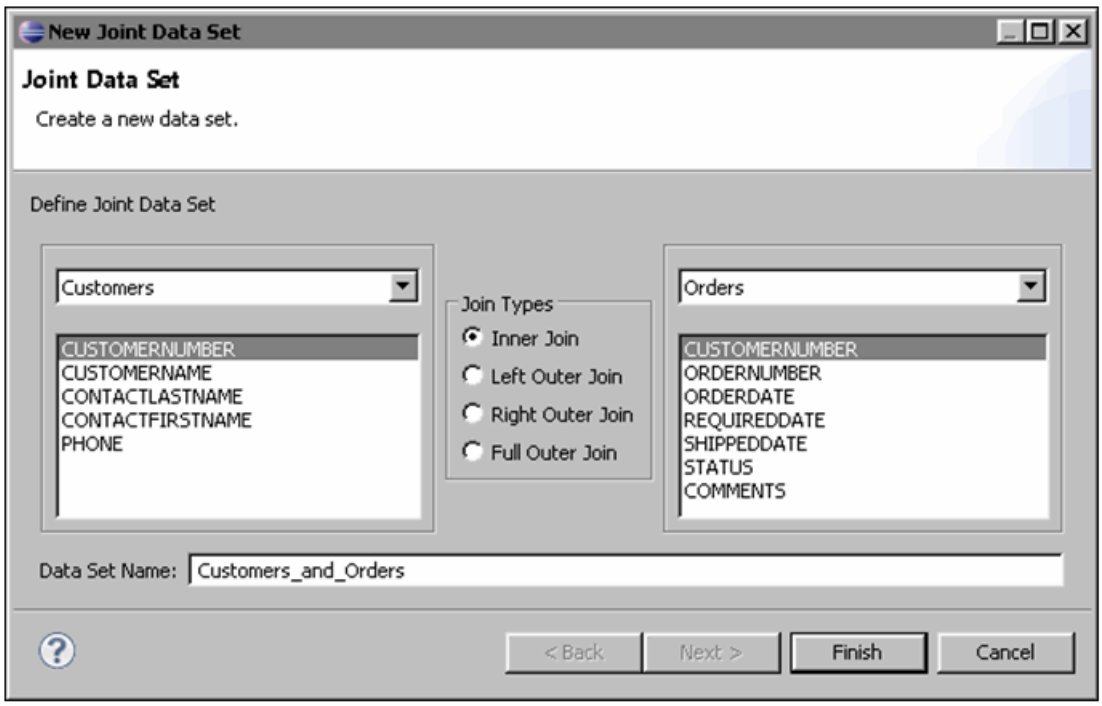
   Check filter expressions to ensure that the data is filtered correctly.
6. Click **OK**.

Merging Data Sets

You can combine data from two data sources into a single data set. For example, you can combine data from an XML file and from a text file (you must first create the XML data set and the text file data set).

When you merge, or join, two data sets, you create a joint data set. You can add computed columns and filters to a joint data set and preview the results. You can also merge joint data sets together to combine more than two data sets into a single joint data set. Figure 6-5 shows two data sets, **Customers** and **Orders**, merged into a single data set.

Figure 6-5 Merging Data Sets



If you want to join an aggregated column, add a computed column to the source data set. See "Defining Computed Columns for Data Sets" for more information.

To merge data sets:

1. In the Report Design perspective, open a report design in the Layout editor.
2. Click the **Data Explorer** tab.

   The Data Explorer view appears.
3. Right-click **Data Sets** and select **New Joint Data Set**.

   The Join Data Set wizard appears.
4. In the **Define Joint Data Set** area, select the data sets to join.

   The columns of the data sets appear.
5. Select one column from each data set to join the columns.

6. For each join, select one of the following join types:
   - **Inner Join**: Returns rows from both data sets when the column values match.
   - **Left Outer Join**: Returns all rows from the left data set and all matched rows from the right data set.
   - **Right Outer Join**: Returns all rows from the right data set and matched rows from the left data set.
   - **Full Outer Join**: Returns all rows from both data sets even when the column values do not match.
7. In the **Data Set Name** field, enter a name for the merged data set.
8. Click **Finish**.

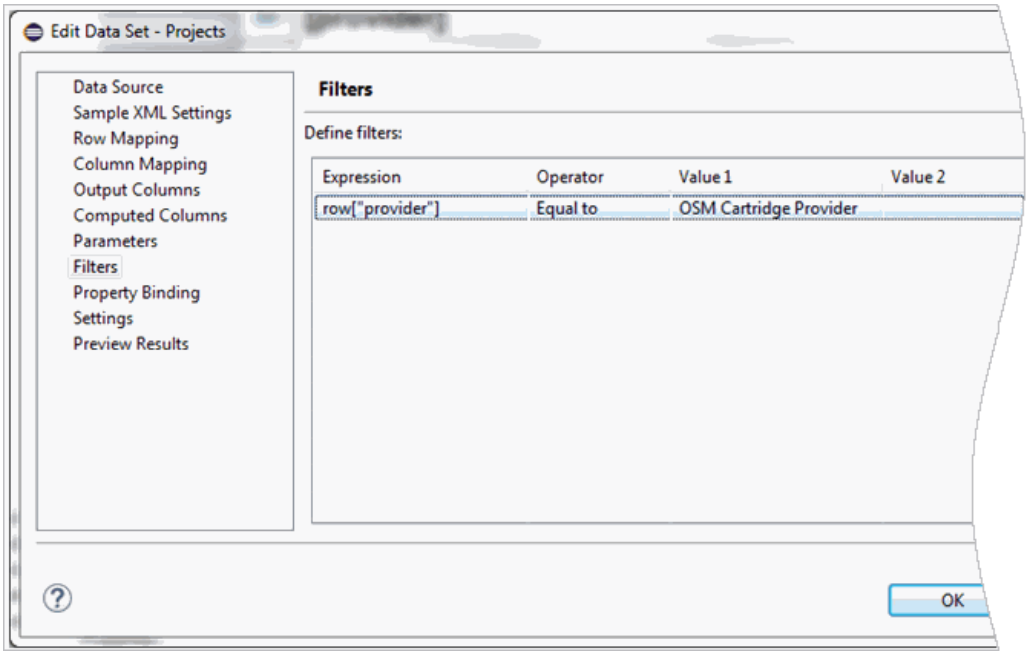
   The Edit Data Set dialog box appears.
9. (Optional) Review and edit the data set.

   For example, click **Output Columns** to review the full set of columns being joined and to customize the column names.
10. (Optional) To review the rows returned by the joint data set, click **Preview Results**.
11. Click **OK**.

Filtering Data Sets for Tables

You use data binding filters to filter a data set for a specific table. Filtering data sets for tables is useful when a data set is used in multiple contexts, each with different filtering criteria. The data binding filter is applied against each instance of the table, rather than once during data set creation. In Figure 6-6, the data set **provider** row is filtered for the value **OSM Cartridge Provider**:

Figure 6-6 Filtering Data Sets



To filter data sets for tables:

1. In the Report Design perspective, open a report design in the Layout editor and select a table in a report design file.

   The Property Editor view appears.

2. In Property Editor view, click the **Filters** tab.

   The Filter By page appears.

3. Click **Add**.

   The New Filter Condition dialog box appears.

4. Enter a filter condition.

   You can enter a condition directly into the first field or you can click the **Fx** button to open the Expression Builder and create a more complex expression. See "Working with XPath Expression Patterns" for more information.

5. Select an operator from the drop-down list.

6. Specify the value on which to search.

   Enter the value directly, select from the list of values, or use the Expression Builder to create a more complex value expression. For some operators, a value is not required.

7. Click **OK**.

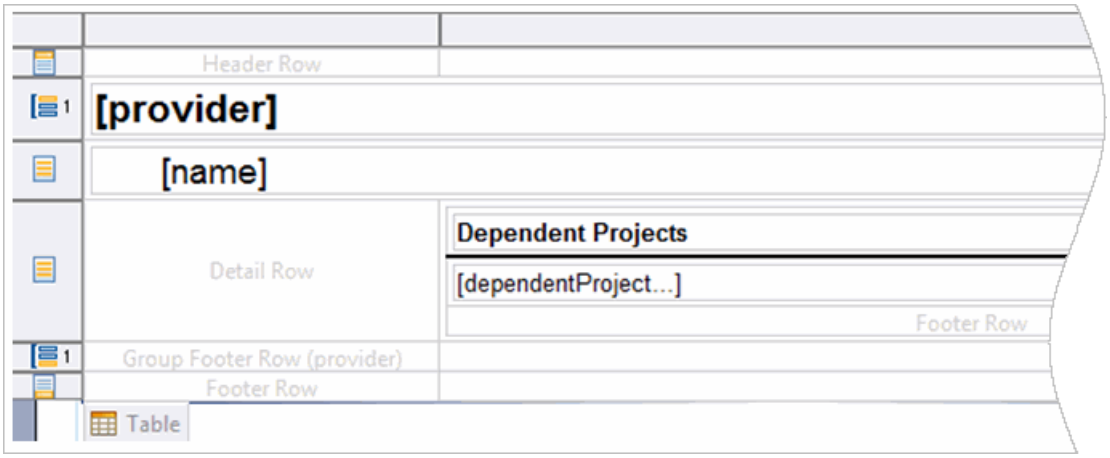
8. Click **Save**.

Nesting Tables

You can nest one table inside of another table. For example, a nested table can represent a filtered data set that is based on the current row of the parent table. The filter is applied to the child table for each row of the parent table.

Figure 6-7 shows one table nested in another in the Layout editor:

Figure 6-7 Nested Tables



> **Note:**
>
> Nested tables can impact performance. Use nested tables for small data sets. If the data sets are large enough to adversely impact performance, join the data sets instead of nesting the tables.

To nest tables:

1. In the Report Design perspective, open a report design in the Layout editor.
2. Create a parent table and bind the table to a data set.
3. Create a child table in a row of the parent table and bind the table to a second data set.

   The child table is repeated for each row of the parent table. To ensure that the child table displays content appropriate for the current row of the parent table, you must apply a data filter to the child table.

   To apply a data filter to the child table:

   a. Select the child table.

      The Property Editor view appears.

   b. Click the **Filters** tab.

   c. Click **Add**.

      The New Filter Condition dialog box appears.

   d. Select the child table column that you want to compare to the key value in the parent table.

   e. Select **Equal to** as the operator.

   f. Click the menu and select **Build expression**.

      The Expression Builder appears.

   g. From the **Category** list, select **Available Column Bindings**.

   h. From the **Sub-Category** list, select the child table.

      The set of columns available in the parent table appear.

   i. Double-click the parent table column to which you want to match the selected child table column.

   j. Click **OK**.

4. Click **OK**.

   The new filter is added to the child table.

Concatenating Rows into Comma-Separated Values

You can concatenate values in separate rows into a list of comma-separated values. The comma-separated values appear in the table footer. For example, you might want to display a list of comma-separated action codes in a report.

> **Note:**
>
> BIRT provides many functions that perform aggregate calculations, such as SUM, AVERAGE, and COUNT. See the BIRT project page documentation for more information.

To concatenate rows into a list of comma-separated values:

1. In the Report Design perspective, open a report design in the Layout editor.
2. Right-click in a table footer cell, select **Insert**, and then select **Aggregation**.

   The Aggregation Builder dialog box appears.
3. In the **Column Binding Name** field, enter a name for the concatenation.

   For example, describe the value that is returned by the column binding, such as **ActionCodes**. Column binding names must be unique in a report.
4. Select a data type.
5. In the **Function** field, select **Concatenate**.
6. In the **Expression** field, enter an expression that composes the value for each row of the table.

   For example, **row["toName"]**.
7. In the **Separator** field, enter the following:

   ", "
8. Click **OK**.

