Oracle Warehouse Builder 10*g* Release 2

Repository Extensibility Cookbook

May 2006

ORACLE

Note:

This document is for informational purposes. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle. This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

Oracle Warehouse Builder 10*g* Release 2

Repository Extensibility Cookbook

INTRODUCTION

This cookbook will walk you through extending the existing OWB objects to store your own information, creating Icon sets, and defining and using your own objects and associations within the OWB repository. These features permit users to get a more 'complete' view of their business by storing relevant information together.

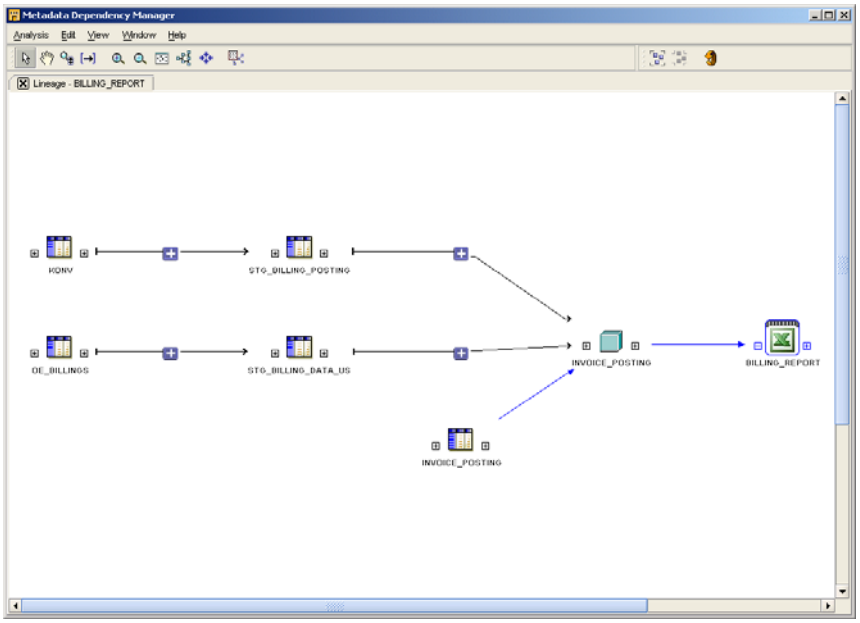


Figure 1 Extending lineage information to Excel spreadsheets

This is particularly useful when combined with OWB's metadata services such as Impact Analysis. As can be seen in Figure 1, by extending the repository to store information on for example Excel spreadsheets and linking these to the rest of the Warehouse Builder metadata true value can be derived from this metadata.

More importantly true information can now be extended into the analyst community, rather than just help the developer community.

CONCEPTS AND TERMINOLOGY

To understand the steps we are going to take in the example in this cookbook, we need to introduce a set of concepts and terms.

*OMB*Plus*

OMB*Plus is a script execution client that is shipped as part of the OWB software distribution. There is a command-line version (Start → Warehouse Builder → OMB*Plus on Windows or $ORACLE_HOME/owb/bin/unix/OMBPlus.sh on all Unix platforms). New in OWB 10gR2 is an OMB*Plus window within the Design Center itself (appearance toggled via the menu Window → OMB*Plus). This shares the Design Center's current connection to the repository.

User-Defined Properties (UDP's)

For every type of object within OWB (Modules, Tables, Files, etc. etc.), there is a set of properties defined to store information about instances of those objects (e.g. Name, Description, Business Name, etc. – to view, right-mouse click on any object from the design center tree → Properties to bring up the Property Inspector). Users can add their OWN properties to store information pertinent to their business. The definition / extending of the object for these new properties is only available through scripting (via OMBREDEFINE command), however once defined, the properties on the *instances* of the objects may be set either in scripting or via the Design Center UI (property inspector).

Icon Set

A named set of icons. A set is comprised of 3 different types of Icons: Canvas, Palette and Tree icon. Icons ideally are of a particular size – however it not, they will be automatically resized. Icon sets are used within projects but are standalone objects (i.e. are not part of any specific project and thus are available from the Global Explorer tree with the Design Center). Icon sets can be associated to an instance of an object within the OWB Client, or at the object type level though scripting (and thus inherited by any instance of that object).

Canvas Icon

An image ideally 32 x 32 pixels in size which will be used when displaying the object/instance of the object in an editor e.g. Mapping editor or the HTML reports (e.g. Lineage). The image must be GIF or JPEG format.

Palette Icon

An image ideally 18 x 18 pixels in size which will be used when displaying the object/instance of the object in an OWB Client palette. The image must be GIF or JPEG format.

Tree Icon

An image ideally 16 x 16 pixels in size which will be used when displaying the object/instance of the object in the tree. The image must be GIF or JPEG format.

First -Class Objects (FCO's), Second-Class Objects (SCO's)

These objects are at the heart of OWB repository model and they are important to understand. As an example, consider a Table in an Oracle Module stored in the OWB repository. The Table object is a **First-Class Object**. A Column within the Table is a **Second-Class object**. Constraints are also Second-Class objects. All of these objects (first and second) have properties, and there exist relationships (rules) between these objects.

The following diagram illustrates this:

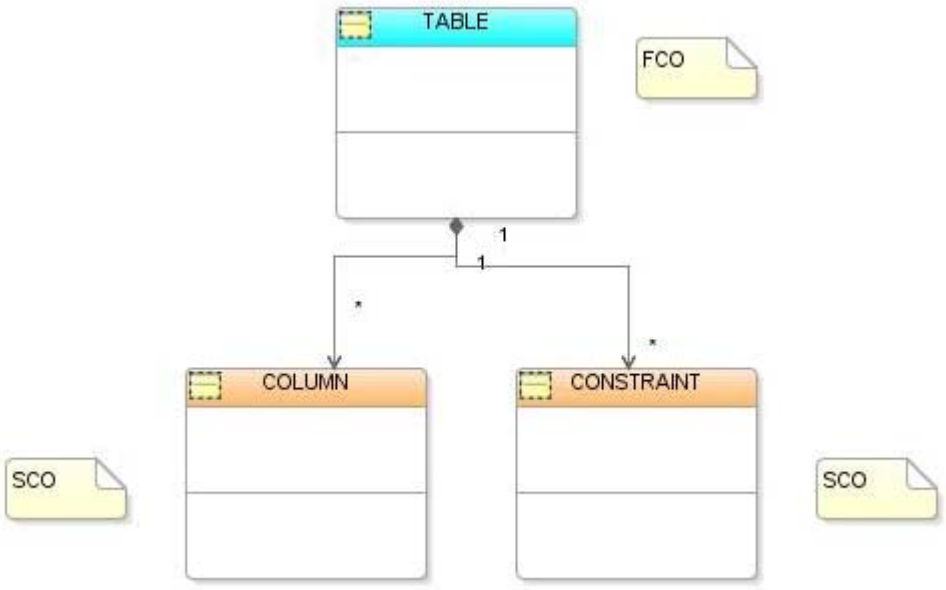


Figure 2 FCO and SCO diagram

The difference between an FCO and an SCO is that an FCO can exist on its own, but an SCO must have a parent. SCO's can have SCO's or FCO's as parents.

Associations

These are defined rules / relationships between objects – they define, for example, what types of objects can be associated with each other as well how many instances (one-to-one or one-to-n/many). For example, an External Table as defined in the Oracle database is an object. The file that it points to is another object. If this was to be modeled in the OWB repository, there would exist an Association between the external table object and the file object.

MaximumCcardinality

MAX_CARDINALITY allows you to limit the number of objects of that class which participate in that association. For example, consider the association between UniqueKey and Foreign Key. The max_cardinality of uniqueKey is 1, because a given foreignKey object can be associated with at most 1 uniqueKey object. MAX_CARDINALITY can be set to any positive integer, or the reserved word 'INFINITE'.

Navigable

ROLE_1_NAVIGABLE is used by Lineage/Impact analyzer, if set to TRUE, it means that the analyzer can traverse the association from either side of the association (E.g. from Class_1 to Class_2 and vice versa). If that property was set to FALSE, it means you only traverse the relationship from Class_1 to Class_2, and not the other way around.

