SYBASE®

# Customizing and Extending PowerDesigner

# PowerDesigner® 15.1

Windows

DOCUMENT ID: DC38628-01-1510-01
LAST REVISED: June 2009

Sybase, Inc., One Sybase Drive, Dublin, CA 94568

# Contents

# Resource Files and the Public Metamodel

The PowerDesigner® modeling environment is powered by resource files, which define the objects available in each model along with the methods for generating and reverse-engineering them. These resource files are based upon the PowerDesigner public metamodel.

## Working with PowerDesigner Resource Files

You can view, copy, and edit the XML-format resource files in order to customize and extend the behavior of the environment.

The following types of resource files are provided:

- *Target languages*: define the standard objects available in a model. Types of target language files include:
  - *Process languages* (.xpl) – define a specific business process language in the BPM.
  - *Object languages* (.xol) - define a specific object-oriented language in the OOM.
  - *DBMSs* (.xdb) - define a specific DBMS in the PDM (see *DBMS Resource File Reference* on page 35).
  - *XML languages* (.xsl) - define a specific XML language definition in the XSM.
- *Extended model definitions* (.xem) – extend the standard definitions of target languages to, for example, specify a persistence framework or server in an OOM. You can create or attach one or more XEMs to a model (see *Extended Model Definitions* on page 18).
- *Report templates* (.rtp) - specify the structure of a report. Editable within the Report Template Editor (see the Reports chapter in the *Core Features Guide*).
- *Report language files* (.xrl) – translate the headings and other standard text in a report (see *Translating Reports with Report Language Resource Files* on page 221).
- *Conversion tables* (.csv) - define conversions between the name and code of an object (see "Using a conversion table" in the Models chapter of the *Core Features Guide*).

### Lists of Resource Files

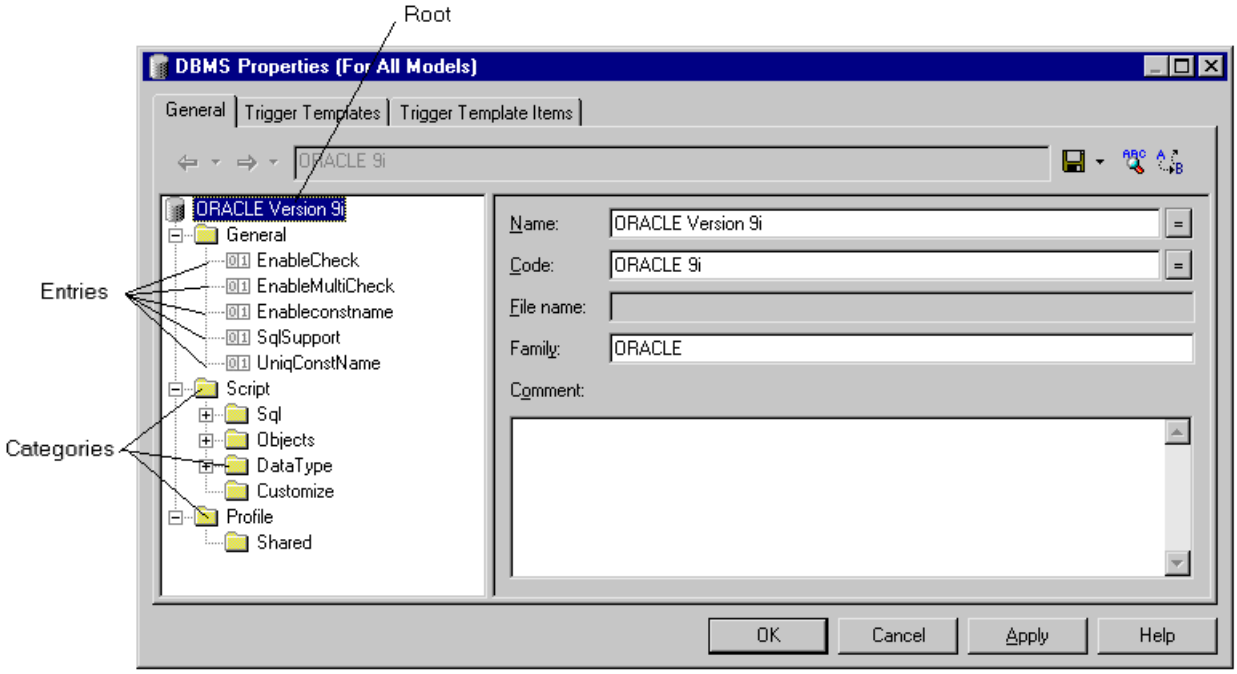
You can review all the available resource files from the lists of resource files. You access a resource file list by selecting **Tools** > **Resources** > ***Type.***

The following tools are available on each resource file list:

| Tool | Description |
|---|---|
|  | Properties - Opens the resource file in the Resource Editor. |
|  | New - Creates a new resource file using an existing file as a model (see *Copying resource files* on page 5). |
|  | Save - Saves the selected resource file. |
|  | Save All - Saves all the resource files in the list. |
|  | Path - Browses to the directory which contains the resource files. |
|  | Compare - Selects two resource files for comparison (see *Comparing resource files* on page 5). |
|  | Merge - Selects two resource files for merging (see *Merging resource files* on page 6). |

## Working with the Resource Editor

The resource editor allows you to view and edit the resources files provided with PowerDesigner. The left-hand pane shows a tree view of the entries contained within the resource file, and the right-hand pane displays the properties of the currently-selected element:




Each entry is a part of the definition of a resource file. For example, you can define entries for a database command, a characteristic of an object language, a report item, and so on.

Entries are organized into logical categories. For example, the Script category in a DBMS language file collects together all the entries relating to database generation and reverse engineering.

Each entry type is identified by a specific symbol in the resource editor tree.

You can drag and drop categories or entries in the tree view of the resource editor and also between two resource editors of the same type (for example two XOL editors).

**Note:** You should never modify the resource files shipped with PowerDesigner. To create your own resource file For each original resource file you want to modify you should create a corresponding new resource file. To do so you have to create a new resource file from the List of Resource Files, define a name and select the original resource file in the Copy From list. This allows you to create a new resource file that is identical to the original file apart from the name.

### Resource Editor Navigation Tools

You can perform the following navigation and saving tasks from the resource editor toolbar:

| Tool | Description |
|---|---|
|  | Back (alt+left) - Go to the previous visited entry or category. If you click the down arrowhead, you can directly select the previous visited entry or category to which you want to go back |
|  | Forward (alt+right) - Go to the next visited entry or category. If you click the down arrowhead, you can directly select the next visited entry or category to which you want to go forth |
|  | Lookup - Search target items by name |

| Tool | Description |
|---|---|
|  | Save (ctrl+shift+s) – Save the current resource file. If you click the down arrowhead, you can save the current resource file under a new name |
|  | Find In Items (ctrl+shift+f) - Search for text, command, template, custom check, criterion and generated file entries |
|  | Replace In Items (ctrl+shift+h) - Search for and replace text, command, template, custom check, criterion and generated file entries |

## Opening Resource Files

When working with a BPM, PDM, OOM, or XSM, you can open the target language file that defines your modeling environment in the Resource Editor for viewing and editing.

1. In a BPM, select **Language > Edit Current Process Language** .
2. In a PDM, select **Database > Edit Current DBMS** .
3. In an OOM, select **Language > Edit Current Object Language** .
4. In an XSM, select **Language > Edit Current Language** .

## To Open a Resource File from a Resource File List:

In addition, you can, at any time, open, inspect, and edit any resource file from the lists of resource files.

1. Select **Tools > Resources > *Type*** to open the relevant resource file list.
2. Select a file in the list, and then click the Properties tool.

### "Not Certified" Resource Files

Some resource files are delivered with "Not Certified" in their names. Sybase will perform all possible validation checks, however we do not maintain specific environments to fully certify these resource files. We will support them by accepting bug reports and providing fixes as per standard policy, with the exception that there will be no final environmental validation of the fix. You are invited to assist us by testing fixes and reporting any continuing inconsistencies.

### Sharing and Copying Resource Files

Some resource files can be shared among different models or copied in a local model. Modifications that you make to a resource file are applied differently depending on whether the resource file is shared or copied into the model:

- Share - Any modifications made to the resource file are shared by other models using this resource file
- Copy - The current resource file is independent of the original resource file so modifications made to the resource file in the resource files library are not available to the model. The copied resource file is saved with the model and cannot be used without it.

When you modify a shared resource file, you should always modify a new resource file created from the original resource files shipped with PowerDesigner.

The File Name box allows you to know where the resource file you are modifying is defined:


### Saving Changes

If you make changes to a resource file and then click OK to close the resource editor without having clicked the Save tool, the changes are saved in memory, the editor is closed and you return to the list of resource files. When you click Close in the list of resource files, a confirmation box is displayed asking you if you really want to save the modified resource file. If you click Yes, the changes are saved in the resource file itself. If you click No, the changes are kept in memory until you close the PowerDesigner session.

The next time you open any model that uses the customized resource file, the model will take modifications into account. However, if you have previously modified the same options directly in the model, the values in the resource file do not change these options.

## Editing Resource Files

When you right-click a category or an entry in the resource file tree view, the following editing options appear:

| Edit option | Description |
|---|---|
| New | Adds a user-defined entry or category . |
| Add items... | Opens a selection dialog box to allow you select one or more of the predefined metamodel categories or entries to add to the present node. You cannot edit the names of these items but you can change their comments and values by selecting their node. |
| Remove | Deletes the selected category or entry. |
| Restore Comment | Restores the default comment for the selected category or entry. |
| Restore value | Restores the default value for the selected entry. |

**Note:** You can rename a category or an entry directly from the resource file tree by selecting it and pressing the f2 key.

## Category and Entry Properties

Each category and entry you select in the resource editor tree view can display the following properties to the right hand side of the editor.

| Property | Description |
|---|---|
| Name | Name of category or entry |
| Comment | Description of selected category or entry |
| Value | Value of entry |

## Searching in Resource Files

You can use the navigation list in the upper part of the resource editor to search for target items. This box lets you type queries on target items, you can also use it to type the fully qualified path of the item you are looking for.

The lookup feature is triggered when you press enter or when you click the Lookup tool.

You can use options to fine-tune your queries. The Lookup options dialog box lets you define the following search options:

| Syntax | Description |
|---|---|
| Metaextension | You select the type of extension to search, for example you can search only stereotypes |
| Allow wildcard | If you select this option you can use the * wildcard to match any string and the ? wildcard to match any single character. For example, type "is*" to retrieve all extensions of type "is..." |
| Match case | Use this option to search for extensions with matching case |

If the query matches one item, this item is selected in the Resource editor.

If the query matches more than one item, a Results list is displayed. You can double-click an item in the list to visualize it in the Resource editor.

### Go to Super-definition

If an extension overrides another item you can use the Go to super-definition command in the corresponding object contextual menu to access the overriden item.

## Copying Resource Files

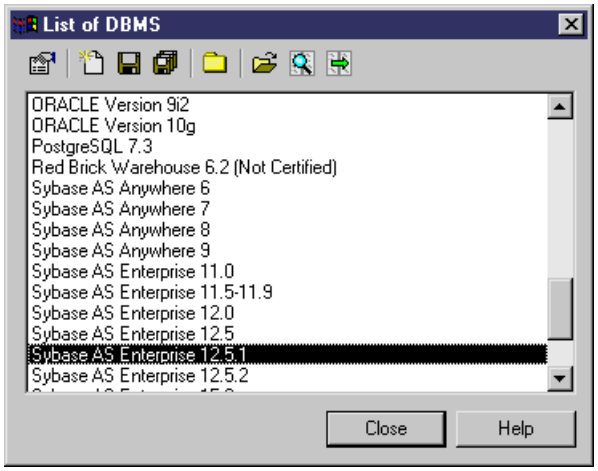
Since each resource file has a unique id, you should only copy resource files within PowerDesigner, and not in Windows Explorer.

1. Open the appropriate list of resource files (eg, select **Tools** > **Resources** > **Process languages** )
2. Click the New tool and enter a name for the new file that you will create and select the file from which you want to copy it.
3. Click OK to create the new resource file as a copy of the original.

## Comparing Resource Files

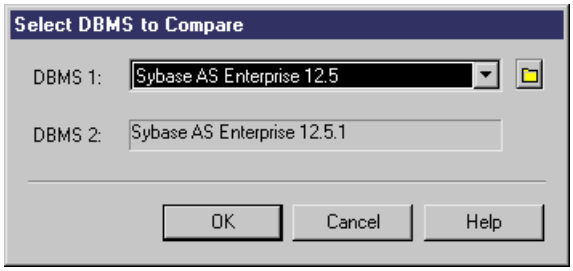
You can select two resource files and compare them.The comparison process allows you to highlight the differences between resource files.

1. Select **Tools** > **Resources** > ***Resource File*** to open the List of *Resource Files*.




2. Select a resource file in the list, and then click the Compare tool to open the Select *resource file* to Compare dialog box. The selected resource file is displayed in the second list in the lower part of the dialog.
3. Select a resource file from the first list.

   If the resource file you want to compare is not in the list, click the Select a Path tool and browse to its directory. Click OK and then select the resource file from the list.




4. Click OK to open the Compare *resource files* dialog box.

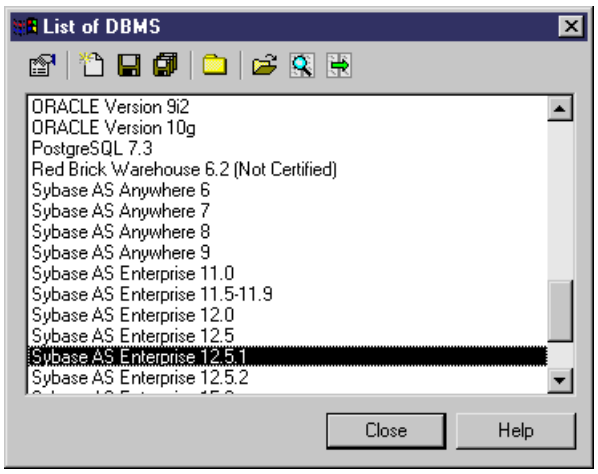
For more information on the comparison process, see the Comparing and Merging Models chapter in the *Core Features Guide*.

## Merging Resource Files

You can select two resource files and merge them. Merging means that you use information from a given resource file to modify another resource file.

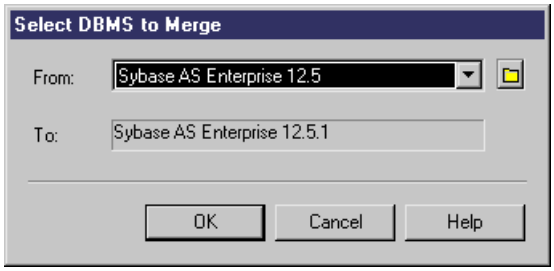
Merge is performed from left to right, the resource file in the right pane is compared to the resource file in the left pane, differences are highlighted and merge actions are proposed in resource file to be merged.

1. Select **Tools** > **Resources** > ***Resource File*** to open the List of *Resource File*s.




2. Select a resource file in the list, and then click the Merge tool to open the Select *resource file* to Merge dialog box. The selected resource file is displayed in the To list in the lower part of the dialog, and will be displayed in the right pane of the Merge dialog box, so that merge actions will be applied to it.
3. Select a resource file from the From list. This file that will appear in the left pane of the Merge dialog box.

   If the resource file you want to compare is not in the list, click the Select a Path tool and browse to its directory. Click OK and then select the resource file from the list.




4. Click OK to open the Merge *resource files* dialog box, in which you can merge the selected *resource files*..

For more information on the merge process, see the Comparing and Merging Models chapter in the *Core Features Guide*.

# Resource File Reference

The following model types use standard PowerDesigner resource files, which you can review and edit in the Resource Editor.

For more information, see *Opening Resource files* on page 3):

- OOM - Object language resource files (.xol)
- BPM – Business process language resource files (.xpl)

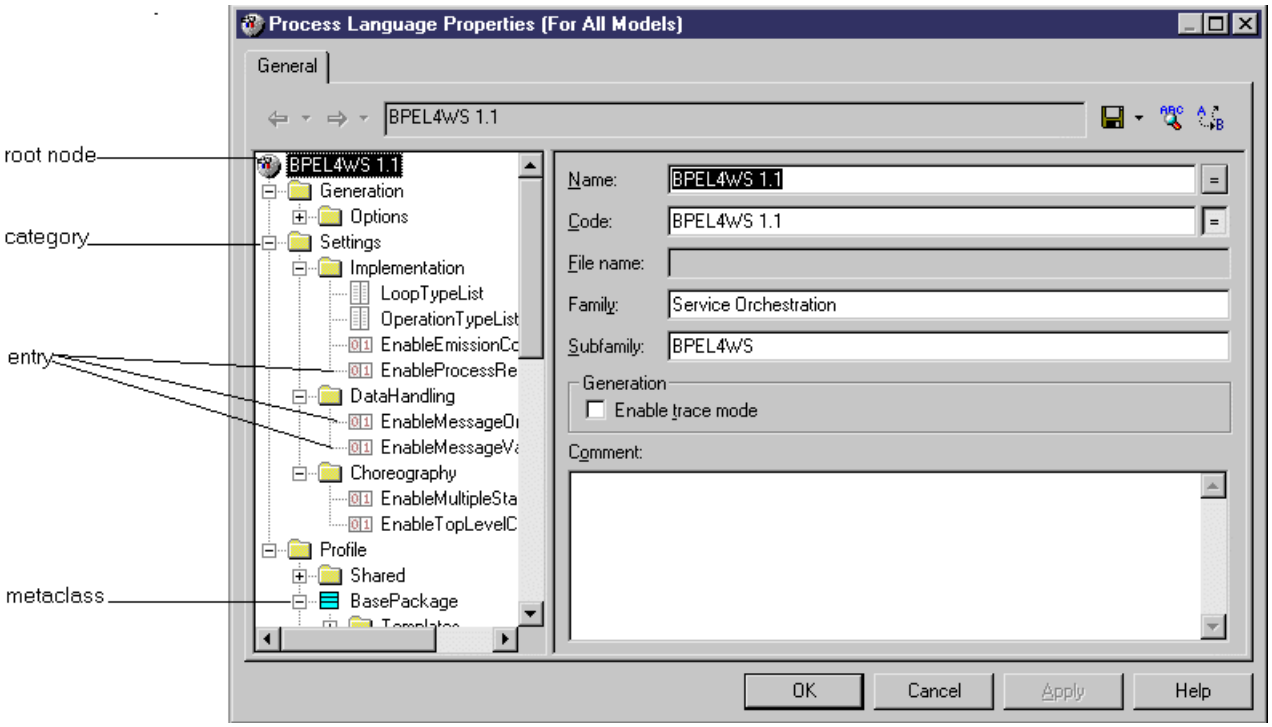
- XSM - XML language resource files (.xsl)

The PDM uses a different form of resource file. For more information, see the DBMS Resource File Reference. The resource files for the RQM, CDM, LDM, and ILM are not accessible.

Resource files extend the PowerDesigner metamodel to model the specific features of a particular language. A separate resource file is supplied for each language supported. Each resource file defines the syntax and guidelines for generating objects and implementing stereotypes, data types, scripts and constants for the language. The appropriate resource file is automatically selected when you create a new model.

## Resource File Properties

All target languages have the same basic category structure, but the detail and values of entries differs for each language. The root node of each file contains the following properties:




| Property | Description |
|---|---|
| Name | Specifies the name of the target language. |
| Code | Specifies the code of the target language. |
| File Name | [read-only] Specifies the path to the .xol, xpl, or .xsl file. If the target language has been copied to your model, this field is empty. |
| Version | [read-only] Specifies the repository version if the resource is shared via the repository. |
| Family | Enables certain non-default features in the model. For example, object languages of the Java, XML, IDL and PowerBuilder® families support reverse engineering. |
| Subfamily | Fine-tunes the features for a given family. For example, in the Java family, the J2EE subfamily supports EJBs, servlets and JSPs. |

| Property | Description |
|---|---|
| Enable Trace Mode | Lets you preview the templates used during generation. Before starting the generation, click the Preview page of the relevant object, and hit the Refresh tool to display these templates.<br>When you double-click on a trace line from the Preview page, the Resource Editor opens to the corresponding template definition in the Profile\Object\Templates category. The code of the template may be colored (see Syntactic coloring in section *Generated Files category* on page 14). |
| Comment | Specifies additional information about the target language. |

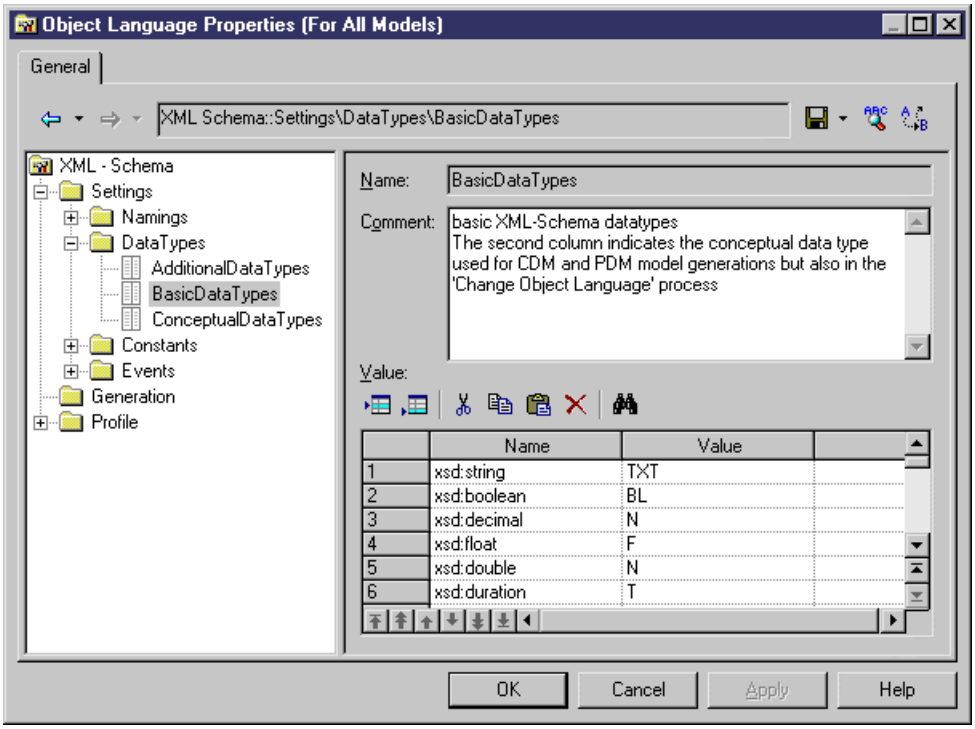
## Settings Category

The Settings category contains data types, constants, namings, and events categories used to customize and manage generation features. The types of items in this category differ depending on the type of resource file.

### Settings Category: Object Language

The Settings category contains the following items used to control the data types, constants, namings, and events categories used to customize and manage OOM generation features:

- *Data Types* - Tables for mapping internal data types with object language data types. The following data types values are defined by default:
  - *BasicDataTypes* – lists the most commonly-used data types. The Value column indicates the conceptual data type used for CDM and PDM model generations.
  - *ConceptualDataTypes* – lists internal PowerDesigner data types. The Value column indicates the object language data type used for CDM and PDM model generations.
  - *AdditionalDataTypes* – lists additional data types added to data type lists. Can be used to add or change data types of your own. The Value column indicates the conceptual data type used for CDM and PDM model generations.
  - *DefaultDataType* – specifies the default data type.




- *Constants* - contains mapping between the following constants and their default values: Null, True, False, Void, Bool.
- *Namings* - contains parameters that influence what will be included in the files that you generate from an OOM:

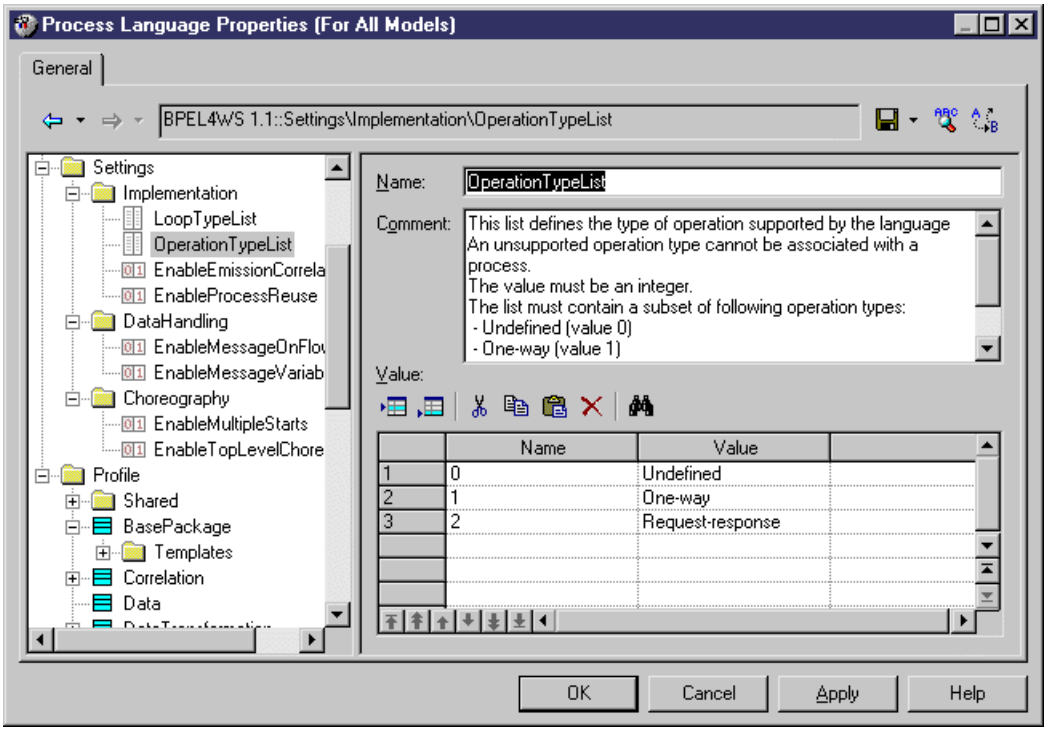
- *GetterName* - Name and value for getter operations
- *GetterCode* - Code and value for getter operations
- *SetterName* - Name and value for setter operations
- *SetterCode* - Code and value for setter operations
- *IllegalChar* - lists illegal characters for the object language. This list populates the Invalid characters field in **Tools** > **Model Options** > **Naming Convention** . For example, `"/!=<>"" ' ( )"`

- *Events* - defines standard events on operations. This category may contain default existing events such as constructors and destructors, depending on the object language. An event is linked to an operation, and the contents of the Events category is displayed in the Event list in operation property sheets to describe the events that can be used by an operation. In PowerBuilder for example, the Events category is used to associate operations with PowerBuilder events.

### Settings Category: Process Language

The Settings category contains the following items used to control the data types, constants, namings, and events categories used to customize and manage BPM generation features:

- *Implementation* – [executable BPM only] Gathers options that influence the process implementation possibilities. The following constants are defined by default:
  - *LoopTypeList* - This list defines the type of loop supported by the language. The value must be an integer
  - *OperationTypeList* - This list defines the type of operation supported by the language. An unsupported operation type cannot be associated with a process. The value must be an integer
  - *EnableEmissionCorrelation* - enables the definition of a correlation for an emitted message
  - *EnableProcessReuse* - allows a process to be implemented by another process
  - *AutomaticInvokeMode* - indicates if the action type of a process implemented by an operation can be automatically deducted from the operation type. You can specify:
    - 0 (default) - the action type cannot be deduced and must be specified
    - 1 - the language enforces a Request-Response and a One-Way operation to be received by the process and a Solicit-Response and a Notification operation to be invoked by the process
    - 2 the language ensures that a Solicit-Response and a Notification operation are always received by the process while Request-Response and One-Way operations are always invoked by the process




- *DataHandling* - [executable BPM only] Gathers options for managing data in the language. The following constant values are defined by default:
  - *EnableMessageOnFlow* - indicates if a message format can be associated to a flow or not. The default value is Yes
  - *EnableMessageVariable* - enables a variable object to store the whole content of a message format. In this case, the message format objects will appear in the data type combo box of the variable
- *Choreography* - Gathers objects that allow the design of the graph of activities (start, end, decision, synchronization, transition...) Contains the following constant values defined by default:
  - *EnableMultipleStarts* - When set to No, ensures that no more than one start is defined under a composite process
  - *EnableTopLevelChoreography* - When set to No, ensures that no flow or choreography object (start, end, decision...) is defined directly under the model or a package. These objects can be defined only under a composite process

### Settings Category: XML Language

The Settings category contains the Data types category that shows a mapping of internal data types with XML language data types.

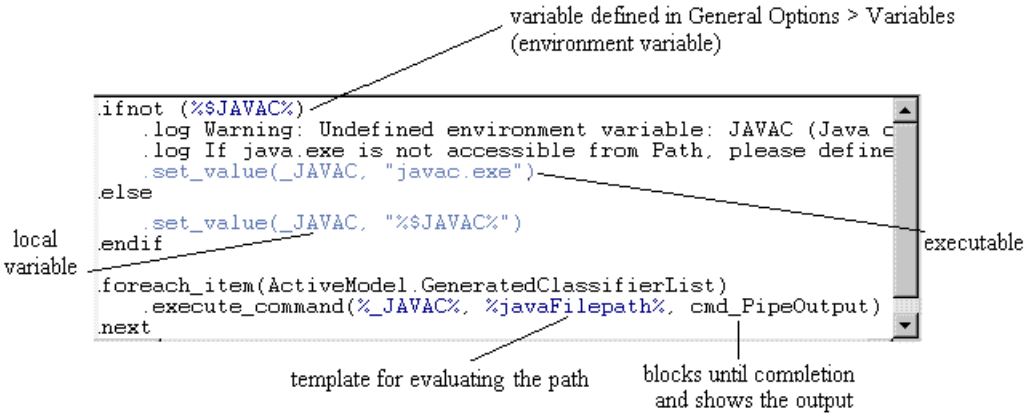
The following data types values are defined by default:

- ConceptualDataTypes - The Value column indicates the XML language data type used for model generations. Conceptual data types are the internal data types of PowerDesigner, and cannot be modified
- XsmDataTypes- Data types for generations from the XML model

## Generation Category

The Generation category contains categories and entries to define and activate a generation process:

- *Commands* - contains generation commands, which can be executed at the end of the generation process, after the generation of all files. Commands are written in GTL (see *Customizing Generation with GTL* on page 187), and must be included within tasks (see below) to be evoked.




- *Options* – contains options, available on the Options tab of the Generation dialog box, the values of which can be tested by generation templates or commands. You can create options that take boolean, string, or list values. The value of an option may be accessed in a template using the following syntax:

```
'%' 'GenOptions.'<option-name> '%'
```

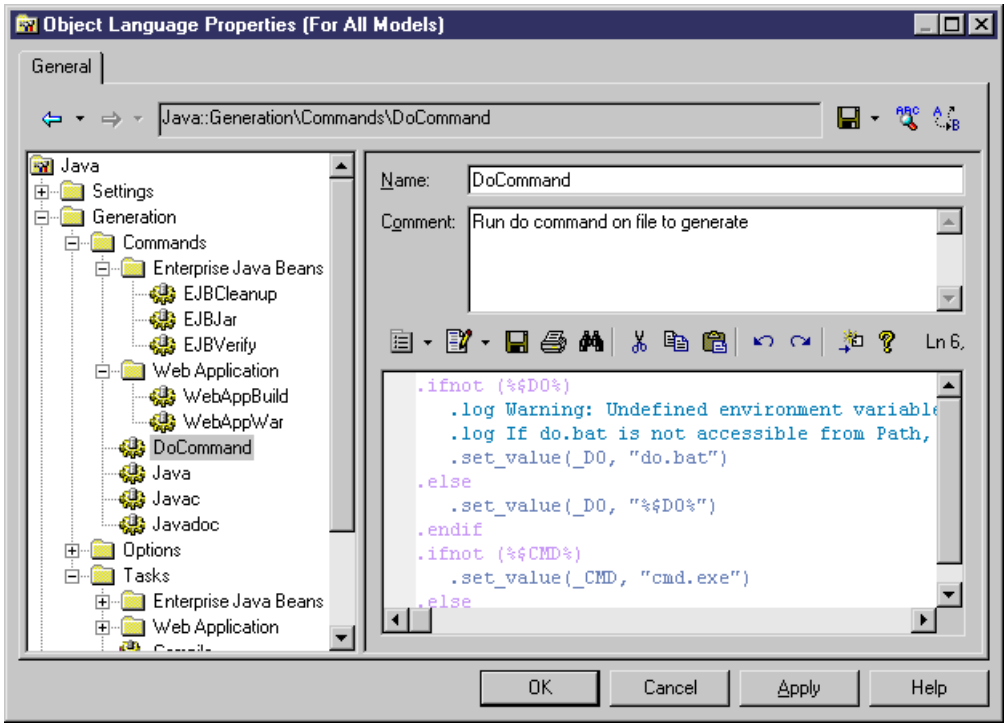
For example, for a boolean option named GenerateComment, %GenOptions.GenerateComment% will evaluate to either true or false in a template, depending on the value specified in the Generation dialog Options tab.

- *Tasks* – contains tasks, available on the Tasks tab of the Generation dialog box, and which contain lists of generation commands (see above). When a task is selected in the Tasks tab, the commands included in the task are retrieved and their templates evaluated and executed.

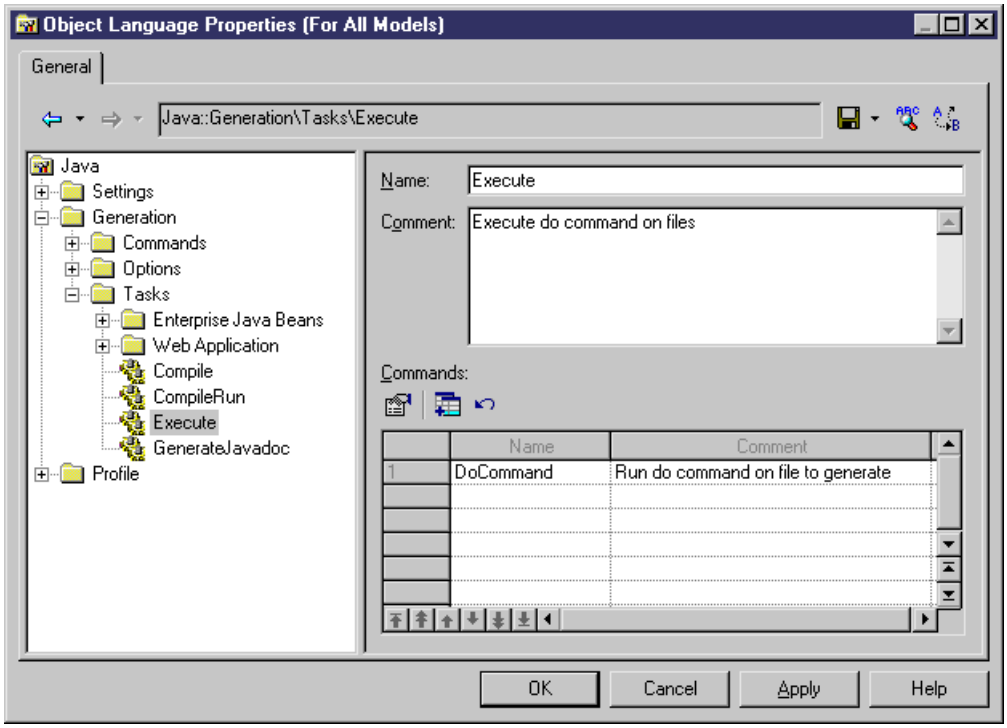
## Example: Adding a Generation Command and Task

In this example, we will add a generation command and associated task to the Java object language

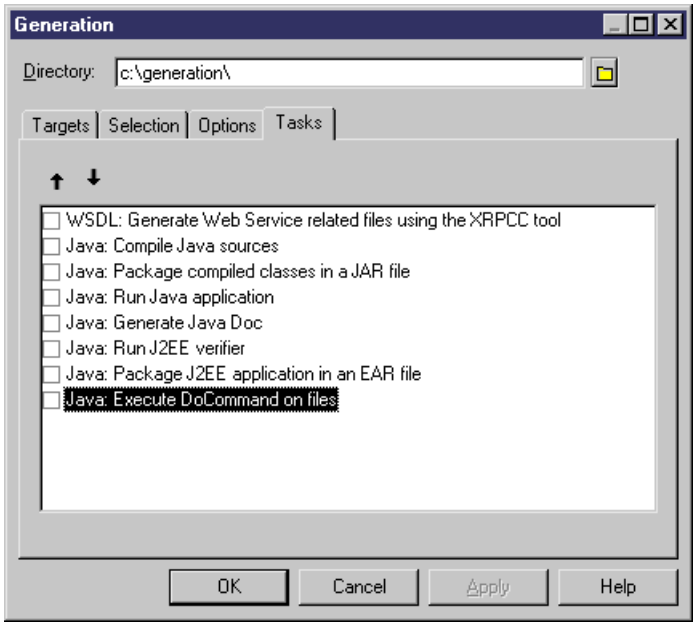
1. Create a new OOM for Java, and then select **Language > Edit Current Object Language** to open the Java resource file.
2. Expand the Generation category, and then right-click the Commands category and select New in the contextual menu to create a new command.
3. Name the command DoCommand and enter an appropriate template:



4. Right-click the Tasks category and select New from the contextual menu, to create a new task. Name the task Execute, click the Add Commands tool, select DoCommand from the list and then click OK to add it to the new task:



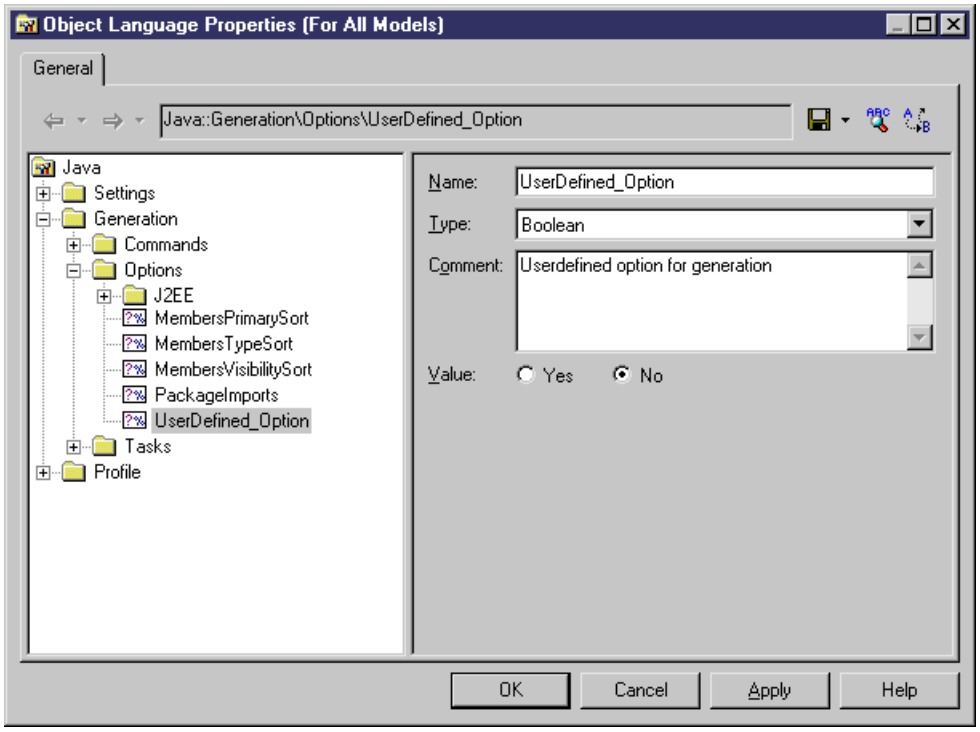

5. Click OK to save your changes and return to the model. Then select **Language > Generate Java code** to open the Generation dialog, and click the Tasks tab. The new task is listed on the tab under its comment (or its name, if no comment has been provided):



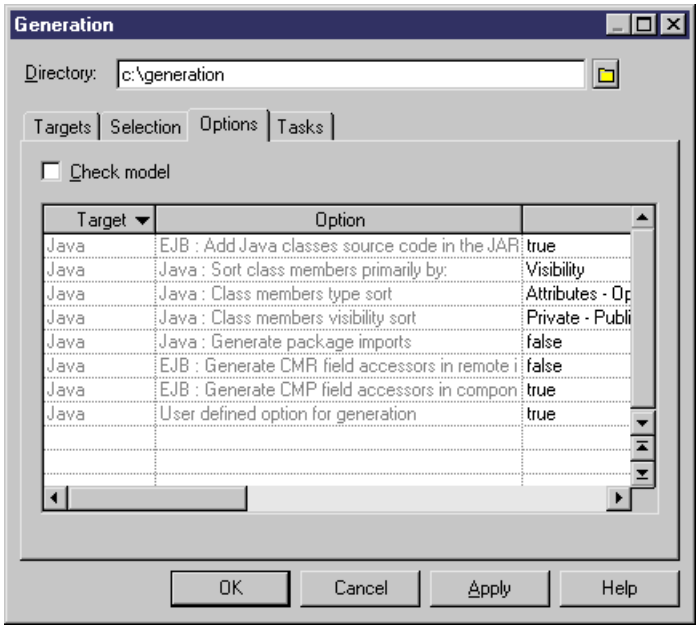

## Example 2: Adding a Generation Option

In this example, we will add a generation option to the Java object language.

1. Select **Language > Edit Current Object Language** to open the Java resource file.
2. Expand the Generation category, and then right-click the Options category and select New in the contextual menu to create a new option:




3. Click OK to save your changes and return to the model. Then select **Language > Generate Java code** to open the Generation dialog, and click the Options tab. The new option is listed on the tab under its comment (or its name, if no comment has been provided):




**Note:** For detailed information about creating and modifying generation templates, see *Customizing Generation with GTL* on page 187. It is strongly advised that you first read this chapter in order to get familiar with the concepts and features of the generation process.

## Profile Category

The resource file Profile category has a sub-category for each metaclass (object type) available in the model. This sub-category may contain Stereotypes, Extended attributes, Methods and so on, to extend the given metaclass.

For more information on the Profile category, see *Extending Your Models with Profiles* on page 121.

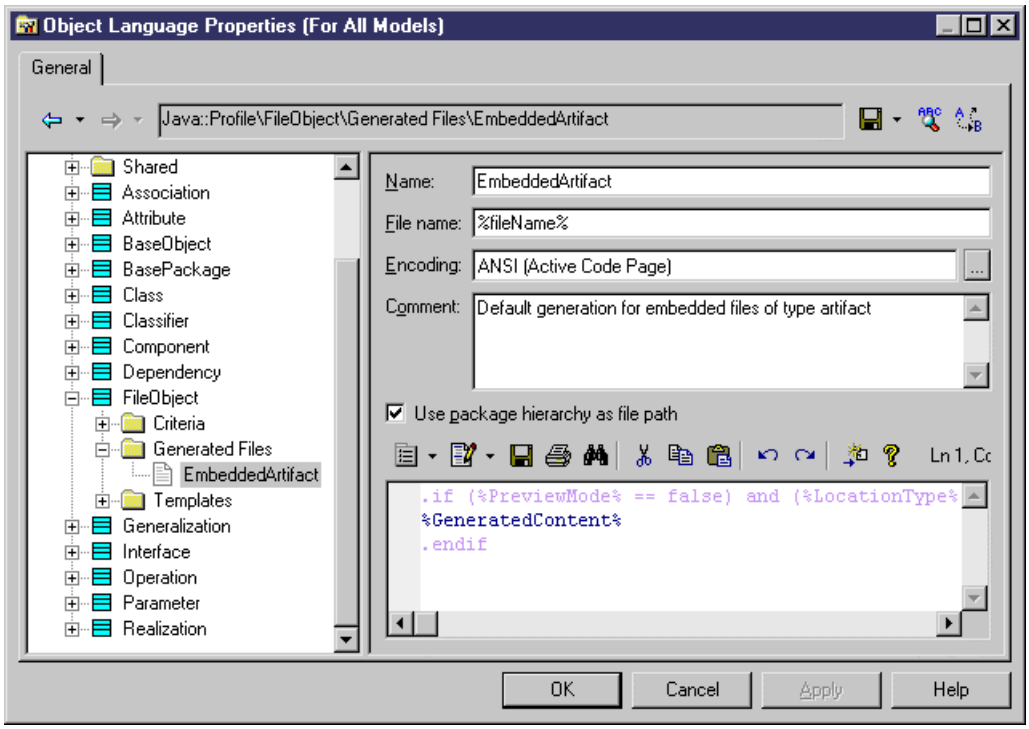
## Generated Files Category

Some objects in the Profile category include a Generated Files category that defines the files that will be generated for the metaclass or for the instances of the metaclass with a selected stereotype or criterion.

## Example

The Generated Files category for file objects in Java contains the EmbeddedArtifact entry that applies to all embedded files of type Artifact to be generated. The EmbeddedArtifact entry contains the File name box that contains the template for the name of the file to be generated.

At the bottom, it contains a text zone that displays the code of the template of the file to generate.




For more information on the Generated Files entry, see *Templates and Generated Files (Profile)* on page 163.

## Encoding

You can define the format for generated files in the Encoding box for each file you generate. A default encoding format is provided to you, but you can also click the Ellipsis button beside the Encoding box to change it. This opens the Text Output Encoding Format dialog box in which you can select the encoding format of your choice.

This dialog box includes the following properties:

| Property | Description |
|---|---|
| Encoding | Encoding format of the generated file |
| Abort on character loss | Allows you to stop generation if characters cannot be identified and are to be lost in current encoding |

## Syntactic Coloring

If the File Name box in the Generated Files entry is empty, there is no file generated. However, it can be very useful to leave this column empty so as to preview the content of the file before generation. You can use the Preview page of the corresponding object at any time for this purpose.

During generation, the template in File Name is evaluated and if one of the following extensions is found, the code is displayed with the corresponding language editor and syntactic coloring (example: .cs for C++):

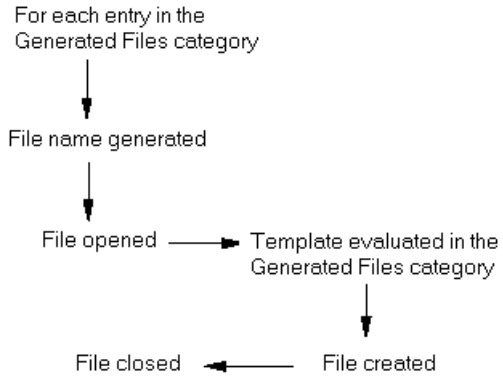
| Extension | Syntactic coloring |
|---|---|
| .java | Java |
| .c, .h | C |
| .sru | PowerBuilder |
| .html | HTML |
| .xml, .xsd, .dtd, .xmi, .jsp, .wsdl, .asp, .aspx, .asmx | XML |
| .cpp, .hpp | CPP |
| .cs | C++ |
| .cls, .vb | Visual Basic 6 |
| .vbs | VB Script |
| .sql | SQL |
| .idl | CORBA |
| .txt | Default text editor |

There are two possible scenarii during generation:

- A file is generated
- No file is generated

## File Generated

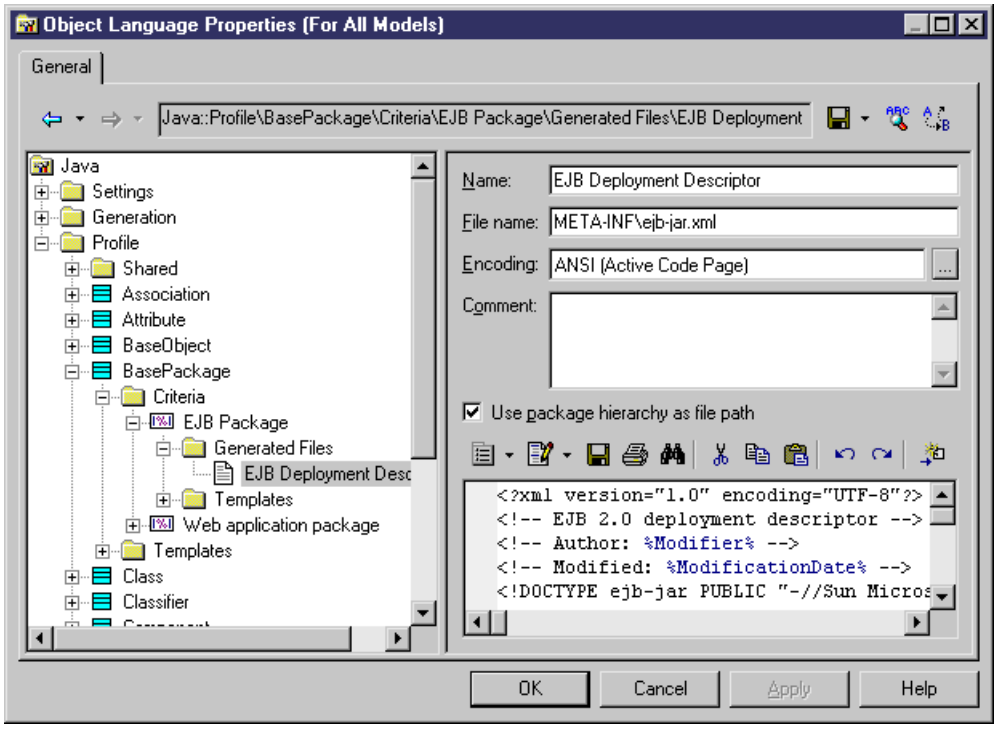
The mechanism of file generation is the following for each object having a Generated Files entry that is not empty:




A file is generated when the File name box contains the name of the file or the template for the name of the file to generate. You can type the name of the file to generate as follows:

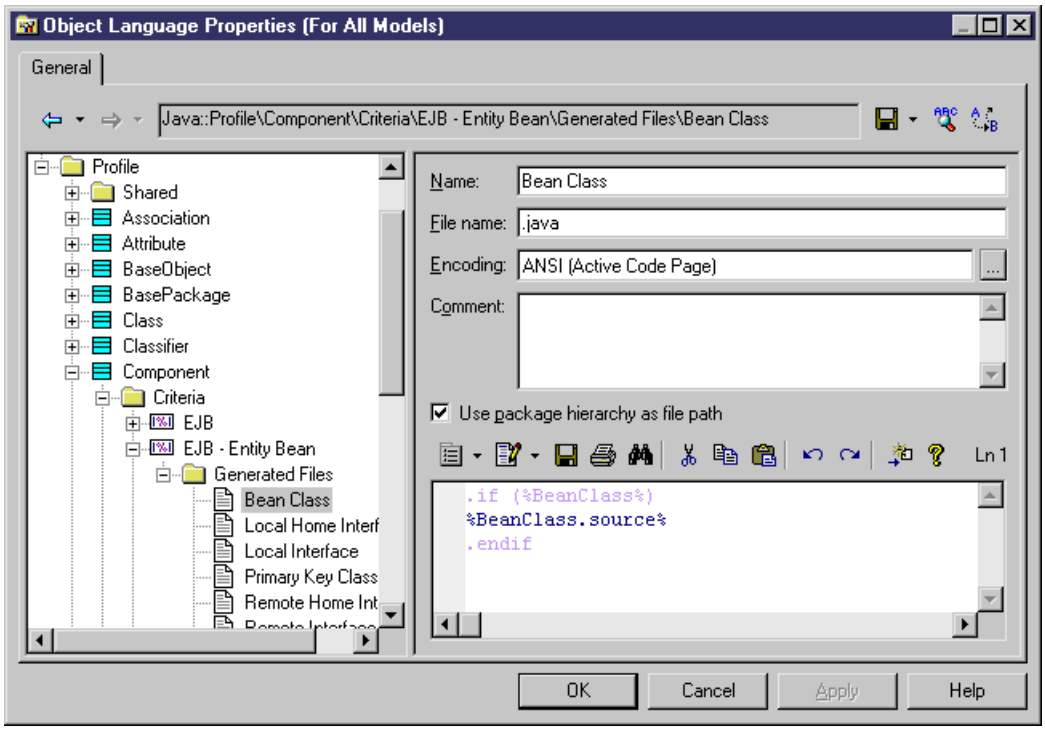
- file_name.extension (for example, ejb-jar.xml)
- %extensionfile_name% (for example, %asmxFileName%)

In this example, a file called ejb-jar.xml located in the META-INF folder is generated.




## No File Generated

In this example, there is no file generated since the File name box starts with a . (dot) character. The contents of the file is only available in the Preview page of the component (EJB - Entity Bean) property sheet.



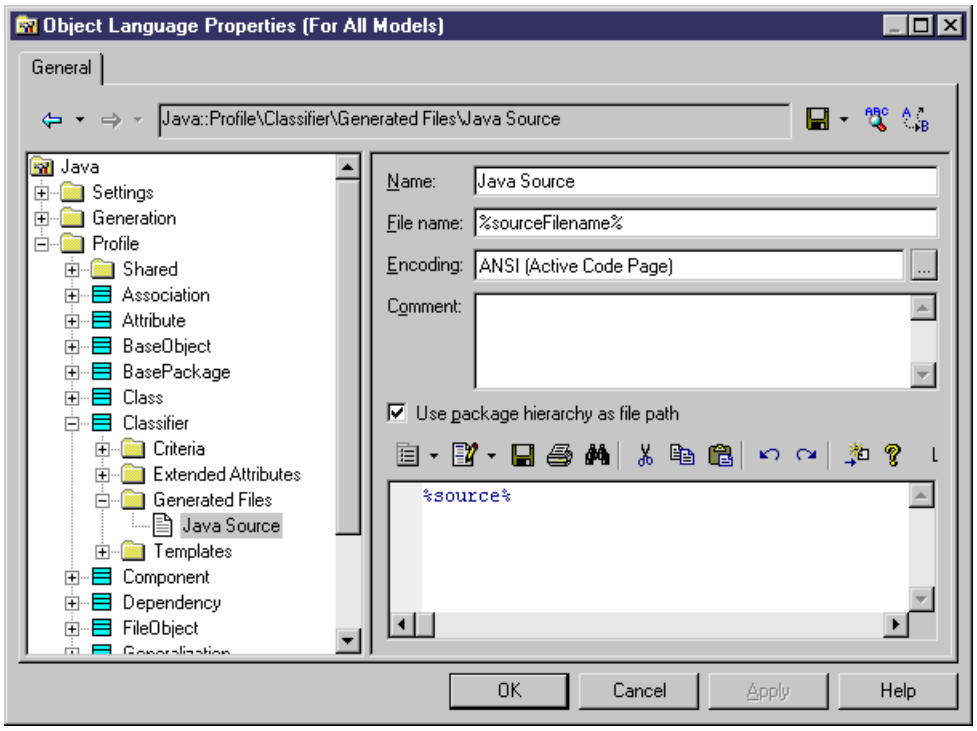

## Templates Category

Templates are used to define what to generate for the current object.

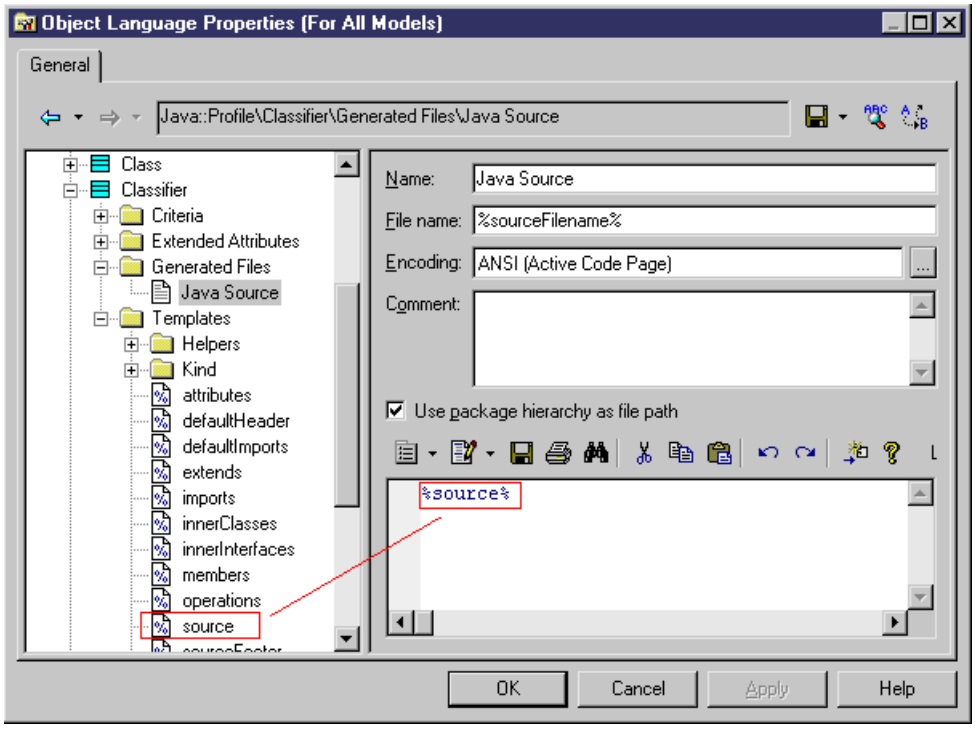
For more information about the Templates category, see *Templates and Generated Files (Profile)* on page 163. For detailed information about writing templates, see *Customizing Generation with GTL* on page 187.

**Note:** Use the F12 key to find templates You can find all templates of the same name using the F12 key. To do so, open a template, position the cursor on a template name in-between % characters, and hit the F12 key. This opens a Browse window that displays all templates prefixed by their metaclass name. Example: position the cursor on %definition% inside a template, hit F12. The Browse window displays all <metaclass_name>::definition. You can then double-click the template of your choice in the Browse window to directly position the cursor on the selected template

In the following example, the Generated Files category for classifiers contains a 'Java Source' entry. This entry contains the template named %source% in the text zone.




When you open the Templates category for classifiers, the template named 'source' is displayed. When the file is generated for a given classifier or for the instances of a classifier with a selected stereotype or criterion, the template evaluated is the 'source' template. The name of the file generated corresponds to the entry in the File name box.




### Association Roles

You can define implementation collections for associations. The evaluated association attributes are: RoleName, Visibility, Multiplicity, Implementation (RoleAContainer or RoleBContainer), ImplementationClass, Ordering, Navigability, Changeability, Initial Value, Persistent, Volatile, Minimum Multiplicity, Maximum Multiplicity, Classifier (ClassA or ClassB). You have to set the active role of the association using attributes *RoleAActive* or *RoleBActive*. When you reference RoleAActive for example, then role A of association becomes active and the implementation script can return attributes corresponding to role A.

### Shared/Extended Attribute Types Category

This section contains various attributes used to control object language support within PowerDesigner.

### Default Association Container

Specifies the default container for implementing associations. This attribute has an editable list of possible values for each object language. You can select here a default value for your language and, if necessary, override this default using the "Default association container" model option.

## Extended Model Definitions

Extended model definitions (.XEM files) provide means for customizing and extending PowerDesigner metaclasses, parameters and generation. XEMs are typed like models in PowerDesigner. You create an XEM for a specific type of model and you cannot share these files between heterogeneous models.

For example, you can attach XEMs to a Java model to help you in working with a particular application server, IDE, or O/R mapping framework. The XEM may provide objects with additional properties or property tabs, and define additional generation targets and options.

PowerDesigner provides a number of predefined XEMs and you can also create your own.

An extended model contains:

- a *profile* definition - a set of metamodel extensions defined on metaclasses
- *generation* parameters - used to develop or complement the default PowerDesigner object generation or for separate generation.

## Attaching an Extended Model Definition to Your Model

When you create a new model, or when you reverse engineer into a new model, you can select one or several extended model definitions and attach them to the model from the New *Model* dialog box.

**Note:** You should never modify the extended model definitions shipped with PowerDesigner. For each original extended model definition you want to modify, you should create a corresponding new extended model definition. To do so you have to create a new extended model definition from the List of Extended Model Definitions, define a name and select the original file in the Copy From list. This allows you to create a new extended model definition that is identical to the original file apart from the name.

You can also attach an extended model definition to your current model.

1. Select **Model** > **Extended Model definitions** to open the List of Extended Model Definitions.
2. Click the Import tool to open the Extended Model Definitions Selection dialog.
3. The available XEMs for this type of model are listed divided by type on one or more sub-tabs. Select one or more XEMs to attach to your model.
4. Select one of the radio buttons:
   - Share – creates a link to the XEM file. Any changes made to the extended model definition will be shared by all those models linked to it.
   - Copy – creates a copy of the XEM private to the model. Any changes made to the XEM affect only the current model, with which it is saved.
5. Click OK to return to your model.:

**Note:** When you import an extended model definition and copy it into a model, the name and code of the extended model definition may be modified in order to respect the naming conventions of the Other Objects category in the Model Options dialog box.

## Creating an Extended Model Definition

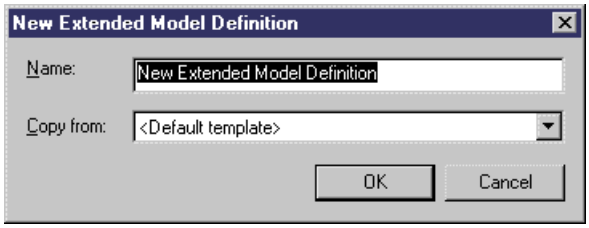
You can create generic and specific extended model definitions:

- A *generic* extended model definition is a library of metamodel extensions and generation parameters saved in a file with the .XEM extension. This file is stored in a central area and can be referenced by models to guarantee data consistency and save time to the user
- A *specific* extended model definition is embedded into a model and develops object definitions and generation parameters in this particular model

### Creating a Generic Extended Model Definition

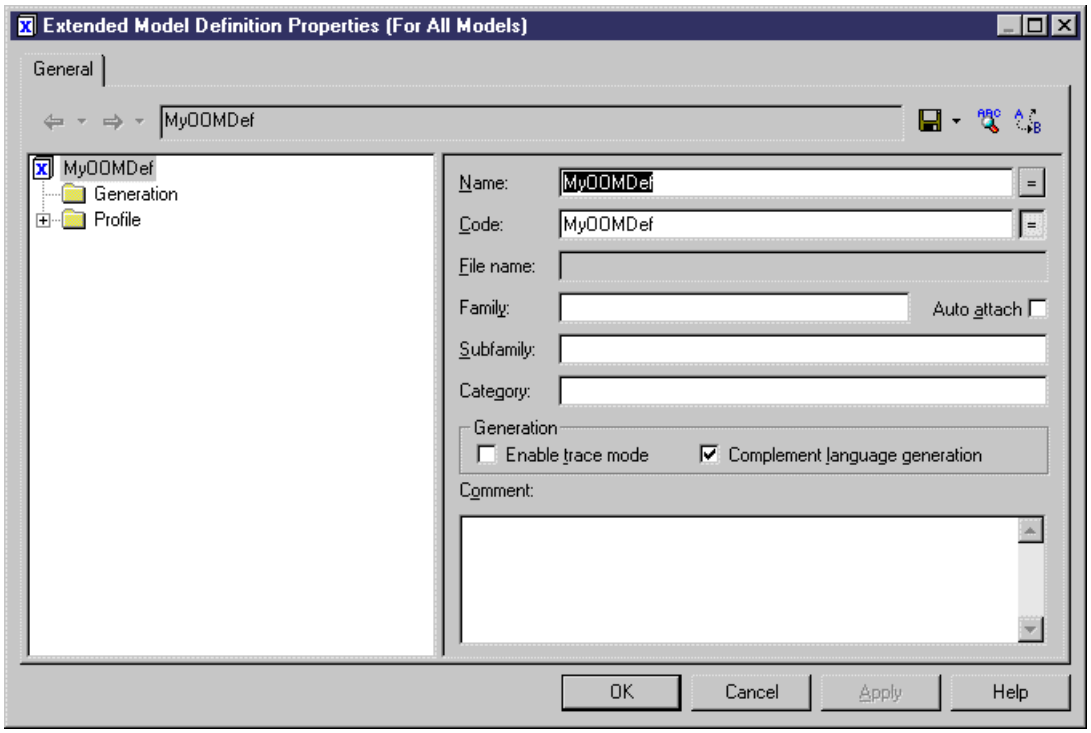
You can create generic extended model definitions to share information between models of the same type.

1. Select **Tools** > **Resources** > **Extended Model Definitions** > **Model** type to open the List of Extended Model Definitions.
2. Click the New tool and enter a name for the new extended model definition in the Name box:



3. [optional] Select an existing XEM in the Copy From box. If you do not select an XEM a standard empty XEM for the selected model type will be created.
4. Click OK, and enter an appropriate file name in the Save As dialog box.

   **Note:** If you change the default path, extended model definitions do not appear in the list of extended model definitions. If you want to save your extended model definitions in a specific folder, you have to define a specific named path in the General Options dialog box. For more information, see "Defining named paths" in the *Core Features Guide*.
5. Click Save to create the file and open it in the Resource Editor:




6. Define the appropriate object extensions and generation tasks for your extended model definition, and then click OK to save it and return to the List of Extended Model Definitions.

   The new XEM is now available to be attached to your models (see *Attaching an extended model definition to your model* on page 19).

### Creating an Extended Model Definition for a Specific Model

You can create an extended model definition for a specific model, in this case, it has the same type as the current model.

1. Open your model, and then select **Model > Extended Model Definitions** to open the List of Extended Model Definitions.
2. Click the Add a Row tool and enter a name for the new XEM.
3. Click the Properties tool to open the new XEM in the Resource Editor.
4. Define the appropriate object extensions and generation tasks for your extended model definition, and then click OK to save it and return to your model.

   The new XEM is now attached to your model, and you can access any object extensions or generation tasks you have defined in the normal fashion.

## Exporting an Extended Model Definition

If you export an XEM created in a model, it becomes available in the List of Extended Model Definitions, and can be shared with other models. When you export an XEM, the original remains embedded in the model.

1. Select **Model** > **Extended model definitions** to open the List of Extended Model Definitions.
2. Select an extended model definition in the list.
3. Click the Export an Extended Model Definition tool.

   A standard Save As dialog box is displayed.
4. Type a name and select a directory for the extended model definition.
5. Click Save.

   The extended model definition is saved in a library directory where it can be shared with other models.

## Extended Model Definition Properties

All extended model definitions have the same basic category structure. The root node of each file contains the following properties:

| Property | Description |
|---|---|
| Name | Specifies the name of the extended model definition. This name must be unique in a model for generic or specific XEMs. |
| Code | Specifies the code of the extended model definition. This code must be unique in a model for generic or specific XEMs. |
| File Name | [read-only] Specifies the path to the extended model definition file. If the XEM has been copied to your model, this field is empty. |
| Family | Restricts the availability of the XEM to a particular target family. For example, when an XEM has the family Java, it is available only for use with targets in the Java object language family. |
| Subfamily | Refines the family. For example, EJB 2.0 is a sub-family of Java. |
| Auto attach | Specifies that the XEM will be automatically attached to new models with a target belonging to the specified family |
| Category | Groups XEMs by type for generation and in the Extended Model Definition Selection window. For example, a category called "Application Server" is used to group all server XEMs. Extended model definitions having the same category cannot be generated simultaneously. If you do not define a category, the XEM is displayed in the General Purpose category and is treated as a generation target. |
| Enable Trace Mode | Lets you preview the templates used during generation. Before starting the generation, click the Preview page of the relevant object, and hit the Refresh tool to display these templates.<br>When you double-click on a trace line from the Preview page, the Resource Editor opens to the corresponding template definition in the Profile\Object\Templates category. |
| Complement language generation | Specifies that the XEM is used to complement the generation of an target language. The generation items of the object language are merged with those of the XEM before generation. All generated files specified in the target resource file and any attached XEMs are generated. If two generated files have identical names, the file in the XEM overrides the one defined in the target.<br>Note that PowerBuilder does not support XEMs for complementary generation. |
| Comment | Provides a descriptive comment for the XEM. |

### Transformation Profile Category

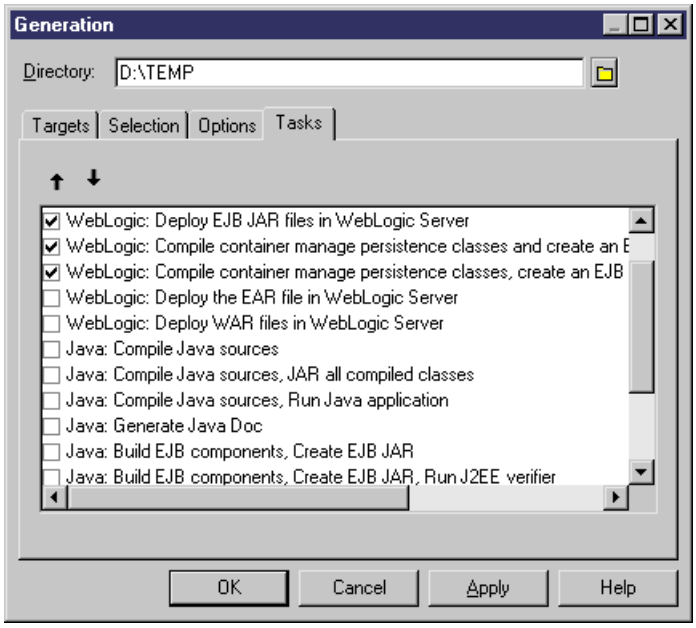
A transformation profile is a group of transformations used during model generation when you need to apply changes to objects in the source or target models.

For information about creating transformations and transformation profiles, see *Extending Your Models with Profiles* on page 121. For information about invoking transformations, see "Applying Model Transformations" in the "Models" chapter of the *Core Features Guide*.

### Generation Category

The Generation category contains Generation commands, options, and tasks to define and activate a generation process.

In the following example, three tasks are defined in the WebLogic extended model definition; the other tasks proceed from the object language of the model (Java):




For more information about these categories, see *Generation category* on page 10.

## Extending Code Generation with Extended Model Definitions

The extended model definition generation parameters influence the content of the generation dialog box. The following table shows how you can customize the generation from the extended model definition editor.

| Generation dialog box | Extended model definition |
|---|---|
| Targets page | The Target page is displayed if the Complement Language Generation check box in the extended model definition properties is set to Yes and if the extended model definition contains at least one task or generated file |
| Options page | Define options in Generation\Options using boolean, list and string entry types |
| Tasks page | Define *commands* using command entries and reference these commands in tasks |

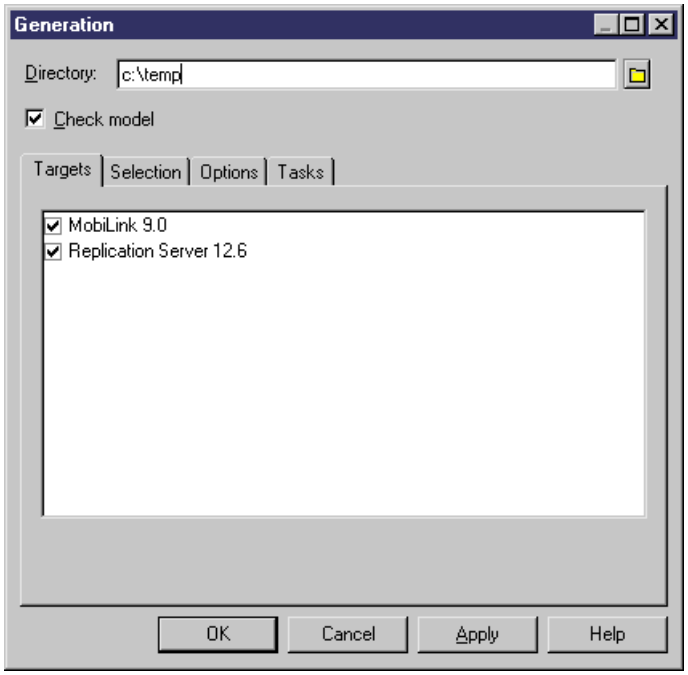
## Creating Separate Generation Targets with Extended Model Definitions

Extended model definitions can be used to create new generation targets provided the following conditions are respected:

- The Complement Language Generation check box in the extended model definition property sheet should not be selected
- The extended model definition contains generated files and templates. During generation, the evaluation of a template generates text which is written to a file.

This type of generation is called *extended generation*, and is available from the **Tools > Extended Generation** command.

If you have several extended model definitions designed for extended generation, these will appear in the Targets page of the extended generation dialog box.




You can create commands in the Tools menu to directly access extended generation for a selected target. To do so you have to:

- Create a menu in the Model metaclass in the Profile category of the extended model definition, name and select the Tools menu in the Location list

  For more information about creating menus, see *Menus (Profile)* on page 161.
- Create a method to invoke extended generation as follows:

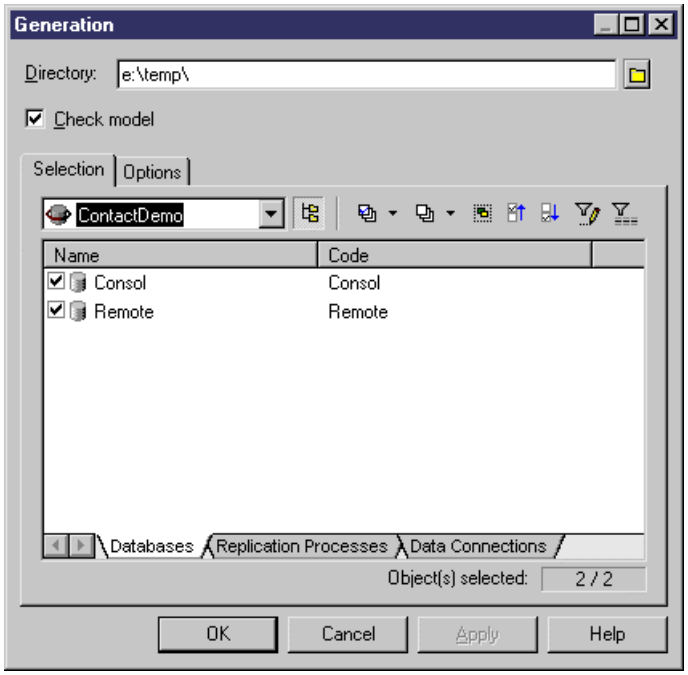
```
Sub %Method%(obj)

 Dim selection ' as ObjectSelection

 ' Create a new selection
 set selection = obj.CreateSelection

 ' Add object of the active selection in the created selection
 selection.AddActiveSelectionObjects

 ' Generate scripts for specific target
 InteractiveMode = im_Dialog
 obj.GenerateFiles "", selection, "specific target"

End Sub
```

Where "specific target" is the code of the extended generation target.

For more information about creating methods, see *Methods (Profile)* on page 159.

- Add the method for extended generation to the menu in order to create a specific command
- Save the extended model definition

The new command is displayed in the Tools menu.




The Targets tab does not display because the underlying method already specifies a generation target.

## The PowerDesigner Public Metamodel

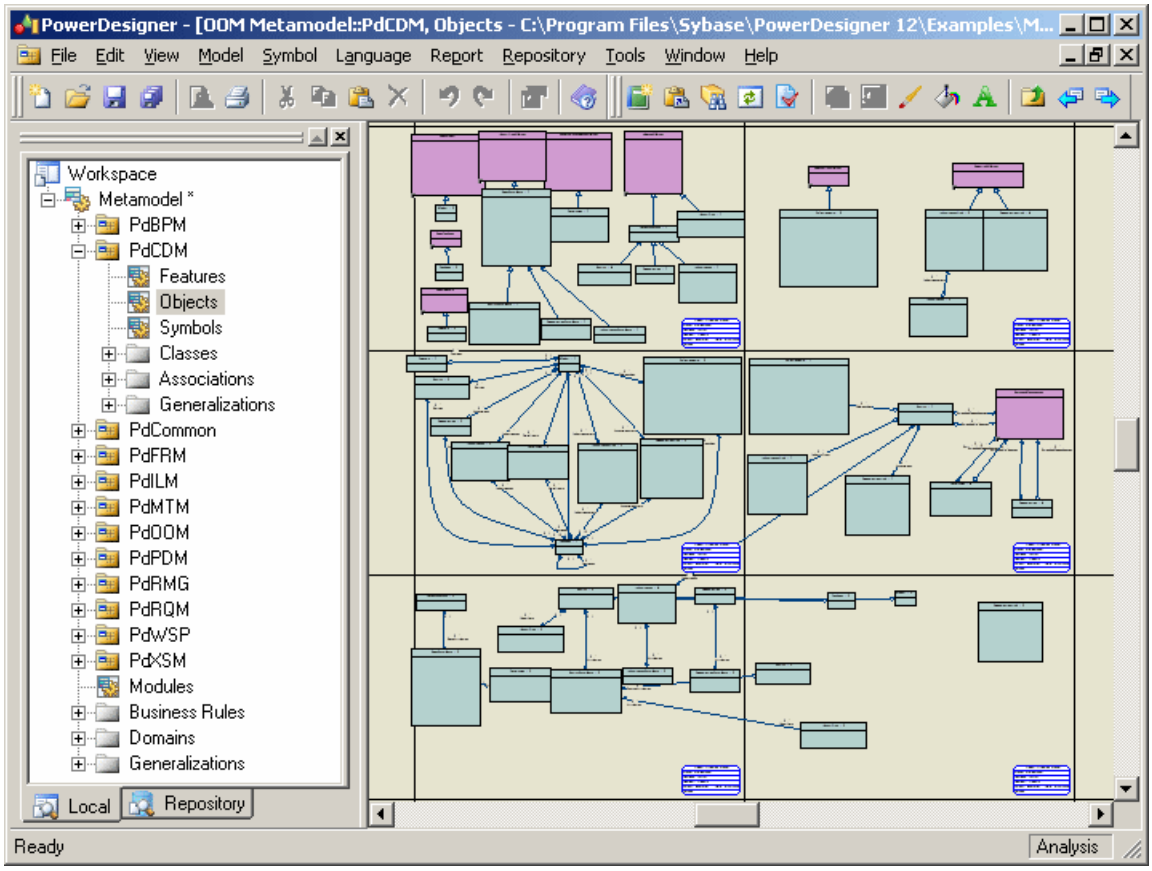
A metamodel describes the elements of a model, and the syntax and semantics of their manipulation. Where a model is an abstraction of data, and can be described using metadata, the metamodel is an abstraction of that metadata.

The PowerDesigner public metamodel is an abstraction of the metadata for all the PowerDesigner models, which is represented in an object-oriented model. It is intended to help you understand the overall structure of the PowerDesigner modeling metadata when working with:

- VB scripts
- Generation Template Language (GTL) templates
- PowerDesigner XML model files (see *The PowerDesigner XML Model File Format* on page 29)

The public metamodel OOM is located at:

```
[PowerDesigner install dir]\Examples\MetaModel.oom
```




For documentation, select **Help > Metamodel Objects Help**

The metamodel is divided into the following main packages:

- PdBPM - Business Process Model
- PdCDM - Conceptual Data Model
- PdCommon - contains all objects shared between two or more models, and the abstract classes of the model. For example, business rules, which are available in all models, and the BaseObject class, from which all model objects are derived, are defined in this package. Other model packages are linked to PdCommon by generalization links indicating that each model inherits common objects from the PdCommon package.
- PdFRM - Free Model
- PdILM - Information Liquidity Model
- PdMTM - Merise Model (available in French only)
- PdOOM - Object Oriented Model
- PdPDM - Physical Data Model
- PdRMG - Repository
- PdRQM - Requirements Model
- PdXSM - XML Model
- PdWSP – Workspace

Each of these top-level packages contains the follow kinds of sub-objects, organized by diagram or, in the case of PdCommon, by sub-packages:

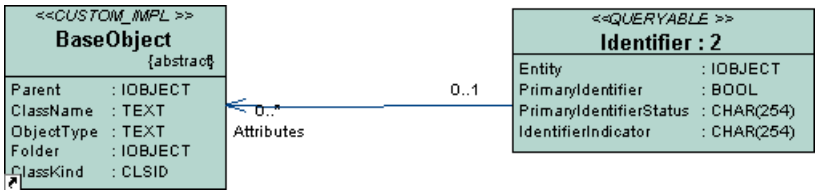
- Features - All the features implemented by classes in the model. For example, Report (available in all models) belongs to PdCommon, and AbstractDataType belongs to PdPDM.
- Objects - Design objects in the model
- Symbols - Graphical representation of design objects

## Metamodel Concepts

The PowerDesigner public metamodel uses standard UML concepts:

- *Public Names* - Each object in the metamodel has a name and a code corresponding to the public name of the object. The public name is the unique identifier of the object in a model library or package (for example, PdCommon) visible in the Modules diagram in the metamodel. Public names are used in the PowerDesigner XML model files (see *The PowerDesigner XML Model File Format* on page 29) and in the GTL (see the *Customizing Generation with GTL* on page 187).The public name does not always match the object's name in the PowerDesigner interface.
- *Classes* - are used to represent metadata in the following ways:
  - *Abstract classes* - are used only to share attributes and behaviors, and are not visible in the PowerDesigner interface. Instantiable classes inherit from abstract classes via generalization links. For example, NamedObject is an abstract class, which stores standard attributes like name, code, comment, annotation, and description, which are inherited by most PowerDesigner design objects.
  - *Instantiable/Concrete classes* - correspond to objects displayed in the interface. They have their own attributes, such as type or persistence, and they inherit attributes and behaviors from abstract classes through generalization links.
- *Class attributes* - are class properties that can be *derived* or not. Classes linked to other classes with generalization links usually contain derived attributes that are calculated from the attributes or collections of the parent class. Neither derived attributes, nor attributes migrated from navigable associations, are stored in the model file. Non-derived attributes are proper to the class, and are stored in the model and saved in the model file.
- *Associations* - are used to express the semantic connections between classes called *collections*. In the association property sheet, the roles carry information about the end object of the association. In the PowerDesigner metamodel, this role has the same name as a collection for the current object. PowerDesigner objects are linked to other objects using collections.

  Associations usually have only one role, which is at the far end of the association from the class for which it represents a collection. In the following example, Identifier has a collection called Attributes:




  When associations have two roles, both collections cannot be saved in the XML file, and only the collection with the *navigable* role will be saved (see *The PowerDesigner XML Model File Format* on page 29).
- *Composition* – expresses an association where the children live and die with the parent and, when the parent is copied, the child is also copied. For example, in package PdCommon, diagram Option Lists, class NamingConvention is associated with class BaseModelOptions with 3 composition associations: NameNamingConventions, CodeNamingConventions, and NamingConventionsTemplate. These composition associations express the fact that class NamingConvention would not exist without class BaseModelOptions.
- *Generalizations* - show the *inheritance* links existing between a more general, usually abstract, class and a more specific, usually instantiable, class. The more specific class inherits from the attributes of the more generic class, these attributes are called derived attributes.
- *Comments and notes* - explains the role of the object in the metamodel. Some internal implementation details are also available in the **Notes** > **Annotation** page of the classes property sheets.

## Navigating in the Metamodel

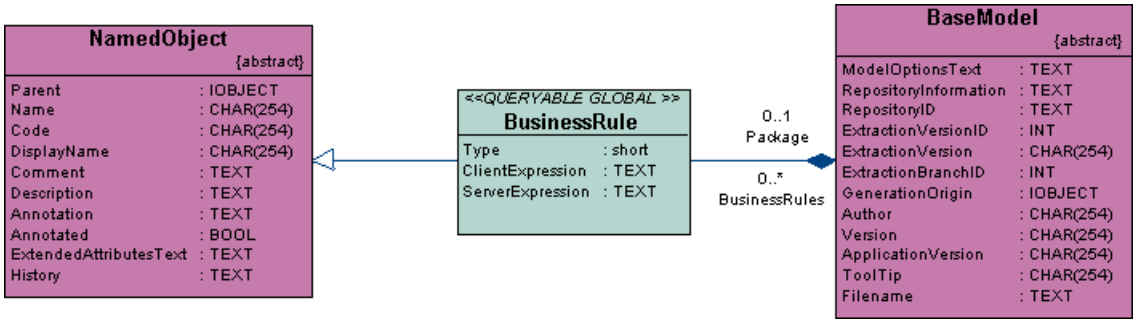
You can use the Browser to expand and collapse the packages in order to explore their contents. Double-click a diagram to display it in the canvas.

Each diagram shows classes that relate to each other via associations and generalizations. Each class has a name (the public name) and is described by zero or more attributes. It may assume various roles in associations with other classes.

Many associations display their roles which makes it possible to identify object collections (see *Metamodel concepts* on page 26).

Classes in *green* are classes whose behavior is explained in the current diagram, while classes in *purple* are usually shortcuts of a class existing in another package, and are presented only to help in understanding the context. The shortcut makes it easier to read the diagram and understand the generalization links between classes. If you want to understand a purple class, right-click it and select *Open Related Diagram* from the contextual menu to open the diagram where the class is actually defined.

In the following example taken from PdCommon/Objects/Common Instantiable Objects, BusinessRule (in green) is developed, while NamedObject and BaseModel are present only to express inheritance and composition links with abstract classes.




Double-click any class to show its property sheet. The *Dependencies* tab contains (among others) the following sub-tabs:

- *Associations* - you can customize the filter in order to display association roles, which provides a list of the collections of the current object
- *Generalizations* - lists the generalization links where the current object is the parent. You can use this list to display all the children of the current class. Child classes inherit attributes from the parent class and do not display derived attributes
- *Specializations* - Displays the parent of the current object. The current class inherits attributes from this parent
- *Shortcuts* - displays the list of shortcuts created for the current object

The *Associations* tab lists the migrated associations for the class.

## Accessing the Metamodel with VB Script

You can access and manipulate PowerDesigner internal objects using VB Script. The metamodel (and its online help, available by selecting **Help** > **Metamodel Objects Help** ) provides useful information about objects:

| Information | Description |
|---|---|
| Public name | The name and code of the metamodel objects are the public names of PowerDesigner internal objects.<br>Examples: AssociationLinkSymbol, ClassMapping, CubeDimensionAssociation |
| Object collections | You can identify the collections of a class by observing the associations linked to it in the diagram. The role of each association is the name of the collection.<br>Example: In PdBPM, the Format association connects the classes MessageFormat and MessageFlow. The role of this association is Usedby, which corresponds to the message flow collection of MessageFormat. |
| Object attributes | You can view the attributes of a class together with the attributes it inherits from other classes via generalization links.<br>Example: In PdCommon/Objects/Common Instantiable Objects, you can view the attributes of BusinessRule, FileObject, and ExtendedDependency, and also those that they inherit from abstract classes via generalization links. |

| Information | Description |
|---|---|
| Object operations | Operations in metamodel classes correspond to object methods used in VBS.<br>Example: BaseModel contains the operation Compare that is can be used in VB scripting |
| <<notScriptable>> stereotype | Objects that do not support VB scripting have the <<notScriptable>> stereotype.<br>Example: RepositoryGroup |

For more information about public names and other metamodel concepts, see *Metamodel concepts* on page 26.

For detailed information about using VB Script with PowerDesigner, see *Scripting PowerDesigner* on page 235.

## Using the Metamodel with GTL

The Generation Template Language (GTL) uses *templates* to generate files. A template is a piece of code defined on a given PowerDesigner metaclass and the metaclasses that inherit from this class. It can be used in different contexts for text and potentially code generation.

These templates can be considered as metamodel extensions as they are special kinds of metamodel class attributes. You can define as many templates as needed for any given metaclass using the following syntax:

```
<metamodel-classname> / <template-name>
```

Templates are inherited by all the descendants of the metaclass they are defined for, and so can be used to share template code between metaclasses with a common ancestor. For example, if you define a template for the BaseObjects abstract class, all the classes linked via generalization links to this class inherit from this template.

The GTL uses macros such as foreach_item, for iterating over object collections. The template specified inside the block is translated over all the objects contained in the specified collection. The metamodel provides useful information about the collections of the metaclass on which you define a template containing an iteration macro.

