SYBASE®

An SAP® Company

Customizing and Extending PowerDesigner

PowerDesigner® 16.0

Windows

Contents

CHAPTER 1 Working with PowerDesigner Resource Files

The PowerDesigner® modeling environment is powered by XML-format resource files, which define the objects available in each model along with the methods for generating and reverse-engineering them. You can view, copy, and edit the provided resource files and create your own in order to customize and extend the behavior of the environment.

The following types of resource files are provided:

- *Definition file*: define the standard objects available in a model:
  - *DBMS definition files* (.xdb) - define a specific DBMS in the PDM (see *Chapter 4, DBMS Definition Files* on page 127).
  - *Process language definition files* (.xpl) – define a specific business process language in the BPM (see *Chapter 3, Object, Process, and XML Language Definition Files* on page 117).
  - *Object language definition files* (.xol) - define a specific object-oriented language in the OOM (see *Chapter 3, Object, Process, and XML Language Definition Files* on page 117).
  - *XML language definition files* (.xsl) - define a specific XML language in the XSM (see *Chapter 3, Object, Process, and XML Language Definition Files* on page 117).
- *Extension files* (.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 *Chapter 2, Extension Files* on page 21).
- *Report templates* (.rtp) - specify the structure of a report. Editable within the Report Template Editor (see *Core Features Guide > The PowerDesigner Interface > Reports*).
- *Report language files* (.xrl) – translate the headings and other standard text in a report (see *Chapter 6, Translating Reports with Report Language Resource Files* on page 305).
- *Impact and lineage analysis rule sets* (.rul) - specify the rules defined for generating impact and lineage analyses (see *Core Features Guide > Linking and Synchronizing Models > Impact and Lineage Analysis*).
- *Object permission profiles* (.ppf) - customize the PowerDesigner interface to hide models, objects, and properties (see *Core Features Guide > Administering PowerDesigner > Customizing the PowerDesigner Interface*).
- *User profiles* (.upf) - store preferences for model options, general options, display preferences, etc (see *Core Features Guide > The PowerDesigner Interface > Customizing Your Modeling Environment > User Profiles*).
- *Model category sets* (.mcc) - customize the New Model dialog to guide model creation (see *Core Features Guide > Administering PowerDesigner > Customizing the PowerDesigner Interface > Customizing the New Model Dialog*).

- *Conversion tables* (.csv) - define conversions between the name and code of an object (see *Core Features Guide > The PowerDesigner Interface > Objects > Object Properties > Naming Conventions*).

These resource files are based upon the PowerDesigner public metamodel (see *The PowerDesigner Public Metamodel* on page 8).

You can review all the available resource files from the lists of resource files, available 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 *Creating and Copying Resource Files* on page 6).
	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 6).
	Merge - Selects two resource files for merging (see *Merging Resource Files* on page 7).
	Check In - [if the repository is installed] Checks the selected resource file into the repository. For information about storing your resource files in the repository, see *Core Features Guide > Administering PowerDesigner > Deploying Shared Resources*.
	Update from Repository - [if the repository is installed] Checks out a version of the selected file from the repository to your local machine.
	Compare with Repository - [if the repository is installed] Compares the selected file with a resource file stored in the repository.

Opening Resource Files in the Editor

When working with a BPM, PDM, OOM, or XSM, you can open the definition file that controls the objects available in your model in the Resource Editor for viewing and editing. You can also open and edit any extension files currently attached to or embedded in your model or access the appropriate list of resource files and open any PowerDesigner resource file.

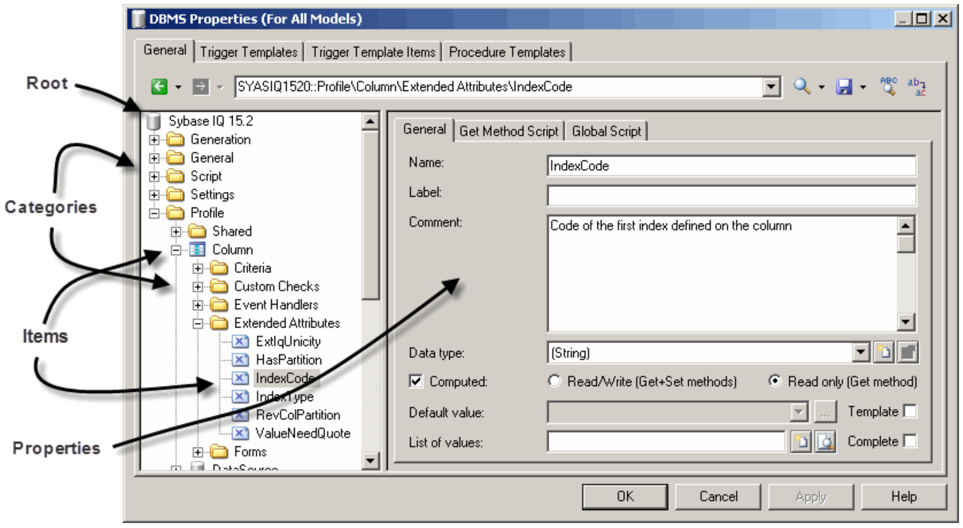
To open the definition file currently used by your model:

- In a PDM, select **Database** > **Edit Current DBMS**.
- In a BPM, select **Language** > **Edit Current Process Language**.
- In an OOM, select **Language** > **Edit Current Object Language**.
- In an XSM, select **Language** > **Edit Current Language**.