Dependency Definition

DEPENDENCY_DEFINTION is a mandatory parameter of an association and for the Warehouse Builder 10*g*R2 release that must always be set to 'DATAFLOW'. In future releases other values will be allowed, allowing users to specify if the relationship is a structural dependency instead (e.g. a Table that implements a dimension – there is no data flow here, but it's useful to know if the dimension changes, the table would be impacted).

Composition

This is a type of relationship between a component and a sub-component - the lifetime of the subcomponent is controlled by the parent (eg a column doesn't exist without a Table)

Folders

These are containers for FCO's. FCO's can only exist within a Folder. Folders can contain other folders as well. Similar to FCOs, folders can contain SCOs of their own.

Modules

A module is a special type of folder. Similar to a folder, a module can contain FCOs, folders and even it's own SCOs. However, a module cannot contain another module, and a module can only have the project as its parent. A module is usually used to represent an application system (like an Oracle DB, a SAP application, a file system, etc.).

User-Defined Objects (UDO's)

Users can define their own object types (which can assume a role of a Module, FCO, SCO or Folders) and properties in the Warehouse Builder repository. E.g. they can define an object called a 'Java Class' that has a property called 'Name', and

has a SCO called 'Methods', and each Method SCO has a property called 'Name' and an association to a Table (so allowing users to model a Java method in a class that uses JDBC, and thus would be impacted by a change to a Table).

Like user-defined properties, UDO definition is done in scripting (via OMBDEFINE CLASS_DEFINITION…).

Note that when creating UDOs, you get a number of properties 'for free', along with each object: NAME (specified when you're defining the object) BUSINESS_NAME, PLURAL_NAME, and DESCRIPTION.

Once defined, instances may be created via scripting or in the UI. UDOs appear under the 'User Defined Modules' tree node in the Project Explorer and also in the Global Explorer. UDO's are regular Warehouse Builder objects and thus they participate in (and can take advantage of) all the powerful meta-data management facilities (such as impact analysis) offered by Warehouse Builder.

The following UML diagram shows the OWB concepts of Module, Folder, FCO and SCO are related to each other:

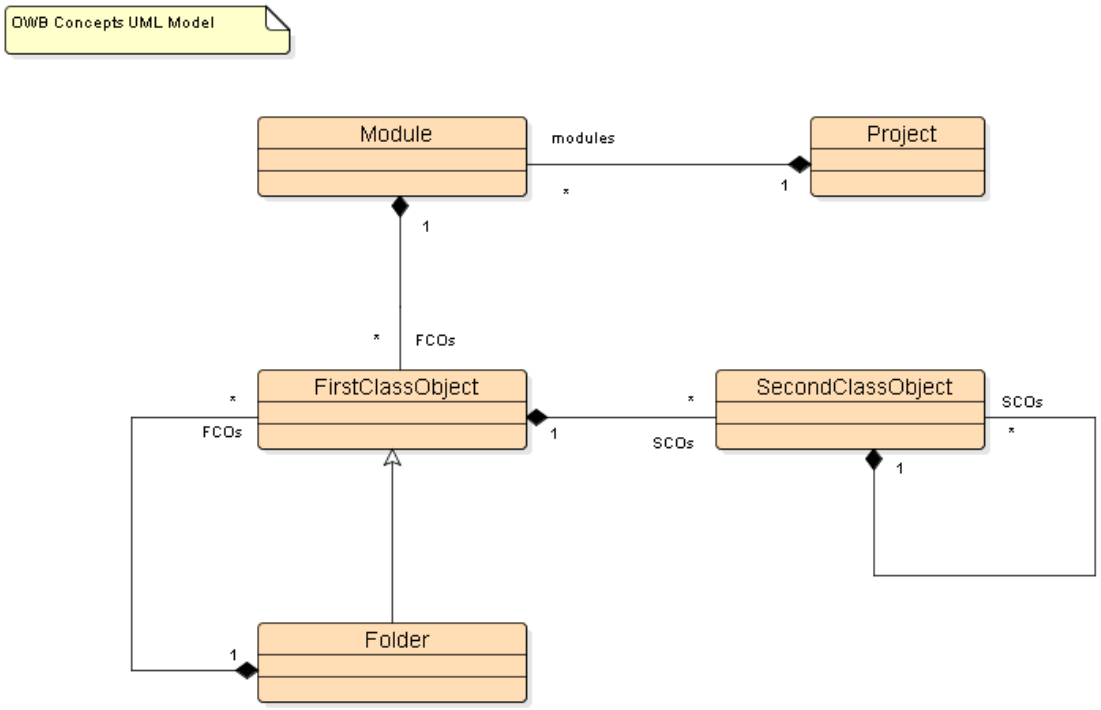


Figure 3 The component model

A lot of terms have been described. To help understand them and their relationship to each other, we can consider that there are actually two models: a class model and a component model. When we talk about the class model, we talk about types (classes), attributes, associations, and compositions. When we talk about the component model, we talk modules, folders, FCOs (aka Components) and SCOs.

The connection between the class and the component model is that each class has a responsibility (it is either a module, a folder, an FCO or an SCO). The above diagram (Figure 3) represents the component model.

DEFINING AND UNDEFINING

If you are used to SQL and Oracle databases, you know that you can do a number of common definition tasks on your objects. At a high level these are Create, Drop, Alter. After creating the objects you can apply DML commands, which is discussed later.

Similar things are done when working with user defined objects. Before we dive into the details on how exactly to create a user defined object, we will walk through the high level task flows.

Defining UDOs

Below is the basic outline to follow when defining user-defined objects:

Begin with a module

There must be a Module. A module is a type of folder. A module needs to be defined as the topmost container for the remainder of the objects in the UDO application. A module may contain no folder, only FCOs.

To define a Module:

1. Create a MODULE CLASS_DEFINITION (optionally with properties)
2. Make this class a Folder by creating a FOLDER_DEFINITION with the **same name** as the Module class

Define an FCO

There should be at least one FCO within the folder.

To define an FCO:

1. Create an FIRST_CLASS_OBJECT CLASS_DEFINITION (optionally with properties)
2. Add this FCO class to the Module class (which has now assumed the role of a Folder) as a CHILD_TYPE. This creates the composition relationship between the module class and the FCO class in the class model. You can add new FCO types at any time.
3. Add the FCO class to the FOLDER_DEFINITION of the module (This includes the FCO in the boundaries of the folder in the component model).

Set the role/responsibility

The FCO class must assume the role/responsibility of a Component (aka Component) in order for any subordinate object(s) to appear on the Design Center tree.

This is accomplished as follows:

1. Create a COMPONENT_DEFINITION with the **same name** as the FCO

Add SCOs – optional

The UDO could now be considered complete, but you can also optionally define one or more SCO objects:

1. Create an SECOND_CLASS_OBJECT CLASS_DEFINITION (optionally with properties)
2. Add the SCO class to FCO/SCO class directly owning the SCO as a CHILD_TYPE. This creates the composition relationship in the class diagram. You can add new SCO types at any time.
3. Add the SCO class to the COMPONENT_DEFINITION owning the SCO (this includes the SCO within the boundaries of the FCO in the component model)

Note:

The COMPONENT_DEFINITION contains the FCO classes and **ALL SCO classes in that hierarchy** (even if the SCO is owned by another SCO which is owned by the FCO). If this rule is not adhered to, the SCO's will not be visible to users.

Add associations

Objects now need associations (rules) to govern what they can be associated with. This only done in/with the class model, so class responsibilities (i.e. module, folder, FCO, SCO) are not relevant here. The association can be between any two classes, regardless of their responsibility.

To create an association between two object types:

1. Create an ASSOCIATION_DEFINITION for each object-to-object definition.
2. Add a DEPENDENCY_DEFINITION of 'DATAFLOW' to the association.

You UDO is now defined, and you are ready for the next steps.

Querying UDO Definitions

There are two main methods of querying the UDO definitions in your repository. The first is to simply use the Design Center's Project Exporer, the second is to use OMB*Plus to query the model.