Defining Data Presentation in Reports

The BIRT Report Designer includes tools for designing, debugging, and previewing report designs. This section includes a small subset of procedures that help you create and design reports for use in Design Studio. For additional documentation, tutorials, and example, see the BIRT project website:

http://www.eclipse.org/birt/

> **Note:**
>
> You can also use Cascading Style Sheets to enable more complex report formatting. You can apply styles globally (to affect the entire report) or to individual report elements. Style sheets enable you to style once and use those styles multiple times. You can use a style library to improve consistency across a suite of report designs.
>
> For more information, see the *Using Styles and Cascading Styles* tutorials on the BIRT Project page.

Hiding Content Based on Output Format

Report elements can include conditions that control the visibility of the element in the report. Conditions can be used to:

- Hide content in specific output formats
- Display contextual values only when appropriate
- Expand or reduce report content based on a report parameter (for example, avoiding duplicate report designs when the report content varies only by level of detail)
- Toggle a set of layouts based on report data (for example, switch between a table format or an in-line list based on the row count)

To hide content based on a report output format:

1. From a report design file, right-click the report element to hide and select **Properties**.

   The Property Editor view opens.
2. Click the **Visibility** tab.
3. Select **Hide Element**.
4. Select the **For specific outputs** option.
5. Select the check box for each output format in which the content should not appear.
6. (Optional) In the **Expression** field, modify the conditional expression for an output format.

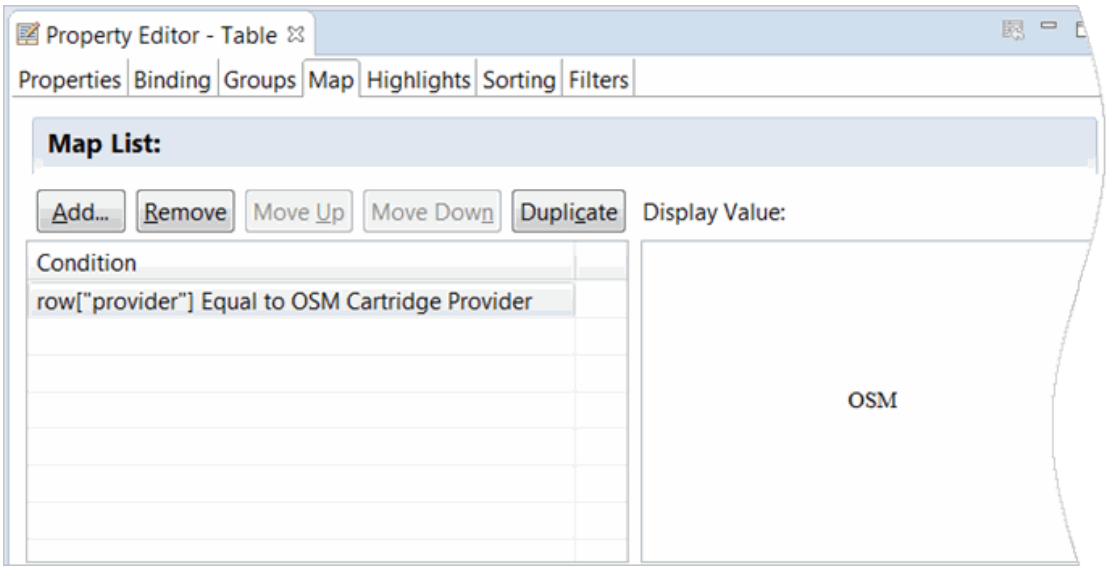
   The default value is **true**, which always hides the content for that specific output format.
7. Click **Save**.

Defining Value Mapping Rules

You can define rules that map values from a database column to values in a report column. For example, you might define rules that map the database status codes **S** and **F** to the display values **Success** and **Fail** respectively. When defining value mapping rules, you define a conditional expression and a display value that appears in a report when the expression evaluates to true.

For example, in Figure 6-8, the conditional expression states that if the data set **provider** row contains the value **OSM Cartridge Provider**, the value **OSM** appears in the report:

Figure 6-8 Value Mapping Rules



To define value mapping rules:

1. Open a report design in the Layout editor and select a data field.

   The Property Editor view appears.
2. In Property Editor, click the **Map** tab.

   The Map List page appears.
3. Click the **Add** button.
4. Define the condition that must evaluate to true.
5. Define the display value that appears in the report.
6. Click **OK**.

Defining Value Highlighting Rules

You can define rules that map values from a database column to specific formatting options when those values appear in a report. When defining highlighting mapping rules, you define a conditional expression and the formatting that is applied to the display value when the expression evaluates to true.

To define value highlighting rules:

1. Open a report design in the Layout editor and select a data field.

   The Property Editor view appears.
2. In Property Editor, click the **Highlights** tab.

   The Highlight page appears.
3. Click the **Add** button.
4. Define the condition that must evaluate to true.
5. Define the formatting that will be applied to the value when it appears in the report.

6. Click **OK**.

Adding Additional Report Design Elements

The BIRT Report Designer includes a tools for designing, debugging, and previewing report designs. This section includes a small subset of procedures that help you create and design reports for use in Design Studio.

See the *BIRT Report Developer Guide* for more information about the procedures in this section, and for the full BIRT Report Designer documentation set.

Adding the Current Date to a Report

BIRT includes functions for manipulating content values. For example, you can use the **BirtDateTime** class to manipulate date and time values. You can use the **now()** function to display the current date and time. And, you can use the **Formatter** class (which includes a single format function similar to Java **MessageFormat**) to format a date by providing a date and format string.

To add the current date to a report:

1. From a report design file, right-click and select **Insert**, and then select **Dynamic Text**.

   The Expression Builder dialog box appears.

2. In the **Expression** field, enter the following:

   Formatter.format(BirtDateTime.now(), "*date_format***")**

   For example:

   Formatter.format(BirtDateTime.now(), "MMMM dd, yy h:mm a")

3. Click **OK**.

Adding Page Numbers

You can add page numbers to a primary page and to a report layout.

> **Note:**
>
> You can suppress page numbering for specific output types, such as for non-paginated HTML output. See "Hiding Content Based on Output Format" for more information.

To add page numbers to a report design:

1. From a report design file, right-click and select **Insert**, and then select **Text**.

   The Edit Text Item dialog box appears.

2. Change the formatting type from **Auto** to **HTML**.

3. Change the content type to **Dynamic Text**.

   This enables you to embed an expression in the text.

4. Do one of the following:
   - To add page numbering to a primary page, paste the following value into the open text area:

     <value-of>pageNumber</value-of> / <value-of>totalPage</value-of>
   - To add page number to a layout, paste the following value into the open text area:

     Page <viewtime-value-of> pageNumber </viewtime-value-of> of <value-of> totalPage </value-of>
5. Click **OK**.

Dynamically Selecting Images

You can create XPath expression conditions that determine which image to use in a report. The BIRT Report Developer supports many URI schemes that enable you to retrieve content. For example, you might use the **http** scheme to interact with web resources or XML namespaces. BIRT also enables you to embed images into a report, or to dynamically select images from a data set.

You can use the Eclipse URI **platform** scheme to retrieve content from an installed Eclipse plug-in. See the Eclipse documentation for more information.

> **Note:**
>
> If a report resource does not resolve, the report may include an error message that states that the resource of the report item is not reachable.

To dynamically select images:

1. From a report design file, right-click and select **Insert**, and then select **Image**.

   The Edit Text Item dialog box appears.
2. In the **Select Image from** field, select the **URI** option.
3. In the **Enter URI** field, click the down-drop menu and select **JavaScript Syntax**.

   The Expression Builder dialog box appears.
4. Enter a conditional expression that represents the image options.

   The following is a simple example that shows a selection between two images based on the **typeId** value:

```
if (row["typeId" ] == "type1") {
        "platform:/plugin/my.report.design.plugin.id/icons/type1.png" ;
} else {
        "platform:/plugin/my.report.design.plugin.id/icons/type2.png" ;
}
```

5. Click **OK**.

Creating Internal Links Between Report Items

You can create links between two related report items in a report design. Most report readers support standard linking functionality. Design Studio does not validate links during report creation.

You create internal links in a report using bookmarks. A bookmark is an identifier used as an anchor point for hyperlink navigation in a report. If a report contains multiple bookmark identifiers, each bookmark identifier must be unique.

> **Note:**
>
> You can also define links using:
>
> - URIs, to reference standard URI protocols. For example, you can create links using HTTP or FTP URI schemes.
> - Drill-through parameters, to enable interactive report features most commonly used with HTML output and a web server that generates reports on-demand.
>
> If you define links with drill-through parameters, you link to a different report using the following report parameter syntax in the link expression:
>
> **params["data_source_model_file"].value**
>
> See the BIRT project page for information about creating links using URI schemes or drill-through parameters.

To create an internal link using a bookmark:

1. Select a data, label, or image element in a report design file.

   The Property Editor view appears.
2. In Property Editor, click the **Properties** tab and then click the **Hyperlink** option.

   The Hyperlink page appears.
3. Click **Edit**.

   The Hyperlink Options dialog box appears.
4. Select **Internal Bookmark**.
5. Do one of the following:
   - From the **Bookmark** list, select a bookmark identifier.
   - In the **Linked Expression** field, click the **Fx** button and define an expression that resolves to a value that matches a bookmark identifier defined in the report.
6. Click **Save**.

Creating Table of Contents Entries

You add table of contents (TOC) entries in various places in a report design to identify the types of information to be included in a TOC. The TOC is not directly represented

in the report, and may not appear in some output formats. In some output format viewers the TOC appears in a separate panel of the reader and not in the report directly.

When designing reports, you pair TOC entries with bookmarks, and define a TOC expression to retrieve the text to be included in the generated TOC. You can also define TOC expressions for table groups and on specific rows in a table.

To create TOC entries:

1. In a report design file, select a data, label, or image element that you want to appear in the TOC.

   The Property Editor view appears.
2. In the Property Editor view, click the **Properties** tab and then click the **Table of Contents** option.

   The Table of Contents page appears.
3. In the **Table of Contents** field, enter a TOC entry for the report element.

   You can define the entry directly in the field or by using the Expression Builder.
4. Click **Save**.

Defining Text as HTML

Data sets can include simple text or text annotated with a markup language. You can use a markup language to embed simple formatting data definition. For example, you can format text with paragraph breaks, bullets, or emphasis.

To format text in a report, you must set up the text column to enable BIRT to interpret the string value in a data set as HTML. This set up enables you to embed HTML tags for report formatting.

> **Note:**
>
> Invalid markup can cause exceptions during report generation.

To define text as HTML:

1. From a report design file, right-click and select **Insert**, and then select **Text**.

   The Edit Text Item dialog box appears.
2. Select **HTML** as the formatting type.
3. Select **Dynamic Text** as the content type.
4. Enter the following text content:

   <VALUE-OF format="HTML">row["*myColumn***"]</VALUE-OF>**

   where *myColumn* is the name of the column that contains the formatted text.
5. Click **OK**.