To open any extension file currently attached to your model, double-click its entry inside the **Extensions** category in the Browser.

To open any other resource file, select **Tools** > **Resources** > ***Type*** to open the relevant resource file list, select a file in the list, and then click the **Properties** tool.

In each case, the file opens in the Resource Editor, in which you can review and edit the structure of the resource. 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:



SYASIQ1520::Profile\Column\Extended Attributes\IndexCode

Name: IndexCode

Label:

Comment: Code of the first index defined on the column

Data type: (String)

[✓] Computed: Read/Write (Get+Set methods) [✓ Read only (Get method)]

Default value:

Template ☐

List of values:

Complete ☐

Note: You should never modify the resource files shipped with PowerDesigner. If you want to modify a file, create a copy using the **New** tool (see *Creating and Copying Resource Files* on page 6).

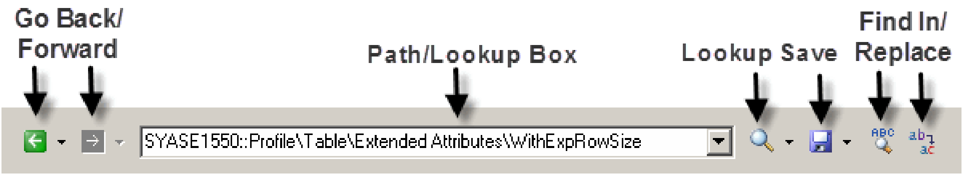
Each entry is a part of the definition of a resource file, and entries are organized into logical categories. For example, the Script category in a DBMS language file collects together all the entries relating to database generation and reverse engineering.

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: 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.

Navigating and Searching in Resource Files

The tools at the top of the Resource Editor help you to navigate through and search in the resource file.



Tool	Description
	Back (**Alt+Left**) - Go to the previous visited entry or category. Click the down arrow to directly select from your history.
	Forward (**Alt+Right**) - Go to the next visited entry or category. Click the down arrow to directly select from your history.
	Lookup (**Enter**) - Go to the item named in the text box to the left of the tool. If more than one item is found, they are listed in a results dialog and you should double-click on the desired item or select it and click **OK** to go to it. Click the down arrow to set lookup options: • [extension type] - select the type of extension to search, for example you can search only stereotypes • Allow wildcard - Enables the use of the characters `*` to match any string and `?` to match any single character. For example, type `is*` to retrieve all extensions called `is...`. • Match case - Search with case sensitivity.
	Save (**Ctrl+Shift+S**) – Save the current resource file. Click the down arrow to save the current resource file under a new name.
	Find In Items (**Ctrl+Shift+F**) - Search for text in entries.
	Replace In Items (**Ctrl+Shift+H**) - Search for and replace text in entries.

Note: To jump to the definition of a template from a reference in a field of other template, place your cursor between the percent signs and press **F12** (see *Templates and Generated Files (Profile)* on page 104).

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.

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.

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.

Sharing and Embedding Resource Files

Resource files can be shared and referenced by multiple models or copied to and embedded in a single model. Any modifications that you make to a shared resource are available to all models using the resource, while modifications to an embedded resource are available only to the model in which it is embedded. Embedded resource files are saved as part of their model and not as a separate file.

Note: You should never modify the original extensions shipped with PowerDesigner. To create a copy of the file to modify, open the List of Extensions, click the **New** tool, specify a name for the new file, and then select the .xem that you want to modify in the **Copy from** field.

The **File Name** field displays the location of the resource file you are modifying is defined. This field is empty if the resource file is embedded.

Creating and Copying Resource Files

You can create a new resource file in the appropriate resource file list. To create a copy of an existing resource file, select it in the **Copy from** field of the **New...** dialog.

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

1. Select **Tools > Resources >** ***Type*** to open the appropriate resource file list.
2. Click the **New** tool, enter a name for the new file and select an existing file to copy. Select the `<Default template>` item to create a minimally completed resource file.
3. Click **OK** to create the new resource file, and then specify a filename and click **Save** to open it in the Resource Editor.

   Note: You can create an extension file directly in your model from the List of Extensions. For more information, see *Creating an Extension File* on page 23.

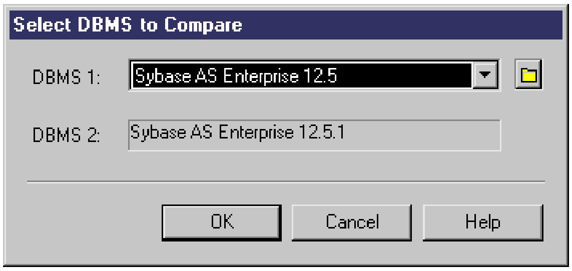
Comparing Resource Files

You can select two resource files and compare them to highlight the differences between them.