Using the UI

Find the UDOs by looking under the User-Defined Modules node in either the Project Explorer or the Global Explorer.

Programmatically

If you go this route, there are two steps to follow.

First, issue the following (remember classes/associations beginning 'UD_' are user-defined):

OMBDESCRIBE MODEL 'OWB' GET CLASS_DEFINITIONS

OMBDESCRIBE MODEL 'OWB' GET ASSOCIATION_DEFINITIONS

Note: This shows you ALL the classes defined in the repository.

Second, once you know the name of the UDO classes, you can use the following:

- OMBDESCRIBE CLASS_DEFINITION '<class name>' GET CHILD_TYPES
  This lists all the child classes directly owned by the named class (and the named class can be FCO or SCO). If there is a hierarchy of classes then this command needs to be executed for **each named type** in the hierarchy to determine if there are any further child types.
- OMBDESCRIBE CLASS_DEFINITION '<class name>' GET ASSOCIATION_DEFINITIONS
  This lists the names of the associations defined at that class level (and the named class can be FCO or SCO). If there is a hierarchy of classes then this command needs to be executed for **each named type** in the hierarchy to determine if there are any further associations.
- OMBDESCRIBE FOLDER_DEFINITION '<folder fco class name>' GET FIRST_CLASS_OBJECTS
  This lists all the FCO's within the named FOLDER_DEFINITION
- OMBDESCRIBE COMPONENT_DEFINITION '<fco class name>' GET SECOND_CLASS_OBJECTS
  This lists all the SCO's within the named COMPONENT_DEFINITION

Note: Remember that ALL SCO's are contained in the COMPONENT_DEFINITION

Un-Defining UDOs

Un-defining UDO's is done in reverse of the definition. The general method to follow is listed in these steps.

Delete the instances

First delete any/all instances of the UDO within the repository. You can look in the User-Defined Modules nodes in the Global Explorer and Project Explorer. You can also look in the public views to find this information across the entire repository:

- ALL_IV_UDO_MODULES

- ALL_IV_UDO_FCOS
- ALL_IV_UDO_SCOS
- ALL_IV_UDO_ASSOCIATIONS
- ALL_IV_UDO_FOLDERS

Note: If you don't delete the instances first, you can still undefine the objects, but the instance data will stay around as garbage in the repository – hence why it is recommended to first delete all instances.

Remove Associations

For **each association**, issue the following:

OMBUNDEFINE ASSOCIATION_DEFINITION 'UD_<association name>'

Start at the **lowest level** object defined and work your way up - so start with SCO's (if created), then FCO's and then finally the Module class.

Issue the following for **each** class defined:

OMBUNDEFINE CLASS_DEFINITION 'UD_<class name>'

Note: You do not have to remove the FOLDER or COMPONENT definitions. These are roles that specific classes take on/are promoted to, so un-defining the class is sufficient.

Removing some but not all FCOs

If you have a UDO module class containing multiple FCO classes and want to remove one or more (but not all) FCO's, you need to do the following. Make sure you do so after deleting all instances!

First remove it from the FOLDER_DEFINTION:

OMBREDEFINE FOLDER_DEFINITION '<name>' DELETE <fco_class_name>

Undefine the associations and then the classes as was described before in the previous sections.

Removing some but not all SCOs

If you have an FCO containing multiple SCO's and you want to remove one or more (but not all) of the SCO's, you need to (after deleting all instances):

First remove it from the COMPONENT_DEFINITION

OMBREDEFINE COMPONENT_DEFINITION '<name>' DELETE <sco_class_name>

Undefine the associations and then the classes as discussed before.

UDO INSTANCES

In SQL, after creating the objects you can apply DML commands to insert data in your objects. With UDOs you will do something similar but in this case you are adding instances to the definitions.

Creating Instances

You can create UDO instances in two ways, graphically via the User-Defined Module nodes in the Project and Global Explorer and programmatically.

Programmatically

Analogous to creating instances of OWB types, as follows:

1. Create a UDO Module (OMBCREATE)
2. Change context to the newly created UDO Module
3. Create a UDO FCO (and optionally specify property values) (OMBCREATE)
4. Alter the FCO to add any SCO (again, optionally specifying any property values) (OMBALTER)
5. Alter the SCO to add any further child SCO's (OMBALTER)
6. Alter the FCO and modify it's SCO to reference the OWB FCO/SCO (via an association) (OMBALTER)

Note: in all instances, you need to reference the UDO class types you previously defined (prefixed with UD_)

Querying of UDO instances

You can query UDO instances in two ways, graphically via the User-Defined Module nodes in the Project and Global Explorer and programmatically.

Programmatically

Querying UDO instances is done as follows:

1. OMBLIST <module class_name>S
   Lists the instances of the UDO module (only valid for module classes)
2. OMBRETRIEVE <class_name> 'name of instance'
   GET < child_class_name>S
   This returns a list of the instances of <child_class_name> within the named instance of <class_name>
3. OMBRETRIEVE <class_name> 'name of instance'
   <child_class_name> 'name of child instance'
   GET <grandchild_class_name>S
   This returns a list of the instances of <grandchild_class_name> within the

named instance of <child_class_name> which itself is within the named instance of <class_name>.

Note: You must be in the appropriate path (set by OMBCC) for this to work. Also, the **trailing 'S' is very important**.

Deleting UDO instances

You can delete UDO instances in two ways, graphically via the User-Defined Module nodes in the Project and Global Explorer and programmatically.

Programmatically

Deleting UDO instances is done as follows:

1. OMBDROP <class_name> 'name of instance'

Note: You must navigate to the appropriate path (using OMBCC) for this to work. Also, if the instance of the class has instances of child types, **these will all be dropped along with it**.

EXAMPLE – CREATE USER DEFINED PROPERTIES

In this example we'll be extending some of the OWB object types with some of our own properties, and then query them.

Note: Any object definition/alteration work requires ADMINISTRATOR privileges on the repository.

Property Definition

1. Login to the Design Center as an Administrator
2. Open the OMB*Plus client: Window → OMB*Plus

   Note that you are already connected to the repository - this client is sharing the Design Center's current connection so no OMBCONNECT is needed. You can test this by executing some commands. E.g. List all projects in the Repository:
3. OMBCC '/' (change context to the root or top level of the repository)
4. OMBLIST PROJECTS

In order to change the structure of the objects in the repository (or even add new object definitions), no one else must be connected to the repository. To enforce this we can prevent users logging into the repository.

This can be achieved in 2 ways: either when connecting to the repository (add the suffix USE SINGLE_USER_MODE to OMBCONNECT) or whilst connected (via OMBSWITCHMODE)

5. At the OMB*Plus prompt type:
   OMBSWITCHMODE SINGLE_USER_MODE

As to what we're trying to achieve (our business requirement): we'd like to store some extra information against view objects – specifically:

Property	User-friendly Name	Type	Default Value
OWNER	Object Owner	STRING	REP_OWNER
FILE		FILE	C:\vw.sql
LINK		URL	http://www.oracle.com
VERSION		DATE	7-Jan-2006