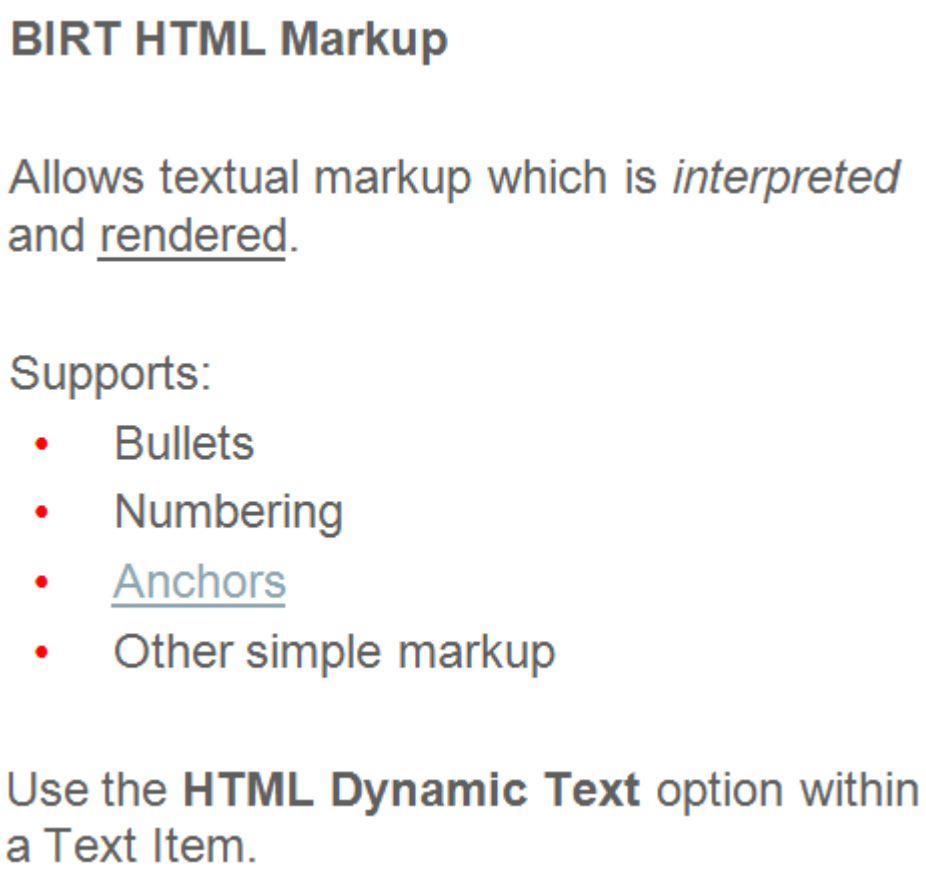
Example 6-1 includes text annotated with a markup language:

Example 6-1 HTML Markup Example

```
<p><b>BIRT HTML Markup</b></p>
<p>Allows textual markup which is <i>interpreted</i> and <u>rendered</u>.</p>
<p>Supports:
<ul>
  <li>Bullets</li>
  <li>Numbering</li>
  <li><a href="http://www.oracle.com/">Anchors</a></li>
  <li>Other simple markup</li>
</ul>
</p>
<p>Use the <b>HTML Dynamic Text</b> option within a Text Item.</p>
```

Figure 6-9 illustrates how the text in Example 6-1 appears in a report:

Figure 6-9 HTML Markup in a Report



Working with XPath Expression Patterns

Design Studio reports use data sets that are structured as table rows and columns. When defining data sets, you map a top-level XML element as a data set row, and other XML elements or attributes as data set columns. You use XPath expressions to define the paths to elements and attributes.

This section describe common XPath expression modeling patterns that you can use to create expressional conditions.

About XPath Expression Patterns for Row Mapping

To improve processing performance, reduce the size of an initial data set by defining specific row mapping XPath expressions. For example, filter the row mapping to include only the information required for a report, and consider using multiple data sets instead of using a single data set when applying a filter in a data binding.

You can apply an attribute filter to only one XML element. You cannot, for example, apply an entity type filter and an element type filter in an XPath expression. You can apply multiple data set filters to any data set to include additional filtering.

If you map to a specific subtype of an abstract XML element, the Data Set wizard generates a warning message that states that the table mapping XPath does not exist. The XPath expression may be valid, but BIRT is not able to reconcile the use of a concrete subtype against an abstract (or otherwise extensible) XML element. If you cannot verify an XPath expression, use a test query on sample data and ensure that the query returns the expected data.

When using XPath predicates, you cannot use **excludes** operations. See the BIRT Project page for information about using data set filters to apply **excludes** logic.

Table 6-1 provides a list of common XPath expressions for row mapping:

Table 6-1 Row Mapping XPath Expression Patterns

To Do This	Use this Syntax
Map to a Design Studio root element	**/***
Include the entity element in the path name	***/*/*entity*
Select all instances of an element	***/*/*entity*//*element*
Use a predicate to select an element based on its **typeId** attribute	***/*/*entity*//*element*[@**typeId='***abc***']**
Select all instances of a first-level child element	***/*/*entity*/***/*element*
Select all instances of a second-level child element	***/*/*entity*/***/*element*/***/*element*
Use a predicate to select a first-level child element based on its **typeId** attribute	***/*/*entity*/*/*element*[@**typeId='***abc***']**
Select a container of elements at any level	***/*/*entity*//*container*/*element*
Use a predicate to select a container of elements at any level based on its **typeId** attribute	***/*/*entity*[@**typeId='***abc***']**//*container*/*element*
Select an element under a parent container	***/*/*entity*/*container*/*element*
Use a predicate to select an element in a parent container based on its **typeId** attribute	***/*/*entity*[@**typeId='***abc***']**/*container*/*element* or ***/*/*entity*/*container*/*element*[@**typeId='***abc***']**
Select all instances of entity relations	***/*/*entity*//*relation*
Select all entity project relations	***/*/*entity*/*project*/*relation*

Table 6-1 (Cont.) Row Mapping XPath Expression Patterns

To Do This	Use this Syntax
Select all element relations	*/*/*entity*/*/*element*/*/*relation*
Select specific element relations	*/*/*entity*/*/*element*/*/*relationListName*/*/*relation*

About XPath Expression Patterns for Column Mapping

XPath expressions for column mapping identify the column value in a data set. The column mapping expressions are relative to the row mapping selection.

When creating column mappings, the target child or attribute value is not required for every subtype of the selected element. For example, you can create a query that applies to a mixed set of elements to which you want to map the aggregated set of columns available across all element types. Reducing the number of distinct queries required for a report improves performance.

Many of the domain-specific fields in the Design Studio model are contained in subtypes of the base entity and element model. Reference the XML schema (included in the **Design Studio Report Design Example**) when creating data set column mappings to fields of a subtype. See "About the Design Studio Report Design Example" for more information about Design Studio XML schema.

Row selection XPath expressions select abstract entities and elements. Therefore, the attributes that are available for column mapping should be concrete attributes and children of those elements (BIRT can recognize the children and attributes defined directly on the base model). The column mapper does not display the concrete subtypes of the entity and elements as options for mapping. If you define a row mapping to select an abstract entity or element, the column mapper recommends column maps that select the base attributes and that select the children in the column mapping. When mapping to a value in a subtype, BIRT returns a warning that states that the table mapping XPath does not exist. Ignore this warning if you have verified that the mapped path does reflect a valid selection from a subtype of the row mapping element.

To map concrete subtypes, you add column mappings directly by specifying the relative path, or you can set the row mapping to select the specific entity or element that your final row mapping generates. Setting the row mapping to select the specific entity or element enables column mapping from the XML structure on the Column Mapping page of the Edit Data Set dialog box.

Test data set mappings with sample XML to ensure that all mappings are correct. You can create sample XML files by using the Design Studio Generate Report wizard to generate the **Design Studio Model in XML** report.

Table 6-2 displays common column mapping XPath expression patterns:

Table 6-2 Column Mapping XPath Expression Patterns

Name	XPath	Description
id	/@id	Get the global ID of the selected element.

Table 6-2 (Cont.) Column Mapping XPath Expression Patterns

Name	XPath	Description
type	/@type	Get the entity or element type of the selected element.
parentId	../../@id	Get the global ID of the selected element's parent. This expression resolves to a null value for entities because entities have no parent.
field1	*/field1*	Get the *field1* value, where *field1* is a child of the selected element. This expression resolves to a null value if there is no child element.
attribute1	*/attribute1*	Get the *attribute1* value, where *attribute1* is an attribute of the selected element. This expression resolves to a null value if there is no child element.
localizedString	*localizedString*[@lang='default']	Get the default value of a *localizedString*. This expression resolves to a null value if the *localizedString* does not have a default value.

Table 6-3 displays column mappings that are typical for data sets with row mappings that map to relations. These data sets are often used by a nested table with a filter to match global identifiers with the outer row's entity or element identifier.

Table 6-3 Column Mappings for Data Sets with Row Mappings to Relations

Name	Path	Description
name	/@name	Name of the relation.
type	/@type	Type of the relation.
fromId	../../@id	Global identifier of the relation source element.
fromName	../../@name	Name of the source element.
fromTypeName	../../@type	Name of the source element type.
toId	/@ref	Reference representing the target global identifier.
toName	/target/@name	Name of the target element.
toTypeName	/target/@typeName	Name of the target element type.

About XPath Expression Parameters

You can add parameters to data set row mapping and to data set column mapping expressions. For example, you can add a report parameter to the predicate of a row-selection expression to filter based on the parameter's value.

To add a parameter to an XPath expression, use the following syntax:

{?*parameterName*?}

where *parameterName* is the name of the parameter whose value you want to use.

For example, you can write an XPath expression to retrieve all library books with a category defined as **mystery**:

library/book/[@category="{?mystery?}"]

Consider that you would now need to define an input parameter that stores the category value that a user enters during a search. The following XPath expression searches for books defined with a category value that is stored in the **bookCategory** parameter:

library/book/[@category="{?bookCategory?}"]

You can link data set parameters to a report parameter that is defined as a constant, or you can populate a parameter using an expression. Parameters created with expressions can include conditions and functions (such as date manipulation). However, parameters cannot use columns from the data set.

Working with Report Data Filters

The manner in which you filter report data can impact reporting performance. Some filters are applied multiple times during report generation. For example, a data binding filter is applied multiple times if the binding includes a reference to an outer binding, such as when tables are nested.

Rather than creating nested, normalized data sets, consider creating data sets that are non-normalized, where you join nested data sets. Consider this approach when the nested filtered content is a small subset of the bound data set and the outer data set contains multiple rows.

Data retrieved from a data source can be filtered at a number of points during report generation. You can apply filters to:

- Data set row mappings
- Data sets
- Data bindings
- Groups
- A chart series

Testing Report Designs

Testing report designs directly in Design Studio shortens the design-test cycles by eliminating the need to package the custom report design.

You can test report designs using the following methods:

- Testing Custom Report Designs Using the Report Designer
- Testing Custom Report Designs Using the Generate Report Wizard

Testing Custom Report Designs Using the Report Designer

You use a sample XML document to test report designs using the Report Designer. You can create a sample XML document by generating the XML Model report, which produces the Design Studio XML format used by the Report Designer. To test custom report designs using the Report Designer:

1. From the Design Studio **Window** menu, select **Open Perspective**, then select **Other**, and then select **Report Design**.
2. Click **OK**.

   The Report Design perspective appears.
3. Click the **Resource Explorer** tab.
4. Double-click a **.rptdesign** file.

   The report design opens in the Report Designer editor.
5. From the **Run** menu, select **View Report** and then select the output format.

   The default Internet browser opens and displays the Parameters dialog box.
6. Enter the fully qualified path to a sample XML file.

   The sample XML file should include data that is representative of the type of data that you intend to include in finished reports. You can:

   - Enter the path to the **reportData.xml** file that is included in the **Design Studio Report Design Example**. See "Adding the Report Design Example to the Workspace" for more information.
   - Run the **Design Studio Model in XML** report to create a new sample XML file. See "Working with the Design Studio Report Examples" for more information.

Testing Custom Report Designs Using the Generate Report Wizard

To test custom report designs using the Generate Report wizard:

1. In Design Studio, from the **Studio** menu, select **Generate Report**.

   The Generate Report wizard appears.
2. Select the **Select a report design from a file** option.
3. Click **Browse** and locate and select the report design that you want to test.
4. Click **Next** and navigate through the wizard to ensure that the report design works as intended.

   For example, select to generate the report content by project or by entity, based on the data that you want to appear in the test report.
5. Review the generated report, the Eclipse Console view, and the error log.

   Use this information to debug reports that are not generated as expected.

Working with the Design Studio Report Examples

Design Studio includes the following example report projects and example reports that support the reporting features.

- The **Design Studio Report Design Example** includes an update site project, a feature project, and a project that contains sample report designs. See "About the Design Studio Report Design Example" for more information.
- The **Design Studio Report Processor Example** is an advanced example that demonstrates how to extend the processing logic of the Design Studio report generation framework. See "About the Design Studio Report Processor Example" for more information.
- The **Design Studio Model in XML** report enables you to create a sample XML file that you can use in the BIRT Report Designer when testing your report design. For example, you can run this report and direct the reporting engine to use the static data defined in the generated XML file to test your column mapping and row mapping against the sample XML file.