### Calculated Attributes

The following calculated attributes are metamodel extensions specific to the GTL:

| Metaclass | Attributes |
|---|---|
| PdCommon.BaseObject | • isSelected (boolean) - True if the object is part of the selection in the generation dialog<br>• isShorctut (boolean) - True if the object was accessed by dereferencing a shortcut |
| PdCommon.BaseModel | • GenOptions (struct) - Gives access to user-defined generation options |
| PdOOM.* | • ActualComment (string) - Cleaned–up comment (with /**, /*, */ and // removed) |
| PdOOM.Association | • RoleAMinMultiplicity (string)<br>• RoleAMaxMultiplicity (string)<br>• RoleBMinMultiplicity (string)<br>• RoleBMaxMultiplicity (string) |

| Metaclass | Attributes |
|---|---|
| PdOOM.Attribute | • MinMultiplicity (string)<br>• MaxMultiplicity (string)<br>• Overridden (boolean)<br>• DataTypeModifierPrefix (string)<br>• DataTypeModifierSuffix (string)<br>• @<tag> [Java-specific] (string) - Javadoc@<tag> extended attribute with additional formatting |
| PdOOM.Class | • MinCardinality (string)<br>• MaxCardinality (string)<br>• SimpleTypeAttribute [XML-specific]<br>• @<tag> [Java-specific] (string) - Javadoc@<tag> extended attribute with additional formatting |
| PdOOM.Interface | • @<tag> [Java-specific] (string) - Javadoc@<tag> extended attribute with additional formatting |
| PdOOM.Operation | • DeclaringInterface (object)<br>• GetSetAttribute (object)<br>• Overridden (boolean)<br>• ReturnTypeModifierPrefix (string)<br>• ReturnTypeModifierSuffix (string)<br>• @<tag> [Java-specific] (string) - Javadoc@<tag> extended attribute with additional formatting (especially for @throws, @exception, @params) |
| PdOOM.Parameter | • DataTypeModifierPrefix (string)<br>• DataTypeModifierSuffix (string) |

### Calculated Collections

The following calculated collections are metamodel extensions specific to the GTL:

| Metaclass name | Collection name |
|---|---|
| PdCommon.BaseModel | Generated <metaclass-name>List - Collection of all objects of type <metaclass-name> that are part of the selection in the generation dialog |
| PdCommon. BaseClassifierMapping | SourceLinks |
| PdCommon. BaseAssociationMapping | SourceLinks |

## The PowerDesigner XML Model File Format

All model files in PowerDesigner have an extension that corresponds to the module in which they are saved. For example, a model saved in the object-oriented module has the extension OOM. Beside the file extension, you can decide a format for saving your models:

- BIN (Binary) - files are smaller and significantly quicker to open and save
- XML – larger and slower than binary files, but manipulable with standard XML editors. There is a DTD for each different kind of model file in the \DTD folder in the PowerDesigner installation directory.

## PowerDesigner XML Model File Markup

The following markups are used in PowerDesigner XML files:

| Markup | Description |
|---|---|
| *<c:collection>*<br>*</c:collection>* | Collection - A collection of objects linked to another object. You can use the PowerDesigner meta-model to visualize the collections of an object. For example <c:Children> |
| *<o:object>*<br>*</o:object>* | Object - An object that you can create in PowerDesigner. For example <o:Model>. When an object is already defined in the file, a reference is created the next time it is browsed in the XML file. For example <o:Class Ref= "xyz"/> |
| *<a:attribute>*<br>*</a:attribute>* | Attribute - An object is made up of a number of attributes each of which you can modify independently. For example <a:ObjectID> |

The format of XML files reflects the way model information is saved: PowerDesigner browses each object in order to save its definition.

The definition of an object implies the definition of its attributes and its collections. This implies that PowerDesigner checks each object and drills down the collections of this object to define each new object and collection in these collections, and so on, until the process finds terminal objects that do not need further analysis.

Since collections can overlap, the format of PowerDesigner model files resemble a tree view, which starts from a root node (the root object containing any model collection) and cascades through collections.

When an object is mentioned in a collection, PowerDesigner either defines this object using the `<o:object Id="XYZ">` syntax or references it with the `<o:object Ref="XYZ"/>` (empty tag) syntax. Object definition is only used in composition collections, where the parent object owns the children in the association.

In both cases, XYZ is a unique identifier automatically assigned to an object when it is found for the first time.

## XML and the PowerDesigner Metamodel

PowerDesigner models are made up of objects, the properties and interactions of which are explained in the public metamodel.

You can use the PowerDesigner public metamodel (see *The PowerDesigner Public Metamodel* on page 24) to better understand the format of PowerDesigner XML files.

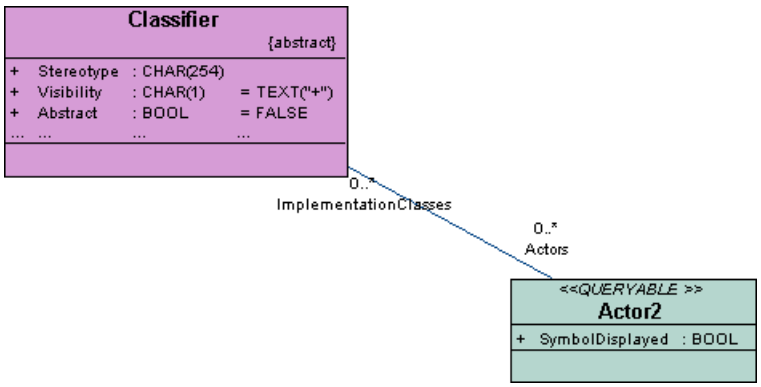
Object names, as declared in markup <o:name of object>, correspond to public names in the metamodel. You can search for an object in the metamodel using the object name found in the XML file.

Once you have found and located the object in the metamodel you can read the following information:

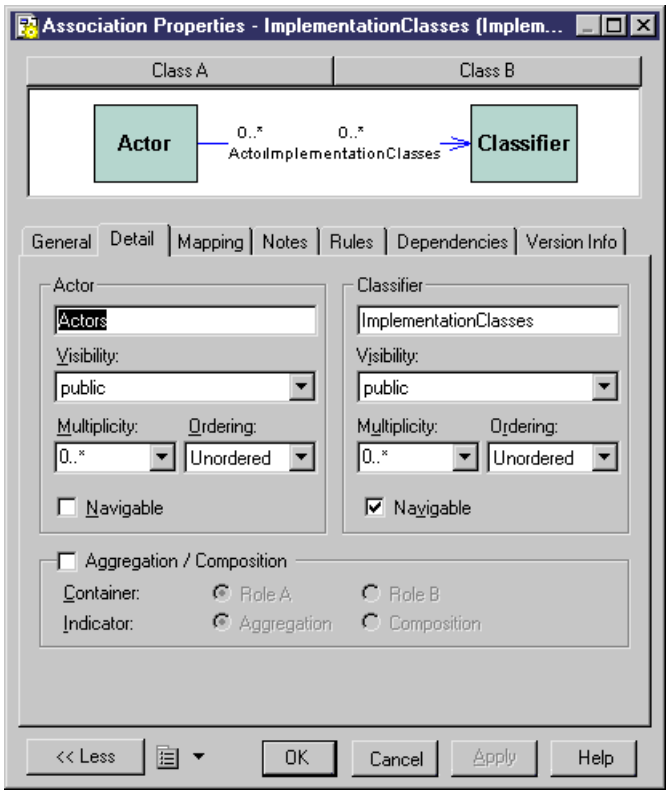
- Each PowerDesigner object can have several collections corresponding to other objects to interact with, these collections are represented by the associations existing between objects. The *roles* of the associations (aggregations and compositions included) correspond to the collections of an object. For example, each PowerDesigner model contains a collection of domains called Domains.

  Usually associations have only one role, the role is displayed at the opposite of the class for which it represents a collection. However, the metamodel also contains associations with two roles, in such case, both collections cannot be saved in the XML file. You can identify the collection that will be saved from the association property sheet: the role where the *Navigable* check box is selected is saved in the file.

  In the following example, association has two roles which means Classifier has a collection Actors, and Actor2 has a collection ImplementationClasses:



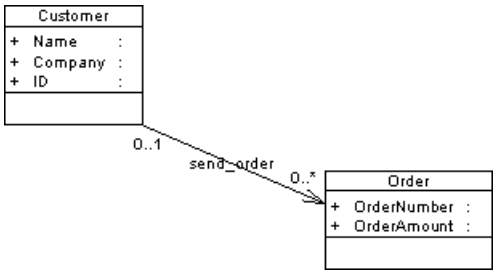

If you display the association property sheet, you can see that the Navigable check box is selected for role ImplementationClass, which means that only collection ImplementationClass will be saved in file.




- Attributes with the *IOBJECT* data type are attributes in the metamodel while they appear as collections containing a single object in the XML file. This is not true for Parent and Folder that do not contain any collection.

## Example: Simple OOM XML File

The following model contains two classes and one association. We are going to explore the XML file corresponding to this model.




The file starts with several lines stating XML and model related details.

The first object to appear is the root of the model <o:RootObject Id="01">. RootObject is a model container that is defined by default whenever you create and save a model. RootObject contains a collection called Children that is made up of models.

In our example, Children contains only one model object that is defined as follows:

```
<o:Model Id="o2">
 <a:ObjectID>3CEC45F3-A77D-11D5-BB88-0008C7EA916D</a:ObjectID>
 <a:Name>ObjectOrientedModel_1</a:Name>
 <a:Code>OBJECTORIENTEDMODEL_1</a:Code>
 <a:CreationDate>1000309357</a:CreationDate>
 <a:Creator>arthur</a:Creator>
 <a:ModificationDate>1000312265</a:ModificationDate>
 <a:Modifier>arthur</a:Modifier>
 <a:ModelOptionsText>
[ModelOptions]
...
```

Below the definition of the model object, you can see the series of ModelOptions attributes. Note that ModelOptions is not restricted to the options defined in the Model Options dialog box of a model, it gathers all properties saved in a model such as intermodel generation options.

After ModelOptions, you can identify collection <c:ObjectLanguage>. This is the object language linked to the model. The second collection of the model is <c:ClassDiagrams>. This is the collection of diagrams linked to the model, in our example, there is only one diagram defined in the following paragraph:

```
<o:ClassDiagram Id="o4">
   <a:ObjectID>3CEC45F6-A77D-11D5-BB88-0008C7EA916D</a:ObjectID>
   <a:Name>ClassDiagram_1</a:Name>
   <a:Code>CLASSDIAGRAM_1</a:Code>
   <a:CreationDate>1000309357</a:CreationDate>
   <a:Creator>arthur</a:Creator>
   <a:ModificationDate>1000312265</a:ModificationDate>
   <a:Modifier>arthur</a:Modifier>
   <a:DisplayPreferences>
...
```

Like for model options, ClassDiagram definition is followed by a series of display preference attributes.

Within the ClassDiagram collection, a new collection called <c:Symbols> is found. This collection gathers all the symbols in the model diagram. The first object to be defined in collection Symbols is AssociationSymbol:

```
<o:AssociationSymbol Id="o5">
   <a:CenterTextOffset>(1, 1)</a:CenterTextOffset>
   <a:SourceTextOffset>(-1615, 244)</a:SourceTextOffset>
   <a:DestinationTextOffset>(974, -2)</a:DestinationTextOffset>
   <a:Rect>((-6637,-4350), (7988,1950))</a:Rect>
   <a:ListOfPoints>((-6637,1950),(7988,-4350))</a:ListOfPoints>
   <a:ArrowStyle>8</a:ArrowStyle>
   <a:ShadowColor>13158600</a:ShadowColor>
   <a:FontList>DISPNAME 0 Arial,8,N
```

AssociationSymbol contains collections <c:SourceSymbol> and <c:DestinationSymbol>. In both collections, symbols are referred to but not defined: this is because ClassSymbol does not belong to the SourceSymbol or DestinationSymbol collections.

```
<c:SourceSymbol>
    <o:ClassSymbol Ref="o6"/>
   </c:SourceSymbol>
   <c:DestinationSymbol>
    <o:ClassSymbol Ref="o7"/>
   </c:DestinationSymbol>
```

The association symbols collection is followed by the<c:Symbols> collection. This collection contains the definition of both class symbols.

```
<o:ClassSymbol Id="o6">
   <a:CreationDate>1012204025</a:CreationDate>
   <a:ModificationDate>1012204025</a:ModificationDate>
   <a:Rect>((-18621,6601), (-11229,12675))</a:Rect>
   <a:FillColor>16777215</a:FillColor>
   <a:ShadowColor>12632256</a:ShadowColor>
   <a:FontList>ClassStereotype 0 Arial,8,N
```

Collection <c:Classes> follows collection <c:Symbols>. In this collection, both classes are defined with their collections of attributes.

```
<o:Class Id="o10">
   <a:ObjectID>10929C96-8204-4CEE-911#-E6F7190D823C</a:ObjectID>
   <a:Name>Order</a:Name>
   <a:Code>Order</a:Code>
   <a:CreationDate>1012204026</a:CreationDate>
   <a:Creator>arthur</a:Creator>
   <a:ModificationDate>1012204064</a:ModificationDate>
   <a:Modifier>arthur</a:Modifier>
   <c:Attributes>
  <o:Attribute Id="o14">
```

Attribute is a terminal object: there is not further ramification required to define this object.

Each collection belonging to an analyzed object is expanded, and analyzed and the same occurs for collections within collections.

Once all objects and collections are browsed, the following markups appear:

```
</o:RootObject>
</Model>
```

## Modifying an XML File

You can modify a model by editing its XML file using a standard text editor such as Notepad or in an XML editor, but you must be very careful, because even a minor syntax error could render the file unusable.

If you create an object in an XML file by copying an existing object of the same type, make sure that you remove the duplicated OID. It is better to remove a duplicated OID than try to create a new one because this new ID may not be unique in the model. PowerDesigner will automatically assign an OID to the new object when you open the model.

# DBMS Resource File Reference

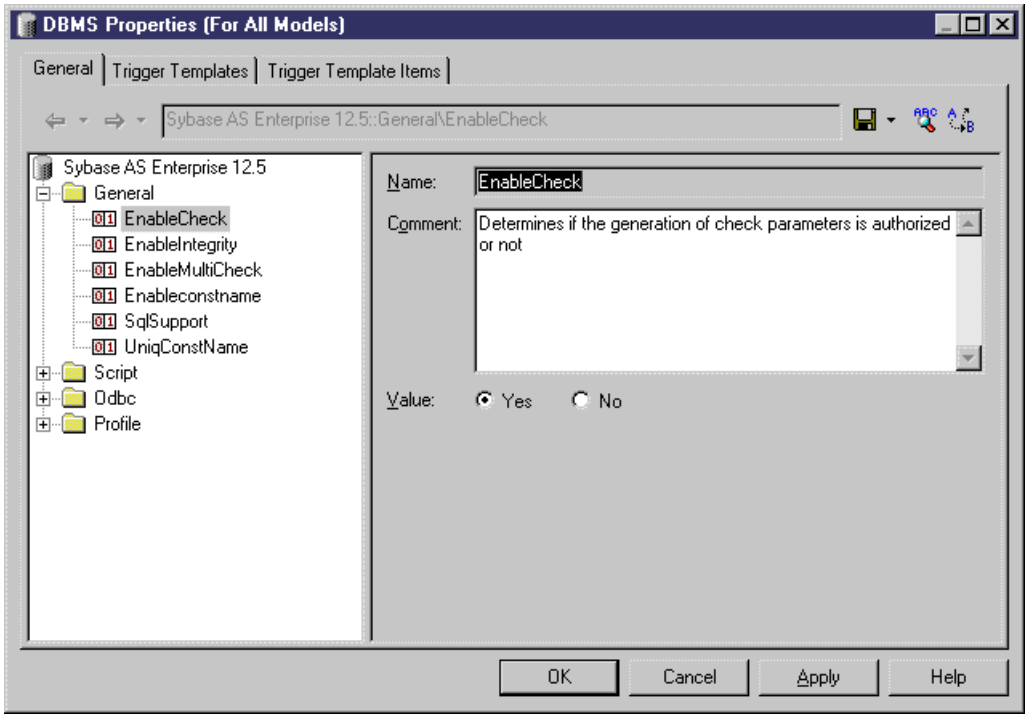
A DBMS definition file is a PowerDesigner resource file that provides PowerDesigner with the information necessary to model, reverse-engineer, and generate a particular DBMS.

PowerDesigner provides definition files for most popular DBMSs. The DBMS definition files are located in install_dir/Resource Files/DBMS, and have an .xdb extension.

**Note:** Modifications to a DBMS definition file can change the way PowerDesigner functions work, especially when generating scripts. Make sure you create backup copies of your database and thoroughly test generated scripts before executing them.

## Opening your Target DBMS Definition File in the Target Editor

You can consult or modify a DBMS definition file using the Resource Editor. When you select a *category* or an item in the left-hand pane, the name, value, and related comment appear in the right side of the dialog box.




Select **Database** > **Edit current DBMS**

The DBMS Properties dialog box is displayed.

**Note:** You should never modify the DBMS files shipped with PowerDesigner. You should instead copy the DBMS to create a new one. To do so, create a new DBMS from the List of DBMS, define a name, and then select the original file in the Copy From list.

For more information on using the editor, see *Working with the Resource Editor* on page 2.

## DBMS Definition File Structure

All DBMS definition files have the same structure made up of a number of categories, each of which may contain items or other categories. The items, and their values are different for each DBMS. Each item is present only if it is relevant to the DBMS. Each value is a SQL statement or other parameter to define how to model, generate and reverse engineer for the DBMS.

Each DBMS file has the following structure:

- *General* - contains general information about the database, without any categories (see *General Category* on page 49). All items defined in the General category apply to all database objects.
- *Script* - used for generation and reverse engineering. Contains the following sub-categories:
  - *SQL* - contains the following sub-categories, each of which contains items whose values define general syntax for the database:
    - *Syntax* - general parameters for SQL syntax (see *Syntax category* on page 50)
    - *Format* - parameters for allowed characters (see *Format category* on page 51)
    - *File* - header, footer and usage text items used during generation (see *File category* on page 52)
    - *Keywords* - the list of SQL reserved words and functions (see *Keywords category* on page 54)
  - *Objects* - contains commands to create, delete or modify all the objects in the database. Also includes commands that define object behavior, defaults, necessary SQL queries, reverse engineering options, and so on (see *Script/Objects Category* on page 55).
  - *Data Type* - contains the list of valid data types for the specified DBMS and the corresponding types in PowerDesigner (see *Script/Data Type Category* on page 96).
  - *Customize* - Retrieves information from PowerDesigner Version 6 DBMS definition files. It is not used in later versions.
- *ODBC* - present only if the DBMS does not support standard statements for generation. In this case the ODBC category contains additional items necessary for live database connection generation .
- *Transformation Profiles* – contains group of transformations used during model generation when you need to apply changes to objects in the source or target models. For more information, see *Transformations and Transformation Profiles (Profile)* on page 165 and "Applying Model Transformations" in the Models chapter of the *Core Features Guide*.
- *Profile* - allows you to define extended attribute types and extended attributes for database objects. For more information, see *Profile Category* on page 97.

### DBMS Property Page

A DBMS has a property page available when you click the root node in the tree view. The following properties are defined:

| Property | Description |
|---|---|
| Name | Name of the DBMS. This name must be unique in a model |
| Code | Code of the DBMS. This code must be unique in a model |
| File Name | [read only] Path and name of the DBMS file. |
| Family | Used to classify a DBMS, and to establish a link between different database resource files. For example, Sybase AS Anywhere, and Sybase AS Enterprise belong to the SQL Server family.<br>Triggers are retained when you change target within the same family.<br>Merge interface allows to merge models from the same family |
| Comment | Additional information about the DBMS |

### Triggers Templates, Trigger Template Items, and Procedure Templates

The DBMS Trigger templates, Trigger template items, and Procedure templates are accessible via the tabs in the Resource Editor window

Templates for stored procedures are defined under the Procedure category in the DBMS tree view.

For more information, see the Triggers and Procedures chapter in the *Data Modeling* guide.

# Managing Generation and Reverse Engineering

PowerDesigner supports reverse engineering and generation through both *scripts* and *live database* connections.

In this section:

- *Statement* is used to define a piece of SQL syntax. Statements often contain variables that will be evaluated during generation and script reverse engineering.
- *Query* is reserved for describing live database reverse engineering

Statements for script generation, script reverse engineering, and live database generation are identical, whereas live database reverse engineering may require specific queries.

The processes of generation and reverse-engineering can be defined as follows:

- *Generation* - statements are parsed and variables are evaluated and replaced by their actual values taken from the current model. The same statements are used for script and live database generation.
- *Reverse engineering* – may be performed by:
  - *Script* - PowerDesigner parses the script and identifies the different statements thanks to the terminator (defined in Script\Sql\Syntax). Each individual statement is "associated" with an existing statement in the DBMS definition file in order to commit the variables in the reversed statement as items in a PowerDesigner model.
  - *Live database connection* - special queries are used to retrieve information from the database system tables. Each column of a query result set is associated with a variable. The query header specifies the association between the columns of the resultset and the variable. The values of the returned records are stored in these variables which are then committed as object attributes.

For more information on variables, see *Optional strings and variables* on page 117.

## Script Category

The Script category contains the following kinds of items:

- *Generation and reverse engineering statements* - used for script and live database generation and script reverse engineering. For example, the standard statement for creating an index is:

```
create index %INDEX%
```

These statements differ from DBMS to DBMS. For example in Oracle 9i, the index create statement contains the definition of an owner:

```
create [%UNIQUE%?%UNIQUE% :[%INDEXTYPE% ]]index [%QUALIFIER%]%INDEX% on
[%CLUSTER%?cluster C_%TABLE%:[%TABLQUALIFIER%]%TABLE% (
 %CIDXLIST%
)]
[%OPTIONS%]
```

The following kinds of generation and reverse engineering statements are also available:

- Drop for deleting an object
- Options for defining the physical options of an object
- ConstName to define the constraint name template for object checks
- *Modify statements* - used to modify the attributes of already existing objects. Most start with the word "Modify", but others include Rename or AlterTableFooter.
  The statement for creating a *key* depending on where the key is defined. If the key is inside the table, then it will be created with a generation order, and if it is created outside the table, it will be a modify order of the table.
- *Database definition items* – used to customize the PowerDesigner interface and behavior according to database features. For example, item Maxlen in the table category, has to be set according to the maximum code length tolerated for a table in the current database.

Permission, EnableOwner, AllowedADT are other examples of items defined to adapt PowerDesigner to the current DBMS.

- *Live database reverse engineering queries* - most start with "Sql". For example, SqlListQuery retrieves a list of objects, and SqlOptsQuery reverse engineers physical options. For more information, see *Live database reverse engineering* on page 41.

## ODBC Category

The ODBC category contains items for live database generation when the DBMS does not support the generation statements defined in the Script category.

For example, data exchange between PowerDesigner and MSACCESS works with VB scprits and not SQL, this is the reason why these statements are located in the ODBC category. You have to use a special program (access.mdb) to convert these scripts into MSACCESS database objects.

## Script Generation

Script generation statements are available in the Script category, under the different object categories. For example, in Sybase ASA 8, the Create statement in the Table category is the following:

```
create table [%QUALIFIER%]%TABLE%
(
  %TABLDEFN%
)
[%OPTIONS%]
```

This statement contains the parameters for creating the table together with its owner and physical options.

### Extension Mechanism

You can extend script generation statements to complement generation using them *extension statements*. The extension mechanism allows you to generate statements immediately before or after Create, Drop, and Modify statements, and to retrieve these statements during reverse engineering.

For more information on reverse engineering additional statements see *Script reverse engineering* on page 40.

### Generation Template Language

Extension statements are defined using the PowerDesigner Generation Template Language (GTL) mechanism.

An extension statement can contain:

- Reference to other *statements* that will be evaluated during generation. These items are text items that must be defined in the object category of the extension statements
- *Variables* used to evaluate object properties and extended attributes. Variables are enclosed between % characters
- *Macros* such as ".if", provide generic programming structures for testing variables. Note: we recommend that you avoid using GTL macros in generation scripts, as they cannot be reconstituted when reverse engineering by script. Generating and reverse engineering via a live database connection are not subject to this limitation.

For more information on the PowerDesigner Generation Template Language (GTL), see *Customizing Generation with GTL* on page 187.

During generation, the statements and variables are evaluated and the result is added to the global script.

### Example 1

The extension statement *AfterCreate* is defined in the table category to complement the table Create statement by adding partitions to the table if the value of the partition extended attribute requires it.

AfterCreate is defined in GTL syntax as follows:

```
.if (%ExtTablePartition% > 1)
%CreatePartition%
```

```
go
.endif
```

The .if macro is used to evaluate variable %ExtTablePartition%. This variable is an extended attribute that contains the number of table partitions. If the value of %ExtTablePartition% is higher than 1, then %CreatePartition% will be generated followed by "go". %CreatePartition% is a statement defined in the table category as follows:

```
alter table [%QUALIFIER%]%TABLE%
 partition %ExtTablePartition%
```

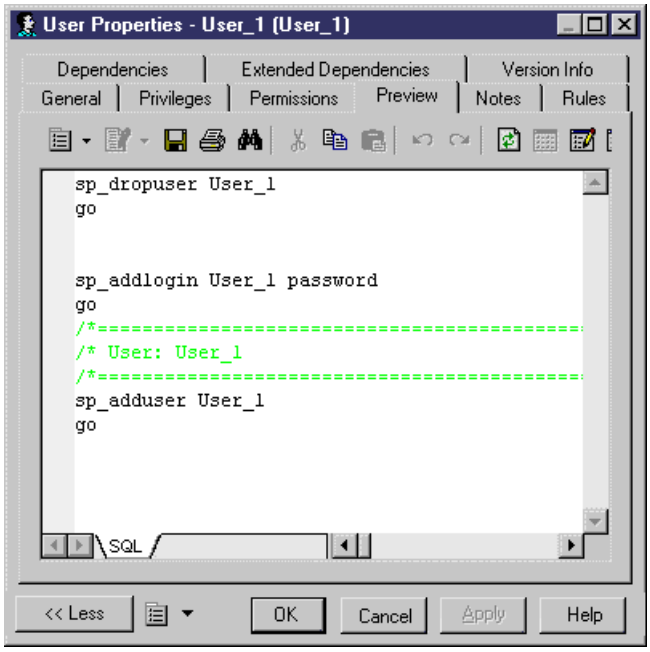
%CreatePartition% generates the statement for creating the number of table partitions specified in %ExtTablePartition %.

### Example 2

You create in Sybase ASE an extended statement to automatically create the login of a user before the Create user statement is executed. The BeforeCreate statement is the following:

```
sp_addlogin %Name% %Password%
go
```

The automatically generated login will have the same name as the user and its password. You can preview the statement in the user property sheet, the BeforeCreate statement is displayed before the user creation statement:




### Modify Statements

You can also add BeforeModify and AfterModify statements to standard *modify* statements.

Modify statements are executed to synchronize the database with the schema created in the PDM. By default, the modify database feature does not take into account extended attributes when it compares changes performed in the model from the last generation. You can bypass this rule by adding extended attributes in the *ModifiableAttributes* list item. Extended attributes defined in this list will be taken into account in the merge dialog box during database synchronization.

To detect that an extended attribute value has been modified you can use the following variables:

- %OLDOBJECT% to access an old value of the object
- %NEWOBJECT% to access a new value of the object

For example, you can verify that the value of the extended attribute ExtTablePartition has been modified using the following GTL syntax:

```
.if (%OLDOBJECT.ExtTablePartition% != %NEWOBJECT.ExtTablePartition%)
```

If the extended attribute value was changed, an extended statement will be generated to update the database. In the Sybase ASE syntax, the ModifyPartition extended statement is the following because in case of partition change you need to delete the previous partition and then recreate it:

```
.if (%OLDOBJECT.ExtTablePartition% != %NEWOBJECT.ExtTablePartition%)
 .if (%NEWOBJECT.ExtTablePartition% > 1)
  .if (%OLDOBJECT.ExtTablePartition% > 1)
%DropPartition%
  .endif
%CreatePartition%
 .else
%DropPartition%
 .endif
.endif
```

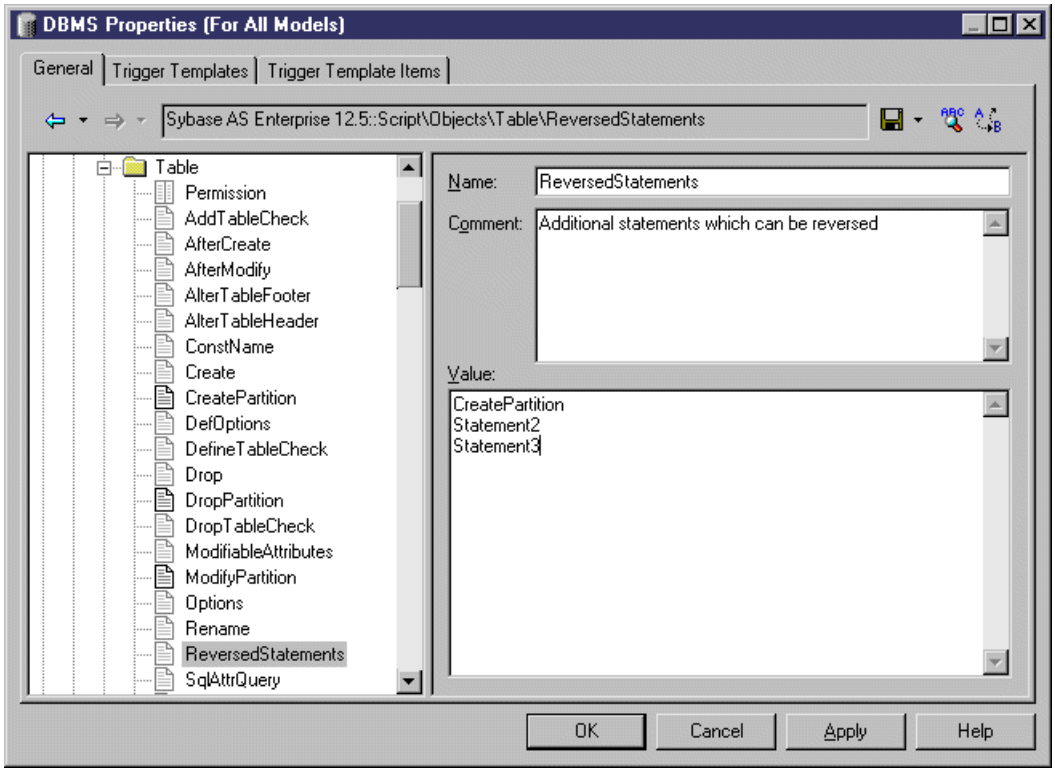
For more information on the PowerDesigner Generation Template Language (GTL), see *Customizing Generation with GTL* on page 187.

## Script Reverse Engineering

The same statements are used for generation and reverse engineering.

If you are using the extension mechanism for script generation, you have to declare statements in the list item *ReversedStatements* in order for them to be properly reversed. Type one statement per line in the ReversedStatement list.

For example, the extension statement AfterCreate uses statement CreatePartition. This text item must be declared in ReversedStatements to be properly reverse engineered. You could declare other statements in the following way:




## Live Database Generation

In general, live database generation uses the same statements as script generation. However, when the DBMS does not support standard SQL syntax, special generation statements are defined in the ODBC category. This is the case for MSACCESS that needs VB scripts to create database objects during live database generation.

These statements are defined in the ODBC category of the DBMS.

## Live Database Reverse Engineering

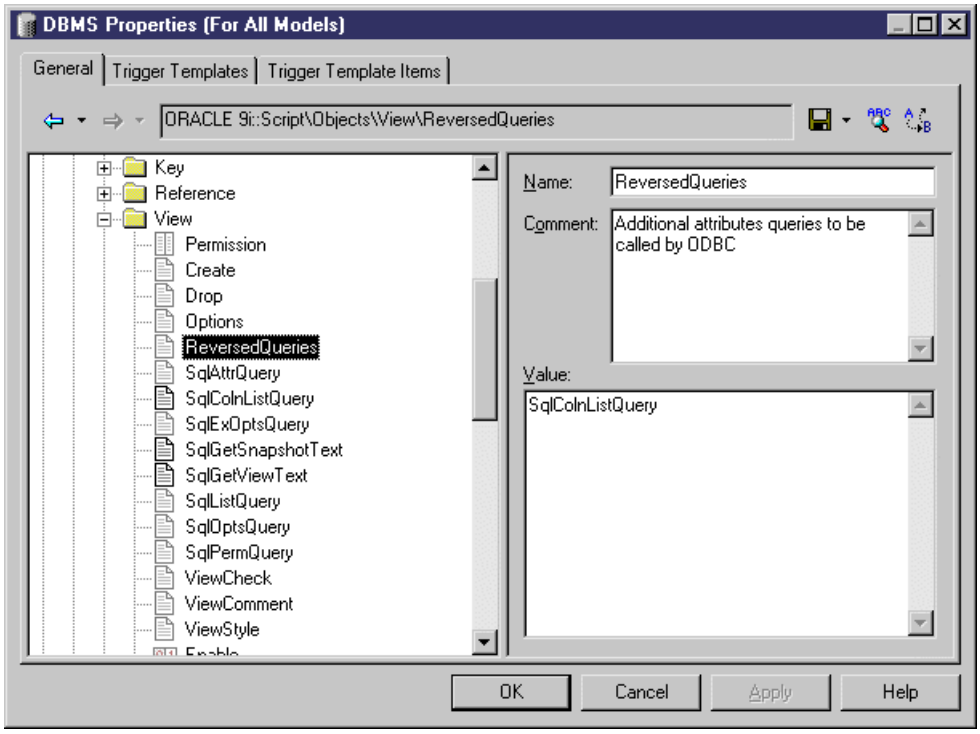
The DBMS contains live database reverse engineering queries for retrieving objects (like Table, Columns, and so on) from the database.

Most queries follow the same naming pattern "Sql...Query".

| Item | Description |
|---|---|
| SqlListQuery | Lists objects for selection in the Selection box. `SqlListQuery` retrieves objects and fills the reverse engineering window. Then, each of the other queries below are executed for each selected object.<br>If `SqlListQuery` is not defined, standard functions are used to retrieve objects. `SqlAttrQuery, SqlOptsQuery` etc. will then be executed, if defined.<br>`SqlListQuery` must retrieve the smallest number of columns possible as the process is memory intensive |
| SqlAttrQuery | Reverse engineers object attributes. `SqlAttrQuery` may be unnecessary if `SqlListQuery` can retrieve all necessary information. For example, in Sybase Adaptive Server® Anywhere 6, `TablespaceListQuery` is sufficient to retrieve all information required for use in a PDM |
| SqlOptsQuery | Reverse engineers physical options |
| SqlListChildrenQuery | Reverse engineers lists child objects, such as columns of a specific index or key, joins of a specific reference |
| SqlSysIndexQuery | Reverse engineers system indexes created by the database |
| SqlChckQuery | Reverse engineers object check constraints |
| SqlPermQuery | Reverse engineers object permissions |

You can define additional queries to recover more attributes during live database reverse engineering. This is to avoid loading *SqlListQuery* with queries for retrieving attributes not supported by *SqlAttrQuery*, or objects not selected for reverse engineering. These additional queries must be listed in the *ReversedQueries* item. For example, *SqlColnListQuery* is used to exclusively retrieve view columns. This query has to be declared in the ReversedQueries item in order to be taken into account during reverse engineering.

Note: extended queries should not be defined in the *ReversedQueries* item. For more information on ReversedQueries, see *Extension mechanism for live database reverse engineering queries* on page 43.




## Query Structure

Each column of a query result set is associated with a variable. A script header specifies the association between the columns of the result set and the variable. The values of the returned records are stored in these variables, which are then committed as object attribute values.

The script header is contained within curly brackets { }. The variables are listed within the brackets, each variable separated by a comma. There is a matching column for each variable in the Select statement that follows the header.

For example:

```
{OWNER, @OBJTCODE, SCRIPT, @OBJTLABL}
SELECT U.USER_NAME, P.PROC_NAME, P.PROC_DEFN, P.REMARKS
FROM SYSUSERPERMS U,SYSPROCEDURE P
WHERE [%SCHEMA% ? U.USER_NAME='%SCHEMA%' AND] P.CREATOR=U.USER_ID
ORDER BY U.USER_NAME
```

The variables can be any listed in *PDM Variables* on page 107.

Each comma-separated part of the header is associated with the following information:

- Name of variable (mandatory). See the example in *Processing with variable names*
- The `ID` keyword follows each variable name. ID means that the variable is part of the identifier
- The `...` (ellipsis) keyword means that the variable must be concatenated for all the lines returned by the SQL query and having the same values for the ID columns
- `Retrieved_value = PD.value` lists the association between a retrieved value and a PowerDesigner value. A conversion table converts each value of the record (system table) to another value ( in PowerDesigner). This mechanism is optionally used. See the example in *Processing with conversion table*

The only mandatory information is the variable name. All others are optional. The `ID` and `...` (ellipsis) keywords are mutually exclusive.

## Processing with Variable Names:

```
{TABLE ID, ISPKEY ID, CONSTNAME ID, COLUMNS ...}
select
 t.table_name,
 1,
```

```
 null,
 c.column_name + ', ',
 c.column_id
from
 systable t,
 syscolumn c
where
etc..
```

In this script, the identifier is defined as TABLE + ISKEY+ CONSTNAME.

In the result lines returned by the SQL script, the values of the fourth field is concatenated in the COLUMNS field as long as these ID values are identical.

```
SQL Result set
Table1,1,null,'col1,'
Table1,1,null,'col2,'
Table1,1,null,'col3,'
Table2,1,null,'col4,'
In PowerDesigner memory
Table1,1,null,'col1,col2,col3'
Table2,1,null,'col4'
```

In the example, COLUMNS will contain the list of columns separated by commas. PowerDesigner will process the contents of COLUMNS field to remove the last comma.

### Processing with Conversion Table:

The syntax inserted just behind a field inside the header is:

```
(SQL value1 = PowerDesigner value1, SQL value2 = PowerDesigner value2, * =
PowerDesigner value3)
```

where * means all other values.

For example:

```
{ADT, OWNER, TYPE(25=JAVA , 26=JAVA)}
SELECT t.type_name, u.user_name,  t.domain_id
FROM sysusertype t, sysuserperms u
WHERE [u.user_name = '%SCHEMA%' AND]
(domain_id = 25 OR domain_id = 26) AND
t.creator = u.user_id
```

In this example, when the SQL query returns the value 25 or 26, it is replaced by JAVA in TYPE variable.

### Extension Mechanism for Live Database Reverse Engineering Queries

During reverse engineering, PowerDesigner executes queries to retrieve information from the columns of the system tables. The result of the query is mapped to PowerDesigner internal variables via the query header. When the system tables of a DBMS store information in columns with LONG, BLOB, TEXT and other incompatible data types, PowerDesigner cannot concatenate these data into strings.

You can bypass this limitation by using the *EX* keyword and creating user-defined queries and variables in the existing reverse engineering queries with the syntax:

```
%UserDefinedQueryName.UserDefinedVariableName%
```

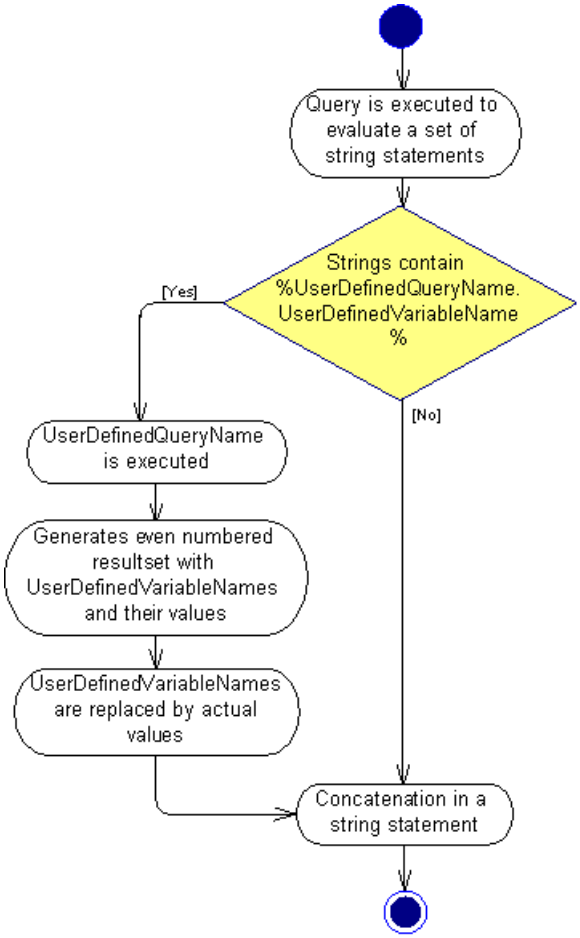
These user-defined variables will be evaluated by sub-queries which you write.

In the following example, the value of OPTIONS is marked as containing a user-defined query, and we see in the body of the query that the 'global partition by range' option contains a user-defined query called :'SqlPartIndexDef', which seeks values for the variables 'i.owner' and 'i.index_name':

```
{OWNER, TABLE, CONSTNAME, OPTIONS EX}

select
```

```
c.owner,
c.table_name,
c.constraint_name,
   ...
   'global partition by range
       (%SqlPartIndexDef.'||i.owner||i.index_name||'%)',
   ...
```

The following graphic illustrates the process of variable evaluation during reverse engineering:




**Note:** Extended queries should not be defined in the ReversedQueries item.

### Step1

A query is executed to evaluate variables in a set of string statements.

If the EX keyword is present in the query header, PowerDesigner searches for user-defined queries and variables to evaluate. These user-defined variables are created to be filled with data proceeding from columns with LONG/BLOB/TEXT... data type.

You can create user-defined queries in any live database reverse engineering query. Each query must have a unique name.

### Step 2

The execution of the user-defined query generates a resultset containing pairs of user-defined variable names (without %) and variable value for each of the variables as needed.

For example, in the following resultset, the query returns 3 rows and 4 columns by row:

| Variable 1 | 1 | Variable 2 | 2 |
|---|---|---|---|
| Variable 3 | 3 | Variable 4 | 4 |
| Variable 5 | 5 | Variable 6 | 6 |

### Step 3

These values replace the user-defined variables in the original query.

The following sections explain user-defined queries defined to address reverse engineering limitations.

### Live Database Reverse Engineering Physical Options

During reverse engineering, physical options are concatenated in a single string statement. However, when the system tables of a database are partitioned (like in Oracle) or fragmented (like in Informix), the partitions/fragments share the same logical attributes but their physical properties like storage specifications, are stored in each partition/fragment of the database. The columns in the partitions/fragments have a data type (LONG) that allows storing larger amount of unstructured binary information.

Since physical options in these columns cannot be concatenated in the string statement during reverse engineering, `SqlOptsQuery` (Tables category in the DBMS) contains a call to a user-defined query that will evaluate these physical options.

In Informix SQL 9, `SqlOptsQuery` is delivered by default with the following user-defined queries and variables (the following is a subset of `SqlOptsQuery`):

```
select
 t.owner,
 t.tabname,
 '%SqlFragQuery.FragSprt'||f.evalpos||'% %FragExpr'||f.evalpos||'% in
%FragDbsp'||f.evalpos||'% ',
 f.evalpos
from
 informix.systables t,
 informix.sysfragments f
where
 t.partnum = 0
 and t.tabid=f.tabid
[  and t.owner = '%SCHEMA%']
[  and t.tabname='%TABLE%']
```

After the execution of `SqlOptsQuery`, the user-defined query `SqlFragQuery` is executed to evaluate `FragDbsp` n, `FragExpr` n, and `FragSprt` n. n stands for `evalpos` which defines fragment position in the fragmentation list. n allows to assign unique names to variables, whatever the number of fragment defined in the table.

`FragDbsp` n, `FragExpr` n, and `FragSprt` n are user-defined variables that will be evaluated to recover information concerning the physical options of fragments in the database:

| User-defined variable | Physical options |
|---|---|
| FragDbsp n | Fragment location for fragment number n |
| FragExpr n | Fragment expression for fragment number n |
| FragSprt n | Fragment separator for fragment number n |

`SqlFragQuery` is defined as follows:

```
{A, a(E="expression", R="round robin", H="hash"), B, b, C, c, D, d(0="",
*=",")}
select
 'FragDbsp'||f.evalpos, f.dbspace,
```

```
 'FragExpr'||f.evalpos, f.exprtext,
 'FragSprt'||f.evalpos, f.evalpos
from
 informix.systables t,
 informix.sysfragments f
where
 t.partnum = 0
 and f.fragtype='T'
 and t.tabid=f.tabid
[  and t.owner = '%SCHEMA%']
[  and t.tabname='%TABLE%']
```

The header of `SqlFragQuery` contains the following variable names.

```
{A, a(E="expression", R="round robin", H="hash"), B, b, C, c, D, d(0="", *=",")}
```

Only the translation rules defined between brackets will be used during string concatenation: "FragSprt0", which contains 0 (f.evalpos), will be replaced by " ", and "FragSprt1", which contains 1, will be replaced by ","

`SqlFragQuery` generates a numbered resultset containing as many pairs of user-defined variable name (without %) and variable value as needed, if there are many variables to evaluate.

The user-defined variable names are replaced by their values in the string statement for the physical options of fragments in the database.

### Live Database Reverse Engineering Function-based Index

In Oracle 8i and later versions, you can create indexes based on functions and expressions that involve one or more columns in the table being indexed. A function-based index precomputes the value of the function or expression and stores it in the index. The function or the expression will replace the index column in the index definition.

An index column with an expression is stored in system tables with a LONG data type that cannot be concatenated in a string statement during reverse engineering.

To bypass this limitation, `SqlListQuery` (Index category in the DBMS) contains a call to the user-defined query `SqlExpression` used to recover the index expression in a column with the LONG data type and concatenate this value in a string statement (the following is a subset of `SqlListQuery`):

```
select
 '%SCHEMA%',
 i.table_name,
 i.index_name,
 decode(i.index_type, 'BITMAP', 'bitmap', ''),
 decode(substr(c.column_name, 1, 6), 'SYS_NC', '%SqlExpression.Xpr'||
i.table_name||i.index_name||c.column_position||'%', c.column_name)||' '||
c.descend||', ',
 c.column_position
from
 user_indexes i,
 user_ind_columns c
where
 c.table_name=i.table_name
 and c.index_name=i.index_name
[  and i.table_owner='%SCHEMA%']
[  and i.table_name='%TABLE%']
[  and i.index_name='%INDEX%']
```

The execution of `SqlListQuery` calls the execution of the user-defined query `SqlExpression`.

`SqlExpression` is followed by a user-defined variable defined as follow:

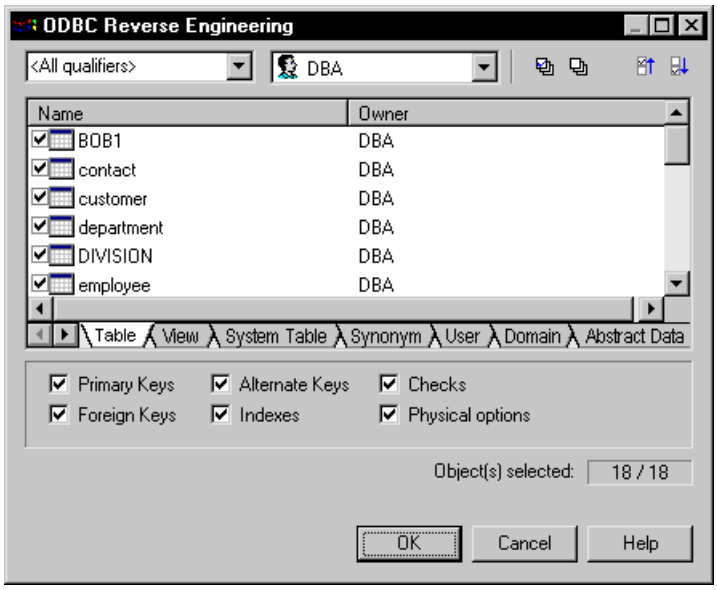
```
{VAR, VAL}

select
 'Xpr'||table_name||index_name||column_position,
```

```
 column_expression
from
 all_ind_expressions
where 1=1
[  and table_owner='%SCHEMA%']
[  and table_name='%TABLE%']
```

The name of the user-defined variable is unique, it is the result of the concatenation of "Xpr", table name, index name, and column position.

## Live Database Reverse Engineering Qualifiers

A qualifier allows the use of the object qualifier that is displayed in the dropdown list box in the upper left corner of the Database Reverse Engineering dialog box. You use a qualifier to select which objects are to be reverse engineered.




You can add a qualifier section when you customize your DBMS. This section must contain the following items:

- enable: YES/NO
- SqlListQuery (script) : this item contains the SQL query that is executed to retrieve the qualifier list. You should not add a Header to this query

The effect of these items are shown in the table below:

| Enable | SqlListQuery present? | Result |
|---|---|---|
| Yes | Yes | Qualifiers are available for selection. Select one as required. You can also type the name of a qualifier. SqlListQuery is executed to fill the qualifier list |
| | No | Only the default (All qualifiers) is selected. You can also type the name of a qualifier |
| No | No | Dropdown list box is grayed. |

For more information on qualifier filters, see "Filters and options for reverse engineering" in the Reverse Engineering chapter of the *Data Modeling* guide.

## Example

In Adaptive Server Anywhere 7, a typical qualifier query is:

```
.Qualifier.SqlListQuery :
select dbspace_name from sysfile
```

## Generating and Reverse Engineering Extended Objects

Some DBMSs have objects that cannot be represented by the standard PowerDesigner model objects. However, you can work with these objects, generate and reverse-engineer them through the use of extended objects. To do this you must first create an extended object, and then define its generation and reverse engineering scripts.

### To Create an Extended Object:

You can create extended objects in a DBMS.

1. Select **Database > Edit Current DBMS** to open the DBMS Properties window, and then expand the Profile category in the left-hand pane.
2. If there is not an entry for Extended Object in this category, then create one by right-clicking Profile and selecting Add Metaclasses from the contextual menu. In the Selection box, click the PdCommon sub-tab, select Extended Object and click OK to add it to the list of objects.
3. Right-click the Extended Object entry, and select **New > Stereotype** from the contextual menu to create a new stereotype, which will be used to define your new object.
4. Enter the name of your new object and select the Use as metaclass checkbox. This will ensure that the new object appears in the PowerDesigner menus and has its own special browser category.

   You can add attributes to the object, create templates to define its form for generation and reverse engineering, and produce custom forms for use in property sheets. For more information, see *Extending Your Models with Profiles* on page 121.

Once you have defined your object, you need to enable its generation.

### To Define Generation and Reverse Engineering Scripts for an Extended Object:

You can define generation and reverse engineering scripts for an extended object

1. Right-click the Script/Objects category, and then select Add Items from the contextual menu to open a Selection dialog that lists all the objects available in the model.
2. Select your new extended object in the list, and then click OK to add it to the list of objects.
3. Right-click the new object entry, and then select Add Items from the contextual menu to open a Selection dialog that lists all the script items that can be added to an extended object.
4. As a minimum, to enable the generation and reverse engineering of the object, you should select the following items:
   - Create
   - Drop
   - AlterStatementList
   - SqlAttrQuery
   - SqlListQuery
5. Click OK to add these script items to your object. You will need to enter values for each of these items. For more information, and guidance on syntax, see *Common object items* on page 56.
6. Your object will now be available for generation and reverse engineering. You can also control the order in which this and the other objects will be generated. For more information, see *GenerationOrder – customizing the order in which objects are generated* on page 55.

## Adding Scripts Before or After Generation and Reverse Engineering

You can specify scripts to be used before or after database generation or reverse engineering.

1. Open the Profile folder. If there is no entry for Model, then right-click the Profile folder and select Add Metaclasses from the contextual menu to open the Metaclass Selection dialog box.

2. On the PdPDM sub-tab, select Model and then click OK to return to the DBMS properties editor. The Model item now appears in the Profile folder.
3. Right-click the Model item, and select **New > Event Handler** from the contextual menu to open a Selection dialog box.
4. Select one or more of the following event handlers depending on where you want to add a script:
   - BeforeDatabaseGenerate
   - AfterDatabaseGenerate
   - BeforeDatabaseReverseEngineer
   - AfterDatabaseReverseEngineer
5. Click OK to return to the DBMS properties editor. The selected event handlers now appear beneath the Model item.
6. Select each of the event handlers in turn, click its Event Handler Script tab, and enter the desired script.
7. Click OK to confirm your changes and return to the model.

## General Category

The General category is located directly beneath root, and contains the following items:

| Item | Description |
|---|---|
| EnableCheck | Specifies whether the generation of check parameters is authorized. The following settings are available:<br>• Yes - Check parameters generated<br>• No - All variables linked to Check parameters will not be evaluated during generation and reverse |
| Enable Constname | Specifies whether constraint names are used during generation. The following settings are available:<br>• Yes - Constraint names are used during generation<br>• No - Constraint names are not used |
| EnableIntegrity | Specifies whether there are integrity constraints in the DBMS. The following settings are available:<br>• Yes - Primary, alternate, and foreign key check boxes are available for database generation and modification<br>• No - Primary, alternate, and foreign key check boxes are not available |
| EnableMulti Check | Specifies whether the generation of multiple check parameters for tables and columns is authorized. The following settings are available:<br>• Yes - Multiple check parameters are generated. The first constraint in the script corresponds to the concatenation of all validation business rules, the other constraints correspond to each constraint business rules attached to an object<br>• No - All business rules (validation and constraint) are concatenated into a single constraint expression |
| SqlSupport | Specifies whether SQL syntax is allowed. The following settings are available:<br>• Yes - SQL syntax allowed and SQL Preview available<br>• No - SQL syntax not allowed. SQL Preview is not available |

| Item | Description |
|---|---|
| UniqConst Name | Specifies whether unique constraint names for objects are authorized . The following settings are available:<br>• Yes - All constraint names (including index names) must be unique in the database<br>• No - Constraint names must be unique for an object<br>Check model takes this item into account in constraint name checking. |

## Script/Sql Category

The SQL category is located in the **Root** > **Script** category. Its sub-categories define the SQL syntax for the DBMS

### Syntax Category

The Syntax category is located in the **Root** > **Script** > **SQL** category, and contains the following items that define the DBMS-specific syntax:

| Item | Description |
|---|---|
| BlockComment | Specifies the character used to enclose a multi-line commentary.<br>Example:<br>`/* */` |
| Block Terminator | Specifies the end of block character, which is used to end expressions for triggers and stored procedures. |
| Delimiter | Specifies the field separation character. |
| Identifier Delimiter | Specifies the identifier delimiter character. When the beginning and end delimiters are different, they must be separated by a space character. |
| LineComment | Specifies the character used to enclose a single line commentary.<br>Example:<br>`%%` |
| Quote | Specifies the character used to enclose string values.<br>Note that the same quote must be used in the check parameter tab to enclose reserved words used as default. |
| SqlContinue | Specifies the continuation character. Some databases require a continuation character when a statement is longer than a single line. For the correct character, refer to your DBMS documentation. This character is attached to each line just prior to the linefeed. |
| Terminator | Specifies the end of statement character, which is used to terminate create table, view, index, or the open/close database, and other statements.<br>If empty, `BlockTerminator` is used instead. |
| UseBlockTerm | Specifies the use of `BlockTerminator`. The following settings are available:<br>• Yes - `BlockTerminator` is always used<br>• No - `BlockTerminator` is used for triggers and stored procedures only |

## Format Category

The Format category is located in the **Root > Script > SQL** category, and contains the following items that define script formatting:

| Item | Description |
|---|---|
| AddQuote | Specifies that object codes are systematically enquoted during the generation. The following settings are available:<br>• Yes – Quotes are systematically added to object codes during generation<br>• No - Object codes are generated without quotes |
| CaseSensitivity Using-Quote | Specifies if the case sensitivity for identifiers is managed using double quotes. You should set this boolean to Yes if the DBMS you are using needs double quotes to preserve the case of object codes. |
| Date and Time formats | See *Date and time format* on page 51. |
| EnableOwner Prefix / EnableDtbs Prefix | Specifies that object codes can be prefixed by the object owner, the database name, or both, using the %QUALIFIER% variable. The following settings are available:<br>• Yes – enables the Owner Prefix and/or Database Prefix check boxes in the Database Generation box. Select one or both of these options to prefix objects. If you select both, the owner and database are concatenated when %QUALIFIER% is evaluated.<br>• No - The Owner Prefix and Database Prefix options are unavailable |
| IllegalChar | [generation only] Specifies invalid characters for names. If there is an illegal character in a Code, the code is set between quotes during generation.<br>Example:<br>`+-*/!=<>'"()`<br>If the name of the table is "SALES+PROFITS", the generated create statement will be:<br>`CREATE TABLE "SALES+PROFITS"`<br>Double quotes are placed around the table name to indicate that an invalid character is used. During reverse engineering, any illegal character is considered as a separator unless it is located within a quoted name. |
| LowerCase Only | When generating a script, all objects are generated in lowercase independently of the model Naming Conventions and the PDM codes. The following settings are available:<br>• Yes - Forces all generated script characters to lowercase<br>• No - Generates all script unchanged from the way objects are written in the model |
| MaxScriptLen | Specifies the maximum length of a script line. |
| UpperCase Only | When generating a script, all objects are generated in uppercase independently of the model Naming Conventions and the PDM codes. The following settings are available:<br>• Yes - Forces all generated script characters to uppercase<br>• No - Generates all script unchanged from the way objects are written in the model<br>Note that the `UpperCaseOnly` and `LowerCaseOnly` items are mutually exclusive. In the event that both items are enabled, the script is generated in *lowercase*. |

### Date and Time Format

You can customize the date and time format for test data generation to a script or live database connection using DBMS items in the Format category.

PowerDesigner uses the `PhysDataType` map item in the script\data types category to convert the physical data types of columns to conceptual data types because the DBMS items are linked with conceptual data types.

Example for Sybase AS Anywhere 7:

| Physical data type | Conceptual data type | DBMS entry used for SQL | DBMS entry used for live connection |
|---|---|---|---|
| datetime | DT | DateTimeFormat | OdbcDateTimeFormat |
| timestamp | TS | DateTimeFormat | OdbcDateTimeFormat |
| date | D | DateFormat | OdbcDateFormat |
| time | T | TimeFormat | OdbcTimeFormat |

If you want to customize the date and time format of your test data generation, you have to verify the data type of the columns in your DBMS, then find the corresponding conceptual data type in order to know which item to customize in your DBMS. For example, if the columns use the datetime data type in your model, you should customize the DateTimeFormat item in your DBMS.

The default date and time format is the following:

- SQL: `'yyyy-mm-dd HH:MM:SS'`
- Live connection: `{ts 'yyyy-mm-dd HH:MM:SS'}`

Where:

| Format | Description |
|---|---|
| yyyy | Year on 4 digits |
| yy | Year on 2 digits |
| mm | Month |
| dd | Day |
| HH | Hour |
| MM | Minute |
| SS | Second |

For example, you can define the following value for the DateTimeFormat item for SQL: `yy-mm-dd HH:MM.` For live database connections, this item should have the following value: `{ts 'yy-mm-dd HH:MM'}.`

## File Category

The File category is located in the **Root** > **Script** > **SQL** category, and contains the following items that define script formatting:

| Item | Description |
|---|---|
| AlterHeader | Specifies header text for a modify database script. |
| AlterFooter | Specifies footer text for a modify database script. |

| Item | Description |
|---|---|
| EnableMulti File | Specifies that multiple scripts are allowed. The following settings are available:<br>• Yes – enables the One File Only check box in the Generate database, Generate Triggers and Procedures, and Modify Database parameters windows. If you deselect this option, a separate script is created for each table (named after the table, and with the extension defined in the `TableExt` item), and a global script summarizes all the single table script items.<br>• The One File Only check box is unavailable, and a single script includes all the statements.<br>The file name of the global script is customizable in the File Name field of the generation or modification windows and has the extension specified in the `ScriptExt` item.<br>The default name for the global script is CREBAS for database generation, CRETRG for triggers and stored procedures generation, and ALTER for database modification. |
| Footer | Specifies the text for the database generation script footer. |
| Header | Specifies the text for the database generation script header. |
| ScriptExt | Specifies the default script extension when you generate a database or modify a database for the first time.<br>Example:<br>`sql` |
| StartCommand | Specifies the statement for executing a script. Used inside the header file of a multi-file generation to call all the other generated files from the header file.<br>Example (Sybase ASE 11):<br>`isql %NAMESCRIPT%`<br>Corresponds to the %STARTCMD% variable (see *PDM Variables* on page 107). |
| TableExt | Specifies the extension of the scripts used to generate each table when the EnableMultiFile item is enabled and the "One File Only" check box is not selected in the Generate or Modify windows.<br>Example:<br>`sql` |
| TrgFooter | Specifies footer text for a triggers and procedures generation script. |
| TrgHeader | Header script for triggers and procedures generation. |
| TrgUsage1 | [when using a single script] Specifies text to display in the Output window at the end of trigger and procedure generation. |
| TrgUsage2 | [when using multiple scripts] Specifies text to display in the Output window at the end of trigger and procedure generation. |
| TriggerExt | Specifies the main script extension when you generate triggers and stored procedures for the first time.<br>Example:<br>`trg` |
| Usage1 | [when using a single script] Specifies text to display in the Output window at the end of database generation. |
| Usage2 | [when using multiple scripts] Specifies text to display in the Output window at the end of database generation. |

## Keywords Category

The Keywords category is located in the **Root > Script > SQL** category, and contains the following items that reserve keywords.

The lists of SQL functions and operators are used to populate the PowerDesigner SQL editor to propose lists of available functions to help in entering SQL code.

| Item | Description |
|---|---|
| CharFunc | Specifies a list of SQL functions to use with characters and strings.<br>Example:<br>`char()`<br>`charindex()`<br>`char_length() etc` |
| Commit | Specifies a statement for validating the transaction by live connection. |
| ConvertFunc | Specifies a list of SQL functions to use when converting values between hex and integer and handling strings.<br>Example:<br>`convert()`<br>`hextoint()`<br>`inttohex() etc` |
| DateFunc | Specifies a list of SQL functions to use with dates.<br>Example:<br>`dateadd()`<br>`datediff()`<br>`datename() etc` |
| GroupFunc | Specifies a list of SQL functions to use with group keywords.<br>Example:<br>`avg()`<br>`count()`<br>`max() etc` |
| ListOperators | Specifies a list of SQL operators to use when comparing values, boolean, and various semantic operators.<br>Example:<br>`=`<br>`!=`<br>`not like etc` |
| NumberFunc | Specifies a list of SQL functions to use with numbers.<br>Example:<br>`abs()`<br>`acos()`<br>`asin() etc` |

| Item | Description |
|---|---|
| OtherFunc | Specifies a list of SQL functions to use when estimating, concatenating and SQL checks.<br>Example:<br>`db_id()`<br>`db_name()`<br>`host_id() etc` |
| Reserved Default | Specifies a list of keywords that may be used as default values. If a reserved word is used as a default value, it will not be enquoted.<br>Example (SQL Anywhere® 10) - USER is a reserved default value:<br>`Create table CUSTOMER (`<br>`Username varchar(30) default USER`<br>`)`<br>When you run this script, CURRENT DATE is recognized as a reserved default value. |
| ReservedWord | Specifies a list of reserved keywords. If a reserved word is used as an object code, it is enquoted during generation (using quotes only in **DBMS** > **Script** > **SQL** > **Syntax** > > **Quote** ). |

# Script/Objects Category

The Objects category is located in the **Root** > **Script** > **SQL** category (and, possibly within **Root** > **ODBC** > **SQL** ), and contains the following items that define the database objects that will be available in your model.

## Commands for All Objects

The following commands are located in the **Root** > **Script** > **Objects** and **Root** > **ODBC** > **Objects** categories, and apply to all objects.

### MaxConstLen – Defining a Maximum Constraint Name Length

Command for defining the maximum constraint name length supported by the target database for tables, columns, primary and foreign keys. This value is implemented in the Check model and produces an error if the code exceeds the defined value. The constraint name is also truncated at generation time.

**Note:** PowerDesigner has a maximum length of 254 characters for constraint names. If your database supports longer constraint names, you must define the constraint names to fit in 254 characters or less.

### EnableOption – Enabling Physical Options

Command for enabling physical options for the model, tables, indexes, alternate keys, and other objects that are supported by the target DBMS. It also controls the availability of the Options tab from an object property sheet.

The following settings are available:

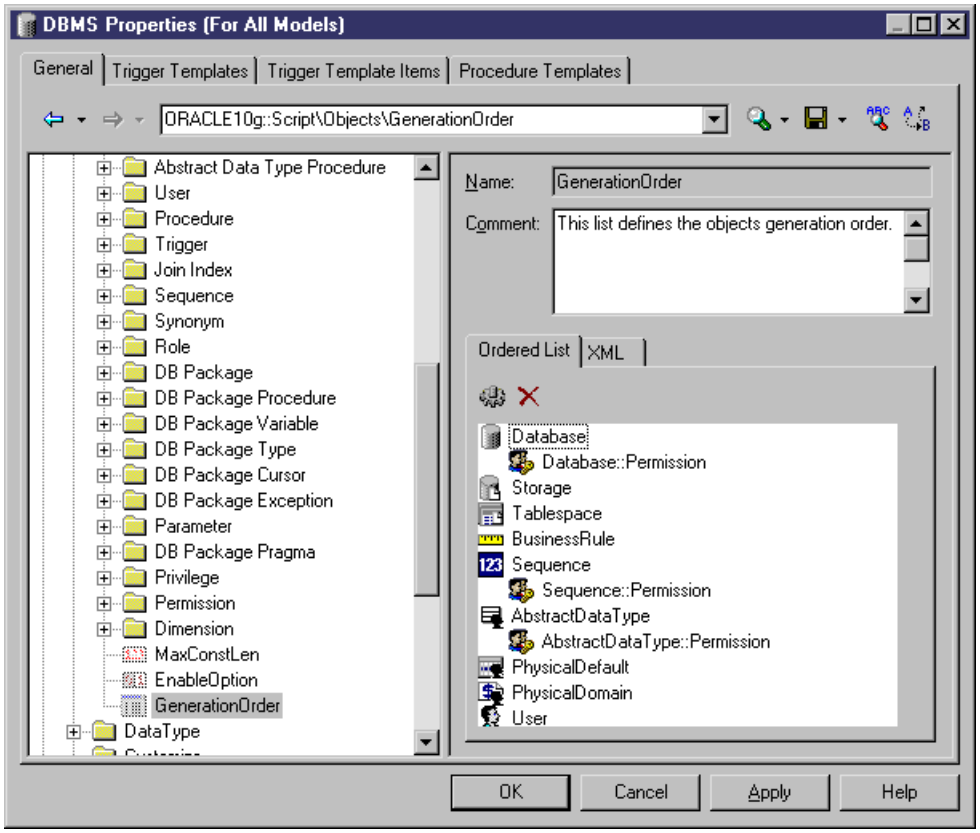
- Yes - The Physical Options tabs are available from the object property sheet
- No - Physical Options tabs are not available from the object property sheet.

For more information, see *Physical Options* on page 101

### GenerationOrder – Customizing the Order in Which Objects Are Generated

Command for specifying the generation order of objects. Disabled by default.

1. Right-click the Script/Objects node and select Add Items from the contextual menu to open a selection window listing all the available objects.
2. Select the GenerationOrder checkbox and click OK. The GenerationOrder command is enabled and added at the foot of the Objects category list.
3. Click the GenerationOrder item to display its properties:




4. You can drag and drop entries in the Ordered List tab to adjust the order in which objects will be created.
5. Note that not all object types are included in this list by default. You can add and remove items to and from the list using the tools on the tab. If an object does not appear on the list, it will still be generated, but after all the other objects. Sub-objects, such as "Sequence::Permissions", can be placed directly below their parent object in the list (where they will be indented to demonstrate their parentage) or separately, in which case they will be displayed without indentation.
6. Click OK to confirm your changes and return to the model.

**Note:** By default, extended objects (see *Generating and reverse engineering extended objects* on page 48) are not automatically included in this list, and are generated after all other objects. To promote these objects in the generation order, simply add them to the list with the tab tools, and then place them in the desired generation position.

## Common Object Items

The following items are available in various objects located in the **Root** > **Script** > **Objects** category.

| Item | Description |
|---|---|
| Add | Specifies the statement required to add the object inside the creation statement of another object.<br>Example (adding a column):<br>`%20:COLUMN% %30:DATATYPE% [default %DEFAULT%] [%IDENTITY%? identity:[%NULL%][%NOTNULL%]] [[constraint %CONSTNAME%] check (%CONSTRAINT%)]` |

| Item | Description |
|---|---|
| AfterCreate/ AfterDrop/ AfterModify | Specifies extended statements executed after the main Create, Drop or Modify statements. For more information, see *Script generation* on page 38. |
| Alter | Specifies the statement required to alter the object. |
| AlterDBIgnored | Specifies a list of attributes that should be ignored when performing a comparison before launching an update database. |
| AlterStatementList | Specifies a list of attributes which, when changed, should give rise to an alter statement. Each attribute in the list is mapped to the alter statement that should be used. |
| BeforeCreate/ BeforeDrop/ BeforeModify | Specifies extended statements executed before the main Create, Drop or Modify statements. For more information, see *Script generation* on page 38. |
| ConstName | Specifies a constraint name template for the object. The template controls how the name of the object will be generated.<br>The template applies to all the objects of this type for which you have not defined an individual constraint name. The constraint name that will be applied to an object is displayed in its property sheet.<br>Examples (ASE 15):<br>• Table: CKT_%.U26:TABLE%<br>• Column: CKC_%.U17:COLUMN%_%.U8:TABLE%<br>• Primary Key: PK_%.U27:TABLE% |
| Create | [generation and reverse] Specifies the statement required to create the object.<br>Example:<br>`create table %TABLE%` |
| DefOptions | Specifies default values for physical options that will be applied to all objects. These values must respect SQL syntax.<br>Example:<br>`in default_tablespace`<br>For more information, see *Physical Options* on page 101. |
| Drop | Specifies the statement required to drop the object.<br>Example (SQL Anywhere 10):<br>`if exists( select 1 from sys.systable`<br>`  where table_name=%.q:TABLE%`<br>`  and table_type in ('BASE', 'GBL TEMP')[%QUALIFIER%?`<br>`  and creator=user_id(%.q:OWNER%)]`<br>`) then drop table [%QUALIFIER%]%TABLE%`<br>`end if` |
| Enable | Specifies whether an object is supported. |

| Item | Description |
|---|---|
| EnableOwner | Enables the definition of owners for the object. The object owner can differ from the owner of the parent table. The following settings are available:<br>• Yes - The Owner list is enabled in the object's property sheet.<br>• No – Owners are not supported for the object.<br>Note that, in the case of index owners, you must ensure that the Create statement takes into account the table and index owner. For example, in Oracle 9i, the Create statement of an index is the following:<br>`create [%UNIQUE%?%UNIQUE% :[%INDEXTYPE% ]]index [%QUALIFIER%]%INDEX% on [%CLUSTER%?cluster C_%TABLE%:[%TABLQUALIFIER%]%TABLE% (`<br>` %CIDXLIST%`<br>`)]`<br>`[%OPTIONS%]`<br>Where %QUALIFIER% refers to the current object (index) and %TABLQUALIFIER% refers to the parent table of the index. |
| EnableSynonym | Enables support for synonyms on the object. |
| Footer | Specifies the object footer. The contents are inserted directly after each `create object` statement. |
| Header | Specifies the object header. The contents are inserted directly before each `create object` statement. |
| MaxConstLen | Specifies the maximum constraint name length supported for the object in the target database, where this value differs from the default. See also *MaxConstLen – defining a maximum constraint name length* on page 55). |
| MaxLen | Specifies the maximum code length for an object. This value is used when checking the model and produces an error if the code exceeds the defined value. The object code is also truncated at generation time. |
| Modifiable Attributes | Specifies a list of extended attributes that will be taken into account in the merge dialog during database synchronization. For more information, see *Script generation* on page 38.<br>Example (ASE 12.5):<br>`ExtTablePartition` |
| Options | Specifies physical options for creating an object.<br>Example (ASA 6):<br>`in %s : category=tablespace`<br>For more information, see *Physical Options* on page 101. |
| Permission | Specifies a list of available permissions for the object. The first column is the SQL name of permission (SELECT for example), and the second column is the shortname that is displayed in the title of grid columns.<br>Example (table permissions in ASE 15):<br>`SELECT / Sel`<br>`INSER / Ins`<br>`DELETE / Del`<br>`UPDATE / Upd`<br>`REFERENCES / Ref` |
| Reversed Queries | Specifies a list of additional attribute queries to be called during live database reverse engineering. For more information, see *Live database reverse engineering* on page 41. |

| Item | Description |
|---|---|
| Reversed Statements | Specifies a list of additional statements that will be reverse engineered. For more information, see *Script reverse engineering* on page 40. |
| SqlAttrQuery | Specifies a SQL query to retrieve additional information on objects reversed by `SQLList-Query`.<br>Example (Join Index in Oracle 10g):<br>`{OWNER ID, JIDX ID,  JIDXWHERE ...}`<br>`select index_owner, index_name,`<br>`outer_table_owner \|\| '.' \|\| outer_table_name \|\| '.' \|\| out-`<br>`er_table_column \|\| '=' \|\| inner_table_owner \|\| '.' \|\| in-`<br>`ner_table_name \|\| '.' \|\| inner_table_column \|\| ','`<br>`from all_join_ind_columns`<br>`where 1=1`<br>`[  and index_owner=%.q:OWNER%]`<br>`[  and index_name=%.q:JIDX%]` |
| SqlListQuery | Specifies a SQL query for listing objects in the reverse engineering dialog. The query is executed to fill header variables and create objects in memory.<br>Example (Dimension in Oracle 10g):<br>`{ OWNER, DIMENSION }`<br>`select d.owner, d.dimension_name`<br>`from sys.all_dimensions d`<br>`where 1=1`<br>`[  and d.dimension_name=%.q:DIMENSION%]`<br>`[  and d.owner=%.q:SCHEMA%]`<br>`order by d.owner, d.dimension_name` |
| SqlOptsQuery | Specifies a SQL query to retrieve physical options from objects reversed by `SqlListQuery`. The result of the query will fill the variable %OPTIONS% and must respect SQL syntax.<br>Example (Table in SQL Anywhere 10):<br>`{OWNER, TABLE, OPTIONS}`<br>`select u.user_name, t.table_name,`<br>`' in '+ f.dbspace_name`<br>`from sys.sysuserperms u`<br>`join sys.systab t on (t.creator = u.user_id)`<br>`join sys.sysfile f on (f.file_id = t.file_id)`<br>`where f.dbspace_name <> 'SYSTEM'`<br>`and t.table_type in (1, 3, 4)`<br>`[  and t.table_name = %.q:TABLE%]`<br>`[  and u.user_name = %.q:OWNER%]` |
| SqlPermQuery | Specifies a SQL query to reverse engineer permissions granted on the object.<br>Example (Procedure in SQL Anywhere 10):<br>`{ GRANTEE, PERMISSION}`<br>`select`<br>`u.user_name grantee, 'EXECUTE'`<br>`from sysuserperms u, sysprocedure s, sysprocperm p`<br>`where (s.proc_name = %.q:PROC% ) and`<br>`(s.proc_id = p.proc_id) and`<br>`(u.user_id = p.grantee)` |

### Default Variable

In a column, if the type of the default variable is text or string, the query must retrieve the value of the default variable between quotes. Most DBMS automatically add these quotes to the value of the default variable. If the DBMS you are using does not add quotes automatically, you have to specify it in the different queries using the default variable.

For example, in IBM DB2 UDB 8 for OS/390, the following line has been added in SqlListQuery in order to add quotes to the value of the default variable:

```
...
 case(default) when '1' then '''' concat defaultvalue concat '''' when '5'
then '''' concat defaultvalue concat '''' else defaultvalue end,
...
```

## Table

The Table category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how tables are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for tables:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• ConstName<br>• Create, Drop<br>• Enable, EnableSynonym<br>• Header, Footer<br>• Maxlen, MaxConstLen<br>• ModifiableAttributes<br>• Options, DefOptions<br>• Permission<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| AddTableCheck | Specifies a statement for customizing the script to modify the table constraints within an `alter table` statement.<br>Example (SQL Anywhere 10):<br>`alter table [%QUALIFIER%]%TABLE%`<br>`  add [constraint %CONSTNAME% ]check (%.A:CONSTRAINT%)` |
| AllowedADT | Specifies a list of abstract data types on which a table can be based. This list populates the Based On field of the table property sheet.<br>You can assign an abstract data type to a table, the table will use the properties of the type and the type attributes become table columns.<br>Example (Oracle 10g):<br>`OBJECT` |
| AlterTable Footer | Specifies a statement to be placed after `alter table` statements (and before the terminator).<br>Example:<br>`AlterTableFooter = /* End of alter statement */` |

| Item | Description |
|---|---|
| AlterTable Header | Specifies a statement to be placed before `alter table` statements. You can place an alter table header in your scripts to document or perform initialization logic.<br>Example:<br>`AlterTableHeader = /* Table name: %TABLE% */` |
| DefineTable Check | Specifies a statement for customizing the script of table constraints (checks) within a `create table` statement.<br>Example:<br>`check (%CONSTRAINT%)` |
| DropTable Check | Specifies a statement for dropping a table check in an `alter table` statement.<br>Example:<br>`alter table [%QUALIFIER%]%TABLE%`<br>`  delete check` |
| InsertIdentityOff | Specifies a statement for enabling insertion of data into a table containing an identity column.<br>Example (ASE 15):<br>`set identity_insert [%QUALIFIER%]%@OBJTCODE% off` |
| InsertIdentityOn | Specifies a statement for disabling insertion of data into a table containing an identity column.<br>Example (ASE 15):<br>`set identity_insert [%QUALIFIER%]%@OBJTCODE% on` |
| Rename | [modify] Specifies a statement for renaming a table. If not specified, the modify database process drops the foreign key constraints, creates a new table with the new name, inserts the rows from the old table in the new table, and creates the indexes and constraints on the new table using temporary tables.<br>Example (Oracle 10g):<br>`rename %OLDTABL% to %NEWTABL%`<br>The %OLDTABL% variable is the code of the table before renaming, and the %NEWTABL% variable is the new code. |
| SqlChckQuery | Specifies a SQL query to reverse engineer table checks.<br>Example (SQL Anywhere 10):<br>`{OWNER, TABLE, CONSTNAME, CONSTRAINT}`<br>`select u.user_name, t.table_name,`<br>`  k.constraint_name,`<br>`  case(lcase(left(h.check_defn, 5))) when 'check' then sub-`<br>`string(h.check_defn, 6) else h.check_defn end`<br>`from sys.sysconstraint k`<br>`  join sys.syscheck h on (h.check_id = k.constraint_id)`<br>`  join sys.systab t on (t.object_id = k.table_object_id)`<br>`  join sys.sysuserperms u on (u.user_id = t.creator)`<br>`where k.constraint_type = 'T'`<br>`  and t.table_type in (1, 3, 4)`<br>`[  and u.user_name = %.q:OWNER%]`<br>`[  and t.table_name = %.q:TABLE%]`<br>`order by 1, 2, 3` |

| Item | Description |
|---|---|
| SqlListRefr Tables | Specifies a SQL query used to list the tables referenced by a table.<br>Example (Oracle 10g):<br>`{OWNER, TABLE, POWNER, PARENT}`<br>`select c.owner, c.table_name, r.owner,`<br>`r.table_name`<br>`from sys.all_constraints c,`<br>`sys.all_constraints r`<br>`where (c.constraint_type = 'R' and c.r_constraint_name =`<br>`r.constraint_name and c.r_owner = r.owner)`<br>`[ and c.owner = %.q:SCHEMA%]`<br>`[ and c.table_name = %.q:TABLE%]`<br>`union select c.owner, c.table_name,`<br>`r.owner, r.table_name`<br>`from sys.all_constraints c,`<br>`sys.all_constraints r`<br>`where (r.constraint_type = 'R' and r.r_constraint_name =`<br>`c.constraint_name and r.r_owner = c.owner)`<br>`[ and c.owner = %.q:SCHEMA%]`<br>`[ and c.table_name = %.q:TABLE%]` |
| SqlListSchema | Specifies a query used to retrieve registered schemas in the database. This item is used with tables of XML type (a reference to an XML document stored in the database).<br>When you define an XML table, you need to retrieve the XML documents registered in the database in order to assign one document to the table, this is done using the SqlListSchema query.<br>Example (Oracle 10g):<br>`SELECT schema_url FROM dba_xml_schemas` |
| SqlStatistics | Specifies a SQL query to reverse engineer column and table statistics. See SqlStatistics in *Column* on page 63. |
| SqlXMLTable | Specifies a sub-query used to improve the performance of SqlAttrQuery (see *Common object items* on page 56). |
| TableComment | [generation and reverse] Specifies a statement for adding a table comment. If not specified, the Comment check box in the Tables and Views tabs of the Database Generation box is unavailable.<br>Example (Oracle 10g):<br>`comment on table [%QUALIFIER%]%TABLE% is`<br>`%.q:COMMENT%`<br>The %TABLE% variable is the name of the table defined in the List of Tables, or in the table property sheet. The %COMMENT% variable is the comment defined in the Comment textbox of the table property sheet. |
| TypeList | Specifies a list of types (for example, DBMS: relational, object, XML) for tables. This list populates the Type list of the table property sheet.<br>The XML type is to be used with the SqlListSchema item. |
| UniqConstraint Name | Specifies whether the same name for index and constraint name may be used in the same table. The following settings are available:<br>• Yes – The table constraint and index names must be different, and this will be tested during model checking<br>• No - The table constraint and index names can be identical |

## Column

The Column category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how columns are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for columns:<br>• Add<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• ConstName<br>• Create, Drop<br>• Enable<br>• Maxlen, MaxConstLen<br>• ModifiableAttributes<br>• Options, DefOptions<br>• Permission<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| AddColnCheck | Specifies a statement for customizing the script for modifying column constraints within an alter table statement.<br>Example (Oracle 10g):<br>`alter table [%QUALIFIER%]%TABLE%`<br>`add [constraint %CONSTNAME%] check (%.A:CONSTRAINT%)` |
| AlterTableAdd Default | Specifies a statement for defining the default value of a column in an alter statement.<br>Example (SQL Server 2005):<br>`[[ constraint %ExtDeftConstName%] default %DEFAULT% ]for %COLUMN%` |
| AltEnableAdd ColnChk | Specifies if a column check constraint, built from the check parameters of the column, can or cannot be added in a table using an `alter table` statement. The following settings are available:<br>• Yes - `AddColnChck` can be used to modify the column check constraint in an `alter table` statement.<br>• No - PowerDesigner copies data to a temporary table before recreating the table with the new constraints.<br>See also `AddColnChck`. |
| AltEnableTS Copy | Enables timestamp columns in insert statements. |
| Bind | Specifies a statement for binding a rule to a column.<br>Example (ASE 15):<br>`[%R%?[exec ]][execute ]sp_bindrule [%R%?['[%QUALIFIER%]%RULE%'][[%QUALIFIER%]%RULE%]:['[%QUALIFIER%]%RULE%']], '%TABLE%.%COLUMN%'` |
| CheckNull | Specifies whether a column can be null. |

| Item | Description |
|---|---|
| Column Comment | Specifies a statement for adding a comment to a column.<br>Example:<br>`comment on column [%QUALIFIER%]%TABLE%.%COLUMN% is %.q:COMMENT%` |
| DefineColn Check | Specifies a statement for customizing the script of column constraints (checks) within a `create table` statement. This statement is called if the create, add, or alter statements contain %CONSTDEFN%.<br>Example:<br>`[constraint %CONSTNAME%] check (%CONSTRAINT%)` |
| DropColnChck | Specifies a statement for dropping a column check in an `alter table` statement. This statement is used in the database modification script when the check parameters have been removed on a column.<br>If `DropColnChck` is empty, PowerDesigner copies data to a temporary table before recreating the table with the new constraints.<br>Example (SQL Anywhere 10):<br>`alter table [%QUALIFIER%]%TABLE%`<br>`drop constraint %CONSTNAME%` |
| DropColnComp | Specifies a statement for dropping a column computed expression in an alter table statement.<br>Example (SQL Anywhere 10):<br>`alter table [%QUALIFIER%]%TABLE%`<br>`alter %COLUMN% drop compute` |
| DropDefault Constraint | Specifies a statement for dropping a constraint linked to a column defined with a default value<br>Example (SQL Server 2005):<br>`[%ExtDeftConstName%?alter table [%QUALIFIER%]%TABLE%`<br>`drop constraint %ExtDeftConstName%]` |
| EnableBindRule | Specifies whether business rules may be bound to columns for check parameters. The following settings are available:<br>• Yes - The Create and Bind entry of Rule are generated<br>• No - The check is generated inside the column Add order |
| Enable ComputedColn | Specifies whether computed columns are permitted. |

| Item | Description |
|---|---|
| EnableDefault | Specifies whether predefined default values are permitted. The following settings are available:<br>• Yes - The default value (if defined) is generated for columns. It can be defined in the check parameters for each column. The %DEFAULT% variable contains the default value. The Default Value check box for columns must be selected in the Tables & Views tabs of the Database Generation box<br>• No - The default value can not be generated, and the Default Value check box is unavailable.<br>Example (AS IQ 12.6):<br>EnableDefault is enabled and the default value for the column employee function EMPFUNC is Technical Engineer. The generated script is:<br>`create table EMPLOYEE ( EMPNUM numeric(5) not null, EMP_EMPNUM numeric(5) , DIVNUM numeric(5) not null, EMPFNAM char(30) , EMPLNAM char(30) not null, EMPFUNC char(30) default 'Technical Engineer', EMPSAL numeric(8,2) , primary key (EMPNUM) );` |
| EnableIdentity | Specifies whether the Identity keyword is supported. Identity columns are serial counters maintained by the database (for example Sybase and Microsoft SQL Server). The following settings are available:<br>• Yes - Enables the Identity check box in the column property sheet.<br>• No - The Identity check box is not available.<br>When the Identity check box is selected, the Identity keyword is generated in the script after the column data type. An identity column is never null, and so the Mandatory check box is automatically selected. PowerDesigner ensures that:<br>• Only one identity column is defined per table<br>• A foreign key cannot be an identity column<br>• The Identity column has an appropriate data type. If the Identity check box is selected for a column with an unsupported data type, the data type is changed to *numeric*. If the data type of an identity column is changed to an unsupported type, the error "Identity cannot be used with the selected data type" is displayed.<br>Note that, during generation, the %IDENTITY% variable contains the value "identity" but you can easily change it, if needed, using the following syntax :<br>`[%IDENTITY%?new identity keyword]` |
| EnableNotNull WithDflt | Specifies whether default values are assigned to columns containing Null values. The following settings are available:<br>• Yes - The With Default check box is enabled in the column property sheet. When it is selected, a default value is assigned to a column when a Null value is inserted.<br>• No - The With Default check box is not available. |

| Item | Description |
|---|---|
| ModifyColn Chck | Specifies a statement for modifying a column check in an `alter table` statement. This statement is used in the database modification script when the check parameters of a column have been modified in the table.<br>If `AddColnChck` is empty, PowerDesigner copies data to a temporary table before recreating the table with the new constraints.<br>Example (AS IQ 12.6):<br>`alter table [%QUALIFIER%]%TABLE%`<br>`modify %COLUMN% check (%.A:CONSTRAINT%)`<br>The %COLUMN% variable is the name of the column defined in the table property sheet. The % CONSTRAINT % variable is the check constraint built from the new check parameters.<br>`AltEnableAddColnChk` must be set to YES to allow use of this statement. |
| ModifyColn Comp | Specifies a statement for modifying a computed expression for a column in an alter table.<br>Example (ASA 6):<br>`alter table [%QUALIFIER%]%TABLE%`<br>`alter %COLUMN% set compute (%COMPUTE%)` |
| ModifyColnDflt | Specifies a statement for modifying a column default value in an `alter table` statement. This statement is used in the database modification script when the default value of a column has been modified in the table.<br>If `ModifyColnDflt` is empty, PowerDesigner copies data to a temporary table before recreating the table with the new constraints.<br>Example (ASE 15):<br>`alter table [%QUALIFIER%]%TABLE%`<br>`replace %COLUMN% default %DEFAULT%`<br>The %COLUMN% variable is the name of the column defined in the table property sheet. The %DEFAULT% variable is the new default value of the modified column. |
| ModifyColnNull | Specifies a statement for modifying the null/not null status of a column in an `alter table` statement.<br>Example (Oracle 10g):<br>`alter table [%QUALIFIER%]%TABLE%`<br>`modify %COLUMN% %MAND%` |
| ModifyColumn | Specifies a statement for modifying a column. This is a different statement from the `alter table` statement, and is used in the database modification script when the column definition has been modified.<br>Example (SQL Anywhere 10):<br>`alter table [%QUALIFIER%]%TABLE%`<br>`modify %COLUMN% %DATATYPE% %NOTNULL%` |
| NullRequired | Specifies the mandatory status of a column. This item is used with the NULLNOTNULL column variable, which can take the "null", "not null" or empty values. For more information, see *Working with Null values* on page 67. |
| Rename | Specifies a statement for renaming a column within an `alter table` statement.<br>Example (Oracle 10g):<br>`alter table [%QUALIFIER%]%TABLE%`<br>`rename column %OLDCOLN% to %NEWCOLN%` |

| Item | Description |
|---|---|
| SqlChckQuery | Specifies a SQL query to reverse engineer column check parameters. The result must conform to proper SQL syntax.<br>Example (SQL Anywhere 10):<br>`{OWNER, TABLE, COLUMN, CONSTNAME, CONSTRAINT}`<br>`select u.user_name, t.table_name,`<br>` c.column_name, k.constraint_name,`<br>` case(lcase(left(h.check_defn, 5))) when 'check' then sub-`<br>`string(h.check_defn, 6) else h.check_defn end`<br>`from sys.sysconstraint k`<br>` join sys.syscheck h on (h.check_id = k.constraint_id)`<br>` join sys.systab t on (t.object_id = k.table_object_id)`<br>` join sys.sysuserperms u on (u.user_id = t.creator)`<br>` join sys.syscolumn c on (c.object_id = k.ref_object_id)`<br>`where k.constraint_type = 'C'`<br>`[  and u.user_name=%.q:OWNER%]`<br>`[  and t.table_name=%.q:TABLE%]`<br>`[  and c.column_name=%.q:COLUMN%]`<br>`order by 1, 2, 3, 4` |
| SqlStatistics | Specifies a SQL query to reverse engineer column and table statistics.<br>Example (ASE 15):<br>`[%ISLONGDTTP%?{ AverageLength }`<br>`select [%ISLONGDTTP%?[%ISSTRDTTP%?avg(char_length(%COLUMN`<br>`%)):avg(datalength(%COLUMN%))]:null] as average_length`<br>`from [%QUALIFIER%]%TABLE%`<br>`:{ NullValuesRate, DistinctValues, AverageLength }`<br>`select`<br>`[%ISMAND%?null:(count(*) - count(%COLUMN%)) * 100 /`<br>`count(*)]  as null_values,`<br>`[%ISMAND%?null:count(distinct %COLUMN%)]  as distinct_val-`<br>`ues,`<br>`[%ISVARDTTP%?[%ISSTRDTTP%?avg(char_length(%COLUMN`<br>`%)):avg(datalength(%COLUMN%))]:null] as average_length`<br>`from [%QUALIFIER%]%TABLE%]` |
| Unbind | Specifies a statement for unbinding a rule to a column.<br>Example (ASE 15):<br>`[%R%?[exec ]][execute ]sp_unbindrule '%TABLE%.%COLUMN%'` |

## Working with Null Values

The NullRequired item specifies the mandatory status of a column. This item is used with the NULLNOTNULL column variable, which can take the "null", "not null" or empty values. The following combinations are available

### When the Column Is Mandatory

"not null" is always generated whether NullRequired is set to True or False as shown in the following example:

```
create domain DOMN_MAND char(33) not null;
create domain DOMN_NULL char(33)   null;

create table TABLE_1
(
 COLN_MAND_1 char(33)  not null,
 COLN_MAND_2 DOMN_MAND not null,
 COLN_MAND_3 DOMN_NULL not null,
);
```

### When the Column Is not Mandatory

- If NullRequired is set to True, "null" is generated. The NullRequired item should be used in ASE for example, where nullability is a database option, and the "null" or "not null" keywords are required.