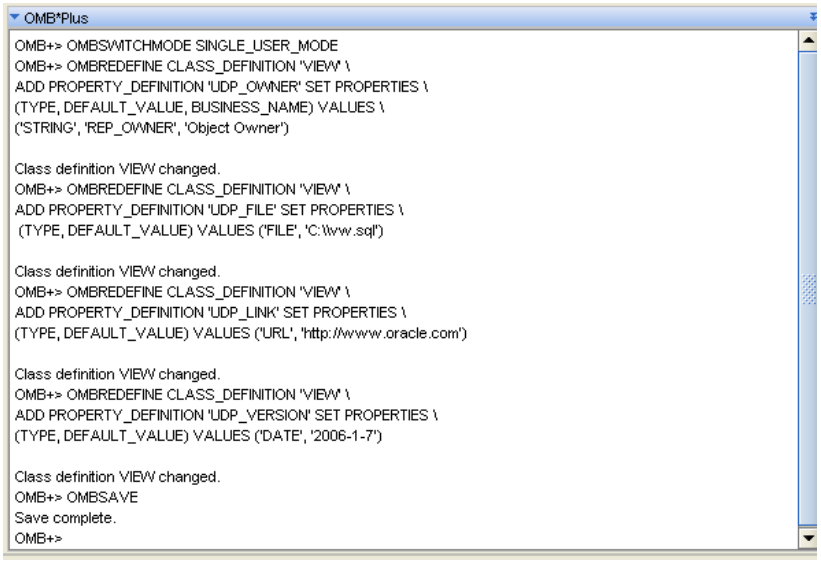
6. To define the properties, type the following (note: all user-defined properties **must** be prefixed with UD_ or UDP_):

```
OMBREDEFINE CLASS_DEFINITION 'VIEW' \
ADD PROPERTY_DEFINITION 'UDP_OWNER' SET
PROPERTIES \
(TYPE, DEFAULT_VALUE, BUSINESS_NAME) VALUES \
('STRING', 'REP_OWNER', 'Object Owner')

OMBREDEFINE CLASS_DEFINITION 'VIEW' \
ADD PROPERTY_DEFINITION 'UDP_FILE' SET PROPERTIES \
(TYPE, DEFAULT_VALUE) VALUES ('FILE', 'C:\\vw.sql')
```

Note the double-backslash in the filename – the first backslash is the escape character and forces Tcl to interpret the next character (backslash, which is normally a special character used in escape sequences) literally.

```
OMBREDEFINE CLASS_DEFINITION 'VIEW' \
ADD PROPERTY_DEFINITION 'UDP_LINK' SET PROPERTIES \
(TYPE, DEFAULT_VALUE) \
VALUES ('URL', 'http://www.oracle.com')

OMBREDEFINE CLASS_DEFINITION 'VIEW' \
ADD PROPERTY_DEFINITION 'UDP_VERSION' \
SET PROPERTIES \
(TYPE, DEFAULT_VALUE) VALUES ('DATE', '2006-1-7')
```

7. Make this permanent to the repository
   Type: OMBSAVE



Using the properties

Now create a view in a module in your repository (it doesn't matter which module since we won't be deploying this), and bring up the property inspector for it (right-mouse click → Properties).

Click on the User Defined tab. Then click on the name of the view in the left-hand panel:

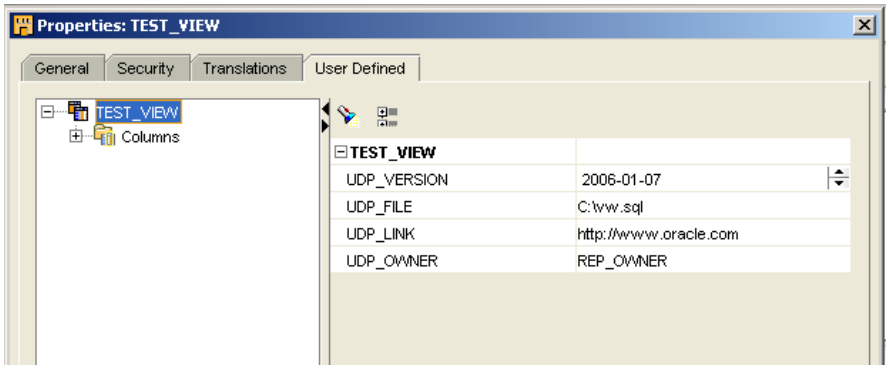


Figure 4 Properties on the view object with default values

Change the values of the properties. Notice that the type definition governs validation:

Change the values of different parts of UDP_VERSION using the arrows.

Click the ellipsis next to the UDP_FILE filename – a file selector dialog will appear allowing you to pick another file (directories are not valid).

In OMB*Plus, change context to the module containing the view (hint: Use 'OMBCC' to change. To see the current context, issue an 'OMBDCC')

Type the following to set the value of the UDP_VERSION property:
OMBALTER VIEW '<VIEW NAME>' \
SET PROPERTIES (UDP_VERSION) VALUES ('a')

The following error is returned: MMM1020: Wrong format a, format should be "YYYY-MM-DD"

The valid types for user defined properties are: INTEGER, STRING, FLOAT, DOUBLE, DATE, TIMESTAMP, BOOLEAN, LONG, FILE, URL

Deleting properties

Properties can also be deleted via the DELETE clause to the OMBREDEFINE command. In OMB*Plus issue the following:

1. OMBREDEFINE CLASS_DEFINITION 'VIEW' \
   DELETE PROPERTY_DEFINITION 'UDP_VERSION'
2. OMBSAVE

Verify that the property has been deleted in the property inspector.

Modifying Properties

The MODIFY clause allows you to alter the definition of the user defined property.

Note that you cannot change the type of the property once it is created - to change the type, you will have to delete it and recreate it.

In OMB*Plus issue the following:

1. OMBREDEFINE CLASS_DEFINITION 'VIEW' \
   MODIFY PROPERTY_DEFINITION 'UDP_OWNER' \
   SET PROPERTIES (BUSINESS_NAME) VALUES ('Owner')

Querying Properties

The OMBDESCRIBE command is used to query objects. This also applies to user defined properties.

1. Type the following in OMB*Plus:

   OMBDESCRIBE CLASS_DEFINITION 'VIEW' \
   PROPERTY_DEFINITION 'UDP_LINK' \
   GET PROPERTIES (TYPE, DEFAULT_VALUE)

   You should get the following:
   URL http://www.oracle.com

2. Type the following to find out all the properties on a view object:

   OMBDESCRIBE CLASS_DEFINITION 'VIEW' \
   GET PROPERTY_DEFINITIONS

   You should get the following:
   BUSINESS_NAME CONSTRAINTS DEPLOYABLE DESCRIPTION GENERATION_COMMENTS SHADOW_TABLESPACE SHADOW_TABLE_NAME UDP_FILE_UDP_LINK UDO_OWNER UOID

EXAMPLE – ICON SETS

Warehouse Builder has a set of objects defined in the repository (e.g. Views) and also icons to help users graphically distinguish different types of objects.

In this example we'll be extending the GUI for Warehouse Builder with some custom icons to be used when you create your own UDO.

Creating an Icon Set

Go to the Global Explorer.

1. Right click on the tree node that says 'Icon Sets' and pick 'New…'

   The following dialog appears:

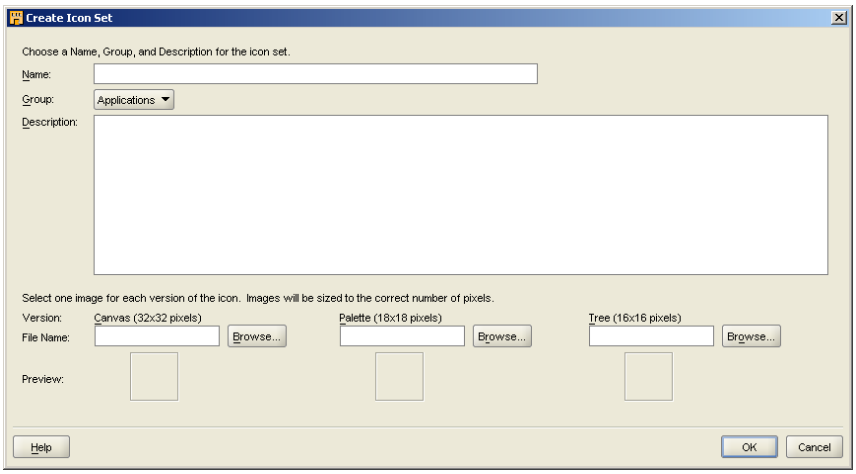


Figure 5 Creating an icon set from the UI

2. Fill in the following as an example:
   1. Specify MY_JAVA_ICONSET as the Name
   2. Specify 'Generic' for the group.
      This is a mechanism to help users find icon sets (useful when there are many).
   3. Optionally enter a description.
   4. Now specify the full location for each of the icons (or, easier, use the 'Browse..' button to locate them). If you have no icons, you can find some within the Microsoft windows product that could act as an icon.