  When generating this report, use an existing project or generate a new project that includes enough content to adequately test your report design. See "About the Design Studio Report Design Example" for information about the sample XML delivered with Design Studio.
- The **Entity and Element Type Reference** report enables you to view the IDs assigned to all Design Studio model entity and element types. You can reference the IDs in this report when you need to write XPath expressions that filter for a specific value, such as when filtering and when row mapping during report design.

  For example, when writing a query, you may need to select for Product entities only. You can run the Entity and Element Type Reference report to obtain the ID for the Product entity type. See "Working with XPath Expression Patterns" and "Working with Report Data Filters" for more information.

For more information about adding examples to the workspace, see "Adding the Report Design Example to the Workspace". For more information about generating reports, see the Design Studio Help.

About the Design Studio Report Design Example

The **Design Studio Report Design Example** is included in the Design Studio installation. The **Design Studio Report Design Example** includes projects that you can use as a reference or as a starting point for creating your own custom report designs.

This example includes the following projects:

- **design.studio.example.report.design.update.site**, which contains an example of a standard Eclipse update site project.
- **design.studio.example.report.design.feature**, which contains an example of a standard Eclipse feature project.
- **design.studio.example.report.designs**, which contains an Eclipse plug-in project with report designs and content useful for designing reports.

This example includes the following:

- Sample report designs
  - A sample that demonstrates a simple report design. This sample report design includes the Design Studio data source setup and the required **data_source_model_file** report parameter. The layout includes a basic report header and a sample heading.

  - A sample that demonstrates a complete report design. This sample report design includes a list of Design Studio entities and their relationships. The sample demonstrates how to define a data set to retrieve data from a data source, and demonstrates common design patterns such as nested tables, grouping, hyperlinks, bookmarks, and styling options.
- The **.rptdesign** files for all of the Design Studio reference reports
- XML schemas that describe the Design Studio Exchange Format

  See "Working with the Design Studio Exchange Format" for more information.
- A sample XML file

  You can use this file to test your report designs.
- A Design Studio report library

  This library stores reporting objects, and enables you to reuse existing report objects, such as tables, data sources, and styles. For example, you can copy into a report design the data source and the data source report parameter that is contained in the library. You can use the Resource Explorer view to review the library and objects that exist in your workspace.
- Extension point declarations for sample report designs

Troubleshooting Report Designs

Use the following tips and guidelines to help troubleshoot common issues.

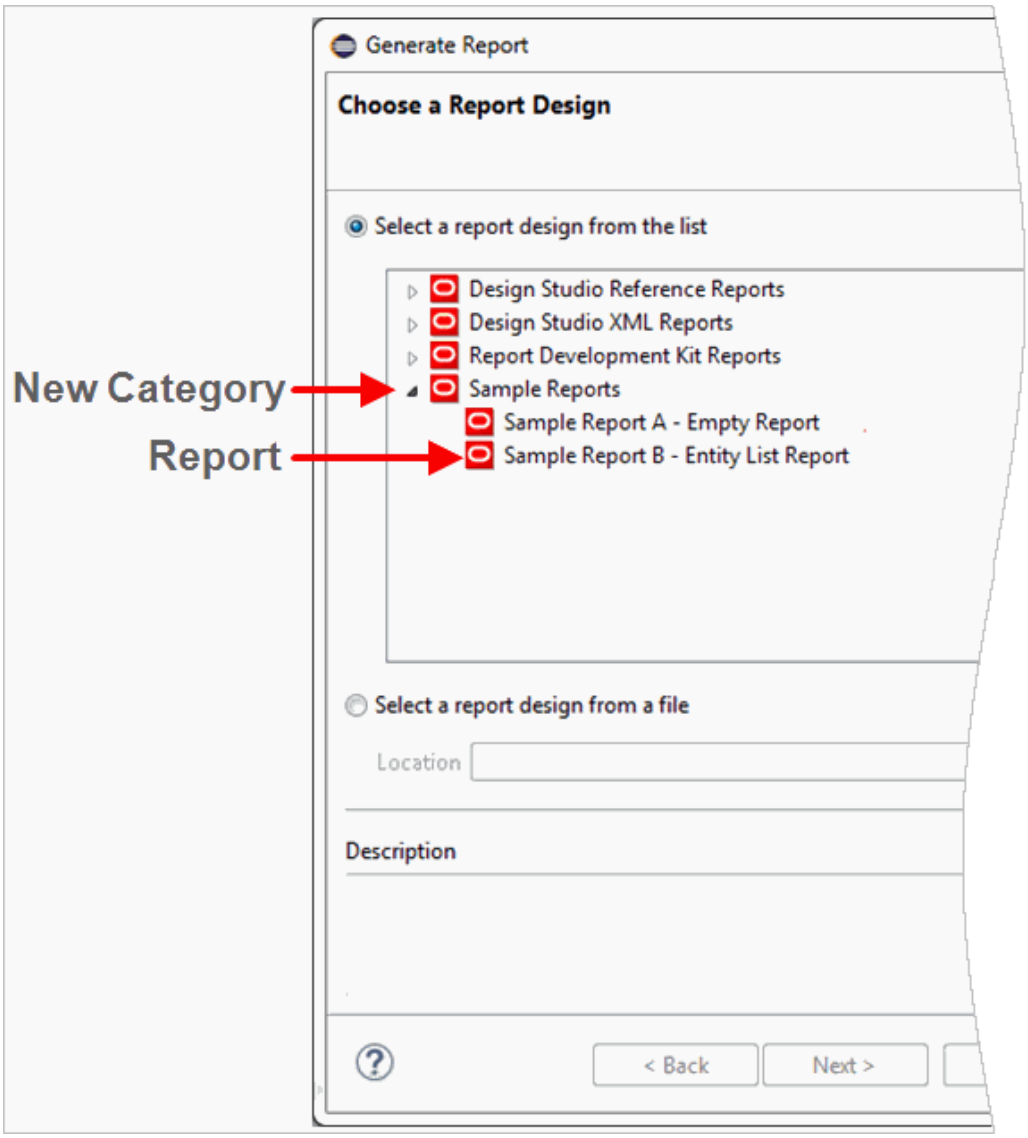
- Some report design aspects may be ignored in certain output formats. Ensure that a report design is optimized for the output format. See "About Report Designs" for more information.
- Alignment issues are typically caused by the inconsistent use of padding and margins in the report structure. Nested grids contribute to the padding and margins of a container and can cause misalignment. Use adjustment strategies consistently to avoid additive padding and margins.

Adding Reports and Report Categories to the Generate Report Wizard

If you intend for Design Studio users to install your custom reports from an update site, you must add your custom reports to the Design Studio Generate Report wizard. Adding the new custom reports to the Generate Report wizard ensures that your reports are available for selection on the first page of the wizard.

You can create categories in the Generate Report wizard and organize your reports under these categories. Users can expand a category folder on the first page of the wizard to access custom reports. Figure 6-10 shows a new report category called **Sample Reports**:

Figure 6-10 New Report Categories



> **Note:**
>
> If the report design will always be selected from a file (for example, by selecting the **Select a report design from a file** option in the Generate Report wizard), you are not required to add your reports to the wizard.
>
> The **plugin.xml** project file that contains the **sampleA.rptdesign** and **sampleB.rptdesign** files, located at the root directory of the **design.studio.example.report.designs** project, includes an example of the configuration required to add your custom reports to the Design Studio Generate Report wizard.

To add your custom reports to the Design Studio Generate Report wizard:

1. From the Design Studio Package Explorer view, expand your plug-in project folder.
2. Double-click the **plugin.xml** file.

   The file opens in the Plug-in Manifest editor.
3. Click the **Extensions** tab.
4. Click **Add**.

   The Extension Point Selection dialog box appears.
5. On the **Extensions Point** tab, select **oracle.communications.studio.report.type** from the list.

   If the Report Type extension does not appear in the list, deselect the **Show only extension points from the required plug-ins** option.
6. Click **Finish**.

   If prompted to add the **oracle.communications.studio.report.core** plug-in to the list of plug-in dependencies, click **Yes**.

   The Report Type extension point is added to the plug-in project.
7. In the **All Extensions** area, right-click the extension and select **New**, and then select **category**.

   A new category appears under the extension.
8. Select the category.
9. In the **Extension Element Details** area, enter a unique ID for the category.
10. In the **Name** field, enter the category name.

    The name that you define here is the label that appears for the category in the Generate Reports wizard.
11. (Optional) To associate the category with an icon in the Generate Report wizard, click the **icon** field **Browse** button and select an icon.
12. (Optional) In the **All Extensions** area, expand the category folder and then select **description**.

    In the **Extension Element Details** area, enter a description for the category.
13. In the **All Extensions** area, right-click the extension and select **New**, and then select **report-type**.

    A new report type appears under the extension.
14. Select the report type.
15. In the **Extension Element Details** area, enter a unique ID for the category.
16. In the **Name** field, enter the category name.

    The name that you define here is the label that appears for the report type in the Generate Reports wizard.
17. (Optional) To associate the report type with an icon in the Generate Report wizard, click the **icon** field **Browse** button and select an icon.
18. In the **reportDesign** field, enter the relative path of the report design file or click **Browse** to select the file from a list.
19. (Optional) In the **generatorTypeId** field, enter the relative path to your custom report generator.

Leave this field blank if you don't have a custom report generator. Design Studio users the default BIRT report generator if no other report generator is specified.

20. (Optional) In the **categoryId** field, associate the report type with an existing category.
21. (Optional) In the **All Extensions** area, expand the report type folder and then select description.

    In the **Extension Element Details** area, enter a description for the category.
22. (Optional) To view additional information about the extension point, click the **Show extension point description** link.
23. Click the **Build** tab.

    The Build page contains information required to build, package, and export the plug-in project.
24. In the **Binary Build** area, select the report design files that you want to include in the plug-in packaging, or select the folders that contain the report designs.

    For example, if you organized all of your report design files under a reports folder, select this folder in the **Binary Build** area.
25. Click **Save**.

For more information about the Plug-in Manifest editor and extensions, see the Eclipse *Plug-in Development Environment Guide*.

Extending Design Studio Reporting

You can extend the Design Studio reporting engine by creating report processors. You can create processors that:

- Perform actions before the report is generated, such as:
  - Create images to be used in the report, such as a diagram
  - Generate additional data source content
  - Send or log notifications
- Perform actions after the report is generated, such as:
  - Email a report
  - Post a report to a website
  - Clean-up report files

> **Note:**
>
> - Use Ant tasks for post-generation tasks whenever possible. See the *Design Studio System Administrator's Guide* for more information.
> - Report processors run during command-line report generation and do not support user input. Ensure that your report processors include no references to Eclipse UI plug-ins.

About the Design Studio Report Processor Example

The **Design Studio Report Processor Example** is an advanced example project that demonstrates how to extend the processing logic of the Design Studio report generation framework. You can extend the reporting functionality, for example, to dynamically produce graphs, charts, or images. You can use the example project as a reference or as a starting point for your own project.

The **Design Studio Report Processor Example** includes the basic components needed to produce a custom report processor. The example demonstrates how to define and register a custom report processor. The report processor is paired with a sample report that demonstrates how information provided by the custom report processor appears in a report.

The **Design Studio Report Processor Example** contains:

- Java code for implementing the sample report processor
- **AbstractProcessor.java**, which includes boilerplate code for implementing the required IReportProcessor interface
- **SampleProcessor.java**, which is the class that provides the logic for the sample report processor
- **SampleProcessorLogger.java**, which provides functions for logging to the Eclipse error log
- **SampleProcessorPlugin.java**, which is an implementation of a BundleActivator interface required for Eclipse plug-ins, and which includes useful utility methods
- Extension point declarations for the sample report processor

> **Note:**
>
> The BIRT framework is also extensible. You can extend the BIRT controls, BIRT functions library, and supported data source technologies. See the BIRT project web page for information about extending BIRT.