  In the following example, all "null" values are generated:

```
create domain DOMN_MAND char(33) not null;
create domain DOMN_MAND char(33)   null;

create table TABLE_1
(
 COLN_NULL_1 char(33)  null,
 COLN_NULL_2 DOMN_NULL   null,
 COLN_NULL_3 DOMN_MAND   null
)
```

- If NullRequired is set to False, an empty string is generated. However, if a column attached to a mandatory domain becomes non-mandatory, "null" will be generated.

  In the following example, "null" is generated only for COLUMN_NULL3 because this column uses the mandatory domain, the other columns generate an empty string:

```
create domain DOMN_MAND char(33) not null;
create domain DOMN_NULL char(33)   null;

create table TABLE_1
(
 COLUMN_NULL1 char(33)    ,
 COLUMN_NULL2 DOMN_NULL   ,
 COLUMN_NULL3 DOMN_MAND   null
);
```

## Index

The Index category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how indexes are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for indexes:<br>• Add<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable, EnableOwner<br>• Header, Footer<br>• Maxlen<br>• ModifiableAttributes<br>• Options, DefOptions<br>• ReversedQueries<br>• ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlOptsQuery<br>For a description of each of these common items, see *Common object items* on page 56. |

| Item | Description |
|---|---|
| AddColIndex | Specifies a statement for adding a column in the `Create Index` statement. This parameter defines each column in the column list of the `Create Index` statement.<br>Example (ASE 15):<br>`%COLUMN%[ %ASC%]`<br>%COLUMN% is the code of the column defined in the column list of the table. %ASC% is ASC (ascending order) or DESC (descending order) depending on the Sort radio button state for the index column. |
| Cluster | Specifies the value to be assigned to the Cluster keyword. If this parameter is empty, the default value of the %CLUSTER% variable is CLUSTER. |
| CreateBefore Key | Controls the generation order of keys and indexes. The following settings are available:<br>• Yes – Indexes are generated before keys.<br>• No – Indexes are generated after keys. |
| DefIndexType | Specifies the default type of an index.<br>Example (DB2):<br>`Type2` |
| DefineIndex Column | Specifies the column of an index. |
| EnableAscDesc | Enables the Sort property in Index property sheets, which allows sorting in ascending or descending order. The following settings are available:<br>• Yes – The Sort property is enabled for indexes, with Ascending selected by default. The variable %ASC% is calculated, and the ASC or DESC keyword is generated when creating or modifying the database<br>• No – Index sorting is not supported.<br>Example (SQL Anywhere 10):<br>A primary key index is created on the TASK table, with the PRONUM column sorted in ascending order and the TSKNAME column sorted in descending order:<br>`create index IX_TASK on TASK (PRONUM asc, TSKNAME desc);` |
| EnableCluster | Enables the creation of cluster indexes. The following settings are available:<br>• Yes - The Cluster check box is enabled in index property sheets.<br>• No – Cluster indexes are not supported. |
| EnableFunction | Enables the creation of function-based indexes. The following settings are available:<br>• Yes - You can define expressions for indexes.<br>• No – Function-based indexes are not supported. |
| IndexComment | Specifies a Statement for adding a comment to an index.<br>Example (SQL Anywhere 10):<br>`comment on index [%QUALIFIER%]%TABLE%.%INDEX% is %.q:COMMENT%` |

| Item | Description |
|---|---|
| IndexType | Specifies a list of available index types.<br>Example (IQ 12.6):<br>`CMP`<br>`HG`<br>`HNG`<br>`LF`<br>`WD`<br>`DATE`<br>`TIME`<br>`DTTM` |
| MandIndexType | Specifies whether the index type is mandatory for indexes. The following settings are available:<br>• Yes – The index type is mandatory.<br>• No - The index type is not mandatory. |
| MaxColIndex | Specifies the maximum number of columns that may be included in an index. This value is used during model checking. |
| SqlSysIndex Query | Specifies a SQL query used to list system indexes created by the database. These indexes are excluded during reverse engineering.<br>Example (AS IQ 12.6):<br>`{OWNER, TABLE, INDEX, INDEXTYPE}`<br>`select u.user_name, t.table_name, i.index_name, i.in-`<br>`dex_type`<br>`from sysindex i, systable t, sysuserperms u`<br>`where t.table_id = i.table_id`<br>`and u.user_id = t.creator`<br>`and i.index_owner != 'USER'`<br>`[and u.user_name=%.q:OWNER%]`<br>`[and t.table_name=%.q:TABLE%]`<br>`union`<br>`select u.user_name, t.table_name, i.index_name, i.in-`<br>`dex_type`<br>`from sysindex i, systable t, sysuserperms u`<br>`where t.table_id = i.table_id`<br>`and u.user_id = t.creator`<br>`and i.index_type = 'SA'`<br>`[and u.user_name=%.q:OWNER%]`<br>`[and t.table_name=%.q:TABLE%]` |
| UniqName | Specifies whether index names must be unique within the global scope of the database. The following settings are available:<br>• Yes – Index names must be unique within the global scope of the database.<br>• No – Index names must be unique per object |

## Pkey

The Pkey category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how primary keys are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for primary keys:<br>• Add<br>• ConstName<br>• Create, Drop<br>• Enable<br>• Options, DefOptions<br>• ReversedQueries<br>For a description of each of these common items, see *Common object items* on page 56. |
| EnableCluster | Specifies whether clustered constraints are permitted on primary keys.<br>• Yes - Clustered constraints are permitted.<br>• No - Clustered constraints are not permitted. |
| PkAutoIndex | Determines whether a `Create Index` statement is generated for every Primary key statement. The following settings are available:<br>• Yes - Automatically generates a primary key index with the primary key statement. If you select the primary key check box under create index when generating or modifying a database, the primary key check box of the create table will automatically be cleared, and vice versa.<br>• No - Primary key indexes are not automatically generated. Primary key and create index check boxes can be selected at the same time. |
| PKeyComment | Specifies a statement for adding a primary key comment. |
| UseSpPrimKey | Specifies the use of the `Sp_primarykey` statement to generate primary keys. For a database that supports the procedure to implement key definition, you can test the value of the corresponding variable %USE_SP_PKEY% and choose between the creation key in the table or launching a procedure. The following settings are available:<br>• Yes - The `Sp_primarykey` statement is used to generate primary keys.<br>• No - Primary keys are generated separately in an `alter table` statement.<br>Example (ASE 15):<br>If UseSpPrimKey is enabled the Add entry for Pkey contains:<br>`UseSpPrimKey = YES`<br>`Add entry of`<br>`[%USE_SP_PKEY%?[execute] sp_primarykey %TABLE%, %PKEYCOLUMNS%`<br>`:alter table [%QUALIFIER%]%TABLE%`<br>`add [constraint %CONSTNAME%] primary key [%IsClustered%] (%PKEYCOLUMNS%)`<br>`[%OPTIONS%]]` |

## Key

The Key category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how keys are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for keys:<br>• Add<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• ConstName<br>• Create, Drop<br>• Enable<br>• MaxConstLen<br>• ModifiableAttributes<br>• Options, DefOptions<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlOptsQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| AKeyComment | Specifies a statement for adding an alternate key comment. |
| AllowNullable Coln | Specifies whether non-mandatory columns are permitted. The following settings are available:<br>• Yes - Non mandatory columns are permitted.<br>• No - Non mandatory column are not permitted. |
| EnableCluster | Specifies whether clustered constraints are permitted on alternate keys.<br>• Yes - Clustered constraints are permitted.<br>• No - Clustered constraints are not permitted. |
| SqlAkeyIndex | Specifies a reverse-engineering query for obtaining the alternate key indexes of a table by live connection.<br>Example (SQL Anywhere 10):<br>`select distinct  i.index_name`<br>`from sys.sysuserperms u`<br>`  join sys.systable t on`<br>`  (t.creator=u.user_id)`<br>`  join sys.sysindex i on`<br>`  (i.table_id=t.table_id)`<br>`where i."unique" not in ('Y', 'N')`<br>`[  and t.table_name = %.q:TABLE%]`<br>`[  and u.user_name = %.q:SCHEMA%]` |
| UniqConstAuto Index | Determines whether a `Create Index` statement is generated for every key statement. The following settings are available:<br>• Yes - Automatically generates an alternate key index within the alternate key statement. If you select the alternate key check box under create index when generating or modifying a database, the alternate key check box of the create table will automatically be cleared, and vice versa.<br>• No - Alternate key indexes are not automatically generated. Alternate key and create index check boxes can be selected at the same time. |

## Reference

The Reference category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how references are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for references:<br>• Add<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• ConstName<br>• Create, Drop<br>• Enable<br>• MaxConstLen<br>• ModifiableAttributes<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| CheckOn Commit | Specifies that referential integrity testing is performed only after the COMMIT. Contains the keyword used to specify a reference with the CheckOnCommit option.<br>Example:<br>`CHECK ON COMMIT` |
| DclDelIntegrity | Specifies a list of declarative referential integrity constraints allowed for delete. The list can contain any or all of the following values, which control the availability of the relevant radio buttons on the Integrity tab of reference property sheets:<br>• RESTRICT<br>• CASCADE<br>• SET NULL<br>• SET DEFAULT |
| DclUpdIntegrity | Specifies a list of declarative referential integrity constraints allowed for update. The list can contain any or all of the following values, which control the availability of the relevant radio buttons on the Integrity tab of reference property sheets:<br>• RESTRICT<br>• CASCADE<br>• SET NULL<br>• SET DEFAULT |
| DefineJoin | Specifies a statement to define a join for a reference. This is another way of defining the contents of the `create reference` statement, and corresponds to the %JOINS% variable.<br>Usually the `create` script for a reference uses the %CKEYCOLUMNS% and %PKEYCOLUMNS% variables, which contain the lists of child and parent columns separated by commas.<br>If you use %JOINS%, you can refer to each paired parent and child columns separately. A loop is executed on Join for each paired parent and child columns, allowing to have a syntax mix of PK and FK.<br>Example (Access 2000):<br>`P=%PK% F=%FK%` |

| Item | Description |
|---|---|
| EnableChange JoinOrder | Specifies whether, when a reference is linked to a key as shown in the Joins tab of reference properties, the auto arrange join order check box and features are available. The following settings are available:<br>• Yes - The join order can be established automatically, using the Auto arrange join order check box. Selecting this check box sorts the list according to the key column order. Clearing this check box allows manual sorting of the join order with the move buttons.<br>• No - The auto arrange join order property is unavailable. |
| EnableCluster | Specifies whether clustered constraints are permitted on foreign keys.<br>• Yes - Clustered constraints are permitted.<br>• No - Clustered constraints are not permitted. |
| EnablefKey Name | Specifies the foreign key role allowed during database generation. The following settings are available:<br>• Yes - The code of the reference is used as role for the foreign key.<br>• No - The foreign key role is not allowed. |
| FKAutoIndex | Determines whether a `Create Index` statement is generated for every foreign key statement. The following settings are available:<br>• Yes - Automatically generates a foreign key index with the foreign key statement. If you select the foreign key check box under create index when generating or modifying a database, the foreign key check box of the create table will automatically be cleared, and vice versa.<br>• No – Foreign key indexes are not automatically generated. Foreign key and create index check boxes can be selected at the same time. |
| FKeyComment | Specifies a statement for adding an alternate key comment. |
| SqlListChildren Query | Specifies a SQL query used to list the joins in a reference.<br>Example (Oracle 10g):<br>`{CKEYCOLUMN, FKEYCOLUMN}`<br>`[%ISODBCUSER%?select`<br>`p.column_name, f.column_name`<br>`from sys.user_cons_columns f,`<br>`sys.all_cons_columns p`<br>`where f.position = p.position`<br>`and f.table_name=%.q:TABLE%`<br>`[ and p.owner=%.q:POWNER%]`<br>`and p.table_name=%.q:PARENT%`<br>`and f.constraint_name=%.q:FKCONSTRAINT%`<br>`and p.constraint_name=%.q:PKCONSTRAINT%`<br>`order by f.position`<br>`:select p.column_name, f.column_name`<br>`from sys.all_cons_columns f,`<br>`sys.all_cons_columns p`<br>`where f.position = p.position`<br>`and f.owner=%.q:SCHEMA%`<br>`and f.table_name=%.q:TABLE%`<br>`[ and p.owner=%.q:POWNER%]`<br>`and p.table_name=%.q:PARENT%`<br>`and f.constraint_name=%.q:FKCONSTRAINT%`<br>`and p.constraint_name=%.q:PKCONSTRAINT%`<br>`order by f.position]` |

| Item | Description |
|---|---|
| UseSpFornKey | Specifies the use of the `Sp_foreignkey` statement to generate a foreign key. The following settings are available:<br>• Yes - The `Sp_foreignkey` statement is used to create references.<br>• No - Foreign keys are generated separately in an `alter table` statement using the Create order of reference.<br>See also `UseSpPrimKey` (*Pkey* on page 70). |

## View

The View category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how views are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for views:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable, EnableSynonym<br>• Header, Footer<br>• ModifiableAttributes<br>• Options<br>• Permission<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| EnableIndex | Specifies a list of view types for which a view index is available.<br>Example (Oracle 10g):<br>`MATERIALIZED` |
| SqlListSchema | Specifies a query used to retrieve registered schemas in the database. This item is used with views of XML type (a reference to an XML document stored in the database).<br>When you define an XML view, you need to retrieve the XML documents registered in the database in order to assign one document to the view, this is done using the SqlListSchema query.<br>Example (Oracle 10g):<br>`SELECT schema_url FROM dba_xml_schemas` |
| SqlXMLView | Specifies a sub-query used to improve the performance of SqlAttrQuery. |
| TypeList | Specifies a list of types (for example, DBMS: relational, object, XML) for views. This list populates the Type list of the view property sheet.<br>The XML type is to be used with the SqlListSchema item. |

| Item | Description |
|---|---|
| ViewCheck | Specifies whether the With Check Option check box in the view property sheet is available. If the check box is selected and the `ViewCheck` parameter is not empty, the value of `ViewCheck` is generated at the end of the view select statement and before the terminator.<br>Example (SQL Anywhere 10):<br>If ViewCheck is set to with check option, the generated script is:<br>`create view TEST as`<br>`select CUSTOMER.CUSNUM, CUSTOMER.CUSNAME, CUSTOMER.CUSTEL`<br>`from CUSTOMER`<br>`with check option;` |
| ViewComment | Specifies a statement for adding a view comment. If this parameter is empty, the Comment check box in the Views groupbox in the Tables and Views tabs of the Generate Database box is unavailable.<br>Example (Oracle 10g):<br>`[%VIEWSTYLE%=view? comment on table [%QUALIFIER%]%VIEW% is %.q:COMMENT%]` |
| ViewStyle | Specifies a view usage. The value defined is displayed in the Usage list of the view property sheet.<br>Example (Oracle 10g):<br>`materialized view` |

## Tablespace

The Tablespace category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how tablespaces are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for tablespaces:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable<br>• ModifiableAttributes<br>• Options, DefOptions<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlOptsQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| Tablespace Comment | Specifies a statement for adding a tablespace comment. |

## Storage

The Storage category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how storages are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for storages:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable<br>• ModifiableAttributes<br>• Options, DefOptions<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| Storage Comment | Specifies a statement for adding a storage comment. |

## Database

The Database category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how databases are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for databases:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable<br>• ModifiableAttributes<br>• Options, DefOptions<br>• Permission<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| BeforeCreate Database | Controls the order in which databases, tablespaces, and storages are generated. The following settings are available:<br>• Yes – [default] Create Tablespace and Create Storage statements are generated before the Create Database statement.<br>• No - Create Tablespace and Create Storage statements are generated after the Create Database statement |
| CloseDatabase | Specifies the command for closing the database. If this parameter is empty, the Database/Close option on the Options tab of the Generate Database box is unavailable. |
| EnableMany Databases | Enables support for multiple databases in the same model. |
| OpenDatabase | Specifies the command for opening the database. If this parameter is empty, the Database/Open option on the Options tab of the Generate Database box is unavailable.<br>Example (ASE 15):<br>`use %DATABASE%`<br>The %DATABASE% variable is the code of the database associated with the generated model. |

## Domain

The Domain category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how domains are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for domains:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable, EnableOwner<br>• Maxlen<br>• ModifiableAttributes<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| Bind | Specifies the syntax for binding a business rule to a domain.<br>Example (ASE 15):<br>`[%R%?[exec ]][execute ]sp_bindrule [%R%?['[%QUALIFIER%]%RULE%']][[%QUALIFIER%]%RULE%]:['[%QUALIFIER%]%RULE%']], %DOMAIN%` |
| EnableBindRule | Specifies whether business rules may be bound to domains for check parameters. The following settings are available:<br>• Yes - The Create and Bind entry of Rule are generated<br>• No - The check inside the domain Add order is generated |
| EnableCheck | Specifies whether check parameters are generated.<br>This item is tested during column generation. If User-defined Type is selected for columns in the Generation dialog box, and EnableCheck is set to Yes for domains, then the check parameters are not generated for columns, since the column is associated with a domain with check parameters. When the checks on the column diverge from those of the domain, the column checks are generated.<br>The following settings are available:<br>• Yes - Check parameters are generated<br>• No - Variables linked to check parameters are not evaluated during generation and reverse |
| EnableDefault | Specifies whether default values are generated. The following settings are available:<br>• Yes - Default values defined for domains are generated. The default value can be defined in the check parameters. The %DEFAULT% variable contains the default value<br>• No - Default values are not generated |
| SqlListDefault Query | Specifies a SQL query to retrieve and list domain default values in the system tables during reverse engineering. |
| UddtComment | Specifies a statement for adding a user-defined data type comment. |
| Unbind | Specifies the syntax for unbinding a business rule from a domain.<br>Example (ASE 15):<br>`[%R%?[exec ]][execute ]sp_unbindrule %DOMAIN%` |

## Abstract Data Type

The Abstract Data Type category is located in the **Root > Script > Objects** category, and can contain the following items that define how abstract data types are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for abstract data types:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable<br>• ModifiableAttributes<br>• Permission<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlPermQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| ADTComment | Specifies a statement for adding an abstract data type comment. |
| AllowedADT | Specifies a list of abstract data types which can be used as data types for abstract data types.<br>Example (Oracle 10g):<br>`OBJECT`<br>`TABLE`<br>`VARRAY` |
| Authorizations | Specifies a list of those users able to invoke abstract data types. |
| CreateBody | Specifies a statement for creating an abstract data type body.<br>Example (Oracle 10g):<br>`create [or replace ]type body [%QUALIFIER%]%ADT% [.O:[as] [is]]`<br>` %ADTBODY%`<br>`end;` |
| EnableAdtOn Coln | Specifies whether abstract data types are enabled for columns. The following settings are available:<br>• Yes - Abstract Data Types are added to the list of column types provided they have the valid type.<br>• No - Abstract Data Types are not allowed for columns. |
| EnableAdtOn Domn | Specifies whether abstract data types are enabled for domains. The following settings are available:<br>• Yes - Abstract Data Types are added to the list of domain types provided they have the valid type<br>• No - Abstract Data Types are not allowed for domains |
| Enable Inheritance | Enables inheritance for abstract data types. |
| Install | Specifies a statement for installing a Java class as an abstract data class (in ASA, abstract data types are installed and removed rather than created and deleted). This item is equivalent to a `create` statement.<br>Example (SQL Anywhere 10):<br>`install JAVA UPDATE from file %.q:FILE%` |
| JavaData | Specifies a list of available instantiation mechanisms for SQL Java abstract data types. |

| Item | Description |
|---|---|
| Remove | Specifies a statement for installing a Java class as an abstract data class.<br>Example (SQL Anywhere 10):<br>`remove JAVA class %ADT%` |

## Abstract Data Type Attribute

The Abstract Data Types Attribute category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how abstract data type attributes are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for abstract data type attributes:<br>• Add<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop, Modify<br>• ModifiableAttributes<br>• ReversedQueries, ReversedStatements<br>• SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| AllowedADT | Specifies a list of abstract data types which can be used as data types for abstract data type attributes.<br>Example (Oracle 10g):<br>`OBJECT`<br>`TABLE`<br>`VARRAY`<br>If you select the type OBJECT for an abstract data type, an Attributes tab appears in the abstract data type property sheet, allowing you to specify the attributes of the object data type. |

## User

The User category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how users are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for users:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable<br>• Maxlen<br>• ModifiableAttributes<br>• Options, DefOptions<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| UserComment | Specifies a statement for adding a user comment. |

## Rule

The Rule category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how rules are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for rules:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable<br>• Maxlen<br>• ModifiableAttributes<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| ColnDefault Name | Specifies the name of a default for a column. This item is used with DBMSs that do not support check parameters on columns. When a column has a specific default value defined in its check parameters, a name is created for this default value.<br>The corresponding variable is %DEFAULTNAME%.<br>Example (ASE 15):<br>`D_%.19:COLUMN%_%.8:TABLE%`<br>The `EMPFUNC` column of the EMPLOYEE table has a default value of `Technical Engineer.` The `D_EMPFUNC_EMPLOYEE` column default name is created:<br>`create default D_EMPFUNC_EMPLOYEE`<br>`as 'Technical Engineer'`<br>`go`<br>`execute sp_bindefault D_EMPFUNC_EMPLOYEE, "EMPLOYEE.EMP-FUNC"`<br>`go` |
| ColnRuleName | Specifies the name of a rule for a column. This item is used with DBMSs that do not support check parameters on columns. When a column has a specific rule defined in its check parameters, a name is created for this rule.<br>The corresponding variable is %RULE%.<br>Example (ASE 15):<br>`R_%.19:COLUMN%_%.8:TABLE%`<br>The TEASPE column of the Team table has a list of values - Industry, Military, Nuclear, Bank, Marketing - defined in its check parameters:<br>The R_TEASPE_TEAM rule name is created and associated with the TEASPE column:<br>`create rule R_TEASPE_TEAM`<br>`as @TEASPE in ('Industry','Military','Nuclear','Bank','Mar-keting')`<br>`go`<br>`execute sp_bindrule R_TEASPE_TEAM, "TEAM.TEASPE"`<br>`go` |
| MaxDefaultLen | Specifies the maximum length that the DBMS supports for the name of the column Default name |
| RuleComment | Specifies a statement for adding a rule comment. |

| Item | Description |
|---|---|
| UddtDefault Name | Specifies the name of a default for a user-defined data type. This item is used with DBMSs that do not support check parameters on user-defined data types. When a user-defined data type has a specific default value defined in its check parameters, a name is created for this default value.<br>The corresponding variable is %DEFAULTNAME%.<br>Example (ASE 15):<br>`D_%.28:DOMAIN%`<br>The `FunctionList` domain has a default value defined in its check parameters: `Technical Engineer`. The following SQL script will generate a default name for that default value:<br>`create default D_FunctionList`<br>`as 'Technical Engineer'`<br>`go` |
| UddtRuleName | Specifies the name of a rule for a user-defined data type. This item is used with DBMSs that do not support check parameters on user-defined data types. When a user-defined data type has a specific rule defined in its check parameters, a name is created for this rule.<br>The corresponding variable is %RULE%.<br>Example (ASE 15):<br>`R_%.28:DOMAIN%`<br>The `Domain_speciality` domain has to belong to a set of values. This domain check has been defined in a validation rule. The SQL script will generate the rule name following the template defined in the item `UddtRuleName`:<br>`create rule R_Domain_speciality`<br>`as (@Domain_speciality in ('Industry','Military','Nu-clear','Bank','Marketing'))`<br>`go`<br>`execute sp_bindrule R_Domain_speciality, T_Domain_speciali-ty`<br>`go` |

## Procedure

The Procedure category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how procedures are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for procedures:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable, EnableOwner, EnableSynonym<br>• Maxlen<br>• ModifiableAttributes<br>• Permission<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlPermQuery<br>For a description of each of these common items, see *Common object items* on page 56. |

| Item | Description |
|---|---|
| CreateFunc | Specifies the statement for creating a function.<br>Example (SQL Anywhere 10):<br>`create function [%QUALIFIER%]%FUNC%[%PROCPRMS%?([%PROCPRMS%])] %TRGDEFN%` |
| CustomFunc | Specifies the statement for creating a user-defined function, a form of procedure that returns a value to the calling environment for use in queries and other SQL statements.<br>Example (SQL Anywhere 10):<br>`create function [%QUALIFIER%]%FUNC% (<arg> <type>) RETURNS <type>`<br>`begin`<br>`end` |
| CustomProc | Specifies the statement for creating a stored procedure.<br>Example (SQL Anywhere 10):<br>`create procedure [%QUALIFIER%]%PROC% (IN <arg> <type>)`<br>`begin`<br>`end` |
| DropFunc | Specifies the statement for dropping a function.<br>Example (SQL Anywhere 10):<br>`if exists(select 1 from sys.sysprocedure where proc_name = %.q:FUNC%[ and user_name(creator) = %.q:OWNER%]) then`<br>`drop function [%QUALIFIER%]%FUNC%`<br>`end if` |
| EnableFunc | Specifies whether functions are allowed. Functions are forms of procedure that return a value to the calling environment for use in queries and other SQL statements. |
| Function Comment | Specifies a statement for adding a function comment. |
| ImplementationType | Specifies a list of available procedure template types. |
| MaxFuncLen | Specifies the maximum length of the name of a function. |
| Procedure Comment | Specifies a statement for adding a procedure comment. |

## Trigger

The Trigger category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how triggers are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for triggers:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable, EnableOwner<br>• Maxlen<br>• ModifiableAttributes<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| DefaultTrigger Name | Specifies a template to define default trigger names.<br>Example (SQL Anywhere 10):<br>`%TEMPLATE%_%.L:TABLE%` |
| EnableMulti Trigger | Enables the use of multiple triggers per type. |
| Event | Specifies a list of trigger event attributes to populate the Event list on the Definition tab of Trigger property sheets.<br>Example:<br>`Delete`<br>`Insert`<br>`Update` |
| EventDelimiter | Specifies a character to separate multiple trigger events. |
| ImplementationType | Specifies a list of available trigger template types. |
| Time | Specifies a list of trigger time attributes to populate the Time list on the Definition tab of Trigger property sheets.<br>Example:<br>`Before`<br>`After` |
| Trigger Comment | Specifies a statement for adding a trigger comment. |
| UniqName | Specifies whether trigger names must be unique within the global scope of the database. The following settings are available:<br>• Yes – Trigger names must be unique within the global scope of the database.<br>• No – Trigger names must be unique per object |

| Item | Description |
|---|---|
| UseErrorMsg Table | Specifies a macro for accessing trigger error messages from a message table in your database.<br>Enables the use of the User-defined radio button on the Error Messages tab of the Trigger Rebuild dialog box (see "Creating and generating user-defined error messages" in the Generating Triggers and Procedures chapter of the *Data Modeling* guide).<br>If an error number in the trigger script corresponds to an error number in the message table, the default error message of the .ERROR macro is replaced your message.<br>Example (ASE 15):<br>`begin`<br>`  select @errno  = %ERRNO%,`<br>`  @errmsg = %MSGTXT%`<br>`  from %MSGTAB%`<br>`  where  %MSGNO% = %ERRNO%`<br>`  goto error`<br>`end`<br>Where:<br>• %ERRNO% - error number parameter to the .ERROR macro<br>• %ERRMSG% - error message text parameter to the .ERROR macro<br>• %MSGTAB% - name of the message table<br>• %MSGNO% - name of the column that stores the error message number<br>• %MSGTXT% - name of the column that stores the error message text<br>See also UseErrorMsgText. |
| UseErrorMsg Text | Specifies a macro for accessing trigger error messages from the trigger template definition.<br>Enables the use of the Standard radio button on the Error Messages tab of the Trigger Rebuild dialog box.<br>The error number and message defined in the template definition are used.<br>Example (ASE 15):<br>`begin`<br>`  select @errno  = %ERRNO%,`<br>`  @errmsg = %MSGTXT%`<br>`  goto error`<br>`end`<br>See also UseErrorMsgTable. |
| ViewTime | Specifies a list of available times available for trigger on view. |

## DBMS Trigger

The DBMS Trigger category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how DBMS triggers are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for DBMS triggers:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Alter, AlterStatementList, AlterDBIgnored<br>• Enable, EnableOwner<br>• Header, Footer<br>• Maxlen<br>• ModifiableAttributes<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| EventDelimiter | Specifies a character to separate multiple trigger events. |
| Events_*scope* | Specifies a list of trigger event attributes to populate the Event list on the Definition tab of Trigger property sheets for the selected *scope*, for example, schema, database, server. |
| Scope | Specifies a list of available scopes for the DBMS trigger. Each scope must have an associated Events_*scope* item. |
| Time | Specifies a list of trigger time attributes to populate the Time list on the Definition tab of Trigger property sheets.<br>Example:<br>`Before`<br>`After` |
| Trigger Comment | Specifies a statement for adding a trigger comment. |

## Join Index

The Join Index category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how join indexes are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for join indexes:<br>• Add<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable, EnableOwner<br>• Header, Footer<br>• Maxlen<br>• ModifiableAttributes<br>• Options, DefOptions<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlOptsQuery<br>For a description of each of these common items, see *Common object items* on page 56. |

| Item | Description |
|---|---|
| AddJoin | Specifies the SQL statement used to define joins for join indexes.<br>Example:<br>`Table1.coln1 = Table2.coln2` |
| EnableJidxColn | Enables support for attaching multiple columns to a join index. In Oracle 9i, this is called a bitmap join index. |
| JoinIndex Comment | Specifies a statement for adding a join index comment. |

## Qualifier

The Qualifier category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how qualifiers are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for qualifiers:<br>• Enable<br>• ReversedQueries<br>• SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| Label | Specifies a label for <all> in the qualifier selection list. |

## Sequence

The Sequence category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how sequences are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for sequences:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable, EnableOwner, EnableSynonym<br>• Maxlen<br>• ModifiableAttributes<br>• Options, DefOptions<br>• Permission<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlPermQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| Rename | Specifies the command for renaming a sequence.<br>Example (Oracle 10g):<br>`rename %OLDNAME% to %NEWNAME%` |
| Sequence Comment | Specifies a statement for adding a sequence comment. |

## Synonym

The Synonym category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how synonyms are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for synonyms:<br>• Create, Drop<br>• Enable, EnableSynonym<br>• Maxlen<br>• ReversedQueries<br>• SqlAttrQuery, SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| EnableAlias | Specifies whether synonyms may have a type of alias. |

## Group

The Group category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how groups are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for groups:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable<br>• Maxlen<br>• ModifiableAttributes<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlPermQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| Bind | Specifies a command for adding a user to a group.<br>Example (SQL Anywhere 10):<br>`grant membership in group %GROUP% to %USER%` |
| Group Comment | Specifies a statement for adding a group comment. |
| ObjectOwner | Allows groups to be object owners. |

| Item | Description |
|---|---|
| SqlListChildren Query | Specifies a SQL query for listing the members of a group.<br>Example (ASE 15):<br>`{GROUP ID, MEMBER}`<br>`select g.name, u.name`<br>`from`<br>`[%CATALOG%.]dbo.sysusers u, [%CATALOG%.]dbo.sysusers g`<br>`where`<br>`u.suid > 0 and`<br>`u.gid = g.gid and`<br>`g.gid = g.uid`<br>`order by 1, 2` |
| Unbind | Specifies a command for removing a user from a group.<br>Example (SQL Anywhere 10):<br>`revoke membership in group %GROUP% from %USER%` |

## Role

The Role category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how roles are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for roles:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable<br>• Maxlen<br>• ModifiableAttributes<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlPermQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| Bind | Specifies a command for adding a role to a user or to another role.<br>Example (ASE 15):<br>`grant role %ROLE% to %USER%` |
| SqlListChildren Query | Specifies a SQL query for listing the members of a group.<br>Example (ASE 15):<br>`{ ROLE ID, MEMBER }`<br>`SELECT r.name, u.name`<br>`FROM`<br>`master.dbo.sysloginroles l,`<br>`[%CATALOG%.]dbo.sysroles s,`<br>`[%CATALOG%.]dbo.sysusers u,`<br>`[%CATALOG%.]dbo.sysusers r`<br>`where`<br>`l.suid = u.suid`<br>`and s.id =l.srid`<br>`and r.uid = s.lrid` |

| Item | Description |
|---|---|
| Unbind | Specifies a command for removing a role from a user or another role. |

## DB Package

The DB Package category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how database packages are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for database packages:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable, EnableSynonym<br>• Maxlen<br>• ModifiableAttributes<br>• Permission<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery, SqlPermQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| Authorizations | Specifies a list of those users able to invoke database packages. |
| CreateBody | Specifies a template for defining the body of the database package. This statement is used in the extension statement AfterCreate.<br>Example (Oracle 10g):<br>`create [or replace ]package body [%QUALIFIER%]%DBPACKAGE%`<br>`[.O:[as][is]][%IsPragma% ? pragma serially_reusable]`<br>` %DBPACKAGEBODY%`<br>`[begin`<br>` %DBPACKAGEINIT%`<br>`]end[ %DBPACKAGE%];` |

## DB Package Sub-objects

The following categories are located in the **Root** > **Script** > **Objects** category:

- DB Package Procedure
- DB Package Variable
- DB Package Type
- DB Package Cursor
- DB Package Exception
- DB Package Pragma

Each contains many of the following items that define how database packages are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for database packages:<br>• Add<br>• ReversedQueries<br>For a description of each of these common items, see *Common object items* on page 56. |
| DBProcedure Body | [database package procedures only] Specifies a template for defining the body of the package procedure in the Definition tab of its property sheet.<br>Example (Oracle 10g):<br>`begin`<br>`end` |
| ParameterTypes | [database package procedures and cursors only] Specifies the available types for procedures or cursors.<br>Example (Oracle 10g: procedure):<br>`in`<br>`in nocopy`<br>`in out`<br>`in out nocopy`<br>`out`<br>`out nocopy` |

## Parameter

The Parameter category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how parameters are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for database packages:<br>• Add<br>• ReversedQueries<br>For a description of each of these common items, see *Common object items* on page 56. |

## Privilege

The Privilege category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how privileges are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for privileges:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable<br>• ModifiableAttributes<br>• ReversedQueries, ReversedStatements<br>For a description of each of these common items, see *Common object items* on page 56. |

| Item | Description |
|---|---|
| GrantOption | Specifies the grant option for a privileges statement.<br>Example (Oracle 10g):<br>`with admin option` |
| RevokeInherited | Allows you to revoke inherited privileges from groups and roles. |
| RevokeOption | Specifies revoke option for a privileges statement. |
| System | Specifies a list of available system privileges.<br>Example (ASE 15):<br>`CREATE DATABASE`<br>`CREATE DEFAULT`<br>`CREATE PROCEDURE`<br>`CREATE TRIGGER`<br>`CREATE RULE`<br>`CREATE TABLE`<br>`CREATE VIEW` |

## Permission

The Permission category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how permissions are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for permissions:<br>• Create, Drop<br>• Enable<br>• ReversedQueries<br>• SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| GrantOption | Specifies the grant option for a permissions statement.<br>Example (ASE 15):<br>`with grant option` |
| RevokeInherited | Allows you to revoke inherited permissions from groups and roles. |
| RevokeOption | Specifies the revoke option for a permissions statement.<br>Example (ASE 15):<br>`cascade` |

## Default

The Default category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how defaults are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for defaults:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable, EnableOwner<br>• Maxlen<br>• ModifiableAttributes<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| Bind | Specifies the command for binding a default object to a domain or a column.<br>When a domain or a column use a default object, a *binddefault* statement is generated after the domain or table creation statement. In the following example, column Address in table Customer uses default object CITYDFLT:<br>`create table CUSTOMER (`<br>`  ADDRESS  char(10)  null`<br>`)`<br>`sp_bindefault CITYDFLT, 'CUSTOMER.ADDRESS'`<br>If the domain or column use a default value directly typed in the Default list, then the default value is declared in the column creation line:<br>`ADDRESS  char(10)  default 'StdAddr' null` |
| PublicOwner | Enables PUBLIC to own public synonyms. |
| Unbind | Specifies the command for unbinding a default object from a domain or a column.<br>Example (ASE 15):<br>`[%R%?[exec ]][execute ]sp_unbindefault %.q:BOUND_OBJECT%` |

## Web Service and Web Operation

The Web Service and Web Operation categories are located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how web services and web operations are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for web services and web operations:<br>• AfterCreate, AfterDrop, AfterModify<br>• Alter<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable, EnableOwner<br>• Header, Footer<br>• MaxConstLen (web operations only)<br>• Maxlen<br>• ModifiableAttributes<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |

| Item | Description |
|---|---|
| Enable Namespace | Specifies whether namespaces are supported. |
| EnableSecurity | Specifies whether security options are supported. |
| OperationType List | [web operation only] Specifies a list of web service operation types.<br>Example (DB2 UDB 8.x CS):<br>`query`<br>`update`<br>`storeXML`<br>`retrieveXML`<br>`call` |
| ServiceTypeList | [web service only] Specifies a list of web service types.<br>Example (SQL Anywhere 10):<br>`RAW`<br>`HTML`<br>`XML`<br>`DISH` |
| UniqName | Specifies whether web service operation names must be unique in the database. |
| WebService Comment/ WebOperation Comment | Specifies the syntax for adding a comment to web service or web service operation. |

## Web Parameter

The Web Parameter category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how web parameters are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for web parameters:<br>• Add<br>• Enable<br>For a description of each of these common items, see *Common object items* on page 56. |
| EnableDefault | Allows default values for web service parameters. |
| ParameterDttp List | Specifies a list of data types that may be used as web service parameters. |

## Result Column

The Result Column category are located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how web services and web operations are modeled for your DBMS.

| Item | Description |
|---|---|
| ResultColumn DttpList | Specifies a list of data types that may be used for result columns. |

## Dimension

The Dimension category is located in the **Root** > **Script** > **Objects** category, and can contain the following items that define how dimensions are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for dimensions:<br>• AfterCreate, AfterDrop, AfterModify<br>• Alter<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• Enable<br>• Header, Footer<br>• Maxlen<br>• ReversedQueries<br>• SqlAttrQuery, SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| AddAttr Hierarchy | Specifies the syntax for defining a list of hierarchy attributes.<br>Example (Oracle 10g):<br>`child of %DIMNATTRHIER%` |
| AddAttribute | Specifies the syntax for defining an attribute.<br>Example (Oracle 10g):<br>`attribute %DIMNATTR% determines [.O:[(%DIMNDEPCOLNLIST%)] [%DIMNDEPCOLN%]]` |
| AddHierarchy | Specifies the syntax for defining a dimension hierarchy.<br>Example (Oracle 10g):<br>`hierarchy %DIMNHIER% ( %DIMNATTRHIERFIRST% %DIMNATTRHIERLIST%)` |
| AddJoin Hierarchy | Specifies the syntax for defining a list of joins for hierarchy attributes.<br>Example (Oracle 10g):<br>join key [.O:[(%DIMNKEYLIST%)][%DIMNKEYLIST%]] references %DIMNPARENTLEVEL% |
| AddLevel | Specifies the syntax for dimension level (attribute).<br>Example (Oracle 10g):<br>level %DIMNATTR% is [.O:[(%DIMNCOLNLIST%)][%DIMNTABL%.%DIMNCOLN%]] |

## Extended Object

The Extended Object category is located in the **Root > Script > Objects** category, and can contain the following items that define how extended objects are modeled for your DBMS.

| Item | Description |
|---|---|
| [Common items] | The following common object items may be defined for extended objects:<br>• AfterCreate, AfterDrop, AfterModify<br>• BeforeCreate, BeforeDrop, BeforeModify<br>• Create, Drop<br>• EnableSynonym<br>• Header, Footer<br>• ModifiableAttributes<br>• ReversedQueries, ReversedStatements<br>• SqlAttrQuery, SqlListQuery<br>For a description of each of these common items, see *Common object items* on page 56. |
| AlterStatement List | Specifies a list of text items representing statements modifying the corresponding attributes |
| Comment | Specifies the syntax for adding a comment to an extended object. |

## Script/Data Type Category

The Data type category defines mappings between PowerDesigner and DBMS data types.

The following variables may be used:

- %n - Length of the data type
- %s - Size of the data type
- %p - Precision of the data type

| Item | Description |
|---|---|
| AmcdAmcdType | Lists mappings between specialized data types (such as `XML`, `IVL`, `MEDIA`, etc) and standard PowerDesigner data types. These mappings are used to help conversion from a one DBMS to another, when the new DBMS does not support one or more of these specialized types. For example, if the `XML` data type is not supported, `TXT` is used. |
| AmcdDataType | Lists mappings between PowerDesigner data types and DBMS data types.<br>These mappings are used during CDM to PDM generation and with the Change Current DBMS command. The following variables are used to qualify the data type:<br>Examples (ASE **15** > **PowerDesigner** ):<br>• A% **n** > **char** (%n)<br>• VA% **n** > **varchar** (%n)<br>• LA%varchar(%n) |
| PhysDataType | Lists mappings between DBMS data types and PowerDesigner data types.<br>These mappings are used during PDM to CDM generation and with the Change Current DBMS command.<br>Examples ( **PowerDesigner** > **ASE** 15):<br>• **sysname** > **VA30**<br>• **integer** > **I** |

| Item | Description |
|---|---|
| PhysDttpSize | Lists the storage sizes of non-default DBMS data types. Used when estimating the size of the database.<br>Examples (ASE 15):<br>• **smallmoney > 8**<br>• **smalldatetime > 4** |
| OdbcPhysData Type | Lists mappings between database data types and internal PowerDesigner data types. Used during live database reverse engineering.<br>The way data types are stored in the database may differ from the DBMS notation. For example, Sybase SQL Anywhere stores a decimal data type as decimal(30,6).<br>Examples (ASA **10** > **PowerDesigner** ):<br>• char(1)char<br>• decimal(30,6)decimal |
| PhysOdbcData Type | Lists mappings between DBMS data types and PowerDesigner data types.<br>Examples (Access **2000** > **PowerDesigner** ):<br>• **Integer > Short**<br>• **LongInteger > Long** |
| PhysLogADT Type | Lists mappings between DBMS abstract data types and PowerDesigner abstract data types.<br>Examples (Oracle **10g** > **PowerDesigner** ):<br>• **VARRAY > Array**<br>• SQLJ_ **OBJECT > JavaObject** |
| LogPhysADT Type | Lists mappings between PowerDesigner abstract data types and DBMS abstract data types.<br>Examples ( **PowerDesigner** > **Oracle** 10g):<br>• **Java > <Undefined>**<br>• **List > TABLE** |
| AllowedADT | Lists the abstract data types that may be used as types for columns and domains.<br>Example (ASE 15):<br>• JAVA |
| HostDataType | Lists mappings between DBMS data types and data types used in PowerDesigner procedures and trigger code.<br>Examples (Oracle **10g** > **PowerDesigner** ):<br>• **DEC > number**<br>• **SMALLINT > integer** |

## Profile Category

The Profile category is used to extend standard PowerDesigner objects. You can:

- Refine the definition, behavior, and display of existing objects by creating extended attributes, stereotypes, criteria, forms, symbols, generated files, etc.

- Add new objects by creating and stereotyping extended objects and sub-objects

The Profile category contains the following categories:

| Category | Description |
|---|---|
| Shared\ Extended attribute types | For defining extended attribute types and shared templates. Extended attribute types are data types reused among extended attributes. Shared templates are pieces of code used in text generation. |
| Metaclasses | For defining metaclass extensions like custom symbol, stereotypes, criteria, or generated files. |

For more information on profiles, see *Extending Your Models with Profiles* on page 121.

For more information on templates and generated files, see *Customizing Generation with GTL* on page 187.

### Extended Attributes

When you create an extended attribute in a metaclass, an Extended Attributes tab is displayed in the corresponding object property sheet. You can customize the extended attribute display using user-defined tabs. These tabs allow you to add an extended attribute value to the object definition.

Some DBMS are delivered with extended attributes that are needed during generation, this is why we advise you not to modify these extended attributes, or at least to make a backup copy of each DBMS file before you start modifying them.

If you wish to enhance model generation, you can copy the pattern of existing extended attributes and assign them to other object categories.

### Extended Model Definition

If you want to complement the definition of modeling objects and expand the PowerDesigner metamodel, you should define extended attributes in an extended model definition. Such extended attributes are not used during the generation process.

For more information on extended model definitions, see *Extended Model Definitions* on page 18.

The Extended Attribute Category is divided into the following categories:

- Types
- Objects

## Defining an Extended Attribute in a DBMS

Each extended attribute has the following properties:

| Property | Description |
|---|---|
| Name | Name of category or item. |
| Comment | Description of selected category or item. |
| Data type | Predefined or user-defined extended attributes types. |
| Default value | Default value from the list of values. Depends on the data type. |

### Example

In DB2 UDB 7 OS/390, the extended attribute `WhereNotNull` allows you to add a clause enforcing the uniqueness of index names if they are not null.

In the `Create index` order, `WhereNotNull` is evaluated as follows:

create [%INDEXTYPE% ][%UNIQUE% [%WhereNotNull%?where not null ]]index [%QUALIFIER%]%INDEX% on [%TABLQUALIFIER%]%TABLE% (

%CIDXLIST%

)

[%OPTIONS%]

If the index name is unique, and if you set the type of the `WhereNotNull` extended attribute to True, the "where not null" clause is inserted in the script.

In the `SqlListQuery` item:

```
{OWNER, TABLE, INDEX, INDEXTYPE, UNIQUE, INDEXKEY, CLUSTER, WhereNotNull}

select
 tbcreator,
 tbname,
 name,
 case indextype when '2' then 'type 2' else 'type 1' end,
 case uniquerule when 'D' then '' else 'unique' end,
 case uniquerule when 'P' then 'primary' when 'U' then 'unique' else '' end,
 case clustering when 'Y' then 'cluster' else '' end,
 case uniquerule when 'N' then 'TRUE' else 'FALSE' end
from
 sysibm.sysindexes
where 1=1
[  and tbname=%.q:TABLE%]
[  and tbcreator=%.q:OWNER%]
[  and dbname=%.q:CATALOG%]
order by
 1 ,2 ,3
```

### To Add an Extended Attribute:

You can add and extended attribute in the Profile category.

1. Right-click a metaclass category in the Profile category and select **New** > **Extended Attributes** from the contextual menu.

   A new extended attribute is created.
2. Enter a name and comment, and then select a data type from the Data Type dropdown list box.
3. [optional] Select a default value in the Default Value list.
4. Click Apply.

## Using Extended Attributes in the PDM

Extended attributes are used in a DBMS to control model generation.

You can also define extended attributes in an extended model definition, to further define an object. These extended attributes are not used during generation.

For more information on extended model definitions, see *Extended Model Definitions* on page 18.

By default, extended attributes appear in the Extended Attributes tab, in a user-defined tab or in object lists.

Each extended attribute has the following properties:

| Property | Description |
|---|---|
| Name | Name of extended attribute |
| Data type | Extended attribute data type including boolean, color, date, file, float, font, etc or customized data types |
| Value | Value of the extended attribute. This field displays the default value defined for the extended attribute data type |

| Property | Description |
|---|---|
| R | Redefined value. Selected if you modify the default value in the Value column, using either the arrow or the ellipsis button |

The name and data type of the extended attributes can only be edited in the Resource Editor.

### To Define the Value of an Extended Attribute:

You can define the value of an extended attribute from a property sheet or from a list of objects.

1. Open the property sheet of an object.

   *or*

   Select **Model** > **Object** to display a list of objects.
2. Click the Extended Attributes tab or the user-defined tab to display the corresponding tab.

   *or*

   Click the Customize Columns and Filter tool, select extended attributes in the list of columns, and click OK.

   The extended attributes appear in different tabs or as additional columns in the list.
3. Click the Value column of an extended attribute if you want to modify its value and select a value from the list.

   *or*

   Type a value in the corresponding box.

   *or*

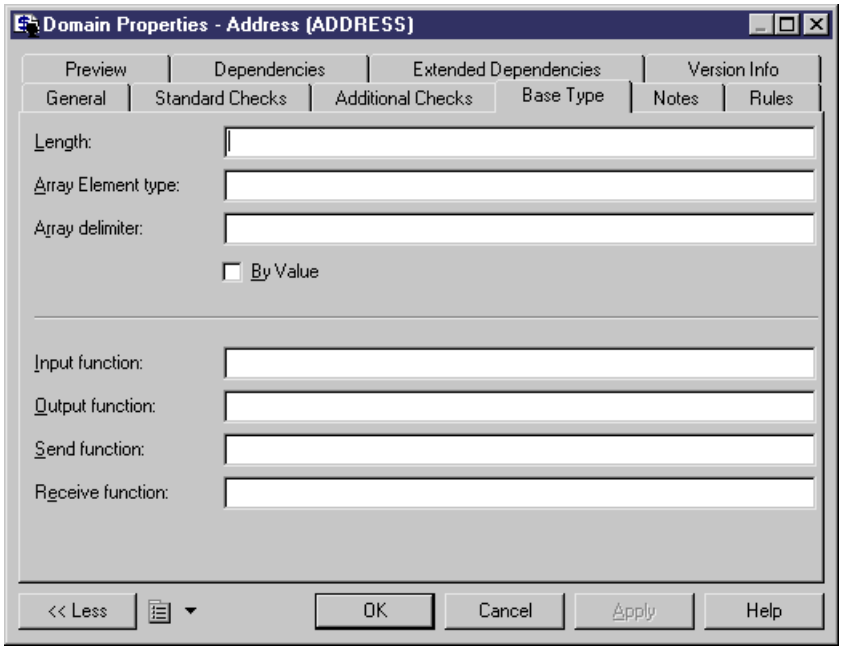
   Type or select a value in the value cell in the list.

   The Ellipsis button allows you to further define the value of the selected extended attribute.
4. Click OK.

## Using Extended Attributes During Generation

Extended attributes are created to control generation: each extended attribute value can be used as a variable that can be referenced in the scripts defined in the Script category.

Some DBMS include predefined extended attributes. For example in PostgreSQL, domains include default extended attributes used for the creation of user-defined data types.



You can create as many extended attributes as you need, for each DBMS supported object.

**Note:** PowerDesigner variable names are case sensitive. The variable name must be an exact match of the extended attribute name.

### Example

In DB2 UDB 7, extended attribute `WhereNotNull` allows you to add a clause that specifies that index names must be unique provided they are not null.

In the `Create index` order, `WhereNotNull` is evaluated as shown below:

create [%INDEXTYPE% ][%UNIQUE% [%WhereNotNull%?where not null ]]index [%QUALIFIER%]%INDEX% on [%TABLQUALIFIER%]%TABLE% (

%CIDXLIST%

)

[%OPTIONS%]

If the index name is unique, and if you set the type of the `WhereNotNull` extended attribute to True, the "where not nul" clause will be inserted in the script.

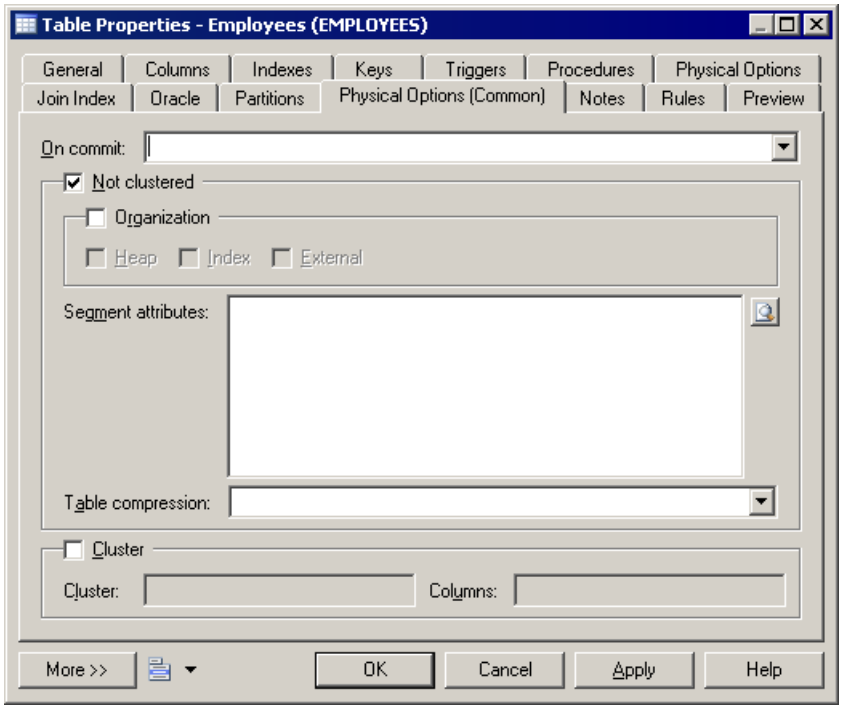
In the `SqlListQuery` item:

```
{{OWNER, TABLE, INDEX, INDEXTYPE, UNIQUE, INDEXKEY, CLUSTER, WhereNotNull}

select
 tbcreator,
 tbname,
 name,
 case indextype when '2' then 'type 2' else 'type 1' end,
 case uniquerule when 'D' then '' else 'unique' end,
 case uniquerule when 'P' then 'primary' when 'U' then 'unique' else '' end,
 case clustering when 'Y' then 'cluster' else '' end,
 case uniquerule when 'N' then 'TRUE' else 'FALSE' end
from
 sysibm.sysindexes
where 1=1
[  and tbname=%.q:TABLE%]
[  and tbcreator=%.q:OWNER%]
[  and dbname=%.q:CATALOG%]
order by
 1 ,2 ,3
```

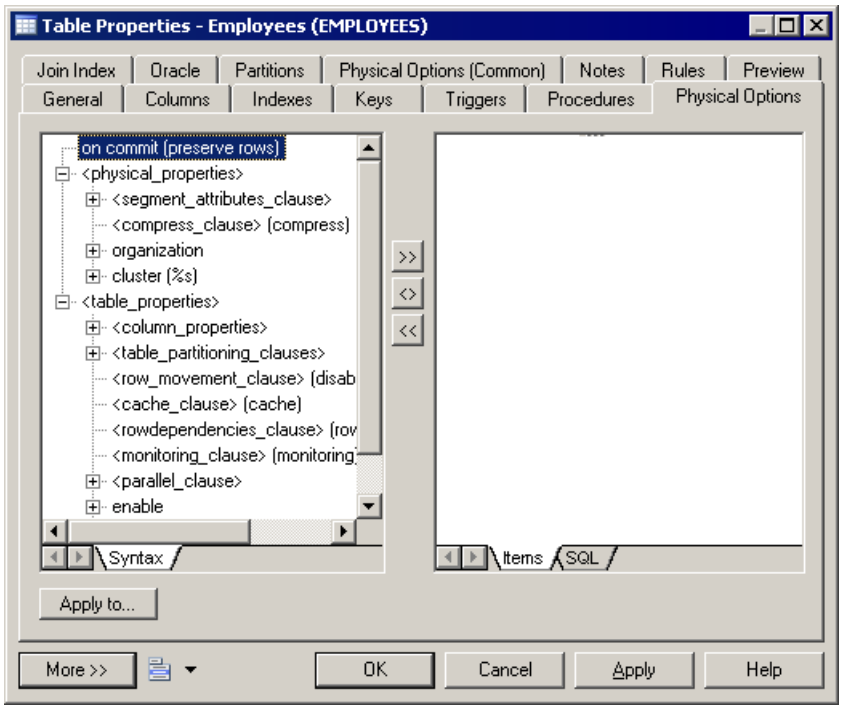
## Physical Options

In some DBMSs, physical options are used to specify how an object is optimized or stored in a database. You define physical options in object property sheets on the following tabs:

- Physical Options (Common) – displays the physical options most commonly set for the object in a standard form format:




- Physical Options – displays all the available physical options for the object in a tree format:




**Note:** The Physical Options (Common) tab is configurable and the options that appear on it are associated with extended attributes. You can add other options to this tab or to your own custom tab by associating them with extended attributes. For more information, see *Adding DBMS Physical Options to Your Forms* on page 144.

For information about setting physical options, see "Physical Options" in the Building Physical Diagrams chapter of the *Data Modeling* guide.

## Physical Option Syntax

If physical options are supported for an object, they are stored in the Options entry beneath the object in the Script/Object category of the DBMS resource file.

For more information, see *Common object items* on page 56. Default values are stored in the DefOptions entry.

During generation, the options selected in the model for each object are stored as a SQL string in the %OPTIONS% variable, which must appear at the end of the Create statement of the object, and cannot be followed by anything else. The following example uses the correct syntax:

```
create table
[%OPTIONS%]
```

During reverse engineering by script, the section of the SQL query determined as being the physical options is stored in %OPTIONS%, and will then be parsed when required by an object property sheet.

During live database reverse engineering, the `SqlOptsQuery` SQL statement is executed to retrieve the physical options which is stored in %OPTIONS% to be parsed when required by an object property sheet.

You can use PowerDesigner variables (see *PDM Variables* on page 107) to set physical options for an object. For example, in Oracle, you can set the following variable for a cluster to make the cluster take the same name as the table.

```
Cluster %TABLE%
```

### Defining Physical Options Specified by a Value

Option items contain text that is used to display the option on the Physical Options tabs. Entries may contain %d or %s variables to let the user specify a value. For example:

```
with max_rows_per_page=%d
on %s: category=storage
```

- the %d variable - requires a numeric value
- %s variable - requires a string value

Variables between % signs (%--%) are not allowed inside physical options.

You can specify a constraint (such as a list of values, default values, the value must be a storage or a tablespace, some lines can be grouped) on any line containing a variable. Constraints are introduced by a colon directly following the physical option and separated by commas.

### Example

`With max_rows_per_page` is a physical option for Sybase ASE 11.x,which limits the number of rows per data page. The syntax is as follows:

```
with max_row_per_page = x
```

The `with max_rows_per_page` option is shown on the Options tabs with a default value of zero (0):

This option is defined in the DBMS definition file as follows:

```
with max_rows_per_page=%d
on %s : category=storage
```

The `%d` and `%s` variables must be in the last position and they must not be followed by other options.

### Physical Options Without Names

A line in an option entry must have a name in order to be identified by PowerDesigner. If a physical option does not have any name, you must add a name between angled brackets <> before the option.

For example, the syntax to define a segment in Sybase ASE 11, is as follows:

```
sp_addsegment segmentname, databasename, devicename
```

`segmentname` corresponds to the storage code defined in PowerDesigner, and `databasename` corresponds to the model code. These two entries are automatically generated. `devicename` must be entered by the user, and becomes an option.

In SYSTEM11, this option is defined as follows:

```
Create = execute sp_addsegment %STORAGE%, %DATABASE%, %OPTIONS%
OPTIONS = <devname> %s
```

Note that a physical option without name must be followed by the `%d` or `%s` variable.

### Defining a Default Value for a Physical Option

A physical option can have a default value specified by the `Default= x` keyword, which is placed after the option name or after the `%d` or `%s` value, and separated by a colon.

### Example

The default value for `max_row_per_page` is 0. In Sybase Adaptive Server® Enterprise 11, this default value for the index object is defined as follows:

```
max_rows_per_page=%d : default=0
```

### Defining a List of Values for a Physical Option

When you use the `%d` and `%s` variables, a physical option value can correspond to a list of possible options specified by the `list= x | y` keyword, which is placed after the option name or after the `%d` or `%s` value, and separated by a colon. Possible values are separated by the `|` character.

For example, the `dup_prow` option of a Sybase ASE 11 index has two mutually exclusive options for creating a non-unique, clustered index:

```
IndexOption =
<duprow> %s: list=ignore_dup_row | allow_dup_row
```

A list with the values is displayed on the Physical Options tabs.

**Note:** If `Default=` and `List=` are used at the same time, they must be separated by a comma. For example IndexOption = <duprow> %s: `default= ignore_dup_row`, list=ignore_dup_row | allow_dup_row

### Defining a Physical Option for a Tablespace or a Storage

A physical option can use the code of a tablespace or a storage. The `Category=tablespace` and `category=storage` options build lists of all the tablespace or storage codes defined in the model.

For example, in Sybase ASE 11, the `on segmentname` option specifies that the index is created on the segment specified. An ASE segment corresponds to a PowerDesigner storage. The syntax is:

```
on segmentname
```

The default value for the index object is defined in option items as follows:

```
on %s: category=storage
```

A list with the values is displayed on the Physical Options tabs.

### Composite Physical Option Syntax

A composite physical option is a physical option that includes other dependent options. These options are selected together in the right pane of the physical options tab.

The standard syntax for composite physical options is as follows:

```
with : composite=yes, separator=yes, parenthesis=no
{
```

```
fillfactor=%d : default=0
max_rows_per_page=%d : default=0
}
```

The `With` physical option includes the other options between curly brackets { }, separated by a comma. To define a composite option, a composite keyword is necessary.

| Keyword | Value and result |
|---|---|
| composite | The following settings are available:<br>• yes - brackets can be used to define a composite physical option<br>• no – brackets cannot be used |
| separator | The following settings are available:<br>• yes - options are separated by a comma<br>• no [default] - options have no separator character |
| parenthesis | The following settings are available:<br>• yes - the composite option is delimited by parenthesis, including all the other options, for example: with (max_row_per_page=0, ignore_dup_key)<br>• no [default] - nothing delimits the composite option |
| nextmand | The following settings are available:<br>• yes - the next line in the physical option is mandatory.<br>• no - you will not be able to generate/reverse the entire composite physical option |
| prevmand | The following settings are available:<br>• yes - the previous line in the physical option is mandatory<br>• no - you will not be able to generate/reverse the entire composite physical option |
| chldmand | The following settings are available:<br>• yes - at least one child line is mandatory<br>• no – children are not mandatory |
| category | The following settings are available:<br>• tablespace - the item is linked to a tablespace<br>• storage - the item is linked to a storage<br>Note that, in Oracle, the storage category is used as a template to define all the storage values in a storage entry. This is to avoid having to set values independently each time you need to use the same values in a storage clause. Thus, the Oracle physical option does not include the storage name (%s):<br>`storage : category=storage, composite=yes, separator=no, parenthesis=yes`<br>`{` |
| list | List in which values are separated by a pipe (\|) |
| dquoted | The following settings are available:<br>• yes - the value is enclosed in double quotes (“” “”)<br>• no - the value is not enclosed in double quotes (“” “”) |

| Keyword | Value and result |
|---|---|
| squoted | The following settings are available:<br>• yes - the value is enclosed in single quotes (' ')<br>• no - the value is not enclosed in single quotes (' ') |
| enabledbprefix | The following settings are available:<br>• yes - the database name is used as prefix (see tablespace options in DB2 OS/390)<br>• no - the database name is not used as prefix |

`Default=` and/or `List=` can also be used with the `composite=, separator=` and `parenthesis=` keywords. `Category=` can be used with the three keywords of a composite option.

### Example

The IBM DB2 index options contain the following composite option:

```
<using_block> : composite=yes
{
 using vcat %s
 using stogroup %s : category=storage, composite=yes
 {
  priqty %d : default=12
  secqty %d
  erase %s : default=no, list=yes | no
 }
```

### Repeating Options Several Times

Certain databases repeat a block of options, grouped in a composite option, several times. In this case, the composite definition contains the multiple `multiple`:

```
with: composite=yes, multiple=yes
```

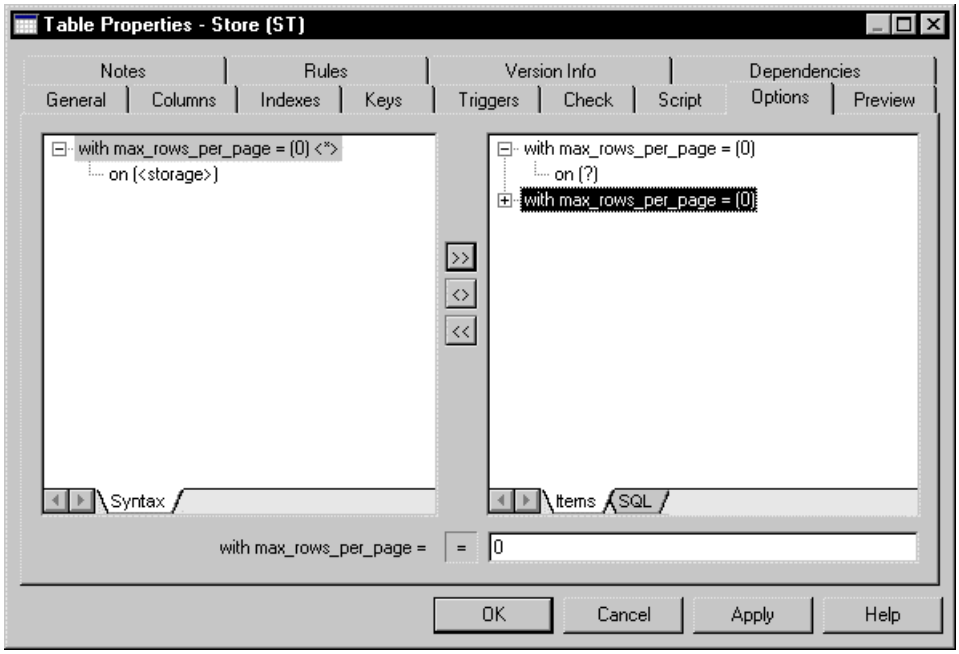
For example, the Informix fragmentation options can be repeated *n* times as follows:

```
IndexOption =
fragment by expression : composite=yes, separator=yes
{
 <list> : composite=yes, multiple=yes
 {
  <frag-expression> %s
  in %s : category=storage
 }
 remainder in %s : category=storage
}
```

The `<list>` sub-option is used to avoid repeating the `fragment` keyword with each new block of options.

When you repeat a composite option, the option is displayed with <*> in the available physical options pane (left pane) of the physical options tab.

```
max_rows_per_page=0 <*>
```




You can add the composite option to the right pane several times using the Add button between the panes of the physical options tab.

If the selection is on the composite option in the right pane and you click the same composite option in the left pane to add it, a message box asks you if you want to reuse the selected option. If you click No, the composite option is added to the right pane as a new line.

# PDM Variables

The SQL queries recorded in the DBMS definition file items make use of various PDM variables. These variables are replaced with values from your model when the scripts are generated, and are evaluated to create PowerDesigner objects during reverse engineering.

PowerDesigner variables are written between percent signs (%).

### Example

```
CreateTable = create table %TABLE%
```

The evaluation of variables depends on the parameters and context. For example, the %COLUMN% variable cannot be used in a CreateTablespace parameter, because it is only valid in a column parameter context.

When referencing object attributes you can use the following variables or, alternately, the public names available through the PowerDesigner metamodel (see *Customizing Generation with GTL* on page 187 and *Resource Files and the Public Metamodel* on page 1.

## Variables for Database Generation, and Triggers and Procedures Generation

You can use the following variables:

| Variable | Comment |
|---|---|
| %DATE% | Generation date & time |
| %USER% | Login name of User executing Generation |

| Variable | Comment |
|---|---|
| %PATHSCRIPT% | Path where File script is going to be generated |
| %NAMESCRIPT% | Name of File script where SQL orders are going to be written |
| %STARTCMD% | Description to explain how to execute Generated script |
| %ISUPPER% | TRUE if upper case generation option is set |
| %ISLOWER% | TRUE if lower case generation option is set |
| %DBMSNAME% | Name of DBMS associated with Generated model |
| %DATABASE% | Code of Database associated with Generated model |
| %USE_SP_PKEY% | Use stored procedure primary key to create primary keys (SQL Server specific) |
| %USE_SP_FKEY% | Use stored procedure foreign key to create primary keys (SQL Server specific) |

## Variables for Reverse Engineering

You can use the following variables:

| Variable | Comment |
|---|---|
| %R% | Set to TRUE during reverse engineering |
| %S% | Allow to skip a word. The string is parsed for reverse but not generated |
| %D% | Allow to skip a numeric value. The numeric value is parsed for reverse but not generated |
| %A% | Allow to skip all Text. The text is parsed for reverse but not generated |
| %ISODBCUSER% | True if Current user is Connected one |
| %CATALOG% | Catalog name to be used in live database connection reverse queries |
| %SCHEMA% | Variable representing a user login and the object belonging to this user in the database. You should use this variable for queries on objects listed in database reverse dialog boxes, because their owner is not defined yet. Once the owner of an object is defined, you can use SCHEMA or OWNER |
| %SIZE% | Data type size of column or domain. Used for live database reverse, when the length is not defined in the system tables |
| %VALUE% | One value from the list of values in a column or domain |
| %PERMISSION% | Allow to reverse engineer permissions set on a database object |
| %PRIVILEGE% | Allow to reverse engineer privileges set on a user, a group, or a role |

## Variables for Database Synchronization

You can use the following variables:

| Variable | Comment |
|---|---|
| %OLDOWNER% | Old owner name of Object. See also OWNER |
| %NEWOWNER% | New owner name of Object. See also OWNER |

| Variable | Comment |
|---|---|
| %OLDQUALIFIER% | Old qualifier of Object. See also QUALIFIER |
| %NEWQUALIFIER% | New qualifier of Object. See also QUALIFIER |
| %OLDTABL% | Old code of Table |
| %NEWTABL% | New code of Table |
| %OLDCOLN% | Old code of Column |
| %NEWCOLN% | New code of Column |
| %OLDNAME% | Old code of Sequence |
| %NEWNAME% | New code of Sequence |

## Variables for Database Security

You can use the following variables:

| Variable | Comment |
|---|---|
| %PRIVLIST% | List of privileges for a grant/revoke order |
| %PERMLIST% | List of permissions for a grant/revoke order |
| %USER% | Name of the user |
| %GROUP% | Name of the group |
| %ROLE% | Name of the role |
| %GRANTEE% | Generic name used to design a user, a group, or a role |
| %PASSWORD% | Password for a user, group, or role |
| %OBJECT% | Database objects (table, view, column, and so on) |
| %GRANTOPTION% | Option for grant: with grant option / with admin option |
| %REVOKEOPTION% | Option for revoke: with cascade |

## Variables for Metadata

You can use the following variables:

| Variable | Comment |
|---|---|
| %@CLSSNAME% | Localized name of Object class. Ex: Table, View, Column, Index |
| %@CLSSCODE% | Code of Object class. Ex: TABL, VIEW, COLN, INDX |

## Common Variables for All Named Objects

You can use the following variables:

| Variable | Comment |
|---|---|
| %@OBJTNAME% | Name of Object |

| Variable | Comment |
|---|---|
| %@OBJTCODE% | Code of Object |
| %@OBJTLABL% | Comment of Object |
| %@OBJTDESC% | Description of Object |

## Common Variables for Objects

These objects can be Tables, Indexes, Views, etc.

| Variable | Comment |
|---|---|
| %COMMENT% | Comment of Object or its name (if no comment defined) |
| %OWNER% | Generated code of User owning Object or its parent. You should not use this variable for queries on objects listed in live database reverse dialog boxes, because their owner is not defined yet |
| %DBPREFIX% | Database prefix of objects (name of Database + '.' if database defined) |
| %QUALIFIER% | Whole object qualifier (database prefix + owner prefix) |
| %OPTIONS% | SQL text defining physical options for Object |
| %CONSTNAME% | Constraint name of Object |
| %CONSTRAINT% | Constraint SQL body of Object. Ex: (A <= 0) AND (A >= 10) |
| %CONSTDEFN% | Column constraint definition. Ex: constraint C1 checks (A>=0) AND (A<=10) |
| %RULES% | Concatenation of Server expression of business rules associated with Object |
| %NAMEISCODE% | True if the object (table, column, index) name and code are identical (AS 400 specific) |

## Variables for DBMS, Database Options

You can use the following variables:

| Variable | Comment |
|---|---|
| %TABLESPACE% | Generated code of Tablespace |
| %STORAGE% | Generated code of Storage |

## Variables for Tables

You can use the following variables:

| Variable | Comment |
|---|---|
| %TABLE% | Generated code of Table |
| %TNAME% | Name of Table |
| %TCODE% | Code of Table |
| %TLABL% | Comment of Table |
| %PKEYCOLUMNS% | List of primary key columns. Ex: A, B |

| Variable | Comment |
|---|---|
| %TABLDEFN% | Complete body of Table definition. It contains definition of columns, checks and keys |
| %CLASS% | Abstract data type name |
| %CLUSTERCOLUMNS% | List of columns used for a cluster |

## Variables for Domains and Columns Checks

You can use the following variables:

| Variable | Comment |
|---|---|
| %UNIT% | Unit attribute of standard check |
| %FORMAT% | Format attribute of standard check |
| %DATATYPE% | Data type. Ex: int, char(10) or numeric(8, 2) |
| %DTTPCODE% | Data type code. Ex: int, char or numeric |
| %LENGTH% | Data type length. Ex: 0, 10 or 8 |
| %PREC% | Data type precision. Ex: 0, 0 or 2 |
| %ISRDONLY% | TRUE if Read-only attribute of standard check has been selected |
| %DEFAULT% | Default value |
| %MINVAL% | Minimum value |
| %MAXVAL% | Maximum value |
| %VALUES% | List of values. Ex: (0, 1, 2, 3, 4, 5) |
| %LISTVAL% | SQL constraint associated with List of values. Ex: C1 in (0, 1, 2, 3, 4, 5) |
| %MINMAX% | SQL constraint associated with Min and max values. Ex: (C1 <= 0) AND (C1 >= 5) |
| %ISMAND% | TRUE if Domain or column is mandatory |
| %MAND% | Contains Keywords "null" or "not null" depending on Mandatory attribute |
| %NULL% | Contains Keyword "null" if Domain or column is not mandatory |
| %NOTNULL% | Contains Keyword "not null" if Domain or column is mandatory |
| %IDENTITY% | Keyword "identity" if Domain or Column is identity (Sybase specific) |
| %WITHDEFAULT% | Keyword "with default" if Domain or Column is with default |
| %ISUPPERVAL% | TRUE if the upper-case attribute of standard check has been selected |
| %ISLOWERVAL% | TRUE if the lower-case attribute of standard check has been selected |

## Variables for Columns

Parent Table variables are also available.

| Variable | Comment |
|---|---|
| %COLUMN% | Generated code of Column |
| %COLNNO% | Position of Column in List of columns of Table |
| %COLNNAME% | Name of Column |
| %COLNCODE% | Code of Column |
| %PRIMARY% | Contains Keyword "primary" if Column is primary key column |
| %ISPKEY% | TRUE if Column is part of Primary key |
| %FOREIGN% | TRUE if Column is part of one foreign key |
| %COMPUTE% | Compute constraint text |
| %NULLNOTNULL% | Mandatory status of a column. This variable is always used with NullRequired item, see *Working with Null values* on page 67 |

## Variables for Abstract Data Types

You can use the following variables:

| Variable | Comment |
|---|---|
| %ADT% | Generated code of Abstract data type |
| %TYPE% | Type of Abstract data type. It contains keywords like "array", "list", ... |
| %SIZE% | Abstract data type size |
| %FILE% | Abstract data type Java file |
| %ISARRAY% | TRUE if Abstract data type is of type array |
| %ISLIST% | TRUE if Abstract data type is of type list |
| %ISSTRUCT% | TRUE if Abstract data type is of type structure |
| %ISOBJECT% | TRUE if Abstract data type is of type object |
| %ISJAVA% | TRUE if Abstract data type is of type JAVA class |
| %ADTDEF% | Contains Definition of Abstract data type |

## Variable for Abstract Data Type Attributes

You can use the following variables:

| Variable | Comment |
|---|---|
| %ADTATTR% | Generated code of Abstract data type attribute |

## Variable for Domains

You can use the following variables:

| Variable | Comment |
|---|---|
| %DOMAIN% | Generated code of Domain (also available for columns) |
| %DEFAULTNAME% | Name of the default object associated with the domain (SQL Server specific) |

## Variables for Rules

You can use the following variables:

| Variable | Comment |
|---|---|
| %RULE% | Generated code of Rule |
| %RULENAME% | Rule name |
| %RULECODE% | Rule code |
| %RULECEXPR% | Rule client expression |
| %RULESEXPR% | Rule server expression |

## Variables for ASE & SQL Server

You can use the following variables:

| Variable | Comment |
|---|---|
| %RULENAME% | Name of Rule object associated with Domain |
| %DEFAULTNAME% | Name of Default object associated with Domain |
| %USE_SP_PKEY% | Use sp_primary key to create primary keys |
| %USE_SP_FKEY% | Use sp_foreign key to create foreign keys |

## Variables for Sequences

You can use the following variables:

| Variable | Comment |
|---|---|
| %SQNC% | Name of sequence |
| %SQNCOWNER% | Name of the owner of the sequence |

## Variables for Indexes

You can use the following variables:

| Variable | Comment |
|---|---|
| %INDEX% | Generated code of Index |
| %TABLE% | Generated code of the parent of an index, can be a table or a query table (view) |
| %INDEXNAME% | Index name |
| %INDEXCODE% | Index code |

| Variable | Comment |
|---|---|
| %UNIQUE% | Contains Keyword "unique" when Index is unique |
| %INDEXTYPE% | Contains Index type (available only for a few DBMS) |
| %CIDXLIST% | List of index columns with separator, on the same line. Example: A asc, B desc, C asc |
| %INDEXKEY% | Contains Keywords "primary", "unique" or "foreign" depending on Index origin |
| %CLUSTER% | Contains Keyword "cluster" when Index is cluster |
| %INDXDEFN% | Used for defining an index within a table definition |

## Variables for Join Indexes (IQ)

You can use the following variables:

| Variable | Comment |
|---|---|
| %JIDX% | Generated code for join index |
| %JIDXDEFN% | Complete body of join index definition |
| %REFRLIST% | List of references (for live database connections) |
| %RFJNLIST% | List of reference joins (for live database connections) |

## Variables for Index Columns

You can use the following variables:

| Variable | Comment |
|---|---|
| %ASC% | Contains Keywords "ASC" or "DESC" depending on sort order |
| %ISASC% | TRUE if Index column sort is ascending |

## Variables for References

You can use the following variables:

| Variable | Comment |
|---|---|
| %REFR% | Generated code of Reference |
| %PARENT% | Generated code of Parent table |
| %PNAME% | Name of Parent table |
| %PCODE% | Code of Parent table |
| %PQUALIFIER% | Qualifier of Parent table. See also QUALIFIER. |
| %CHILD% | Generated code of Child table |
| %CNAME% | Name of Child table |
| %CCODE% | Code of Child table |

| Variable | Comment |
|---|---|
| %CQUALIFIER% | Qualifier of Child table. See also QUALIFIER. |
| %REFRNAME% | Reference name |
| %REFRCODE% | Reference code |
| %FKCONSTRAINT% | Foreign key (reference) constraint name |
| %PKCONSTRAINT% | Constraint name of Primary key used to reference object |
| %CKEYCOLUMNS% | List of parent key columns. Ex: C1, C2, C3 |
| %FKEYCOLUMNS% | List of child foreign key columns. Ex: C1, C2, C3 |
| %UPDCONST% | Contains Update declarative constraint keywords "restrict", "cascade", "set null" or "set default" |
| %DELCONST% | Contains Delete declarative constraint keywords "restrict", "cascade", "set null" or "set default" |
| %MINCARD% | Minimum cardinality |
| %MAXCARD% | Maximum cardinality |
| %POWNER% | Parent table owner name |
| %COWNER% | Child table owner name |
| %CHCKONCMMT% | TRUE when check on commit is selected on Reference (ASA 6.0 specific) |
| %REFRNO% | Reference number in child table collection of references |
| %JOINS% | References joins. |

## Variables for Reference Columns

You can use the following variables:

| Variable | Comment |
|---|---|
| %CKEYCOLUMN% | Generated code of Parent table column (primary key) |
| %FKEYCOLUMN% | Generated code of Child table column (foreign key) |
| %PK% | Generated code of Primary key column |
| %PKNAME% | Primary key column name |
| %FK% | Generated code of Foreign key column |
| %FKNAME% | Foreign key column name |
| %AK% | Alternate key column code (same as PK) |
| %AKNAME% | Alternate key column name (same as PKNAME) |
| %COLTYPE% | Primary key column data type |
| %DEFAULT% | Foreign key column default value |
| %HOSTCOLTYPE% | Primary key column data type used in procedure declaration. For example: without length |

## Variables for Keys

You can use the following variables:

| Variable | Comment |
|---|---|
| %COLUMNSCOLNLIST% | List of columns of Key. Ex: "A, B, C" |
| %ISPKEY% | TRUE when Key is Primary key of Table |
| %PKEY% | Constraint name of primary key |
| %AKEY% | Constraint name of alternate key |
| %KEY% | Constraint name of the key |
| %ISMULTICOLN% | True if the key has more than one column |
| %CLUSTER% | Cluster keyword |

## Variables for Views

You can use the following variables:

| Variable | Comment |
|---|---|
| %VIEW% | Generated code of View |
| %VIEWNAME% | View name |
| %VIEWCODE% | View code |
| %VIEWCOLN% | List of columns of View. Ex: "A, B, C" |
| %SQL% | SQL text of View. Ex: Select * from T1 |
| %VIEWCHECK% | Contains Keyword "with check option" if this option is selected in View |
| %SCRIPT% | Complete view creation order. Ex: create view V1 as select * from T1 |

## Variables for Triggers

Parent Table variables are also available.

| Variable | Comment |
|---|---|
| %ORDER% | Order number of Trigger (in case DBMS support more than one trigger of one type) |
| %TRIGGER% | Generated code of trigger |
| %TRGTYPE% | Trigger type. It contains Keywords "beforeinsert", "afterupdate", ...etc. |
| %TRGEVENT% | Trigger event. It contains Keywords "insert", "update", "delete" |
| %TRGTIME% | Trigger time. It contains Keywords NULL, "before", "after" |
| %REFNO% | Reference order number in List of references of Table |
| %ERRNO% | Error number for standard error |
| %ERRMSG% | Error message for standard error |

| Variable | Comment |
|---|---|
| %MSGTAB% | Name of Table containing user-defined error messages |
| %MSGNO% | Name of Column containing Error numbers in User-defined error table |
| %MSGTXT% | Name of Column containing Error messages in User-defined error table |
| %SCRIPT% | SQL script of trigger or procedure. |
| %TRGBODY% | Trigger body (only for Oracle live database reverse engineering) |
| %TRGDESC% | Trigger description (only for Oracle live database reverse engineering) |
| %TRGDEFN% | Trigger definition |

## Variables for Procedures

You can use the following variables:

| Variable | Comment |
|---|---|
| %PROC% | Generated code of Procedure (also available for trigger when Trigger is implemented with a procedure) |
| %FUNC% | Generated code of Procedure if Procedure is a function (with a return value) |

## Optional Strings and Variables

You can use square brackets [ ] to:

- Include optional strings and variables, or lists of strings and variables in the syntax of SQL statements [%--%]
- Test the value of a variable and insert or reconsider a value depending of the result of the test. [%--%? is true : is false]
- Test the content of a variable [%--%=--? if true : if false]

| Variable | Generation | Reverse |
|---|---|---|
| [%--%] | Generated if variable is defined. If the variable is empty or assigned NO or FALSE it is not generated | Valuated if the parser detects a piece of SQL order corresponding to the variable. If the variable is empty or assigned NO or FALSE it is not valuated |
| [%--%? Is true : Is false]<br>to test the value of the variable (conditional value) | If the variable is not empty, `Is true` is generated, if the variable is empty, `Is false` is generated | If the parser detects Is true, `Is true` is reversed, if the parser detects Is false, `Is false` is reversed and the % % variable is set to True or False respectively |
| [%--%=--? Is true : Is false]<br>to test the content of the variable (conditional value) | If the variable equals the constant value, `Is true` is generated, if the variable is different, `Is false` is generated | If the parser detects Is true, `Is true` is reversed, if the parser detects Is false, `Is false` is reversed |
| [.Z: [s1][s2]...] | .Z is ignored | Specifies that the strings and variables between square brackets are not ordered |
| [.O: [s1][s2]...] | Only first item listed is generated | Specifies that the reverse parser must find one of the listed item to validate the full statement |

### Examples

- [%--%]

```
[%OPTIONS%]
```

If %OPTIONS% is not FALSE, nor empty or assigned NO, the variable is generated, this text is replaced by the value of %OPTIONS% (physical options for the objects visible in the object property sheet).

```
[default %DEFAULT%]
```

In reverse engineering, if a text `default 10` is found during reverse engineering, %DEFAULT% is filled with the value 10. However this specification is not mandatory and the SQL statement is reversed even if the specification is absent. In script generation, if default has a value (10 for example) during generation, the text is replaced by `default 10` otherwise nothing is generated for the block.