      Notice how a preview is shown (if there is no image visible, that format is not compatible with OWB – refer to the concepts section on what is expected).

5. Click OK.

You have now defined an Icon Set! Make sure to save!

Using an Icon Set

The icon sets are located in the global explorer. The following shows some icon sets that are designed in the above way. These are merely shown as an example.

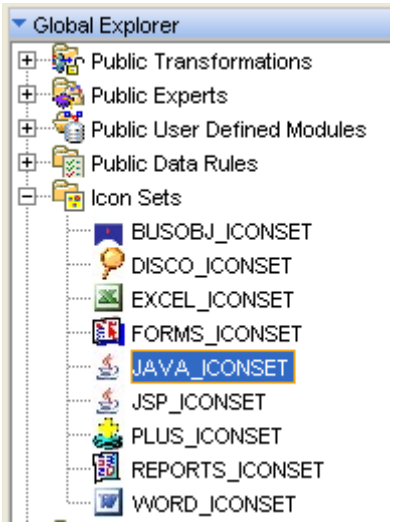


Figure 6 An example of a set of icon sets

1. Now go back to the Project Explorer window, select the View you were previously looking at, and bring up the Property Inspector for it.
2. On the General tab, look at the last property: **Icon Object**.

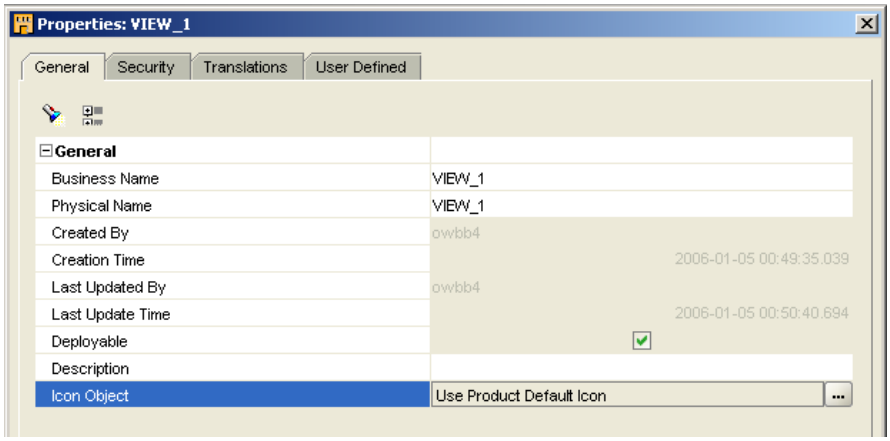


Figure 7 Icons in the property of an object

3. Click in the field (where it says 'Use Product Default Icon') and notice an ellipsis appears.

4. Click the ellipsis. An icon selector dialog appears:

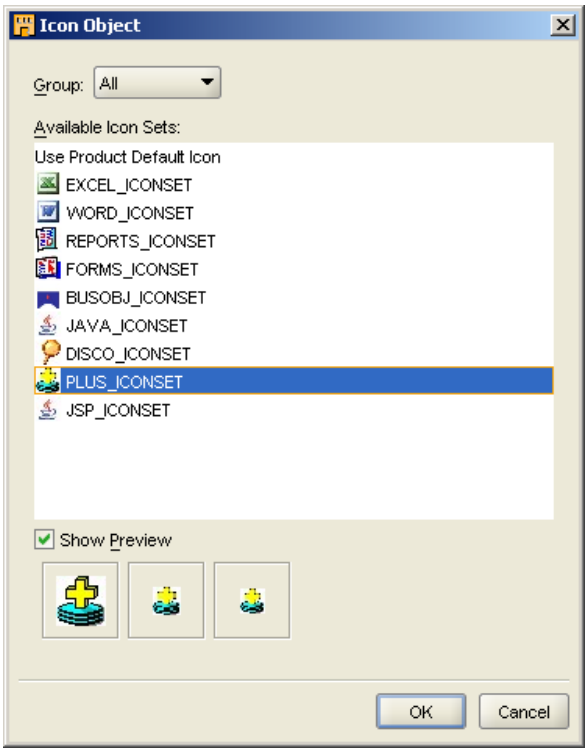


Figure 8 Example of icons shown

5. Pick an icon set and click OK.
   The Group field is a filtering mechanism to help find related icon sets. The group is specified at icon set creation time.

Go back and look at the View object in the Project Explorer. Notice how the icon has changed. If this View were to be used in a mapping now, the icon for that object would be different inside the mapping also.

EXAMPLE – USER DEFINED OBJECTS

We've now extended the base objects in the OWB repository. In this exercise we'll add new object types to the repository.

We will be modeling a Java application. These contain classes, and those classes contain methods. Within a method, we will model the lines of code. From a business standpoint, this is of interest since a particular line of code in an application may be impacted by a change in a database table if it is used within a SQL (JDBC) statement.

The model

Our object design looks as follows (the user-defined properties we'll be adding have also been included in the diagram):

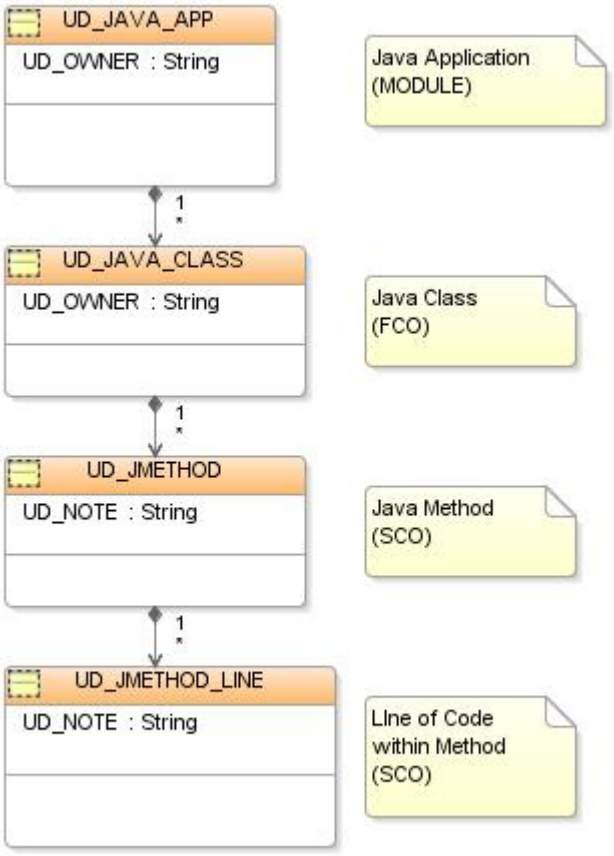


Figure 9 Model of our example

Once modeled, we will create the instances ourselves. However, it's possible, for example, to write 'crawler' programs using the OWB API's which would automatically create/update the instance information.

Note: It is *highly* recommended that whilst going through this section, you refer back to the concepts section to understand the general procedure.

Object Definition

To define the example UDO we will follow the same steps as are outlined in the previous sections.

Folder Definition

First, we need to define the module class and the folder that this is based on that will contain all the instances of the FCO's (the Java classes).

1. In OMB*Plus type: OMBDISPLAYCURRENTMODE and verify you are in SINGLE_USER_MODE (switch to it if not).
2. Define the module class as follows:

   ```
   OMBDEFINE MODULE CLASS_DEFINITION 'UD_JAVA_APP' \
   SET PROPERTIES (BUSINESS_NAME, PLURAL_NAME) \
   VALUES ('Java Application', 'Java Applications')
   ```

   This defines the Class and at the same time also sets some properties (these are properties that all OWB objects have).
   BUSINESS_NAME is the user-friendly name for an object – if the Naming mode preference for the Design Center is switched into Business mode (Tools → Preferences and then scroll down), then the BUSINESS_NAME is the name displayed for the object.
   The PLURAL_NAME is the label that is used to show where multiple instances of these objects are shown – for example, the label used for a tree node in the Design Center that contains several instances of the object.
3. Add a property to the class to record the owner (or author) of the application:

   ```
   OMBREDEFINE CLASS_DEFINITION 'UD_JAVA_APP' \
   ADD PROPERTY_DEFINITION 'UD_OWNER' \
   SET PROPERTIES (TYPE, DEFAULT_VALUE_STRING) \
   VALUES ('STRING', 'Unknown.')
   ```
4. Now have this module class assume the role of a Folder by creating a folder of exactly the same name:

   ```
   OMBDEFINE FOLDER_DEFINITION 'UD_JAVA_APP'
   ```

We have now defined our module (folder) that will contain the instances of the objects.