Extending Reporting Tasks by Adding Report Processors

You can extend reporting tasks by creating an implementation of the following IReportProcessor interface: **oracle.communications.studio.report.core.generator.processor.IReportProcessor**.

This processor extends the processing logic of the Design Studio report generation framework. For example, you can generate graphs, incorporate document signing, or add other features not directly supported by the report generation framework.

> **Note:**
>
> This task is intended for advanced users who are familiar with Eclipse plug-in development and Java coding. The **Design Studio Report Processor Example** project includes a simple implementation. See "About the Design Studio Report Processor Example" for more information.

To extend reporting tasks by adding report processors:

1. Create a plug-in project.

   For example, you can use a report design plug-in project. See "Creating Plug-in Projects" for more information. The plug-in project **plugin.xml** file is added to the Package Explorer view.

2. Double-click the plug-in project **plugin.xml** file.

   The file opens in the Plug-in Manifest editor.

3. Click the **Extensions** tab.

4. Click **Add**.

   The New Extension dialog box appears.

5. Locate and select **oracle.communications.studio.report.generator**.

   If you don't see the **oracle.communications.studio.report.generator** value, deselect the **Show only extension points from the required plug-ins** option. If prompted to add dependencies, select **Yes**.

6. Click **Finish**.

   The extension appears in the **Extension** area.

7. Select the extension.

8. In the **Extension Details** area, click **Show extension point description**.

   The Design Studio Report Generator documentation page appears. Use this documentation to finish creating an implementation of the IReportProcessor interface.

7

Working with Design Studio Model Java API

This chapter describes the Oracle Communications Service Catalog and Design - Design Studio Model Java API, its utility class and methods, and the plug-in dependencies that are required in your Eclipse installation when using the API to extend Design Studio.

About the Design Studio Model Java API

The Design Studio Model Java API wraps the published Exchange Format XML files produced by Design Studio. When combined with other public Eclipse APIs, the Design Studio Model Java API enables you to extend Design Studio with custom functionality and features. For example, when adding a new action to a Design Studio menu, you can use the Design Studio Model Java API to read information required from the published Exchange Format.

The **Design Studio Action Command Examples** project includes examples of how to use the Design Studio Model Java API and its utility classes and methods. See "About the design.studio.example.action.command Example Project" for more information. See "Working with the Design Studio Exchange Format" for more information about the Exchange Format.

Table 7-1 lists the top-level packages that you can browse to review Design Studio model information. These packages are located in the **design.studio.example.action.command/ Plug-in Dependencies** folder.

Table 7-1 Top-Level Design Studio Packages

Package	Description
oracle.communications.studio.model	Includes Design Studio platform information.
oracle.communications.studio.model.activation	Includes the Design Studio for ASAP product model.
oracle.communications.studio.model.common	Includes conceptual model information.
oracle.communications.studio.model.data	Includes information about the Data Dictionary.
oracle.communications.studio.model.integration	Includes information about the Activation task and about elements and entities that integrate Design Studio for OSM with Design Studio for ASAP.
oracle.communications.studio.model.integrity	Includes the Design Studio for Integrity product model.
oracle.communications.studio.model.inventory	Includes the Design Studio for Inventory product model.
oracle.communications.studio.model.osm	Includes the Design Studio for OSM product model.

About Design Studio Model Java API Utility Classes and Methods

Design Studio provides a utility class and methods that enable you to work with the information published by the Exchange Format.

The `ModelLocator` class is a helper class that you use to obtain the Design Studio resource and relation model information from the Exchange Format. You require the `ModelLocator` class and the following fully qualified class name to extend Design Studio using the Design Studio Model Java API:

oracle.communications.studio.model.modellocator.ModelLocator

The following methods are required when extending Design Studio using the `ModelLocator` class:

- `loadModel`
- `getRelationTarget`

About the loadModel Method

Consider the following when using the `loadModel` method:

- The `loadModel` method has the following signature:

  ```
  public Model loadModel(final IResource resource)
  ```

- You use the `loadModel` method to load a model referenced by the passed `IResource`.
- The `loadModel` method has the following input:

  ```
  org.eclipse.core.resources.IResource
  ```

- The `loadModel` method has the following output:

  ```
  oracle.communications.studio.model.Model
  ```

About the getRelationTarget Method

Consider the following when using the `getRelationTarget` method:

- The `getRelationTarget` method has the following signature:

  ```
  public Element getRelationTarget(final Relation relation)
  ```

- You use the `getRelationTarget` method to obtain the model from an element or entity relation.
- The `getRelationTarget` method has the following input:

  ```
  oracle.communications.studio.model.Relation
  ```

- The `getRelationTarget` method has the following output:

  ```
  oracle.communications.studio.model.Element
  ```

About the getReferencedBy Method

You can use the `getReferencedBy` method to find a set of entities that are associated with another entity. For example, you can find all technical actions associated with a specific resource facing service.

Consider the following when using the `getReferencedBy` method:

- The `getReferencedBy` method has the following signature:

  ```
  public List<Entity> getReferencedBy(final Entity entity)
  ```

- You use the `getReferencedBy` method to obtain the all the first-level referenced by and realized entities of given entity.
- The `getReferencedBy` method has the following input:

  ```
  oracle.communications.studio.model.Entity
  ```

- The `getReferencedBy` method has the following output:

  ```
  List<oracle.communications.studio.model.Entity>
  ```

You can use two additional APIs to filter the `getReferenceBy` method:

- The first API enables you to return a list of entities based on the entity file extension:

  ```
  List<Entity> getReferenceBy(Entity entity, String fileExtension)
  ```

  For example, if you call:

  ```
  getReferenceBy(myEntity, "cmnAction")
  ```

  the API returns only the action entities that reference `myEntity`.
- The second API enables you to return a list of entities based on a list of file extensions:

  ```
  List<Entity> getReferenceBy(Entity entity, List<String> fileExtensions)
  ```

About Design Studio Model Java API Package Dependencies

Table 7-2 displays the plug-ins that are required in your Eclipse installation when using the Design Studio Model Java API to extend Design Studio.

> **Note:**
>
> See the Eclipse *Plug-in Development Environment Guide* for information about defining plug-in project dependencies in the Plug-in Manifest editor.

Dependencies that you configure for a plug-in project are saved in the **MANIFEST.MF** file. You can configure these dependencies in the Plug-in Manifest editor using the fields on the **Dependencies** tab or by editing the file directly on the **MANIFEST.MF** tab.

Open the **Design Studio Action Command Example** project to review examples that illustrate how dependencies are configured in a plug-in project. See "About the design.studio.example.action.command Example Project" for more information.

Table 7-2 Java API Plug-in Dependencies

Plug-in	Description
org.eclipse.ui	Required for command and menu extension points.
oracle.communications.studio.model	Contains the Design Studio Model Java API.
org.eclipse.core.expressions	Required if the `PropertyTester` class is used.
org.eclipse.jface	Required for handling a selection in a view.
org.eclipse.core.resources	Required for handling resource files and folders.
org.eclipse.ui.workbench	Required for handling the command process.
org.eclipse.core.runtime	Required for handling the selection in a view.
org.eclipse.emf.common	Required for handling common EMF model constructs.
org.eclipse.emf.ecore	Required for handling the EMF core model.

8

Importing Entities into Design Studio

This chapter describes how to use Design Studio examples as a starting point for importing externally created Inventory entities into Oracle Communications Service Catalog and Design - Design Studio.

Importing Conceptual Model Entities

You can import conceptual model entities from external catalogs. You import external definitions using Exchange Format XML files. Earlier versions of Design Studio enabled you to use the Exchange Format XML files only for exporting entities. This enhancement enables you to also import entities, more fully integrating Design Studio into an overall design environment that includes other systems.

You can import the following conceptual model entities, along with supporting data elements, data schemas, and model projects, into Design Studio:

- Product
- Customer facing service
- Resource facing service
- Resource
- Service action
- Technical action

The Exchange Format XML file is a consistent representation of Design Studio entities and external catalog system entities. You can import multiple entities across multiple projects with a single input XML file.

You can perform a partial or complete import of the XML file. A partial import is intended to preserve existing definitions, allow new projects, entities, and elements to be added, and allow information on existing projects, entities, and elements to be added or updated. In addition to supporting incremental operations, complete imports remove enumerations and information from existing entities when this information is not included in the import file.

For more information on performing the import, see "Importing Exchange Format Data from External Catalog".

> **Note:**
>
> **Partial** import is the default option.

A partial import:

- Creates new projects, entities, and elements if they are not present in the workspace
- Appends new information and updates the existing information for entities or elements

- Leaves existing information on entities and elements, if information is not included in the import file

> **Note:**
>
> Partial import does not support the removal of information from existing entities or elements.

A complete import:

- Creates new entities or elements if they are not present in the workspace
- Replaces existing information on entities and elements with information that is provided in the import file
- Removes information from existing entities or elements if that information is not included for these entities or elements in the input file
- Handles enumerations as follows:
  - Removes existing information of entities or elements if they are missing from the input file
  - Updates existing entities or elements with information that is modified in the input file
- Handles non-list or elements (for example, Copyright information, project name) as follows:
  - Defaults existing entities or elements with information that is missing from the input file
  - Updates existing entities or elements with information modified in the input file

> **Note:**
>
> Unique external identifiers must be provided for new instances of projects, entities, and elements that are imported.
>
> Identifiers for existing or imported projects, entities, and elements cannot be modified by import operations.

The limitations on Conceptual Model imports are as follows:

- Import operations will not update sealed projects, read-only projects, or read-only entities
- Import operations will not rename existing projects, entities, or elements
- Import operations will not delete existing projects, entities, or elements of an entity

Importing Exchange Format Data from External Catalog

To import exchange format data from an external catalog:

1. Open Design Studio.

2. Ensure all changes made in workspace are saved.
3. Run a full build to ensure all model files are up-to-date.
4. Open **Studio Projects** or **Solution** view.
5. Open the Context menu by right-clicking the page.
6. Select **Import** > **Import Exchange Format**.
7. Click **Browse** and select the required exchange format file (.xml file).
8. (Optional) Select **Complete** for a complete import.
9. Click **Next**.

   The Summary Page appears.
10. Check all the entities and elements that are imported and then click **Finish**.

    A dialog box appears to open the generated import log.
11. Click **Yes** to open the log file, or click **No** to go back to the previous view.

Importing Inventory Entities

You can use the published API information included in the Exchange Format to import Inventory entities from a different application or import entities from a different Inventory application.

You can import Service specifications, Service Configuration specifications, Logical Device specifications, Custom Object specifications, Ruleset extension points, and Rulesets from external Inventory systems into Design Studio. After import, Design Studio creates an Inventory project and adds these specifications to that project.

For example, by leveraging the information in the Design Studio Exchange Format, you can enable Design Studio users import Inventory entities by creating an Import menu action that appears in the Studio Projects view.

Adding the Design Studio Import Inventory Examples to a Workspace

Design Studio includes the **Design Studio Import Inventory Examples** example, which includes projects that demonstrate how to import external Inventory entities into Design Studio. These example projects are included in the Design Studio installation and can be added to your workspace.

To add the **Design Studio Import Inventory Examples** example to a workspace:

1. From the Design Studio **File** menu, select **New**, and then select **Example**.

   The New Example wizard appears.
2. Expand the **Design Studio Examples** folder and select **Design Studio Import Inventory Examples**.
3. Click **Next**.

   The Example Projects page appears. The **Design Studio Import Inventory Examples** example includes three example projects.
4. Click each of the following example projects to read a summary of the example project:
   - The **design.studio.example.import.inventory.update.site** project creates a project that demonstrates how to export installable features into an update site.

- The **design.studio.example. import.inventory.feature** project creates a project that demonstrates how to add the import inventory example plug-in project to a feature project.
- The **design.studio.example.import.inventory** project creates a project that contains sample code for importing external Inventory entities into Design Studio.