- [%--%? Is true : Is false]
  You can use a conditional value for an optional string or variable. The two conditions are separated by a colon within the brackets used with the optional string or variable. For example, [%MAND%?Is true:Is false]. If %MAND% is evaluated as true or filled with a value (different from False or NO) during generation, this text is replaced by `Is true`. If not true, it is replaced by `Is false`.
- [%--%=--? Is true : Is false]
  You can also use keywords to test the content of a variable.

```
[%DELCONST%=RESTRICT?:[on delete %DELCONST%]]
```

- Create table abc (a integer not null default 99)

```
Create table abc (a integer default 99 not null)
```

Both creation orders are identical but attributes are inverted.

Usually, the target XDB file supports only one notation with a specific order in the strings and variables. If you reverse engineer both orders, one of them will not go through because of the variable order. You can bypass this limitation using the .Z macro in the following way:

```
%COLUMN% %DATATYPE%[.Z: [%NOTNULL%][%DEFAULT%]]
```

With this macro, the reverse engineering parser no longer considers order within variables.

- [.O:[procedure][proc]]
  This statement will generate "procedure".
  During reverse engineering, the parser will match either "procedure" and "proc" keywords, if none of them is present in the script, matching will fail.

### Use of Strings

A string between square brackets is always generated; however, whether this string is present or not in the SQL statement will not cancel the reverse engineering of the current statement since it is optional in the SQL syntax of the statement. For example, the syntax for creating a view includes a string:

```
create [or replace] view %VIEW% as %SQL%
```

When you reverse a script, if it contains only `create` or `create or replace`, in both situations the statement is reversed because the string is optional.

## Variable Formatting Options

Variables have a syntax that can force a format on their values. Typical uses are as follows:

- Force values to lowercase or uppercase characters
- Truncate the length of values

- Enquote text

You embed formatting options in variable syntax as follows:

```
%[[?][-][width][.[-]precision][c][H][F][U|L][T][M][q][Q]:]<varname>%
```

The variable formatting options are the following:

| option | Description |
|---|---|
| ? | Mandatory field, if a null value is returned the translate call fails |
| *n* (where n is an integer) | Blanks or zeros added to the right to fill the width and justify the output to the left |
| -*n* | Blanks or zeros added to the left to fill the width and justify the output to the right |
| width | Copies the specified minimum number of characters to the output buffer |
| .[-]precision | Copies the specified maximum number of characters to the output buffer |
| .L | Lower-case characters |
| .U | Upper-case characters |
| .F | Combined with L and U, applies conversion to first character |
| .T | Leading and trailing white space trimmed from the variable |
| .H | Converts number to hexadecimal |
| .c | Upper-case first letter and lower-case next letters |
| .*n* | Truncates to *n* first characters |
| .-*n* | Truncates to *n* last characters |
| M | Extracts a portion of the variable name, this option uses the width and precision parameters to identify the portion to extract |
| q | Enquotes the variable (single quotes) |
| Q | Enquotes the variable (double quotes) |

You can combine format codes. For example, %.U8:CHILD% formats the code of the child table with a maximum of eight uppercase letters.

## Example

The following examples show format codes embedded in the variable syntax for the constraint name template for primary keys, using a table called CUSTOMER_PRIORITY:

| Format | Use |
|---|---|
| .L | Lower-case characters.<br>Example: PK_%.L:TABLE%<br>Result: PK_customer_priority |
| .U*n* | Upper-case characters + left justify variable text to fixed length where *n* is the number of characters.<br>Example: PK_%.U12:TABLE%<br>Result: PK_CUSTOMER_PRI |

| Format | Use |
|---|---|
| .T | Trim the leading and trailing white space from the variable.<br>Example: PK_%.T:TABLE%<br>Result: PK_customer_priority |
| *.n* | Maximum length where *n* is the number of characters.<br>Example: PK_%.8:TABLE%<br>Result: PK_Customer |
| *-n* | Pad the output with blanks to the right to display a fixed length where *n* is the number of characters.<br>Example: PK_%-20:TABLE%<br>Result: PK_ Customer_priority |
| M | Extract a portion of a variable.<br>Example: PK%3.4M:TABLE%<br>Result: PK_CUST |

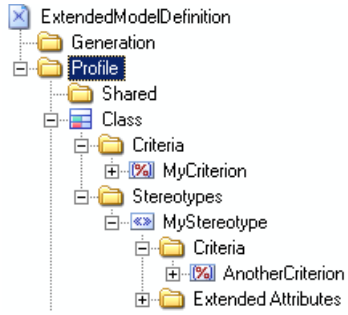
# Extending Your Models with Profiles

All PowerDesigner resource files contain a Profile category directly beneath root. A profile is a UML extension mechanism, which is used for extending a metamodel for a particular target.

Profiles are used in PowerDesigner for adding additional metadata to objects and creating new kinds of links between them, sub-dividing object types (via stereotypes and criteria), customizing symbols, menus, and forms, and modifying generation output. For example:

- The Java 5.0 object language resource file - extends the Component metaclass via several levels of criteria to model various forms of EJBs.
- The BPEL4WS 1.1 process language resource file - extends the Event metaclass through stereotypes to model Compensation, Fault, and Timer events.
- The MSSQLSRV2005 DBMS resource file - uses stereotyped extended objects in order to model aggregates, assemblies, and other SQL Server-specific objects.

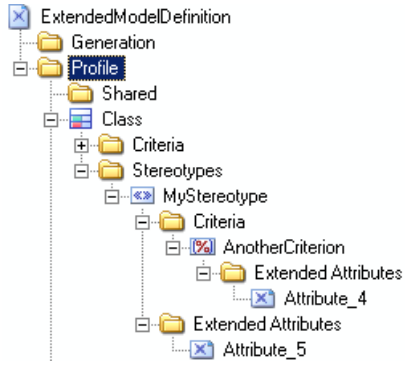
You can review and edit the profile in a resource file by opening it in the Resource Editor and expanding the top-level Profile category. You can add extensions to a metaclass (a type of object, such as Class in an OOM or Table in a PDM), or to a stereotype or criterion, which has previously been defined on a metaclass:




In the example above:

- Class is a metaclass. Metaclasses are drawn from the PowerDesigner metamodel, and always appear at the top level, directly beneath the Profile category
- MyCriterion is a criterion that refines the Class metaclass. Those classes that meet the criterion can be presented and processed differently from other classes.
- MyStereotype is a stereotype that refines the Class metaclass. Those classes that bear the MyStereotype stereotype can be presented and processed differently from other classes.
- AnotherCriterion is a criterion that refines further those classes that bear the MyStereotype stereotype. Classes bearing the stereotype AND meeting the criterion can be presented and processed differently from those that merely bear the stereotype.

Extensions are inherited, so that any extensions made to a metaclass are available to its stereotyped children, and those that are subject to criteria.



Thus, in the example above, classes that bear the MyStereotype stereotype have available the Attribute_5 extended attribute, and those that bear this stereotype AND meet AnotherCriterion have Attribute_4 and Attribute_5 available.

## Extending Objects in the Profile

You can extend the metamodel in the following ways:

- Add or sub-classify new kinds of objects:
  - *Metaclasses (Profile)* on page 122 – to sub-classify objects.
  - *Stereotypes (Profile)* on page 124 [for metaclasses and stereotypes only] – to sub-classify objects.
  - *Criteria (Profile)* on page 128 – to evaluate conditions to sub-classify objects.
  - *Extended Objects, Sub-Objects, and Links (Profile)* on page 131 – to create new kinds of objects.
- Provide new ways of viewing connections between objects:
  - *Dependency Matrices (Profile)* on page 129 – to show connections between two types of objects.
- Add new properties to objects:
  - *Extended Attributes (Profile)* on page 132 – to provide extra metadata.
  - *Extended Collections and Compositions (Profile)* on page 137 – to enable manual linking between objects.
  - *Calculated Collections (Profile)* on page 139 – to automate linking between objects.
  - *Forms (Profile)* on page 141 – to display custom property tabs or dialog boxes.
  - *Custom Symbols (Profile)* on page 150 – to help you visually distinguish objects.
- Add constraints and validation rules to objects:
  - *Custom Checks (Profile)* on page 151 – to provide data testing.
  - *Event Handlers (Profile)* on page 156 – to invoke methods when triggered by an event.
- Execute commands on objects:
  - *Methods (Profile)* on page 159 – to be invoked by other profile extensions such as menus and form buttons (written in VBScript).
  - *Menus (Profile)* on page 161 [for metaclasses and stereotypes only] – to customize PowerDesigner menus.
- Generate objects in new ways:
  - *Templates and Generated Files (Profile)* on page 163 – to customize generation.
  - *Transformations and Transformation Profiles (Profile)* on page 165 – to automate changes to objects at generation or on demand.

**Note:** Since you can attach several resource files to a model (for example, a target language and one or more extended model definitions) you can create conflicts, where multiple extensions with identical names (for example, two different stereotype definitions) are defined on the same metaclass in separate resource files. In case of such conflicts, the extended model definition extension usually prevails. When two XEMs are in conflict, priority is given to the one highest in the list.

## Metaclasses (Profile)

Metaclasses are classes drawn from the PowerDesigner metamode, and appear at the top level of the Profile category.

Concrete metaclasses are defined for specific object types that can be created in a model, while abstract metaclasses are never instantiated but are instead used to define common extensions. For example BasePackage is an ancestor to both model and package.

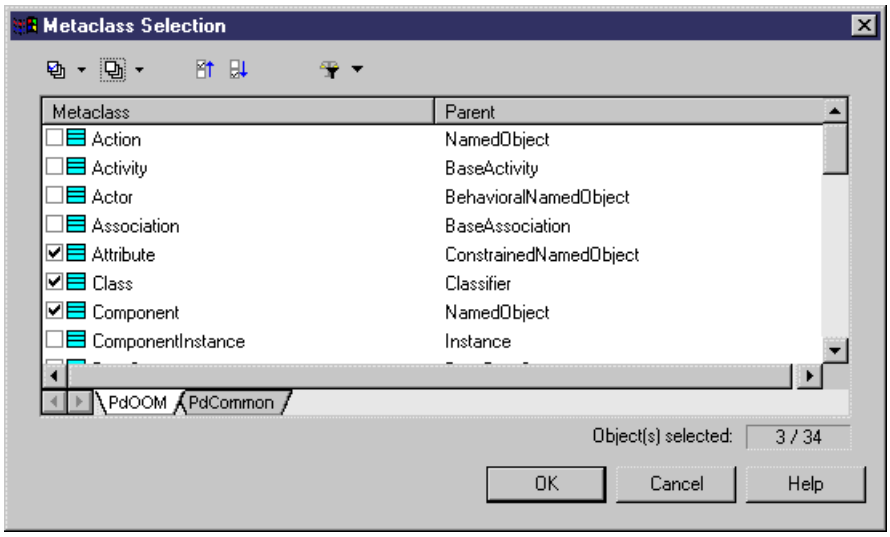
For mode information, see *Resource Files and the Public Metamodel* on page 1.

**Note:** These are special metaclasses that can be used to create entirely new kinds of objects. See *Extended Objects, Sub-Objects, and Links (Profile)* on page 131.

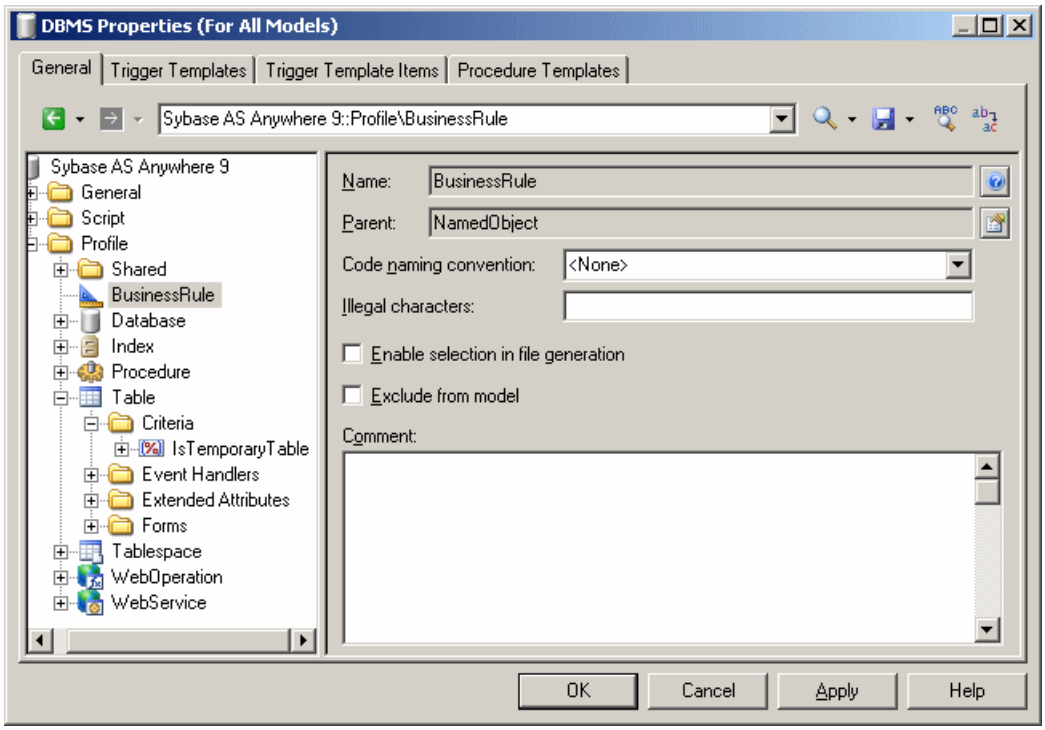
## To Add a Metaclass to a Profile:

You can create a metaclass in a profile.

1. Right-click the Profile category and select Add Metaclasses from the contextual menu to open the Metaclass Selection dialog box:




2. Select one or several metaclasses to add to the profile. You can use the sub-tabs to switch between metaclasses belonging to the present module (for example, the OOM), and standard metaclasses belonging to the PdCommon module. You can also use the Modify Metaclass Filter tool to display all metaclasses, or only concrete or abstract conceptual metaclasses in the list.
3. Click OK to add the selected metaclasses to your profile:




## Metaclass Properties

The following properties are available for metaclasses:

| Property | Description |
|---|---|
| Name | [not editable] Specifies the name of the metaclass.<br>Click the button to the right of this field to open the Metamodel Objects Help for the metaclass. |
| Parent | [not editable] Specifies the parent of the metaclass.<br>Click the button to the right of this field to open the parent metaclass properties. If the parent metaclass is not present in the profile, a message invites you to add it automatically. |
| Code naming convention | [concrete metaclasses only] Specifies a name to code conversion script for instances of the metaclass.<br>The following conversion scripts are available:<br>• firstLowerWord - First word in lowercase, then other first letters of other words in uppercase<br>• FirstUpperChar - First character of all words in uppercase<br>• lower_case - All words in lowercase and separated by an underscore<br>• UPPER_CASE - All words in uppercase and separated by an underscore<br>For more information on conversion scripts and naming conventions, see "Naming Conventions" section in the Models chapter of the *Core Features Guide*. |
| Illegal characters | [concrete metaclasses only] Specifies a list of illegal characters that may not be used in code generation for the metaclass. The list must be placed between double quotes, for example:<br>`"/!=<>""'()"`<br>When working with an OOM, this object-specific list overrides any values specified in the IllegalChar parameter for the object language (see *Settings Category: Object Language* on page 8). |
| Enable selection in file generation | Specifies that the corresponding metaclass instances will appear in the Selection tab of the extended generation dialog box. If a parent metaclass is selected for file generation, children metaclasses also appear in the Selection tab. |
| Exclude from model | [concrete metaclasses only] Prevents the creation of instances of the metaclass in the model and removes all references to the metaclass from the menus, palette, property sheets and so on, to simplify the interface. For example, if you do not use business rules, you can select this check box in the business rule metaclass page to hide them.<br>When several resource files are attached to a model, the metaclass is excluded if at least one file excludes it and the others do not explicitly enable it. For models that already have instances of this metaclass, the objects will be preserved but it will not be possible to create new ones. |
| Comment | Specifies a descriptive comment for the metaclass. |

## Stereotypes (Profile)

*Stereotypes* are a per-instance extension mechanism. When a stereotype is applied to a metaclass instance (by selecting it in the Stereotype field of the object's property sheet), any extensions that you add to the stereotype are then applied to the instance.

Stereotypes can be promoted to the status of *metaclasses* to give them greater visibility in the interface, with a specific list, Browser category and, optionally custom symbol and palette tool. For more information, see *Defining a stereotype as a metaclass* on page 126.

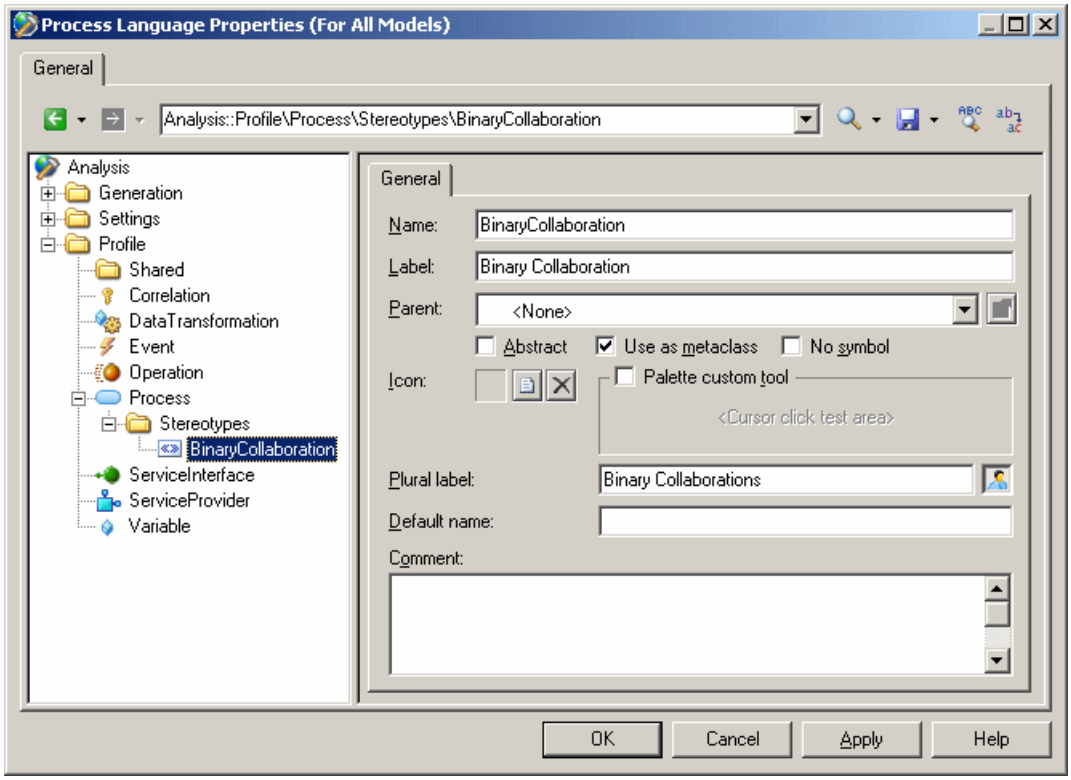
You can define more than one stereotype for a given metaclass, but you can only apply a single stereotype to each instance. Stereotypes support *inheritance*: extensions to a parent stereotype are inherited by its children.

### To Create a Stereotype:

You can create a stereotype within a metaclass, a criterion, or another stereotype.

1. Right-click a metaclass, criterion, or stereotype, and select **New** > **Stereotype** in the contextual menu.

   A new stereotype is created with a default name.
2. Type a stereotype name in the Name box, and fill in any of the other properties that are relevant.




Once you have created the stereotype, you can define extensions like a custom tool, or custom checks for the stereotype. These extensions will apply to all metaclass instances that carry the stereotype.

## Stereotype Properties

The following properties are available for stereotypes:

| Property | Description |
|---|---|
| Name | Specifies the internal name of the stereotype, which can be used for scripting. |
| Label | Specifies the display name of the stereotype, which will appear in the PowerDesigner interface. |
| Parent | Specifies a parent stereotype of the stereotype. You can select a stereotype defined in the same metaclass or in a parent metaclass. Click the Properties button to go to the parent stereotype in the tree and display its properties. |
| Abstract | Specifies that the stereotype cannot be applied to metaclass instances. The stereotype will not appear in the stereotype list in the object property sheet, and can only be used as a parent of other child stereotypes. If you select this property, the Use as metaclass check box is not available. |
| Use as metaclass | Specifies that the stereotype is a sub-classification for instances of the selected metaclass. The stereotype will have its own list of objects and Browser category, and its own tab in multi-pane selection boxes such as those used for generation. For more information, see *Defining a stereotype as a metaclass* on page 126. |
| No Symbol | [available when Use as metaclass is selected] Specifies that when instances of the stereotyped metaclass are created, they will not have diagram symbols. This can be useful when you want to model sub-objects or other objects that do not need to appear in the diagram. The Palette custom tool option is disabled when this option is selected. |

| Property | Description |
|---|---|
| Icon | Specifies an icon for stereotyped instances of the metaclass. Click the tools to the right of this field in order to browse for .cur or .ico files. |
| Palette custom tool | Associates a tool in a palette to the current stereotype. This option is available for objects supporting symbols, it cannot be used for the stereotype of an attribute for example. For more information, see *Attaching a tool to a stereotype* on page 127. |
| Plural label | [available when Use as metaclass is selected] Specifies the plural form of the display name that will appear in the PowerDesigner interface. |
| Default name | [available when Use as metaclass or Palette Custom Tool is selected] Specifies a default name for objects created. A counter will be automatically appended to the name specified to generate unique names.<br>A default name can be useful when designing for a target language or application with strict naming conventions. Note that the default name does not prevail over model naming conventions, so if a name is not correct it is automatically modified. |
| Comment | Additional information about the stereotype. |

## Defining a Stereotype as a Metaclass

You can promote a stereotype to metaclass status by selecting the Use as Metaclass check box in the stereotype property page. This can be useful when you need to:

- Create new kinds of objects that share much of the behavior of an existing object type, such as business transactions and binary collaborations in a BPM for ebXML.
- Have objects with identical names but different stereotypes in the same namespace (a metaclass stereotype creates a sub-namespace in the current metaclass).

**Note:** Stereotypes defined on sub-objects (such as table columns or entity attributes), cannot be turned into metaclass stereotypes.

In the Stereotype property page, select the Use as metaclass check box

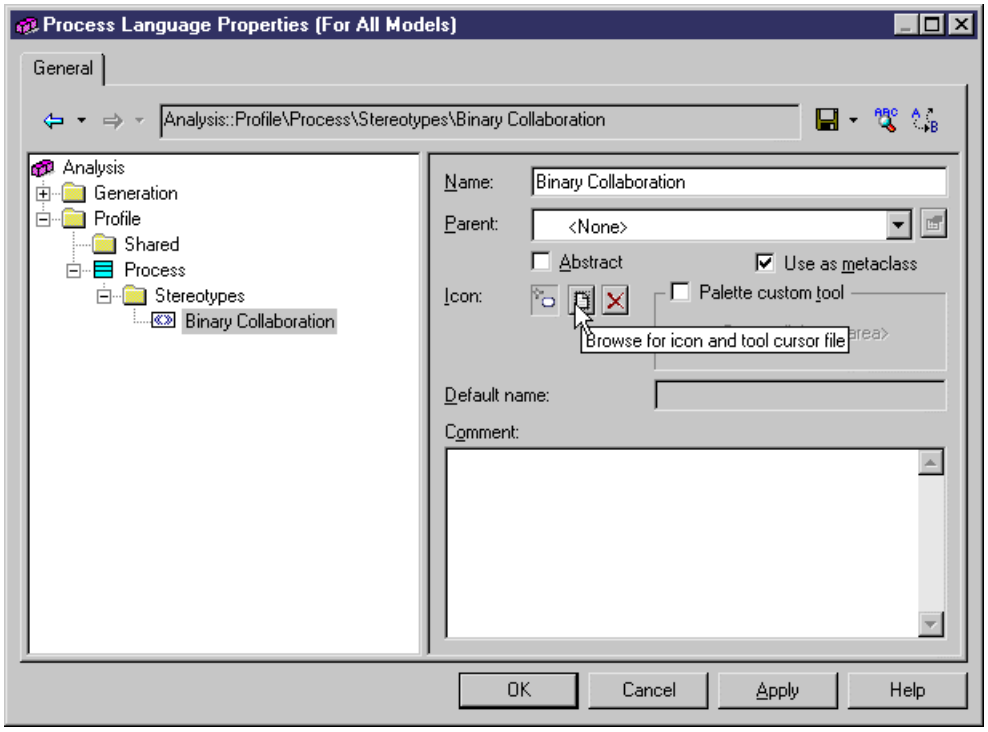
The new metaclass stereotype behaves like a standard PowerDesigner metaclass, and has:

- A separate list in the Model menu - the parent metaclass list will not display objects with the metaclass stereotype. These objects will be displayed in a separate list, under the parent metaclass list. Objects created in the new list bear the new metaclass stereotype by default. If you change the stereotype, the object will be removed from the list the next time it is opened.
- Its own Browser folder and command in the New contextual menu.
- Property sheet titles based on the metaclass stereotype.

## Attaching an Icon to a Stereotype

You can attach an icon to the stereotype you have defined in order to identify stereotyped instances of the metaclass. You can create your own icons or cursors using 3rd party editors or you can purchase them from graphic designers.

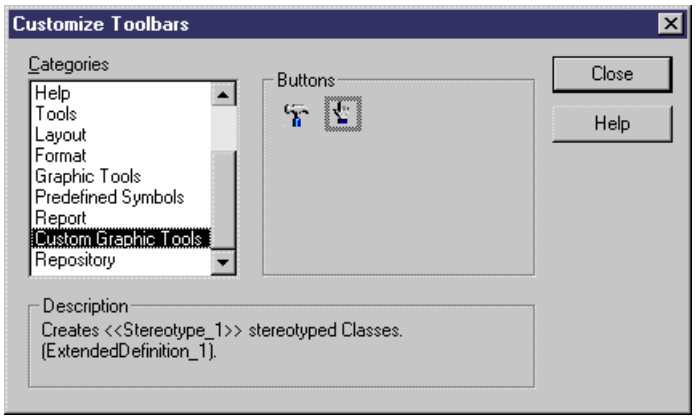
1. In the Stereotype property page, click the Browse for Icon and Tool Cursor File tool to display a standard Open dialog box in which you can select a file with the .cur or .ico extension.




2. Click Apply.

   If you select the Palette Custom Tool check box, the icon is automatically initialized with the default system icon, which you can change with the browse tool.

   When you select a new icon, this icon is copied and saved in the resource file. It is displayed in the list of icons available for the Custom Graphic Tools category in the Customize Toolbars dialog box:




## Attaching a Tool to a Stereotype

You can attach a tool to the stereotype you have defined in order to simplify the creation of stereotyped instances of the metaclass. Custom tools appear in a tool palette named after the resource file to which they belong.

The tool is identical to the stereotype icon. If you do not select an icon, the default system icon is assigned to the tool. You have to select an icon to modify the custom tool of the stereotype.

For more information on how to attach an icon to a stereotype, see *Attaching an icon to a stereotype* on page 126.

1. In the Stereotype property page, select the Palette Custom Tool check box to enable the fields in the lower part of the dialog box.

2. [optional] Select an icon to modify the default tool. You can click inside the <Cursor Click Text Area> to verify how the cursor looks.
3. [optional] Type a default name in the corresponding box, and then
4. Click Apply to save the changes.

# Criteria (Profile)

You can control the treatment of metaclass instances based on whether they conform to one or more criteria. Whereas you can apply only one stereotype to a metaclass instance, you can test the instance against multiple forms of criteria.

You define one or several criteria for a selected metaclass. Criteria let you define the same extensions as stereotypes.

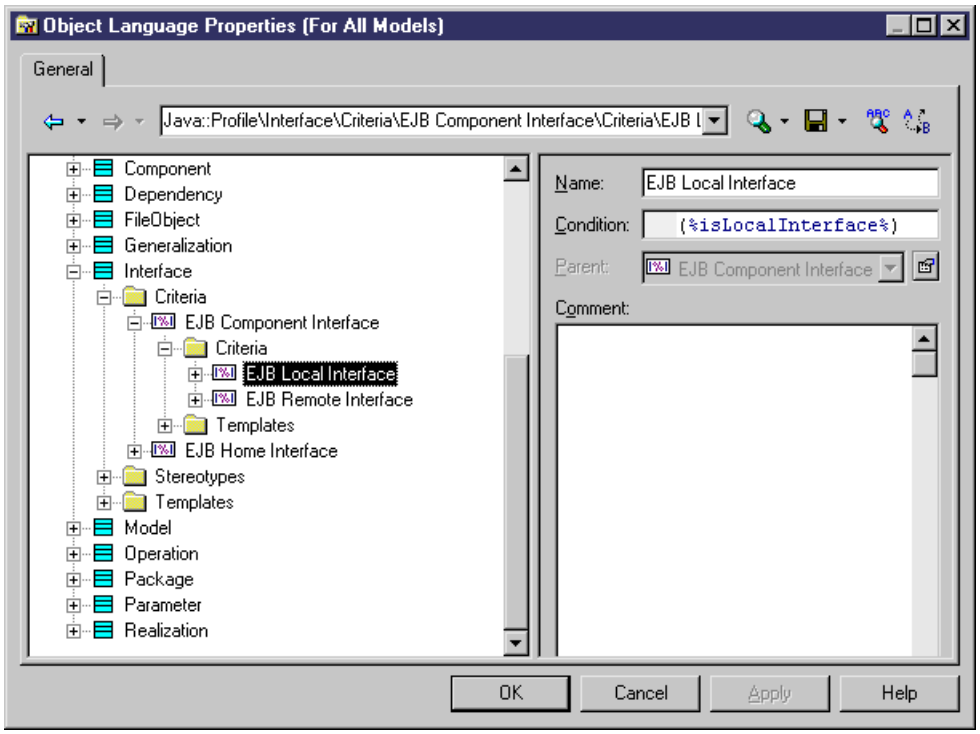
When a metaclass instance meets the criterion condition, the extensions defined on the criterion are applied to this instance. In case of sub-criteria, both the criterion and sub-criterion conditions must be met for the relevant extensions to be applied to the instance.

## To Create a Criterion:

You can create a criterion in a profile

1. Right-click a metaclass and select **New** > **Criterion** in the contextual menu.

   A new criterion is created with a default name.
2. Modify the default name in the Name box, and type a condition in Condition box. You can use any valid expression used by the .if macro (see *.if Macro* on page 213).




3. Click Apply to save your changes.

## Criterion Properties

The following properties are available for criteria:

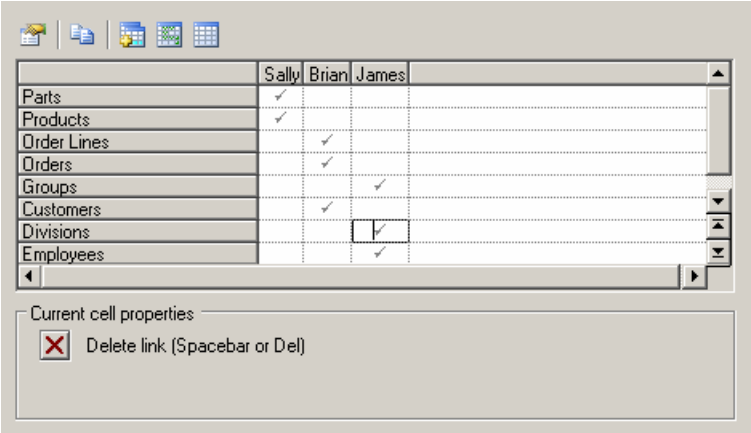
| Property | Description |
|---|---|
| Name | Specifies the name of the criterion. |
| Condition | Specifies the condition which instances must meet in order to access the criterion extensions. You can use any expressions valid for the PowerDesigner GTL .if macro (see *Customizing Generation with GTL* on page 187. You can reference the extended attributes defined at the metaclass level in the condition, but not those defined in the criterion itself.<br>For example, in a PDM, you can customize the symbols of fact tables by creating a criterion that will test the type of the table using the following condition:<br>`(%DimensionalType% == "1")`<br>DimensionalType is an attribute of the BaseTable object, which has a set of defined values, including "1", which corresponds to "fact". For more information, select **Help** > **Metamodel Objects Help** , and navigate to **Libraries** > **PdPDM** > **Abstract Classes** > **BaseTable** . |
| Parent | Specifies a parent criterion of the criterion. You can select a criterion defined in the same metaclass or in a parent metaclass. Click the Properties tool to go to the parent in the tree and view its properties. |
| Comment | Additional information about the criterion. |

# Dependency Matrices (Profile)

Dependency matrices allow you to review and create links between any kind of objects. You specify one metaclass (with, optionally, a stereotype) for the matrix rows, and the same or another metaclass for the columns. The cells are then calculated from a collection or link object.

For example, you could create dependency matrices that show links between:

- OOM Classes and Classes – connected by Association link objects
- PDM Tables and Users – connected by the Owner collection




- PDM Tables and OOM Classes – connected by extended dependencies

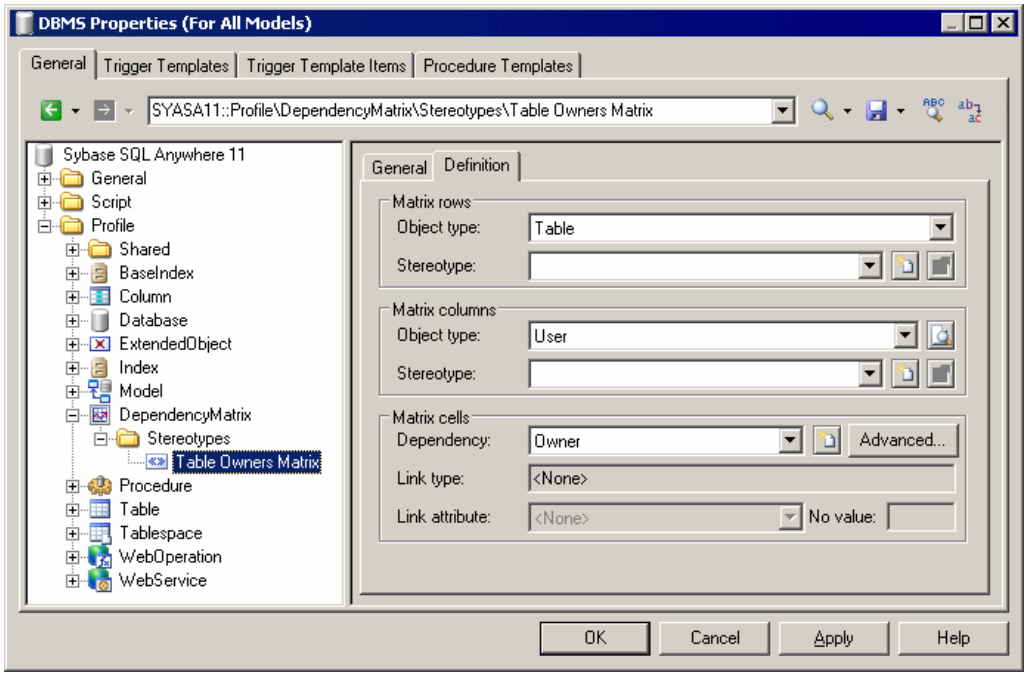
## Creating a Dependency Matrix

You can create a dependency matrix in a profile.

1. Right-click the Profile category and select Add Dependency Matrix. This adds the DependencyMatrix metaclass to the profile and creates a stereotype under it, in which you will define the matrix properties.
2. Enter a name for the matrix (for example "Table Owners Matrix") along with a label and plural label for use in the PowerDesigner interface, as well as a default name for the matrices that users will create based on this definition.
3. Click the Definition tab to specify the rows and columns of your matrix.

4. Select an object type from the current model to populate your matrix rows and an object type from the current or another model type to populate the columns.
5. Specify how the rows and columns of your matrix will be associated by selecting a dependency from the list.

   Only direct dependencies are available from the list. To specify a more complex dependency, click the Advanced button to open the Dependency Path Definition dialog (see *Specifying Advanced Dependencies* on page 130).




6. If you select a dependency in the form of a link, the type of the link will be displayed and you can select an attribute of the link to display in the matrix cells.
7. Click OK to save your matrix and close the resource editor.

   You can now create instances of the matrix in your model as follows:

   - Select **View > Diagram > New Diagram > Matrix Name** .
   - Right-click a diagram background and select **Diagram > New Diagram > Matrix Name** .
   - Right-click the model in the browser and select **New > Matrix Name** .

   **Note:** For information about using dependency matrices, see "Dependency Matrices" in the Diagrams and Symbols chapter of the *Core Features Guide*.

## Specifying Advanced Dependencies

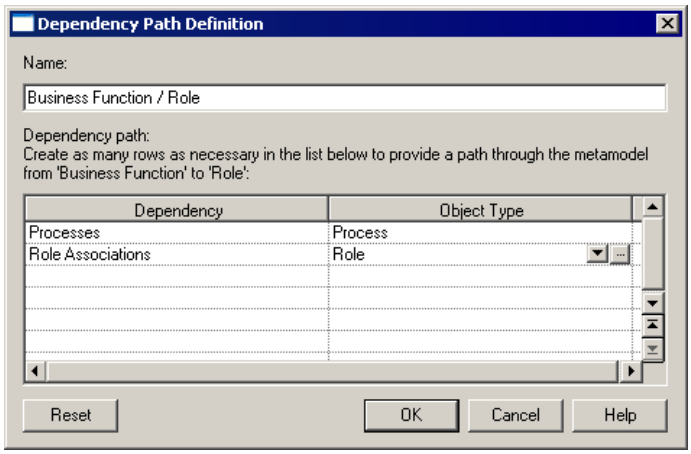
You can examine dependencies between two types of objects that are not directly associated with each other, using the Dependency Path Definition dialog, which is accessible by clicking the Advanced button on the Definition tab, and which allows you to specify a path passing through as many intermediate linking objects as necessary.

Each line in this dialog represents one step in a dependency path:

| Property | Description |
|---|---|
| Name | Specifies a name for the dependency path. By default, this field is populated with the origin and destination object types. |
| Dependency | Specifies the dependency for this step in the path. The list is populated with all the possible dependencies for the previous object type. |

| Property | Description |
|---|---|
| Object Type | Specifies the specific object type that is linked to the previous object type by the selected dependency. This field is autopopulated if only one object type is available through the selected dependency. |

In the following example, a path is identified between business functions and roles, by passing from the business function through the processes it contains, to the role linked to it by a role association:




## Dependency Matrix Properties

Dependency matrices are based on stereotypes.

For information about the properties on the General tab, see *Stereotype properties* on page 125. The following properties are available on the Dependency Matrix Definition tab:

| Property | Description |
|---|---|
| Matrix Rows | Specifies the object type with which to populate your matrix rows.<br>You can, optionally, specify a stereotype to restrict the objects that can appear in the matrix. Click the Create button to the right of the list to create a new stereotype (see *Stereotypes (Profile)* on page 124) to filter by. |
| Matrix Columns | Specifies the object type to populate your matrix columns. Click the Select Metaclass button to the right of the list to select a metaclass from another model type.<br>You can, optionally, specify a stereotype to restrict the objects that can appear in the matrix. Click the Create button to the right of the list to create a new stereotype (see *Stereotypes (Profile)* on page 124) to filter by. |
| Matrix Cells | Specifies how the rows and columns of your matrix will be associated by selecting one of the Matrix Cells radio buttons:<br>• Dependency – select a collection from the list, which contains all standard and extended collections (but not compositions) defined between the two objects. Click the Create button to the right of the list to create a new extended collection (see *Extended Collections and Compositions (Profile)* on page 137) between them, or the Advanced button to specify a complex dependency path (see *Specifying Advanced Dependencies* on page 130).<br>• Link type – select a link object from the list, which contains all the kinds of link possible between the objects. You can optionally specify an attribute of the link to display in the matrix and a symbol to signify a null value for that attribute. |

# Extended Objects, Sub-Objects, and Links (Profile)

Extended objects, sub-objects, and links are special metaclasses that are designed to allow you to add completely new types of objects to your models, rather than basing them on existing PowerDesigner objects.

For more information on metaclasses, see *Metaclasses (Profile)* on page 122. You should use extended objects, sub-objects, and links as follows:

- Extended objects – can be created anywhere
- Extended sub-objects – can only be created in the property sheet of their parent object where they are defined via an extended composition (see *Extended Collections and Compositions (Profile)* on page 137)
- Extended links – can be defined to link extended objects

### Adding the Extended Object, Sub-object, and Link Metaclasses to a Profile

By default, extended objects and links do not appear in models other than the free model. You have to add them in the Profile category of the resource file attached to your model. The resource file can be an extended model definition for those models, such as the CDM that do not support target languages.

Once added, you can refine the extended objects, sub-objects, and links using stereotypes (see *Stereotypes (Profile)* on page 124, and extend them in the same ways as you would other metaclasses (see *Extending objects in the profile* on page 122).

1. Open your resource file in the resource editor.

   You may have to create and attach an extended model definition if the current model is a CDM.
2. Right-click the Profile category and select Add Metaclasses in the contextual menu to open the Metaclass Selection dialog box, and click the PdCommon tab at the bottom of the dialog box to display the list of objects common to all models.
3. Select one or more of the ExtendedLink, ExtendedSubObject, and ExtendedObject check boxes and click OK.

   The metaclasses are added to the profile.

### Adding the Extended Object and Link Tool to the Palette

The tools for creating extended objects and extended links do not appear by default in the palette of models other than the free model.

**Note:** There is no tool for creating extended sub-objects in the diagram, as extended sub-objects can only be created in the property sheet of their parent object (see *Extended Collections and Compositions (Profile)* on page 137).

1. Select **Tools** > **Customize Toolbars** to open the Toolbars dialog box.
2. Select the Palette check box in the toolbars list and click the Customize button to display the Customize Toolbars window.
3. Select the Graphical Tools category and drag the tools corresponding to the extended object or the extended link to the palette of your model, and release the mouse button.

   The tools appear in the palette of your model.
4. Click Close in each of the dialog boxes.

## Extended Attributes (Profile)

Extended attributes allow you to define additional metadata for your objects, and can be defined for metaclasses, stereotypes, and criteria, in order to:

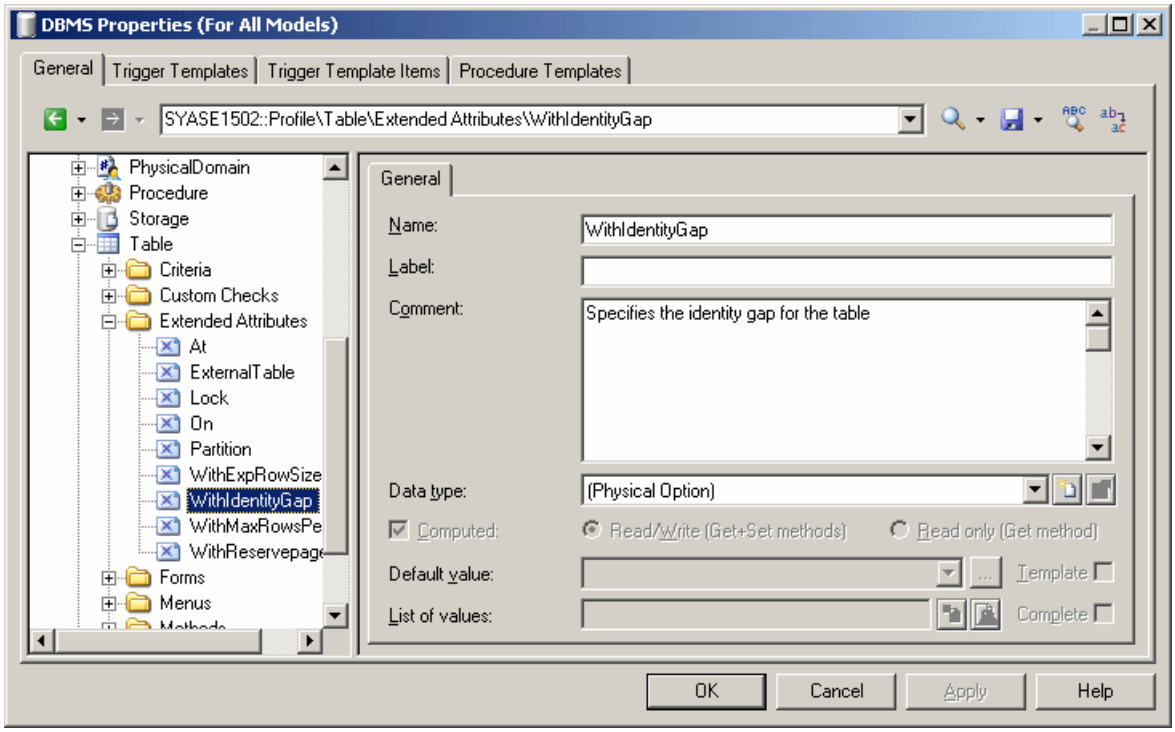
- *Control generation* for a given generation target. In this case, extended attributes are defined in the target language or DBMS of the model. For example, in the Java object language, several metaclasses have extended attributes used for generating Javadoc comments.
- *Further define model objects* in extended model definitions. For example, in the extended model definition for Sybase ASA Proxy tables, the extended attribute called GenerateAsProxyServer in the DataSource metaclass is used to define the data source as a proxy server.

**Note:** By default, extended attributes are listed on a generic Extended Attributes tab in the object property sheet. You can customize the display of attributes by inserting them into forms (see *Forms (Profile)* on page 141). If all the extended attributes are allocated to forms, the generic page will not be displayed.

## To Create an Extended Attribute:

You can create an extended attribute in a profile.

1. Right-click a metaclass, stereotype, or criterion in the Profile category and select **New > Extended Attribute** .
2. Specify the appropriate properties.




3. Click Apply to save your changes.

## Extended Attribute Properties

The following properties are available for extended attributes:

| Property | Description |
|---|---|
| Name | Specifies the internal name of the attribute, which can be used for scripting. |
| Label | Specifies the display name of the attribute, which will appear in the PowerDesigner interface. |
| Comment | Provides additional information about the extended attribute. |
| Data type | Specifies the form of the data to be held by the extended attribute, such as String, Font, Boolean, Object, or others.<br>You can create your own data types (see *Creating an extended attribute type* on page 135), or you can link one object to another by way of an extended attribute, by selecting the [Object] type (see *Linking objects through extended attributes* on page 135). |

| Property | Description |
|---|---|
| Computed | Specifies that the extended attribute is calculated from other values using Get and/or Set VBScript methods. If an attribute is not computed, the value is stored in the object.<br>You can select from the following access types:<br>• Read/Write (Get+Set methods) - read and write access to the extended attribute value is defined by VBScript Get and Set methods.<br>• Read only (Get method) – read-only access is defined by a VBScript Get method.<br>This checkbox enables the display of Get Method Script, Set Method Script, and Global Script tabs, on which you must define the relevant scripts.<br>In the following example script, the FileGroup computed extended attribute gets and sets its value from the physical options of the table object:<br>`Function %Get%(obj)`<br>`%Get% = obj.GetPhysicalOptionValue("on/<filegroup>")`<br>`End Function`<br><br>`Sub %Set%(obj, value)`<br>`obj.SetPhysicalOptionValue "on/<filegroup>", value`<br>`End Sub` |
| Default value | [if not "computed"] Specifies a default value for the extended attribute type. You can specify the value in any of the following ways:<br>• Type the value directly in the list.<br>• [predefined data types] Click the Ellipsis button to obtain a range of possible default values. For example, if the data type is set to Color, the Ellipsis button opens a palette window.<br>• [user-defined data types] Select a value from the list. |
| Template | [if not "computed"] Specifies that the extended attribute is treated as a GTL template and its code is replaced by model values during generation. For example, the string %Code%, will be replaced by the value of the code attribute of the relevant object.<br>If this checkbox is cleared, the extended attribute is treated literally during generation. For example, the string %Code%, will be generated as %Code%. |
| List of values | Specifies a list of possible values for the extended attribute. You can enter static values in the list (separated by a semi-colon or by a carriage return) or generate them using a GTL template.<br>You can use the tools to the right of the list to create a GTL template or to select an existing template in the resource file. The template is evaluated each time the list is called.<br>For example, the following GTL template uses the foreach_item macro to iterate on the Storages collection (when the extended attribute is an object, the list of values must contain the OID of the object, then a tab, and then the name that will be displayed in the list, and ends with a carriage return):<br>`.foreach_item (Model.Storages)`<br>`%ObjectID%\t %Name%`<br>`.next (\n)`<br>The following template returns all the storages in the model:<br>`.collection (Model.Storages)`<br>If the extended attribute is based on an extended attribute type, the List of values box is unavailable because the values of the extended attribute type will be used. |
| Complete | Specifies that all possible values for the extended attribute are defined in the list of values, from which the user must choose. |
| Edit method | [if not "complete"] Specifies a method to override the default action associated with the tool or the Ellipsis button to the right of the extended attribute in the object property sheet. See the Table metaclass in the Profile category of the Oracle Version 10g DBMS resource file for an example of a custom edit method. |

| Property | Description |
|---|---|
| Object type | [for [Object] data types only] Specifies the type of the object that the extended attribute will be (for example, User, Table, Class). |
| Object stereotype | [for [Object] data types only] Specifies the stereotype that objects of this type must bear to be available in the extended attribute list. |
| Inverse collection name | [for [Object] data types only, if not "computed"] Specifies the name under which the links to the object will be listed on the Dependencies tab of the target object.<br>An extended collection with the same name as the extended attribute, which handles these links, is automatically created for all non-computed extended attributes of the Object type, and is deleted when you delete the extended attribute, change its type, or select the Computed checkbox. |
| Physical option | [for [Physical Option] data types only] Specifies the physical option with which the attribute is associated. Click the ellipsis to the right of this field to select a physical option. For more information, see *Adding DBMS physical options to your forms* on page 144 |

## Linking Objects Through Extended Attributes

When you specify the [Object] data type, you enable the display of the Object type, Object stereotype, and Inverse collection name fields.

The Object type field specifies the kind of object you want to link to, and the stereotype field allows you to filter the objects that are available for selection.

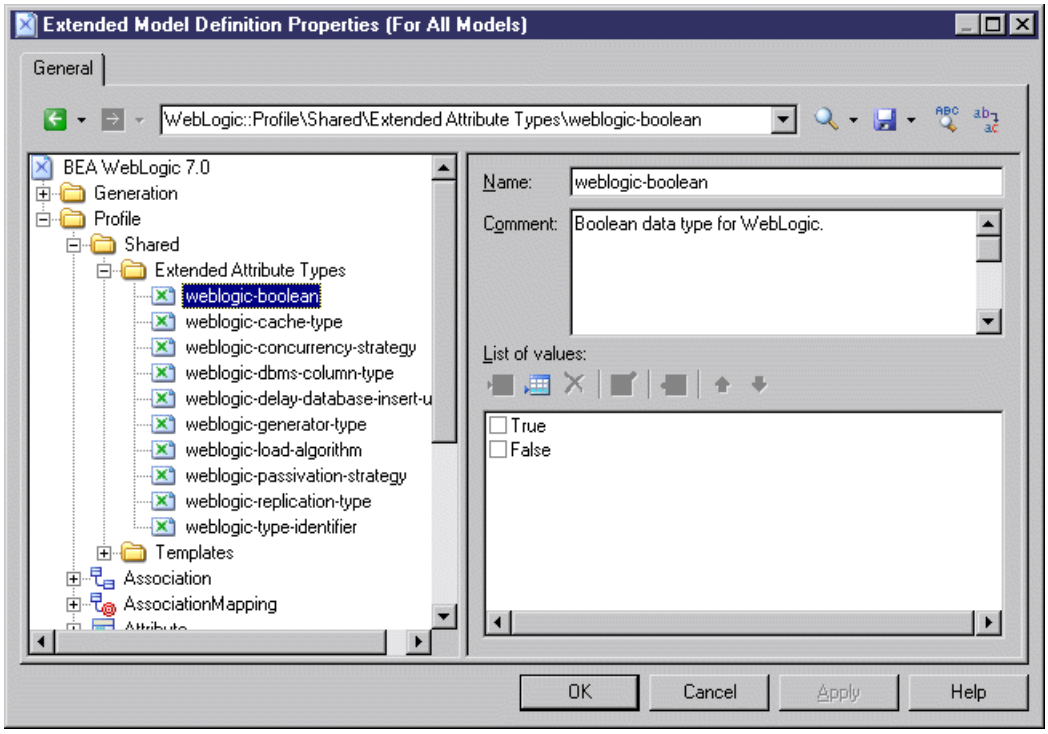
For example, under the Table metaclass, I create an extended attribute called Owner, select [Object] in the Data type field, and User in the Object type field. I name the inverse collection "Tables owned". I can set the Owner attribute in the property sheet of a table, and the table will be listed on the Dependencies tab of the user property sheet, under the name of "Tables owned".

## Creating an Extended Attribute Type

You can create an extended attribute type in the Shared folder in order to define the data type and authorized values of extended attributes. Creating extended attribute types allows you to reuse the same list of values for several extended attributes without having to write code. These types are available in the extended attribute Data Type list.

You can also define a list of values for a given extended attribute from its property page using the Data Type list. For more information, see *Extended attribute properties* on page 133.

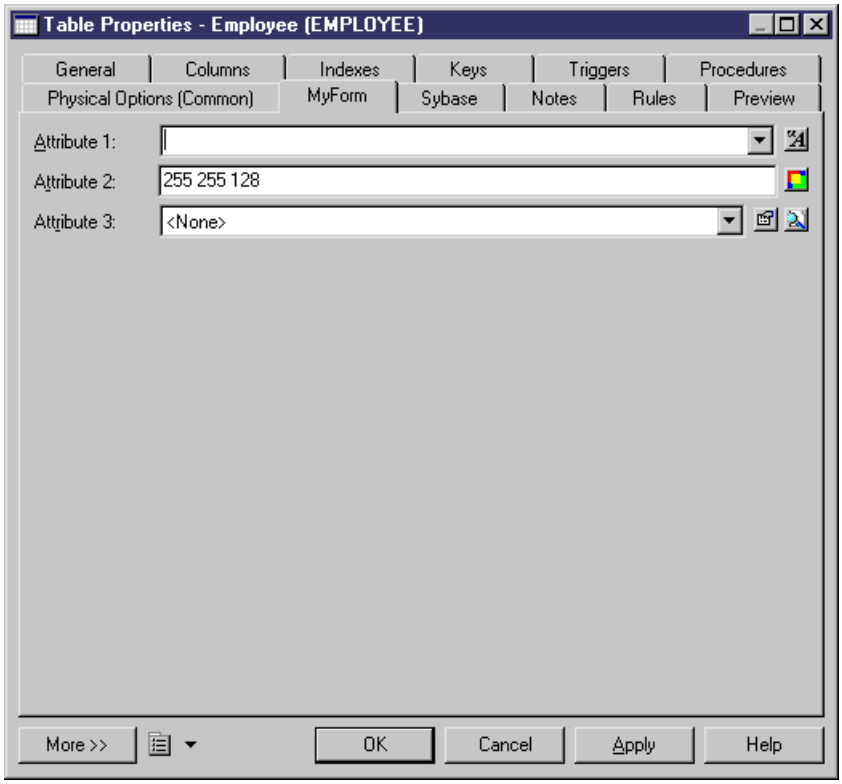
1. Right-click the Profile\Shared category and select **New** > **Extended** Attribute Type in the contextual menu.
2. Enter the appropriate properties, including a list of values and a default value.




3. Click Apply to save your changes.

Once created the new type is available to all other extended attributes with a name as follows: <Attribute Name> type. You can click the Properties tool to the right of the Data type field to edit the type.

Data types display as tools in custom forms helping you to specify a default value for the extended attribute type, as in the following example:



## Extended Collections and Compositions (Profile)

An extended collection allows you to associate multiple instances of one metaclass with an instance of another.

For example, to attach documents containing use case specifications to the different packages of a model you can create an extended collection in the Package metaclass and define FileObject as the target metaclass. You could create an extended collection on the OOM process metaclass to show the components used as resources to the process, in order to have a more accurate vision of the physical implementation of the process.

The association between the parent and child objects is relatively weak, so that:

- If you copy and paste an object with extended collections, the related objects are not copied.
- If you move an object with extended collections, the link with the related objects is preserved (using shortcuts if required).

An extended composition allows you to associate multiple instances of the extended sub-object metaclass with a metaclass. The association is stronger than that created by an extended collection – sub-objects can only be created within the parent object and are moved, copied, and/or deleted along with their parent.

When you create an extended collection or extended composition in a metaclass, a new tab with the name of the collection or composition is added to the metaclass property sheet.

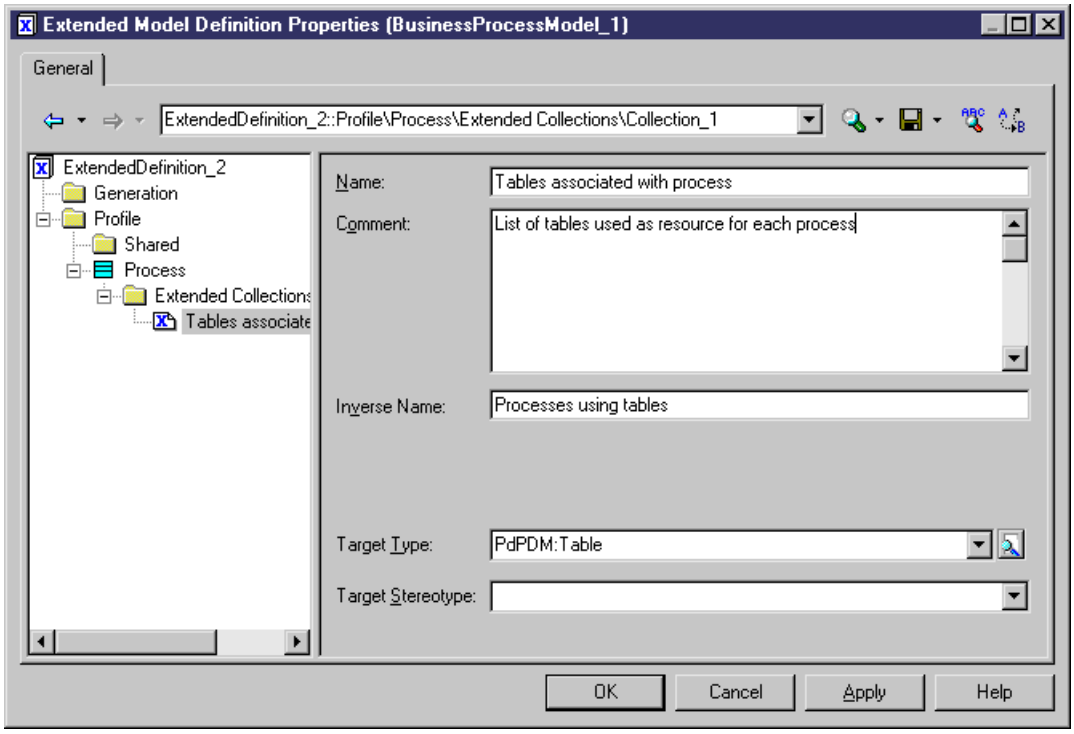
**Note:** If you create an extended collection or composition on a stereotype or criterion, the corresponding tab is displayed only if the metaclass instance bears the stereotype or meets the criterion.

For extended collections, the property sheets of the objects contained within the collection list the parent object on their Dependencies tab.

### To Create an Extended Collection:

You can create an extended collection in a profile.

1. Right-click a metaclass, stereotype, or criterion and select **New > Extended Collection** .
2. Enter a name in the Name box. This name will be used as the name of the related tab in the object property sheet.
3. [optional] Enter a comment and an inverse name.
4. Select a metaclass in the Target Type list to specify the kind of object that will be contained in the collection.
5. [optional] Select or enter a stereotype to further refine the instances of the target metaclass that may appear in the collection. Click the Create tool to the right of this field to create a new stereotype.




6. Click Apply or OK to save your changes and return to your model.

   You can view the tab associated with the collection by opening the property sheet of a metaclass instance. The tab contains Add Objects and Create an Object tools, which allow you to populate the collection.

## To Create an Extended Composition:

You can create an extended composition in a profile.

1. Right-click a metaclass, stereotype, or criterion and select **New** > **Extended Composition** .
2. Enter a name in the Name box. This name will be used as the name of the related tab in the object property sheet.
3. [optional] Enter a comment.
4. Select or enter a stereotype to further refine the instances of the target metaclass that may appear in the collection. Click the Create tool to the right of this field to create a new stereotype.
5. Click Apply or OK to save your changes and return to your model.

   You can view the tab associated with the composition by opening the property sheet of a metaclass instance. The tab contains Add and Insert Row, and Add Objects and Create an Object tools, which allow you to populate the composition.

## Extended Collection/composition Properties

The following properties can be set for extended collections:

| Property | Description |
|---|---|
| Name | Specifies the name of the extended collection. |
| Comment | Describes the extended collection. |
| Inverse Name | [extended collection only] Specifies the name to appear in the Dependencies tab of the target metaclass. If you do not enter a value, an inverse name is automatically generated. |

| Property | Description |
|---|---|
| Target Type | Specifies the metaclass whose instances will appear in the collection. For extended collections, the list displays only metaclasses that can be directly instantiated in the current model or package, such as classes or tables, and not sub-objects such as class attributes or table columns. Click the Select a Metaclass tool to the right of this field to choose a metaclass from another type of model.<br>For extended compositions, only the ExtendedSubObject is available. |
| Target Stereotype | [required for extended compositions] Specifies a stereotype to filter the target type. You can select an existing stereotype from the list or enter a new one. |

When you open a model containing extended collections or compositions and associate it with a resource file that does not support them, the collections are still visible in the different property sheets in order to let you delete objects in the collections no longer supported.

## Calculated Collections (Profile)

You define a calculated collection on a metaclass, stereotype, or criterion, when you need to display a list of associated objects with a user-defined semantic.

Calculated collections (unlike extended collections) cannot be modified by the user (see *Extended Collections and Compositions (Profile)* on page 137).

You create calculated collections to:

- Display user-defined dependencies for a selected object, the calculated collection is displayed in the Dependencies tab of the object property sheet. You can double-click items and navigate among user-defined dependencies.
- Fine-tune impact analysis by creating your own calculated collections in order to be able to better evaluate the impact of a change. For example, in a model where columns and domains can diverge, you can create a calculated collection on the domain metaclass that lists all the columns that use the domain and have identical data type.
- Improve your reports. You can drag and drop any book or list item under any other report book and modify its default collection in order to document a specific aspect of the model (see "Modifying the collection of a report item" in the Reports chapter of the *Core Features Guide*).
- Improve GTL generation since you can loop on user-defined calculated collections.

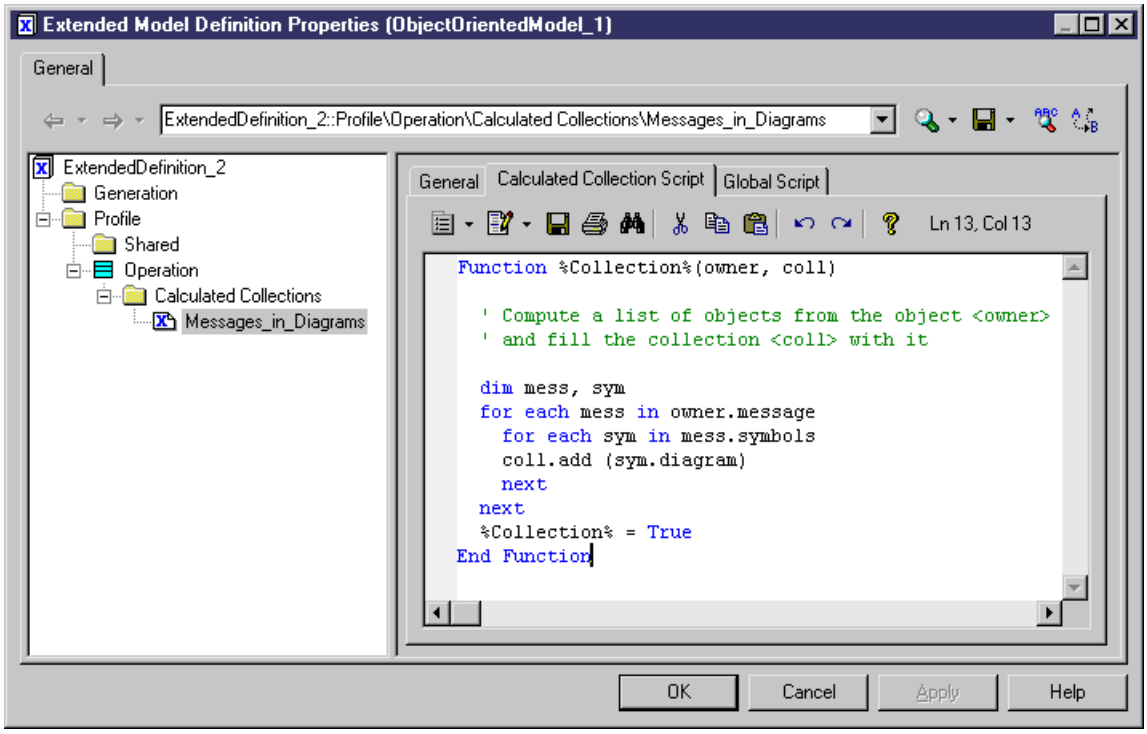
For example, in an OOM, you may need to create a list of sequence diagrams using an operation, and can create a calculated collection on the operation metaclass that retrieves this information.

In a BPM, you could create a calculated collection on the process metaclass that lists the CDM entities created from data associated with the process.

### To Create a Calculated Collection:

You can create a calculated collection in a profile.

1. Right-click a metaclass, stereotype, or criterion and select **New** > **Calculated Collection** .
2. Enter a name in the Name box. This name will be used as the name of the related tab in the object property sheet.
3. [optional] Enter a comment to describe the collection.
4. Select a metaclass in the Target Type list to form the basis of the collection.
5. [optional] Select or enter a stereotype to further refine the instances of the target metaclass that may appear in the collection.
6. Click the Calculated Collection Script tab and enter a script that will calculate which objects will form the collection. If appropriate, you can reuse functions on the Global Script tab.




7. Click Apply to save your changes.

## Calculated Collection Properties

The following properties can be set for extended collections:

| Property | Description |
|---|---|
| Name | Specifies the name of the calculated collection. |
| Comment | Describes the calculated collection. |
| Target Type | Specifies the metaclass whose instances will appear in the collection. The list displays only metaclasses that can be directly instantiated in the current model or package, such as classes or tables, and not sub-objects such as class attributes or table columns.<br>Click the Select a Metaclass tool to the right of this field to choose a metaclass from another type of model. |
| Target Stereotype | Specifies a stereotype to filter the target type. You can select an existing stereotype from the list or enter a new one. |

The *Calculated Collection Script* tab contains the definition of the body of the calculated collection function.

The *Global Script* tab is used for sharing library functions and static attributes in the resource file. You can declare *global variables* on this tab, but you should be aware that they will not be reinitialized each time the collection is calculated, and keep their value until you modify the resource file, or the PowerDesigner session ends. This may cause errors, especially when variables reference objects that can be modified or deleted. Make sure you reinitialize the global variable if you do not want to keep the value from a previous run.

For more information on defining a script and using the Global Script tab, see *Defining the script of a custom check* on page 152 and *Using the global script* on page 155.

# Forms (Profile)

You can use forms to create new property sheet tabs or to replace existing tab, or to create dialog boxes that are launched from menus or by clicking on buttons in your property sheet tabs. Building a new form is fast and easy, using the form tools in the resource editor.

By default, extended attributes are listed alphabetically on the Extended Attributes tab of the object's property sheet. By creating your own form, you can make these attributes more visible and easy to use, by organizing them logically, grouping related ones, and emphasizing those that are most important. Custom forms are used in PDMs to emphasize the most commonly-used physical options on the "Physical Options (Common)" tabs.

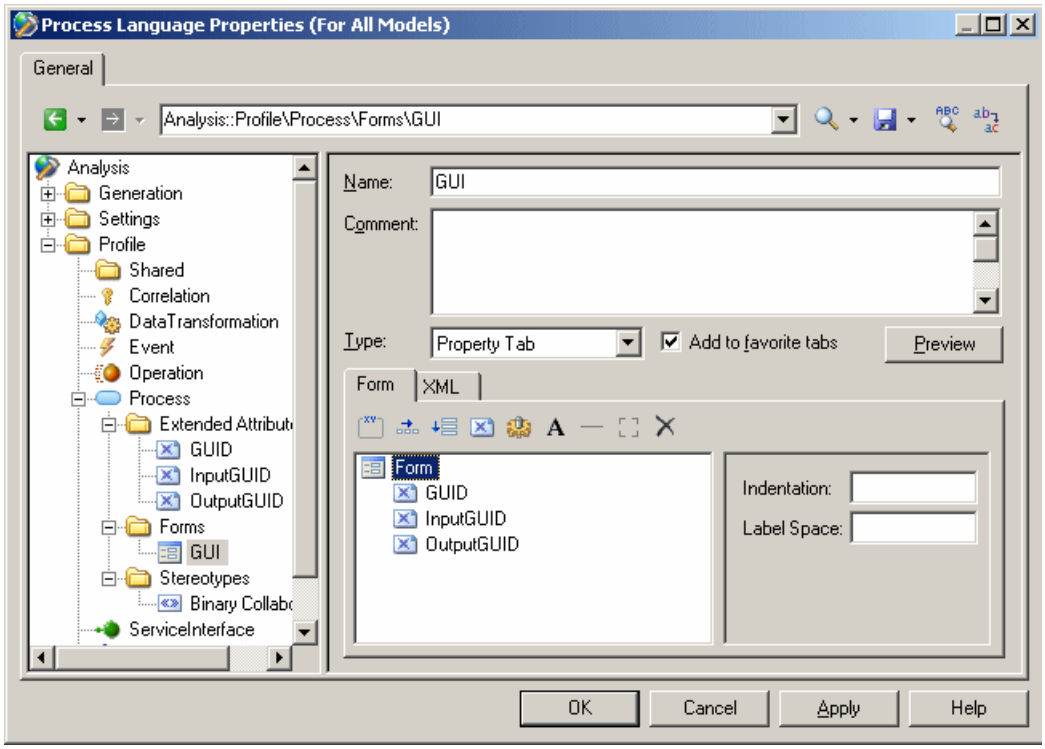
You can create a form on any metaclass that has a property sheet, or on a stereotype or a criterion. For property tabs, if the tab is linked to a stereotype or criterion, it is displayed only when the metaclass instance bears the stereotype or meets the criterion.

## To Create a Form:

You can create a form in a profile.

1. Right-click a metaclass, stereotype or criterion and select **New** > **Form** .

   The form is created.




2. Enter a descriptive name for the form. This name will display in the tab of the property tab or in the title bar of the dialog box. You can also, optionally, enter a description of the form in the Comment field.
3. Select a Type. You can choose from the following:

   - Property Tab – creates a new tab in the property sheet of the metaclass, stereotype or criterion
   - Replace *<standard>* Tab – replaces a standard tab in the property sheet of the metaclass, stereotype or criterion. If your form is empty, it will be filled with the standard controls from the tab that you are replacing.
   - Dialog Box – creates a dialog box that can be launched from a menu or via a form button

4. [optional, for property tabs only] Select the Add to favorite tabs check box to have the tab displayed by default in the property sheet.
5. Insert and arrange extended attributes and other controls using the toolbar on the Form tab at the bottom of the form (see *Adding extended attributes and other controls to your form* on page 142).
6. Click the Preview button to review the layout of your form and, when satisfied, click Apply to save your changes.

## Adding Extended Attributes and Other Controls to Your Form

You insert controls into your form using the tools in the Form tab toolbar.

You can reorder controls in the form control tree by dragging and dropping them. To place a control inside a container control (group box or horizontal or vertical layout), drop it onto the container. For example, if you want the extended attributes GUID, InputGUID, and OutputGUID to be displayed in a GUI group box, you should create a group box, name it GUI and drag and drop all three extended attributes under the GUI group box.

| Tool | Description |
|---|---|
|  | Add Group Box - inserts a group box, intended to contain other controls within a named box. |
|  | Add Horizontal Layout - inserts a horizontal layout, intended to contain other controls placed side by side. |
|  | Add Vertical Layout - inserts a vertical layout, intended to contain other controls placed one under the other. |
|  | Add Attribute – opens a selection box in which you select standard or extended attributes belonging to the metaclass. Select one or more attributes and then click OK to insert them into the form.<br>The type of control associated with the attribute depends on its type: booleans are associated with check boxes, lists with combo boxes, text fields with multi-line edit boxes, and so on.<br>Unless you enter a label, the attribute name is used as its form label, and any comment that you have entered is displayed as its tooltip. |
|  | Add Collection – opens a selection box in which you select standard collections belonging to the metaclass. Select one or more collections and then click OK to insert them into the form.<br>Collections are displayed as standard grids with all the appropriate tools.<br>Unless you enter a label, the collection name is used as its form label and any comment that you have entered is displayed as its tooltip. |
|  | Add Method Push Button - opens a selection box in which you select one or more methods, which will be associated with the form via form buttons. This list is limited to methods defined under the same metaclass in the profile. Select one or more methods and then click OK to insert them into the form.<br>Each method is displayed as a button on the form that, when clicked, invokes the method. Unless you enter a label, the method name is used as the button label. Any comment that you have entered for the method is displayed as its tooltip in the form. |
|  | Add Edit Field [dialog boxes only] inserts an edit field. |
|  | Add Multi-Line Edit Field [dialog boxes only] - inserts a multi-line edit field below the selected item in the tree. |
|  | Add Combo Box [dialog boxes only] - inserts a combo box. |
|  | Add List Box [dialog boxes only] - inserts a list box. |
|  | Add Check Box [dialog boxes only] - inserts a check box. |
|  | Add Text - inserts a text control. |
|  | Add Separator Line – inserts a separator line. The line is vertical when its parent control is a vertical layout. |

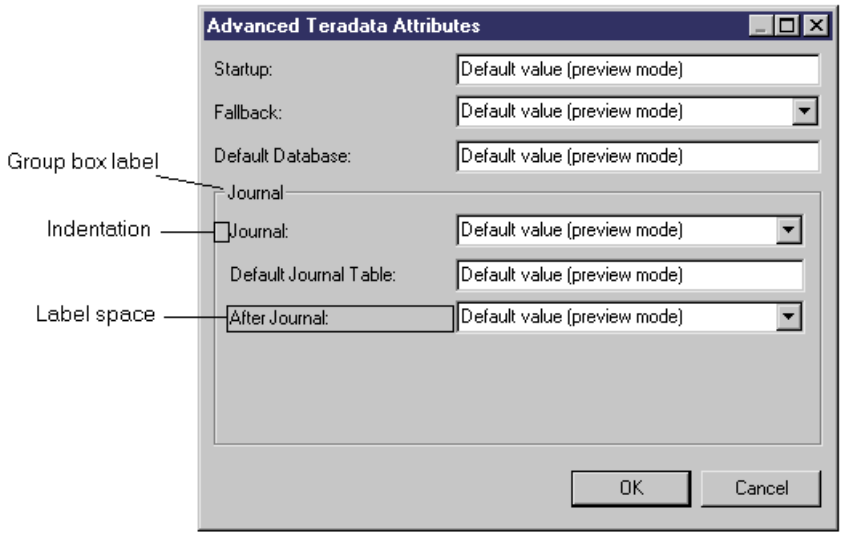
| Tool | Description |
|---|---|
|  | Add Spacer – inserts an area of blank space. |
|  | Delete – deletes the currently selected control. |

### Form Control Properties

The following properties can be set for form controls:

| Property | Definition |
|---|---|
| Name | Internal name of the control. This name must be unique within the form. The name can be used in scripts to get and set dialog box control values (see *Example: Creating a dialog box launched from a menu* on page 149). |
| Label | Label identifying the control on the form. If this field is left blank, the name of the control is used. If you enter a space, then no label is displayed.<br>The label accepts line breaks in the form of \n.<br>You can create keyboard shortcuts to navigate among controls by prefixing the letter that will serve as the keyboard shortcut with an & character. If you do not specify a shortcut key, PowerDesigner will choose one by default.<br>To use the & character in a label, you must escape it with a second & (for example: "&Johnson && Son" will display as "Johnson & Son". |
| Indentation | [container controls only] Specifies the space in pixels between the left margin of the container (form, group box, or horizontal or vertical layout) and the beginning of the labels of its child controls. |
| Label space | [container controls only] Specifies the space in pixels reserved for displaying the labels of child controls between the indentation of the container control (form, group box, or horizontal or vertical layout) and the control fields.<br>If a child control label is larger than this value, the label space property is ignored; to display this label, you need to type a number of pixels greater than 50. |
| Show control as label | [group boxes only] Use the first control contained within the group box as its label. |
| Show Hidden Attribute | [extended attributes only] Displays controls that are not valid for a particular form (because they do not bear the relevant stereotype, or do not meet the criteria) as greyed. If this option is not set, irrelevant options are hidden. |
| Value | [dialog box entry fields only] Specifies a default value for the control. Note that default values for extended attributes must be specified in the attribute's properties (see *Extended attribute properties* on page 133). |
| List of Values | [combo and list boxes only] Specifies a list of possible values for the control. Note that lists of values for extended attributes must be specified in the attribute's properties (see *Extended attribute properties* on page 133). |
| Exclusive | [combo box only] Specifies that only the values defined in the List of values can be entered in the combo box. |
| Minimum Size (chars) | Specifies the minimum width (in characters) to which the control may be reduced when the window is resized. |
| Minimum Line Number | Specifies the minimum number of lines to which a multiline control may be reduced when the window is resized. |
| Horizontal Resize | Specifies that the control may be resized horizontally when the window is resized. |
| Vertical resize | Specifies that the multiline control may be resized vertically when the window is resized. |

| Property | Definition |
|---|---|
| Read-Only | [dialog box entry fields only] Specifies that the control is read-only, and will be greyed in the form. |
| Left Text | [booleans only] Places the label text to the left of the checkbox. |
| Display | [booleans only] Specifies the form in which the options are displayed. You can choose between:<br>• Check box<br>• Vertical radio buttons<br>• Horizontal radio buttons |
| Width | [spacer only] Specifies the width, in pixels, of the spacer. |
| Height | [spacer only] Specifies the height, in pixels, of the spacer. |




## Adding DBMS Physical Options to Your Forms

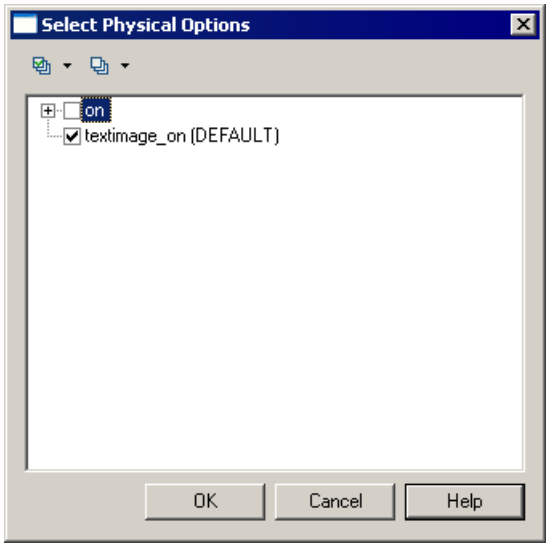
Many of the DBMSs supported in the PowerDesigner Physical data Model use "physical options" as part of the definition of their objects. PowerDesigner displays all of the available options on the Physical Options tab of the relevant object's property sheet, which is powered by the Options entry in the Script/Objects/<object> category in the DBMS resource file.

For more information about physical options, see *Physical Options* on page 101.

The most commonly-used physical options are displayed on a pre-configured custom form, which is called Physical Options (Common), and is located in the object's profile. You can edit this form by adding or removing controls, or create your own form to manage your preferred physical options.

For a physical option to be displayed in a profile form, it must be promoted to the status of an extended attribute (with a "physical option" type), and is then added to the form in the same way as other attributes.

1. Right-click the metaclass and select **New > Extended Attribute from Physical Options** to open the Select Physical Options dialog:




Note that this dialog will be empty if no physical options are defined in the Options entry in the Script/Objects/<object> category.

2. Select the physical option required and click OK to create an extended attribute associated with it.
3. Specify any other appropriate properties.
4. Select the form in which you want to insert the physical option and click the Extended Attribute tool to insert it as a control (see *Adding extended attributes and other controls to your form* on page 142).

**Note:** You can access the Select Physical Options dialog to change the associated physical option by clicking the ellipsis to the right of the Physical Options field in the Extended Attribute property sheet.

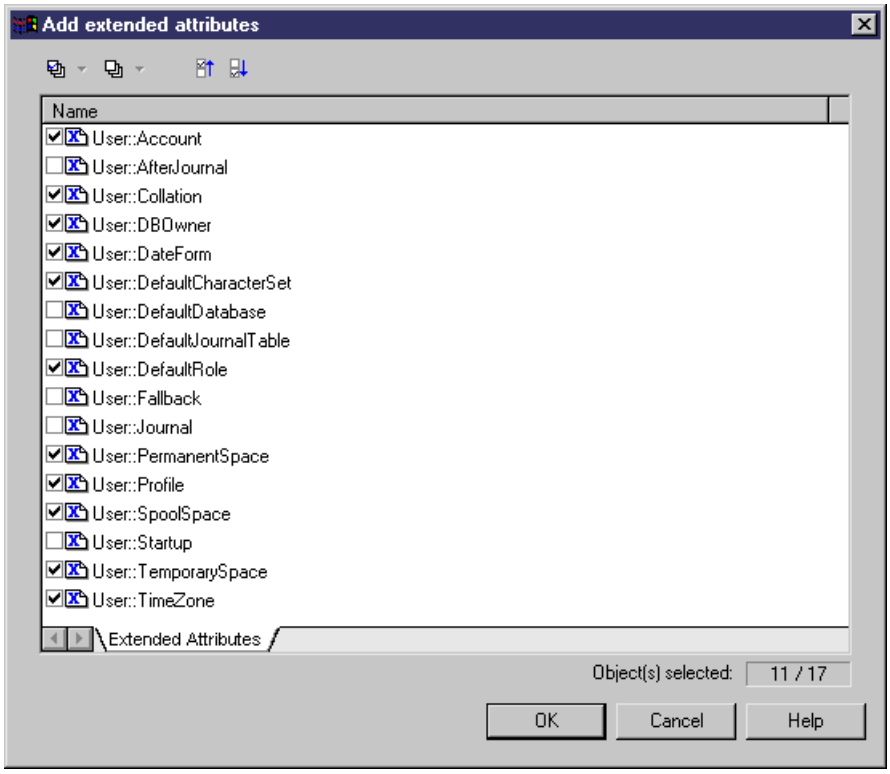
## Example: Creating a Property Sheet Tab

This example guides you through creating a form for presenting extended attributes in a Teradata v2r5 PDM. Note that this is just an example, and that Teradata extended attributes are already organized in forms.

To follow along with this example, create a new Teradata v2r5 PDM and then select **Database > Edit Current DBMS** to open the Resource Editor. Expand the **Profile > User > Extended Attributes** folder to review the extended attributes defined for the user object.

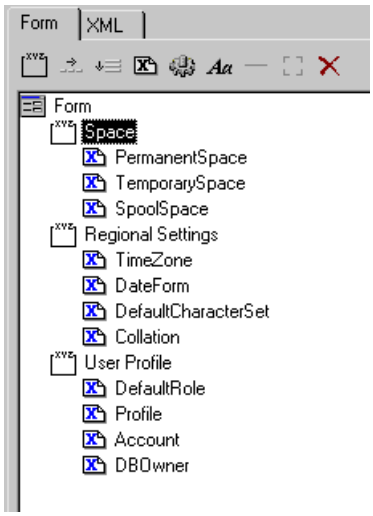
We will create a new property tab to present the most commonly-used of these attributes.

1. Right-click the User metaclass and select **New > Form** .
2. In the new form property sheet, enter "Teradata" in the Name box, select Property Tab from the Type list, and then clear the Add to favorite tabs check box.
3. Click the Extended Attribute tool in the Form tab toolbar to open the Add Extended Attributes dialog box:

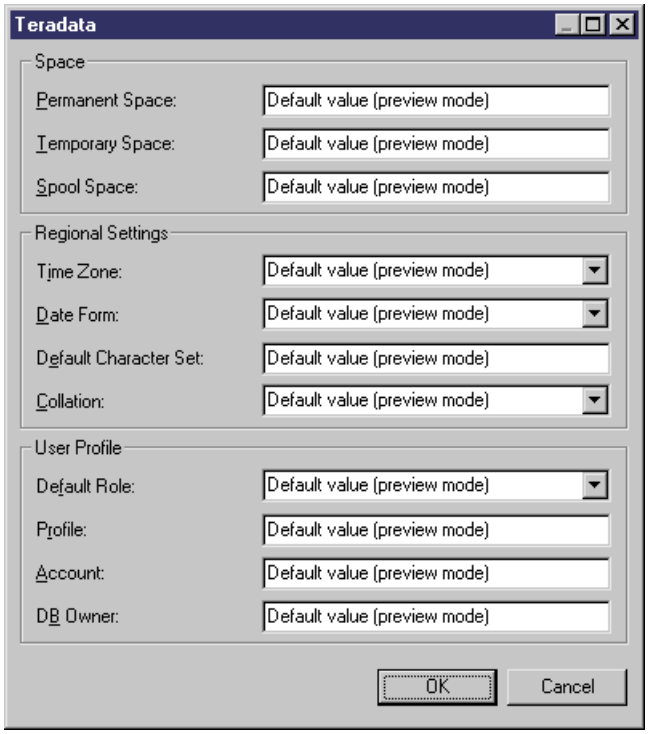



Select the attributes as in the above screenshot and then click OK to add them to the form.

4. Click the Group Box tool to create a group box control, and enter the name "Space" in its Name field.
5. Create two additional group boxes, name them "Regional Settings" and "User Profile", and then drag and drop the attributes in the list in order to organize them into the group boxes as follows:.




6. Click the Preview button to view the new tab. The different input boxes are not aligned, so click Cancel to return to the Resource Editor.
7. Click the Space group box control in the list and enter 140 in its Label Space field. Then repeat this process for the two other group boxes. Now, when you click Preview, all the controls are aligned together:




Continue with the next example to add a dialog box that will be launched by clicking on a button on the form.

## Example: Creating a Dialog Box to Launch from a Property Tab

In this example, we will continue to work with the Teradata property tab, by creating a dialog box for editing more specific attributes.

This dialog will be launched when you click a button on the property tab (see *Example: Creating a property sheet tab* on page 145).

You call a dialog box by invoking a method (see *Methods (Profile)* on page 159).

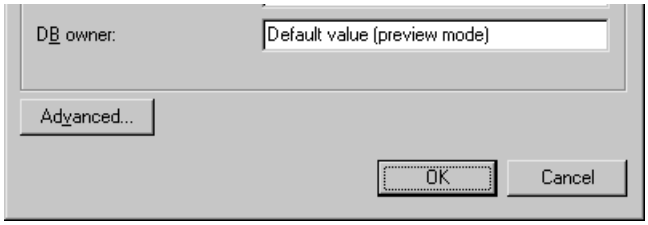
### To Create the Method to Call the Dialog:

You need first to create the method, and then to add it to the property tab to create a button for calling the dialog.