First Class Object Definition

We now need to define the FCO that will go into the Folder.

1. Enter the following to define the class:

   ```
   OMBDEFINE FIRST_CLASS_OBJECT CLASS_DEFINITION \
   'UD_JCLASS' SET PROPERTIES \
   (BUSINESS_NAME, PLURAL_NAME) \
   VALUES ('Java Class File', 'Java Class Files')
   ```

2. Now we add this class as a child of to the folder class:

   ```
   OMBREDEFINE CLASS_DEFINITION 'UD_JAVA_APP' \
   ADD CHILD_TYPE 'UD_JCLASS'
   ```

3. Add a property that will allow users to specify the owner/author of the class:

   ```
   OMBREDEFINE CLASS_DEFINITION 'UD_JCLASS' \
   ADD PROPERTY_DEFINITION 'UD_OWNER' \
   SET PROPERTIES (TYPE, DEFAULT_VALUE_STRING) \
   VALUES ('FILE', '')
   ```

Component Definition

As mentioned in the Concepts section, any FCO that is the parent (directly or indirectly) of one or more SCO's needs to be a component definition.

1. Have the FCO class assume the role of a Component definition by creating a component definition of the same name:

   ```
   OMBDEFINE COMPONENT_DEFINITION 'UD_JCLASS'
   ```

2. Now add the component definition to the folder definition:

   ```
   OMBREDEFINE FOLDER_DEFINITION 'UD_JAVA_APP' \
   ADD 'UD_JCLASS'
   ```

We now have our FCO defined. Referring back to the diagram, we have 2 more objects to go, plus then we need to create associations.

Second Class Object Definition

We are now adding second class objects to the already defined stack of objects.

1. To define the first of the SCO's, enter the following:

   ```
   OMBDEFINE SECOND_CLASS_OBJECT \
   CLASS_DEFINITION 'UD_JMETHOD' \
   SET PROPERTIES (BUSINESS_NAME, PLURAL_NAME) \
   ```

```
VALUES ('Method', 'Methods')
```

Once again, we are defining a class and simultaneously setting some properties.

Recalling the note in the Concepts section, it is **extremely** important that any SCO definitions be added to the Component definition.

2. Now add this SCO as a child of our FCO:

```
OMBREDEFINE CLASS_DEFINITION 'UD_JCLASS' \
ADD CHILD_TYPE 'UD_JMETHOD'
```

3. Add a property that will allow users to specify some notes against the object:

```
OMBREDEFINE CLASS_DEFINITION 'UD_JMETHOD' \
ADD PROPERTY_DEFINITION 'UD_NOTE' \
SET PROPERTIES (TYPE, DEFAULT_VALUE_STRING) \
VALUES ('STRING', '')
```

Now we have dealt with the class model, we need deal with the component model, and add this class to it.

4. To add this SCO to the Component definition, type the following:

```
OMBREDEFINE COMPONENT_DEFINITION 'UD_JCLASS' \
ADD 'UD_JMETHOD'
```

We are now on our last object definition! It is an SCO owned by an SCO.

5. Define the SCO as follows:

```
OMBDEFINE SECOND_CLASS_OBJECT \
CLASS_DEFINITION 'UD_JMETHOD_LINE' \
SET PROPERTIES (BUSINESS_NAME, PLURAL_NAME) \
VALUES ('Java Method Line', 'Java Method Lines')
```

6. Now add this SCO as a child of the SCO we previously defined:

```
OMBREDEFINE CLASS_DEFINITION 'UD_JMETHOD' \
ADD CHILD_TYPE 'UD_JMETHOD_LINE'
```

7. Add a property that will allow users to specify some notes against the object:

```
OMBREDEFINE CLASS_DEFINITION 'UD_JMETHOD_LINE' \
ADD PROPERTY_DEFINITION 'UD_NOTE' \
SET PROPERTIES (TYPE, DEFAULT_VALUE_STRING) \
VALUES ('STRING', 'No Note.')
```

8. Finally, add this SCO to the Component definition:

```
OMBREDEFINE COMPONENT_DEFINITION 'UD_JCLASS' \
ADD 'UD_JMETHOD_LINE'
```

This completes the object definitions. The next step is to define the associations.

Association Definition

A very useful aspect of UDO's is the fact that they may be related to other objects. By forming these relationships, users can encounter UDO's just like any other object when doing Lineage and Impact Analysis. We will be creating 2 associations at different levels.

We'd like to associate a Java Method with one or more Tables. As mentioned before, this is modeling the fact that a Method could be referencing Table(s) in JDBC calls. So it is useful to store this relationship.

1. This is done as follows (enter the following):

```
OMBDEFINE ASSOCIATION_DEFINITION
'UD_XJMETHOD2TABLE' \
SET PROPERTIES (CLASS_1,CLASS_2,ROLE_1,ROLE_2    \
,ROLE_1_MAX_CARDINALITY,ROLE_1_NAVIGABLE) \
VALUES ('UD_JMETHOD','TABLE','TABLEUSED','JM2TABLE' \
,'INFINITE','true') ADD \
DEPENDENCY_DEFINITION 'DATAFLOW'
```

CLASS_1 and CLASS_2 can be any classes (FCO or SCO). At least one of them should be a user-defined class (doesn't matter which one). The other one can be either a user-defined class or one of the main OWB classes (e.g. TABLE, COLUMN, etc.). In this example, the association is between the UDO UD_JMETHOD and the OWB type TABLE.

Role_1 and Role_2 are the names you use to identify class_1 from the point of view of this association. A class may have multiple associations and it plays a role in each one of them.

Note: The value of Role_1 (in this case 'TABLEUSED') is displayed prominently when creating instances of these objects later in scripting and in the Design Center, so take care in choosing a meaningful name!

In addition to the main Table-Java Method relationship, we'd like to store the fact that that particular column is referenced by a particular line within the Method.

2. Define this relationship as follows:

```
OMBDEFINE ASSOCIATION_DEFINITION \
'UD_XJMETHODLINE2COLUMN' \
```

```
SET PROPERTIES (CLASS_1,CLASS_2 \
,ROLE_1,ROLE_2 ,ROLE_1_MAX_CARDINALITY,\
ROLE_1_NAVIGABLE) VALUES \
('UD_JMETHOD_LINE','COLUMN' \
,'COLUMNUSED','JML2COLUMN' ,'INFINITE','true') \
ADD DEPENDENCY_DEFINITION 'DATAFLOW'
```

This concludes or association definition.

Using Icons

As a last step we can (optionally) add an icon to graphically identify the UDO we just created.

In a previous example you created an icon set, MY_JAVA_ICONSET.

1. Enter the following to associate it with all instances of the FCO you just defined:

   ```
   OMBREDEFINE CLASS_DEFINITION 'UD_JCLASS' \
   SET REF ICONSET 'MY_JAVA_ICONSET'
   ```

   For reference, you can delete/undo this by issuing the same

   OMBREDEFINE but use the 'UNSET REF ICONSET' syntax instead.

Also note: you don't have to associate an icon at class level (i.e. with all instances of an object) – icons can be changed for a particular instance of an object (once defined): OMBCREATE <udo_type> 'name' SET ICONSET '<name of iconset>'

You have now defined a User-Defined Object, association, and associated it with your own icon set. The following example allows you see how you can use the UDO.

USING USER DEFINED OBJECTS

There are two methods of using UDO's. Graphically (in the Project and Global Explorers) or via scripting – and these can be done (just like any other OWB object) in multi-user mode. We'll look at both.