5. Click **Finish**.

   The example projects are added to the current workspace.

About the design.studio.example.import.inventory Example Project

The **design.studio.example.import.inventory** example project includes a **plugin.xml** file that illustrates how to import Inventory entities and by leveraging the information published by the Design Studio Exchange Format.

> **Note:**
>
> The examples presented in this chapter are displayed in text form, such as that displayed on the **plugin.xml** tab of the Plug-in Manifest editor. You can configure extensions in the Plug-in Manifest editor using the form-based representation that appears on the **Extensions** tab as well. The **plugin.xml** tab and the **Extensions** tab display two views of the same information.

The **design.studio.example.import.inventory** example project illustrates how to complete the following tasks:

- Adding Import Commands to the Studio Projects View Context Menu
- Invoking the Import Inventory API Using an XML File
- Invoking the Import Inventory API Using a Resource Object
- Adding External Data to an Inventory Project
- Accessing Import Errors and Warnings

Adding Import Commands to the Studio Projects View Context Menu

The **design.studio.example.import.inventory** example project demonstrates how to add an import command to the Studio Projects view context menu. Adding Import Commands to the Studio Projects View Context Menu displays an example of the configuration of the extensions for each of the following command classes:

```
design.studio.example.action.command.handler.ImportInventoryCommandHandler
```

```
design.studio.example.action.command.handler.ImportInventoryFileCommandHandler
```

> **Note:**
>
> These command classes are provided to demonstrate two different examples, where each example uses a different menu action. You can use either menu action example as a starting point to import your inventory data.

In Example 8-1, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Import Inventory Examples** project for more information.

Example 8-1 Adding Import Commands to the Studio Projects View Context Menu

```
<extension
    point="org.eclipse.ui.commands">
    <command
        defaultHandler="design.studio.example.action.command.handler.
                        ImportInventoryCommandHandler"
        id="design.studio.example.action.command.importInventoy.command"
        name="Import Example Inventory">
    </command>

    <command
        defaultHandler="design.studio.example.action.command.handler.
                        ImportInventoryFileCommandHandler"
        id="design.studio.example.action.command.importInventoyFile.command"
        name="Import Example Inventory File">
    </command>
</extension>

<extension
    point="org.eclipse.ui.menus">
    <menuContribution
        allPopups="true"
        locationURI="popup:imports?after=additions">
        <command
            commandId="design.studio.example.action.command.
                       importInventoyFile.command"
            icon="icons/sample.gif"
            id="design.studio.example.action.command.
                importInventoyFile.command"
            label="Import Example Inventory File"
            mnemonic="%contributions.menu.importInventoryDataFile.mnemonic"
            tooltip="Import Example Inventory Data">
        </command>
    </menuContribution>

    <menuContribution
        allPopups="true"
        locationURI="popup:imports?after=additions">
        <command
            commandId="design.studio.example.action.command.
                       importInventoy.command"
            icon="icons/sample.gif"
            id="design.studio.example.action.command.importInventoy.command"
            label="Import Example Inventory"
            mnemonic="%contributions.menu.importInventoryData.mnemonic"
            tooltip="Import Example Inventory">
        </command>
```

```
    </menuContribution>
</extension>
```

Invoking the Import Inventory API Using an XML File

The **design.studio.example.import.inventory** example project demonstrates how to import inventory data from an XML file using the Design Studio Import Inventory Data Job Java API. Example 8-2 illustrates the example using the following command class:

```
design.studio.example.action.command.handler.ImportInventoryFileCommandHandler
```

In Example 8-2, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Import Inventory Examples** project for more information.

Example 8-2 Invoking the Import Inventory API Using an XML File

```
IWorkbenchWindow window = HandlerUtil.getActiveWorkbenchWindowChecked(event);
ImportInventoryDataDialog diag =
   new ImportInventoryDataDialog(window.getShell());
int result = diag.open();
if (result == 0) {
   final File dataFile = diag.getInventoryDataFile();
if (dataFile != null) {
   final ImportInventoryDataJob job =
      new ImportInventoryDataJob("Import Example Inventory File", dataFile);
   job.setUser(true);
   job.setRule(ResourcesPlugin.getWorkspace().getRoot());
   job.schedule();
} else {
   ExampleLogger.logError("Unable to import inventory data file.", null);
  }
}
```

Invoking the Import Inventory API Using a Resource Object

The **design.studio.example.import.inventory** example project demonstrates how to import inventory data from an XML resource object using the Design Studio Import Inventory Data Job Java API.

Example 8-3 illustrates the example using the following command class:

```
design.studio.example.action.command.handler.ImportInventoryCommandHandler
```

The example demonstrates how to create an inventory resource object using the Design Studio Import Inventory Data Utility (**ImportInventoryDataUtil**) and how to populate the project details using Exchange Format APIs. In Example 8-3, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Import Inventory Examples** project for more information.

Example 8-3 Invoking the Import Inventory API Using a Resource Object

```
Resource inventoryResource = ImportInventoryDataUtil.createInventoryResource();
Mappings mappings = INVENTORY_FACTORY.createMappings();
inventoryResource.getContents().add(mappings);
InventoryMappingList mappingList =
   INVENTORY_ FACTORY.createInventoryMappingList();
InventoryMapping inventoryMapping =
   INVENTORY_FACTORY.createInventoryMapping();
```

```
StudioModelEntityType inventoryCartridgeType = StudioModelEntityType.getTypeById
   (InventoryCartridgeModelFactory.ID);
Project project = MODEL_FACTORY.createProject();
final String projectName = "ExampleInventory";
project.setName(projectName);
project.setTargetVersion(getTargetVersion());
project.setVersion(getBuildVersion());
project.setKind(ElementKind.ENTITY);
project.setIdentifier(projectName);
project.setType(inventoryCartridgeType.getExternalKey());
project.setTypeName(inventoryCartridgeType.getName());
project.setId(inventoryCartridgeType.getExternalKey() + "=" + projectName);
inventoryMapping.setProject(project);
mappingList.getElement().add(inventoryMapping);
mappings.setInventoryMappings(mappingList);
// Below line shows creation of DataElementList
DataElementList dataElements = DataFactory.eINSTANCE.createDataElementList();
final ImportInventoryDataJob job = new ImportInventoryDataJob
   ("Import Example Inventory Resource", inventoryResource);
job.setUser(true);
job.setRule(ResourcesPlugin.getWorkspace().getRoot());
job.schedule();
// Adds a job change listener to add external metadata to Resource after
// finishing import job. See ExampleJobChangeListener for more
// information.
job.addJobChangeListener(new ExampleJobChangeListener(projectName));
```

Adding External Data to an Inventory Project

After you import specifications, you add all external data files required by the Inventory project to the **resources** directory. The **design.studio.example.import.inventory** example project demonstrates how to add external data to the Inventory project that is created by the Design Studio Import Inventory Data Job Java API (**ImportInventoryDataJob**). The Design Studio Import Inventory Data Job Java API is asynchronous, so you must add a job change listener to listen to the job state changes.

Example 8-4 illustrates how to add external data files using the following class:

```
design.studio.example.inventory.job.ExampleJobChangeListener
```

In Example 8-4, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Import Inventory Examples** project for more information.

Example 8-4 Adding External Data to an Inventory Project

```
URL fileURL =
   Platform.getBundle(Activator.PLUGIN_ID).getEntry("samples/sample.xml");
File dataFile;
try {
   dataFile = new File(FileLocator.toFileURL(fileURL).toURI());
   IProject project =
      ResourcesPlugin.getWorkspace().getRoot().getProject(projectName);
   if ((project != null) && (project.isOpen())) {
      IFolder resourcesFolder = project.getFolder(IStudioCartridge.DIR_RESOURCES);
      if (resourcesFolder.exists()) {
         IFile newFile = resourcesFolder.getFile("sample_copy.xml");
         FileInputStream fileStream = new FileInputStream(dataFile);
         if (!newFile.exists())
            newFile.create(fileStream, false, null);
       }
```

```
    }
} catch (CoreException | URISyntaxException | IOException e) {
     e.printStackTrace();
}
```

Accessing Import Errors and Warnings

The **design.studio.example.import.inventory** example project demonstrates how to access errors and warnings generated when you import specifications. The Design Studio Import Inventory Data Job Java API (**ImportInventoryDataJob**) is asynchronous, so you must add a job change listener to listen to the job state changes.

Example 8-5 illustrates how to access import errors and warnings using the following class:

```
design.studio.example.inventory.job.ExampleJobChangeListener
```

In Example 8-5, italics represent code that requires customization to meet your business needs. Add and review the **Design Studio Import Inventory Examples** project for more information.

Example 8-5 Accessing Import Errors and Warnings

```
Job job = event.getJob();
if (job instanceof ImportInventoryDataJob) {
   IStatus status = ((ImportInventoryDataJob) job).getStatus();
   if (status.isMultiStatus()) {
      // This can be used to find out any errors or warnings,
      // such as, missing dependencies and invalid files, and how to
      // take necessary action.
      }
      ExampleLogger.log(status);
}
```

Viewing the Design Studio Inventory Data Schema

You can view the details of the Design Studio Inventory data model by viewing the Design Studio Inventory data schema.

> **Note:**
>
> To view the Design Studio Inventory data schema, you must first add the **Design Studio Import Inventory example**. See "Adding the Design Studio Import Inventory Examples to a Workspace" for more information.

To view the Design Studio Inventory data schema:

1. In Design Studio, switch to the Java perspective.

   See the Design Studio Help for more information about switching perspectives.

2. Click the **Package Explorer** tab.

3. In the Package Explorer view, expand the **design.studio.example.import.inventory** folder, and then expand the **schemas** folder.

4. Double-click a schema file.

   The schema opens in the Data Schema editor.

9
Working with Source Control

This chapter provides information to enable you to collaborate in teams by using a source control system to share projects and describes which files must be source controlled in Oracle Communications Service Catalog and Design - Design Studio.

About Source Control

Oracle recommends that you use a source control system to manage the quality of service fulfillment design solutions. With a source control system, you can share projects across development teams, submit changes to various projects, and import projects into Design Studio. Oracle recommends that you always use a source control system in your workflow, even if only one team member is working on a development project. Using source control, you can track changes to systematically correct mistakes, roll back to previous versions, and support quality control. Design Studio can integrate with source control software, such as Apache Subversion, GIT, and Concurrent Versions System (CVS).

For team development projects, you install the source control software on a server. A server installation enables multiple developers to remotely access the hosted repository. For development projects with one developer, you can install the source control software locally on the individual computer. A local installation enables a single user to access the repository on the computer's file system. Commands run directly without the need for a server.

Source control systems are unique. Some integrate with Eclipse-based applications and others provide separate tooling. Some source control systems distribute integration plug-ins through Eclipse Marketplace, while others distribute plug-ins independently.

Configuration capabilities and procedures vary among source control systems. Oracle recommends that you consult your source control system documentation when setting up the Eclipse integration. You can integrate Design Studio with your source control feature after you configure the source control system feature for Eclipse.

About Source Control Strategies for Design Studio Files

Oracle recommends that developers work in a source control system when developing cartridges in Design Studio. Oracle Enterprise for Eclipse provides support for integrating with source control systems (plug-ins are available for most common source control systems). The behavior of Design Studio when used in an environment where the files are backed by a source control system depends on the source control system and the source control team plug-in being used. For more information about working in the team environment, see *Eclipse Workbench User Guide*. See the Eclipse Marketplace page for more information about supported source control solutions:

`http://marketplace.eclipse.org/`

Table 9-1 describes the structure of the directories and the files in a Design Studio and recommends a source control management strategy.