1. Right-click the User metaclass and select **New** > **Method** .
2. Name the new method ShowAdvancedExtendedAttributes, and then click the Method Script tab and enter the following script:

```
Sub %Method%(obj)
 ' Show custom dialog for advanced extended attributes
 Dim dlg
 Set dlg = obj.CreateCustomDialog("%CurrentTargetCode%.Advanced Teradata
Attributes")
 If not dlg is Nothing Then
  dlg.ShowDialog()
 End If
End Sub
```

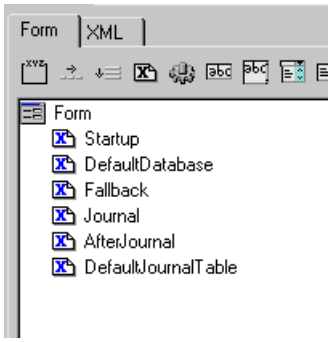
3. Select the Teradata property tab in the ProfileUserForms folder, click the Method Push Button tool in the Form tab toolbar, select the new method, and then click OK to add it to the form.
4. Enter Advanced... in the Label field, and then click Preview to see the new button at the bottom of the property tab:



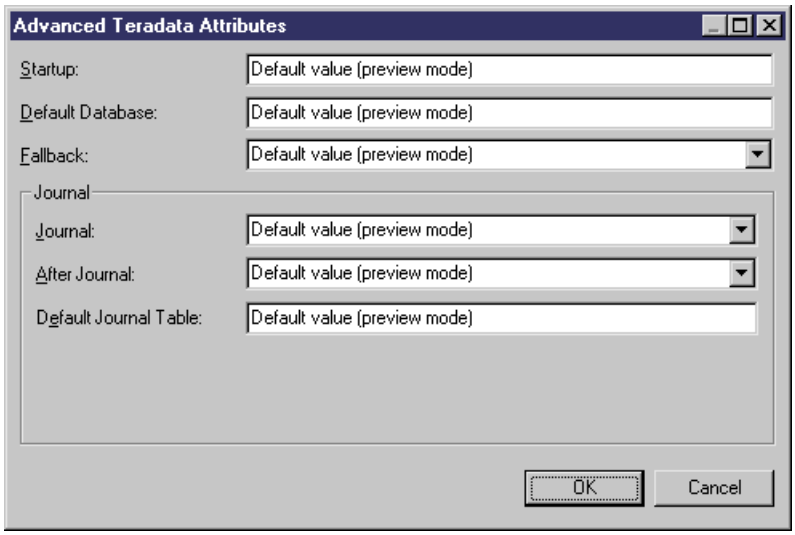

### To Create the Advanced Attributes Dialog Box:

Now we will continue on to create the dialog box.

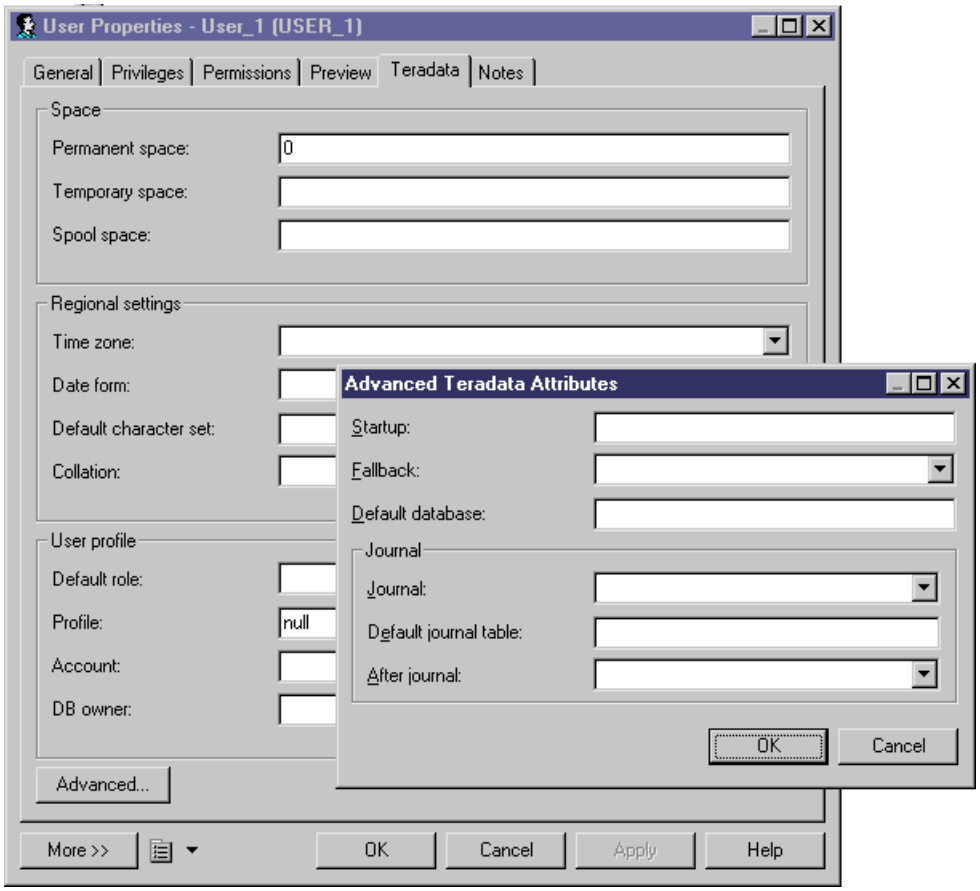
1. Right-click the Forms category in the User metaclass and select New to create a new form.
2. Enter Advanced Teradata Attributes in the Name box and select Dialog Box in the type list. Note that additional tools appear in the Form tab toolbar.
3. Click the Extended Attribute tool in the palette, select the following attributes, and then click OK to add them to the form:




4. Click the Group Box tool to create a group box control and name it Journal. Then drag the Journal, AfterJournal, and DefaultJournalTable extended attributes into the group box.
5. Click the Form entry in the controls tree and enter 140 in the Label Space box to ensure that all the controls are aligned. Then click Preview.




6. You cannot test launching the dialog box in preview mode, so click OK to save your changes and close the Resource Editor. Then, create a user, open its property sheet, select the Teradata tab, and click the Advanced button to launch the dialog box:




## Example: Creating a Dialog Box Launched from a Menu

You can create a dialog box when you need to enter parameters during OLE automation through VB scripts.

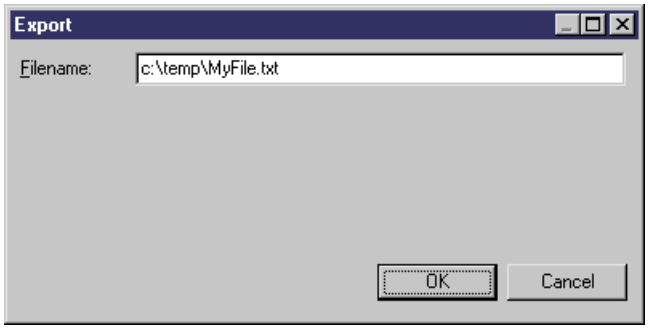
In this example, we will create a new dialog box that will be launched from a new "Export command in the contextual menu of extended objects, and which allows you to enter a path to where the extended object should be exported.

1. Right-click the ExtendedObject metaclass in the resource editor and select **New** > **Form** .
2. Type Export in the Name box, and select Dialog Box in the type list.
3. Click the Edit Field tool to add an edit field control, and name it "Filename".
4. Right-click the ExtendedObject metaclass and select **New** > **Method** . This creates a new method, you can call it Export.
5. Name the method "Export", click the Method Script tab and enter the following code:

```
Sub %Method%(obj)
 ' Exports an object in a file

 ' Create a dialog to input the export file name
Dim dlg
Set dlg = obj.CreateCustomDialog
       ("%CurrentTargetCode%.Export")
If not dlg is Nothing Then

 ' Initialize filename control value
 dlg.SetValue "Filename", "c:\temp\MyFile.txt"

 ' Show dialog
 If dlg.ShowDialog() Then

  ' Retrieve customer value for filename control
  Dim filename
```

```
   filename = dlg.GetValue("Filename")

   ' Process the export algorithm...
   Output "Exporting object " + obj.Name + " to file " + filename

  End If

  ' Free dialog object
  dlg.Delete
  Set dlg = Nothing

 End If

End Sub
```

6. Right-click the ExtendedObject metaclass and select **New** > **Menu** to create a new menu entry in its contextual menu (see *Menus (Profile)* on page 161).
7. Enter the name "Export" and then click the Add Commands from Methods and Transformations tool and select the Export method.
8. Click OK to save your changes and close the Resource Editor.
9. Right-click an extended object and select the Export command in its contextual menu to launch the Export dialog box.



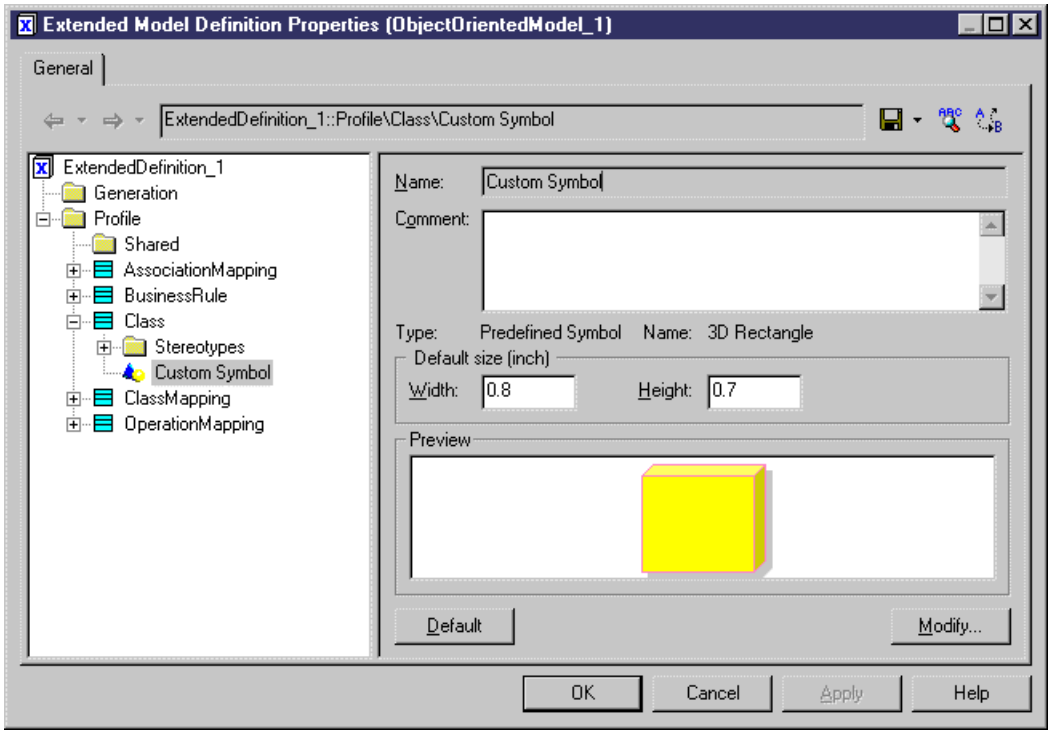

## Custom Symbols (Profile)

A custom symbol allows you to modify the appearance of instances of the metaclass, stereotype, or criterion.

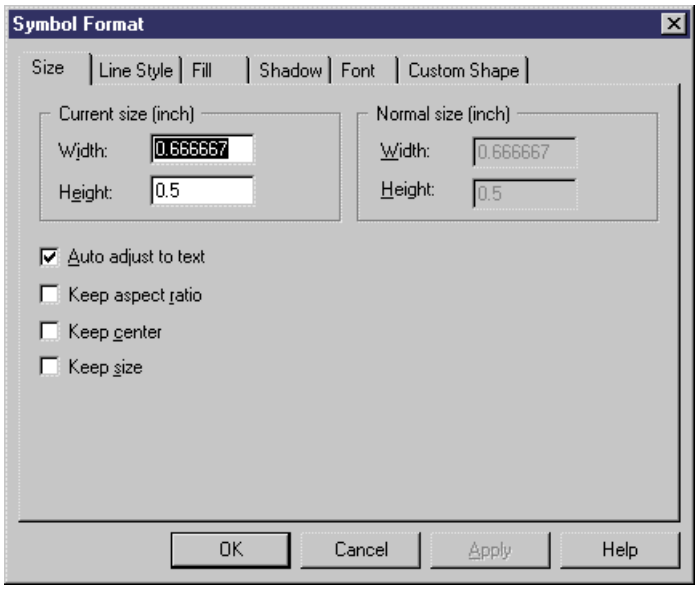
When you customize the line style of a link symbol, such as a PDM reference for example, the parameters you select in the Style list and in the Arrow groupbox in the Line Style tab replace the one you may have selected in the Display Preferences dialog box. This can provoke confusion in the model coherence. To avoid that confusion and preserve the method definition of your model, you should use the Notation attribute in the Style list and or in the Arrow groupbox. This attribute is only available in the Profile.

1. Right-click a metaclass, stereotype, or criterion in the Profile category and select **New** > **Custom Symbol** .

   A new custom symbol is created under the selected category.




2. Specify a default width and height in the Default Size groupbox.
3. Click the Modify button to open the Symbol Format dialog box, and set the required properties on the various tabs:




For more information on the Symbol Format dialog box, see "Symbol format properties" in the Diagrams and Symbols chapter of the *Core Features Guide*.

4. Click OK to return to the resource editor, where you can view your changes in the Preview field.
5. Click Apply to save your changes.

## Custom Checks (Profile)

Custom checks are model checks, written in VBScript, which enable you to verify that your model objects are well-defined. Custom checks are listed with standard model checks in the Check Model Parameters dialog box.

For more information about using VBScript, see *Scripting PowerDesigner* on page 235.

## Custom Check Properties

When you create custom checks you have to define the following general properties:

| Parameter | Description |
|---|---|
| Name | Name of the custom check. This name is displayed under the selected object category in the Check Model Parameters dialog box. This name is also used (concatenated) in the check function name to uniquely identify it |
| Comment | Additional information about the custom check |
| Help Message | Text displayed in the message box that is displayed when the user selects Help in the custom check context menu in the Check Model Parameters dialog box |
| Output message | Text displayed in the Output window during check execution |
| Default severity | Allows you to define if the custom check is an error (major problem that stops generation) or a warning (minor problem or just recommendation) |
| Execute the check by default | Allows you to make sure that this custom check is selected by default in the Check Model Parameters dialog box |
| Enable automatic correction | Allows you to authorize automatic correction for the custom check |
| Execute the automatic correction by default | Allows you to make sure that automatic correction for this custom check is executed by default |
| Check Script | This tab contains the custom check script. See *Defining the script of a custom check* on page 152. |
| Autofix Script | This tab contains the autofix script. See *Defining the script of an autofix* on page 154 |
| Global Script | This tab is used for sharing library functions and static attributes in the resource file. See *Using the global script* on page 155. |

## Defining the Script of a Custom Check

This section also applies for defining the script of a custom method, a calculated collection, an event handler, or a transformation.

You type the script of a custom check in the Check Script tab of the custom check properties. By default, the Check Script tab displays the following script items:

- %Check% is the function name, it is passed on parameter obj. It is displayed as a variable, which is a concatenation of the name of the resource file, the name of the current metaclass, the name of the stereotype or criterion, and the name of the check itself defined in the General tab. If any of these names contains an empty space, it is replaced by an underscore
- A comment explaining the expected script behavior
- The return value line that indicates if the check succeeded (true) or not (false)

In Sybase AS IQ, you need to create additional checks on indexes in order to verify their columns. The custom check you are going to create verifies if indexes of type HG, HNG, CMP, or LF are linked with columns which data type VARCHAR length is higher than 255.

1. Right-click a metaclass, stereotype or a criterion under Profile, and select **New > Custom Check** .
2. Click the Check Script tab in the custom check properties to display the script editor.

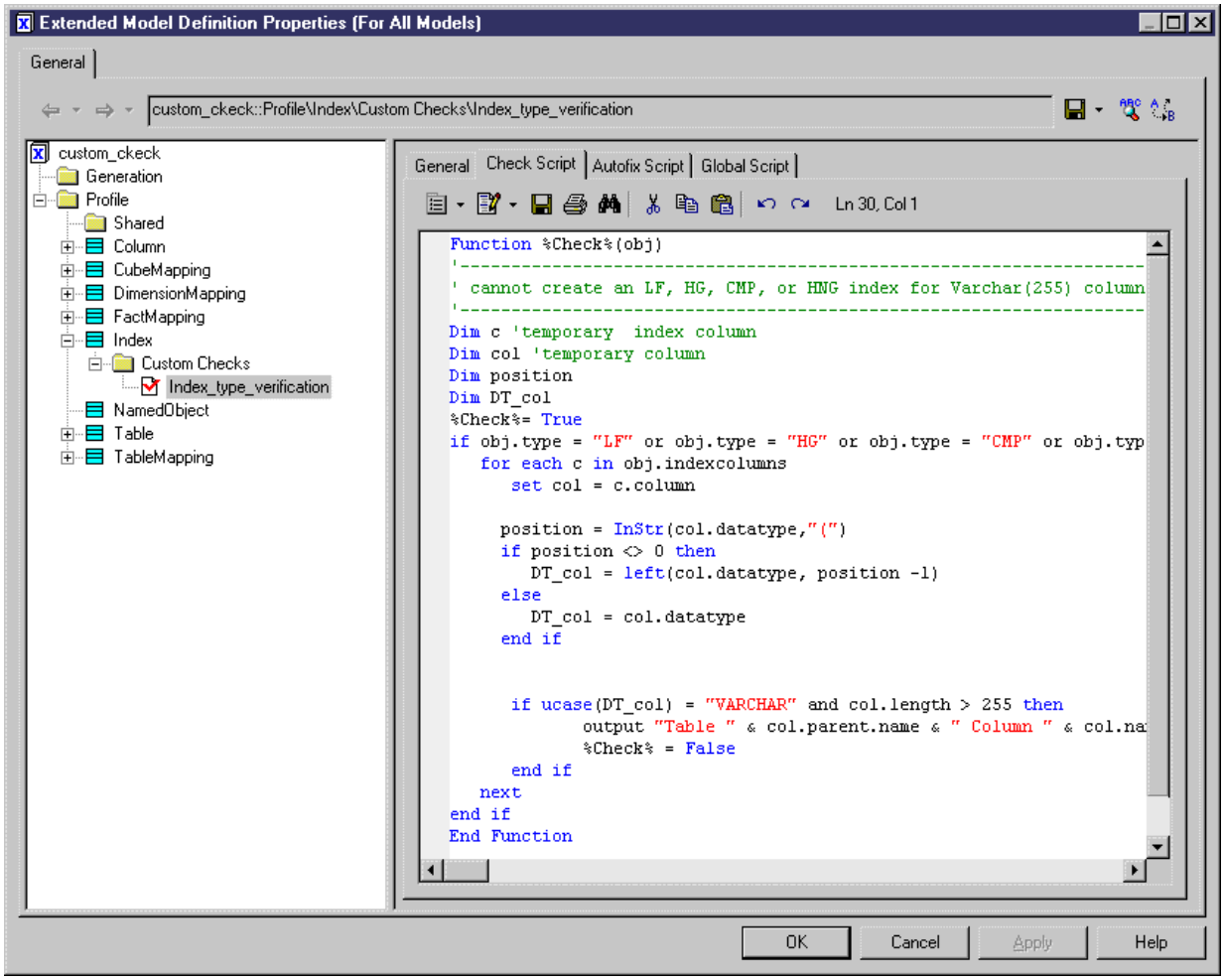
   By default, the function is declared at the beginning of the script. You should not modify this line.

3. Type a comment after the function declaration in order to document the custom check, and then declare the different variables used in the script.

```
Dim c 'temporary  index column
Dim col 'temporary column
Dim position
Dim DT_col
```

4. Enter the function body.

```
%Check%= True

if obj.type = "LF" or obj.type = "HG" or obj.type = "CMP" or obj.type ="HNG"
then
 for each c in obj.indexcolumns
  set col = c.column

   position = InStr(col.datatype,"(")
   if position <> 0 then
    DT_col = left(col.datatype, position -1)
   else
    DT_col = col.datatype
   end if
if ucase(DT_col) = "VARCHAR" and col.length > 255 then
     output "Table " & col.parent.name & " Column " & col.name & " : Data
type is not compatible with Index " & obj.name & " type " & obj.type
     %Check% = False
  end if
```




5. Click Apply to save your changes.

## Defining the Script of an Autofix

If the custom check you have defined supports an automatic correction, you can type the body of this function in the Autofix Script tab of the custom check properties.

The autofix is visible in the Check Model Parameters dialog box, it is selected by default if you select the Execute the Automatic Correction by Default check box in the General tab of the custom check properties.

By default, the Autofix Script tab displays the following script items:

- %Fix% is the function name, it is passed on parameter obj. It is displayed as a variable, which is a concatenation of the name of the resource file, the name of the current metaclass, the name of the stereotype or criterion, and the name of the fix. If any of these names contains an empty space, it is replaced by an underscore
- The variable *outmsg* is a parameter of the fix function. You need to specify the fix message that will appear when the fix script will be executed
- The return value line that indicates if the fix succeeds or not

We will use the same example as in section Defining the script of a custom check, to define an autofix script that removes the columns with incorrect data type from index.

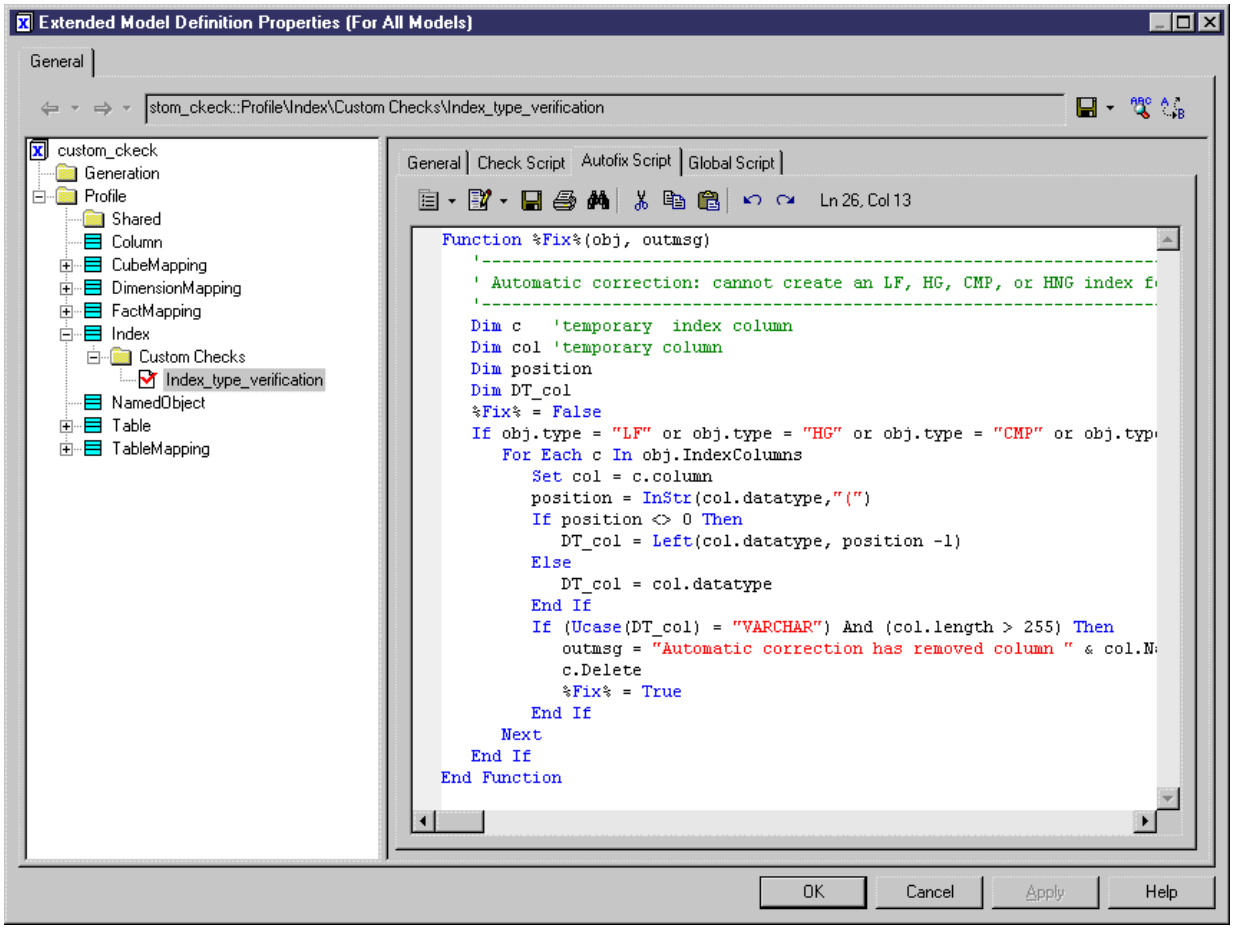
1. Click the Autofix Script tab in the custom check properties.

   By default, the function is declared at the beginning of the script. You should not modify this line.
2. Type a comment after the function declaration in order to document the custom check, and then declare the different variables used in the script:

```
Dim c 'temporary  index column
Dim col 'temporary column
Dim position
Dim DT_col
```

3. Enter the function body:

```
%Fix% = False
 If obj.type = "LF" or obj.type = "HG" or obj.type = "CMP" or obj.type
="HNG" Then
  For Each c In obj.IndexColumns
   Set col = c.column
   position = InStr(col.datatype,"(")
   If position <> 0 Then
    DT_col = Left(col.datatype, position -1)
   Else
    DT_col = col.datatype
   End If
   If (Ucase(DT_col) = "VARCHAR") And (col.length > 255) Then
    outmsg = "Automatic correction has removed column " & col.Name & " from
index."
    c.Delete
    %Fix% = True
   End If
  Next
 End If
```




4. Click Apply to save your changes.

## Using the Global Script

This section also applies for defining the script of a custom method, a calculated collection, an event handler, or a transformation.

The Global Script tab is used to store functions and static attributes that may be reused among different functions. This tab displays a library of available sub-functions.

### Example

In the Sybase AS IQ example, you could have a function called DataTypeBase that retrieves the data type of an item in order to further analyze it.

This function is defined as follows:

```
Function DataTypeBase(datatype)
 Dim position
 position = InStr(datatype, "(")
 If position <> 0 Then
  DataTypeBase = Ucase(Left(datatype, position -1))
 Else
  DataTypeBase = Ucase(datatype)
 End If
End Function
```

In this case, this function only needs to be referenced in the check and autofix scripts:

```
Function %Check%(obj)
Dim c 'temporary  index column
```

```
 Dim col 'temporary column
 Dim position
 %Check%= True
 If obj.type = "LF" or obj.type = "HG" or obj.type = "CMP" or obj.type ="HNG"
then
  For Each c In obj.IndexColumns
   Set col = c.column
   If (DataTypeBase(col.datatype) = "VARCHAR") And (col.length > 255) Then
    Output "Table " & col.parent.name & " Column " & col.name & " : Data type
is not compatible with Index " & obj.name & " type " & obj.type
    %Check% = False
   End If
  Next
 End If
End Function
```

### Global Variables

You can also declare *global variables* in the Global Script. These variables are reinitialized each time you run the custom check.

## Running Custom Checks and Troubleshooting Scripts

All custom checks defined in any resource files attached to the model are merged and all the functions for all the custom checks are appended to build one single script. The Check Model Parameters dialog box displays all custom checks defined on metaclasses, stereotypes and criteria under the corresponding categories.

If there are errors in your custom check scripts, the user will be prompted with the following options:

| Button | Action |
|---|---|
| Ignore | Allows you to skip the problematic script and resume check |
| Ignore All | Allows you to skip all problematic scripts and resume process with standard checks |
| Abort | Stops check model |
| Debug | Stops check model, opens the resource editor and indicate on which line the problem is. You can correct error and restart check model |

# Event Handlers (Profile)

An event handler can automatically launch a VBScript when an event occurs on an object. You can associate an event handler with a metaclass or a stereotype; criteria do not support event handlers.

The following kinds of event handlers are available in PowerDesigner:

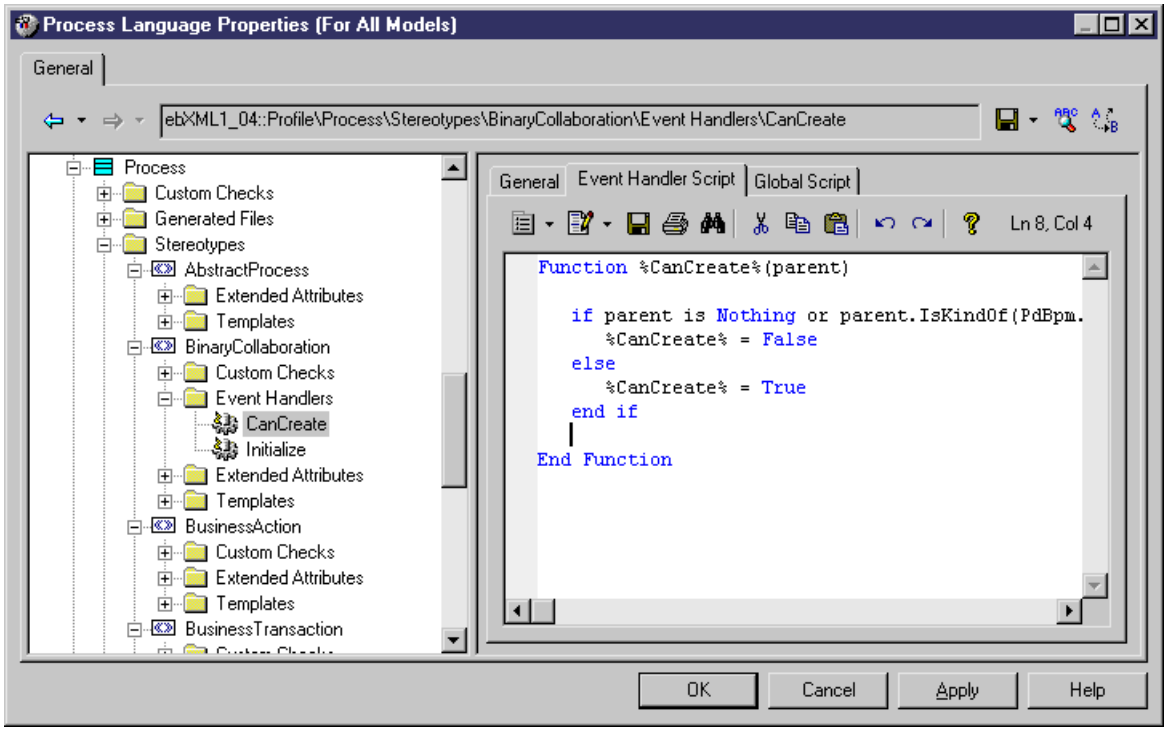
| Event handler | Description |
|---|---|
| CanCreate | Used to implement a creation validation rule to prevent objects from being created in an invalid context. For example, in a BPM for ebXML, a process with a Business Transactions stereotype can only be created under a process with a Binary Collaboration stereotype. The script of the CanCreate event handler associated with the Business Transaction process stereotype is the following:<br>`Function %CanCreate%(parent)`<br>`  if parent is Nothing or`<br>`  parent.IsKindOf(PdBpm.Cls_Process) then`<br>`  %CanCreate% = False`<br>`  else`<br>`  %CanCreate% = True`<br>`  end if`<br>`End Function`<br>If this event handler is set on a *stereotype* and returns True, then you can use the custom tool to create the stereotyped object. Otherwise the tool is not available, and the stereotype list excludes corresponding stereotype. If it is set on a *metaclass* and returns True, then you can create the object from the palette, from the Browser or in a list.<br>Note that, when you import or reverse engineer a model, the CanCreate functions are ignored since they could modify the model and make it diverge from the source. |
| Initialize | Used to instantiate objects with predefined templates. For example, in a BPM, a Business Transaction must be a composite process with a predefined sub-graph. The script of the Initialize event handler associated with the Business Transaction process stereotype contains all the functions needed to create the sub-graph. The following piece of script is a subset of the Initialize event handler for a Business Transaction.<br>`...`<br>`' Search for an existing requesting activity`<br>`    symbol`<br>`Dim ReqSym`<br>`Set ReqSym = Nothing`<br>`If Not ReqBizAct is Nothing Then`<br>`  If ReqBizAct.Symbols.Count > 0 Then`<br>`   Set ReqSym = ReqBizAct.Symbols.Item(0)`<br>`  End If`<br>`End If`<br>`' Create a requesting activity if not found`<br>`If ReqBizAct is Nothing Then`<br>`  Set ReqBizAct =`<br>`     BizTrans.Processes.CreateNew`<br>`  ReqBizAct.Stereotype =`<br>`    "RequestingBusinessActivity"`<br>`  ReqBizAct.Name = "Request"`<br>`End If`<br>`...`<br>If the Initialize event handler is set on a *stereotype* and returns True, the initialization script will be launched whenever the stereotype is assigned, either with a custom tool in the palette, or from the object property sheet. If it is set on a *metaclass* and returns True, the initialization script will be launched when you create a new object from the tool palette, from the Browser, in a list or in a property sheet.<br>If it is set on the *model* metaclass and returns True, the initialization script is launched when you assign a target (DBMS or object, process, or schema language) to the model at creation time, when you change the target of the model or when you assign an extended model definition to the model. |

| Event handler | Description |
|---|---|
| Validate | Used to validate changes to object properties and to implement cascade updates. It is triggered when change tabs or click OK or Apply in the object property sheet.<br>You can define an error message that will appear when the condition is not satisfied. To do so, fill the message variable and set the %Validate% variable to False.<br>In the following example, the validate event handler verifies that a comment is added to the definition of an object when the user validates the property sheet. A message is displayed to explain the problem.<br>`Function %Validate%(obj, ByRef message)`<br>` if obj.comment = "" then`<br>`  %Validate% = False`<br>`  message = "Comment cannot be empty"`<br>` else`<br>`  %Validate% = True`<br>` end if`<br>`End Function` |
| CanLinkKind | [link objects only] Used to restrict the kind and stereotype of the objects that can be linked together. It is triggered when you create a link with a palette tool or modify link ends in a property sheet.<br>This event handler has two input parameters: its source and destination extremities. You can also use the sourceStereotype and destinationStereotype parameters. These are optional and used to perform additional checks on stereotypes.<br>In the following example, the source of the extended link must be a start object:<br>`Function %CanLinkKind%(sourceKind,`<br>`  sourceStereotype, destinationKind,`<br>`  destinationStereotype)`<br>` if sourceKind = cls_Start Then`<br>` %CanLinkKind% = True`<br>` end if`<br>`End Function` |

## To Add an Event Handler to a Metaclass or a Stereotype:

You can create an event handler in a profile.

1. Right-click a metaclass or a stereotype and select **New > Event Handler** to open a selection box, listing the available event handlers.
2. Select one or more event handlers and click OK to add them to the metaclass.
3. Click on the event handler in the tree view, and enter a name and comment.
4. Click the Event Handler Script tab and enter your script:




5. Click Apply to save your changes.

## Event Handler Properties

The following properties are available for event handlers:

| Property | Description |
|---|---|
| Name | Specifies the name of the event handler. |
| Comment | Provides a description of the event handler. |
| Event Handler Script | This tab specifies the VBScript that will run when the event occurs. Note that you should not use statements such as msgbox, or input box to open a dialog box in the event handler function. |
| Global Script | This tab can be used for sharing library functions and static attributes in the resource file.<br>For more information on defining a script and using the Global Script tab, see *Defining the script of a custom check* on page 152 and *Using the global script* on page 155. |

# Methods (Profile)

Methods allow you to perform actions on objects.

They are written in VBScript, and are invoked by other profile components, such as menu items (see *Menus (Profile)* on page 161) or form buttons (see *Forms (Profile)* on page 141).

The following example method, created in the Class metaclass, converts classes into interfaces. It copies class basic properties and operations, deletes the class (to avoid namespace problems), and creates the new interface.

Note that this script does not deal with other class properties, or with interface display, but a method can be used to launch a custom dialog box to ask for end-user input before performing its action (see *Example: Creating a dialog box launched from a menu* on page 149).

```
Sub %Mthd%(obj)
 ' Convert class to interface
```

```
' Copy class basic properties
Dim Folder, Intf, ClassName, ClassCode
Set Folder = obj.Parent
Set Intf = Folder.Interfaces.CreateNew
ClassName = obj.Name
ClassCode = obj.Code
Intf.Comment = obj.Comment

' Copy class operations
Dim Op
For Each Op In obj.Operations
 ' ...
 Output Op.Name
Next

' Destroy class
obj.Delete

' Rename interface to saved name
Intf.Name = ClassName
Intf.Code = ClassCode
End Sub
```

For detailed information about using VBScript in PowerDesigner, see *Scripting PowerDesigner* on page 235.

## To Create a Method:

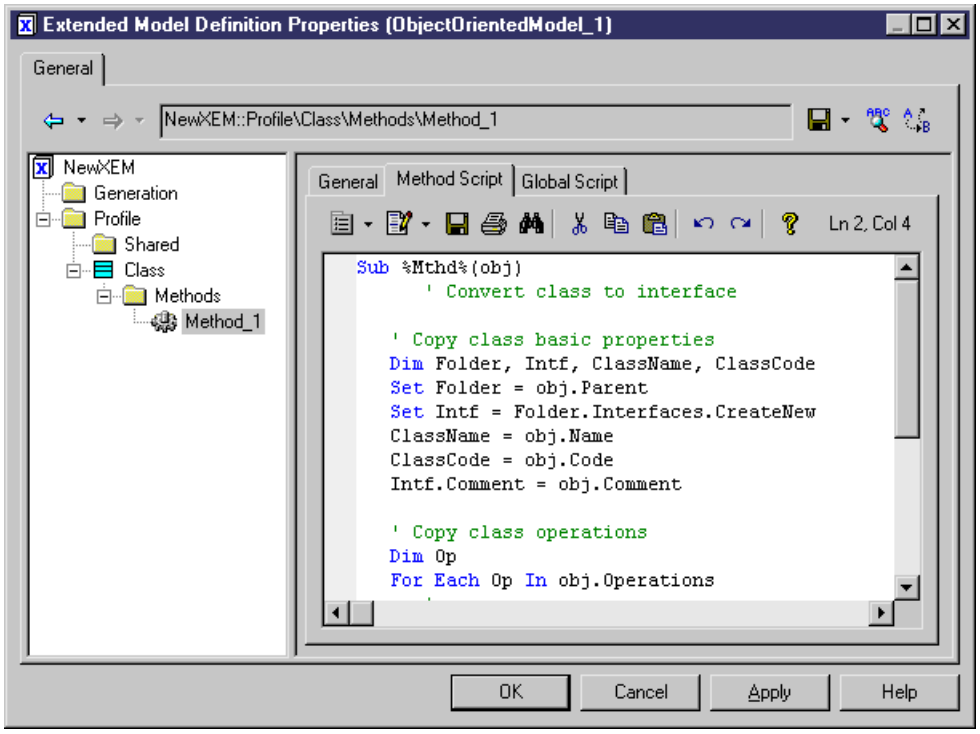
You can create a method in a profile.

1. Right-click a metaclass, stereotype or criterion and select **New** > **Method** .
2. Enter a name and a comment to describe the method.
3. Click the Method Script tab, and enter the VBscript. If appropriate, you can reuse functions on the Global Script tab.

   By default, this tab contains the following skeleton script:

   ```
   Sub %Method%(obj)
     ' Implement your method on <obj> here
   End Sub
   ```

   %Method% is a concatenation of the name of the resource file, the metaclass (and any stereotype or criterion), and the name of the method itself defined in the General tab. If any of these names contains an empty space, it is replaced by an underscore.




4. Click Apply.

## Method Properties

The following properties can be set for methods:

| Property | Description |
|---|---|
| Name | Name of the method that identifies a script |
| Comment | Additional information about the method |

The *Method Script* tab contains the body of the method function.

The *Global Script* tab is used for sharing library functions and static attributes in the resource file. This tab is shared with event handlers and transformations.

You can declare *global variables* on this tab, but you should be aware that they will not be reinitialized each time the method is executed, and keep their value until you modify the resource file, or the PowerDesigner session ends. This may cause errors, especially when variables reference objects that can be modified or deleted. Make sure you reinitialize the global variable at the beginning of a method if you do not want to keep the value from a previous run.

For more information on defining a script and using the Global Script tab, see *Defining the script of a custom check* on page 152 and *Using the global script* on page 155.

# Menus (Profile)

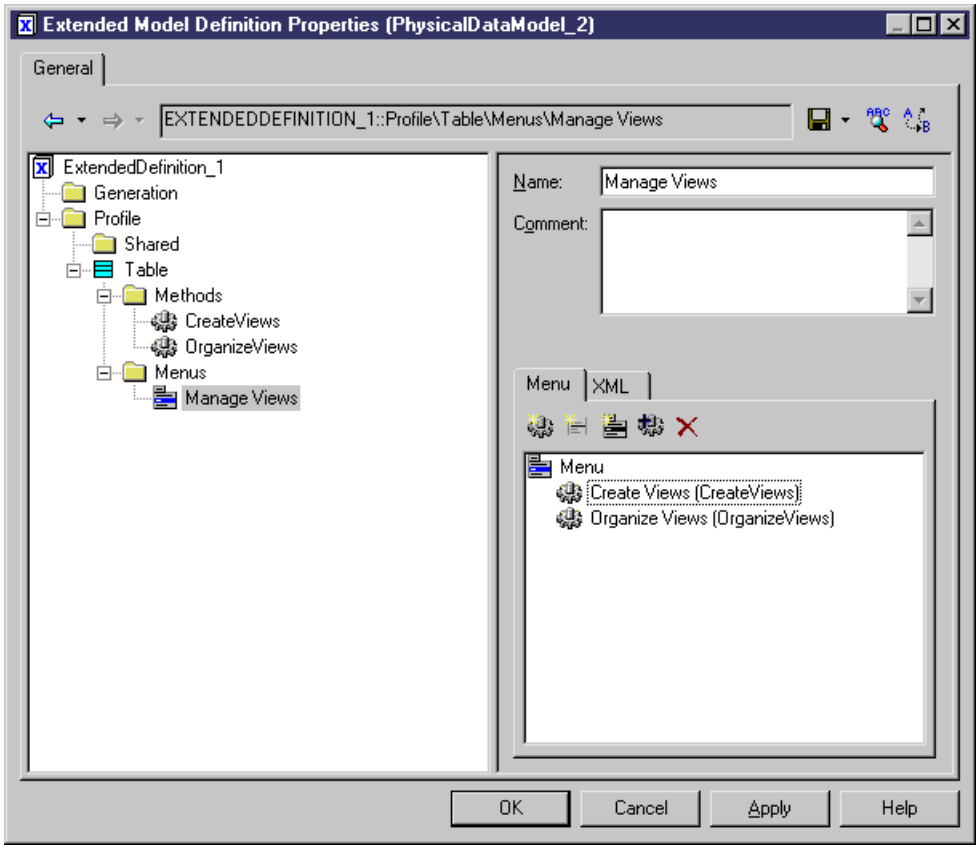
You can add menus in the PowerDesigner interface and fill them with commands that call method or transformations.

For more information on methods and transformations, see *Methods (Profile)* on page 159 and *Transformations and Transformation Profiles (Profile)* on page 165.

Menus can be added to the PowerDesigner File, Tools, and Help menus when defined on the model or a diagram metaclass, or on the contextual menus of diagram symbols and browser items. Menus defined on a parent metaclass are

inherited by its children. For example, you could generalize a contextual menu by defining it on a parent metaclass like BaseObject.

1. Right-click a metaclass, stereotype or criterion and select **New** > **Menu** .
2. Enter a name and comment (and, in the case of model or diagram metaclasses, a location).
3. Use the tools on the Menu sub-tab to create the items in your menu (see *Adding commands and other items to your menu* on page 163).




4. Click Apply to save your changes.

## Menu Properties

The following properties are available for menus:

| Property | Description |
|---|---|
| Name | Specifies the internal name of the menu. This name will not appear in the menu |
| Comment | Provides a description of the menu. |
| Location | [model and diagram only] Specifies where the menu will be displayed. You can choose between:<br>• File > Export menu<br>• Help menu<br>• Object Contextual Menu<br>• Tools menu<br>Menus created on other metaclasses are only available on the contextual menu, and do not display a Location field. |
| Menu | This sub-tab provides tools to add items to your menu (see *Adding commands and other items to your menu* on page 163). |

| Property | Description |
|---|---|
| XML | This sub-tab displays the XML generated from the Menu sub-tab. |

## Adding Commands and Other Items to Your Menu

You insert items into your menu using the tools in the Menu tab toolbar.

You can reorder items in the menu tree by dragging and dropping them. To place an item inside a submenu item, drop it onto the submenu.

| Tool | Function |
|---|---|
|  | Add Command - Opens a selection dialog box to allow you to add one or more methods or transformations to the menu as commands. This list is limited to methods and transformations defined in the current metaclass and its parents.<br>When you click OK, each selected method is added to your menu in the format: <Caption> (<Method/Transformation name>).<br>The *caption* is the command name that will appear in the menu. You can define a shortcut key in the caption by adding an ampersand before the shortkey letter.<br>Methods or transformations associated with menu commands are not synchronized with those defined in a metaclass. Thus, if you modify the name or the script of a method or transformation, you should use the Find tool to locate and update all the commands using this method or transformation.. |
|  | Add Separator -Creates a menu separator under the selected item. |
|  | Add Submenu - Creates a submenu under the selected item. |
|  | Deletes the selected item. |

# Templates and Generated Files (Profile)

The PowerDesigner Generation Template Language (GTL) is used to generate files from metaclasses and for scripting. You write a template in GTL, using variables that allow you to access properties of the current object or any other object in the model.

You can define templates and generated files for metaclasses, stereotypes, and criteria. If a template applies to all metaclasses, then you should create it in the Shared category.

For detailed information about the GTL, see *Customizing Generation with GTL* on page 187.

## Creating a Template

Templates can be created in the Shared category when they apply to all metaclasses. They can also be created at the metaclass level or for a given stereotype or criterion.

**Note:** Previously, you would bind the use of a particular template to a stereotype using the template name <<stereotype>> syntax. Now, you create the template beneath the stereotype in the profile.

You can use the Browse tool to find all templates of the same name. To do so, open a template, position the cursor on a template name in-between % characters, and click Browse (or F12). This opens a Result window that displays all templates prefixed by their metaclass name. You can double-click a template in the result window to locate its definition in the resource editor.

1. Right-click a metaclass, a stereotype, or a criterion and select **New** > **Template** to create a template.
2. Enter a name in the Name box. We recommend that you do not use spaces in the name.

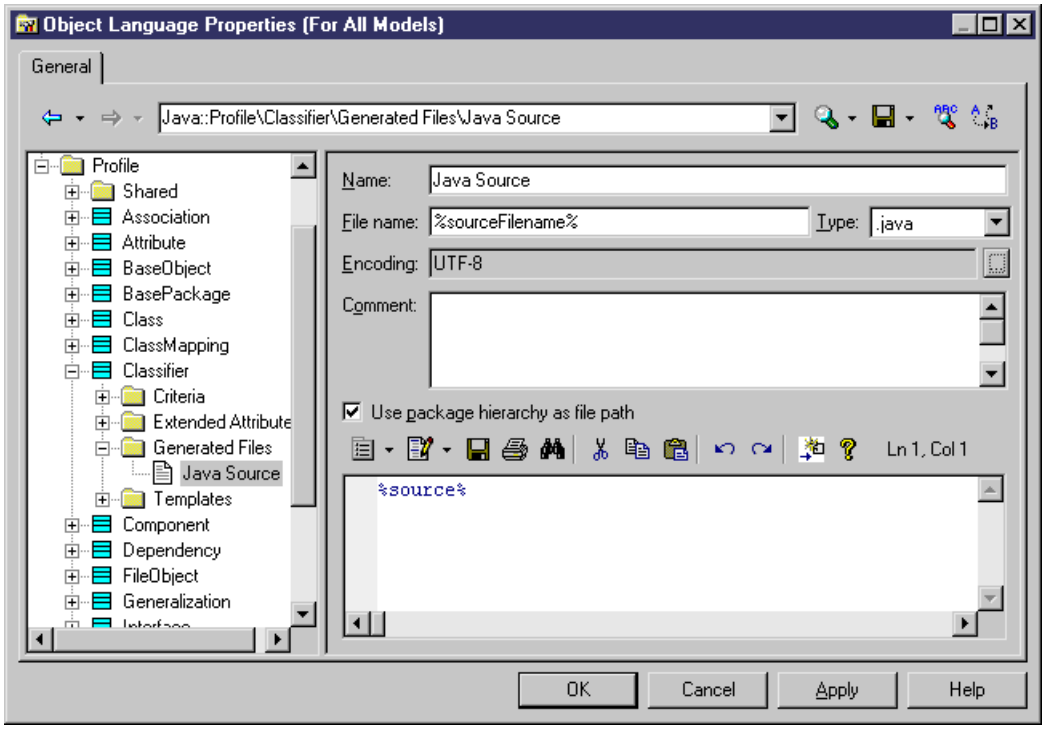
3. [optional] Enter a comment to explain the use of the template.
4. Enter the template body using GTL in the central box.

## Creating a Generated File

The Generated Files category contains an entry for each type of file that will be generated for the metaclass, stereotype, or criterion. Only files defined for objects belonging directly to a model or a package collection will be generated. Sub-objects, like attributes, columns, or parameters do not support file generation, but it can be interesting to see the code generated for these sub-objects in their Preview tab.

**Note:** If an extended model definition attached to the model contains a generated file name identical to one defined in the main resource file, then the extended model definition generated file will be generated.

1. Right-click a metaclass, stereotype, or criterion, and select **New** > **Generated File** .
2. Enter a name in the Name box, specify a file name, and select a file type to provide syntax coloring.
3. Enter the template for the contents of the generated file in the text zone:




4. Click Apply to save your changes.

Each generated file has the following properties:

| Property | Description |
|---|---|
| Name | Specifies a name for the file entry in the resource editor. |
| File Name | Specifies the name of the file that will be generated. This field can contain GTL variables. For example, to generate an XML file with the code of the object for its name, you would enter %code%.xml.<br>If you leave this field empty, then no file will be generated, but you can view the code produced in the object's Preview tab. |
| Type | Specifies the type of file to provide appropriate syntax coloring in the Preview window. |

| Property | Description |
|---|---|
| Encoding | Specifies the encoding format for the file. Click the ellipsis tool to the right of the field to choose an alternate encoding from the Text Output Encoding Format dialog box.. In this dialog box, select the Abort on Character Loss check box to stop generation if characters will be lost. |
| Comment | Specifies additional information about the generated file |
| Use package hierarchy as file path | Specifies that the package hierarchy should be used to generate the hierarchy of file directories. |
| Generated file template (text zone) | Specifies the template of the file to generate. You can enter the template directly here or reference a template defined elsewhere in the profile. (see *Customizing Generation with GTL* on page 187). |

## Transformations and Transformation Profiles (Profile)

A transformation defines a set of actions to be executed during generation or upon request. You define a transformation in the profile category of an extended model definition on a metaclass or a stereotype or other criteria.

You define a transformation when you want to:

- Modify objects for a special purpose. For example, you can create a transformation in an OOM that converts <<control>> classes into components.
- Modify objects in a reversible way. This can be useful during round-trip engineering. For example, if you generate a PDM from an OOM in order to create O/R mappings, and the source OOM contains components, you can pre-transform components into classes for easy mapping to PDM tables. When you update the source OOM from the generated PDM, you can use a post-transformation to recreate the components from the classes.

Transformations can be used:

- In a transformation profile (see *Creating a Transformation Profile* on page 167) during model generation, or on demand. For more information, see "Applying Model Transformations" in the Models chapter of the *Core Features Guide*.
- As a command in a user-defined menu (see *Menus (Profile)* on page 161)

Transformations can be used to implement *Model Driven Architecture (MDA),* a process defined by the OMG, and which separates the business logic of an application from the technological means used to implement it. The goal is to improve the integration and interoperability of applications and as a result, reduce the time and effort spent in application development and maintenance.

MDA development uses UML modeling to describe an application at different levels of detail, starting with the construction of a *Platform-independent model (PIM)* which models the basic business logic and functionality, and ending in a *Platform-Specific Model (PSM)* which includes implementation technologies (like CORBA, .NET, or Java). Between the initial PIM and the final PSM, there may be other intermediate models.

PowerDesigner allows you to create an initial PIM and refine it progressively in different models containing increasing levels of implementation and technology-dependent information. You can define transformations that will generate a more refined version of a model, based on the desired target platform. When changes are made to the PIM, they can be cascaded down to the generated models.

Transformations can also be used to apply design patterns to your model objects.

1. Right-click a metaclass or stereotype, and select **New** > **Transformation** from the contextual menu.
2. Enter the appropriate properties including a transformation script.

### Transformation Properties

A transformation has the following properties:

| Property | Description |
|---|---|
| Name | Name of the transformation. Make sure you type understandable names in order to easily identify them in selection lists |
| Comment | Additional information about the transformation used to explain the script |

### Transformation Script Tab

The Transformation Script tab contains the following properties:

| Name | Definition |
|---|---|
| CopyObject | Duplicates an existing object and set a source for the duplicated object.<br>Parameters:<br>• source: object to duplicate<br>• tag [optional]: identifier<br>Returns: A copy of the new object |
| SetSource | Sets the source object of a generated object. It is recommended to always set the source object to keep track of the origin of a generated object.<br>Parameters:<br>• source: source object<br>• target: target object<br>• tag [optional]: identifier<br>Returns: |
| GetSource | Retrieves the source object of a generated object.<br>Parameters:<br>• target: target object<br>• tag [optional]: identifier<br>Returns: Source object |
| GetTarget | Retrieves the target object of a source object.<br>Parameters:<br>• source: source object<br>• tag [optional]: identifier<br>Returns: Target object |

### Generation History

Since a source object can be transformed and have several targets, you may have problems identifying the origin of an object, especially in the merge dialog box. The following mechanism is used to help identify the origin of an object:

### If the Source Object Is Transformed into a Single Object

Then the transformation is used as an internal identifier of the target object.

### If the Source Object Is Transformed into Several Objects

Then you can define a specific *tag* to identify the result of transformation. You should use only alphanumeric characters, and we recommend that you use a "stable" value such as a stereotype, which will not be modified during repetitive generations.

For example, OOM1 contains the class Customer, to which you apply a transformation script to create an EJB. Two new classes are created from the source class, one for the home interface, and one for the remote interface. In the transformation script, you should assign a tag "home" for the home interface, and "remote" for the remote interface. The tag is displayed between <> signs in the Version Info tab of an object, beside the source object.

In the following example, the tag mechanism is used to identify the classes attached to a component:

```
 'setting tag for all classes attached to component
 for each Clss in myComponent.Classes
  if clss <> obj then
   trfm.SetSource obj,Clss," GenatedFromEJB"+ obj.name +"target" +Clss.Name
   For each ope in clss.Operations
     if Ope.Name = Clss.Code Then 'then it is a constructor _Bean operation
      trfm.SetSource obj,Ope," GenatedFromEJB"+ obj.name +"target" +Ope.Name
     end if
     if Ope.Name = Clss.Name Then 'then it is a constructor operation
      trfm.SetSource obj,Ope," GenatedFromEJB"+ obj.name +"target" +Ope.Name
     end if
     if Ope.name = "equals" Then 'then it is an equals operation and should be
tagged
      trfm.SetSource obj,Ope," GenatedFromEJB"+ obj.name +"target" +Ope.Name
     end if
   next
   end if
 next
```

### Script Checks

Transformation scripts do not require as many checks as standard scripts, which require that you verify the content of a model in order to avoid errors, because transformations are always implemented in a temporary model where there is no existing object. The temporary model is merged with the generation target model if the Preserve modification option is selected during update.

If you create a transformation using an existing script, you can remove these controls.

### Internal Transformation Object

Internal transformation objects do not appear in the PowerDesigner interface; they are created as temporary objects passed to the script so that the user can access the helper functions, and also to record the execution of a sequence of transformations in order to be able to execute them later.

Internal transformation objects are preserved when the transformations are used by the Apply Transformations feature or in a menu, so that when you update a model (regenerate) in which these kind of transformations have been executed, the transformations can be re-executed in the source model in order to maintain an equality between the source and the target model.

For example, CDM1 contains an entity A. You generate an OOM from CDM1 and class B is created. You apply some transformations to class B in OOM1 in order to create class C. When you re-generate CDM1 and update OOM1, class B will be generated from entity A but class C is missing in the generated model, and shows as a difference in the merge dialog box. However, thanks to the internal transformation objects, the transformations which were executed in the generated OOM are re-executed and you obtain class C and the models to be merged are more similar than before.

### Global Script and Dependencies Tabs

The Global Script tab provides access to definitions shared by all VBScript functions defined in the profile, and the Dependencies tab lists the transformation profiles in which the transformation is used.

## Creating a Transformation Profile

A transformation profile is a group of transformations applied during model generation when you want to apply changes to objects in the source or target models.

You define a transformation profile in the Transformation Profiles category of an extended model definition (see *Transformations and Transformation Profiles (Profile)* on page 165). Each profile is identified by the model in which the current extended model definition is defined, a model type, a family and a subfamily.

1. [if the Transformation Profiles category is not present] Right-click the root node, select Add Items from the contextual menu, select Transformation Profiles and click OK to create this folder beneath the root node.
2. Right click the Transformation Profiles folder, and select New from the contextual menu. A new transformation profile is created in the folder.
3. Define the appropriate properties and add one or more transformations using the Add Transformations tool on the Pre-generation or Post-generation tabs. These transformations should have been previously defined in the Profile \Transformations category.

### Transformation Profile Properties

You define a transformation profile using the following properties:

| Property | Description |
|---|---|
| Name | Name of the transformation profile |
| Comment | Additional information about the transformation profile |
| Model Type | [optional] Specifies the type of model with which the transformation profile can be used. This is a way to filter profiles during generation. For example, if you select OOM when the current extended model definition is in a PDM, the transformation profile can be used during PDM to OOM generation or reverse engineering |
| Family and sub-family | [optional] If the model type supports a target resource file, you can also define a family and subfamily to filter the display of profiles in the generation dialog box. The family is used to establish a link between the resource file of a model and an extended model definition. When the resource file family corresponds to the extended model definition family, it suggests that the extended model definition complements the resource file |
| Pre-generation | The Pre-generation tab lists the transformations to be executed before model generation. These transformations are executed when the current model in which you have created the extended model definition is the source model, and when the constraints defined in the model type, family, and subfamily boxes are met.<br>Any object created during pre-generation is automatically added to the list of objects used in generation.<br>These changes to the source model are temporary and are reversed after generation is complete.<br>For example, you can define a transformation profile with a transformation that cancels the creation of EJBs from classes before generating an OOM into a PDM in order to establish a better mapping between classes and tables during generation. |
| Post-generation | The Post-generation tab lists the transformations to be executed after generation. These transformations are executed when the current model in which you have created the extended model definition is the target model.<br>For example, you can define a transformation profile with a transformation that automatically applies the correct naming conventions to the generated model. |

## Using Profiles: a Case Study

To illustrate the concept of profile, you are going to build an extended model definition for an OOM. This model extension will let you design a *robustness diagram*.

### Robustness Diagram

The robustness diagram is used to get across the gap between what the system has to do, and how it is actually going to accomplish this task. In the UML analysis, the robustness diagram is between use case and sequence diagram analysis. It allows you to verify that the use case is correct and that you have not specified a system behavior that is unreasonable given the sets of objects you have. The robustness diagram also enables to verify if no object is missing in the model.

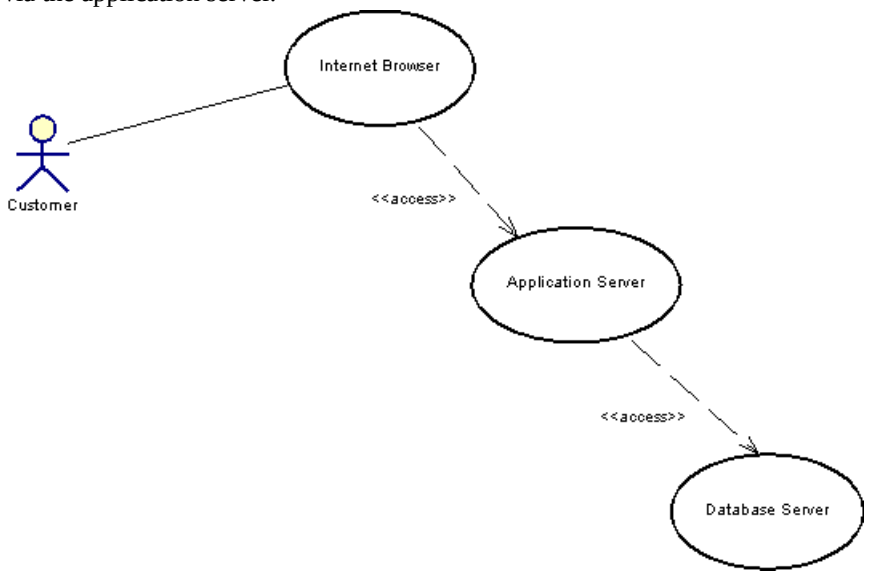
You are going to build a robustness diagram using the communication diagram in the OOM, together with an extended model definition containing a user-defined profile. This profile contains the following extensions:

- Stereotypes for objects
- Custom tool and symbol for each object stereotype
- Custom checks for instance links
- Generated file to output a description of messages exchanged between objects

You will follow this case study to create a new extended model definition. This extended model definition corresponds to the resource file Robustness.XEM, delivered by default and located in the \Resource Files\Extended Model Definitions folder of the PowerDesigner installation directory.

## Scenario

You are going to build the robustness extended model definition starting from a concrete example. The following use case represents a very basic Web transaction: a customer wants to know the value of his stocks in order to decide to sell or not. He sends a stock value query from his Internet Browser. The query is transferred from the browser to the database server via the application server.




You are going to use the robustness diagram to verify this use case diagram.

To do so, you will use a standard communication diagram and extend it with a profile.

## Attaching a New Extended Model Definition to the Model

In your workspace, you already have created an OOM with a use case diagram containing an actor and 3 use cases.

You have to create a new extended model definition and import it into the current model before starting the profile definition.

1. Select **Tools > Resources > Extended Model Definitions > Object-Oriented Model** .

   The List of Extended Model Definitions for an OOM is displayed.
2. Click the New tool in the list toolbar.

   The New Extended Model Definition dialog box is displayed.
3. Type Robustness_Extension in the Name box.
4. Keep <Default Template> in the Copy From box.
5. Click OK.

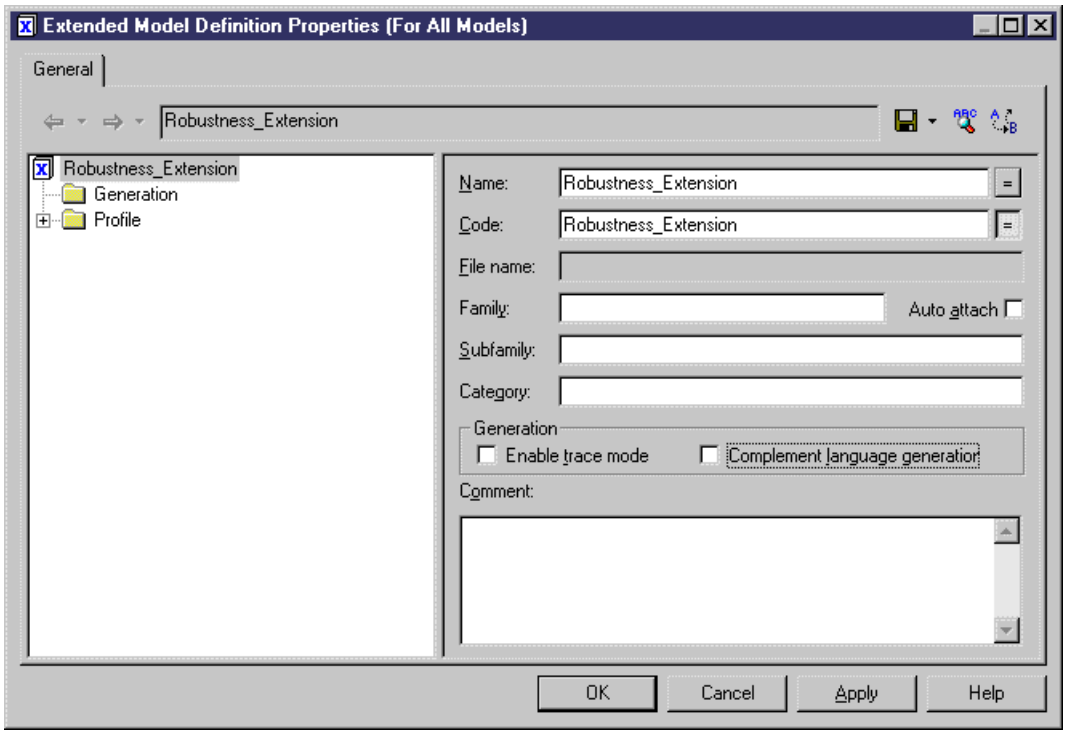
   A standard Save As dialog box is displayed. By default the file name is identical to the name of the extended model definition.

6. Click Save.

   The Extended Model Definition editor is displayed.

7. Clear the Complement Language Generation check box.

   This extended model definition does not belong to any object language family and will not be used to complement any object language generation.




8. Click OK to close the Extended Model Definition editor.
9. Click Close in the List of Extended Model Definitions.

   A Confirmation box asks you to save the extended model definition file.

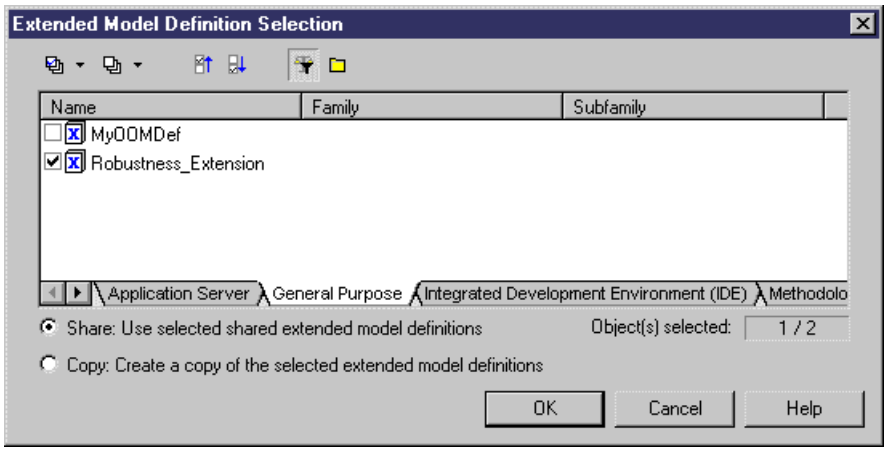
10. Click Yes.
11. Select **Model > Extended Model Definitions** .

    The List of Extended Model Definitions is displayed.

12. Click the Import an Extended Model Definition tool in the List toolbar.

    The Extended Model Definition Selection dialog box is displayed.

13. Click the General Purpose tab and select the Robustness_Extension check box.



14. Click OK, the extended model definition is displayed in the List of Extended Model Definitions.
15. Click OK in the list.
16. Right-click the model in the Browser, and select **New** > **Communication Diagram** in the contextual menu.

    The diagram property sheet is displayed.
17. Type Robustness Diagram in the name box and click OK in the property sheet.

## Create Object Stereotypes

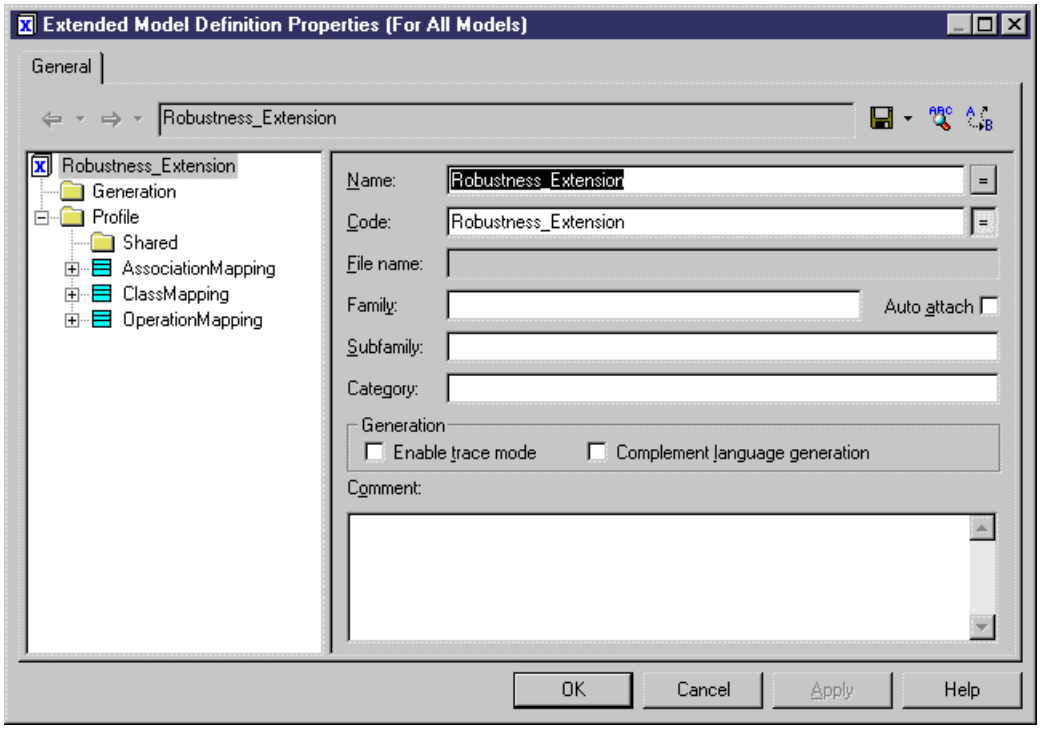
The robustness analysis classifies objects in three categories:

- *Boundary* objects are used by actors when communicating with the system; they can be windows, screens, dialog boxes or menus
- *Entity* objects represent stored data like a database, database tables, or any kind of transient object such as a search result
- *Control* objects are used to control boundary and entity objects, and represent transfer of information

To implement the robustness analysis in PowerDesigner, you are going to create stereotypes for objects in the communication diagram. These stereotypes correspond to the three object categories defined above. You will also attach custom tools in order to create a palette specially designed for creating objects with the <<Entity>>, <<Control>>, or <<Boundary>> stereotype.

You create these stereotypes in the Profile category of the extended model definition attached to your model.

1. Select **Model** > **Extended** Model Definition and double-click the arrow beside Robustness Extension in the list to display the resource editor.




2. Right-click the Profile category and select Add Metaclasses.

   The Metaclass Selection dialog box is displayed.
3. Select UMLObject in the list of metaclasses displayed in the PdOOM tab and click OK.

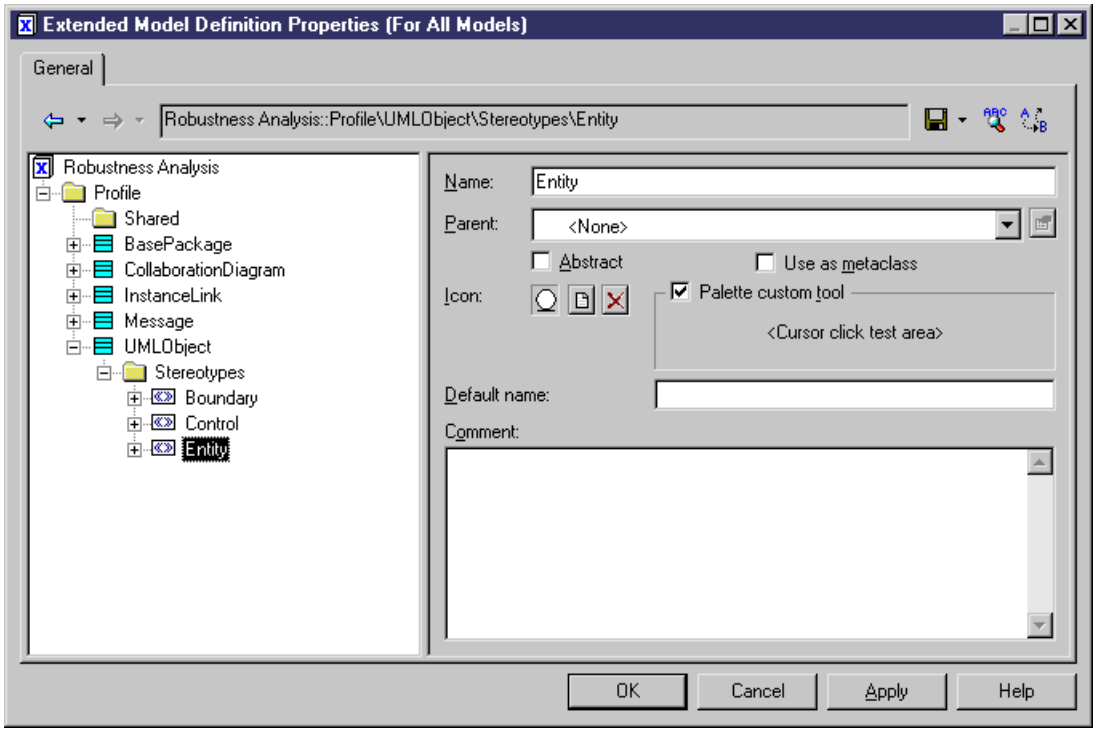
   The UMLObject category is displayed under Profile.
4. Right-click the UMLObject category and select **New** > **Stereotype** .

   A new stereotype is created under the UMLObject category.

5. Type Boundary in the Name box.
6. Select the Palette Custom Tool check box.
7. Click the Browse tool and select file boundary.cur in folder \Examples\Tutorial.
8. Repeat steps 4 to 7 to create the following stereotypes and tools:

| Stereotype | Cursor file |
|---|---|
| Entity | entity.cur |
| Control | control.cur |

You should see the 3 stereotypes under the UMLObject metaclass category.

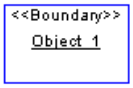



9. Click OK.
10. Click Yes to save the extended model definition.

The tool palette with the stereotype tools is displayed in the diagram.


11. Move the List of Extended Model Definitions to the bottom of the screen.
12. Click one of the tools and click in the communication diagram.

An object with the predefined stereotype is created:




13. Right-click to release the tool.

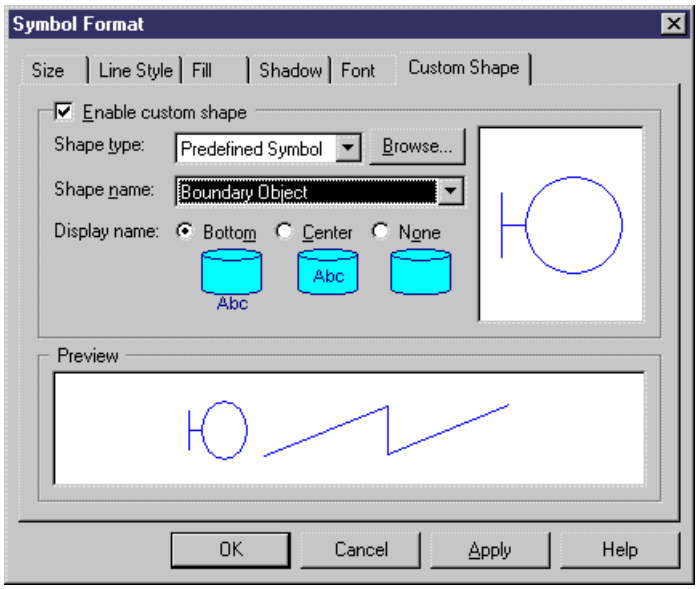
## Define Custom Symbols for Stereotypes

You are going to define a custom symbol for UMLObjects related to the Boundary, Control, or Entity stereotypes. The Custom Symbol feature lets you apply the standard robustness graphics into your communication diagram.

1. Double-click the arrow beside Robustness Extension in the list of extended model definitions to display the resource editor.
2. Right-click stereotype Boundary in the UMLObject category and select **New** > **Custom Symbol** .

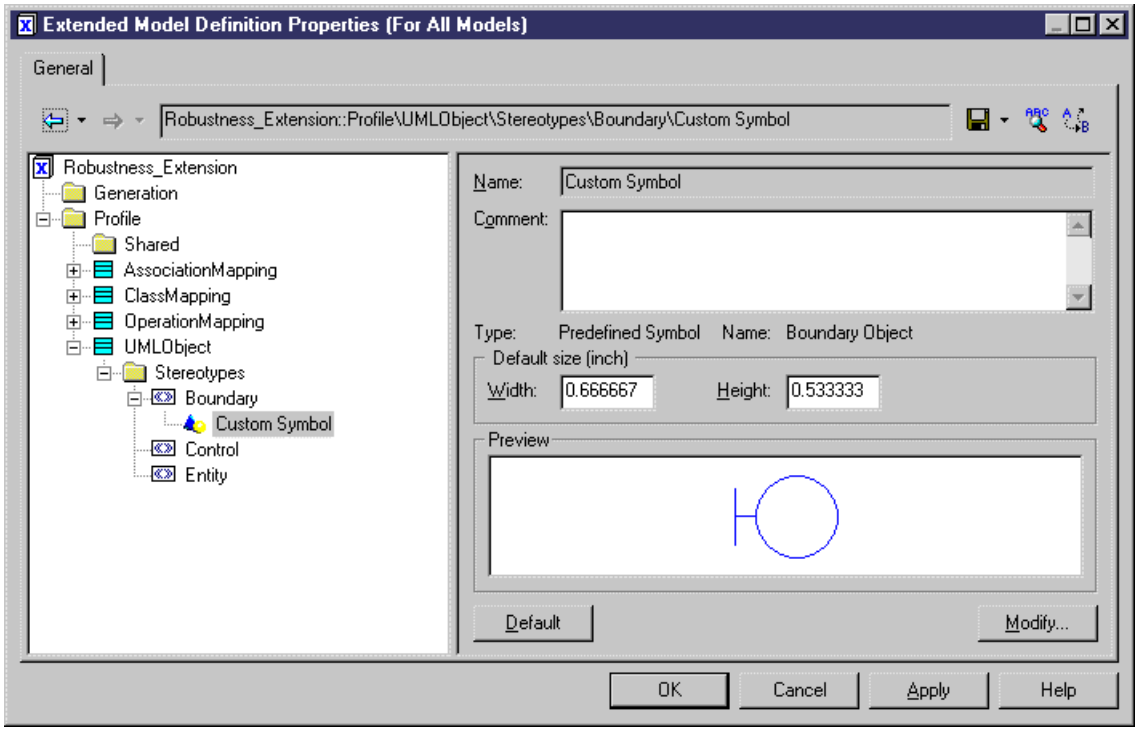
   A custom symbol is created.
3. Click the Modify button.

   The Symbol Format dialog box is displayed.
4. Click the Custom Shape tab.
5. Select the Enable Custom Shape check box.
6. Select Predefined Symbol in the Shape Type list.
7. Select Boundary Object in the Shape Name list.



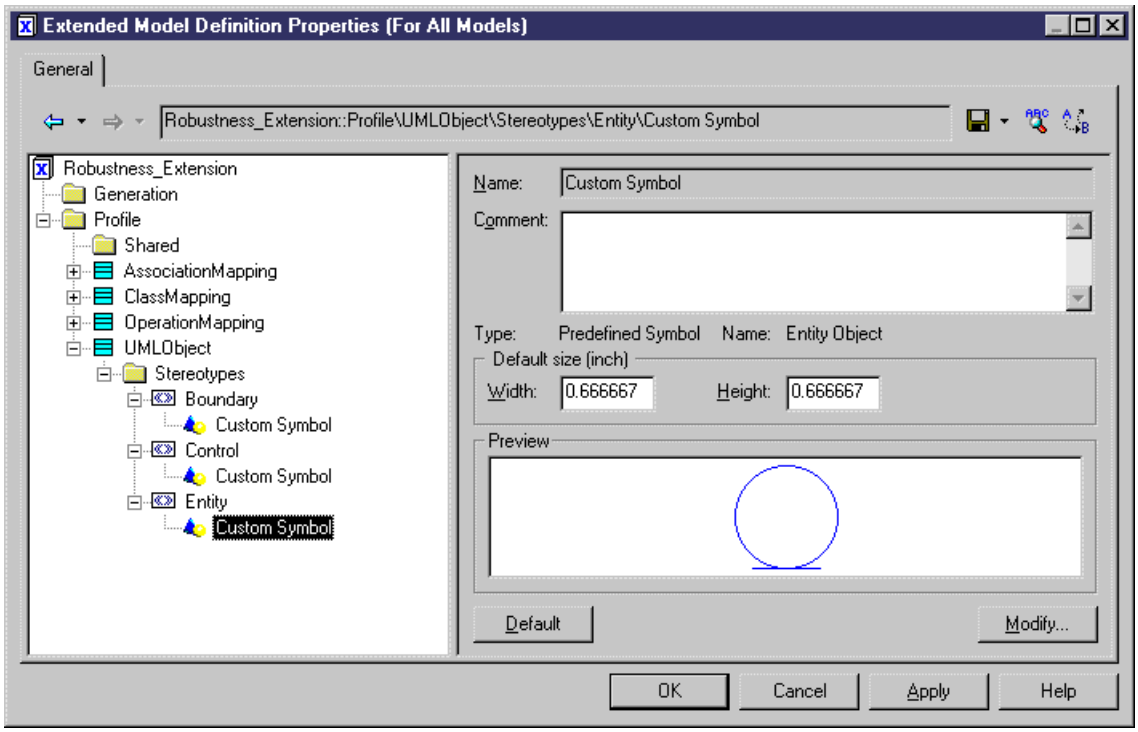

8. Click OK.

   The custom symbol is displayed in the Preview box.




**9.** Repeat steps 2 to 8 for the following stereotypes:

| Stereotype | Shape Name |
|---|---|
| Entity | Entity Object |
| Control | Control Object |




**10.** Click OK in each of the dialog boxes.

The symbol of the object you had previously created changes according to its stereotype:


11. In the communication diagram, create an object corresponding to each stereotype.

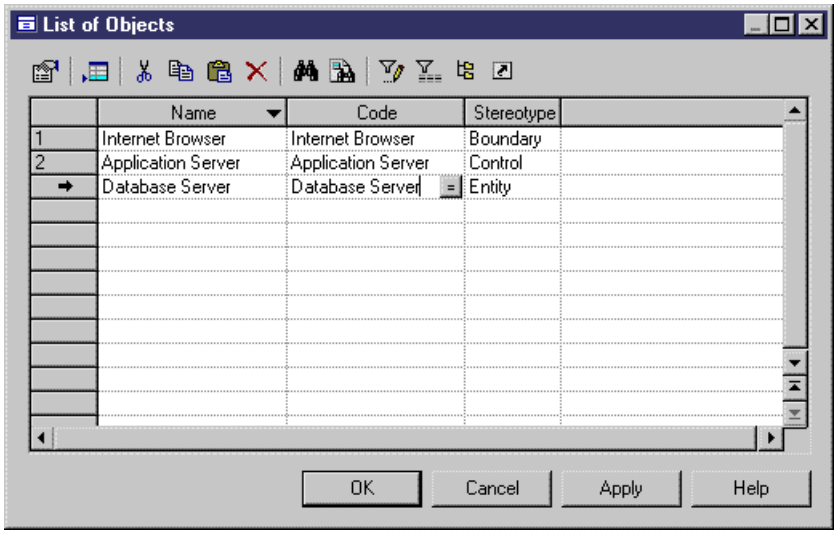
    Your diagram now contains 3 objects with different symbols corresponding to different stereotypes.




12. Select **Model > Objects** to display the list of objects.
13. Click the Customize Columns and Filter tool in the list toolbar and select Stereotype in the list of columns.

    The object stereotypes appear in the list. You are going to define the name and code of each object based on their stereotype.

| Object | Stereotype | Name & Code |
|---|---|---|
| Object_1 | <<Boundary>> | Internet Browser |
| Object_2 | <<Control>> | Application Server |
| Object_3 | <<Entity>> | Database Server |




14. Click OK in the List of Objects.
15. Drag the actor Customer from the Browser to the communication diagram in order to create a symbol for Customer.

## Create Instance Links and Messages Between Objects

You are going to create instance links between objects to express the collaboration between objects:

| Source | Destination |
|---|---|
| Customer | Internet Browser |
| Internet Browser | Application Server |
| Application Server | Database Server |

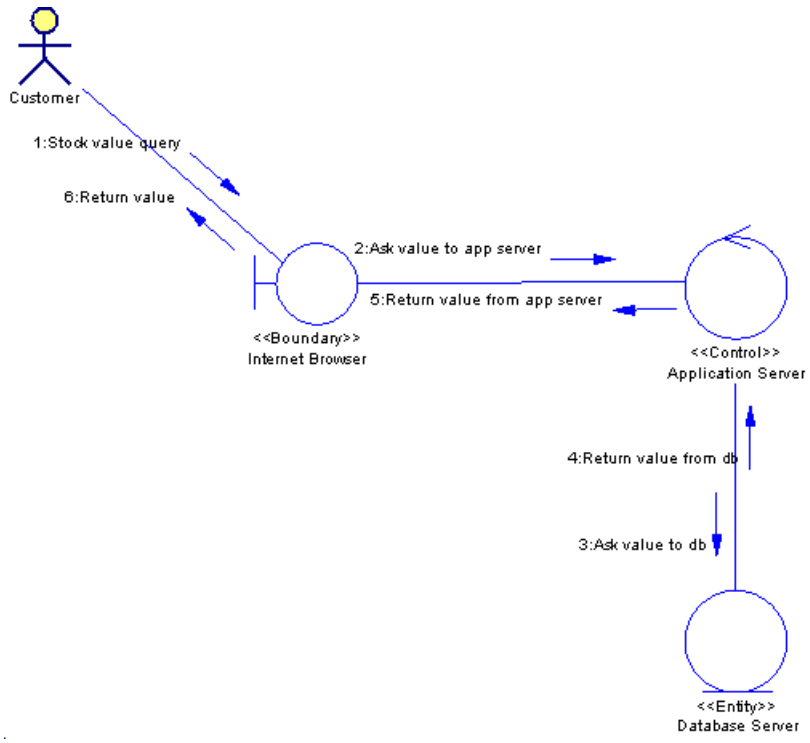
For more information on how to create instance links in the communication diagram, see "Creating an instance link in a communication diagram" in the Building Dynamic Diagrams chapter of the *Object-Oriented Modeling guide*.

You then define messages on the different instance links to express the data carried by the links.

For more information on how to create messages on instance links in the communication diagram, see "Creating a message in a communication diagram" in the Building Dynamic Diagrams chapter of the *Object-Oriented Modeling guide*.

You are going to create the following messages:

| Direction | Message name | Sequence number |
|---|---|---|
| Customer - Internet Browser | Stock value query | 1 |
| Internet Browser - Application Server | Ask value to app server | 2 |
| Application Server - Database Server | Ask value to db | 3 |
| Database Server - Application Server | Return value from db | 4 |
| Application Server - Internet Browser | Return value from app server | 5 |
| Internet Browser - Customer | Return value | 6 |



## Create Custom Checks on Instance Links

The robustness analysis implies some behavioral rules between objects. For example, an actor should always communicate with a boundary object (an interface), entity objects should always communicate with control objects, and so on. To represent these constraints, you are going to define custom checks on the instance links.

Custom checks do not prevent users from performing a syntactically erroneous action, but they allow you to define rules that will be verified by the Check Model function.

You define custom checks with VB scripting.

For more information on VBS syntax, see *Scripting PowerDesigner* on page 235.

1. Double-click the arrow beside Robustness Extension in the List of Extended Model Definitions to display the resource editor.
2. Right-click the Profile category and select Add Metaclass.