Before we start, in order to be able to use your UDOs you will need to create project and a module. The module should have some tables imported. For example you can import metadata from the OE sample schema in the Oracle database, make sure to import the Customers table. In this cookbook we use a schema/module called Xweek and a table called Customers, all in a project called BI_DEMO.

If you use one of your own projects, simply substitute the names below with your project, module, table and column names.

Within the User Interface

1. Expand the your project in the Project Explorer.
2. Expand the node 'User Defined Modules'
3. You should see 'Java Applications' listed. Right-click on that and pick 'New..'
4. Specify a name for the Application (e.g. MY_APP1)
5. Now you'll see another node called 'Java Class Files' appear (this text is the value of the PLURAL_NAME specified earlier). Right-click on that and pick 'New…'.
6. Specify the name for the Java Class (e.g. CustomerUpdate)

Figure 10 Creating via the UI

7. Now right-click on that object you just created, and pick 'Open Editor'. The generic UDO editor is displayed.
8. Click on the 'Object and Association Tree' tab.
9. Highlight CUSTOMERUPDATE. You are now looking at the FCO UD_JCLASS (see title bar of the editor) and the properties and what they're set to.

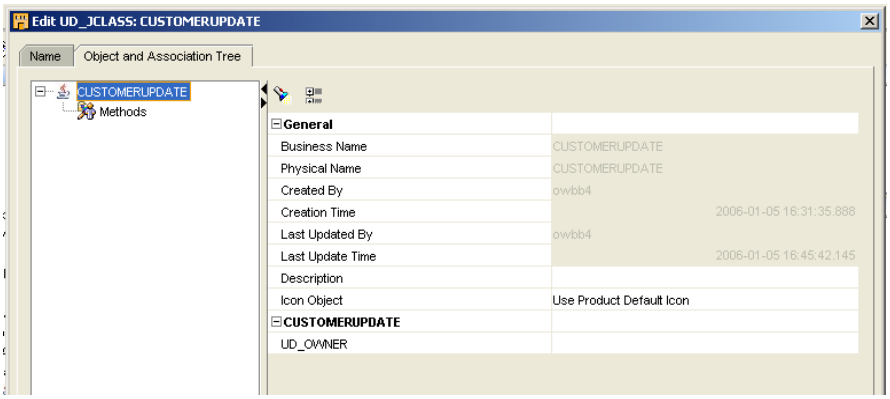


Figure 11 Editing from with the UI

10. Right-click on Methods, and click 'Create'.
11. Expand the node and you'll see an object (SCO) called JMETHOD_1 has been created (you can rename this if you wish).
12. Expand JMETHOD_1 and you'll see two nodes: Java Method Lines (which is another SCO you defined earlier) and TABLEUSED – this is the value specified for ROLE_1 when the association 'UD_XJMETHOD2TABLE' was created.
13. Right-click TABLEUSED and select Reference. The object selector dialog appears, and allows you to graphically pick the object (Table in this case) to relate/reference this object to.

Note that only objects of the type specified when creating the association are valid (i.e. you can navigate around, but the OK button is only available when the object of the correct type is highlighted).

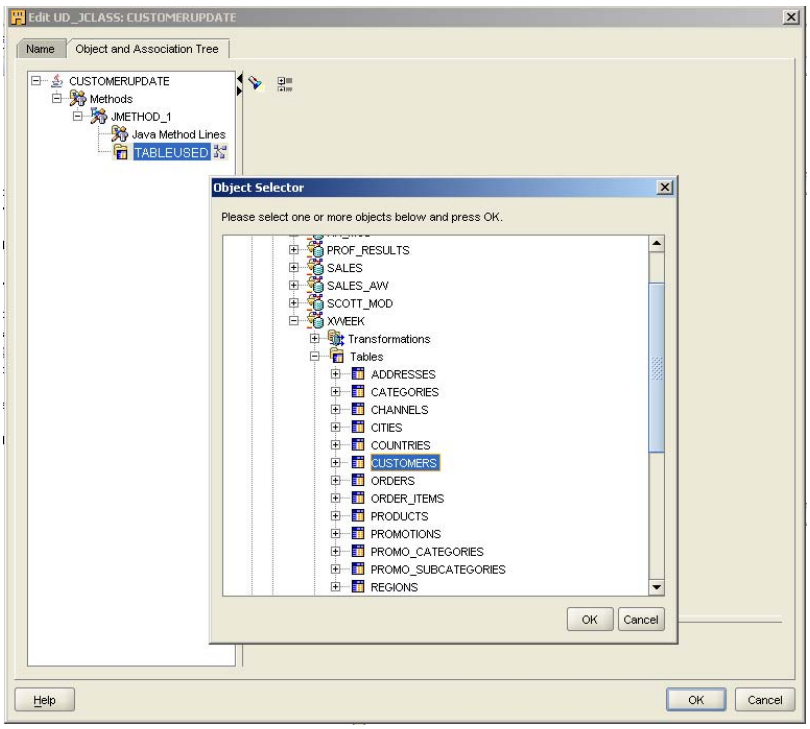


Figure 12 Choosing the table

14. Pick a Table (e.g. CUSTOMERS in the XWEEK Module) and click OK.
15. Now right-click 'Java Method Lines' and select Create.
16. A default object (JMETH_1) is created. Expand the node notice that it has one child node – another reference (COLUMNUSED) – this is the name allocated to ROLE_1 in the association 'UD_XJMETHODLINE2COLUMN'.
17. In a similar manner to the table, reference a column and click OK.
18. Right-click TABLEUSED association. Notice that you can remove (unreferenced) the association
19. Right-click JMETH_1 (the UD_JMETHOD_LINE). Notice that you can delete the object. This is true for all the UDO's you have created.

This completes the usages from the UI, but don't worry we'll go into the benefits a little later in an example!

Within the Scripting Language

You can also programmatically create UDO instances – this is extremely useful when building crawlers to automatically create/update instance/relationship information.

Remember that we used a project called BI_DEMO, a module called Xweek. We also used a different table, orders. You can import order from OE as well to get a similar effect.

Creating

1. In OMB*Plus, first ensure you are not locking the repository:

   OMBSWITCHMODE MULTIPLE_USER_MODE

2. Move to the root of the BI_DEMO project:

   OMBCC '/BI_DEMO'

3. Create a JAVA_APP UDO module

   OMBCREATE UD_JAVA_APP 'MY_APP2'

4. Move into the new module

   OMBCC 'MY_APP2'

5. Create an instance of the Java Class:

   OMBCREATE UD_JCLASS 'CUSTOMERDELETE'

6. Add an instance of a Java Method to the Java Class, set a property on the Method and also add a reference to a table in the XWEEK module.

   Note that the name used in the reference - 'TABLEUSED' – is the value specified for ROLE_1 when the association between UD_JMETHOD AND TABLES (UD_XJMETHOD2TABLE) was defined:

   ```
   OMBALTER UD_JCLASS 'CUSTOMERDELETE' \
   ADD UD_JMETHOD 'JMETHOD_2' \
   SET PROPERTIES (BUSINESS_NAME) \
   VALUES ('Method2') \
   SET REF TABLEUSED TABLE '/BI_DEMO/XWEEK/ORDERS'
   ```

7. Add an instance of an SCO to the parent SCO (notice the 'OF' syntax)

   ```
   OMBALTER UD_JCLASS 'CUSTOMERDELETE' \
   ADD UD_JMETHOD_LINE 'JMETH_2' \
   OF UD_JMETHOD 'JMETHOD_2'
   ```

8. Add a reference from the (lowest level) SCO to a column in a table

   ```
   OMBALTER UD_JCLASS 'CUSTOMERDELETE' \
   MODIFY UD_JMETHOD_LINE 'JMETH_2' \
   ```

```
OF UD_JMETHOD 'JMETHOD_2' \
SET REF COLUMNUSED COLUMN 'CUSTOMER_ID' \
OF TABLE '/BI_DEMO/XWEEK/ORDERS'
```

9. Now look in the Design Center in the BI_DEMO project under the User Defined Modules node. You should see the UDO instance(s) just created.