Table 9-1 Design Studio Source Control Management

Directory or File	Description	Source Control Management
*ProjectDir***/**	Project's top level directory.	Source control all files directly under this directory.
*ProjectDir***/cartridgeBin/**	Contains deployable archive files	Source control the directory but do not source control the contents.
*ProjectDir***/dataDictionary/**	Contains Data Dictionary schema files and companion files.	Source control.
*ProjectDir***/doc/**	Contains documentation files.	Source control.
*ProjectDir***/generated/**	Contains generated artifacts of the build process.	Source control. However, do not source control the **src** folder or its contents.
*ProjectDir***/generated/src/**	Contains generated artifacts of the build process.	Source control the directory but not its contents.
*ProjectDir***/integrityLib/**	Contains Design Studio for Network Integrity JAR files that are included in the Network Integrity server Enterprise Archive (EAR). These JARS are in the project's classpath.	Source control the directory, but do not source control the files in this directory.
*ProjectDir***/integrityLib/packaged**	Contains JAR files that are created by Design Studio for Network Integrity and which are packaged into the cartridge IAR file. The JAR files are added to the Network Integrity EAR when the cartridge is deployed. These JAR files are in the project's classpath.	Source control the directory, but do not source control the files in this directory.
*ProjectDir***/lib/**	Contains JAR files.	Source control. The **mds.mar** file is output to this directory. Do not source control the **mds.mar** file.
*ProjectDir***/mdsArtifacts/**	Contains files that contribute to the UI Hints infrastructure.	Source control the directory and the following files: • **MDSAvailablePagePanels.xml** • **MDSAvailablePagePanels.xsd** • **MDSMetaData.xml** Do not source control the the remaining files in this directory.

Table 9-1 (Cont.) Design Studio Source Control Management

Directory or File	Description	Source Control Management
ProjectDir/**model/**	Contains files that are used to persist the information about Cartridges, Actions, Processors, Model Collections, and Address Handlers.	Source control.
ProjectDir/**out**	Contains output classes.	Do not source control.
ProjectDir/**src/**	Contains the user-supplied code for the cartridge.	Source control.

10
Deploying Cartridges to Environments

This chapter provides information about the Oracle Communications Service Catalog and Design - Design Studio Cartridge Management Tool (CMT) and about other tools that you can use when deploying cartridges to run-time environments.

Deploying Cartridges to Run-Time Environments with the Cartridge Management Tool

The CMT enables you to deploy cartridges to run-time environments outside of the Design Studio environment. The CMT is bundled with the Design Studio Media Pack on the Oracle software delivery Web site.

> **Note:**
>
> Oracle recommends that you deploy to production environments using a controlled and scripted process. This process should be verified in a staging environment prior to running against a production environment. Though Design Studio enables you to deploy cartridges to design and test environments consistently across all Oracle Communications features, Oracle recommends that you use the CMT to deploy to production run-time environments.

To deploy cartridges to run-time environments using CMT:

1. Navigate to Oracle software delivery website:

   `https://edelivery.oracle.com`
2. Download and extract the Design Studio Media Pack.
3. Open the **cartridge_management_tools** folder.
4. Extract **cartridge_management_tools.jar**.
5. Open the **build.properties** file and update the file with environment-specific variables.

   For example, update web service connection information for deploying and undeploying cartridges, model variable values to be used at deploy time, and so forth.

   If you are deploying Oracle Communications Unified Inventory Management (UIM) or Oracle Communications Network Integrity cartridges, you must add the following properties to the file:

   ```
   deploy.wladmin.host.name=WebLogic administration server host name
   deploy.wladmin.host.port=WebLogic administration server port number
   deploy.wl.target.name=adminstration server name or cluster name where the
   application is deployed
   ```
6. Add the files in the **lib** folder (in the **cartridge_management_tools** file) to the classpath.

Adding libraries to the classpath can prevent exceptions from being thrown while running the Ant script.

7. Set the JAVA_HOME to JDK 8.
8. From a command prompt, navigate to the directory where the **cartridge_management_tools.jar** file is extracted and run the following Ant commands, as applicable:
   - **ant**: Displays a list of available targets.
   - **ant -lib ..\lib -f build.xml deploy-cartridge**: Deploys a cartridge to a run-time environment.
   - **ant -lib ..\lib -f build.xml undeploy-cartridge**: Undeploys a cartridge from a run-time environment.

     > **Note:**
     >
     > Undeploy is not supported for UIM cartridges.

   - **ant -lib ..\lib -f build.xml list-cartridge**: Lists the cartridges deployed to a run-time environment.

Working with Additional Cartridge Deployment Tools

Oracle recommends using the CMT for automated, scripted, and command line cartridge deployment for all the applications supported by Design Studio. Under specific circumstances, applications might recommend other tools for cartridge deployment, such as the following:

- Service Activation Deployment Tool (SADT), used to deploy SAR files to Oracle Communications ASAP run-time environments. See *ASAP Cartridge Development Guide* for more information.
- Cartridge Deployer Tool, used to deploy JAR files to Oracle Communications Unified Inventory Management (UIM) run-time environments. See *UIM System Administrator's Guide* for more information.
- Cartridge Deployer Tool, used to deploy cartridges to Oracle Communications Network Integrity run-time environments. See *Network Integrity Installation Guide* for more information.
- XML Import/Export Tool, which is used to manage data in the Oracle Communications Order and Service Management (OSM) database. See *OSM System Administrator's Guide* for more information.

See the Oracle Communications application documentation for specific instructions and applicability notes and to determine the level of support for application-specific tools.

11

Working with Externally Created Data Schemas

This chapter describes how you work with data schemas that you create outside of Oracle Communications Service Catalog and Design - Design Studio. Design Studio uses XML data schemas for domain modeling. XML data schemas store data definitions and standardize data usage across domains and platforms.

About Design Studio Data Schemas

You create an XML data schema by either creating a Model project or by creating a schema in an application project. Design Studio saves the data schema entity to the root level of the project's **dataDictionary** directory by default. The Data Schema entity is represented by an XSD file and a companion file in the local file system.

You can also import any XML data schema file that was not created in Design Studio into a Design Studio project. If you create an XML data schema externally and edit it in Design Studio, you may lose data at the schema level and in the data model.

Modeling Data Using XML Data Schemas

Design Studio uses XML data schemas for data modeling. XML data schemas provide precise descriptions of data models that are bound by strict sets of rules, and they generate entities and associated features of a model. XML data schemas also support domain-specific requirements, such as inheritance and abstraction.

Design Studio simplifies using XML data schemas with the Data Dictionary. The Data Dictionary is a logical collection of data elements within the workspace and is presented within a set of views. These views enable you to visualize and manage the data elements configured in the workspace.

The Data Dictionary presents the data types available for use in the workspace. Various entities in the workspace contribute to the contents of the Data Dictionary. Common entities like Schema and Business Entity enable non-product-specific data type modeling. Product entities can also contribute data type information to the Data Dictionary.

The data schema is represented in Design Studio as two files: an XML schema file and a companion file. Design Studio uses the companion file to save payloads that are declared in a data schema. For example, a provisioning system may require that all root level elements have cardinality. However, the XML schema does not support cardinality for root level-type definitions. Design Studio saves this information in the data schema companion file. The companion file is hidden and is not visible in the Studio perspective.

About Supported XML Schema Features

Design Studio data schemas support a subset of the features of the XML Schema language. Some of the supported features are enhanced to optimize their use when modeling data.

Table 11-1 lists the XML Schema features that are supported.

Table 11-1 Supported XML Schema Features

Feature (Type)	Description
Type Declaration	Supports the same definitions as the XML Schema specification.
Target Name Space	Supports the same definitions as the XML Schema specification.
Complex Content Type Definitions	Supports child structures and complex types.
Import Directives	Supports the same definitions as the XML Schema specification.
Cardinality/Occurrence	Supports the cardinality/occurrence of the elements (child simple element from the XML Schema perspective). Also supports the cardinality on the type definitions on the root level, as well as child complex elements. Supports the various modeling needs of provisioning systems.
Max Length	Supports maximum length facets of XML Schema. All rules of maximum length facets apply.
Min Length	Supports minimum length facets of XML Schema.
Enumeration	Supports the enumeration feature of the XML Schema specification, and enumerations for non-string elements. Fulfills the requirements of the various modeling needs of provisioning and inventory systems. Derived elements can extend the base enumerations, or exclude (restrict) them.
Annotations	Supports annotations on the elements and type definitions.
Deriving types by extension	Supports type definitions extended from other type definitions. While you can model recursive structures in Design Studio, the system restricts the presentation of recursive structures. In Design Studio, a data schema is represented as a hierarchical tree. Design Studio does not allow infinite expansion of recursive tree nodes. You can limit the number of levels to which nodes are expanded. See Design Studio Help for information about defining preferences.

Table 11-1 (Cont.) Supported XML Schema Features

Feature (Type)	Description
Primitive Data Types	Supports the following primitive data types: • int • string • long • float • double • date • date and time • time • boolean • decimal • hexidecimal

About Unsupported Schema Directives and Elements

Design Studio data schemas support a subset of the features of the XML Schema language.

Table 11-2 lists the XML Schema features that are not supported.

Table 11-2 Unsupported Schema Directives and Elements

Schema Directives and Elements	Description
Include Directives	Valid external schemas that are imported into Design Studio and have an include directive may cause unresolved type definitions and can be considered invalid by Design Studio validation framework.
No target namespace	Schemas that have no Target Name space defined are not supported.
Redefine	If an external schema is using the Redefine element, the validation may result in unresolved elements.
Abstract Element and Types	If an external schema contains an Abstract definition, Design Studio considers it a regular element and does force a substitution according to the XML Schema specification.
Attribute Declaration and Attributes Groups	If an external schema contains an Attribute Declaration, Design Studio considers it to be read-only element. If an external schema contains an Attributes Group, Design Studio may not recognize it as a valid type.
Substitution groups	Design Studio ignores this attribute of an element.
Element Declarations	Design Studio considers any external schema containing element declarations as read-only type definition.

Table 11-2 (Cont.) Unsupported Schema Directives and Elements

Schema Directives and Elements	Description
Unsupported Primitive Types	If an external schema has an element declaration with a primitive type that is not supported, Design Studio considers the type definition as undefined and shown as none.

12

Design Studio Platform Tools

This chapter provides an overview of Oracle Enterprise Packet for Eclipse and Java Development Tools, that you use with Oracle Communications Service Catalog and Design - Design Studio.

Working with Oracle Enterprise Packet for Eclipse

Design Studio supports Oracle Enterprise Pack for Eclipse, a set of certified plug-ins designed to help you develop and debug Java EE applications that can be deployed on Oracle WebLogic Server from Eclipse.

Design Studio uses the Eclipse platform as a product framework and as an integrated development environment (IDE) to support plug-in architecture and customizations. Eclipse provides a GUI to manage and configure data across Oracle Communications products.

Eclipse supports application development tool construction, independent tool vendors, GUI and non-GUI application development, numerous content types (including Java, HTML, C, and XML), tool integration, and use of Java language for writing the tools.

For more information about installing Oracle Enterprise for Eclipse, see *Design Studio Installation Guide*.

> **Note:**
>
> To run Oracle Enterprise for Eclipse, system administrators must install the correct Java Runtime Environment and Java Developer Kit. See *Design Studio Installation Guide* for more information.

Oracle Enterprise for Eclipse includes a number of tools that are useful for middleware development and includes all of the required features to support Design Studio. These tools compliment Design Studio features to provide a more complete design environment for building solutions.

About Java Development Tools

Java Development Tools (JDT) provide a set of workbench plug-ins that add the capabilities of a full-featured Java IDE to the Eclipse platform. JDT plug-ins provide APIs that can be further extended by other tool builders. Additionally, the JDT includes a built-in Java compiler that compiles Java code and creates error messages when compilation fails.

About Database Development Tools

Oracle Enterprise for Eclipse provides tools to help you develop applications that use Oracle Database. These tools include:

- Support for the integration of Oracle Database with Eclipse Data Tools Platform.
- Diagram viewers for visualizing database schemas and object-relational mapping.
- Database Explorer, which supports data editing, data load/extract, and Data Definition Language generation.
- SQL tools that support SQL editing, running and stored procedures.