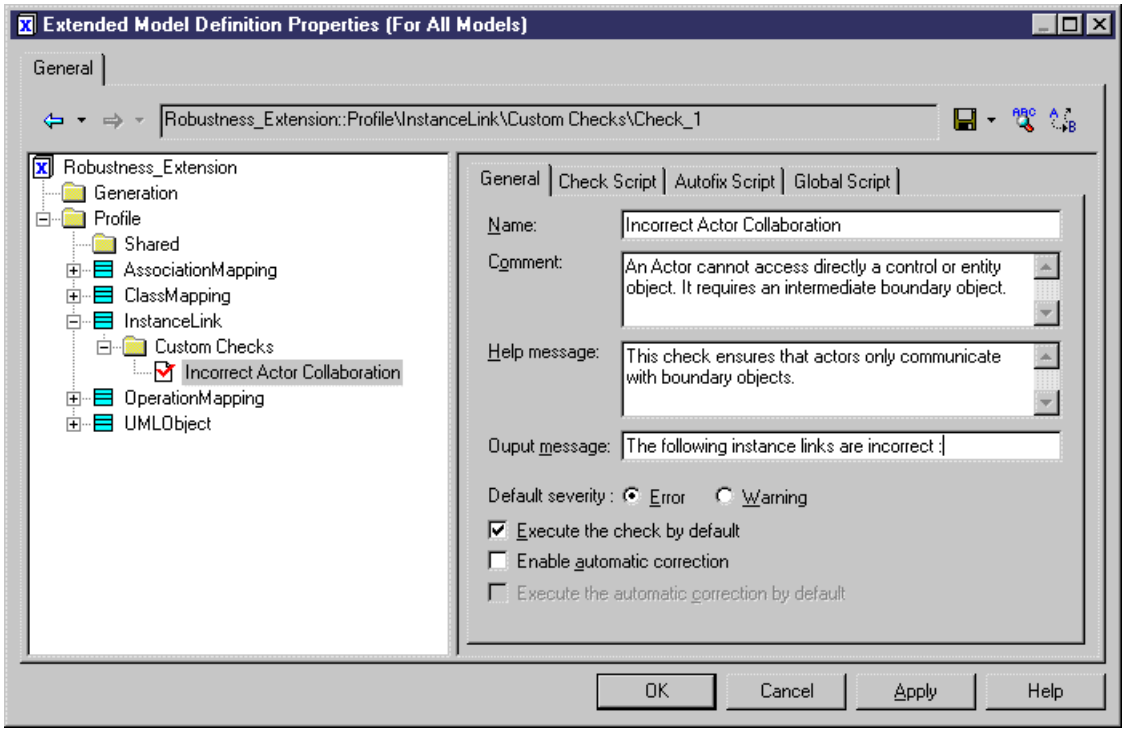
   The Metaclass Selection dialog box is displayed.
3. Select InstanceLink in the list of metaclasses displayed in the PdOOM tab and click OK.

   The InstanceLink category is displayed under Profile.
4. Right-click the InstanceLink category and select **New** > **Custom Check** .

   A new check is created.
5. Type Incorrect Actor Collaboration in the Name box.

   This check verifies if actors are linked to boundary objects. Linking actors to control or entity objects is not allowed in the robustness analysis.
6. Type "This check ensures that actors only communicate with boundary objects." in the Help Message box.

   This text is displayed in the message box that is displayed when the user selects Help in the custom check context menu in the Check Model Parameters dialog box.
7. Type " The following instance links are incorrect:" in the Output Message box.
8. Select Error as default severity.
9. Select the Execute the Check by Default check box.



**10.** Click the Check Script tab.

The Check Script tab is the tab where you type the script for the additional check.

**11.** Type the following script in the text zone.

```
Function %Check%(link)
 ' Default return is True
 %Check% = True
 ' The object must be an instance link
 If link is Nothing or not link.IsKindOf(PdOOM.cls_InstanceLink) then
  Exit Function
 End If
 ' Retrieve the link extremities
 Dim src, dst
 Set src = link.ObjectA
 Set dst = link.ObjectB
 ' Source is an Actor
 ' Call CompareObjectKind() global function defined in Global Script pane
 If CompareObjectKind(src, PdOOM.Cls_Actor) Then
  ' Check if destination is an UML Object with "Boundary" Stereotype
  If not CompareStereotype(dst, PdOOM.Cls_UMLObject, "Boundary") Then
   %Check% = False
  End If
Elsif CompareObjectKind(dst, PdOOM.Cls_Actor) Then
  ' Check if source is an UML Object with "Boundary" Stereotype
  If not CompareStereotype(src, PdOOM.Cls_UMLObject, "Boundary") Then
   %Check% = False
  End If
 End If
End Function
```

**12.** Click the Global Script tab.

The Global Script tab is the tab where you store functions and static attributes that may be reused among different functions.

**13.** Type the following script in the text zone.

```
' This global function check if an object is of given kind
' or is a shortcut of an object of given kind
Function CompareObjectKind(Obj, Kind)
 ' Default return is false
 CompareObjectKind = False
' Check object
 If Obj is Nothing Then
  Exit Function
 End If
' Shortcut specific case, ask to its target object
 If Obj.IsShortcut() Then
  CompareObjectKind = CompareObjectKind(Obj.TargetObject)
  Exit Function
 End If
If Obj.IsKindOf(Kind) Then
  ' Correct object kind
  CompareObjectKind = True
 End If
End Function
' This global function check if an object is of given kind
' and compare its stereotype value
Function CompareStereotype(Obj, Kind, Value)
 ' Default return is false
 CompareStereotype = False
 ' Check object
 If Obj is Nothing or not Obj.HasAttribute("Stereotype") Then
  Exit Function
 End If
```

```
' Shortcut specific case, ask to its target object
 If Obj.IsShortcut() Then
  CompareStereotype = CompareStereotype(Obj.TargetObject)
  Exit Function
 End If
If Obj.IsKindOf(Kind) Then
  ' Correct object kind
  If Obj.Stereotype = Value Then
   ' Correct Stereotype value
   CompareStereotype = True
  End If
 End If
End Function
```

**14.** Click Apply.

You are going to repeat steps 4 to 14 and create one check to verify that an instance link is not defined between two boundary objects:

| Check definition | Content |
|---|---|
| Name | Incorrect Boundary to Boundary Link |
| Help Message | This check ensures that an instance link is not defined between two boundary objects |
| Output Message | The following links between boundary objects are incorrect: |
| Default Severity | Error |
| Execute the check by default | Selected |

**15.** Type the following check in the Check Script tab:

```
Function %Check%(link)

 ' Default return is True
 %Check% = True

 ' The object must be an instance link
 If link is Nothing or not
link.IsKindOf(PdOOM.cls_InstanceLink) then
  Exit Function
 End If

 ' Retrieve the link extremities
 Dim src, dst
 Set src = link.ObjectA
 Set dst = link.ObjectB

 ' Error if both extremities are 'Boundary' objects
 If CompareStereotype(src, PdOOM.Cls_UMLObject, "Boundary") Then
  If CompareStereotype(dst, PdOOM.Cls_UMLObject, "Boundary") Then
   %Check% = False
  End If
 End If

End Function
```

**16.** Repeat steps 4 to 14 and create one check to verify that entity objects are accessed only from control objects:

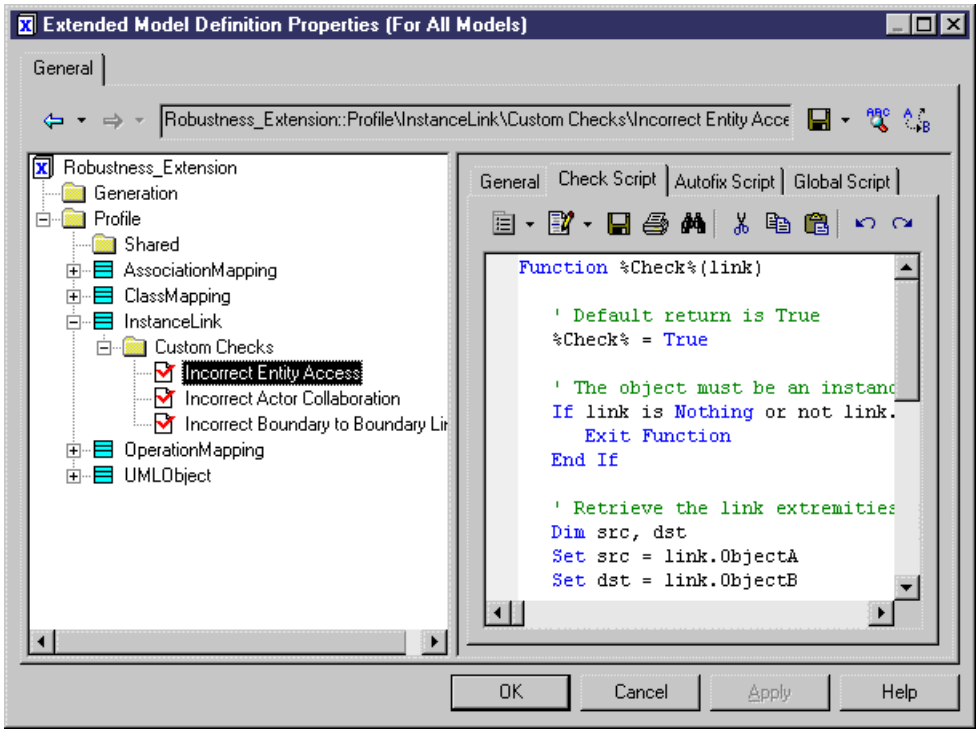
| Check definition | Content |
|---|---|
| Name | Incorrect Entity Access |

| Check definition | Content |
|---|---|
| Help Message | This check ensures that entity objects are accessed only from control objects |
| Output Message | The following links are incorrect: |
| Default Severity | Error |
| Execute the check by default | Selected |

17. Type the following check in the Check Script tab:

```
Function %Check%(link)
 ' Default return is True
 %Check% = True
 ' The object must be an instance link
 If link is Nothing or not link.IsKindOf(PdOOM.cls_InstanceLink) then
  Exit Function
 End If
 ' Retrieve the link extremities
 Dim src, dst
 Set src = link.ObjectA
 Set dst = link.ObjectB
 ' Source is and UML Object with "Entity" stereotype?
 ' Call CompareStereotype() global function defined in Global Script pane
 If CompareStereotype(src, PdOOM.Cls_UMLObject, "Entity") Then
  ' Check if destination is an UML Object with "Control" Stereotype
  If not CompareStereotype(dst, PdOOM.Cls_UMLObject, "Control") Then
   %Check% = False
  End If
 Elsif CompareStereotype(dst, PdOOM.Cls_UMLObject, "Entity") Then
  ' Check if source is an UML Object with "Control" Stereotype
  If not CompareStereotype(src, PdOOM.Cls_UMLObject, "Control") Then
   %Check% = False
  End If
 End If
End Function
```

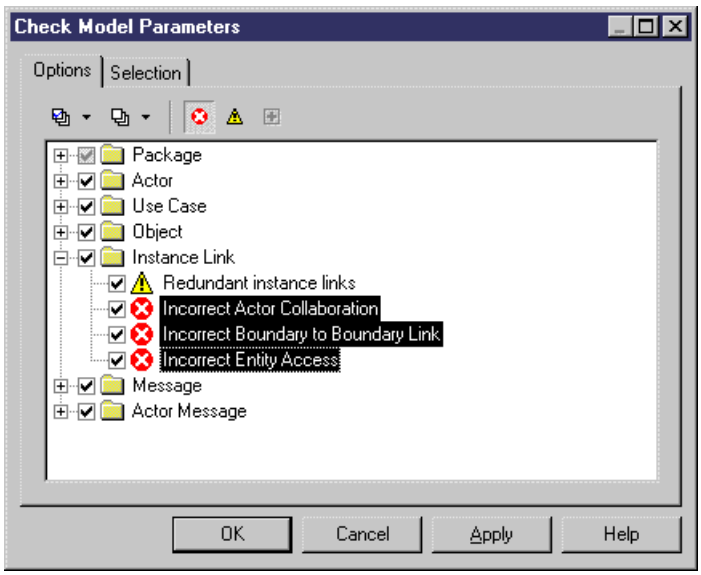
18. Click Apply.




**19.** Click OK in the resource editor.

**20.** Select **Tools** > **Check Model** to display the Check Model Parameters dialog box.

The custom checks appear in the Instance Link category.




You can test the checks by creating instance links between Customer and Application server for example, and then press F4 to start the check model feature.

For more information on the Check Model feature, see "Checking a Model" in the Models chapter of the *Core Features Guide*.

## Generate a Textual Description of Messages

You are going to generate a textual description of the messages existing in the communication diagram. The description should provide for each diagram in the model, the name of the message sender, the name of the message and the name of the receiver.

You generate this description using templates and generated files, because by default PowerDesigner does not provide such a feature. Templates and generated files use the PowerDesigner Generation Template Language (GTL). Generated files evaluate templates defined on metaclasses and output the evaluation result in files.

For more information on GTL, see *Customizing Generation with GTL* on page 187.

### What Template to Define and Where?

To generate a textual description of the communication diagram message, you have to define templates on the following metaclasses:

- *Message*: this metaclass contains details about the message sequence number and the message name, sender and receiver, this is why you define a template to evaluate the message sequence number and description in this metaclass
- *CommunicationDiagram*: you define in this metaclass the templates that will sort the messages in the diagram and gather all message descriptions from the current diagram

For more information on the PowerDesigner metaclasses, see *Resource Files and the Public Metamodel* on page 1.

The generated file is defined on metaclass *BasePackage* in order to scan the entire model, that is to say the model itself and the packages it may contain. This is to make sure that all messages in the model and its packages are described in the generated file. There will be one generated file since metaclass BasePackage has only one instance.

### Defining a Template for Generation

A generated file uses templates defined in metaclasses. The first template you have to define is on messages. The role of this template is to evaluate the message sequence number and to provide the name of the sender, the name of the message, and the name of the receiver for each message in the diagram.

The syntax for defining this template is the following:

```
.set_value(_tabs, "", new)
.foreach_part(%SequenceNumber%, '.')
.set_value(_tabs, " %_tabs%")
.next
%_tabs%%SequenceNumber%) %Sender.ShortDescription% sends message "%Name%" to
%Receiver.ShortDescription%
```

The first part of the template aims at creating indentation according to the sequence number of the message. The macro foreach_parts loops on the sequence numbers:

```
.foreach_part(%SequenceNumber%, '.')
.set_value(_tabs, " %_tabs%")
```

It browses each sequence number, and whenever a dot is found, it adds 3 empty spaces automatically for indentation. This calculates the _tab variable, which is later used to create the correct indentation based on the sequence numbering.

The last line contains the actual generated text of the template: for each sequence number, the appropriate tab value is created, followed by the name of the sender (short description), the text "sends message", then the name of the message, the text "to", and the name of the receiver.

1. Select **Model** > **Extended** Model Definition and double-click the arrow beside Robustness Extension in the List of Extended Model Definitions to display the resource editor.
2. Right-click the Profile category and select Add Metaclasses.

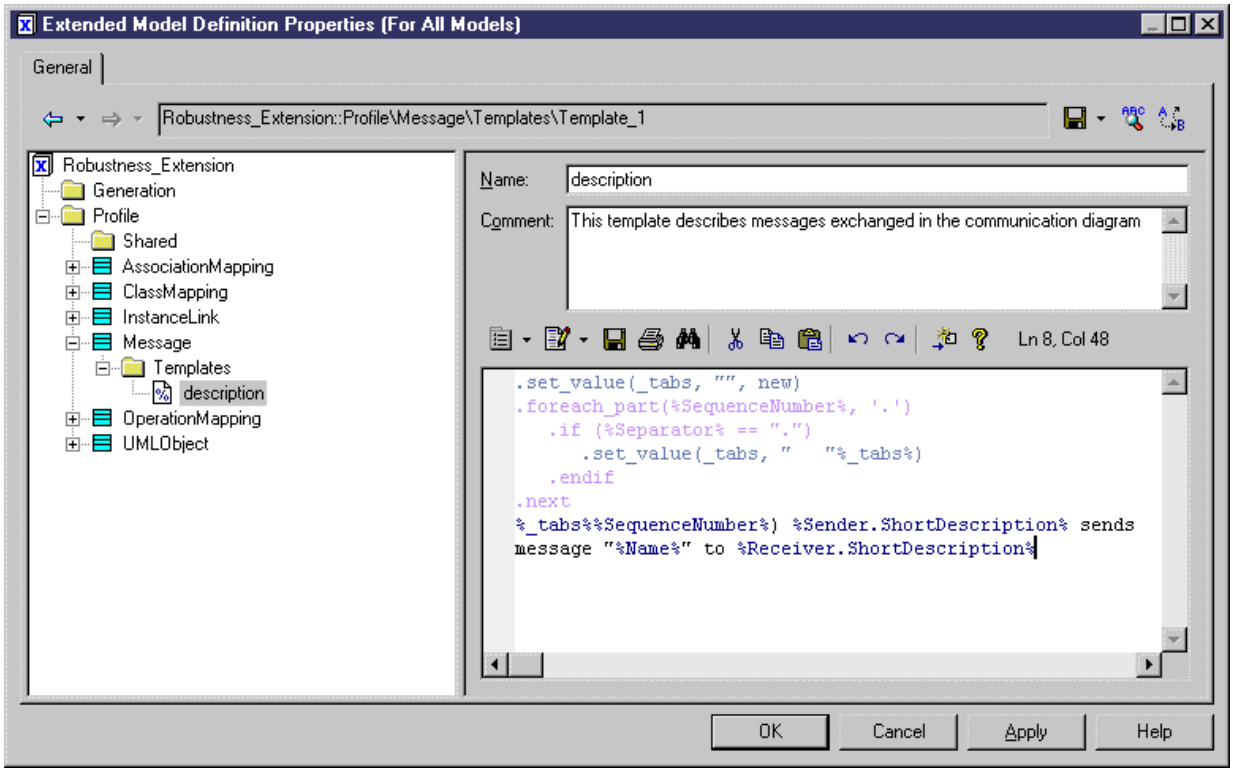
   The Metaclass Selection dialog box is displayed.
3. Select Message in the list of metaclasses displayed in the PdOOM tab and click OK.

   The Message category is displayed under Profile.
4. Right-click the Message category and select **New** > **Template** .

   A new template is created.
5. Type description in the Name box of the template
6. (optional) Type a comment in the Comment box of the template.

7. Type the following code in the text area:

```
.set_value(_tabs, "", new)
.foreach_part(%SequenceNumber%, '.')
 .if (%Separator% == ".")
  .set_value(_tabs, " "%_tabs%)
 .endif
.next
%_tabs%%SequenceNumber%) %Sender.ShortDescription% sends message "%Name%" to
%Receiver.ShortDescription%
```




8. Click Apply.

## Defining the Templates for the Communication Diagram Metaclass

Once you have defined the template used to evaluate each message sequence number, you have to create a template to sort these sequence numbers (`compareCbMsgSymbols`), and another template to retrieve all messages from the communication diagram (`description`).

The template `compareCbMsgSymbols` is a boolean that allows verifying if a message number is greater than another message number. The syntax of this template is the following:

```
.bool (%Item1.Object.SequenceNumber% >= %Item2.Object.SequenceNumber%)
```

The returned value for this template is used with parameter compare in template `description` which code is the following:

```
Communication Scenario %Name%:
\n
.foreach_item(Symbols,,, %ObjectType% == CommunicationMessageSymbol,
%compareCbMsgSymbols%)
 %Object.description%
.next(\n)
```

In this template, the first line is used to generate the title of the scenario using the name of the communication diagram %Name%. Then it creates a new line.

Then the macro for each item loops on each message symbol, verifies and sorts the message number, and outputs the message description using the syntax defined in previous section.

1. Right-click the Profile category and select Add Metaclasses.

   The Metaclass Selection dialog box is displayed.
2. Click the Modify Metaclass Filter tool and select Show All Metaclasses.
3. Select CommunicationDiagram in the list of metaclasses displayed in the PdOOM tab and click OK.

   The CommunicationDiagram category is displayed under Profile.
4. Right-click the CommunicationDiagram category and select **New** > **Template** .

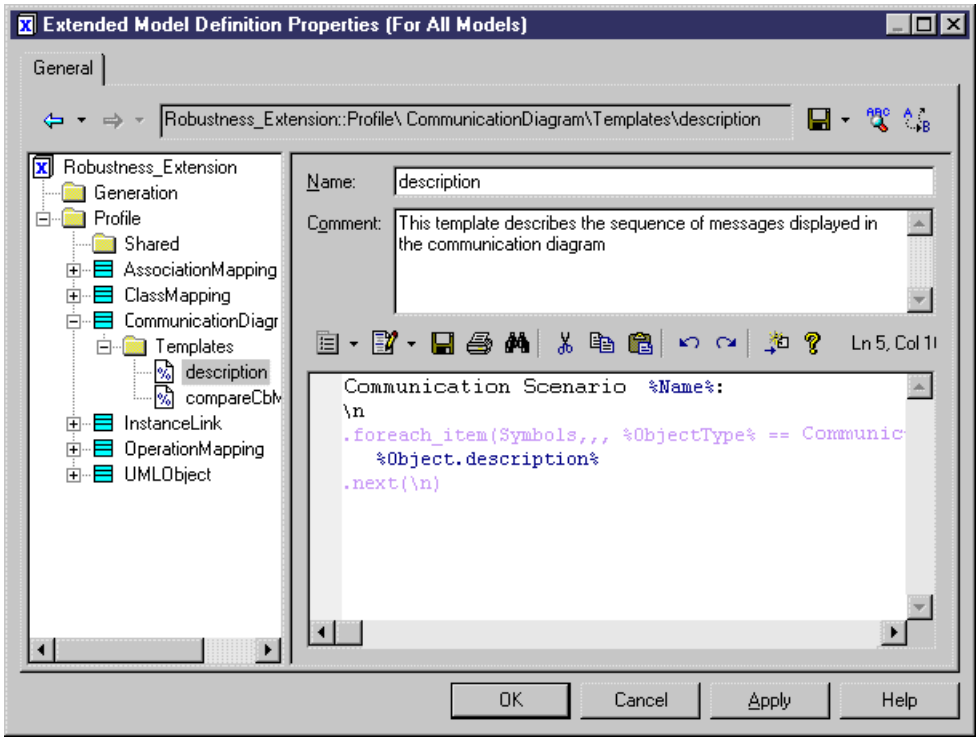
   A new template is created.
5. Type compareCbMsgSymbols in the Name box of the template
6. (optional) Type a comment in the Comment box of the template.
7. Type the following code in the text area:

```
.bool (%Item1.Object.SequenceNumber% >= %Item2.Object.SequenceNumber%)
```

8. Click Apply.
9. Right-click the CommunicationDiagram category and select **New** > **Template** .

   A new template is created.
10. Type description in the Name box of the template
11. (optional) Type a comment in the Comment box of the template.
12. Type the following code in the text area:

```
Communication Scenario %Name%:
\n
.foreach_item(Symbols,,, %ObjectType% == CommunicationMessageSymbol,
%compareCbMsgSymbols%)
 %Object.description%
.next(\n)
```




13. Click Apply.

### Defining a Generated File

You are going to define the generated file in order to list the messages of each communication diagram existing in your model. To do so, you have to define the generated file in the BasePackage metaclass. This metaclass is the common class for all packages and models, it owns objects, diagrams and other packages.

The generated file will contain the result of the evaluation of the template `description` defined on the CommunicationDiagram metaclass. The code of the generated file also contains a foreach_item macro in order to loop on the different communication diagrams of the model.

1. Right-click the Profile category and select Add Metaclasses.

   The Metaclass Selection dialog box is displayed.
2. Click the Modify Metaclass Filter tool, select Show Abstract Modeling Metaclasses, and click the PdCommon tab.
3. Select BasePackage in the list of metaclasses and click OK.

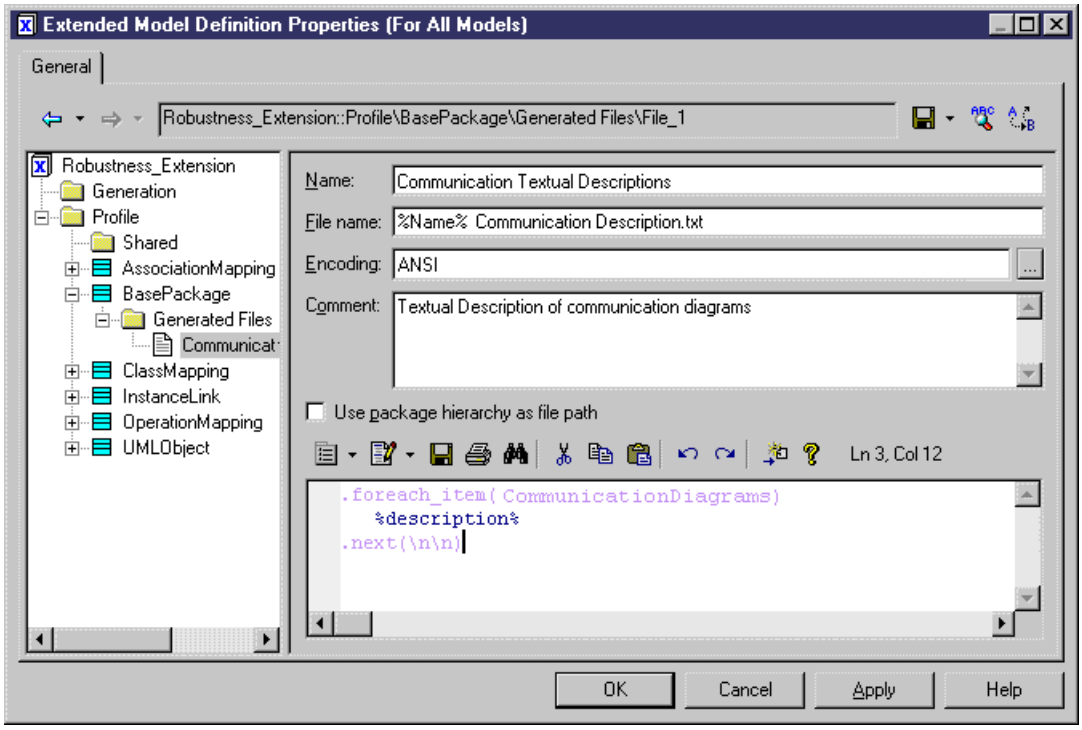
   The BasePackage category is displayed under Profile.
4. Right-click the BasePackage category and select **New** > **Generated File** .

   A new generated file is created.
5. Type Communications Textual Descriptions in the Name box.

   This name is used in the resource editor.
6. Type %Name% Communication Description.txt in the File Name box.

   This is the name of the file that will be generated. It will be a .txt file, and it will contain the name of the current model thanks to variable %Name%.
7. Keep the Encoding value to ANSI.
8. (optional) Type a comment in the Comment box.
9. Deselect the check box Use Package Hierarchy as file path because you do not need the hierarchy of files to be generated.
10. Type the following code in the text box:

```
.foreach_item(CommunicationDiagrams)
 %description%
.next(\n\n)
```




**11.** Click OK and accept to save the extended model definition.

**12.** Click OK to close the List of Extended Model Definitions.

## Preview the Textual Description of the Communication Diagram

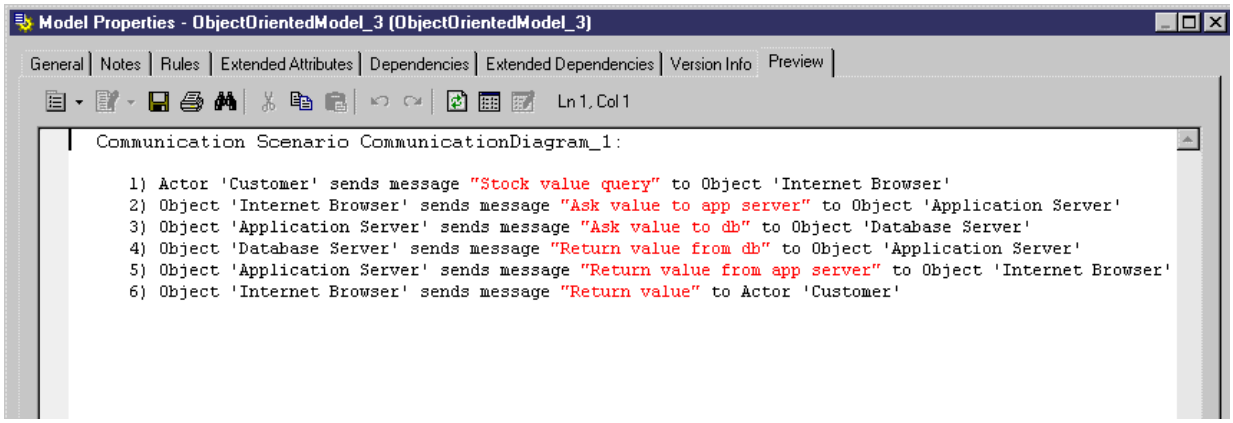
Since the generated file is defined in the metaclass BasePackage, you can preview the generated text in the Preview tab of the model property sheet. This also allows you to verify that the syntax of the templates and the output correspond to what you want to generate.

**1.** Right-click the communication diagram background and select Properties.

The model property sheet is displayed.

**2.** Click the Preview tab to display the Preview tab.

The Preview tab displays the content of the file to generate.




**3.** Click OK.

# Customizing Generation with GTL

The PowerDesigner Generation Template Language (GTL) is a template-based language, which is used to generate text for the metaclasses defined in the PowerDesigner metamodel, and on any extensions that are defined in the model profile.

Each template is associated with a given metaclass (for example, a CDM entity attribute, a PDM table, or an OOM operation). You can define as many templates as you want for each metaclass, and they will be available to all objects (instances) of the metaclass.

When you generate a model, PowerDesigner evaluates which metaclasses must have files generated, and creates a file for each instance of the metaclass, by applying the appropriate templates and resolving any variables.

GTL is object-oriented, and supports inheritance and polymorphism for reusability and maintainability. Macros provide generic programming structures for testing variables and iterating through collections, etc.

**Note:** To examine the set of templates used to generate code for operations in a Java OOM, open the Java object language in the resource editor and expand the Profile\Operation\Templates category.

A GTL template can contain text, macros, and variables, and can reference:

- metamodel attributes, such as the name of a class or data type of an attribute
- collections, such as the list of attributes of a class or columns of a table
- other elements of the model, such as environment variables

GTL templates can be either simple or complex

### Simple Templates

A simple template can contain text, variables, and conditional blocks, but cannot contain macros. For example:

```
%Visibility% %DataType% %Code%
```

When this template is evaluated, the three variables Visibility, DataType, and Code will be resolved to the values of these properties for the object.

### Complex Templates

A complex template can contain any element from a simple template, and also macros. For example:

```
.if (%isInner% == false) and ((%Visibility% == +)
                              or (%Visibility% == *))
    [%sourceHeader%\n\n]\
    [%definition%\n\n]
    .foreach_item(ChildDependencies)
        [%isSameFile%?%InfluentObject.definition%\n\n]
    .next
    [%sourceFooter%\n]
.endif
```

This template begins with an .if macro which tests the values of the isInner and Visibility properties. Several variables are enclosed in square brackets, which ensures that the text enclosed with them (in this case, new line characters) will not be generated if the variable evaluates to void. The .foreach_item macro loops over all the members of the ChildDependencies collection.

## Creating a Template and a Generated File

GTL templates are commonly used for generating files. If your template is going to be used in generation, it must be referenced in a generated file.

### To Create a Generated File:

Generated files are created in the Profile category in the resource editor.

In the resource editor, right click a metaclass in the Profile category, and select **New** > **Generated File** from the contextual menu.

### To Create a Template:

Templates are created in the Profile category in the resource editor.

In the resource editor, right click a metaclass in the Profile category, and select **New** > **Template** from the contextual menu.

**Note:** We encourage you to name your templates using camelCase, starting with a lowercase letter, in order to avoid clashes with property and collection names which, by convention use full CamelCase.

### To Reference a Template in a Generated File:

You use a template by referencing it in a generated file.

In the resource editor, select the appropriate generated file in the Profile category, and insert the name of the template between percent signs. For example:

```
%myTemplate%
```

## Accessing Object Properties

Object properties are treated as variables, and enclosed between percent signs, as follows:

```
%variable%
```

Example Template:

```
This file is generated for %Name%. It has the form of a %Color% %Shape.
```

Output:

```
This file is generated for MyObject. It has the form of a Red Triangle.
```

For more information see *Object members* on page 191.

## Formatting Output

To control the format of your output, insert format options between the percent signs before the variable as follows:

```
%.format:variable%
```

Example Template:

The following template reformats the Name variable to uppercase and encloses it in double-quotes.

```
This file is generated for %.UQ:Name%. It has the form of a %.L:Color% %.L:Shape.
```

Output:

```
This file is generated for "MYGADGET". It has the form of a red triangle.
```

For more information see *Formatting options* on page 193.

## Using Conditional Blocks

If you have text that you want to appear only if a variable resolves to a non-null value, you should place them together between square brackets.

Example Template:

```
[This line is generated if "Exist" is not null: %Exist%]
This line is generated even if "Exist" is null: %Exist%
```

Output (if Exist is null):

```
This line is generated even if "Exist" is null:
```

Output (if Exist is not null):

```
This line is generated if "Exist" is not null: Y
This line is generated even if "Exist" is null: Y
```

For more information see *Conditional blocks* on page 192.

## Accessing Collections of Sub-objects

Tables have multiple columns, classes have multiple attributes and operations. To iterate over such collections of associated objects, use a macro, such as .foreach_item.

Example:

```
%Name% contains the following widgets:
.foreach_item(Widgets)
    \n\t%Name% (%Color% %Shape%)
.next
```

Output:

```
MyObject contains the following widgets:
    Widget1 (Red Triangle)
    Widget1 (Yellow Square)
    Widget1 (Green Circle)
```

For more information see *Collection members* on page 191.

## Accessing Global Variables

You can insert information such as your user name and the current date, by accessing global variables.

Example template:

```
This file was generated by %CurrentUser% on %CurrentDate%.
```

Output:

```
This file was generated by jsmith on Tuesday, November 06, 2007 4:06:41 PM.
```

For more information see *Global variables* on page 192.

## GTL Variable Reference

Variables are qualified values enclosed in % characters and optionally preceded by formatting options. At evaluation-time, they are substituted by their corresponding value in the active translation scope.

A variable can be of the following types:

- An attribute of an object
- A member of a collection or an extended collection
- A template
- An environment variable

For example, the variable %Name% of an interface can be directly evaluated by a macro and replaced by the name of the interface in the generated file.

**Note:** Be careful when using variable names as they are case sensitive. The variable name must have the first letter with an upper case, as in %Code%.

### Variables Syntax

The following variables are shown with their possible syntaxes:

variable-block:

```
%[.formatting-options:]variable%
```

variable

```
[outer-scope.][variable-object.][object-scope.]object-member
[outer-scope.][variable-object.][collection-scope.]collection-member
[outer-scope.]local-variable
[outer-scope.]global-variable
```

object-member:

```
volatile-attribute
property
[target-code::]extended-attribute
[target-code::][metaclass-name::]template-name[(parameter-list)]
[*]+local-value[(parameter-list)]
```

object-member-object =

```
objecttype-property
oid-valued-object-member
this
```

collection-member

```
First
IsEmpty
Count
```

collection-member-object =

```
First
```

local-variable

```
local-object
[*]local-value
```

global-variable

```
global-object
global-value
$environment variable
```

variable-object

```
global-object
local-object
```

outer-scope

```
[outer-scope.]Outer
```

object-scope

```
[object-scope.]object-member-object
collection-scope.collection-member-object
```

collection-scope

```
[object-scope.]collection
[object-scope.]semi-colon-terminated-oid-valued object-member
```

For more information on extended collections, see *Extended Collections and Compositions (Profile)* on page 137.

## Object Members

An object member can be a standard property, an extended attribute, a template or a volatile attribute. There can be three types of standard property: boolean, string or object. The value of a standard property can be:

- 'true' or 'false' if it is of boolean type
- 'null' or object OID if it is of object type

The value of a template is the result of its translation (note that a template may be defined in terms of itself, that is to say recursively).

The value of an extended attribute may itself be a template, in which case it is translated. This allows for the definition of templates on a per object (instance) basis instead of a per metaclass basis.

To avoid name collisions when a template evaluation spans multiple targets, one may prefix both extended attributes and templates by their parent target code. For example: %Java::strictfp% or %C++::definition%

Template names may also be prefixed by their parent metaclass name. This allows for the invocation of an overridden template, actually bypassing the standard dynamic template resolution mechanism. For example : %Classifier::definition %

A parameter list can optionally be specified. Parameter values should not contain any % characters and should be separated by commas. Parameters are passed as local variables @1, @2, @3... defined in the translation scope of the template.

If the template MyTemplate is defined as:

```
Parameter1 = %@1%
Parameter2 = %@2%
```

Then the evaluation of %MyTemplate(MyParam1, MyParam2)% will yield:

```
Parameter1 = MyParam1
Parameter2 = MyParam2
```

## Collection Members

Each object can have one or more collections, which contain objects with which it interacts. For example, a table has collections of columns, indexes, business rules and so on.

Collections are represented in the PowerDesigner metamodel (see *Resource Files and the Public Metamodel* on page 1) by associations between objects, with roles named after the collections.

The available collection members are:

| Name | Type | Description |
|---|---|---|
| First | Object | Returns the first element of the collection |

| Name | Type | Description |
|---|---|---|
| IsEmpty | Boolean | Used to test whether a collection is empty or not. True if the collection is empty, false otherwise |
| Count | Integer | Number of elements in the collection |

Note: Count is particularly useful for defining criteria based on collection size, for example (Attributes.Count>=10).

## Conditional Blocks

Conditional blocks can be used to specify different templates based on the value of a variable. Two different forms are available:

The first form is similar to C and Java ternary expressions. The first template is evaluated, unless the value of the variable is false, null, or the null string, in which case, the second, optional, template, is evaluated:

[ *variable* ? *simple-template* [ : *simple-template* ] ]

The second form syntax is translated only if the value of the variable is not the null string:

[ *text variable text* ]

Example: an attribute declaration in Java:

```
%Visibility% %DataType% %Code% [= %InitialValue%]
```

## Global Variables

Global variables are available regardless of the current scope. A number of GTL-specific variables are defined as global as listed in the following table:

| Name | Type | Description |
|---|---|---|
| ActiveModel | Object | Active model |
| GenOptions | struct | Gives access to user-defined generation options |
| PreviewMode | boolean | True if in Preview mode, false if in File Generation model |
| CurrentDate | String | Current system date and time formatted using local settings |
| CurrentUser | String | Current user login |
| NewUUID | String | Returns a new UUID |

## Local Variables

You can define local variables with the .set_object and .set_value macros

For more information, see *.set_object and .set_value macros* on page 217. Local variables are only visible in the scope where they are defined and inside its inner scopes.

Volatile attributes may be defined through the .set_object and .set_value macros.

### If the Scope Is an Object Scope:

A volatile attribute is defined. This attribute will be available on the corresponding object regardless of the scope hierarchy. Volatile attributes shadow standard attributes. Once defined, they remain available until the end of the current generation process.

The "this" keyword returns an object scope and allows you to define volatile attributes on the object which is active in the current scope.

### If the Scope Is a Template Scope:

, a standard local variable is defined.

Examples:

```
.set_value(this.key, %Code%-%ObjectID%)
```

defines the key volatile attribute on the current object

```
eg. .set_object(this.baseClass, ChildGeneralizations.First.ParentObject)
```

defines the baseClass object-type volatile attribute on the current object.

### Dereferencing Operator

Variables defined through the set_object macro are referred to as local objects, whereas those defined with the set_value macro are called local values. The * dereferencing operator may be applied to local values.

The * operator allows for the evaluation of the variable whose name is the value of the specified local variable.

```
%[.formatting-options:]*local-variable%
```

For example, the following code:

```
.set_value(i, Code)
%*i%
```

Is equivalent to:

```
%Code%
```

## Formatting Options

You can change the formatting of variables by embedding formatting options in variable syntax as follows:

```
%.format:variable%
```

The variable formatting options are the following:

| Format option | Description |
|---|---|
| *n* | Extracts the first n characters. Blanks or zeros are added to the left to fill the width and justify the output to the right |
| *-n* | Extracts the last n characters. Blanks or zeros are added to the right to fill the width and justify the output to the left |
| L | Converts to lowercase characters |
| U | Converts to uppercase characters |
| c | Upper-case first letter and lower-case next letters |
| A | Removes indentation and aligns text on the left border |
| D | Returns the human-readable value of an attribute used in the PowerDesigner interface when this value differs from the internal representation. In the following example, the value of the Visibility attribute is stored internally as "+", but is displayed as "public" in the property sheet: %Visibility% = + %.D:Visibility% = public |
| F | Applies case conversion to the first character only. Used with L or U. |

| Format option | Description |
|---|---|
| T | Leading and trailing white space trimmed from the variable |
| q | Surrounds the variable with single quotes |
| Q | Surrounds the variable with double quotes |
| X | Escapes XML forbidden characters |
| E | [deprecated – use the ! power evaluation operator instead, see *Operators* on page 194] |

You can combine format codes. For example, %.U8:CHILD% formats the first eight characters of the code of the CHILD table in uppercase letters.

## Operators

The following operators are supported in GTL:

| Symbol | Description |
|---|---|
| * | Dereferencing operator - The syntax [*]+*local-value*[(*parameter-list*)] returns the object member defined by the evaluation of [*]+ *local-value*. If the given object member happens to be a template, a parameter list may be specified. Applying the star operator corresponds to a double evaluation (the * operator acts as a dereferencing operator).<br>If a local variable is defined as: `.set_value(C, Code)`, then `%C%` will return "Code" and `%*C%` will return the result of the evaluation of %Code%. In other words, `%*C%` can be thought of as `%(%C%)%` (the latter syntax being invalid). |
| ! | Power evaluation operator - Evaluates the results of the evaluation of the variable as a template. For example, you define a comment containing a variable like `%Code%.` When you use the ! operator in `%!Comment%`, the actual value of %Code% is substituted for the variable block. Without the ! operator, the variable remains unevaluated.<br>The ! operator may be applied any number of times. For example:<br>`%!!template%`<br>This outputs the results of the evaluation of the evaluation of the evaluation of template 'template' |
| ? | The ? operator is used to test the existence of a template, a local variable, a volatile or an extended attribute. It returns "true" if the variable exists, "false" otherwise.<br>For example, if custname is defined whereas custid is not, then the template:<br>`.set_value(foo, tt)`<br>`%custname?%`<br>`%custid?%`<br>outputs:<br>`true`<br>`false` |
| + | The + operator is used to test if an object property is visible in the interface.<br>For example, you could test if the Type box is displayed in the General tab of a database property sheet in an ILM, and thus that the Replication Server or MobiLink XEM is attached to the model.<br>The %Database.Type+% template will output false if no XEM is attached to the model. |

## Translation Scope

The translation scope defines the context for evaluating a template, by determining the object to which the template is applied. The scope can change during the translation of a template, but only one object is active at any given time.

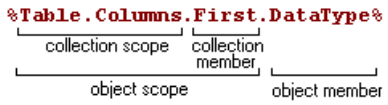
The initial scope is always the metaclass on which the template is defined. All metamodel attributes and collections defined on the active object metaclass and its parents are visible, as well as the corresponding extended attributes and templates.

You can change scope using the '.' (dot) character, which behaves like the Java indirection operator, with the right-hand side corresponding to a member of the object referred to by the left-hand side.

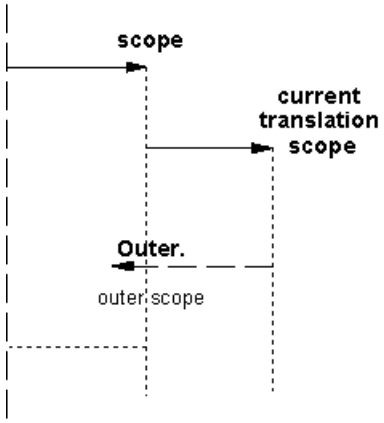
The following types of scope are available:

- Object scope - To access the members of an object that is not currently active, specify its object scope.
- Collection scope - To gain access to the members of a collection, one should specify a collection scope. For more information on object collections, see *Resource Files and the Public Metamodel* on page 1.

  For example:



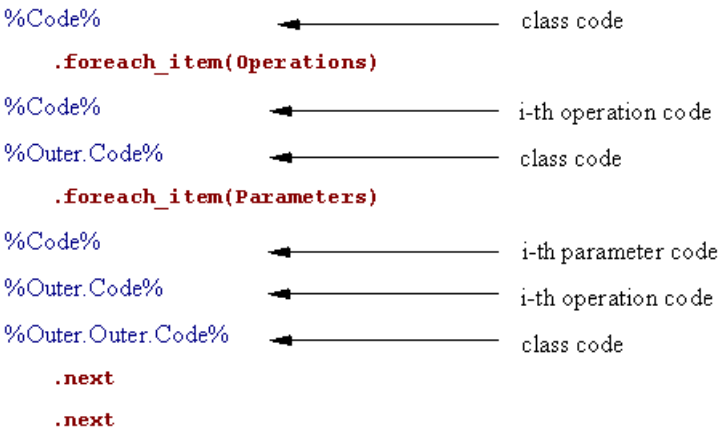

- Outer scope - accessed using the Outer keyword. The following rules apply:
  - When a scope is created, the old scope becomes the outer scope.
  - When a scope is exited, the outer scope is restored as the current translation scope




New scopes may be created during evaluation of a template that forces the active object to change. For example, foreach_item macro (see *.foreach_item macro* on page 210) that allows for iteration on collections defines a new scope, and the foreach_line macro (see *.foreach_line macro* on page 211). The outer scope is restored when leaving the block.

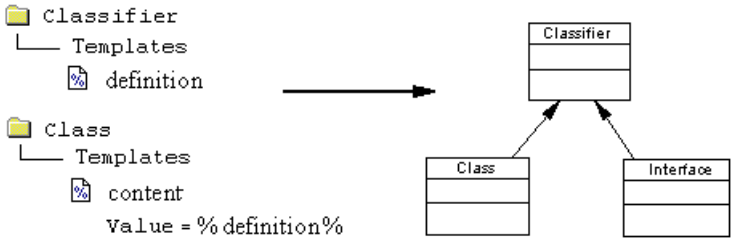
Nested scopes form a hierarchy that can be viewed as a tree, the top level scope being the root.

The following example shows the scope mechanism using a Class template:




## Inheritance and Polymorphism

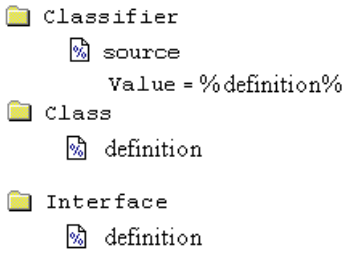
Templates are defined with respect to a given metaclass and are inherited by and available to the children of the metaclass. In the following example, the definition template defined on the parent metaclass is available to, and used in the evaluation of the content template on the child metaclass.




GTL supports the following OO concepts as part of inheritance:

- *Polymorphism* - Templates are dynamically bound; the choice of the template to be evaluated is made at translation-time. Polymorphism allows template code defined on a classifier to use templates defined on its children (class, interface), the template being used does not have to be defined on the parent metaclass. Coupled with inheritance, this feature helps you share template code.

  In the following example, the content of %definition% depends on whether a class or an interface is being processed:

- *Template overriding* - A template defined on a given metaclass may be redefined on a child class. The template defined on the child overrides the template defined on the parent for objects of the child metaclass. You can view the overridden parent using the Go to super-definition command in the child class contextual menu, and specify the use of the parent template with the "::" qualifying operator. For example:

```
Profile
   Classifier
      Templates
         isAbstract
               Value = false
   Class
      Templates
         isAbstract
               Value = true
```

The same template name "isAbstract" is used in two different categories: Classifier and Class. "false" is the original value that has just been overridden by the new "true" value. You retrieve the original value back by using the following syntax: <metaclassName::template>, in this case:

```
%isAbstract%
%Classifier::isAbstract%
```

- *Template overloading* - You can overload your template definitions and test for different conditions. Templates can also be defined under criteria or stereotypes (see *Criteria (Profile)* on page 128 and *Stereotypes (Profile)* on page 124), and the corresponding conditions are combined. At translation-time, each condition is evaluated and the appropriate template (or, in the event of no match, the default template) is applied. For example:

```
full-template-name = {syntax1} <template-name>                          |
                     {syntax2} <template-name>'<<' stereotype '>>'      |
                     {syntax3} <template-name>'<' <simple-condition> '>'

template-name      = <text>
```

## Shortcut Translation

Shortcuts are dereferenced during translation: the scope of the target object replaces the scope of the shortcut.

For example, the following generated file defined in the package metaclass provides the list of classes in the package. If a class shortcut is found, the code of its target object followed by (Shortcut) is generated, followed by the parent object ID and the shortcut ID which clearly shows that the scope of the shortcut is replaced by the scope of the shortcut target object:

```
.foreach_item(Classes)
 .if (%IsShortcut%)
%Code% (Shortcut)
oid = %ObjectID%
shortcut oid = %Shortcut.ObjectID%
 .else
%Code%
%Shortcut%
 .endif
.next(\n)
```

This is the opposite behavior as in VB Script where shortcut translation retrieves the shortcut itself.

If you want the shortcut itself to be generated instead of the target object, you can use the %Shortcut% variable.

### External Shortcut

If the target model of an external shortcut is not open, a confirmation dialog box is displayed to let you open the target model. You can use the set_interactive_mode macro to change this behavior. This macro allows you to decide if the GTL execution must interact with the user or not.

For more information on the set_interactive_mode macro, see *.set_interactive_mode macro* on page 217.

## Escape Sequences

Escape sequences are specific characters sequences used for layout of the generated file output.

The following escape sequences can be used inside templates:

| Escape sequence | Description |
|---|---|
| \n | New line character, creates a new line |
| \t | Tab character, creates a tab |
| \\ | Creates a backslash |
| \ at end of line | Creates a continuation character (ignores the new line) |
| . at beginning of line | Ignores the line |
| .. at beginning of line | Creates a dot character (to generate a macro) |
| %% | Creates a percent character |

For more information on escape sequences, see *Using new lines in head and tail string* on page 199.

## Sharing Templates

In the GTL mechanism you can share conditions, templates and sub-templates to ease object language maintenance and readability.

### Sharing Conditions

A template can contain a condition expression. You can also create templates to share long and fastidious condition expressions:

| Template name | Template value |
|---|---|
| %ConditionVariable% | .bool (condition) |

Instead of repeating the condition in other templates, you simply use %ConditionVariable% in the conditional macro:

```
.if (%ConditionVariable%)
```

### Example

The template %isInner% contains a condition that returns true if the classifier is inner to another classifier.

```
.bool (%ContainerClassifier%!=null)
```

This template is used in the %QualifiedCode% template used to define the qualified code of the classifier:

```
.if (%isInner%)
    %ContainerClassifier.QualifiedCode%::%Code%
.else
    %Code%
.endif
```

### Using Recursive Templates

A recursive template is a template that is defined in terms of itself.

### Example

Consider three classes X, Y, and Z. X is inner to Y, and Y is inner to Z.

The variable %topContainerCode% is defined to retrieve the value of the parent container of a class.

The value of the template is the following:

```
.if (%isInner%)
    %ContainerClassifier.topContainerCode%
.else
    %Code%
.endif
```

If the class is inner to another class, %topContainerCode% is applied to the container class of the current class (%ContainerClassifier.topContainerCode%).

If the class is not an inner class, the code of the class is generated.

## Using New Lines in Head and Tail String

The head and tail string are only generated when necessary. If no code is generated, the head and tail strings do not appear. This can be useful when controlling new lines.

### Example

You want to generate the name of a class and its attributes under the following format (one empty line between attributes and class):

```
Attribute 1  attr1
Attribute 2  attr2

Class
```

You can insert the separator "\n" after the .foreach statement to make sure each attribute is displayed in a separate line. You can also add "\n\n " after the .endfor statement to insert an empty line after the attribute list and before the word "Class".

```
.foreach (Attribute) ("\n")
Attribute %Code%
.endfor ("\n\n")
Class
```

### Additional Example

Consider a class named *Nurse*, with a class code Nurse, and two attributes:

| Attribute name | Data type | Initial value |
|---|---|---|
| NurseName | String | __ |
| NurseGender | Char | 'F' |

The following templates are given as examples, together with the text generated for each of them, and a description of each output:

### Template 1

```
class "%Code%" {
 // Attributes
 .foreach_item(Attributes)
 %DataType% %Code%
  .if (%InitialValue%)
 = %InitialValue%
  .endif
 .next
 // Operations
 .foreach_item(Operations)
 %ReturnType% %Code%(...)
 .next
}
```

### Text Generated 1

```
class "Nurse" {
```

```
 // Attributes String nurseName char nurseGender = 'F' // Operations}
```

### Description 1

Below the class code, the code is generated on one line. It is an example of a block macro (.if, .endif macro).

### Template 2 (new Line)

```
class "%Code%" {
 // Attributes
 .foreach_item(Attributes)
 %DataType% %Code%
  .if (%InitialValue%)
 = %InitialValue%
  .endif
 .next(\n)
 // Operations
 .foreach_item(Operations)
 %ReturnType% %Code%(...)
 .next(\n)
}
```

### Text Generated 2

```
class "Nurse" {
```

```
 // Attributes String nurseName
```

```
 char nurseGender = 'F' // Operations}
```

### Description 2

String nurseName and char nurseGender are on two lines

In Template 1, String nurseName and char nurseGender were on the same line, whereas in Template 2, the addition of the \n at .next(\n) puts String nurseName and char nurseGender on two different lines.

In addition, // Operations is displayed in the output even if there is no operation (see Description 3).

### Template 3 (blank Space)

```
class "%Code%" {
 .foreach_item(Attributes, // Attributes\n,\n)
 %DataType% %Code%
  .if (%InitialValue%)
 = %InitialValue%
  .endif
 .next(\n)
 .foreach_item(Operations, // Operations\n,\n)
 %ReturnType% %Code%(...)
 .next(\n)
}
```

### Text Generated 3

```
class "Nurse" {// Attributes
```

```
 String nurseName
```

```
 char nurseGender = 'F'
```

```
}
```

### Description 3

The blank space between *.foreach_item(Attributes,* and *// Attributes\n,\n)* is not generated, as shown in the output: class "Nurse" {// Attributes instead of .... { // Attributes

// Operations is not displayed in the output because it is positioned in the .foreach_item macro. It is positioned in the head of the macro for this purpose.

### Template 4 (blank Space)

```
class "%Code%" {\n
 .foreach_item(Attributes," // Attributes\n",\n)
 %DataType% %Code%[ = %InitialValue%]
 .next(\n)
 .foreach_item(Operations," // Operations\n",\n)
 %ReturnType% %Code%(...)
 .next(\n)
}
```

### Text Generated 4

```
class "Nurse" {

 // Attributes

 String nurseName

 char nurseGender = 'F'

}
```

### Description 4

The double quote characters ("") in " // Attributes\n" allows you to insert a blank space as shown in the output: // Attributes

**Note:** The newline immediately preceding a macro is ignored as well as the one immediately following it, as in the following example:

Jack .set_value(v, John) Paul yields: JackPaul

instead of: Jack Paul

## Using Parameter Passing

You can pass in, out or in/out parameters to a template through local variables by taking advantage of nested translation scopes. You can access parameters with the %@<number>% variable.

### Example

Class templates:

### Template 1

```
<show> template
<<<
Class "%Code%" attributes :
// Public
%publicAttributes%

// Protected
%protectedAttributes%

// Private
%privateAttributes%
>>>
```

**Template 2**

```
<publicAttributes> template
<<<
.foreach_item(Attributes)
 .if (%Visibility% == +)
 %DataType %Code%
 .endif
.next(\n)
>>>
```

**Template 3**

```
<protectedAttributes> template
<<<
.foreach_item(Attributes)
 .if (%Visibility% == #)
 %DataType %Code%
 .endif
.next(\n)
>>>
```

**Template 4**

```
<privateAttributes> template
<<<
.foreach_item(Attributes)
 .if (%Visibility% == -)
 %DataType %Code%
 .endif
.next(\n)
>>>
```

To give you more readability and to enhance code reusability, these four templates can be written in just two templates by using parameters:

**First Template**

```
<show> template
<<<
Class "%Code%" attributes :
// Public
%attributes(+)%

// Protected
%attributes(#)%

// Private
%attributes(-)%
>>>
```

**Second Template**

```
<attributes> template
<<<
.foreach_item(Attributes)
 .if (%Visibility% == %@1%)
 %DataType %Code%
 .endif
.next(\n)
>>>
```

### Description

The first parameter in this example %attributes(+, or #, or -)% can be accessed using the variable %@1%, the second parameter when it exists, is accessed using the %@2% variable, etc ...

## Error Messages

Error messages stop the generation of the file in which errors have been found, these errors are displayed in the Preview tab of the corresponding object property sheet.

Error messages have the following format:

```
target::catg-path full-template-name(line-number)
active-object-metaclass active-object-code):
    error-type error-message
```

The following types of errors can be encountered:

- Syntax errors
- Translation errors

### Syntax Errors

You may encounter the following syntax errors:

| Syntax error message | Description and correction |
|---|---|
| condition parsing error | Syntax error in a boolean expression |
| expecting .endif | Add a .endif |
| .else with no matching .if | Add a .if to the .else |
| .endif with no matching .if | Add a .if to the .endif |
| expecting .next | Add a .next |
| expecting .end%s | Add a .end%s (for example, .endunique, .endreplace, ...) |
| .end%s with no matching .%s | Add a .*macro* to the .end*macro* |
| .next with no matching .foreach | Add a .foreach to the .next |
| missing or mismatched parentheses | Correct any mismatched braces |
| unexpected parameters: *extra-params* | Remove unnecessary parameters |
| unknown macro | The macro is not valid |
| .execute_command incorrect syntax | The correct syntax is displayed in the Preview tab, or in the Output window. It should be: .execute_command(executable [,*arguments*[, {cmd_ShellExecute\|cmd_PipeOutput}]]) |
| Change_dir incorrect syntax | The syntax should be: .change_dir(*path*) |
| convert_name incorrect syntax | The syntax should be: .convert_name(*name*) |
| convert_code incorrect syntax | The syntax should be: .convert_code(*code*) |
| set_object incorrect syntax | The syntax should be: .set_object(*local-var-name*[,[*scope*.] *object-scope* [,{new\|update}]]) |
| set_value incorrect syntax | The syntax should be: .set_value(*local-var-name*,*simple-template*[, {new\|update}]) |

| Syntax error message | Description and correction |
|---|---|
| execute_vbscript incorrect syntax | The syntax should be: .execute_vbscript(*script-file* [,*script-input_params*]) |

### Translation Errors

Translation errors are evaluation errors on a variable when evaluating a template.

You may encounter the following translation errors:

| Translation error message | Description and correction |
|---|---|
| unresolved collection: *collection* | Unknown collection |
| unresolved member: *member* | Unknown member |
| no outer scope | Invalid use of Outer keyword |
| null object | Trying to access a null object member |
| expecting object variable: *object* | Occurs when using string instead of object |
| VBScript execution error | VB script error |
| Deadlock detected | Deadlock due to an infinite loop |

## GTL Macro Reference

Macros can be used to express template logic, and to loop on object collections. Each macro keyword must be preceded by a . (dot) character and has to be the first non blank character of a line. Make sure you also respect the macro syntax in terms of line breaks.

You define a macro inside a template, or a command entry.

There are three types of macros:

- *Simple macros* are single line macros.
- *Block macros* consist of a begin and an end keyword delimiting a block to which the macro is applied. They have the following structure:

```
.macro-name [(parameters)]
    block-input
.endmacro-name [(tail)]
```

- *Loop macros* are used for iteration. At each iteration, a new scope is created. The template specified inside the block is translated successively with respect to the iteration scope.

```
.foreach_macro-name [(parameters[,head[,tail]])]
    complex-template
.next[(separator)]
```

**Note:** Macro parameters may be delimited by double quotes. The delimiters are required whenever the parameter value includes commas, braces, leading or trailing blanks. The escape sequence for double quotes inside a parameter value is \".

The following macros are available:

- *Conditional and loop / iterative macros*:
  - *.if macro* on page 213

- *.foreach_item macro* on page 210 – iterates on object collections
- *.foreach_line macro* on page 211 – iterates on lines
- *.foreach_part macro* on page 212 – iterates on parts
- *.break macro* on page 206 – breaks the loop

- *Assignment macros* - define a local variable of object or value type as well as volatile attributes:
  - *.set_object and .set_value macros* on page 217
  - *.unset macro* on page 218
- *Output and error reporting macros*:
  - *.log macro* on page 214
  - *.error and .warning macros* on page 209
- *Command macros* - only available in the context of the execution of a generic command:
  - *.vbscript macro* on page 218 - embed VB script code inside a template
  - *.execute_vbscript macro* on page 210 - launch vbscripts
  - *.execute_command macro* on page 209 - launch executables
  - *.abort_command macro* on page 205 - stop command execution
  - *.change_dir macro* on page 206 - changing a directory
  - *.create_path macro* on page 208 - creating a specified path
- *Formatting macros*:
  - *.lowercase and .uppercase macros* on page 214
  - *.convert_code and .convert_code macros* on page 207 – converts codes into names
- *String manipulation macros*:
  - *.replace macro* on page 216
  - *.delete macro* on page 208
  - *.unique macro* on page 218
  - *.block macro* on page 205 - adds a header and a footer to a text block
- *Miscellaneous macros*:
  - *.comment and .// macro* on page 207 - inserts a comment in a template
  - *.collection macro* on page 207 - returns a collection of objects based on the specified scope and condition
  - *.object macro* on page 215 - returns an object based on the specified scope and condition
  - *.bool macro* on page 206 - evaluates a condition
  - *.set_interactive_mode macro* on page 217 – defines whether the GTL execution must interact with the user

## .abort_command Macro

This macro stops command execution altogether. It is available to execute generation commands only, and may be used in addition to standard GTL macros when defining commands.

Example:

```
.if %_JAVAC%
  .execute (%_JAVAC%,%FileName%)
.else
  .abort_command
.endif
```

## .block Macro

The .block macro is used to add a header and/or a footer to its content when it is not empty.

```
.block [(head)]
    block-input
.endblock[(tail)]
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *head* | [optional] Generated before output, if there is one.<br>Type: Simple template |
| *block-input* | Parameter used to input text<br>Type: Complex template |
| *tail* | [optional] Appended to the output, if there is one<br>Type: Text |

The output is a concatenation of *head*, the evaluation of the *block-input* and *tail*.

Example:

```
.block (<b>)
The current text is in bold
.endblock (</b>)
```

## .bool Macro

This macro returns 'true' or 'false' depending on the value of the condition specified.

```
.bool (condition)
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *condition* | Condition to be evaluated<br>Type: Condition |

Example:

```
.bool(%.3:Code%= =ejb)
```

## .break Macro

This macro may be used to break out of .foreach loops.

```
.break
```

Example:

```
.set_value(_hasMain, false, new)
.foreach_item(Operations)
 .if (%Code% == main)
  .set_value(_hasMain, true)
  .break
 .endif
.next
%_hasMain%
```

## .change_dir Macro

This macro changes the current directory. It is available to execute generation commands only, and may be used in addition to standard GTL macros when defining commands.

```
.change_dir (path)
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *path* | New current directory<br>Type: Simple template (escape sequences ignored) |

Example:

```
.change_dir(C:\temp)
```

## .collection Macro

This macro returns a collection of objects based on the specified scope and condition. Collections are represented as the concatenation of semi-colon terminated OIDs.

```
.collection (collection-scope [,filter])
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *collection-scope* | Scope over which to iterate.<br>Type: <simple-template> returning a collection scope |
| *filter* | [optional] Filter condition<br>Type : condition |

Example:

The following macro returns a subset of the attributes defined on the current classifier whose code starts with a letter between a and e included.

```
.object(Attributes, (%.1:Code% >= a) and (%.1:Code% <= e))
```

Result:

```
C3ADA38A-994C-4E15-91B2-08A6121A514C;58CE2951-7782-49BB-
B1BB-55380F63A8C9;F522C0AE-4080-41C2-83A6-2A2803336560;
```

## .comment and .// Macro

These macros can be used to insert comments in a template. Lines starting with .// or .comment are ignored during generation.

Example:

```
.// This is a comment
.comment This is also a comment
```

## .convert_code and .convert_name Macros

These macros use conversion tables to convert from the object code to name or vice versa. When no occurrence is found in the table, the input code or name is returned.

Use the following syntax to convert a code to a name:

```
.convert_code (expression [separator-character [,pattern-separator]])
```

Use the following syntax to convert a name to a code:

```
.convert_name (expression [separator-character [,pattern-separator]])
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *expression* | The code to be converted in the corresponding conversion table.<br>Type: Simple template |
| *separator-character* | [optional] Character generated each time a separator declared in *pattern-separator* is found in the code. For example, " " (space).<br>Type: Text |
| *pattern-separator* | [optional] Declaration of the different separators likely to exist in a code. These separators will be replaced by the <separator character>. You can declare several separators, for example "_ " and "tab"<br>Type: Text |

You can use these macros together with a user-defined conversion table that you select in the Conversion Table list. To do so, open the Model Options dialog box, select the required object in the Naming Convention folder and click the Code To Name tab.

For more information about naming conventions, see "Naming conventions" in the Models chapter of the *Core Features Guide*.

**Note**

You can also use these macros outside the naming conventions context provided the conversion table is the table of the current object of the script. Here is an example of a macro that can be added from the Profile\Column folder in a new Generated Files entry:

```
.foreach_item(Columns)
 %Name%,
 .foreach_part(%Name%)
  .convert_name(%CurrentPart%)
 .next("_")
.next(\n)
```

For more information on these macros, see ".convert_name & .convert_code macros" in the Models chapter of the *Core Features Guide*.

## .create_path Macro

This macro creates a specified path if it does not exist.

```
.create_path (path)
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *path* | Path to be created<br>Type: Simple template (escape sequences ignored) |

Example:

```
.create_path(C:\temp)
```

## .delete Macro

This macro deletes all instances of the string *del-string* from *delete-block-input*.

```
.delete (del-string)
    block-input
.enddelete
```

This macro is particularly useful when you work with naming conventions (see "Naming conventions" in the Models chapter of the *Core Features Guide*).

The following parameters are available:

| Parameter | Description |
|---|---|
| *del-string* | String to be deleted in the input block<br>Type: Text |
| *delete-block-input* | Parameter used to input text<br>Type: Complex template |

Example:

In the following example, GetCustomerName is converted to CustomerName:

```
.delete( get )
    GetCustomerName
.enddelete
```

In the following example, the variable %Code% is m_myMember and is converted to myMember:

```
.delete(m_)
    %Code%
.enddelete
```

## .error and .warning Macros

These macros are used to output errors and warnings during translation. Errors stop generation, while warnings are purely informational and can be triggered when an inconsistency is detected while applying the template on a particular object. The messages are displayed in both the object Preview tab and the Output window.

Use the following syntax to insert an error message:

```
.error message
```

Use the following syntax to insert a warning message:

```
.warning message
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *message* | Error message<br>Type: Simple template |

Example:

```
.error no initial value supplied for attribute %Code% of class %Parent.Code%
```

## .execute_command Macro

This macro is used to launch executables as separate processes. It is available to execute generation commands only, and may be used in addition to standard GTL macros when defining commands.

```
.execute_command (cmd [,args [,mode]])
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *cmd* | Executable path<br>Type: Simple template (escape sequences ignored) |
| *args* | [optional] Arguments for the executable<br>Type: Simple template (escape sequences ignored) |
| *mode* | [optional] You can choose one of the following:<br>• cmd_ShellExecute - runs as an independent process<br>• cmd_PipeOutput - blocks until completion, and shows the executable output in the output window |

Note that if an .execute_command fails for any given reason (executables not found, or output sent to stderr), the command execution is stopped.

Example:

```
.execute_command(notepad, file1.txt, cmd_ShellExecute)
```

## .execute_vbscript Macro

This macro is used to execute a VB script specified in a separate file.

```
.execute_vbscript (vbs-file [,script-parameter])
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *vbs-file* | VB script file path<br>Type: Simple template (escape sequences ignored) |
| *script-parameter* | [optional] Passed to the script through the ScriptInputParameters global property.<br>Type: Simple template |

The output is the ScriptResult global property value.

Example:

```
.execute_vbscript(C:\samples\vbs\login.vbs, %username%)
```

Note: the active object of the current translation scope can be accessed through the ActiveSelection collection as ActiveSelection.Item(0).

For more information on ActiveSelection, see *Global Properties* on page 239.

## .foreach_item Macro

This macro is used for iterating on object collections:

```
.foreach_item (collection [,head [,tail [,condition [,comparison]]]])
    complex-template
.next [(separator)]
```

The template specified inside the block is applied to all objects contained in the specified collection.

If a comparison is specified, items in the collection are pre-sorted according to the corresponding rule before being iterated upon.

The following parameters are available:

| Parameter | Description |
|---|---|
| *collection* | Collection over which iteration is performed<br>Type: Simple template |
| *head* | [optional] Generated before output, if there is one<br>Type: Text |
| *tail* | [optional] Appended to the output, if there is one<br>Type: Text |
| *condition* | [optional] If specified, only objects satisfying the given condition are considered during the iteration<br>Type: Simple condition |
| *comparison* | [optional] evaluated in a scope where two local objects respectively named 'Item1' and 'Item2' are defined. These correspond to items in the collection. <comparison> should evaluate to true if Item1 is to be placed after Item2 in the iteration<br>Type: Simple condition |
| *complex-template* | Template to apply to each item.<br>Type: Complex template |
| *separator* | [optional] Generated between non empty evaluations of <complex-template><br>Type: Text |

**Note:** Macro parameters may be delimited by double quotes. The delimiters are required whenever the parameter value includes commas, braces, leading or trailing blanks. The escape sequence for double quotes inside a parameter value is `\"`.

Example:

| Attribute | Data type | Initial value |
|---|---|---|
| cust_name | String | __ |
| cust_foreign | Boolean | false |

```
.foreach_item(Attributes,,,,%Item1.Code% >= %Item2.Code%))
    Attribute %Code%[ = %InitialValue%];
.next(\n)
```

The result is:

Attribute cust_foreign = false

Attribute cust_name;

### Note

The four commas after (Attributes,,,, means that all parameters (head, tail, condition and comparison) are skipped.

## .foreach_line Macro

This macro is a simple macro that iterates on the lines of the input template specified as the first argument to the macro. The template specified inside the block is translated for each line of the input. This macro creates a new scope with the local variable CurrentLine. This one is defined inside the block to be the i-th line of the input template at iteration i.

```
.foreach_line (input [,head [,tail]])
    complex-template
.next [(separator)]
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *input* | Input text over which iteration is performed<br>Type: Simple template |
| *head* | [optional] Generated before output, if there is one<br>Type: Text |
| *tail* | [optional] Appended to the output, if there is one<br>Type: Text |
| *complex-template* | Template to apply to each line.<br>Type: Complex template |
| *separator* | [optional] Generated between non empty evaluations of *complex-template*<br>Type: Text |

Example:

```
.foreach_line(%Comment%)
// %CurrentLine%
.next(\n)
```

## .foreach_part Macro

This macro iterates on the part of the input template specified as the first argument to the macro. The template specified inside the block is translated for each part of the input, with parts delimited by a separator pattern.

```
.foreach_part (input [,"separator-pattern" [,head [,tail]]])
    simple-template
.next[(separator)]
```

This macro creates a new scope wherein the local variable CurrentPart is defined to be the i-th part of the input template at iteration i. The Separator local variable contains the following separator.

This macro is particularly useful when you work with naming conventions.

For more information about naming conventions, see "Naming conventions" in the Models chapter of the *Core Features Guide.*

The following parameters are available:

| Parameter | Description |
|---|---|
| *input* | Input text over which iteration is performed<br>Type: Simple template |

| Parameter | Description |
|---|---|
| *separator-pattern* | Char and word separators<br>• Any character specified in the pattern can be used as separator<br>• [<c1> - <c2>] specifies a character within the range defined between both characters <c1> and <c2><br>For example, the pattern " -_,[A-Z]" specifies that each part can be separated by a space, a dash, an underscore, a comma or a character between A and Z (in capital letter).<br>By default, the <separator-pattern> is initialized with the pattern (). If the specified pattern is empty, the pattern is initialized using the default value.<br>A separator <separator> can be concatenated between each part. <head> and <tail> expressions can be added respectively at the bottom or at the end of the generated expression.<br>There are two kinds of separator:<br>• Char separator - for each char separator, the separator specified in the next statement of the macro is returned (even for consecutive separators)<br>• Word separator - they are specified as interval, for example [A-Z] specifies that all capital letters are separator. For a word separator, no separator (specified in next statement) is returned<br>Default: " -_,\t"<br>Type: Text |
| *head* | [optional] Generated before output, if there is one<br>Type: Text |
| *tail* | [optional] Appended to the output, if there is one<br>Type: Text |
| *simple-template* | Template to apply to each part.<br>Type: Complex template |
| *separator* | [optional] Generated between non empty evaluations of *complex-template*<br>Type: Text |

Examples:

Convert a name into a class code (Java naming convention). In the following example, the variable *%Name%* is equal to Employee shareholder, and it is converted to EmployeeShareholder:

```
.foreach_part (%Name%, " _-'")
    %.FU:CurrentPart%
.next
```

Convert a name into a class attribute code (Java naming convention). In the following example, the variable *%Name%* is equal to Employee shareholder, and it is converted to EmployeeShareholder:

```
.set_value(_First, true, new)
.foreach_part(%Name%,"' _-'")
    .if (%_First%)
        %.L:CurrentPart%
        .set_value(_First, false, update)
    .else
        %.FU:CurrentPart%
    .endif
.next
```

## .if Macro

This macro is used for conditional generation, it has the following syntax:

```
.if[not] condition
        complex-template
    [(.elsif[not] condition
        complex-template)*]
    [.else
        complex-template]
.endif [(tail)]
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *condition* | The condition to evaluate, in the form:<br>`variable [operator comparison]`<br>Where *operator* may be any of ==, =, <=, >=, <, or >. If both operands are integers, the <, >, >=, and <= operators perform integer comparisons; otherwise they perform a string comparison that takes into account embedded numbers (example: Class_10 is greater than Class_2).<br>Where *comparison* may be any of:<br>• A simple template<br>• "*text*"<br>• true<br>• false<br>• null<br>• notnull<br>If no operator and condition are specified, the condition evaluates to true unless the value of the variable is false, null or the null string.<br>You can chain conditions together using the and or or logical operators.<br>Type: Simple template |
| *complex-template* | The template to apply is the condition is true.<br>Type: Complex template |
| *tail* | Appended to the output, if there is one<br>Type: Text |

## .log Macro

This macro logs a message to the Generation tab of the Output window, located in the lower part of the main window. It is available to execute generation commands only, and may be used in addition to standard GTL macros when defining commands.

```
.log message
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *message* | Message to be logged<br>Type: Simple template |

Example:

```
.log undefined environment variable: JAVAC
```

## .lowercase and .uppercase Macros

The .lowercase macro transforms a text block in lowercase characters.

```
.lowercase
    block-input
.endlowercase
```

This macro transforms a text block in uppercase characters.

```
.uppercase
    block-input
.enduppercase
```

These macros are particularly useful when working with naming conventions (see "Naming conventions", in the Models chapter of the *Core Features Guide*).

The following parameters are available:

| Parameter | Description |
|---|---|
| *block-input* | Parameter used to input text<br>Type: Complex template |

In the following example, the variable %Comment% contains the string HELLO WORLD, which is converted to hello world.

```
.lowercase
    %Comment%
.endlowercase
```

## .object Macro

This macro returns a collection of objects based on the specified scope and condition. Object references are represented as OID; for example: E40D4254-DA4A-4FB6-AEF6-3E7B41A41AD1.

```
object = .object (scope:simple-template [,filter])
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *scope* | Collection over which we should iterate, the macro will return the first matching object in the collection<br>Type: Simple template returning either an object or a collection scope |
| *simple-template* | Template to be evaluated.<br>Type: Simple template |
| *filter* | Filter condition<br>Type: condition |

Example1:

The following macro returns the first attribute in the collection defined on the current classifier whose code starts with a letter comprised between a and e included.

```
.object(Attributes, (%.1:Code% >= a) and (%.1:Code% <= e))
```

Example 2:

Define template ::myPackage2 as follows:

```
.object(ActiveModel.Packages, %Name% == MyPackage2)
```

Define template OOM.Model::MyTemplate as follows:

```
.foreach_item(myPackage2.Classes)
%Code%
.next(\n)
```

In OOM.Model M = { OOM.Package MyPackage1, OOM.Package MyPackage2 { OOM.Class C1, OOM.Class C2} }Template OOM.Model::MyTemplate evaluates to in model M:

C1

C2

Example 3:

ILM.Publication::getConsolDataConnection

```
.object(Process.DataConnections, %AccessType% == "RO")
```

This template returns the first read-only data connection for the process associated with the current publication.

## .replace Macro

The .replace macro replaces all occurrences of a string with another string in a text block.

This macro is particularly useful when you work with naming conventions.

For more information about naming conventions, see "Naming conventions" in the Models chapter of the *Core Features Guide.*

The .replace macro replaces the old string <OldString> with the <NewString> string in the text block <Block>.

```
.replace (old-string,new-string)
    block-input
.endreplace
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *old-string* | String to be replaced.<br>Type: Text |
| *new-string* | String which replaces *old-string*.<br>Type: Text |
| *block-input* | Parameter used to input text.<br>Type: Complex template |

### Output

The output is that all instances of the string <old-string> are replaced by instances of the string <new-string> in the replace block input.

In the following example, 'GetCustomerName' is converted to 'SetCustomerName'.

```
.replace( get , set )
GetCustomerName
.endreplace
```

In the following example, the variable %Name% is 'Customer Factory' and it is converted to 'Customer_Factory'.

```
.replace(" ", "_")
%Name%
.endreplace
```

## .set_interactive_mode Macro

This macro is used to define if the GTL execution must interact with the user or not.

```
.set_interactive_mode(mode)
```

The following modes are available:

- im_Batch - Never displays dialog boxes and always uses default values
- im_Dialog - Displays information and confirmation dialog boxes that require user interaction for the execution to keep running
- im_Abort - Never displays dialog boxes and aborts execution instead of using default values each time a dialog is encountered

For example, you could use this macro if your model contains external shortcuts. If the target model of an external shortcut is closed and you are in im_Dialog mode, then a dialog box is displayed to prompt you to open the target model.

## .set_object and .set_value Macros

These macros are used to define a local variable of object (local object) or value type.

```
.set_object ( [scope.] name [,object-ref [,mode]])
```

The variable is a reference to the object specified using the second argument.

```
.set_value ( [scope.] name, value [,mode])
```

The variable value is set to be the translated template value specified as the second argument.

The following parameters are available:

| Parameter | Description |
|---|---|
| *scope* | [optional] Qualifying scope.<br>Type: Simple-template returning an object or a collection scope |
| *name* | Variable name<br>Type: Simple-template |
| *object-ref* [.set_object only] | [optional] Describes an object reference. If it is not specified or is an empty string, the variable is a reference to the active object in the current translation scope<br>Type: [*scope.*]*object-scope*] |
| *value* [.set_value only] | Value.<br>Type: Simple template (escape sequences ignored) |
| *mode* | [optional] Specifies the mode of creation. You can choose between:<br>• new - (Re)define the variable in the current scope<br>• update – [default] If a variable with the same name already exists, update the existing variable otherwise define a new one<br>• newifundef - Define the variable in the current scope if it has not been defined in an outer scope, otherwise do nothing |

Example:

```
.set_object(Attribute1, Attributes.First)
```

Example:

```
.set_value(FirstAttributeCode, %Attributes.First.Code%)
```

**Note:** When specifying a new variable, it is recommended to specify 'new' as third argument to ensure that a new variable is created in the current scope.

## .unique Macro

This macro defines a block in which each line of the text generated is guaranteed to be unique. It can be useful for calculating imports, includes, typedefs, or forward declarations in languages such as Java, C++ or C#.

```
.unique
    block-input
.endunique[(tail)]
```

The output is the block input with every redundant line removed.

The following parameters are available:

| Parameter | Description |
|---|---|
| *block-input* | Parameter used to input text<br>Type: Complex template |
| *tail* | [optional] Appended to the output, if there is one<br>Type: Text |

Example:

```
.unique
    import java.util.*;
    import java.lang.String;
    %imports%
.endunique
```

## .unset Macro

Permits the undefining of both local variables and volatile attributes defined through the .set_value and .set_object macros

```
.unset([scope.]name)
```

The following parameters are available:

| Parameter | Description |
|---|---|
| *scope* | [optional] Qualifying scope.<br>Type: Simple-template returning an object or a collection scope |
| *name* | Local variable or volatile attribute name.<br>Type: Simple template |

Example:

```
.set_value(i, 1, new)
%i?%
.unset(i)
%i?%
```

The second line outputs true as the variable 'i' is defined while the last one outputs false.

## .vbscript Macro

The vbscript macro is used to embed VB script code inside a template. It is a block macro.

A vbscript macro has the following syntax:

```
.vbscript [(script-param-list)]
    block-input
.endvbscript [(tail)]
```

The output is the ScriptResultArray value.

The following parameters are available:

| Parameter | Description |
|---|---|
| *script-param-list* | Parameters that are passed onto the script through the ScriptInputArray table.<br>Type: List of simple-template arguments separated by commas |
| *block-input* | VB script text<br>Type: Text |
| *tail* | Appended to the output, if there is one<br>Type: Text |

Example:

```
.vbscript(hello, world)
ScriptResult = ScriptInputArray(0) + " " + ScriptInputArray(1)
.endvbscript
```

The output is:

```
hello world
```

Note: the active object of the current translation scope can be accessed through the ActiveSelection collection (see *Global Properties* on page 239) as ActiveSelection.Item(0).

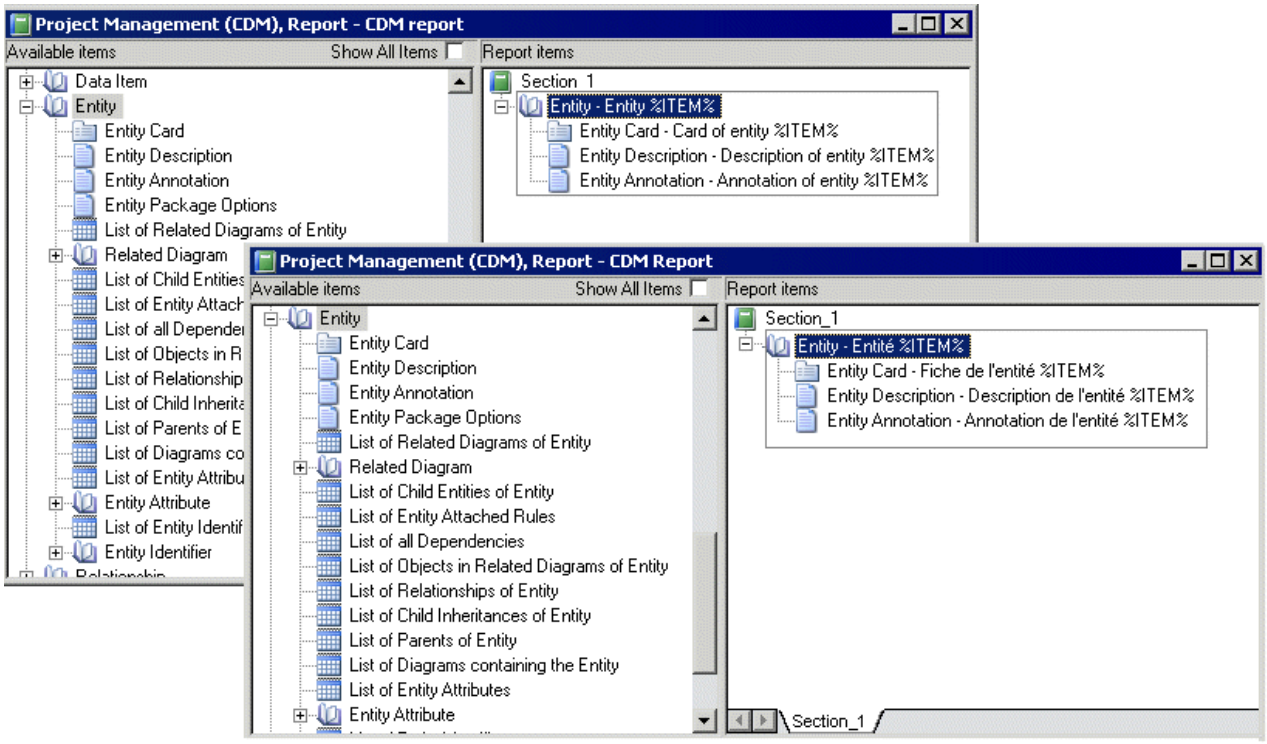
# Translating Reports with Report Language Resource Files

A report language resource file is an XML file with an .xrl extension, which contains all the text used to generate a PowerDesigner model report (such as report section titles, or names of model objects and their attributes (properties)) for a particular language. Report language resource files are stored in the Resource Files directory.

PowerDesigner ships with report language resource files in English (default), French, and simplified and traditional Chinese. You can edit these files, or use them as the basis for creating your own .xrl files to translate reports into other languages.

**Note:** When you create a report, you select a report language to display all the printable texts in the specified language. For more information, see the Reports chapter in the *Core Features Guide*.

In the following example, Entity Card, Entity Description, and Entity Annotation are shown in English and French as they will appear in the Report items pane:




The report language resource files use PowerDesigner Generation Template Language (GTL) templates to factorize the work of translation. Report Item Templates interact with your translations of the names of model objects and Linguistic Variables (that handle syntactic peculiarities such as plural forms and definite articles) to automatically generate all the textual elements in a report.

This mechanism, which was introduced in version 15 of PowerDesigner, dramatically reduces (by around 60%) the number of strings that must be translated in order to render reports in a new language.

For example the French report title Liste des données de l'entité MyEntity is automatically generated as follows:

- the List - object collections report item template (see *Profile/Report Item Templates category* on page 231) is translated as:

```
Liste des %@Value% %ParentMetaClass.OFTHECLSSNAME% %%PARENT%%
```

in which the following variables are resolved:

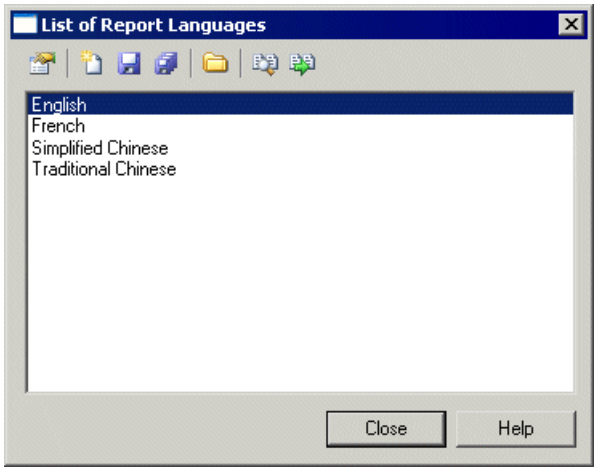
- %@Value% - resolves to the object type of the metaclass (see *Object Attributes category* on page 228). In this case, données.
- %ParentMetaClass.OFTHECLSSNAME% %%PARENT%% - resolves to the object type of the parent metaclass, as generated by the OFTHECLSSNAME linguistic variable (see *Profile/Linguistic Variables category* on page 229). In this case, l'entité.
- %%PARENT%% - resolves to the name of the specific object (see *Object Attributes category* on page 228). In this case, MyEntity.

For more information about templates, see *Customizing Generation with GTL* on page 187.

## Opening a Report Language Resource File

You can review and edit report language resource files in the Resource Editor.

1. Select **Tools** > **Resources** > **Report** Languages to open the List of Report Languages, which lists all the available .xrl files:




2. Select a report language and click the Properties tool to open it in the Resource Editor.

   **Note:** You can open the .xrl file attached to a report open in the Report Editor by selecting **Report** > **Report Properties** , and clicking the Edit Current Language tool beside the Language list. You can change the report language by selecting another language in the list.

   For more information about the tools available in the List of Report Languages, see *Resource Files and the Public Metamodel* on page 1.