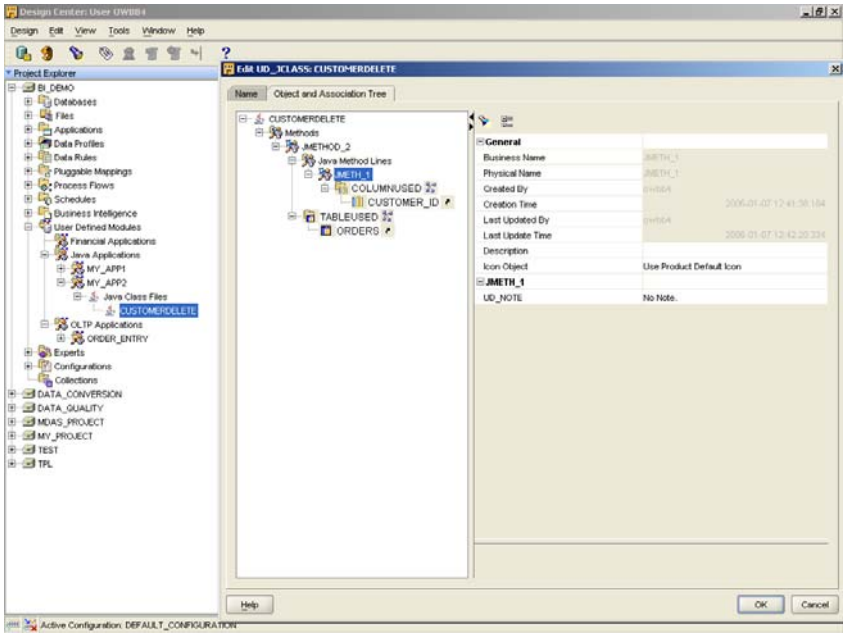


Figure 13 Results in the UI

Querying

Also useful is the ability to programmatically query information about UDO's. For example, this could be combined with the creation instances we did above to ensure duplicates are not created.

1. In OMB*Plus, change to the root of the BI_DEMO project:

   ```
   OMBCC '/BI_DEMO'
   ```

2. Now list all the instances of our Java Application UDO created:

   ```
   OMBLIST UD_JAVA_APPS
   ```

3. Change location into an instance (MY_APP1) of the UDO

   ```
   OMBCC '/BI_DEMO/MY_APP1'
   ```

4. List all instances of Java Classes within that Java Application:

   ```
   OMBLIST UD_JCLASSS
   ```

5. For a particular Java Class (in our example there is only one) get the value of the property 'OWNER'

   ```
   OMBRETRIEVE UD_JCLASS 'CUSTOMERUPDATE'\
   GET PROPERTIES (UD_OWNER)
   ```

6. Query the Java Class to find out what Java Methods it has:

   ```
   OMBRETRIEVE UD_JCLASS 'CUSTOMERUPDATE'\
   GET UD_JMETHODS
   ```

7. For a particular Java Method, find out how many lines are registered in the repository:

   ```
   llength [OMBRETRIEVE UD_JCLASS 'CUSTOMERUPDATE'\
   UD_JMETHOD 'JMETHOD_1'\
   GET UD_JMETHOD_LINES]
   ```

   Notice how in Tcl you pass the result of one command to another using the [] square braces. In this instance the command returns a list, and the Tcl command llength returns the number of elements in a list.

8. Find out the names of the instances of Java Method Lines:

   ```
   OMBRETRIEVE UD_JCLASS 'CUSTOMERUPDATE'\
   UD_JMETHOD 'JMETHOD_1'\
   GET UD_JMETHOD_LINES
   ```

9. Find out what the value of the Note user-defined property is set to for the Java Method 'JMETH_1' (which is located within the Java Class 'CUSTOMERUPDATE'):

   ```
   OMBRETRIEVE UD_JCLASS 'CUSTOMERUPDATE'\
   UD_JMETHOD 'JMETHOD_1'\
   UD_JMETHOD_LINE 'JMETH_1' \
   GET PROPERTIES (UD_NOTE)
   ```

   Notice the syntax requires that the objects in the hierarchy to be specified. This is because the furthest level that is navigable via OMBCC is the module (MY_APP1). So to reference objects further down, the full object path needs to be specified.

10. Find out what Column the Java Method Line 'JMETH_1' is linked to. To do this you need to specify the name of ROLE_1 used when the association (UD_XJMETHODLINE2COLUMN) was created:

    ```
    OMBRETRIEVE UD_JCLASS 'CUSTOMERUPDATE'\
    ```

```
UD_JMETHOD 'JMETHOD_1'\
UD_JMETHOD_LINE 'JMETH_1' \
GET REF COLUMNUSED
```

Notice how the return value: {COLUMN /BI_DEMO/XWEEK/CUSTOMERS/NAME} is surrounded by {} braces. This is because there could be multiple return values (i.e. multiple associations are allowed, so a list of lists is returned).

Results in Lineage

Once objects are defined in the OWB repository, they can take advantage of all the regular metadata services such as Import, Export, Snapshots, and (probably the most compelling reason to implement UDO's) Lineage and Impact Analysis (LIA).

The Lineage and Impact Analyser is not covered in detail here, but we will briefly look at UDO's within the LIA editor to give you an idea of how you can use UDOs (also see Figure 1) for another example.

Remember we are using a specific project here!

1. In the Project Explorer, navigate to /BI_DEMO/XWEEK/Customers table
2. Right-click CUSTOMERS and pick Impact…

This runs the Impact Analyser for the Customer table, and follows the relationships of objects in the repository related to the selected object.

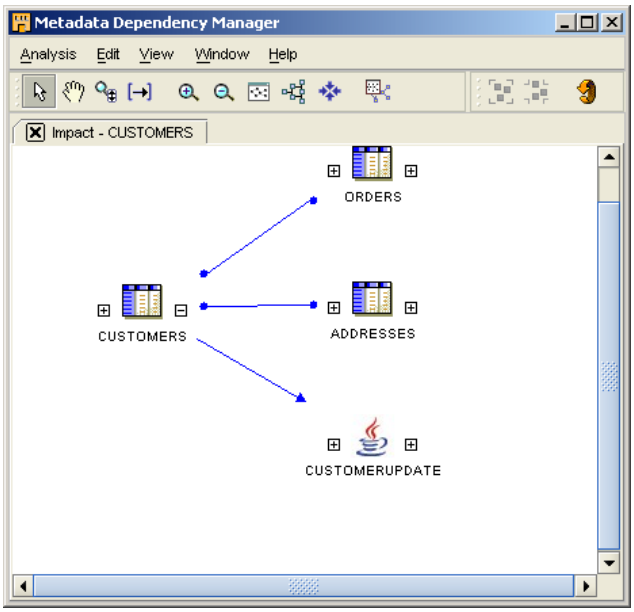


Figure 14 Lineage including UDOs

The resulting diagram shows that a change in the Customers table would impact the Orders table, the Addresses table, and the CustomerUpdate Java Class. If the Customer update class was related to other objects, then hitting the + on either side would traverse the paths and show associated objects.

In the Metadata Dependency Manager, pick View → Expand All to show the SCO's within each object.

Within the CUSTOMERUPDATE object, you can see the JMETHOD_1 and JMETH_1 objects. Right-click on JMETH_1 and pick Show Lineage

The resulting diagram shows highlights that the source of data within JMETH_1 is from the Name column in the Customers table (or, a change to the Name column in the Customers table would impact the JMETH_1 line of code within the method JMethod_1 within the class CUSTOMERUPDATE object.

Whilst our example is small, imagine a much bigger system/application, where UDO's are automatically populated by crawlers. Now users can not only understand the impact (cost) of change within their database, it can be extended to their entire system.

CONCLUSION

As you have seen, OWB 10gR2 has some very powerful object extensibility features – either extending base object types with extra information, adding icons, or defining entirely new object types and relating them to other objects in the OWB repository. When used in conjunction with the other metadata management features in the product, customers can derive real business benefits.

Oracle Warehouse Builder 10*g*R2 - Repository Extensibility Cookbook
May 2006
Author: Paul Narth,
Contributing: Jean-Pierre Dijcks

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com