About Application and WebLogic Server Tools

Oracle Enterprise for Eclipse editors and optimized development tools for application server development simplify work with products like WebLogic Server by providing:

- Fast, iterative deployment for local and remote servers.
- Support for JAX-WS web services (editor and configuration support).
- Oracle Enterprise for Eclipse visual deployment descriptor editors for ***-jms.xml**, **weblogic.xml**, **weblogic-application.xml**, **weblogic-ejb.jar.xml**, **faces-config.xml**, and **persistence.xml** files, and a JSR 88 deployment plan editor.

About Web Application Tools

Oracle Enterprise for Eclipse web application tools simplify working with technologies like JSF, JSP, CSS, ADF, and others.

Web Applications

The following web application tools simplify analyzing and visualizing dependencies to reduce run-time debugging and to improve code quality:

- AppXRay and AppXaminer offer compiler-level awareness of Java, Oracle Application Development Framework (Oracle ADF), HTML, CSS, JSP, JSTL, and JSF at design time, with capabilities in code and annotation completion, code navigation, dependency visualization, consistency checking with generated classes and configuration files, pre-build error checking, and validation.
- An enhanced Eclipse Web Tools Platform (WTP) Web Page Editor (WPE) includes a Smart Property Sheet to simplify tag configuration and data binding. WPE also includes localization support, JSF, JSP, JSTL, and CSS/HTML, and a Tag and Data Palette.

Web Tools Platform

The WTP extends the Eclipse platform to simplify web and Java EE application development. WPT provides the following tools to simplify deploying, running, and testing applications:

- Core Web Tools Platform (EJB Tools, Java EE Tools, Server Tools)
- JavaServer Faces Tools
- Data Tools Platform Project

Oracle ADF Tools

Oracle Enterprise for Eclipse provides design-time support for application development with Oracle ADF. You can create applications that leverage Oracle ADF Faces and Task Flows, validate and refactor Oracle ADF dependencies using AppXRay, deploy

and debug with Oracle WebLogic Server, and create Oracle ADF Libraries for application reuse.

The following Oracle ADF tools simplify Java EE development by minimizing the need to write code that implements the application's infrastructure:

- Oracle ADF Server Extensions to configure WebLogic Server and Eclipse for ADF development
- ADF Project Templates and Facets
- ADF-enabled JSP Templates
- ADF design time support
  - WPE enhancements for ADF source development
  - Smart Property Sheet to simplify tag configuration and data binding
  - Tag Palette enabled for ADF Faces and Data Visualization Tools (DVT) components
  - Tag editors for drag and drop configuration of ADF tags
- AppXRay support for ADF tags

About JPA and Oracle Coherence Tools

The following Oracle Enterprise for Eclipse tools help you create applications that map objects to relational databases.

Java Persistence API (JPA) Tools

JPA tools simplify working with the Java programming language framework to manage relational data in applications using Java Platform, Standard Edition, and Enterprise Edition. These tools enable you to:

- Use object relational mapping tools for Java Persistence API
- Generate entities from schema
  - Start with any database connection
  - Create entities based on table relationships
  - Define new entity associations
- Generate entities from POJO
  - Annotate existing Java class
  - Map POJO fields and properties to database schema
- Use the Entities editor, which includes:
  - A design view to display and edit existing entity relationships
  - Hyperlinked navigation to entity source code
- Use the JPA Details view, enabling you to:
  - Edit entity properties and relationships from the Entities editor
- Support EclipseLink/TopLink, Kodo, OpenJPA, and generic JPA providers

Oracle Coherence Tools

Oracle Enterprise for Eclipse supports Oracle Coherence. You can run, deploy, and debug Coherence servers from Eclipse, as well as create and configure projects, and leverage visual editors for cache configuration and override descriptors.

About Third-Party Tools

Design Studio and the Eclipse platform support third-party tools that are helpful for readiness, testing, and other administrative activities. For example, you might use:

- soapUI, as a test tool for calling web services.
- HermesJMS, as a test tool for sending and receiving JMS messages.
- SQL Developer, as a utility for maintaining the database.
- XQDT for Eclipse or oXygen XML Developer, as an XQuery editor.

13

Extending Solution Designer

This chapter describes how to extend Oracle Communications Service Catalog and Design - Solution Designer using an extended designer class. The extended designer class provides extension points using which you can customize the intended default behavior in the run-time application.

About Extending Solution Designer

You can extend the default behavior that Solution Designer offers using the extended designer class. The extended designer class is an extended java class that provides the ability to add or customize logic to the automatically generated implementation using predefined extension points. Extension points are the specific points where the custom code is run. Each extension point specification contains a signature of a method that calls the extension point.

When you publish the initiative to a workspace in the Solution Designer user interface, the DevOps engine generates the cartridge artifacts and the cartridge workspace. See "Publishing Initiatives to Workspaces" in *Solution Designer User's Guide* for information on publishing initiatives. You can download the cartridge workspace and import in to Design Studio Eclipse Workspace. The cartridge workspace has the following java files:

- *SpecificationName***Designer.java**: This is the base class which contains the implementation that will be run by default. Oracle recommends that you do not modify the base class.
- *SpecificationName***ExtendedDesigner.java**: This is extended designer class or implementation class which lets you write the implementation based on your business requirements. Oracle provides predefined extension points for the methods in base class; however you can extend any method from the base class by copying and editing the method signatures from the existing extension points.

Where *SpecificationName* is the specification name that you entered in the Solution Designer application. The specification includes CFS, RFS, and resource.

These files are located at **UIM**_*DomainName*_**DesignJ\src\oracle\communications\inventory\techpack***SpecificationName* where *DomainName* is the name of the domain that you entered in the Solution Designer application.

About Extended Designer Class

Extended designer class is an extended java class that lets you add or customize logic based on your business requirements to the automatically generated implementation using predefined extension points. This section lists the predefined extension points and also provides the sample extended designer class.

About Extension Points

Extension points are the specific points where the custom code is run. Each extension point specification contains a signature of a method that calls the extension point. This section provides the list of available extension points and their description. Table 13-1 lists the extension points that are available in the extended designer class.

Table 13-1 Predefined Extension Points

Extension Points	Description
chooseSpecFor*Component* where *Component* is the component that you selected while defining an advanced policy in Solution Designer application. See "Defining Advanced Policies" in *Solution Designer User's Guide* for information on defining advanced policies.	Add custom logic to implement the advanced design policy.
preDesignAdd	Add custom logic to run before the default design and assign logic for add action.
postDesignAdd	Add custom logic to run after the default design and assign logic for add action.
preDesignChange	Add custom logic to run before the default design and assign logic for change action.
postDesignChange	Add custom logic to run after the default design and assign logic for change action.
preDesignDisconnect	Add custom logic to run before the default design and assign logic for disconnect action.
postDesignDisconnect	Add custom logic to run after the default design and assign logic for disconnect action.
postDesignSuspend	Add custom logic to run after the default design and assign logic for suspend action.
postDesignResume	Add custom logic to run after the default design and assign logic for resume action.
postDesignMove	Add custom logic to run after the default design and assign logic for move action.
preDesign	Add custom logic to run before the **Design** method in the base class. The **Design** method is the boilerplate method for automating the design and assign logic.
postParseActionParameters	Add custom logic to run after **ParseActionParameters** method in the base class. **ParseActionParameters** method parses the design parameters from the inbound request.
postAutoconfigProperties	Add custom logic to run after **AutoconfigProperties** method in the base class. **AutoconfigProperties** method is used to populate characteristics based on the action parameters from the inbound request.

Table 13-1 (Cont.) Predefined Extension Points

Extension Points	Description
validate	Add custom validation logic based on your business requirements.
beforeComplete	Add custom logic that must be run after the **beforeComplete** method in the base class.
complete	Add custom logic that must be run after the **Complete** method in the base class.
cancel	Add custom logic that must be run for canceling the service for various design actions such as add, disconnect, suspend, and so on.
issue	Add custom logic that must be run for changing the service status to issued for various design actions such as add, disconnect, suspend, and so on.
suspend	Add custom logic to run after the default design and assign logic for suspend action.
resume	Add custom logic to run after the default design and assign logic for resume action.
postSet*ChildSpecification*For*Component*	Add custom logic to map any design parameters from the specification to its child specifications.

Working with Extended Designer Class

You can write the custom logic in the extended designer class for defining advanced policy, design action mapping, delivery action mapping, parameter mapping, and so on. Oracle provides predefined extension points for the methods in the base class. You can extend any method that is available in the base class and override the default behavior in the extended designer class. The extended designer class is generated for each specification in the cartridge. When you define custom logic in the extended designer class, the custom logic is included in the cartridge only when you enter the advanced policy implementation assets information in the Solution Designer application. See "Adding Advanced Policy Implementation Assets" in *Solution Designer User's Guide* for information on adding advanced policy implementation assets.

To extend using extended designer class:

1. Publish the initiative to the **Test** workspace for generating the cartridge workspace. See "Publishing Initiatives to the Test Workspace" in *Solution Designer User's Guide* for information on how to publish an initiative.
2. Download the cartridge workspace from the location displayed in the **Test** workspace for the initiative that you published.
3. Import the cartridge workspace into Design Studio Eclipse environment. See "Importing Projects" in *Design Studio Modeling Basics* for more information on importing projects.
4. Locate the extended designer class at **UIM**_*DomainName*_**DesignJ\src\oracle\communications\inventory\techpack***SpecificationName*.

5. You can write the custom code for the predefined extension points. You can also extend and override any of the methods available in the base class. The notes that you add in the advanced policies in the Solution Designer user interface are added as comments in the extended designer class. See "About Extension Points" for the list of predefined extension points.
6. After you complete the custom code, upload the extended designer class with the file extension as java in S3-compatible object store.
7. Add the object store's relative path in the Solution Designer user interface and also select the advanced policies for which the custom code has been implemented in the extended designer class. See "Adding Advanced Policy Implementation Assets" in *Solution Designer User's Guide* for information on adding implementation assets for advanced policies.
8. (Optional) You can use helper classes in the extended designer class. Helper Class is a Java class that contains reusable basic functions such that you do not have to implement again and again. You must compress the helper class java files to a ZIP file and upload the ZIP file in S3-compatible object store. Add the ZIP file's relative path in the **Implementation Assets** field in **Domains** in the Solution Designer user interface. See "Creating Domains" in *Solution Designer User's Guide* for information on adding helper class details.
9. After advanced policy implementation assets information is added, re-publish the initiative until the publish operation is successful. Download and deploy the cartridge artifacts in the UIM run-time environment. See "Publishing Initiatives to Workspaces" in *Solution Designer User's Guide* for information on publishing initiatives.

The base class contains the design and assign logic for the add action. You can choose to override the implementation for add action in the extended designer class. For all the other actions such as change, suspend, resume, disconnect, and so on, you can write the implementation code using the extension points in the extended designer class.

Example: Extended Designer Class

The DevOps engine generates the extended designer class with a placeholder for custom logic. The following java methods are an example for adding custom logic for design and assign for add action. Similarly, the DevOps engine generates methods for all the extension points that is listed in Table 13-1.

```
// Custom code blocks for actions specific operations starts here.

    /**
     * Custom code that executes at the beginning of the designAdd
method.
     */
    @Override
    protected void preDesignAdd(ServiceConfigurationVersion config)
            throws ValidationException {

        // Custom code can be added here if needed.
    }
      /**
     * Custom code that executes at the end of the designAdd method.
```

```
     */
    @Override
    protected void postDesignAdd(ServiceConfigurationVersion config)
        throws ValidationException {

        // Custom code can be added here if needed.

        // Like switching the context to a specific BI by setting its
        // value to a specific BI value or setting it to null as shown
        // below based on the requirement.

        // BusinessInteraction currentBi =
oracle.communications.inventory.api.entity.utils.ConfigurationUtils.getAssoci
atedBusinessInteraction(config);
        // biManager.switchContext(currentBi, null);
    }
```