## Creating a Report Language Resource File for a New Language

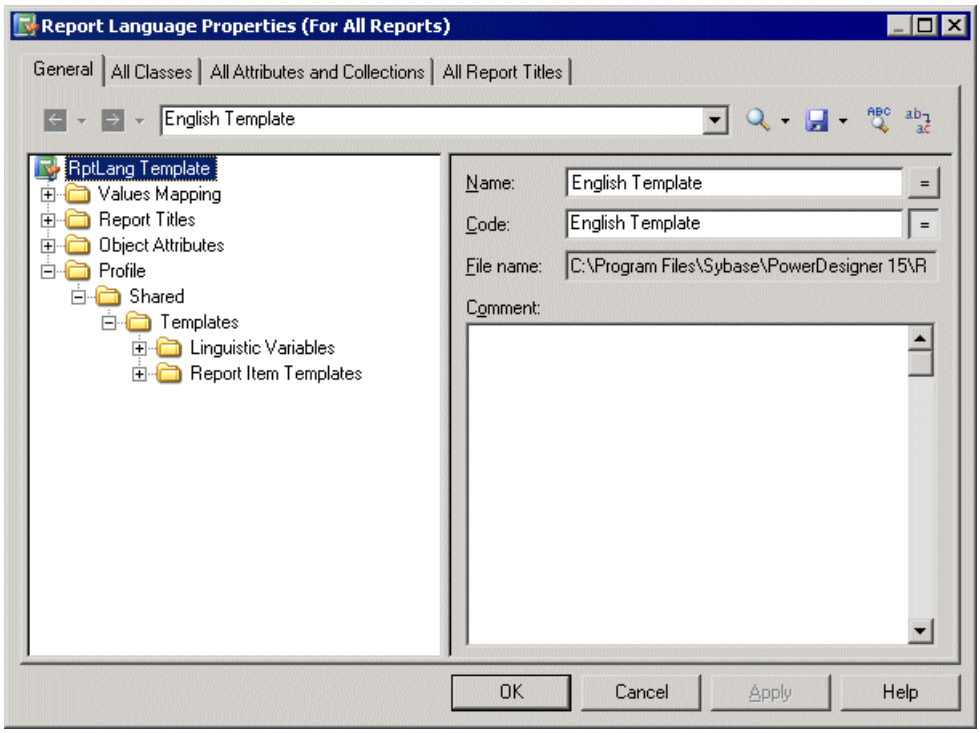
You can translate reports and other text items used to generate PowerDesigner reports into a new language.

1. Select **Tools** > **Resources** > **Report** Languages to open the List of Report Languages, which shows all the available report language resource files.
2. Click the New tool to open the New Report Language dialog box, and enter the name that you want to appear in the List of Report Languages.
3. [optional] Select a report language in the Copy from list.
4. Click OK to open the new file in the Report Language Editor.

5. Open the Values Mapping category, and translate each of the keyword values. For more information, see *Values Mapping category* on page 224.
6. Open the **Profile** > **Linguistic** Variables category to create the grammar rules necessary for the correct evaluation of the report item templates. For more information, see *Profile/Linguistic Variables category* on page 229.
7. Open the **Profile** > **Report** Items Templates category, and translate the various templates. For more information, see *Profile/Report Item Templates category* on page 231. As you translate, you may discover additional linguistic variables that you should create (see previous step).
8. Click the All Classes tab to view a sortable list of all the metaclasses available in the PowerDesigner metamodel. Translate each of the metaclass names. For more information, see *All Classes tab* on page 231.
9. Click the All Attributes and Collections tab to view a sortable list of all the attributes and collections available in the PowerDesigner metamodel. Translate each of the attribute and collection names. For more information, see *All Attributes and Collections tab* on page 232.
10. Click the All Report Titles tab, and review the automatically generated report titles. For more information, see *All Report Titles tab* on page 233. Note that this tab may take several seconds to display.
11. Click the Save tool, and click OK to close the Report Language Editor. The report language resource file is now ready to be attached to a report.

## Report Language Resource Files Properties

All report language resource files can be opened in the Resource Editor, and have the same basic category structure:




For more information about using the Resource Editor, see *Working with the Resource Editor* on page 2.

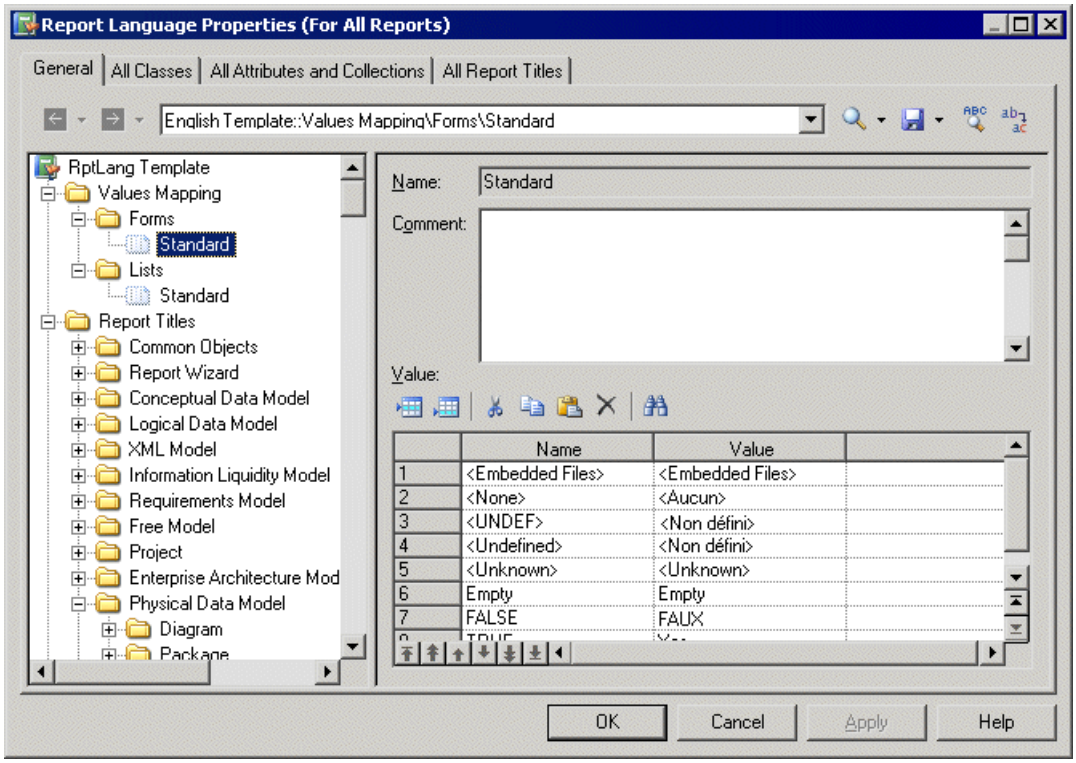
The root node of each file contains the following properties:

| Property | Description |
|---|---|
| Name | Specifies the name of the report language. |
| Code | Specifies the code of the report language. |

| Property | Description |
|---|---|
| File Name | [read-only] Specifies the path to the .xrl file. |
| Comment | Specifies additional information about the report language. |

## Values Mapping Category

The Values Mapping category contains a list of keywords values (such as Undefined, Yes, False, or None) for object properties displayed in cards, checks, and lists. You must enter a translation in the Value column for each keyword in the Name column:




This category contains the following sub-categories:

| Sub-category | Description |
|---|---|
| Forms | Contains a Standard mapping table for keywords of object properties in cards and checks, which is available to all models. You have to provide translations for keywords values in the Value column.<br>Example: Embedded Files. |
| Lists | Contains a Standard mapping table for keywords of object properties in lists, which is available to all models. You have to provide translations for keywords values in the Value column.<br>Example: True. |

You can create new mapping tables containing keywords values specific to particular types of model objects.

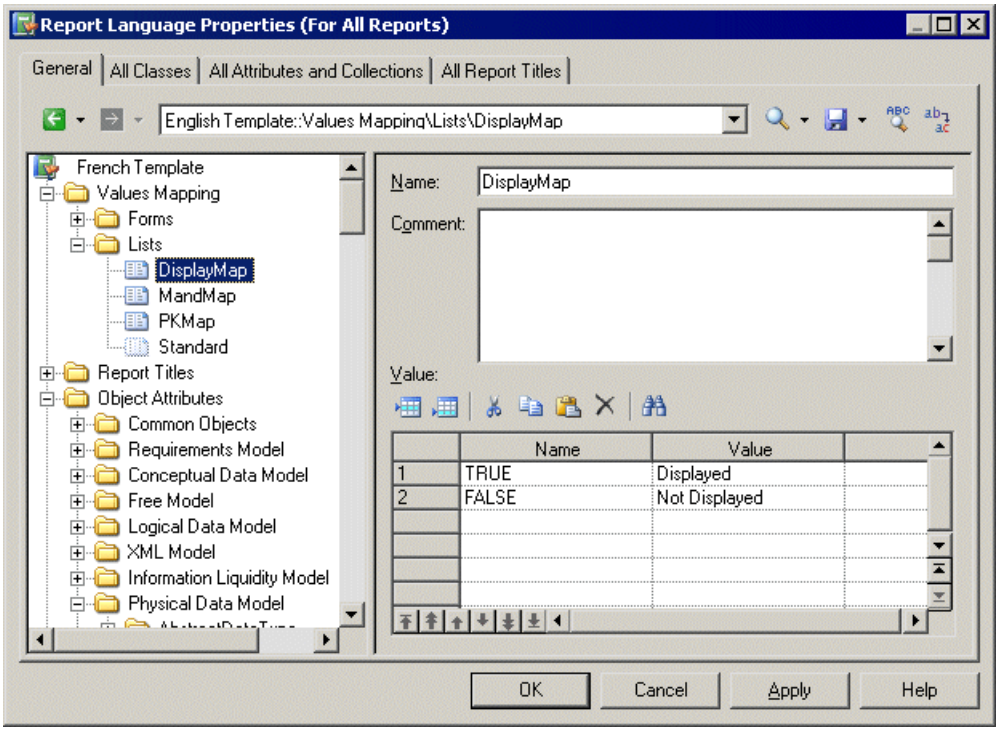
### Example: Creating a Mapping Table, and Attaching It to a Specific Model Object

You can override the values in the Standard mapping tables for a specific model object by creating a new mapping table, and attaching it to the object.

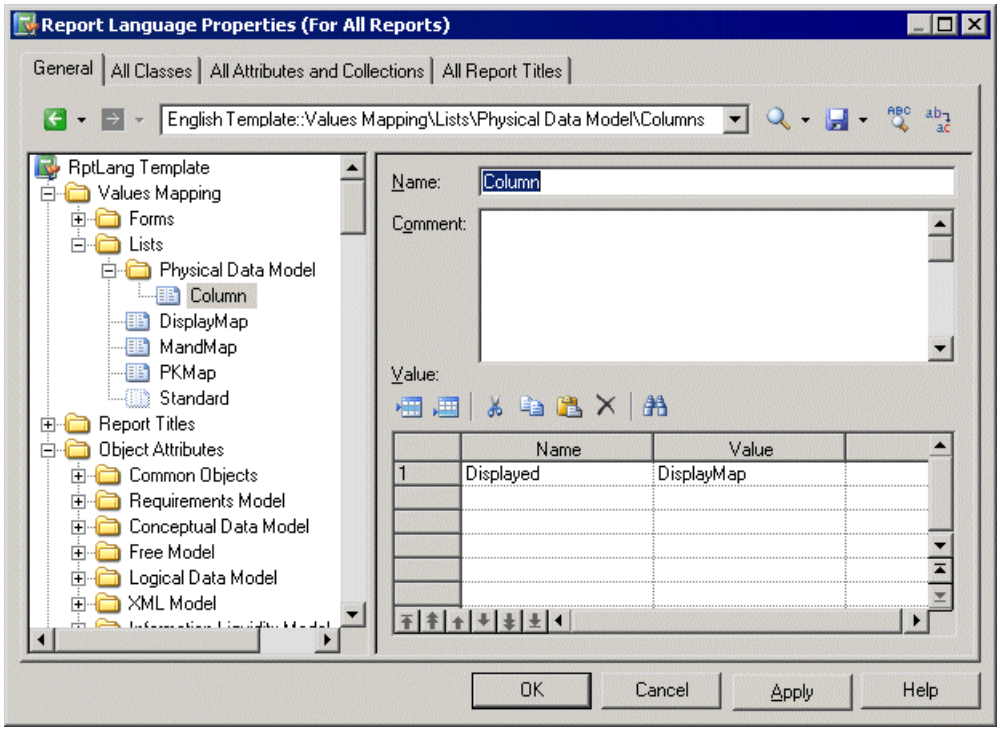
In the following example, the DisplayMap mapping table is used to override the Standard mapping table for PDM columns to provide custom values for the Displayed property, which controls the display of the selected column in the table symbol. This situation can be summarized as follows:

| Name | Value |
|---|---|
| TRUE | Displayed |
| FALSE | Not Displayed |

1. Open the **Values Mapping > Lists category** .
2. Right-click the Lists category, select **New > Map Item** to create a new list, and open its property sheet.
3. Enter DisplayMap in the Name field, enter the following values in the Value list, and click Apply:
   - Name: TRUE, Value: Displayed.
   - Name:FALSE, Value: Not Displayed.




4. Right-click the Lists category, select **New > Category** , name the category Physical Data Model, and click Apply.
5. To complete the recreation of the PDM Object Attributes tree, right-click the new Physical Data Model category, select **New > Map Item** , name the category Column, and click Apply.
6. Click the Name column to create a value and enter Displayed, which is the name of the PDM column attribute (property).
7. Click the Value column and enter DisplayMap to specify the mapping table to use for that attribute.



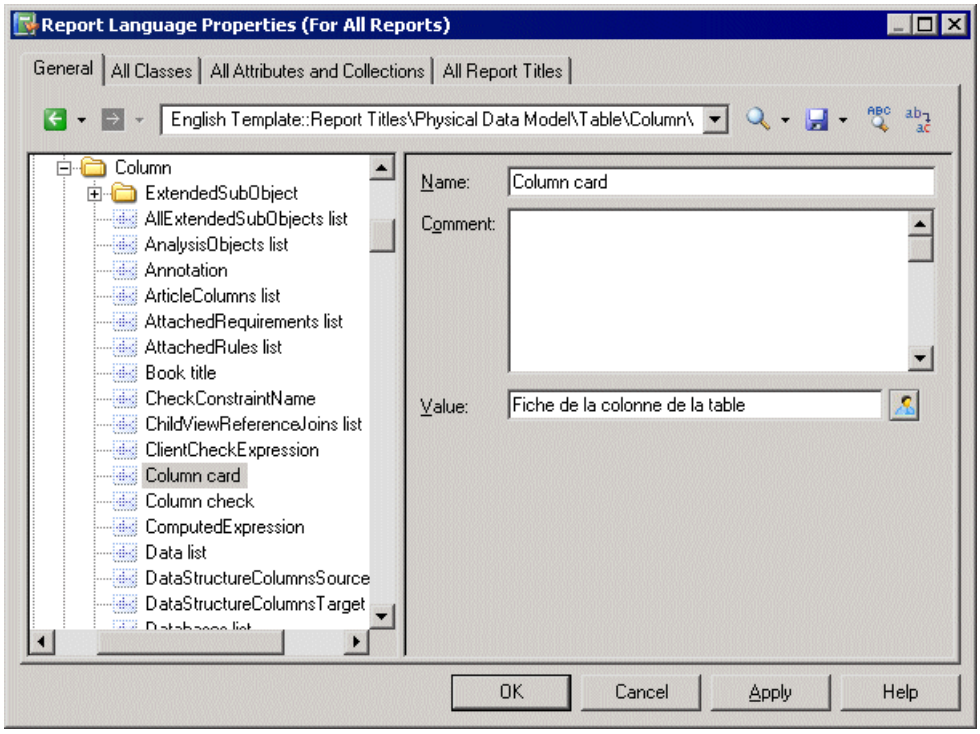

8. Click Apply to save your changes. When you generate a report, the Displayed property will be shown using the specified values:

**1 List of table columns**

| *Name* | *Code* | *Displayed* |
|---|---|---|
| id | id | Displayed |
| name | name | Not Displayed |
| size | size | Not Displayed |
| supplier | supplier | Not Displayed |
| quantity | quantity | Displayed |
| unit_price | unit_price | Displayed |

## Report Titles Category

The Report Titles category contains translations for all the possible report titles that appear in the Available Items pane in the Report Editor, those that are generated with the Report Wizard, and other miscellaneous text items.




This category contains the following sub-categories:

| Sub-category | Description |
|---|---|
| Common Objects | Contains the text items available to all models. You must provide translations of these items here.<br>Example: HTMLNext provides the text for the Next button in an HTML report. |
| Report Wizard | Contains the report titles generated with the Report Wizard. You must provide translations of these items here.<br>Example: Short description title provides the text for a short description section when you generate a report with the Report Wizard. |
| [Models] | Contain the report titles and other text items available to each model. These are automatically generated, but you can override the default values.<br>Example: DataTransformationTasks list provides the text for the data transformation tasks list of a given transformation process in the Information Liquidity Model. |

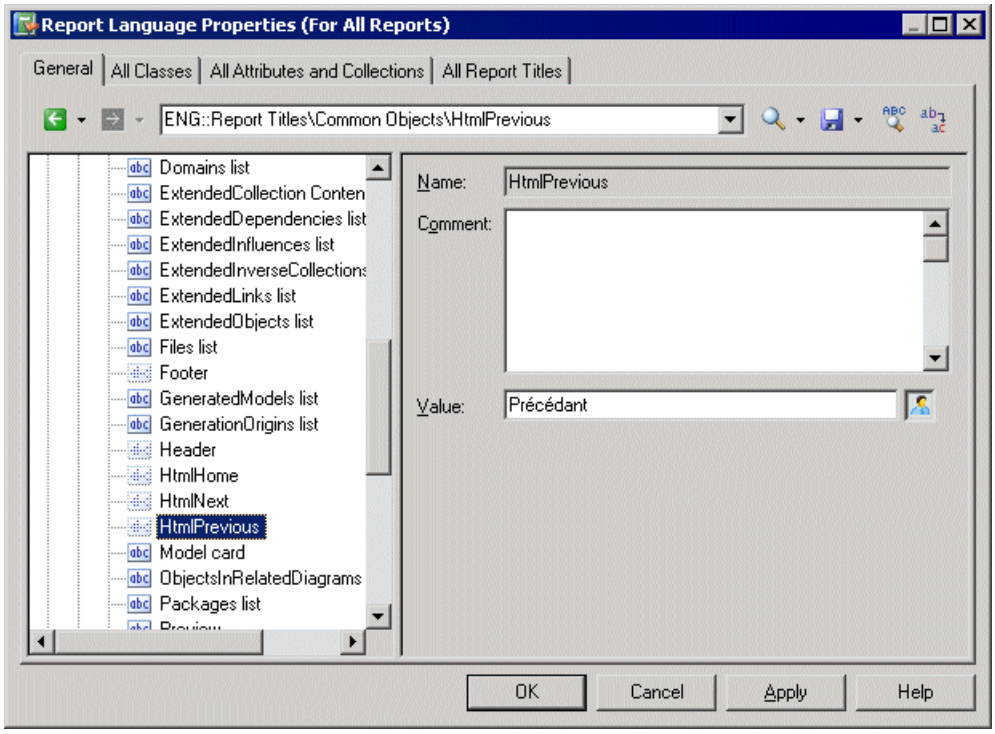
By default (with the exception of the Common Objects and Report Wizard sub-categories) these translations are automatically generated from the templates in the Profile category (See *Profile/Report Item Templates category* on page 231). You can override the automatically generated values by entering your own text in the Localized name field. The User-Defined button is automatically depressed to indicate that the value is not generated.

**Note:** The All Report Titles tab (see *All Report Titles tab* on page 233) displays the same translations shown in this category in a simple, sortable list form. You may find it more convenient to check and, where appropriate, to override generated translations on this tab.

### Example: Translating the HTML Report Previous Button

The HTML report Previous button is a common object available to all models, and located in the Common Objects category. You must translate this text item manually along with the other items in this, and the Report Wizard categories.

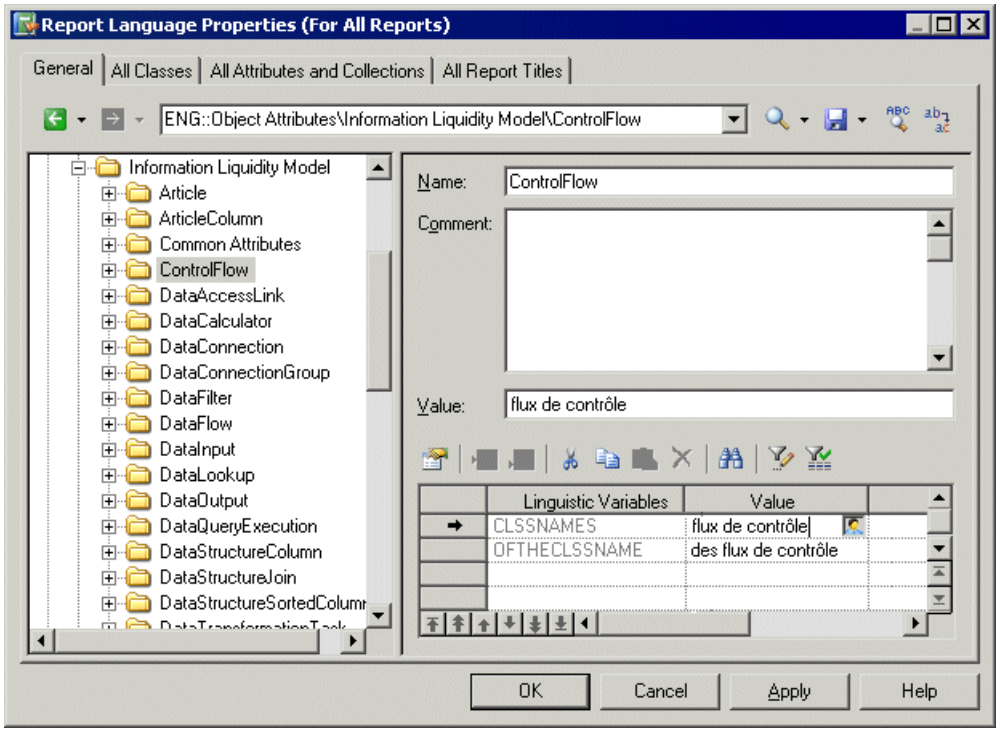
1. Open the **Report Titles** > **Common Objects** category.
2. Click the HtmlPrevious entry to display its properties, and enter a translation in the Localized name box. The User-Defined button is automatically depressed to indicate that the value is not generated.




3. Click Apply to save your changes.

## Object Attributes Category

The Object Attributes category contains all the metaclasses, collections and attributes available in the PowerDesigner metamodel, organized in tree form:




This category contains the following sub-categories:

| Sub-category | Description |
|---|---|
| [Models] | Contain text items for metaclasses, collections and attributes available to each model, for which you must provide translations.<br>Example: Action provides the text for an attribute of a process in the Business Process Model. |
| Common Objects | Contains text items for metaclasses, collections and attributes available to all models, for which you must provide translations.<br>Example: Diagram provides the text for a diagram in any model. |

For each item the name is given, and you must provide a translation in the Localized name field.This value is retrieved by the templates you have specified in the Profile category to generate default report titles (see *Report Titles category* on page 226).

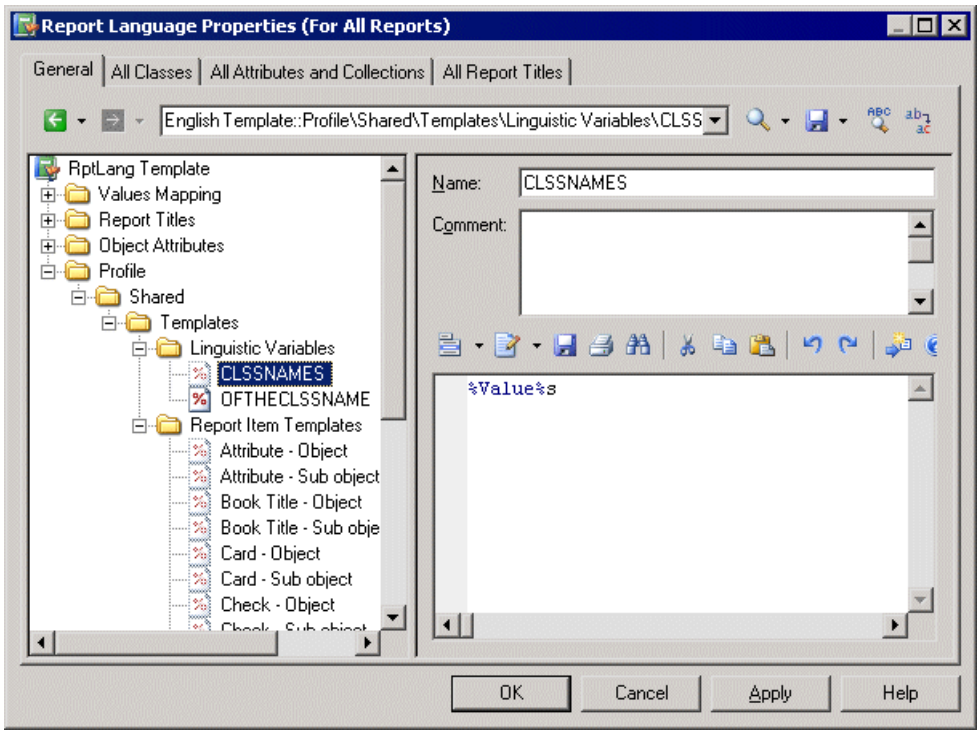
For metaclasses only, the linguistic variables you have specified (see *Profile/Linguistic Variables category* on page 229) are listed along with the results of their application to the translations given in the Localized name field. If necessary, you can override the automatically generated values by entering your own text in the Value column. The User-Defined button is automatically depressed to indicate that the value is not generated.

**Note:** These tabs display the same translations shown in the Object Attributes category in a simple, sortable list form. You may find it more convenient to provide translations in these tabs (see *All Classes tab* on page 231 and *All Attributes and Collections tab* on page 232).

## Profile/Linguistic Variables Category

The Linguistic Variables category contains templates, which specify grammar rules to help build the report item templates.

Examples of grammar rules include the plural form of a noun, and the correct definite article that must precede a noun. For more informations, see *Profile/Report Item Templates category* on page 231.




Specifying appropriate grammar rules for your language, and inserting them into your report item templates will dramatically improve the quality of the automatic generation of your report titles. You can create as many variables as your language requires.

Each linguistic variable and the result of its evaluation is displayed for each metaclass in the Object Attributes category (see *Object Attributes category* on page 228).

The following are examples of grammar rules specified as linguistic variables to populate report item templates in the French report language resource file:

- GENDER – Identifies as feminine a metaclass name %Value%, if it finishes with "e" and as masculine in all other cases:

```
.if (%.-1:@Value% == e)
F
.else
M
.endif
```

For example: la table, la colonne, le trigger.

- CLSSNAMES – Creates a plural by adding "x" to the end of the metaclass name %Value%, if it finishes with "eau" or "au" and adds "s" in all other cases:

```
.if (%.-3:@Value% == eau) or (%.-2:@Value% == au)
%@Value%x
.else
%@Value%s
.endif
```

For example: les tableaux, les tables, les entités.

- THECLSSNAME – Inserts the definite article before the metaclass name %Value% by inserting " l' ", if it begins with a vowel, "le" if it is masculine, and "la" if not:

```
.if (%.1U:@Value% == A) or (%.1U:@Value% == E) or (%.1U:@Value% == I) or (%.
1U:@Value% == O) or (%.1U:@Value% == U)
l'%@Value%
.elsif (%GENDER% == M)
le %@Value%
.else
la %@Value%
.endif
```

For example: l'association, le package, la table.

- OFTHECLSSNAME – Inserts the preposition "de" plus the definite article before the metaclass name %Value%,if it begins with a vowel or if it is feminine, otherwise "du".

```
.if (%.1U:@Value% == A) or (%.1U:@Value% == E) or (%.1U:@Value% == I) or (%.
1U:@Value% == O) or (%.1U:@Value% == U) or (%GENDER% == F)
de %THECLSSNAME%
.else
du %@Value%
.endif
```

For example: de la table, du package.

- OFCLSSNAME – Inserts the preposition " d' " before the metaclass name %Value%,, if it begins with a vowel, otherwise "de".

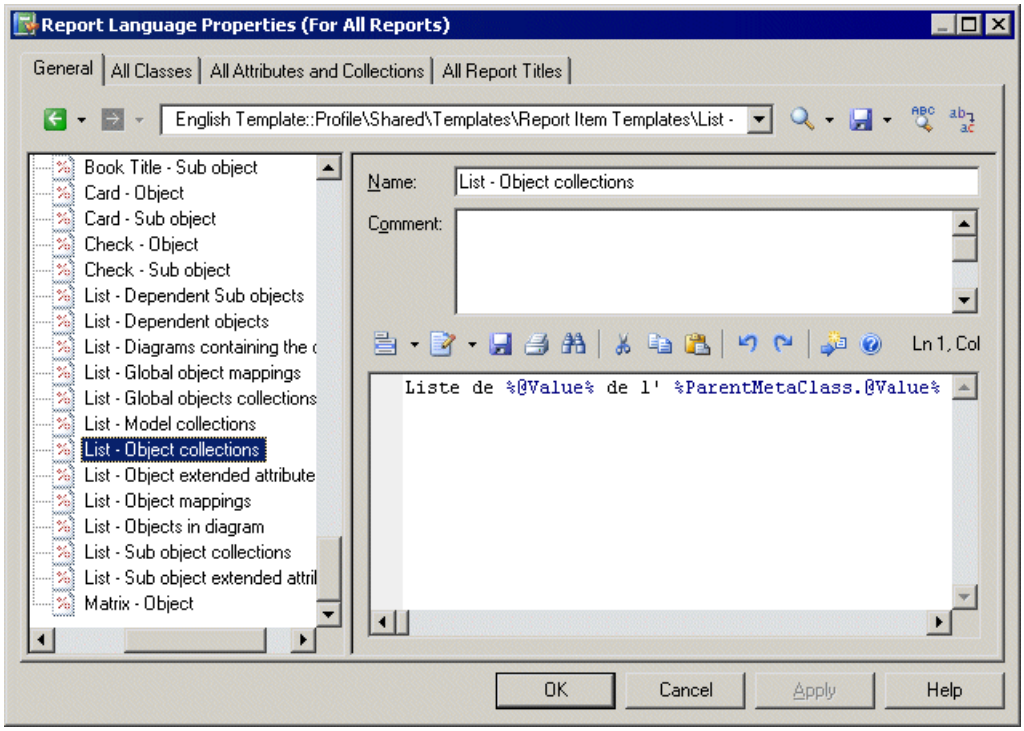
```
.if (%.1U:@Value% == A) or (%.1U:@Value% == E) or (%.1U:@Value% == I) or (%.
1U:@Value% == O) or (%.1U:@Value% == U)
d'%@Value%
.else
de %@Value%
.endif
```

For example: d'association, de table.

## Profile/Report Item Templates Category

The Report Item Templates category contains a set of templates that, in conjunction with the translations that you will provide for metaclass, attribute and collection names, are evaluated to automatically generate all the possible report titles for report items (book, list, card etc.)

For more information, see *Object Attributes category* on page 228.




You must provide translations for each template by entering your own text. Variables (such as %text%) must not be translated.

For example the template syntax for the list of sub-objects contained within a collection belonging to an object is the following:

```
List of %@Value% of the %ParentMetaClass.@Value% %%PARENT%%
```

When this template is evaluated, the variable %@Value% is resolved to the value of the localized name for the object, %ParentMetaClass.@Value% is resolved to the value of the localized name for the parent of the object, and %%PARENT %% is resolved to the name for the parent of the object.

In this example, you translate this template as follows:

- Translate the non-variable items in the template. For example:
- Create a linguistic variable named OFTHECLSSNAME to specify the grammar rule used in the template (see *Profile/ Linguistic Variables category* on page 229).

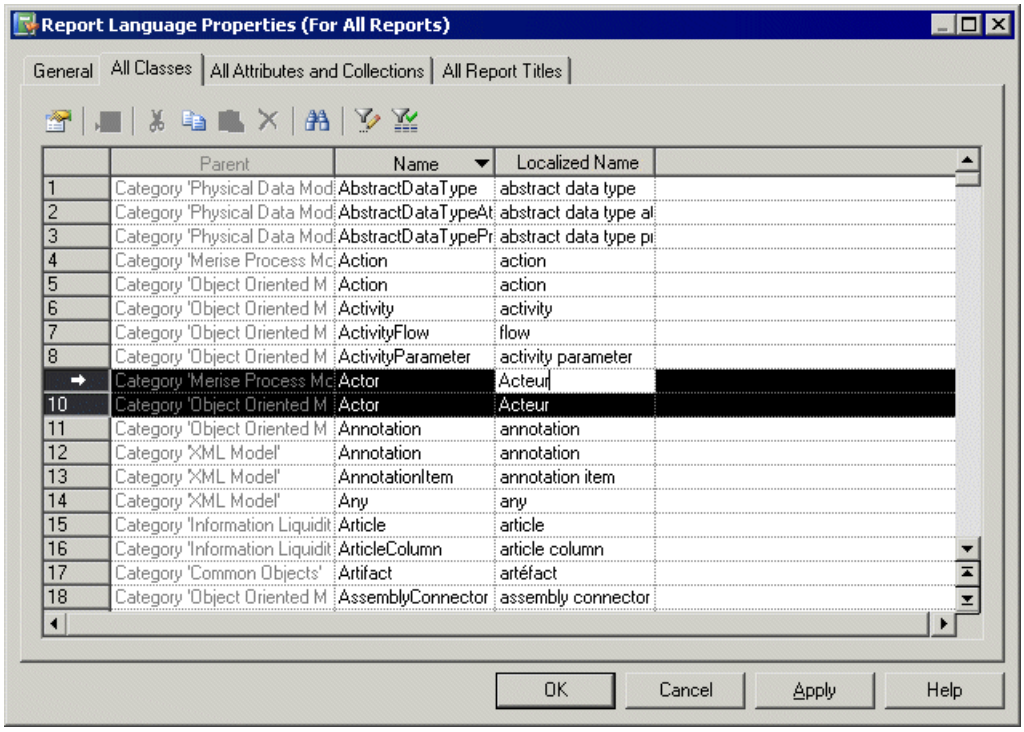
This template will be reused to create report titles for all the lists of sub-objects contained within a collection belonging to an object.

You cannot delete templates nor create new ones.

## All Classes Tab

The All Classes tab lists all the metaclasses available in the Object Attributes category on the General tab but the flat structure makes it more convenient to work with.

For more information, see *Object Attributes category* on page 228.

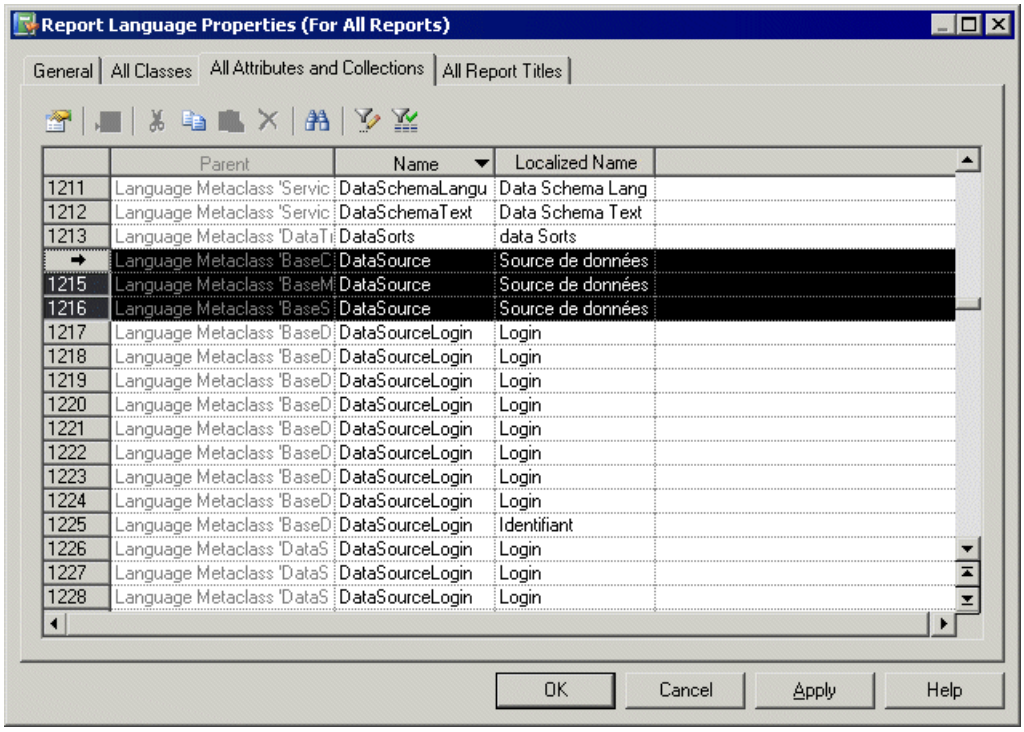



For each metaclass listed in the Name column, you must enter a translation in the Localized name column. You can sort the list to group similarly-named objects, and translate identical items together by selecting multiple lines.

## All Attributes and Collections Tab

The All Attributes and Collections lists all the collections and attributes available in the Object Attributes category on the General tab, but the flat structure makes it more convenient to work with.

For more information, see *Object Attributes category* on page 228.

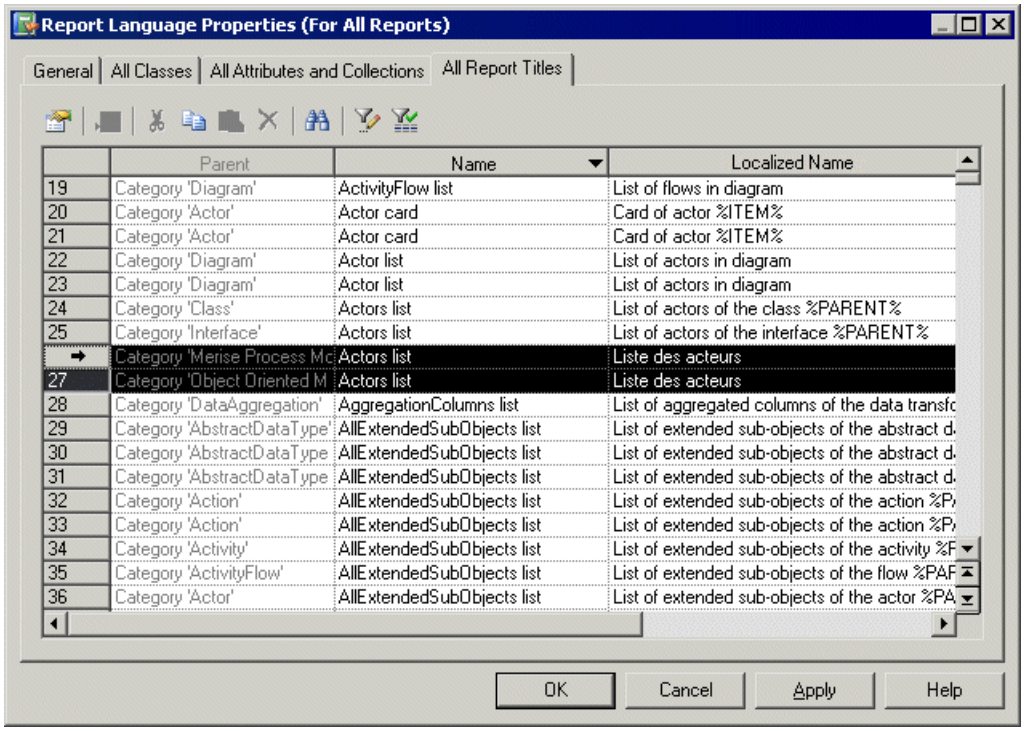


For each attribute or collection listed in the Name column, you must enter a translation in the Localized name column. You can sort the list to group similarly-named objects, and translate identical items together by selecting multiple lines.

## All Report Titles Tab

The Report Titles tab lists all the report titles and other miscellaneous text items available in the Report Titles category on the General tab, but the flat structure makes it more convenient to work with.

For more information, see *Object Attributes category* on page 228.




For each report listed in the Name column, you can review or override a translation in the Localized name column. You can sort the list to group similarly-named objects, and translate identical items together by selecting multiple lines.

# Scripting PowerDesigner

When working with large or multiple models, it can sometimes be tedious to perform repetitive tasks, such as modifying objects using global rules, importing or generating new formats or checking models.

Such operations can be automated through scripts. Scripting is widely used in various PowerDesigner features. For example, to:

- Create custom checks, event handlers, transformations, customs commands and custom popup menus (see *Extending Your Models with Profiles* on page 121)
- Communicate with PowerDesigner from another application (see *Communicating With PowerDesigner Using OLE Automation* on page 280)
- Customize PowerDesigner menus by adding your own menu items (see *Customizing PowerDesigner Menus Using Add-Ins* on page 284).
- Create VBscript macros and embed VBscript code inside a template for generation (see *GTL Macro Reference* on page 204).

You can access PowerDesigner objects using any scripting language such as Java, VBScript or C# (C Sharp). However, the scripting language used to illustrate our examples in this chapter is VBScript.

VBScript is a Microsoft scripting language. PowerDesigner provides integrated support for Microsoft VBScript so that you can write and run scripts to interact with metamodel objects in a development environment using *properties* and *methods*. Every PowerDesigner object can be read and modified (creation, update or deletion).

## Accessing PowerDesigner Metamodel Objects

PowerDesigner ships with a metamodel published in an Object Oriented Model (metamodel.oom) that illustrates how metadata interact in the software. All objects in the PowerDesigner metamodel have a name and a code. They correspond to the *public name* of the metadata. An HTML help file is also provided to allows you to find out which properties and methods can be used to drill down to a PowerDesigner object.

For more information on metadata, see *Resource Files and the Public Metamodel* on page 1.

PowerDesigner also provides a set of pre-written scripts that you can modify to meet your own needs.

Scripting allows you to perform any kind of data manipulation but you can also insert and customize commands in the Tools menu that will allow you to automatically launch your own scripts.

### Objects

*Objects* refer to any PowerDesigner objects. They can be:

- Design objects, such as tables, classes, processes or columns.
- Diagrams or symbols.
- Functional objects, such as the report or the repository.

An object belongs to a metaclass of the PowerDesigner metamodel.

Each object has properties, collections and methods that it inherits from its metaclass.

Root objects like models for example are created or retrieved using global methods. For more information, see *Global properties* on page 239.

Non root objects are created or retrieved using collections. For example, you create these objects using a Create method on collections and delete them using a Delete method on collections. For more information, see *Collections* on page 236.

You can browse the PowerDesigner metamodel to get information about the properties and collections available for each metaclass.

### Example

```
'Variables are not typed in VBScript. You create them and the
'location where you use them determines what they are
' get the current active model
Dim mdl ' the current model
Set mdl = ActiveModel
```

## Properties

A *property* is an elementary information available for the object. It can be the name, the code, the comment etc.

### Example

```
'How to get a property value in a variable from table 'Customer
Dim Table_name
'Assuming MyTable is a variable that already contains a 'table object
Get the name of MyTable in Table_name variable
Table_name = MyTable.name
'Display MyTable name in output window
output MyTable.name
'How to change a property value : change value for name 'of MyTable
MyTable.name = 'new name'
```

## Collections

A *collection* is a set of objects.

The model is the root object and the other objects can be reached by browsing the corresponding collection. The objects are grouped together within collections that can be compared to the category nodes appearing in the Browser tree view of the Workspace.

If an object CUSTOMER has a collection, it means the collection contains the list of objects with which the object CUSTOMER is in relation.

Some functions are available on collections. You can:

- Browse a collection
- Get the number of objects a collection contains
- Create a new object inside a collection, if it is a composition collection

Collections can be of the following types:

- Read-only collections are collections that can only be browsed
- Unordered collections are collections for which objects order in the list is not significant. For example the Relationships collection of a CDM Entity object is an unordered collection
- Ordered collections are collections for which object order is set by the user and must be respected. For example the Columns collection of the PDM Table object is an ordered collection
- Composition collections are collections for which objects belong to the collection owner. They are usually displayed in the Browser. Non composition collections can also be accessed using scripting and can be for example the list of business rules attached to a table or a class and displayed in the Rules tab of its property sheet or the list of objects displayed in the Dependencies tab of an object property sheet.

### Read-only Collections

Models (global collection for opened models) is an example of read-only collection.

The property and method available for read-only collections are the following:

| Property or Method | Use |
|---|---|
| Count As Long | Retrieves the number of objects in collection |
| Item(idx As Long = 0) As BaseObject | Retrieves the item in collection for a given index. Item(0) is the first object |
| MetaCollection As BaseObject | Retrieves the MetaCollection object that defines this collection |
| Kind As Long | Retrieves the kind of objects the collection can contain. It returns a predefined constant such as cls_ |
| Source As BaseObject | Retrieves the object that owns the collection |

Example:

```
'How to get the number of open models and display it
'in the output window
output Models.count
```

### Unordered Collections

All methods and properties for read-only collections are also available for unordered collections.

Properties and methods available for unordered collections are the following:

| Property or Method | Use |
|---|---|
| Add(obj As BaseObject) | Adds object as the last object of the collection |
| Remove(obj As BaseObject, delete As Boolean = False) | Removes the given object from collection and optionally delete the object |
| CreateNew(kind As Long = 0) As BaseObject | Creates an object of a given kind, and adds it at the end of collection. If no object kind is specified the value 0 is used which means that the Kind property of the collection will be used. See the Metamodel Objects Help file for restrictions on using this method |
| Clear(delete As Boolean = False) | Removes all objects from collection and optionally delete them |

Example:

```
'remove table TEST from the active model
Set MyModel = ActiveModel
For each T in Mymodel.Tables
 If T.code = "TEST" then
   set MyTable = T
 End if
next
ActiveModel.Tables.Remove MyTable, true
```

### Ordered Collections

All methods and properties for read-only and unordered collections are also available for ordered collections.

Properties and methods available for ordered collections are the following:

| Property or Method | Use |
|---|---|
| Insert(idx As Long = -1, obj As BaseObject) | Inserts objects in collection. If no index is provided, the index -1 is used, which means the object is simply added as the last object of the collection |

| Property or Method | Use |
|---|---|
| RemoveAt(idx As Long = -1, delete As Boolean = False) | Removes object at given index from collection. If no index is provided the index -1 is used, which means the removed object is the last object in collection (if any). Optionally deletes the object |
| Move(source As Long, dest As Long) | Moves object from source index to destination index |
| CreateNewAt( idx As Long = -1, kind As Long = 0) As BaseObject | Creates an object of a given kind, and inserts it at given position. If no index is provided the index -1 is used which means the object is simply added as the last object of the collection. If no object kind is specified the value 0 is used which means that the Kind property will be used. See the Metamodel Objects Help file for restrictions on using this method |

Example:

```
'Move first column in last position
'Assuming the variable MyTable contains a table
MyTable.Columns.move(0,-1)
```

### Composition Collections

Composition collections can be ordered or unordered.

All methods and properties for unordered collections are also available for unordered compositions.

Properties and methods available for unordered composition collections are the following:

| Property or Method | Use |
|---|---|
| CreateNew(kind As Long = 0) As BaseObject | Creates an object of a given kind, and adds it at the end of collection. If no object kind is specified the value 0 is used, which means the Kind property of the collection will be used |

All methods and properties for ordered collections are also available for ordered compositions.

All methods and properties for unordered compositions are also available for ordered compositions.

Properties and methods available for ordered composition collections are the following:

| Property or Method | Use |
|---|---|
| CreateNewAt( idx As Long = -1, kind As Long = 0) As BaseObject | Creates an object of a given kind, and inserts it at a given position. If no index is provided the index -1 is used, which means the object is simply added as the last object of the collection. If no object kind is specified the value 0 is used which means that the Kind property of the collection will be used |

These methods can be called with no object kind specified, but this is only possible when the collection is strongly typed. That is, the collection is designed to contain objects of a precise non-abstract object kind. In such cases, the Kind property of the collection corresponds to an instantiable class and the short description of the collection states the object kind name.

Example:

The Columns collection of a table is a composition collection as you can create columns from it. But the Columns collection of a key is not a composition collection as you cannot create objects (columns) from it, but only list them.

```
'Create a new table in a model
'Assuming the variable MyModel contains a PDM
'Declare a new variable object MyTable
```

```
Dim MyTable
'Create a new table in MyModel
Set MyTable = MyModel.Tables.Createnew
```

```
'Create a new column in a table
'Declare a new variable object MyColumn
Dim MyColumn
'Create a new column in MyTable in 3rd position
Set MyTable = MyTable.Columns.CreateNewAt(2)
' the column is created with a default name and code
```

**Note:** When you browse the collections of a model and want to retrieve its objects, be aware that you will also retrieve the shortcuts of objects of the same type.

## Global Properties

The available global properties can be gathered as follows:

| Type | Global property | Use |
|---|---|---|
| Global accessor | ActiveModel As BaseObject<br>ActivePackage As BaseObject<br>ActiveDiagram As BaseObject | Retrieves the model, package, or diagram that corresponds to the active view |
| | ActiveSelection As ObjectSet | Read-only collection that retrieves the list of selected objects in the active diagram |
| | ActiveWorkspace As BaseObject | Retrieves the Application active Workspace |
| | MetaModel As BaseObject | Retrieves the Application MetaModel |
| | Models As ObjectSet | Read-only collection that lists opened models |
| | RepositoryConnection As BaseObject | Retrieves the current repository connection, which is the object that manages the connection to the repository server and then provides access to documents and objects stored under the repository |
| Execution mode | ValidationMode As Boolean | Enables or disables the validation mode (True/False). |
| | InteractiveMode As long | Manages the user interaction by displaying dialog boxes or not using the following constants (im_+Batch, +Dialog or +Abort). |
| Application | UserName As String | Retrieves the user login name |
| | Viewer As Boolean | Returns True if the running application is a Viewer version that has limited features |
| | Version As String | Returns the PowerDesigner version |
| OLE specific | ShowMode As | Checks or changes the visibility status of the main application window in the following way:<br>• It returns True if the application main window is visible and not minimized<br>• False otherwise |
| | Locked As Boolean | Can be set to True to ensure that the application continues to run after an OLE client disconnects otherwise the application closes |

Example:

```
'Create a new table in a model
'Get the active model in MyModel variable
Set MyModel = ActiveModel
```

You can use two types of execution mode when running a script in the editor. A default value can be specified for each mode:

- Validation mode
- Interactive mode

### Validation Mode

The validation mode is enabled by default (set to True), but you may choose to temporarily disable it by setting it to False.

| State | Constant | Code | Use |
|---|---|---|---|
| Enabled (default value) | True | ValidationMode = True | Each time you act over a PowerDesigner object, all internal PowerDesigner methods are invoked to check the validity of your actions. In case of a forbidden action, an error occurs. This mode is very useful for debugging but is necessarily performance consuming |
| Disabled | False | ValidationMode = False | You use it for performance reasons or because your algorithm temporarily requires an invalid state. However, be aware, that no validation rules such as name uniqueness or link object with missing extremities are applied to your model in this case |

Example:

```
ValidationMode = true
```

### Interactive Mode

The interactive mode is Batch by default.

The interactive mode supports the following constants:

| Constant | Code | Description |
|---|---|---|
| im_Batch | InteractiveMode = im_Batch | Never displays dialog boxes and always uses default values. You use it for Automation scripts that require no user interaction |
| im_Dialog | InteractiveMode = im_Dialog | Displays information and confirmation dialog boxes that require user interaction for the script to keep running |
| im_Abort | InteractiveMode = im_Abort | Never displays dialog boxes and aborts the script instead of using default values each time a dialog is encountered |

### Option Explicit Statement

We recommend to use the Option Explicit statement to declare your variables in order to avoid confusion in code as this option is disabled by default in VBScript. You have to declare a variable before using this option.

Example:

```
Option Explicit
ValidationMode = True
```

```
InteractiveMode = im_Batch
' get the current active model
Dim mdl ' the current model
Set mdl = ActiveModel
```

## Global Functions

The following global functions are available:

| Global functions | Use |
|---|---|
| CreateModel<br>(modelkind As Long, filename As String = "", flags As Long =omf_Default) As BaseObject | Creates a new model |
| CreateModelFromTemplate<br>(filename As String, flags As Long =omf_Default) As BaseObject | Creates a new model using given model file as template |
| OpenModel<br>(filename As String, flags As Long =omf_Default) As BaseObject | Opens an existing model (including V6 models) |
| Output<br>(message As String = "") | Writes a message in the Script tab of the Output window of PowerDesigner main window |
| NewPoint<br>(X As Long = 0, Y As Long = 0) As APoint | Creates a point to position a symbol |
| NewRect<br>(Left As Long = 0, Top As Long = 0, Right As Long = 0, Bottom As Long = 0) As Arect | Creates a rectangle to manipulate symbols position |
| NewPtList () As PtList | Creates a list of points to position a link |
| NewGUID() As String | Creates a new Global Unique IDentifier (GUID). This new GUID is returned as a string without the usual surrounding "{" "}" |
| IsKindOf(childkind As Long, parentkind As Long) As Boolean | Returns True if childkind corresponds to a metaclass derived from the metaclass of kind parentkind, False otherwise |
| ExecuteCommand<br>(cmd As String, Optional arglist As String, Optional mode As Long) As String | Opens an external application |
| Rtf2Ascii<br>(rtf As String) As String | Removes RTF (Rich-Text-File) tags from an RTF formatted text |
| ConvertToUTF8<br>(InputFileName As String, OutputFileName As String) | Converts <InputFileName> file into UTF8 (8-bit Unicode Transformation Format, where byte order is specified by an initial Byte-Order Mark) and writes the result to the file <OutputFileName>. The two filenames must be different |
| ConvertToUTF16<br>(InputFileName As String, OutputFileName As String) | Converts <InputFileName> file into UTF16 (16-bit Unicode Transformation Format Little Endian, where byte order is specified by an initial Byte-Order Mark) and writes the result to the file <OutputFileName>. The two filenames must be different |

| Global functions | Use |
|---|---|
| EvaluateNamedPath<br>(FileName As String, QueryIfUnknown As Boolean = True, FailOnError As Boolean = False) As String | Replaces a variable in a path by the corresponding named path |
| MapToNamedPath<br>(FileName As String) As String | Replaces the path of a file by the corresponding named path |
| Progress(Key As String, InStatusBar Boolean = False) As BaseObject | Create or retrieve a given progress indicator |
| BeginTransaction() | Starts a new transaction |
| CancelTransaction() | Cancels the ongoing transaction |
| EndTransaction() | Commits the ongoing transaction |

### OpenModel(), CreateModel() and CreateModelFromTemplate Flags

OpenModel, CreateModel and CreateModelFromTemplate functions use the following global constants:

| Constant | Use |
|---|---|
| Omf_Default | Default behavior for OpenModel/CreateModel |
| Omf_DontOpenView | Does not open default diagram view for OpenModel/CreateModel/ CreateModelFrom-Template |
| Omf_QueryType | For CreateModel ONLY: Forces querying initial diagram type |
| Omf_NewFileLock | For CreateModel ONLY: Creates and locks corresponding file |
| Omf_Hidden | Does not let the model appear in the workspace for OpenModel/CreateModel/CreateMo-delFromTemplate |

### Command Execution Modes

Command execution modes use the following global constants:

| Constant | Use |
|---|---|
| cmd _ShellExec | Default behavior: lets MS-Windows shell execute the command |
| cmd _PipeOutput | Redirects the command output to the General tab of PowerDesigner Output window |
| cmd _PipeResult | Captures the whole command output to the returned string |
| cmd_InternalScript | Indicates that the first parameter of the Execute Command is a VBScript file to be executed as an internal script rather than letting the system run the application associated with the file type |

Example:

```
'Create a new model and print its name in output window
CreateModel(PDOOm.cls_Model, "C:\Temp\Test.oom|Language=Java|
Diagram=SequenceDiagram")
Output ActiveModel.name
```

## Global Constants

The following global constants are available:

| Global constants | Use |
|---|---|
| Version As String | Returns the application version string |
| HomeDirectory As String | Returns the application home directory string |
| RegistryHome As String | Returns the application registry home path string |
| cls_... As Long | Identifies the class of an object. This value is used when you need to specify an object kind in creation method for example. This value is also used by IsKindOf method available on all PowerDesigner objects |

### Classes Ids Constants

Constants are unique within a model and are used to identify object classes in each library. All classes Ids start with "cls_" followed by the public name of the object. For example cls_Process identifies the Process object class using the public name of the object.

However, when dealing with several models, some constants may be common, for example cls_Package.

To avoid confusion in code, you must prefix the constant name with the name of the module, for example PdOOM.cls_Package. Same, when you want to create a model, you need to prefix the cls_Model constant with the name of the module.

### IsKindOf Method

You can use the IsKindOf (ByVal Kind As Long) As Boolean method together with a class constant in order to check if an object inherits from a given class kind.

Example:

You can have a script with a loop that browses the Classifiers collection of an OOM and wants to check the type of encountered objects (in this case interfaces or classes) in order to perform different actions according to their type.

```
'Assuming the Activemodel is an OOM model
For each c in Activemodel.Classifiers
If c.IsKindOf(cls_Class) then
Output "Class " & c.name
ElsIf c.IsKindOf(cls_Interface) then
Output "Interface" & c.name
End If
Next
```

Example:

All the collections under a model can contain objects of a certain type but also shortcuts for objects of the same type. You can have a script with a loop that browses the Tables collection of a PDM and want to check the type of encountered objects (in this case tables or shortcuts) in order to perform different actions according to their type.

```
For each t in Activemodel.Tables
If t.IsKindOf(cls_Table) then
Output t.name
End If
Next
```

## Libraries

The following libraries are available. Each of them (apart from PdCommon, PdRMG and PdWSP) is equivalent to a model type:

| Library name | Corresponding model |
|---|---|
| PdCommon | Objects common to all models |
| PdWSP | Workspace |
| PdRMG | Repository |
| PdPDM | Physical Data Model |
| PdBPM | Business Process Model |
| PdCDM | Conceptual Data Model |
| PdLDM | Logical Data Model |
| PdILM | Information Liquidity Model |
| PdFRM | Free Model |
| PdPRJ | Project |
| PdEAM | Enterprise Architecture Model |
| PdIAM | Impact Analysis Model |
| PdOOM | Object Oriented Model |
| PdRQM | Requirements Model |
| PdXSM | XML Model |

*PdCommon* does not correspond to a particular model, it gathers all objects shared among two or more models. For example, business rules are defined in this library.

It also defines the abstract classes of the model, for example, BaseObject is defined in diagram Common Abstract Objects in the Objects package of PdCommon.

Models are linked to PdCommon by generalization links indicating that each model inherits common objects from the PdCommon library.

For each library, you can browse a list of:

- *Abstract classes* (located in the Abstract Classes expanded node). They are general classes that are used to factorize attributes and behaviors. They are not visible in PowerDesigner. Instantiable classes inherit from abstract classes
- *Instantiable classes* (located directly at the root of each library node). They are specific classes that correspond to interface objects, they have proper attributes like name or code, and they also inherit attributes and behaviors from abstract classes via generalization links. For example, NamedObject is the common class for most PowerDesigner design objects, it stores standard attributes like name, code, comment, annotation, and description

For more information on PowerDesigner libraries, see *Resource Files and the Public Metamodel* on page 1 .

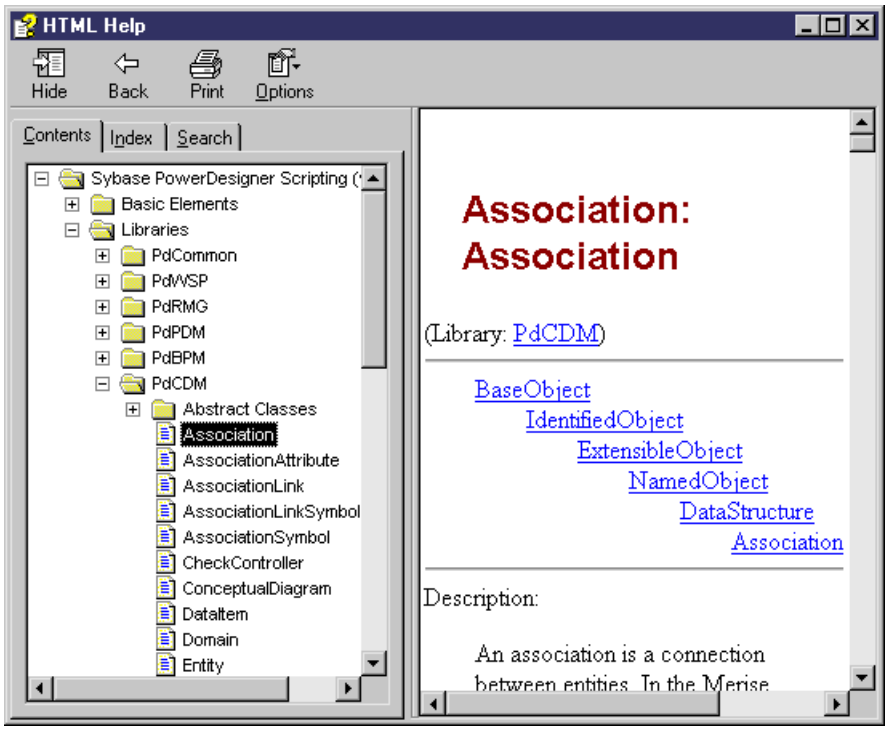
## Using the Metamodel Objects Help File

PowerDesigner provides a compiled HTML help file that you can open from the **Help** > **Metamodel Objects Help** command or from the Edit/Run Script editor dialog box. This reference guide is intended to help you get familiar with the PowerDesigner objects properties, collections and methods that you can use in scripting.

For more information on the Edit/Run Script editor, see *Using the Edit/Run Script Editor* on page 246.

## Metamodel Objects Help File Structure

The Metamodel Objects Help file is composed of two distinct parts: the node tree view displayed on the left hand side to navigate through the objects hierarchy and their corresponding description displayed to the right of the tree view:




You can expand the following nodes from the tree view:

| Nodes | What you can find... |
|---|---|
| Basic Elements | General information on:<br>Collections per type (read-only, ordered and unordered)<br>Structured Types (points, rectangles, lists of points)<br>Global properties, constants and functions |
| Libraries | PdCommon that contains: Basic object classes library used by all modules, for example File Object and Business Rules, or by at least two modules such as the Organization Unit used in the OOM and the BPM<br>PdRMG that contains Repository object classes library<br>PdWSP that contains Workspace object classes library<br>PdPRJ that contains Project object classes library<br>Object classes libraries per module (in PdCDM, PdOOM, PdBPM, PdPDM, PdXSM, PdRQM, PdILM, PdFRM, PdIAM, PdLDM and PdEAM) |
| Appendix | Hierarchical representation of the PowerDesigner metamodel<br>List of constants used to identify objects of each library |

For more information on collections, see *Collections* on page 236.

For more information on global properties, constants and functions, see *Global properties* on page 239, *Global constants* on page 242 and *Global functions* on page 241.

For more information on libraries, see *Libraries* on page 243.

### Metamodel Objects Help File Content

The scripting objects provided by PowerDesigner correspond to the design objects (tables, entities, classes, processes etc.) that appear in the user interface.

For each PowerDesigner object you can browse a list of:

- Properties (Example: Name, Data Type, Transport)
- Collections (Example: Symbols, Columns of a table)
- Methods (Example: Delete (), UpadateNamingOpts())

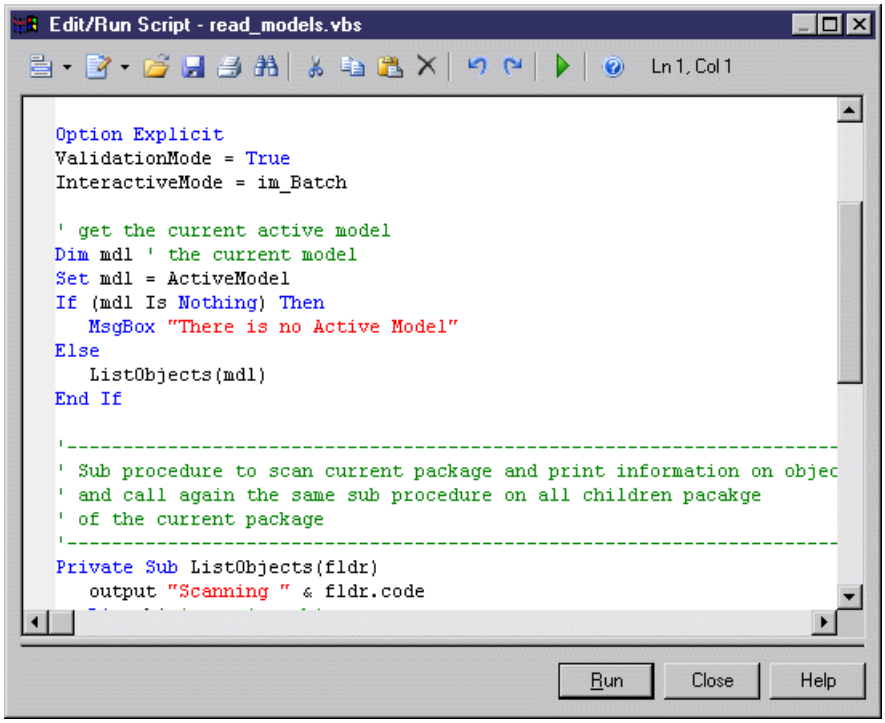
The nature of each collection is indicated: read-only, ordered, unordered, or composition.

## Using the Edit/Run Script Editor

The Edit/Run Script editor runs in the PowerDesigner environment and provides access to the scripting environment. You open it from the **Tools > Execute Commands** menu. It is available whatever the type of the active model and also when no model is active.

You can see the date and time when the script begins and ends in the Script tab of the Output window located in the lower part of the PowerDesigner main window, if you have used the Output global function.

The Edit/Run Script editor looks like the following:




The following tools and keyboard shortcuts are specific to the Edit/Run Script editor toolbar:

| Tool | Description | Keyboard shortcut |
|---|---|---|
|  | Editor Menu Note: When you use the Find feature, the parameter "Regular Expression" allows the use of wildcards in the search expression. For more information, see "Finding text using regular expressions" in the Objects chapter of the *Core Features Guide* | shift + F11 |
|  | Edit With. Opens the previously defined default editor or allows you to select another editor if you click the down arrow beside this tool | ctrl + E |

| Tool | Description | Keyboard shortcut |
|---|---|---|
|  | Run. Executes the current script | F5 |
|  | Metamodel Objects Help provided to allow you to find out which properties and methods can be used to drill down to a PowerDesigner object | ctrl + F1 |

For more information on defining a default editor, see "Specfiying text editors" in the Models chapter of the *Core Features Guide*.

### Script Bookmarks

In the Edit/Run Script editor window, you can add and remove bookmarks at specific points in the code and then navigate forwards or backwards from bookmark to bookmark:

| Keyboard shortcut | Description |
|---|---|
| ctrl + F2 | Adds a new bookmark. A blue bookmark box is displayed. If you repeat this action from the same position, the bookmark is deleted and the blue marker disappears |
| F2 | Jumps to bookmark |
| shift + F2 | Jumps to previous bookmark |

### Visual Basic

If you have Visual Basic (VB) installed on your machine, you can use the VB interface for your script writing in order to have access to the VB IntelliSense feature that checks all the standard methods and properties that you invoke and suggests the valid alternatives ones that you can choose in order to correct the code. However the PowerDesigner Edit/Run Script editor automatically recognizes VBScript keywords.

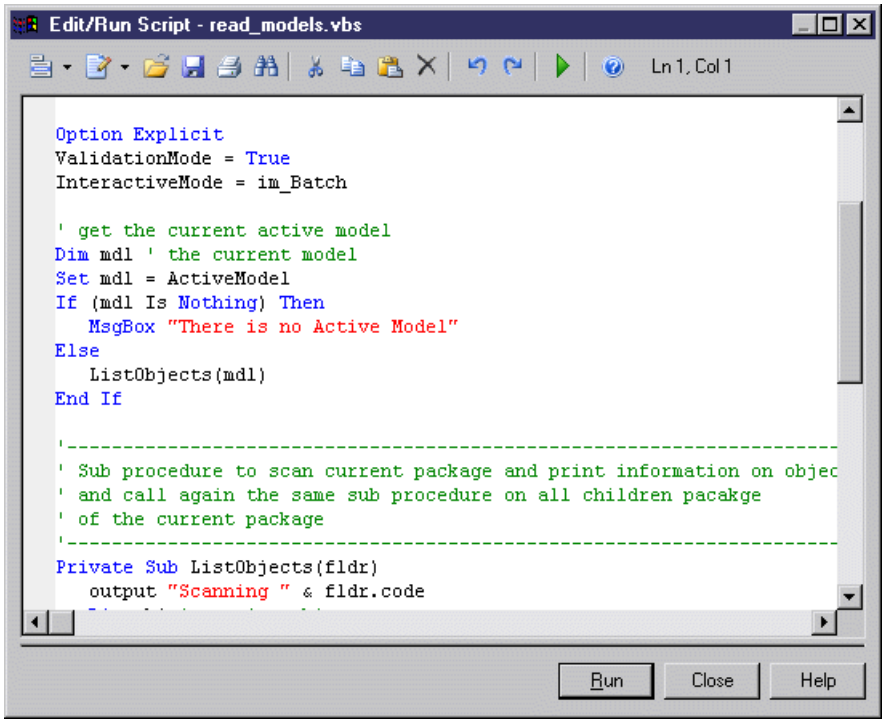
The Edit/Run Script editor lets you:

- Create a script
- Modify a script
- Save a script
- Run a script
- Use a sample script

## Creating a VBScript File

The Edit/Run Script dialog box lets you create a VBScript file.

1. Select **Tools** > **Execute Commands** > **Edit/Run Script** to display the Edit/Run Script dialog box.
2. Type the script instructions directly in the script editor window.




The script syntax is displayed as in Visual Basic.

For more information on VB syntax, see the *Microsoft Visual Basic* documentation

## Modifying a VBScript File

The Edit/Run Script dialog box lets you edit a VBScript file.

1. Open the Edit/Run Script editor.
2. Click the Open tool.

   A standard dialog box opens.
3. Select a VBScript file (.VBS) and click Open.

   The VBScript file opens in the Edit/Run Script editor window. You can then modify it.

   **Note:** You can insert a script file in a current script using the Insert command in the Editor Menu. The script will be inserted at the cursor position.

## Saving a VBScript File

It is strongly recommended to save your model and your script file before executing it.

1. Open the Edit/Run Script editor.
2. Type the script instructions directly in the script editor window.
3. Click the Editor Menu tool and select Save from the list.

   *or*

   Click the Save tool.

   A standard dialog box opens if your VBScript file has never been saved before.
4. Browse to the directory where you want to save the script file.
5. Type a name for the script file and click Save.

## Running a VBScript File

You can run a VBScript file from PowerDesigner.

Open a script and click the Run tool or the Run button.

The script is executed and the Output window located in the lower part of the PowerDesigner main window shows the execution progress if you have used the Output global function that lets you display execution progress and errors in the Script tab.

If a compilation error occurs, a message box is displayed to inform you of the kind of error. A brief description error also is displayed in the Result pane of the Edit/Run Script dialog box and the cursor is set at the error position.

The Edit/Run Script editor supports multiple levels of Undo and Redo commands. However, if you run a script that modifies objects in several models, you must use the Undo or Redo commands in each of the models called by the script.

**Note:** In order to avoid application abortions, you can catch errors using the On Error Resume Next statement. But you cannot catch errors with this statement when you use the im_Abort interactive mode.

You can also insert and customize commands in the Tools menu that will allow you to automatically launch your own scripts.

For more information on customizing commands, see *Customizing PowerDesigner Menus Using Add-Ins* on page 284.

## Using VBScript File Samples

PowerDesigner ships with a set of script samples, that you can use as a basis to create your own scripts. They are located in the VB Scripts folder of the PowerDesigner installation directory.

These scripts are intended to show you a range of the type of actions you can do over PowerDesigner objects using VBScript and also to help you in the code writing of your own scripts as you can easily copy/paste some code pieces from the sample into your script.

It is always recommended to make a backup copy of the sample file for it to remain intact.

### Model Scan Sample

The following example illustrates a script with a loop that browses a model and its sub-packages to display objects information:

```
' Scan CDM Model and display objects information
' going down each package
Option Explicit
ValidationMode = True
InteractiveMode = im_Batch
' get the current active model
Dim mdl ' the current model
Set mdl = ActiveModel
If (mdl Is Nothing) Then
 MsgBox "There is no Active Model"
Else
 Dim fldr
 Set Fldr = ActiveDiagram.Parent
 ListObjects(fldr)
End If
' Sub procedure to scan current package and print information on objects from
current package
' and call again the same sub procedure on all children package
' of the current package
Private Sub ListObjects(fldr)
 output "Scanning " & fldr.code
 Dim obj ' running object
```

```
 For Each obj In fldr.children
  ' Calling sub procedure to print out information on the object
  DescribeObject obj
 Next
 ' go into the sub-packages
 Dim f ' running folder
 For Each f In fldr.Packages
  'calling sub procedure to scan children package
  ListObjects f
 Next
End Sub
' Sub procedure to print information on current object in output
Private Sub DescribeObject(CurrentObject)
 if CurrentObject.ClassName ="Association-Class link" then exit sub
 'output "Found "+CurrentObject.ClassName
 output "Found "+CurrentObject.ClassName+" """+CurrentObject.Name+""", Created
by "+CurrentObject.Creator+" On "+Cstr(CurrentObject.CreationDate)
End Sub
```

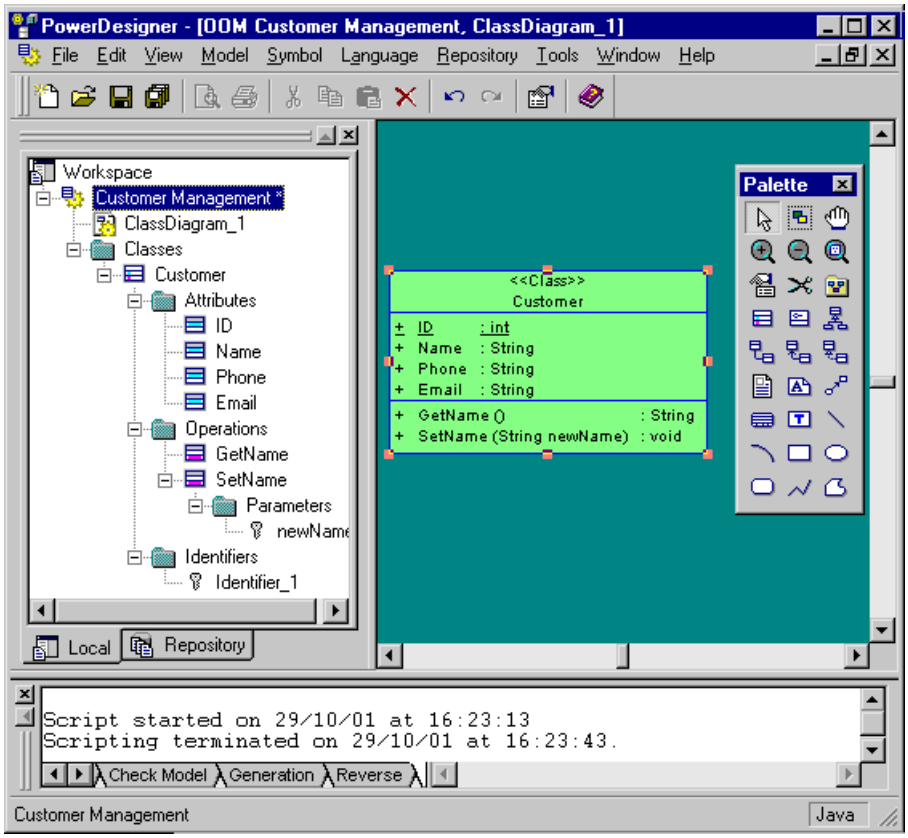
### Model Creation Sample

The following example illustrates a script that creates a new OOM model:

```
Option Explicit
' Initialization
' Set interactive mode to Batch
InteractiveMode = im_Batch
' Main function
' Create an OOM model with a class diagram
Dim Model
Set model = CreateModel(PdOOM.cls_Model, "|Diagram=ClassDiagram")
model.Name = "Customer Management"
model.Code = "CustomerManagement"
' Get the class diagram
Dim diagram
Set diagram = model.ClassDiagrams.Item(0)
' Create classes
CreateClasses model, diagram
' Create classes function
Function CreateClasses(model, diagram)
 ' Create a class
 Dim cls
 Set cls = model.CreateObject(PdOOM.cls_Class)
 cls.Name = "Customer"
 cls.Code = "Customer"
 cls.Comment = "Customer class"
 cls.Stereotype = "Class"
 cls.Description = "The customer class defines the attributes and behaviors of
a customer."
 ' Create attributes
 CreateAttributes cls
 ' Create methods
 CreateOperations cls
 ' Create a symbol for the class
 Dim sym
 Set sym = diagram.AttachObject(cls)
 CreateClasses = True
End Function
' Create attributes function
Function CreateAttributes(cls)
 Dim attr
 Set attr = cls.CreateObject(PdOOM.cls_Attribute)
 attr.Name = "ID"
 attr.Code = "ID"
 attr.DataType = "int"
```

```
 attr.Persistent = True
 attr.PersistentCode = "ID"
 attr.PersistentDataType = "I"
 attr.PrimaryIdentifier = True
 Set attr = cls.CreateObject(PdOOM.cls_Attribute)
 attr.Name = "Name"
 attr.Code = "Name"
 attr.DataType = "String"
 attr.Persistent = True
 attr.PersistentCode = "NAME"
 attr.PersistentDataType = "A30"
 Set attr = cls.CreateObject(PdOOM.cls_Attribute)
 attr.Name = "Phone"
 attr.Code = "Phone"
 attr.DataType = "String"
 attr.Persistent = True
 attr.PersistentCode = "PHONE"
 attr.PersistentDataType = "A20"
 Set attr = cls.CreateObject(PdOOM.cls_Attribute)
 attr.Name = "Email"
 attr.Code = "Email"
 attr.DataType = "String"
 attr.Persistent = True
 attr.PersistentCode = "EMAIL"
 attr.PersistentDataType = "A30"
 CreateAttributes = True
End Function
' Create operations function
Function CreateOperations(cls)
 Dim oper
 Set oper = cls.CreateObject(PdOOM.cls_Operation)
 oper.Name = "GetName"
 oper.Code = "GetName"
 oper.ReturnType = "String"
 Dim body
 body = "{" + vbCrLf
 body = body + " return Name;" + vbCrLf
 body = body + "}"
 oper.Body = body
 Set oper = cls.CreateObject(PdOOM.cls_Operation)
 oper.Name = "SetName"
 oper.Code = "SetName"
 oper.ReturnType = "void"
 Dim param
 Set param = oper.CreateObject(PdOOM.cls_Parameter)
 param.Name = "newName"
 param.Code = "newName"
 param.DataType = "String"
 body = "{" + vbCrLf
 body = body + " Name = newName;" + vbCrLf
 body = body + "}"
 oper.Body = body
 CreateOperations = True
End Function
```

The previous script gives the following result in the interface:




# Basic Scripting Tasks

You can use scripts to create and open models, and to manipulate objects and symbols in PowerDesigner.

## Creating a Model by Script

You create a model using the CreateModel (modelkind As Long, filename As String = "", flags As Long =omf_Default) As BaseObject global function together with the cls_Model constant prefixed with the Module name to identify the type of model you want to create.

Note that additional arguments may be specified in the filename parameter depending on the type of model (Language, DBMS, Copy, Diagram). The diagram argument uses the public name, but the localized name (the one in the Target selection dialog box) is also accepted. However, it is not recommended to use the localized name as your script will only work for the localized version of PowerDesigner.

### Example

```
Option Explicit
' Call the CreateModel global function with the following parameters:
'   - The model kind is an Object Oriented Model (PdOOM.Cls_Model)
'   - The Language is enforced to be Analysis
'   - The first diagram will be a class diagram
'   - The language definition (for Analysis) is copied inside the model
'   - The first diagram will not be opened in a window
'   - The new created model will not appear in the workspace
Dim NewModel
set NewModel = CreateModel(PdOOM.Cls_Model, "Language=Analysis|
Diagram=ClassDiagram|Copy", omf_DontOpenView Or omf_Hidden)
If NewModel is Nothing then
 msgbox "Fail to create UML Model", vbOkOnly, "Error"  ' Display an error
message box
```

```
Else
 output "The UML model has been successfully created"  ' Display a message in
the application output window
' Initialize model name and code
 NewModel.Name = "Sample Model"
 NewModel.Code = "Sample"
' Save the new model in a file
 NewModel.Save "c:\temp\MySampleModel.oom"
' Close the model
 NewModel.Close
' Release last reference to the model object to free memory
 Set NewModel = Nothing
End If
```

## Opening a Model by Script

You open a model using the OpenModel (filename As String, flags As Long =omf_Default) As BaseObject global function.

### Example

```
Option Explicit
' Call the OpenModel global function with the following parameters:
'  - The model file name
'  - The default diagram will not be opened in a window
'  - The opened model will not appear in the workspace
Dim ExistingModel, FileName
FileName = "c:\temp\MySampleModel.oom"
On Error Resume Next    ' Avoid generic scripting error message like 'Invalid
File Name
Set ExistingModel = OpenModel(FileName, omf_DontOpenView Or omf_Hidden)
On Error Goto 0     ' Restore runtime error detection
If ExistingModel is nothing then
 msgbox "Fail to open UML Model:" + vbCrLf + FileName, vbOkOnly, "Error"  '
Display an error message box
Else
 output "The UML model has been successfully opened"  ' Display a message in
the application output window
End If
```

## Creating an Object by Script

It is recommended to create an object directly from the collection to which it belongs in order to directly obtain a valid state for the object. When you do so, you only create the object but not its graphical symbol.

You can also use the following method: CreateObject(ByVal Kind As Long, ByVal ParentCol As String = "", ByVal Pos As Long = -1, ByVal Init As Boolean = -1) As BaseObject

### Creating an Object in a Model

```
If not ExistingModel is Nothing Then
' Call the CreateNew() method on the collection that owns the object
 Dim MyClass
 Set MyClass = ExistingModel.Classes.CreateNew()
 If MyClass is nothing Then
  msgbox "Fail to create a class", vbOkOnly, "Error"  ' Display an error
message box
 Else
  output "The class objects has been successfully created"  ' Display a
message in the application output window
   ' Initialize its name and code using a specific method
  ' that ensures naming conventions (Uppercase or lowercase constraints,
```

```
  ' invalid characters...) are respected and that the name and code
  ' are unique inside the model
  MyClass.SetNameAndCode "Customer", "cust"
  ' Initialize other properties directly
  MyClass.Comment = "Created by script"
  MyClass.Stereotype = "MyStereotype"
  MyClass.Final = true
  ' Create an attribute inside the class
  Dim MyAttr
  Set MyAttr = MyClass.Attributes.CreateNew()
  If not MyAttr is nothing Then
   output "The class attribute has been successfully created"
   MyAttr.SetNameAndCode "Name", "custName"
   MyAttr.DataType = "String"
   Set MyAttr = Nothing
  End If
  ' Reset the variable in order to avoid memory leaks
  Set MyClass = Nothing
 End If
End If
```

### Creating an Object in a Package

```
If not ExistingModel is Nothing Then
 ' Create a package first
 Dim MyPckg
 Set MyPckg = ExistingModel.Packages.CreateNew()
 If not MyPckg is Nothing then
  output "The package has been successfully created"
  MyPckg.SetNameAndCode "All interfaces", "intf"
  ' Create an interface object inside the package
  Dim MyIntf
  Set MyIntf = MyPckg.Interfaces.CreateNew()
  If not MyIntf is Nothing then
   output "The interface object has been successfully created inside the
package"
   MyIntf.SetNameAndCode "Customer Interface", "custIntf"
   Set MyIntf = Nothing
  End If
  Set MyPckg = Nothing
 End If
End If
```

## Creating a Symbol by Script

You create the associated symbol of an object by attaching it to the active diagram using the following method: AttachObject(ByVal Obj As BaseObject) As BaseObject.

### Example

```
set symbol1 = ActiveDiagram.AttachObject(entity1)
```

**Note:** The AttachObject method can also be used to create a graphical synonym or a shortcut. For more information, see sections on graphical synonym and shortcut creation.

## Displaying an Object Symbol by Script

You can display objects symbol in a diagram using the following methods:

- AttachObject(ByVal Obj As BaseObject) As BaseObject to create a symbol for a non-link object

- AttachLinkObject(ByVal Link As BaseObject, ByVal Sym1 As BaseObject = NULL, ByVal Sym2 As BaseObject = NULL) As BaseObject to create a symbol for a link object
- AttachAllObjects() As Boolean to create a symbol for each object in package which can be displayed in current diagram

**Example**

```
If not ExistingModel Is Nothing and not MyRealization Is Nothing Then
 ' Symbols are specific kind of objects that can be manipulated by script
 ' We are now going to display the class, interface and realization in the
 ' main diagram of the model and customize their presentation
 ' Retrieve main diagram
 Dim MyDiag
 Set MyDiag = ExistingModel.DefaultDiagram
 If not MyDiag is Nothing and MyDiag.IsKindOf(PdOOM.Cls_ClassDiagram) Then
  ' Display the class, interface shortcut and realization link in the diagram
  ' using default positions and display preferences
  Dim MyClassSym, MyIntfSym, MyRlzsSym
  Set MyClassSym = MyDiag.AttachObject(FoundClass)
  Set MyIntfSym  = MyDiag.AttachObject(IntfShct)
  Set MyRlzsSym  = MyDiag.AttachLinkObject(MyRealization, MyClassSym,
MyIntfSym)
  If not MyRlzsSym is Nothing Then
   output "Objects have been successfully displayed in diagram"
  End If
  ' Another way to do the same is the use of AttachAllObjects() method:
  ' MyDiag.AttachAllObjects
  ' Changes class symbol format
  If not MyClassSym is nothing Then
   MyClassSym.BrushStyle = 1 ' Solid background (no gradient)
   MyClassSym.FillColor = RGB(255, 126, 126) ' Red background color
   MyClassSym.LineColor = RGB(0, 0, 0) ' Black line color
   MyClassSym.LineWidth = 2 ' Double line width
   Dim Fonts
   Fonts = "ClassStereotype " + CStr(RGB(50, 50, 126)) + " Arial,8,I"
   Fonts = Fonts + vbCrLf + "DISPNAME " + CStr(RGB(50, 50, 50)) + " Arial,
12,B"
   Fonts = Fonts + vbCrLf + "ClassAttribute " + CStr(RGB(150, 0, 0)) + " Arial,
8,N"
   MyClassSym.FontList = Fonts ' Change font list
  End If
  ' Changes interface symbol position
  If not MyIntfSym is nothing Then
   Dim IntfPos
   Set IntfPos = MyIntfSym.Position
   If not IntfPos is Nothing Then
    IntfPos.x = IntfPos.x + 5000
    IntfPos.y = IntfPos.y + 5000
    MyIntfSym.Position = IntfPos
    Set IntfPos = Nothing
   End If
  End If
  ' Changes the link symbol corners
  If not MyRlzsSym is Nothing Then
   Dim CornerList, Point1, Point2
   Set CornerList = MyRlzsSym.ListOfPoints
   Set Point1 = CornerList.Item(0)
   Set Point2 = CornerList.Item(1)
   CornerList.InsertPoint 1, Max(Point1.x, Point2.x), Min(Point1.y, Point2.y)
   Set CornerList = Nothing
   ' Max and Min are functions defined at end of this script
  End If
  ' Release the variables
  Set MyDiag = Nothing
```

```
  Set MyClassSym = Nothing
  Set MyIntfSym = Nothing
  Set MyRlzsSym = Nothing
 End If
End If
```

## Positioning a Symbol next to Another by Script

You position a symbol next to another using the X and Y (respectively Abscissa and Ordinate) points, together with a combination of method (Position As Apoint) and function (NewPoint(X As Long = 0, Y As Long = 0) As Apoint)).

### Example

```
Dim diag
Set diag = ActiveDiagram
Dim sym1, sym2
Set sym1 = diag.Symbols.Item(0)
Set sym2 = diag.Symbols.Item(1)
X1 = sym1.Position.X
Y1 = sym1.Position.Y
' Move symbols next to each other using a fixed arbitrary space
sym2.Position = NewPoint(X1+5000, Y1)
' Move symbols for them to be adjacent
sym2.Position = NewPoint(X1 + (sym1.Size.X+sym2.Size.X)/2, Y1)
```

## Deleting an Object by Script

You delete an object from a model using the Delete As Boolean method.

### Example

```
If not ExistingModel is Nothing Then
 ' Create another class first
 Dim MyClassToDelete
 Set MyClassToDelete = ExistingModel.Packages.CreateNew()
 If not MyClassToDelete is Nothing then
  output "The second class has been successfully created"
  ' Just call Delete method to delete the object
  ' This will remove the object from the collection of model classes
  MyClassToDelete.Delete
  ' The object is still alive but it has notified all other
  ' objects of its deletion. It is no more associated with other objects.
  ' Its status is deleted
  If MyClassToDelete.IsDeleted() Then
   output "The second class has been successfully deleted"
  End If
  ' The reset of the VbScript variable will release the last
  ' reference to this object abd provoke the physical destruction
  ' and free the memory
  Set MyClassToDelete = Nothing
 End If
End If
```

## Retrieving an Object by Script

The following example illustrates how you can retrieve an object by its code in the model

### Example

```
' Call a function that is implemented just after in the script
Dim FoundIntf, FoundClass
```

```
Set FoundIntf = RetrieveByCode(ExistingModel, PDOOM.Cls_Interface, "custIntf")
Set FoundClass = RetrieveByCode(ExistingModel, PDOOM.Cls_Class, "cust")
If (not FoundIntf is nothing) and (not FoundClass is Nothing) Then
 output "The class and interface objects have been successfully retrieved by
their code"
End If
' Implement a method that retrieve an object by code
' The first parameter is the root folder on which the research begins
' The second parameter is the kind of object we are looking for
' The third parameter is the code of the object we are looking for
Function RetrieveByCode(RootObject, ObjectKind, CodeValue)
 ' Test root parameter
 If RootObject is nothing Then
  Exit Function       ' Root object is not defined
 End If
 If RootObject.IsShortcut() Then
  Exit Function       ' Root object is a shortcut
 End If
 If not RootObject.IsKindOf(Cls_BaseFolder) Then
  Exit Function       ' Root object is not a folder
 End If
 ' Loop on all objects in folder
 Dim SubObject
 For Each SubObject in RootObject.Children
  If SubObject.IsKindOf(ObjectKind) and SubObject.Code = CodeValue Then
   Set RetrieveByCode = SubObject   ' Initialize return value
   Set SubObject = Nothing
   Exit Function
  End If
 Next
 Set SubObject = Nothing
 ' Recursive call on sub-folders
 Dim SubFolder
 For Each SubFolder in RootObject.CompositeObjects
  If not SubFolder.IsShortcut() Then
   Dim Found
   Set Found = RetrieveByCode(SubFolder, ObjectKind, CodeValue)
   If not Found Is Nothing Then
    Set RetrieveByCode = Found   ' Initialize return parameter
    Set Found = Nothing
    Set SubFolder = Nothing
    Exit Function
   End If
  End If
 Next
 Set SubFolder = Nothing
End Function
```

## Creating a Shortcut by Script

You create a shortcut in a model using the CreateShortcut(ByVal NewPackage As BaseObject, ByVal ParentCol As String = "") As BaseObject method.

### Example

```
' We want to reuse at the model level the interface defined in the package
' To do that, we need to create a shortcut of the interface at the model level
Dim IntfShct
If not FoundIntf is Nothing and not ExistingModel Is Nothing Then
 ' Call the CreateShortcut() method and specify the model
 ' for the package where we want to create the shortcut
 Set IntfShct = FoundIntf.CreateShortcut(ExistingModel)
 If not IntfShct is nothing then
```

```
  output "The interface shortcut has been successfully created"
 End If
End If
```

## Creating a Link Object by Script

You create a link object using the CreateNew(kind As Long = 0) As BaseObject method, then you have to declare its ends.

### Example

```
Dim MyRealization
If (not ExistingModel Is Nothing) and (not FoundClass Is Nothing) and (not
IntfShct is Nothing) Then
 ' We are now going to create a realization link between the class and the
interface
 ' The link is an object like others with two mandatory attributes: Object1
and Object2
 ' For oriented links, Object1 is the source and Object2 is the destination
 Set MyRealization = ExistingModel.Realizations.CreateNew()
 If not MyRealization is Nothing then
  output "The realization link has been successfully created"
  ' Initialize both extremities
  Set MyRealization.Object1 = FoundClass
  Set MyRealization.Object2 = IntfShct
  ' Initialize Name and Code
  MyRealization.SetNameAndCode "Realize Main interface", "Main"
 End If
End If
```

## Browsing a Collection by Script

All collections can be iterated through the usual "For Each variable In collection" construction.

This loop starts with "For each <variable> in <collection>" and ends with "Next".

The loop is iterated on each object of the collection. The object is available in <variable>.

### Example

```
'How to browse the collection of tables available on a model
Set MyModel = ActiveModel
'Assuming MyModel is a variable containing a PDM object.
For each T in MyModel.Tables
 'Variable T now contains a table from Tables collection of the model
 Output T.name
Next
```

## Manipulating Objects in a Collection by Script

In the following example, we are going to manipulate objects in collections by creating business rule objects and attaching them to a class object. To do so, we :

- Create the business rule objects
- Initialize their attributes
- Retrieve the first object in the class attributes collection
- Add the created rules at the beginning and at the end of the attached rules collection
- Move a rule at the end of the the attached rules collection
- Remove a rule from the attached rules collection

### Example

```
If (not ExistingModel Is Nothing) and (not FoundClass Is Nothing) Then
 ' We are going to create business rule objects and attached them to the class
 ' Create first the business rule objects
 Dim Rule1, Rule2
 Set Rule1 = ExistingModel.BusinessRules.CreateNew()
 Set Rule2 = ExistingModel.BusinessRules.CreateNew()
 If (not Rule1 is Nothing) And (not Rule2 Is Nothing) Then
  output "Business Rule objects have been successfully created"
  ' Initialize rule attributes
  Rule1.SetNameAndCode "Mandatory Name", "mandatoryName"
  Rule1.ServerExpression = "The Name attribute cannot be empty"
  Rule2.SetNameAndCode "Unique Name", "uniqueName"
  Rule2.ServerExpression = "The Name attribute must be unique"
  ' Retrieve the first object in the class attributes collection
  Dim FirstAttr, AttrColl
  Set AttrColl = FoundClass.Attributes
  If not AttrColl is Nothing Then
   If not AttrColl.Count = 0 then
    Set FirstAttr = AttrColl.Item(0)
   End If
  End If
  Set AttrColl = Nothing
  If not FirstAttr is Nothing Then
   output "First class attribute successfully retrieved from collection"
   ' Add Rule1 at end of attached rules collection
   FirstAttr.AttachedRules.Add Rule1
   ' Add Rule2 at the beginning of attached rules collection
   FirstAttr.AttachedRules.Insert 0, Rule2
   ' Move Rule2 at end of collection
   FirstAttr.AttachedRules.Move 1, 0
   ' Remove Rule1 from collection
   FirstAttr.AttachedRules.RemoveAt 0
   Set FirstAttr = Nothing
  End If
 End If
  Set Rule1 = Nothing
 Set Rule2 = Nothing
End If
```

## Extending the Metamodel by Script

When you import a file using scripts, you can import as extended attributes or extended collections some properties that may not correspond to standard attributes. In the following example, we:

- Create a new extended model definition
- Initialize model extension attributes
- Define a new stereotype for the Class metaclass in the profile section
- Define an extended attribute for this stereotype

### Example

```
If not ExistingModel Is Nothing Then
 ' Creating a new extended model definition
 Dim ModelExtension
 Set ModelExtension = ExistingModel.ExtendedModelDefinitions.CreateNew()
 If not ModelExtension is Nothing Then
  output "Model extension successfully created in model"
  ' Initialize model extension attributes
  ModelExtension.Name = "Extension for Import of XXX files"
  ModelExtension.Code = "importXXX"
  ModelExtension.Family = "Import"
```

```
  ' Defines a new Stereotype for the Class metaclass in the profile section
  Dim MySttp
  Set MySttp = ModelExtension.AddMetaExtension(PdOOM.Cls_Class,
Cls_StereotypeTargetItem)
  If not MySttp Is Nothing Then
   output "Stereotype extension successfully created in extended model
definition"
   MySttp.Name = "MyStereotype"
   MySttp.UseAsMetaClass = true ' The stereotype will behave as a new
metaclass (specific list and category in browser)
   ' Defines an extended attribute for this stereotype
   Dim MyExa
   Set MyExa = MySttp.AddMetaExtension(Cls_ExtendedAttributeTargetItem)
   If not MyExa is Nothing Then
    output "Extended Attribute successfully created in extended model
definition"
    MyExa.Name = "MyAttribute"
    MyExa.Comment = "custom attribute coming from import"
    MyExa.DataType = 10  ' This corresponds to integer
    MyExa.Value = "-1" ' This is the default value
    Set MyExa = Nothing
   End If
   ' Defines an extended collection for this stereotype
   Dim MyExCol
   Set MyExCol = MySttp.AddMetaExtension(Cls_ExtendedCollectionTargetItem)
   If not MyExCol is Nothing Then
    output "Extended collection successfully created in extended model
definition"
    MyExCol.Name = "MyCollection"
    MyExCol.Comment = "custom collection coming from import"
    MyExCol.DestinationClassKind = PdOOM.Cls_class  ' The collection can store
only classes
    MyExCol.Destinationstereotype = "MyStereotype"  ' The collection can store
only classes with stereotype "MyStereotype"
    Set MyExCol = Nothing
   End If
   Set MySttp = Nothing
  End If
  Set ModelExtension = Nothing
 End If
End If
```

## Manipulating Extended Properties by Script

You can dynamically get and set objects extended properties like attributes and collections using scripts.

The syntax for identifying any object property is:

```
"<TargetCode>.<PropertyName>"
```

For example, to get the extended attribute MyAttribute from the importXXX object, use:

```
GetExtendedAttribute("importXXX.MyAttribute")
```

Note that if the script is inside a profile (for example, in a custom check script), you can use the %CurrentTargetCode% variable instead of a hard-coded TargetCode, in order to improve the portability of your script.

For example:

```
GetExtendedAttribute("%CurrentTargetCode%.MyAttribute")
```

In the following example we:

- Modify extended attribute on the class
- Modify extended collection on the class

- Add the class in its own extended collection to be used as a standard collection

**Example**

```
If (not ExistingModel Is Nothing) and (not FoundClass Is Nothing) Then
 ' Modify extended attribute on the class
 Dim ExaName
 ExaName = "importXXX.MyAttribute"  ' attribute name prefixed by extended
model definition code
 If FoundClass.HasExtendedAttribute(ExaName) Then
  output "Extended attribute can be accessed"
  FoundClass.SetExtendedAttributeText ExaName, "1024"
  FoundClass.SetExtendedAttribute ExaName, 2048
  Dim valAsText, valAsInt
  valAsText = FoundClass.GetExtendedAttributeText(ExaName)
  valAsInt = FoundClass.GetExtendedAttribute(ExaName)
 End If
 ' Modify extended collection on the class
 Dim ExColName, ExCol
 ExColName = "importXXX.MyCollection"  ' collection name prefixed by extended
model definition code
 Set ExCol = FoundClass.GetExtendedCollection(ExColName)
 If not ExCol is Nothing Then
  output "Extended collection can be accessed"
  ' The extended collection can be used as a standard collection
  ' for example, we add the class in its own extended collection
  ExCol.Add FoundClass
  Set ExCol = Nothing
 End If
End If
```

## Creating a Graphical Synonym by Script

You create a graphical synonym by attaching the same object twice to the same package.

**Example**

```
set diag = ActiveDiagram
set pack = ActivePackage
set class = pack.classes.createnew
set symbol1 = diag.AttachObject (class)
set symbol2 = diag.AttachObject (class)
```

## Creating an Object Selection by Script

Object Selection is a model object that is very useful to select other model objects in order to apply to them a specific treatment. You can for example add some objects to the Object Selection to move them to another package in a unique operation instead of repeating the same operation for each and every objects individually.

When dealing with a set of objects in the user interface, you use the Object Selection in scripting.

- Create Object Selection

You create the Object Selection from a model using the CreateSelection method: CreateSelection() As BaseObject.

**Example**

```
Set MySel = ActiveModel.CreateSelection
```

- Add objects individually

You can add objects individually by adding the required object to the Objects collection.

You use the Object Selection following method: Add(obj As BaseObject)

### Example

Adding of an object named Publisher:

```
MySel.Objects.Add(Publisher)
```

- Add objects of a given type

You can add all objects of a given type by using the Object Selection following method: AddObjects(ByVal RootPackage As BaseObject, ByVal ClassType As Long, ByVal IncludeShortcuts As Boolean = 0, ByVal Recursive As Boolean = 0).

RootPackage is the package from which to add objects.

ClassType is the type of object to add.

IncludeShortcuts is the parameter to include shortcuts.

Recursive is the parameter to search in all the sub-packages.

### Example

An adding of classes with no inclusion of shortcuts and no recursiveness into the sub-packages:

```
MySel.AddObjects(folder,cls_class)
```

- Remove objects from the current selection

You can remove objects from the current selection using the Object Selection following method: RemoveObjects(ByVal ClassType As Long, ByVal IncludeShortcuts As Boolean = -1)

### Example

Withdrawal of all classes and shortcuts from the Object Selection:

```
MySel.RemoveObjects(cls_class, -1)
```

- Move objects of the current selection to a destination package

You can move objects of the current selection to a destination package using the Object Selection following method: MoveToPackage(ByVal TargetPackage As BaseObject)

### Example

Move of objects of the selection to a destination package named Pack:

```
MySel.MoveToPackage Pack
```

- Copy objects of the current selection to a destination package

You can copy objects of the current selection to a destination package using the Object Selection following method: CopyToPackage(ByVal TargetPackage As BaseObject)

### Example

Copy of objects of the selection in a destination package named Pack:

```
MySel.CopyToPackage Pack
```

- Filter a selection list by stereotype

You can create an object selection and filter this selection using a stereotype. You have to use the following method:

ShowObjectPicker(ByVal ClassNames As String = "", ByVal StereotypeFilter As String = "", ByVal DialogCaption As String = "", ByVal ShowEmpty As Boolean = True, ByVal InModel As Boolean = True) As BaseObject

### Example

Opens a selection dialog box for selecting a business transaction:

```
If Not Fldr is Nothing then
   ' Create a selection object
   Set Sel = Mdl.CreateSelection
   If Not Sel is Nothing then
     'Show the object picker dialog for selecting a BT
    Set Impl = Sel.ShowObjectPicker ("Process", "BinaryCollaboration", "Select
a Binary Collaboration Process")
    ' Retrieve the selection
    If not Impl is Nothing Then
     If Impl.IsKindOf(PDBPM.Cls_Process) and Impl.Stereotype =
"BinaryCollaboration" then
      Set Shct = Impl.CreateShortcut (Fldr)
      If not Shct is Nothing Then
       obj.Implementer = Shct
        %Initialize% = True
      End If
     End If
     End If
   End If
```

## Creating an Extended Model Definition by Script

As any PowerDesigner object, extended model definitions can be read, created and modified using scripting.

For more information on extended model definitions, see *Extended Model Definitions* on page 18.

The following script illustrates how you can *access* an existing extended model definition, *browse* it , *create* an extended attribute within the definition and at last *modify* the extended attribute values. A function is created to drill down the categories tree view that is displayed in the Extended Model Definition Properties dialog box.

### Example

```
Dim M
Set M = ActiveModel
'Retrieve first extended model definition in the active model
Dim X
Set X = M.ExtendedModelDefinitions.Item(0)
'Drill down the categories tree view using the searchObject function (see
below for details)
Dim C
Set C = SearchObject (X.Categories, "Settings")
Set C = SearchObject (C.Categories, "Extended Attributes")
Set C = SearchObject (C.Categories, "Objects")
Set C = SearchObject (C.Categories, "Entity")
'Create extended attribute in the Entity category
Dim A
Set A = C.Categories.CreateNew (cls_ExtendedAttributeTargetItem)
'Define properties of the extended attribute
A.DataType = 10 'integer
A.Value = 10
A.Name = "Z"
A.Code = "Z"
'Retrieve first entity in the active model
Dim E
Set E = M.entities.Item(0)
'Retrieve the values of the created extended attribute in a message box
msgbox E.GetExtendedAttribute("X.Z")
'Changes the values of the extended attribute
E.SetExtendedAttribute "X.Z", 5
'Retrieve the modified values of the extended attribute in a message box
```

```
msgbox E.GetExtendedAttribute("X.Z")
********************
'Detail SearchObject function that allows you to browse a collection from its
name and the searched object
'* SUB SearchObject
Function SearchObject (Coll, Name)
'For example Coll = Categories and Name = Settings
 Dim Found, Object
 For Each Object in Coll
  If Object.Name = Name Then
   Set Found = Object
  End If
 Next
 Set SearchObject = Found
End Function
```

## Mapping Objects by Script

You can use scripting to map objects from heterogeneous models.

You create or reuse a mapping for an object using the following method on the DataSource object and on the ClassifierMap object: CreateMapping(ByVal Object As BaseObject) As BaseObject.

### Example

Given the following example where an OOM (oom1) contains a class (class_1) with two attributes (att1 and att2) and a PDM (pdm1) contains a table (table_1) with two columns (col1 and col2). To map the OOM class and attributes to the PDM table and columns, you have to do the following:

- Create a data source in the OOM

```
set ds = oom1.datasources.createnew
```

- Add the PDM as source for the data source

```
ds.AddSource pdm1
```

- Create a mapping for class_1 and set this mapping as the default for class_1 (current data source being the default)

```
set map1 = ds.CreateMapping(class_1)
```

- Add table_1 as source for class_1

```
map1.AddSource table_1
```

- Add a mapping for att1

```
set attmap1 = map1.CreateMapping(att1)
```

- Set col1 as source for att1

```
attmap1.AddSource col1
```

- Add a mapping for att2

```
set attmap2 = map1.CreateMapping(att2)
```

- Set col2 as source for att2

```
attmap.AddSource col2
```

You can also get the mapping of an object using the following method on the DataSource object and on the ClassifierMap object: GetMapping(ByVal Object As BaseObject) As BaseObject.

- Get the mapping of class_1

```
Set mymap = ds.GetMapping (class_1)
```

- Get the mapping of att1

```
Set mymap = map1.GetMapping (att1)
```

For more information about objects mapping, see the Mapping Objects chapter in the *Core Feature's Guide*.

# Manipulating Databases by Script

You can use scripts to manipulate databases in PowerDesigner.

## Generating a Database by Script

When you need to generate a database using script, you may use the following methods:

- GenerateDatabase(ByVal ObjectSelection As BaseObject = Nothing)
- GenerateTestData(ByVal ObjectSelection As BaseObject = Nothing)

In the following example, you:

- Open an existing model.
- Generate a script for the model.
- Modify the model.
- Generate a modified database script.
- Generate a set of test data.

### Opening an Existing Model

In the following example, we begin with opening an existing model (ASA 9) using the following method: OpenModel (filename As String, flags As Long =omf_Default) As BaseObject.

Be sure to add a final backslash (\) to the generation directory.

Then we are going to generate a database script for the model, modify the model, generate a modified data script, and generate a set of test data using respectively the following methods:

- GenerateDatabaseScripts pModel
- ModifyModel pModel
- GenerateAlterScripts pModel
- GenerateTestDataScript pModel

Example

```
Option Explicit
Const GenDir = "D:\temp\test\"
Const Modelfile = "D:\temp\phys.pdm"
Dim fso : Set fso = CreateObject("Scripting.FileSystemObject")
Start
Sub Start()
 dim pModel : Set pModel = OpenModel(Modelfile)
 If (pModel is Nothing) then
  Output "Unable to open the model"
  Exit Sub
 End if
End Sub
```

### Generating a Script for the Model

Then you generate a script for this model in the folder defined in the "GenDir" constant using the following method: GenerateDatabase(ByVal ObjectSelection As BaseObject = Nothing).

As you would do in the generation database dialog box, you have to define the generation directory and the sql file name before starting the generation, see the following example.

Example

```
Sub GenerateDatabaseScripts(pModel)
 Dim pOpts : Set pOpts = pModel.GetPackageOptions()
 InteractiveMode = im_Batch ' Avoid displaying generate window
 ' set generation options using model package options
 pOpts.GenerateODBC = False ' Force sql script generation rather than
 ' ODBC
 pOpts.GenerationPathName = GenDir ' Define generation directory
 pOpts.GenerationScriptName = "script.sql" ' Define sql file name
 pModel.GenerateDatabase ' Launch the Generate Database feature
End Sub
```

### Modifying the Model

After, you modify the model by adding a column to each table:

Example

```
Sub ModifyModel(pModel)
 dim pTable, pCol
 ' Add a new column in each table
 For each pTable in pModel.Tables
  Set pCol = pTable.Columns.CreateNew()
  pCol.SetNameAndCode "az" & pTable.Name, "AZ" & pTable.Code
  pCol.Mandatory = False
 Next
End Sub
```

### Generating a Modified Database Script

Before generating the modified database script, you have to get package option and change generation parameters, then you generate the modified database script accordingly.

For more information about the generation options, see section BasePhysicalPackageOptions in the Metamodel Object Help file.

Example

```
Sub GenerateAlterScripts(pModel)
 Dim pOpts : Set pOpts = pModel.GetPackageOptions()
 InteractiveMode = im_Batch ' Avoid displaying generate window
' set generation options using model package options
 pOpts.GenerateODBC = False ' Force sql script generation rather than ODBC
pOpts.GenerationPathName = GenDir
 pOpts.DatabaseSynchronizationChoice = 0 'force already saved apm as source
 pOpts.DatabaseSynchronizationArchive = GenDir & "model.apm"
 pOpts.GenerationScriptName = "alter.sql"
pModel.ModifyDatabase ' Launch the Modify Database feature
End Sub
```

### Generating a Set of Test Data

Finally, you generate a set of test data:

Example

```
Sub GenerateTestDataScript(pModel)
 Dim pOpts : Set pOpts = pModel.GetPackageOptions()
 InteractiveMode = im_Batch ' Avoid displaying generate window
' set generation options using model package options
 pOpts.TestDataGenerationByODBC = False ' Force sql script generation rather
than ODBC
 pOpts.TestDataGenerationDeleteOldData = False
pOpts.TestDataGenerationPathName = GenDir
 pOpts.TestDataGenerationScriptName = "Test.sql"
```

```
pModel.GenerateTestData ' Launch the Generate Test Data feature
End Sub
```

## Generating a Database Via a Live Connection by Script

You can generate a database via ODBC using script.

To do so, you first begin with connecting to the database using the ConnectToDatabase(ByVal Dsn As String, ByVal User As String, ByVal Password As String) As Boolean method from the model, then you set up the generation options and launch the generation feature.

For more information about the generation options, see section BasePhysicalPackageOptions in the Metamodel Object Help file.

Example:

```
Const cnxDSN = "ODBC:ASA 9.0 sample"
Const cnxUSR = "dba"
Const cnxPWD = "sql"

Const GenDir = "C:\temp\"
Const GenFile = "test.sql"
Const ModelFile = "C:\temp\phys.pdm"

set pModel = openModel(ModelFile)

 set pOpts=pModel.GetPackageOptions()

  pModel.ConnectToDatabase cnxDSN, cnxUSR, cnxPWD
  pOpts.GenerateODBC = true

 pOpts.GenerationPathName = GenDir
 pOpts.GenerationScriptName = 'script.sql"
 pModel.GenerateDatabase
```

## Generating a Database Using Setting and Selection

You can use settings and selections with scripting before starting the database generation using respectively the following methods from the model: UseSettings(ByVal Function As Long, ByVal Name As String = "") As Boolean and UseSelection(ByVal Function As Long, ByVal Name As String = "") As Boolean.

Given the PDM sample (Project.PDM) in the PowerDesigner installation folder, which contains two selections:

- "Organization" selection includes tables DIVISION, EMPLOYEE, MEMBER & TEAM.
- "Materials" selection includes tables COMPOSE, MATERIAL, PROJECT & USED.

The following example shows you how to

- Generate a first script of this model for the "Organization" selection using first setting (setting1)
- Generate a test data creation script for the tables contained in this selection.
- Generate a second script of this model for the "Materials" selection and a test data creation script for the tables it contains using second setting (setting2).

Example:

```
' Generated sql scripts will be created in 'GenDir' directory
' there names is the name of the used selection with extension ".sql" for DDL
scripts
' and extension "_td.sql" for DML scripts (for test data generations).
Option Explicit

Const GenDir = "D:\temp\test\"

Const setting1 = "Tables & Views (with permissions)"
```

```
Const setting2 = "Triggers & Procedures (with permissions)"
Start EvaluateNamedPath("%_EXAMPLES%\project.pdm")

Sub Start(sModelPath)
 on error resume next
 dim pModel : Set pModel = OpenModel(sModelPath)
 If (pModel is Nothing) then
  Output "Unable to open model " & sModelPath
  Exit Sub
 End if

 GenerateDatabaseScripts pModel, "Organization" setting1
 GenerateTestDataScript pModel, "Organization" setting1

 GenerateDatabaseScripts pModel, "Materials" setting2
 GenerateTestDataScript pModel, "Materials" setting2
 pModel.Close
 on error goto 0
End Sub

Sub GenerateDatabaseScripts(pModel, sSelectionName, sSettingName)
 Dim pOpts : Set pOpts = pModel.GetPackageOptions()
 InteractiveMode = im_Batch ' Avoid displaying generate window
' set generation options using model package options
 pOpts.GenerateODBC = False ' Force sql script generation rather than ODBC
 pOpts.GenerationPathName = GenDir
 pOpts.GenerationScriptName = sSelectionName & ".sql"
 ' Launch the Generate Database feature with selected objects
 pModel.UseSelection fct_DatabaseGeneration, sSelectionName
 pModel.UseSetting fct_DatabaseGeneration, sSettingName
 pModel.GenerateDatabase
End Sub

Sub GenerateTestDataScript(pModel, sSelectionName)
 Dim pOpts : Set pOpts = pModel.GetPackageOptions()
 InteractiveMode = im_Batch ' Avoid displaying generate window
 ' set generation options using model package options
 pOpts.TestDataGenerationByODBC = False ' Force sql script generation rather
than ODBC
 pOpts.TestDataGenerationDeleteOldData = False
 pOpts.TestDataGenerationPathName = GenDir
 pOpts.TestDataGenerationScriptName = sSelectionName & "_td.sql"
' Launch the Generate Test Data feature for selected objects
 pModel.UseSelection fct_TestDataGeneration, sSelectionName
 pModel.GenerateTestData
End Sub
```

### Selection and Setting Creation

You can create a persistent selection that can be used in database generation by transforming a selection into a persistent selection..

Example:

```
Option Explicit
 Dim pActiveModel
 Set pActiveModel = ActiveModel

 Dim Selection, PrstSel
 Set Selection = pActiveModel.createselection
 Selection.AddActiveSelectionObjects

 Set PrstSel = Selection.CreatePersistentSelectionManager("test")
```

## Reverse Engineering a Database by Script

You reverse engineer a database using scripts using the ReverseDatabase(ByVal Diagram As BaseObject = Nothing) method.

In the following example, the ODBC database is reversed into a new PDM.

The first lines of the script define the constants used:

- cnxDSN is either the ODBC dsn string or the path to an ODBC file dsn.
- cnxUSR is the ODBC connection user name.
- cnxPWD is the ODBC connection password.

Example

```
option explicit

' To use a user or system datasource, define constant with
"ODBC:<datasourcename>"
' -> Const cnxDSN = "ODBC:ASA 9.0 sample"
' To use a datasource file, define constant with the full path to the DSN file
' -> Const cnxDSN = "\\romeo\public\DATABASES\_filedsn\sybase_asa9_sample.dsn"

' use ODBC datasource
Const cnxDSN = "ODBC:ASA 9.0 sample"
Const cnxUSR = "dba"
Const cnxPWD = "sql"
Const GenDir = "C:\temp\"
Const filename = "D:\temp\phys.pdm"

' Call to main function with the newly created PDM
' This sample use an ASA9 database
Start CreateModel(PdPDM.cls_Model, "|DBMS=Sybase AS Anywhere 9")

Sub Start(pModel)

 If (pModel is Nothing) then
  output "Unable to create a physical model for selected DBMS"
  Exit Sub
 End If

 InteractiveMode = im_Batch

' Reverse database phase
' First connect to the database with connection parameters
 pModel.ConnectToDatabase cnxDSN, cnxUSR, cnxPWD
' Get the reverse option of the model
 Dim pOpt
 Set pOpt = pModel.GetPackageOptions()

' Force ODBC Reverse of all listed objects
 pOpt.ReversedScript = False
 pOpt.ReverseAllTables = true
 pOpt.ReverseAllViews = true
 pOpt.ReverseAllStorage = true
 pOpt.ReverseAllTablespace = true
 pOpt.ReverseAllDomain = true
 pOpt.ReverseAllUser = true
 pOpt.ReverseAllProcedures = true
 pOpt.ReverseAllTriggers = true
 pOpt.ReverseAllSystemTables = true
 pOpt.ReverseAllSynonyms = true
' Go !
```

```
 pModel.ReverseDatabase
 pModel.save(filename)
' close model at the end
 pModel.Close false
End Sub
```

# Manipulating the Repository By Script

PowerDesigner lets you access the Repository feature via scripting using the RepositoryConnection as BaseObject global property.

It allows you to retrieve the current repository connection, which is the object that manages the connection to the repository server and provides access to documents and objects stored under the repository.

The *RepositoryConnection* is equivalent to the root node in the Repository browser.

You can access the repository documents, but you cannot access the repository administration objects, like users, groups, configurations, branches, and list of locks.

In addition, only the last version of a repository document is accessible using scripting.

## Connecting to a Repository Database

Before you connect to the repository database using scripting, definitions of repositories must exist on your workstation, as you cannot define a new repository definition via the scripting feature.

*To retrieve the current repository connection:*

| Use the following | Description |
|---|---|
| RepositoryConnection As BaseObject | Global property which manages the connection to the repository database |

*To connect to a repository database:*

| Use the following | Description |
|---|---|
| Open (ByVal RepDef As String = "", ByVal User As String = "", ByVal Pass As String = "", ByVal DBUser As String = "", ByVal DBPass As String = "") As Boolean | Method on RepositoryConnection that allows you to perform a repository connection |

*To disconnect from the repository:*

| Use the following | Description |
|---|---|
| Close() | Method on RepositoryConnection that allows you to disconnect from the repository database |

You can connect to the Repository database using the following method on RepositoryConnection: Open(ByVal RepDef As String = "", ByVal User As String = "", ByVal Pass As String = "", ByVal DBUser As String = "", ByVal DBPass As String = "") As Boolean.

### Example

```
Dim C
Set C = RepositoryConnection
C.Open
```

You disconnect from the repository database using the following method: Close().

### Example

```
C.Close
```

## Accessing a Repository Document

You can drill down to the repository documents located in the Repository browser using the ChildObjects collection (containing both documents and folders) that also allows you to drill down to documents located in folders of the Repository if any.

*To browse for a document:*

| Use the following | Description |
|---|---|
| ChildObjects As ObjectCol | Collection on StoredObject which manages the access to the repository documents |

*To update a document version:*

| Use the following | Description |
|---|---|
| Refresh() | Method on RepositoryConnection which lets you visualize new documents, update versions of existing documents, or hide deleted ones |

*To find a document:*

| Use the following | Description |
|---|---|
| FindInRepository() As BaseObject | Method on BaseModel that allows you to check if a model has already been consolidated |

The repository documents are the following:

| Repository document | Description |
|---|---|
| RepositoryModel | Contains any type of PowerDesigner model (CDM, PDM, OOM, BPM, XSM, ILM, RQM, ILM, FRM, IAM, LDM, EAM) |
| RepositoryReport | Contains consolidated multi-model reports |
| RepositoryDocument | Contains non-PowerDesigner files (text, Word, or Excel) |
| OtherRepositoryDocument | Contains non-PowerDesigner models defined using the Java Repository interface, which allows you to define your metamodels |

You can access a RepositoryModel document and the sub-objects of a RepositoryModel document using the following collection: ChildObjects As ObjectCol.

### Example

```
' Retrieve the deepest folder under the connection
 Dim CurrentObject, LastFolder
 set LastFolder = Nothing
 for each CurrentObject in C.ChildObjects
 if CurrentObject.IsKindOf(cls_RepositoryFolder) then
   set LastFolder = CurrentObject
  end if
 next
```

The ChildObjects collection is not automatically updated when the Repository is modified during a script execution. To refresh all the collections, you can use the following method: Refresh().

### Example

```
C.Refresh
```

You can test if a model has already been consolidated using the following method: FindInRepository() As BaseObject.

### Example

```
Set repmodel = model.FindInRepository()
If repmodel Is Nothing Then
  ' Model was not consolidated yet...
 model.ConsolidateNew
Else
 ' Model was already consolidated...
 repmodel.Freeze
 model.Consolidate
End If
```

## Extracting a Repository Document

There are two ways to extract a repository document using scripting:

- A generic way that is applicable to any repository document
- A specific way that is only applicable to RepositoryModel and RepositoryReport documents

*To extract any document:*

| Use the following | Description |
|---|---|
| ExtractToFile(ByVal FileName As String, ByVal MergeMode As Long = 2, ByVal OpenMode As Boolean = -1, ByRef Actions As String = NULL, ByRef Conflicts As String = NULL) As BaseObject | Method on RepositoryModel that allows you to extract any kind of document |

*To extract a PowerDesigner document:*

| Use the following | Description |
|---|---|
| UpdateFromRepository(ByVal MergeMode As Integer = 2, ByRef actions As String = NULL, ByRef conflicts As String = NULL) As Boolean | Method on BaseModel that allows you to extract PowerDesigner documents |

### Generic Way

To extract a repository document you must:

- Browse for a repository document using the ChildObjects collection
- Extract the document using the method ExtractToFile (ByVal FileName As String, ByVal MergeMode As Long = 2, ByVal OpenMode As Boolean = -1, ByRef Actions As String = NULL, ByRef Conflicts As String = NULL) As BaseObject

### Example

```
set C = RepositoryConnection
C.Open
Dim D, P
set P = Nothing
for each D in C.ChildObjects
if D.IsKindOf (cls_RepositoryModel) then
D.ExtractToFile ("C:\temp\OO.OOM")
end if
next
```

### Specific Way

To extract a RepositoryModel document or a RepositoryReport document you must:

- Retrieve the document from the local model or multi-model report, (provided it has already been consolidated) using the method UpdateFromRepository (ByVal MergeMode As Integer = 2, ByRef actions As String = NULL, ByRef conflicts As String = NULL) As Boolean

### Example

```
set MyModel = OpenModel ("C:\temp\OO3.OOM")
MyModel.UpdateFromRepository
```

## Consolidating a Repository Document

There are two ways to consolidate a repository document using scripting:

- A generic way that is applicable to any repository document
- A specific way that is only applicable to RepositoryModel and RepositoryReport documents

### To consolidate any document:

| Use the following | Description |
|---|---|
| ConsolidateDocument(ByVal FileName As String, ByVal MergeMode As Long = 2, ByRef Actions As String = NULL, ByRef Conflicts As String = NULL) As BaseObject | Method on RepositoryFolder that allows you to consolidate any kind of document |

### To consolidate a PowerDesigner document:

| Use the following | Description |
|---|---|
| ConsolidateNew(ByVal RepositoryFolder As BaseObject, ByRef actions As String = NULL, ByRef conflicts As String = NULL) As BaseObject | Method on BaseModel that allows you to consolidate PowerDesigner documents |
| Consolidate(ByVal MergeMode As Integer = 2, ByRef actions As String = NULL, ByRef conflicts As String = NULL) As BaseObject | Method on BaseModel that allows you to consolidate additional repository versions of a PowerDesigner document |

### Generic Way

To consolidate any repository document you must:

- Specify a filename when using the following method ConsolidateDocument (ByVal FileName As String, ByVal MergeMode As Long = 2, ByRef Actions As String = NULL, ByRef Conflicts As String = NULL) As BaseObject)

Example:

```
set C = RepositoryConnection
C.open
C.ConsolidateDocument ("c:\temp\test.txt")
```

### Specific Way

To consolidate a RepositoryModel document or a RepositoryReport document you can use one of the following methods:

- ConsolidateNew (ByVal RepositoryFolder As BaseObject, ByRef actions As String = NULL, ByRef conflicts As String = NULL) As BaseObject, to consolidate the first repository version of a document

- Consolidate (ByVal MergeMode As Integer = 2, ByRef actions As String = NULL, ByRef conflicts As String = NULL) As BaseObject, to consolidate additional repository versions of a document

Examples:

```
Set model = CreateModel(PdOOM.cls_Model, "|Diagram=ClassDiagram")
set C = RepositoryConnection
C.Open
model.ConsolidateNew c
```

```
set C = RepositoryConnection
C.Open
model.Consolidate
```

## Understanding the Conflict Resolution Mode

If you update a document that has already been modified since last extraction or consolidation, a conflict can occur.

### Consolidation Conflicts

You can resolve conflicts that arise when consolidating a repository document by specifying a merge mode in the second parameter of the following method: ConsolidateDocument(ByVal FileName As String, ByVal MergeMode As Long = 2, ByRef Actions As String = NULL, ByRef Conflicts As String = NULL) As BaseObject.

This parameter (ByVal MergeMode As Long = 2)can contain the following values:

| Value | Description |
|---|---|
| 1 | Replaces the document in the repository with the local document without preserving any repository changes |
| 2 (default value) | Tries to automatically select the default merge actions by taking into account the modification dates of objects and cancels the consolidation if a conflict has been found (objects modified both locally and in the repository) |
| 3 | Selects the default merge actions but always favors local changes in case of conflict instead of canceling the consolidation |
| 4 | Selects the default merge actions and favors the repository changes in case of conflict |

Merge actions performed during consolidation and conflicts that may have occurred can be retrieved in the strings specified in the third and fourth parameters: ByRef Actions As String = NULL and ByRef Conflicts As String = NULL.

### Extraction Conflicts

You can resolve conflicts that arise when extracting a repository document by specifying a merge mode in the second parameter of the following method: ExtractToFile(ByVal FileName As String, ByVal MergeMode As Long = 2, ByVal OpenMode As Boolean = -1, ByRef Actions As String = NULL, ByRef Conflicts As String = NULL) As BaseObject.

This parameter (ByVal MergeMode As Long = 2)can contain the following values:

| Value | Description |
|---|---|
| 0 | Extracts the document without merge, thus erases the existing local document if any, and sets the extracted file as read-only |
| 1 | Extracts the document without merge, thus erases the existing local document if any |
| 2 (default value) | Tries to automatically select the default merge actions by taking into account the modification dates of objects and cancels the extraction if a conflict has been found (objects modified both locally and the repository) |

| Value | Description |
|---|---|
| 3 | Selects the default merge actions but always favors local changes in case of conflict instead of canceling the extraction |
| 4 | Selects default merge actions and favors the repository changes in case of conflict |

Merge actions performed during extraction and conflicts that may have occurred can be retrieved in the strings specified in the fourth and fifth parameters: ByRef Actions As String = NULL and ByRef Conflicts As String = NULL. The third parameter (ByVal OpenMode As Boolean = -1) allows you to keep open the extracted model.

## Managing Document Versions

You can manage document versions using scripts.

### To freeze and unfreeze a document version:

| Use the following | Description |
|---|---|
| Freeze(ByVal Comment As String = "") As Boolean | Method on RepositoryDocumentBase that allows you to create an archived version of a document |
| Unfreeze() As Boolean | Method on RepositoryDocumentBase that allows you to modify the current version in the repository to reflect changes performed on your local machine |

Example:

```
MyDocument.Freeze "Update required"
```

```
MyDocument.Unfreeze
```

### To lock and unlock a document version:

| Use the following | Description |
|---|---|
| Lock(ByVal Comment As String = "") As Boolean | Method on RepositoryDocumentBase that allows you to prevent other users from updating the consolidated version |
| Unlock() As Boolean | Method on RepositoryDocumentBase that allows other users to update the consolidated version |

Example:

```
MyDocument.Lock "Protection required"
```

```
MyDocument.Unlock
```

### To delete a document version:

| Use the following | Description |
|---|---|
| DeleteVersion() As Boolean | Method on RepositoryDocumentBase that allows you to delete a document version |

Example:

```
MyDocument.Delete
```

## Managing the Repository Browser

The repository browser lets you manipulate folders using scripts.

### To create a folder:

| Use the following | Description |
|---|---|
| CreateFolder(ByVal FolderName As String) As BaseObject | Method on RepositoryFolder that allows you to create a new folder in the repository browser |

Example:

```
RepositoryConnection.CreateFolder("VBTest")
```

### To delete an empty folder:

| Use the following | Description |
|---|---|
| DeleteEmptyFolder() As Boolean | Method on RepositoryFolder that allows you to delete an empty folder in the repository browser |

For more information on documents, see *Accessing a Repository Document* on page 271.

Example:

```
Dim C
Set C = RepositoryConnection
C.Open "MyRepDef"
' Retrieve the deepest folder under the connection
Dim D, P
 set P = Nothing
 for each D in C.ChildObjects
  if D.IsKindOf (cls_RepositoryFolder) then
    D.DeleteEmptyFolder
    c.refresh
  end if
 next
```

## Managing Reports by Script

You can generate HTML and RTF reports using scripting, but you cannot create reports.

### Browsing a Model Report by Script

You can browse a model report using the following collection on BaseModelReport: Reports As ObjectCol.

#### Example

```
set m = ActiveModel
For each Report in m.Reports
Output Report.name
```

### Retrieving a Multimodel Report by Script

You can retrieve a multimodel report using the following function: OpenModel( filename As String, flags As Long =omf_Default) As BaseObject

### Example

```
OpenModel ("c:\temp\mmr1.mmr")
```

## Generating an HTML Report by Script

You can generate a model report or a multimodel report as HTML using the following method on BaseModelReport: GenerateHTML(ByVal FileName As String) As Boolean.

### Example

```
set m = ActiveModel
For each Report in m.Reports
 Filename = Report.name & ".htm"
 Report.GenerateHTML (filename)
Next
```

## Generating an RTF Report by Script

You can generate a model report or a multimodel report as RTF using the following method on BaseModelReport: GenerateRTF(ByVal FileName As String) As Boolean

### Example

```
set m = ActiveModel
For each Report in m.Reports
 Filename = Report.name & ".rtf"
 Report.GenerateRTF (filename)
Next
```

# Accessing Metadata by Script

You can access and manipulate PowerDesigner internal objects using Visual Basic Scripting. The scripting lets you access and modify object properties, collections, and methods using the public names of objects.

The PowerDesigner metamodel provides useful information about objects:

| Information | Description | Example |
|---|---|---|
| Public name | The name and code of the metamodel objects are the public names of PowerDesigner internal objects | AssociationLinkSymbol<br>ClassMapping<br>CubeDimensionAssociation |
| Object collections | You can identify the collections of a class by observing the associations linked to this class in the diagram. The role of each association is the name of the collection | In PdBPM, an association exists between classes MessageFormat and MessageFlow. The public name of this association is Format. The role of this association is Usedby which corresponds to the collection of message flows of class MessageFormat |
| Object attributes | You can view the attributes of a class together with the attributes this class inherits from other classes via generalization links | In PdCommon, in the Common Instantiable Objects diagram, you can view objects BusinessRule, ExtendedDependency and FileObject with their proper attributes, and the abstract classes from which they inherit attributes via generalization links |
| Object operations | Operations in metamodel classes correspond to object methods used in VBS | BaseModel contains operation Compare that can be used in VB scripting |

| Information | Description | Example |
|---|---|---|
| <<notScriptable>> stereotype | Objects that do not support VB scripting have the <<notScriptable>> stereotype | CheckModelInternalMessage<br>FileReportItem |

PowerDesigner lets you access the MetaData via scripting using the MetaModel As BaseObject global property. There is only one instance of the MetaModel and it can be reached from anywhere through the global property Application.MetaModel.

This generic feature allows you to acccess the MetaModel on a generic way and implies a neutral code that you can use for any type of model. For example, you can use it to search for the last object modified in a given model.

Properties and collections are read-only for all MetaData objects.

## Accessing Metadata Objects by Script

You can access the MetaData objects using scripts:

| Use the following | Description |
|---|---|
| MetaModel As BaseObject | Global property. Entry point to access MetaData objects |

## Retrieving the Metamodel Version by Script

You can retrieve the MetaModel version using scripts:

| Use the following | Description |
|---|---|
| Version As String | Property. Allows you to retrieve the MetaModel version |

## Retrieving the Available Types of Metaclass Libraries by Script

You can retrieve the available types of MetaClass libraries using scripts:

| Use the following | Description |
|---|---|
| MetaLibrary | Collection. Allows you to retrieve the available MetaClass of libraries of a given module |

## Accessing the Metaclass of an Object by Script

You can use script to access object metaclasses.

You can access the MetaClass of an object using scripts:

| Use the following | Description |
|---|---|
| MetaClass As BaseObject | Property. Provides access to the Metaclass of each object |

You can access the MetaClass of an object using its public name from the MetaModel using scripts:

| Use the following | Description |
|---|---|
| GetMetaClassByPublicName (ByVal name As String) As BaseObject | Method. Provides access to the MetaClass of an object using its public name |

You can access the MetaAttribute and MetaCollection of a MetaClass using its public name (from the MetaClass):

| Use the following | Description |
|---|---|
| GetMetaMemberByPublicName (ByVal name As String) As BaseObject | Method. Provides access to a MetaAttribute or a MetaCollection using its public name |

### Retrieving the Children of a Metaclass by Script

You can retrieve the children of a MetaClass using scripts:

| Use the following | Description |
|---|---|
| Children As ObjectSet | Collection. Lists the MetaClasses that inherit from the parent MetaClass |

## Managing the Workspace by Script

The *Workspace* object corresponds to the workspace root in the Browser. PowerDesigner lets you access the current workspace using the ActiveWorkspace As BaseObject global property.

### Loading, Saving and Closing a Workspace by Script

The following methods are available to load, save and close a workspace using scripts:

To load a workspace

| Use the following | Description |
|---|---|
| Load (ByVal filename As String = "") As Boolean | Loads the workspace from the given location |

To save a workspace:

| Use the following | Description |
|---|---|
| Save (ByVal filename As String = "") As Boolean | Saves the workspace at the given location |

To close a workspace:

| Use the following | Description |
|---|---|
| Close () | Closes the active workspace |

### Manipulating the Content of a Workspace by Script

You can also manipulate the content of a workspace using the following items:

- The *WorkspaceDocument* that corresponds to the documents you can add to a workspace. It contains the *WorkspaceModel* (models attached to a workspace) and the *WorkspaceFile* (external files attached to the workspace)
- The *WorkspaceFolder* that corresponds to the folders of the workspace. You can create, delete and rename them. You can also add documents to folders.

You can use the AddDocument(ByVal filename As String, ByVal position As Long = -1) As BaseObject method on the WorkspaceFolder to add documents to the workspace.

Example of a workspace manipulation:

```
Option Explicit
' Close existing workspace and save it to Temp
```

```
Dim workspace, curentFolder
Set workspace = ActiveWorkspace
workspace.Load "%_EXAMPLES%\mywsp.sws"
Output "Saving current workspace to ""Example directory :
"+EvaluateNamedPath("%_EXAMPLES%\temp.sws")
workspace.Save "%_EXAMPLES%\Temp.SWS"
workspace.Close
workspace.Name = "VBS WSP"
workspace.FileName = "VBSWSP.SWS"
workspace.Load "%_EXAMPLES%\Temp.SWS"
dim Item, subitem
for each Item in workspace.children
 If item.IsKindOf(PdWsp.cls_WorkspaceFolder) Then
  ShowFolder (item)
  renameFolder item,"FolderToRename", "RenamedFolder"
  deleteFolder item,"FolderToDelete"
  curentFolder = item
 ElsIf item.IsKindOf(PdWsp.cls_WorkspaceModel) Then
 ElsIf item.IsKindOf(PdWsp.cls_WorkspaceFile) Then
 End if
next
 Dim subfolder
'insert folder in root
 Set subfolder = workspace.Children.CreateNew(PdWsp.cls_WorkspaceFolder)
 subfolder.name = "Newfolder(VBS)"
 'insert folder in root at pos 6
 Set subfolder = workspace.Children.CreateNewAt(5, PdWsp.cls_WorkspaceFolder)
 subfolder.name = "Newfolder(VBS)insertedAtPos5"'
 ' add a new folder in this folder
 Set subfolder = subfolder.Children.CreateNew(PdWsp.cls_WorkspaceFolder)
 subfolder.name = "NewSubFolder(VBS)"
 subfolder.AddDocument EvaluateNamedPath("%_EXAMPLES%\pdmrep.rtf")
 subfolder.AddDocument EvaluateNamedPath("%_EXAMPLES%\cdmrep.rtf")
 subfolder.AddDocument EvaluateNamedPath("%_EXAMPLES%\project.pdm")
 subfolder.AddDocument EvaluateNamedPath("%_EXAMPLES%\demo.oom")
 dim lastmodel
 set lastmodel = subfolder.AddDocument (EvaluateNamedPath("%_EXAMPLES%
\Ordinateurs.fem"))
 lastmodel.open
 lastmodel.name = "Computers"
 lastmodel.close
 'detaching model from workspace
 lastmodel.delete
workspace.Save "%_EXAMPLES%\Final.SWS"
```

## Communicating With PowerDesigner Using OLE Automation

OLE Automation (or Visual Basic for Applications) is a way to communicate with PowerDesigner from another application using the COM architecture in the same application or in other applications. You can write a program using any language that support COM, such as Word and Excel macros, VB, C++, or PowerBuilder.

OLE Automation samples for different languages are provided in the OLE Automation directory within your PowerDesigner installation directory.

### Differences Between Scripting and OLE Automation

VBScript programs and OLE Automation programs are very similar. You can easily create VB or VBA programs, if you know how to use VBScript. However, some differences remain. The following example program highlights what differentiates OLE Automation from VBScript.

### VBScript Program

The following VBScript program allows you to count the number of classes defined in an OOM and display that number in PowerDesigner Output window, then create a new OOM and display its name in the same Output window.

To do so, the following steps are necessary:

- Get the current active model using the ActiveModel global function
- Check the existence of an active model and if the active model is an OOM
- Count the number of classes in the active OOM and display a message in the Output window
- Create a new OOM and display its name in the Output window

```
'* Purpose:  This script displays the number of classes defined in an OOM in
the output window.
Option Explicit
' Main function
' Get the current active model
Dim model
Set model = ActiveModel
If model Is Nothing Then
 MsgBox "There is no current model."
ElsIf Not Model.IsKindOf(PdOOM.cls_Model) Then
 MsgBox "The current model is not an OOM model."
Else
 ' Display the number of classes
 Dim nbClass
 nbClass = model.Classes.Count
 Output "The model '" + model.Name + "' contains " + CStr(nbClass) + "
classes."
' Create a new OOM
 Dim model2
 set model2 = CreateModel(PdOOM.cls_Model)
 If Not model2 Is Nothing Then
  ' Copy the author name
  model2.author = model.author
  ' Display a message in the output window
  Output "Successfully created the model '" + model2.Name + "'."
 Else
  MsgBox "Cannot create an OOM."
 End If
End If
```

### OLE Automation Program

To do the same with OLE Automation program, you should modify it as follows:

- Add the definition of the PowerDesigner application
- Call the CreateObject function to create an instance of the PowerDesigner Application object
- Prefix all the global functions (ActiveModel, Output, CreateModel) by the PowerDesigner Application object
- Release the PowerDesigner Application object
- Use specific types for the variables "model" and "model2"

```
'* Purpose:  This script displays the number of classes defined in an OOM in
the output window.
Option Explicit
' Main function
Sub VBTest()
 ' Defined the PowerDesigner Application object
 Dim PD As PdCommon.Application
 ' Get the PowerDesigner Application object
 Set PD = CreateObject("PowerDesigner.Application")
' Get the current active model
 Dim model As PdCommon.BaseModel
 Set model = PD.ActiveModel
```

```
 If model Is Nothing Then
  MsgBox "There is no current model."
 ElsIf Not model.IsKindOf(PdOOM.cls_Model) Then
  MsgBox "The current model is not an OOM model."
 Else
  ' Display the number of classes
  Dim nbClass
  nbClass = Model.Classes.Count
  PD.Output "The model '" + model.Name + "' contains " + CStr(nbClass) + "
classes."
' Create a new OOM
  Dim model2 As PdOOM.Class
  Set model2 = PD.CreateModel(PdOOM.cls_Model)
  If Not model2 Is Nothing Then
   ' Copy the author name
   model2.Author = Model.Author
   ' Display a message in the output window
   PD.Output "Successfully created the model '" + model2.Name + "'."
  Else
   MsgBox "Cannot create an OOM."
  End If
 End If
' Release the PowerDesigner Application object
 Set PD = Nothing
End Sub
```

## Preparing for OLE Automation

To use OLE Automation to communicate with PowerDesigner, you need to:

- Create an instance of the PowerDesigner Application object
- Prefix all global functions with the PowerDesigner Application object
- Release the PowerDesigner Application object before exiting the program
- Specify objects type whenever possible (Dim obj As <ObjectType>)
- Adapt the object class ID syntax to the language when you create object
- Add references to the object type libraries you need to use

### Creating the PowerDesigner Application Object

PowerDesigner setup registers the PowerDesigner Application object by default.

You should check if the returned variable is empty.

When you create the PowerDesigner Application object, the current instance of PowerDesigner will be used, otherwise PowerDesigner will be launched.

If PowerDesigner is launched when you create the PowerDesigner Application object, it will be closed when you release the PowerDesigner Application object.

You create the PowerDesigner application object, using the following method in Visual Basic: CreateObject(ByVal Kind As Long, ByVal ParentCol As String = "", ByVal Pos As Long = -1, ByVal Init As Boolean = -1) As BaseObject

#### Example

```
' Defined the PowerDesigner Application object
 Dim PD As PdCommon.Application
 ' Get the PowerDesigner Application object
 Set PD = CreateObject("PowerDesigner.Application")
```

### PowerDesigner Version Number

If you want to make sure that the application works with a selected version of PowerDesigner, you should type the version number in the PowerDesigner application object creation orders:

```
' Defined the PowerDesigner Application object
 Dim PD As PdCommon.Application
  ' Get the PowerDesigner Application object
 Set PD = CreateObject("PowerDesigner.Application.x")
'x represents the version number
```

If you do not use a particular feature of PowerDesigner, your application can work with any version of PowerDesigner and you do not need to specify a version number. In this case, the last version installed is used.

**Note:** You must release the PowerDesigner Application object before you exit the application in which you use it. To do so, you use the following syntax: Set Pd = Nothing.

### Specifying the Object Type

When you create VB or VBA programs, it is strongly recommended to specify the object type.

For example, you should use:

```
Dim cls As PdOOM.Class
```

Instead of:

```
Dim cls
```

If you do not specify object type, you may encounter problems when you execute your program and debugging can be really difficult.

### Shortcuts

If the model contains shortcuts, we recommend to use the following syntax: Dim obj as PdCommon.IdentifiedObject.

If the target model is closed, you will get a runtime error.

### Adapting the Object Class ID Syntax to the Language

When you create an object using VBScript, you indicate the class ID of the object to create in the following way:

```
Dim cls
Set cls = model.CreateObject(PdOOM.cls_Class)
```

This syntax works properly for VBScript, VBA and VB, but it does not work for other languages, as class Ids constants are defined as an enumeration. Only languages that support enumeration defined outside a class can support this syntax.

For C# and VB.NET, you can use the following syntax:

```
Dim cls As PdOOM.Class
Set cls = model.CreateObject(PdOOM.PdOOM_Classes.cls_Class)
'Where PdOOM_Classes is the name of the enumeration.
```

For other languages such as JavaScript or PowerBuilder, you have to define constants that represent the objects you want to create.

For a complete list of class ID constants, see file VBScriptConstants.vbs in the PowerDesigner OLE Automation directory.

### Adding References to Object Type Libraries

You must add references to the PowerDesigner type libraries you want to use, for example Sybase PdCommon, Sybase PdOOM, Sybase PdPDM, etc. for programs like VB, VBA, VB .NET and C#.

### To Add References to Object Type Libraries in a VBA Editor:

Select **Tools** > **References** .

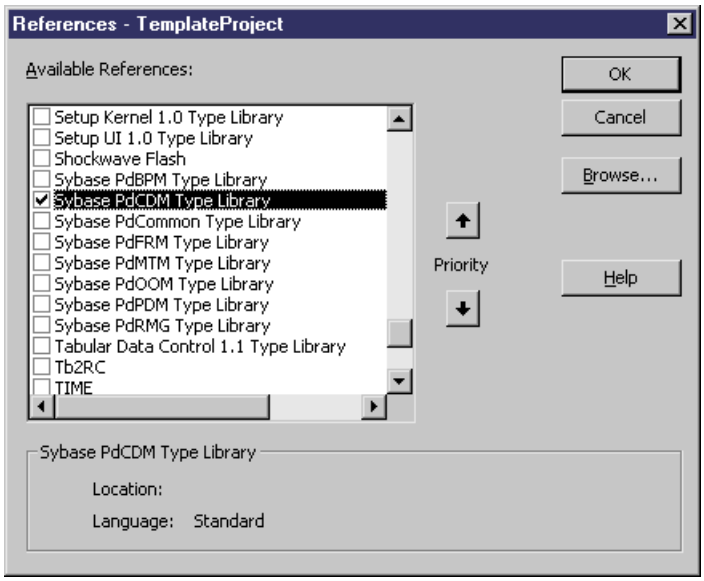
### To Add References to Object Type Libraries in a Visual Basic Editor:

Select **Project** > **References** .

### To Add References to Object Type Libraries in a C# and VB.NET Editor:

Right-click the project in the project explorer, and select Add References from the contextual menu.

Example of a References Window for a VBA Program in Word:




## Customizing PowerDesigner Menus Using Add-Ins

An add-in is a module that adds a specific feature or service to PowerDesigner standard behavior. PowerDesigner add-ins allow you to customize PowerDesigner menus by adding your own menu items. You can customize the following menus:

- All contextual menus of objects that are accessible from the Browser or from a symbol in the diagram
- Main menus of each module from each diagram type (i.e. Import, Export, Reverse, Tools, Help)

You can add the following menu items:

- Commands that call a method script defined using scripting
- Submenus that are cascading menus that appear under a menu item
- Separators that are lines used to organize commands in menus

You can use the following types of add-ins to create menu items in PowerDesigner:

- Customized commands - to call executable programs or VB scripts using the Customize Commands dialog box from the Tools application menu. Commands you define can appear as submenus only in the Execute Commands menu items and in the Import and Export menu items of the File application menu, but not in objects contextual menu. You can hide their display in the menu while keeping their definition. For more information, see *Creating customized commands in the Tools menu* on page 285.
- Resource files – for defining commands for a specific target. Methods and menus are created in the resource file in the Profile category under the corresponding metaclass. You can filter methods and menus using a stereotype or a criterion. However, the resource file must always be attached to the model in order for the commands to be displayed. For more information, see *Menus (Profile)* on page 161.
- ActiveX – for when you require more complex interactions with PowerDesigner, such as enabling and disabling menu items based on object selection, interaction with the windows display environment or for plug-ins written in other languages, such as Visual Basic.NET or C++. For more information, see *Creating an ActiveX add-in* on page 290.

- XML file – for when you want to define several commands that will always be available independently from the target you selected. This XML file contains a simple declarative program with a language linked to an .EXE file or a VB script. Commands linked to the same applications (for example, ASE, IQ etc.) should be gathered into the same XML file. For more information, see *Creating an XML file add-in* on page 291.

**Note:** The XML syntax of a menu defined in the Menu page of the resource editor is the same for XML file and ActiveX add-ins. You can use the interface of the resource editor to visualize in the XML page the syntax of a menu you created in the Menu page to help you construct the same XML syntax in your ActiveX or XML file. For more information on XML files, *Creating An XML File Add-In* on page 291.

## Creating Customized Commands in the Tools Menu

You can create your own menu items in the PowerDesigner Tools menu to access PowerDesigner objects using your own scripts.

From the Tools application menu, you can add your own submenu entries that will allow you to execute the following commands:

- Executable programs
- VB scripts

You can also gather commands into submenus, modify existing commands, and apply to them keyboard shortcut.

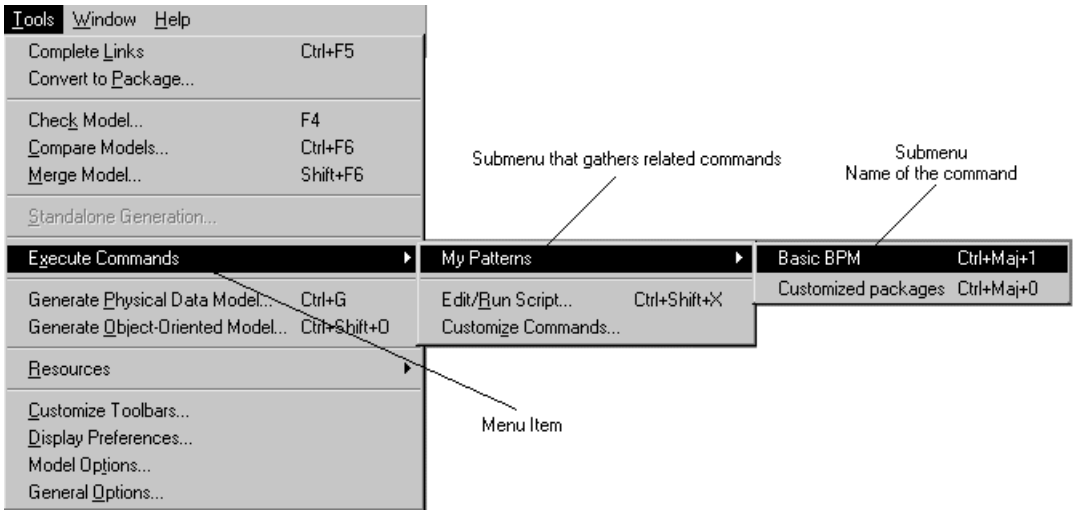
### Defining a Customized Command

You can define commands in the Customize Commands dialog box. The number of commands you can define is limited to 256.

When you define a command, the name you typed for the command is displayed as a submenu entry of the Execute Commands menu item. Command names appear alphabetically sorted.

You can define a context for that command, so it becomes diagram dependent and displays as a submenu only when it is relevant.

The following picture illustrates the result of commands definition performed in the Customize Commands dialog box.




To define a command, you have to specify the following in the Customize commands dialog box:

| Command definition | Description |
|---|---|
| Name | Name of the command that is displayed as a submenu in the Execute Commands menu item. Names are unique and can contain a pick letter (&Generate Java will appear as Generate Java) |

| Command definition | Description |
|---|---|
| Submenu | Name of the submenu that groups commands. It is displayed in the Execute Commands menu item. You can select a default submenu from the list (<None>, Check Model, Export, Generation, Import, Reverse) or create your own submenu that will be added to the listbox. If you select <None> or leave the box empty, the command you defined will be directly added in the submenu of the Execute Commands menu item |
| Context | Optional information that allows the display of the command according to the opened diagram. If you do not define a context for the command, it will appear in the Execute Commands menu item whatever the opened diagram, and even when no diagram is active |
| Type | Type of the command that you select from the list. It can be an executable or a VB script |
| Command Line | Path of the command file. The Ellipsis button allows you to browse for a file or any argument. If the command file is a VB script, you can click the button in the toolbar to directly open the scripting editor and preview or edit the script |
| Comment | Descriptive label for the command. It is displayed in the status bar when you select a command name in the Execute Commands menu item |
| Show in Menu | Indicates whether the command name should be displayed in the Execute Commands menu item or not. It allows you to disable a command in the menu without deleting the command definition |
| Keyboard shortcut | Allows you to apply a keyboard shortcut to the command. You can select one from the list. The use of a keyboard shortcut must be unique |

### Context Option

The Context option allows you to define a diagram dependent command that will appear only when the parameters you declared in its definition match the current diagram.

When no matches are found, the command is unavailable.

When you click the Ellipsis button in the Context column of the Customize Commands dialog box, you open the Context Definition dialog box in which you are asked to select the following optional parameters:

| Parameter | Description |
|---|---|
| Model | Allows you to select a model type from the Model list |
| Diagram | Allows you to select a diagram type for the selected model from the Diagram list |
| Target resource | Allows you to select or type a XEM file from the Target Resource list, which contains all the XEM files defined for the selected model type. The path button allows you to browse for another particular target resource (XOL, XPL, XSL or XDB) in another folder |

Here are some examples of context definitions as they display in the Context column of the Customize Commands dialog box:

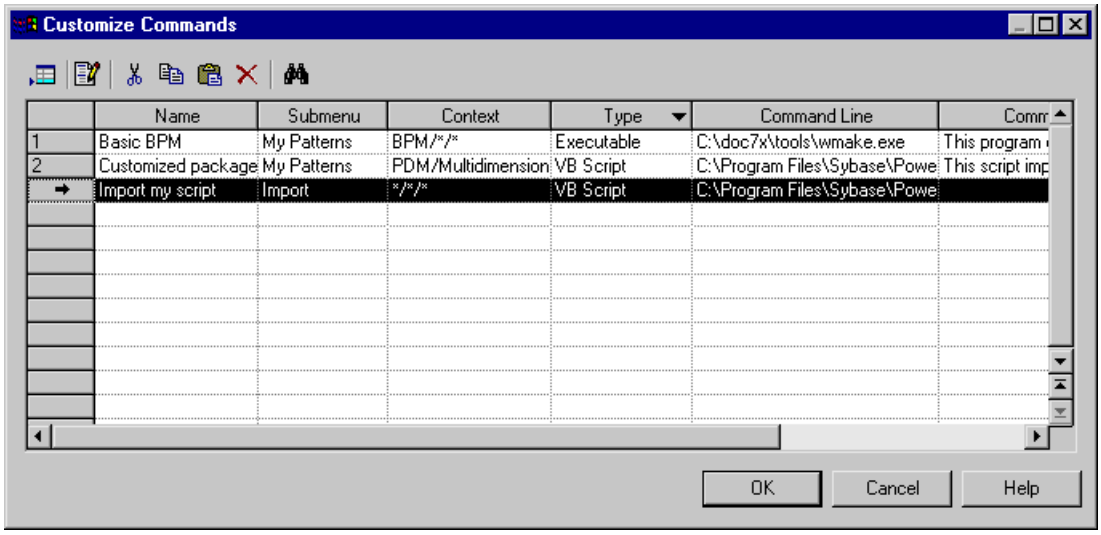
| Context definition | Description |
|---|---|
| */*/* | Default value. The command is displayed in the Execute Commands menu item whatever the opened diagram, and even when no diagram is active |
| OOM/*/* | The command is displayed in the Execute Commands menu item whenever an OOM is opened, whatever the opened diagram and the selected target resource |

| Context definition | Description |
|---|---|
| OOM/Class diagram/* | The command is displayed in the Execute Commands menu item whenever an OOM is opened with a class diagram, whatever the selected target resource |
| OOM/Class diagram/Java | The command is displayed in the Execute Commands menu item whenever an OOM is opened with a class diagram which target resource is Java |

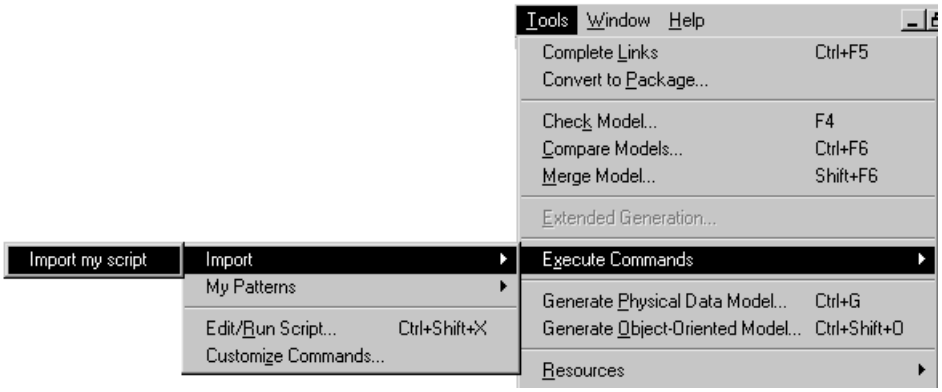
## Import/Export Submenus

When you select Import or Export in the Submenu list of the Customize Commands dialog box, the command you defined is displayed not only as a submenu entry of the Execute Commands menu item of the Tools menu but also as a submenu entry of the Import or Export menu items of the File menu.

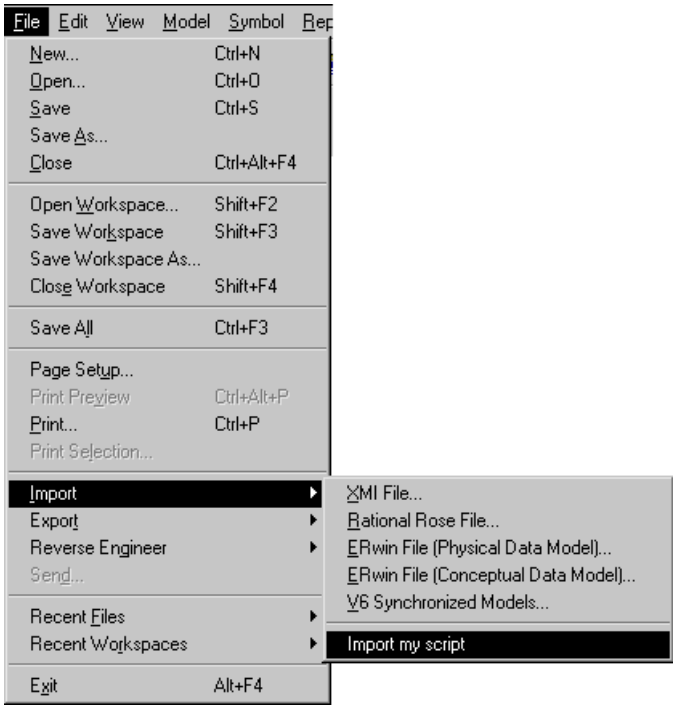
For example you defined the following command in the Customize Commands dialog box:




The command is displayed as follows in the Tools menu:




The command is displayed as follows in the File menu:



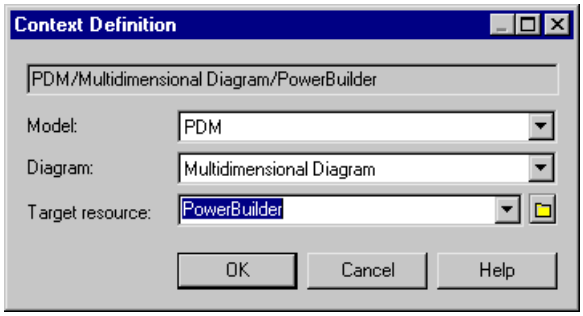

## To Define a Customized Command:

You can define your own customized commands.

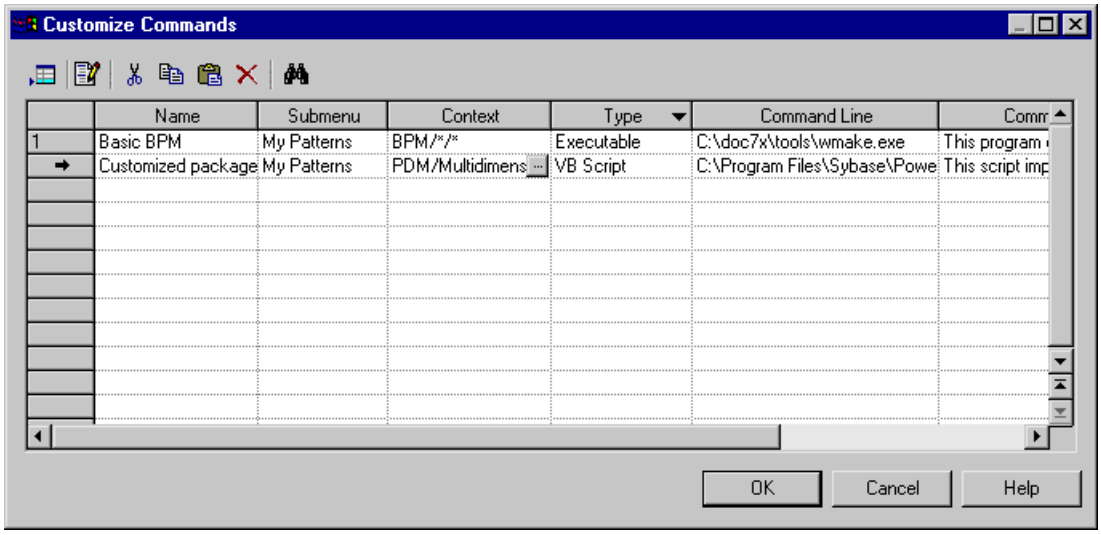
1. Select **Tools** > **Execute Commands** > **Customize Commands** to display the Customize Commands dialog box.
2. Click a blank line in the list.

   *or*

   Click the Add a row tool.
3. Type a name for the command in the Name column.
4. (Optional) Select a submenu from the list in the Submenu column.
5. (Optional) Define a context by clicking the Ellipsis button in the Context column.




6. Select a type from the list in the Type column.
7. Browse to the directory that contains the command file or argument in the Command Line column.
8. (Optional) Type a comment in the Comment column.
9. Select the Show in Menu check box to display the command name in the menu.
10. (Optional) Select a keyboard shortcut from the list in the Shortcut key column.
11. Click OK.




You can vizualise or modify the command you have just defined by selecting **Tools** > **Execute Commands** .

## Managing Customized Commands

Understanding how customized commands are stored in PowerDesigner will allow you to easily plug your programs in PowerDesigner while installing them.

### Storage

Customized Commands are saved in the Registry. You can define values for customized commands in the CURRENT USER Registry or in the LOCAL MACHINE Registry.

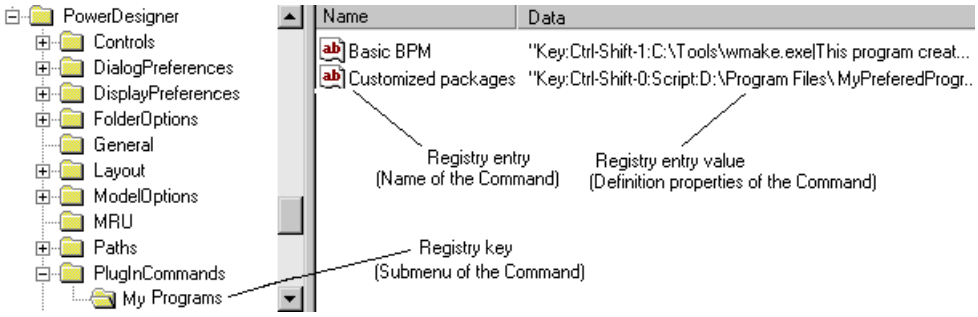
If you define values in the LOCAL MACHINE Registry, customized commands are available for any user of the machine. Thus, when you remove a customized command defined in that Registry from the Customize Commands dialog box, you only remove the line from the list but not the corresponding Registry entry. When you do so, the default value (the one defined in the LOCAL MACHINE Registry) is restored when you open the dialog box again.

The location of customized commands definition can be:

- HKEY_CURRENT_USER\Software\Sybase\PowerDesigner <version>\PlugInCommands
- HKEY_LOCAL_MACHINE\Software\Sybase\PowerDesigner <version>\PlugInCommands

Each customized command is stored in a single Registry string value:

- The name of the customized command is a Registry entry, which has the same name as the command
- The submenu of the customized command is a Registry key, which has the same name as the submenu
- Other command properties are stored in the Value Data field of the Registry entry (Registry entry value)




### Definition Format

The syntax of the Registry entry is the following:

*[Hide:][Key:<key specification>:][Script:]<command>[ |comment]*

Note that none of the above quoted prefix is localized.

| Syntax Keyword | Description |
|---|---|
| Hide: | Defines the command as hidden |
| Key:<key specification>: | Allows the association of a keyboard shortcut to the command. This is an optional field. The <key specification> element can include the following optional prefixes in this order:<br>• ctrl- for CONTROL flag<br>• shift- for SHIFT flag<br>Immediately followed by a single character, included between "0-9" (for example:Ctrl-Shift-0) |
| Script: | Defines the command to be interpreted as an internal script |
| <Command> | Defines the filename with optional arguments for the command. The command is mandatory and is terminated by a '\|' character. If you want to insert a '\|' character within a command, you must double it |
| Comment | Describes the command. This is an optional field |

Note: The Customize Commands dialog box only supports "Ctrl-Shift-0" to "Ctrl-Shift-9" keyboard shortcuts. If you define a keyboard shortcut outside that range, conflicts with some other built-in keyboard shortcuts may occur and lead to unpredictable results. The reuse of the same keyboard shortcut for two distinct commands may also lead to unpredictable results.

## Creating an ActiveX Add-in

You can create your own menu items in PowerDesigner menus by creating an ActiveX add-in. To use your add-in, save it to the Add-ins directory beneath your PowerDesigner installation directory and enable it through the PowerDesigner General Options window (see "Managing add-ins" in the Models chapter of the *Core Features Guide*).

The ActiveX must implement a specific interface called IPDAddIn to become a PowerDesigner add-in.

This interface defines the following methods:

- HRESULT Initialize([in] IDispatch * pApplication)
- HRESULT Uninitialize()
- BSTR ProvideMenuItems([in] BSTR sMenu, [in] IDispatch *pObj)
- BOOL IsCommandSupported([in] BSTR sMenu, [in] IDispatch * pObject, [in] BSTR sCommandName)
- HRESULT DoCommand(in BSTR sMenu, in IDispatch *pObj, in BSTR sCommandName)

Those methods are invoked by PowerDesigner in order to dialog with menus and execute the commands defined by the ActiveX.

### Initialize / Uninitialize Method

The Initialize method initializes the communication between PowerDesigner and the ActiveX. PowerDesigner starts the communication by providing the ActiveX with a pointer to its application object. The application object allows you to handle the PowerDesigner environment (output window, active model etc.) and must be saved for later reference. The application object type is defined into the PdCommon type library.

The Uninitialize method is used to clean references to PowerDesigner objects. It is called when PowerDesigner is closed and must be used to release all global variables.

### ProvideMenuItems Method

The ProvideMenuItems method returns an XML text that describes the menu items to add into PowerDesigner menus. The method is invoked each time PowerDesigner needs to display a menu.

When you right-click a symbol in a diagram, this method is called twice: once for the object and once for the symbol. Thus, you can create a method that is only called on graphical contextual menus.

The ProvideMenuItems is called once at the initialization of PowerDesigner to fill the Import and Reverse menus. No object is put in parameter in the method at this moment.

The XML text that describes a menu can use the following elements (DTD):

```
<!ELEMENT Menu (Command | Separator | Popup)*>
<!ELEMENT Command>
<!ATTLIST Command
    Name    CDATA    #REQUIRED
    Caption    CDATA    #REQUIRED
>
<!ELEMENT Separator>
<!ELEMENT PopUp (Command | Separator | Popup)*>
<!ATTLIST PopUp
    Caption    CDATA    #REQUIRED
>
```

Example:

```
ProvideMenuItems ("Object", pModel)
```

The following text results:

```
<MENU>
<POPUP Caption="&Perforce">
    <COMMAND Name="CheckIn" Caption="Check &In"/>
    <SEPARATOR/>
    <COMMAND Name="CheckOut" Caption="Check &Out"/>
</POPUP>
</MENU>
```

Note: This syntax is the same used in the creation of a menu using a resource file.

**Note:** You can use the interface of the resource editor to visualize in the XML page the syntax of a menu you created in the Menu page that will help you construct the same XML syntax.

For more information on how to customize menus using a resource file, see *Adding Commands and Other Items to Your Menu* on page 163.

### IsCommandSupported Method

The IsCommandSupported method allows you to dynamically disable commands defined in a menu. The method must return true to enable a command and false to disable it.

### DoCommand Method

The DoCommand method implements the execution of a command designated by its name.

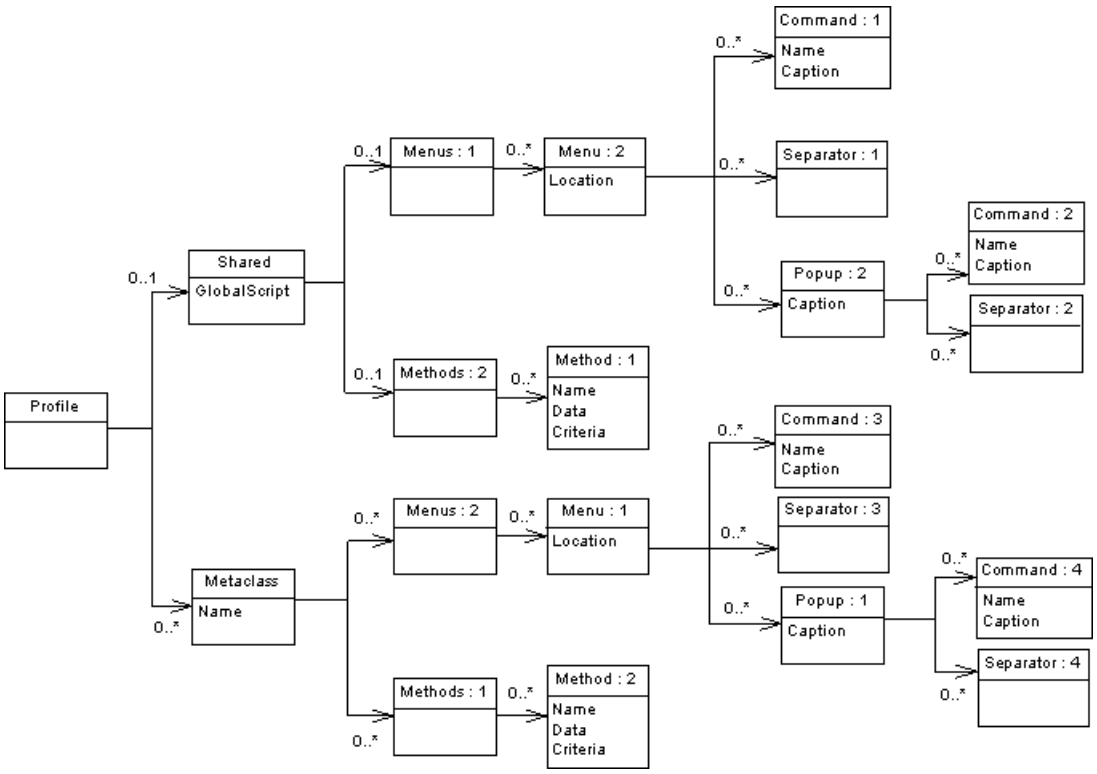
Example:

```
DoCommand ("Object", pModel, "CheckIn")
```

## Creating an XML File Add-in

You can create your own menu items in PowerDesigner menus by using an XML file. To use your add-in, save it to the Add-ins directory beneath your PowerDesigner installation directory and enable it through the PowerDesigner General Options window (see "Managing add-ins" in the Models chapter of the *Core Features Guide*).

The following illustration helps you understand the XML file structure:




The Profile is the root element of the XML file add-in descriptor. It contains the following parts:

- Shared for which menus and commands are defined
- Metaclass which defines commands and menus for a specific metaclass

<!ELEMENT Profile ((Shared)?, (Metaclass)*)>.

## Shared

The Shared element defines the menus that are always available and their associated methods (Menus, and Methods elements) and the shared methods (GlobalScript attribute).

The GlobalScript attribute is used to specify an optional global script (VBS) that can contain shared functions.

The Menus element contains menus that are always available for the application. A Location can be specified to define the menu location. It can take the following values:

- FileImport
- File reverse
- Tools
- Help

You can only define one menu per location.

The Methods defines the methods used in the menus described in the Menus element and that are available for the application.

## Metaclass

The Metaclass element is used to specify menus that are available for a specific PowerDesigner metaclass. A metaclass is identified by a name. You must use the public name.

The Menus element contains menus available for a metaclass.

The Menu element describes a menu available for a metaclass. It contains a series of commands, separators or popups. A location can be specified to define the menu location. It can take the following values:

- FileExport
- Tools
- Help
- Object

Object is the default value for the Location attribute.

The Methods element contains a series of method available for a metaclass.

The Method element defines a method. A method is identified by a name and a VB script.

The Command element defines a command menu item. Its name must be equal to the name of a Method in order to be implemented.

The Popup element defines a sub-menu item that may contain commands, separators or popups.

The Caption is the displayed value in the menu.

A separator indicates that you want to insert a line in the menu.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Profile>
    <Metaclass Name="PdOOM.Model">
    <Menus>
    <Menu Location="Tools">
        <Popup Caption="Perforce">
            <Command Name="CheckIn" Caption="Check In"/>
            <Separator/>
            <Command Name="CheckOut" Caption="Check Out"/>
        </Popup>
    </Menu>
    </Menus>
    <Methods>
        <Method Name="CheckIn">
Sub %Method%(obj)
execute_command( p4, submit %Filename%, cmd_PipeOutput)
End Sub
        </Method>
        <Method Name="CheckOut">
Sub %Method%(obj)
execute_command( p4, edit %Filename%, cmd_PipeOutput)
End Sub
        </Method>
    </Methods>
</Metaclass>
</Profile>
```

A method defined under a metaclass is supposed to have the current object as parameter; its name is calculated from the attribute name of the method tag.

Example:

```
<Method Name="ToInt" >
Sub %Method%(obj)
 Print obj
 ExecuteCommand(&quot;%MORPHEUS%\ToInt.vbs&quot;, &quot;&quot;,
cmd_InternalScript)
End Sub
```

Each metaclass name must be prefixed by its Type Library public name like PdOOM.Class.

Inheritance is taken into account: a menu defined on the metaclass PdCommon.NamedObject will be available for a PdOOM.Class.

You can only define one menu for a given location. If you define several locations only the last one will be preserved.

Menus defined in the Shared section can refer to "FileImport" "Reverse" and "Help" locations.

These menus can only refer to method defined under Shared and no object is put in parameter in the methods defined under Shared.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Profile>
    <Shared>
    <GlobalScript>
Option Explicit
Function Print (obj)
Output obj.classname &amp; &quot; &quot; &amp; obj.name
End Function
    /GlobalScript>
    </Shared>
    <Metaclass Name="PdOOM.Class">
    <Menus>
    <Menu>
        <Popup Caption="Transformation">
            <Command Name="ToInt" Caption="Convert to interface"/>
            <Separator/>
        </Popup>
    </Menu>
    </Menus>
    <Methods>
        <Method Name="ToInt" >
Sub %Method%(obj)
 Print obj
 ExecuteCommand(&quot;%MORPHEUS%\ToInt.vbs&quot;, &quot;&quot;,
cmd_InternalScript)
End Sub
        </Method>
    </Methods>
    </Metaclass>
</Profile>
```

You can find the DTD in the Add-ins folder of the PowerDesigner directory.

Note: You can retrieve in this example the same syntax used in the creation of a menu using a resource file.

**Note:** You can use the interface of the resource editor to visualize in the XML page the syntax of a menu you created in the Menu page that will help you construct the same XML syntax.

# Index

## D



## E

## F



## G

## H



## I



## J



## K



## L



## M

## N



## O

## P



## Q



## R

## S

## T



## U

## V



## W



## X