1. Select **Tools > Resources >** ***Type*** to open the appropriate resource file list.
2. Select the first resource file you want to compare in the list, and then click the **Compare** tool to open a selection dialog.

   The selected file is displayed in the second comparison field.
3. Select the other resource file to compare in the first comparison field.

   If the resource file you want to compare is not in the list, click the **Select Path** tool and browse to its directory.



DBMS 1: Sybase AS Enterprise 12.5

DBMS 2: Sybase AS Enterprise 12.5.1

4. Click **OK** to open the **Compare...** dialog, which allows you to review all the differences between the files.

   For detailed information about this window, see *Core Features Guide > The PowerDesigner Interface > Comparing and Merging Models*.
5. Review the differences and then click **Close** to close the comparison window and return to the list.

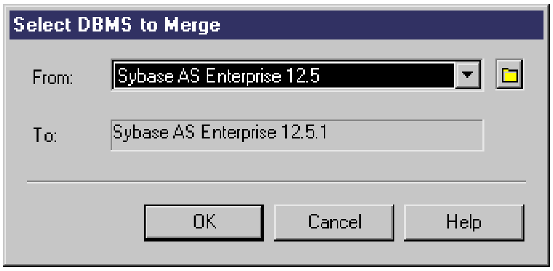
Merging Resource Files

You can select two resource files of the same kind and merge them. 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 the right hand resource file.

1. Select **Tools** > **Resources** > *Type* to open the appropriate resource file list.
2. Select the resource file in which you want to make merge changes in the list, and then click the **Merge** tool to open a selection dialog.

   The selected file is displayed in the **To** field.
3. Select the resource file from which you want to merge in the **From** field.

   If the resource file you want to merge is not in the list, click the **Select Path** tool and browse to its directory.



From: Sybase AS Enterprise 12.5

To: Sybase AS Enterprise 12.5.1

4. Click **OK** to open the **Merge...** dialog, which allows you to review all the merge actions before you complete them.

   For detailed information about this window, see *Core Features Guide > The PowerDesigner Interface > Comparing and Merging Models*.

5. Select or reject the proposed merge actions as necessary, and then click **OK** to perform the merge.

The PowerDesigner Public Metamodel

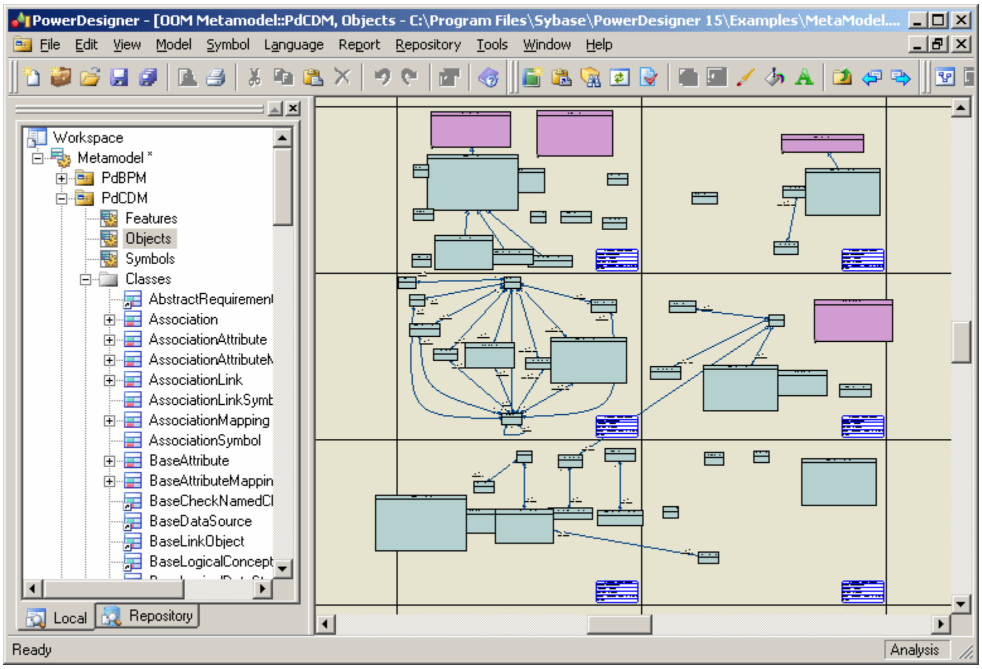
A metamodel describes the elements of a model, and the syntax and semantics of their manipulation. Where a model is an abstraction of data, and can be described using metadata, the metamodel is an abstraction of that metadata.

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 *Model Files and the PowerDesigner Metamodel* on page 15)

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.
- PdEAM - Enterprise Architecture Model
- PdFRM - Free Model
- PdILM - Data Movement Model (the DMM was previously named Information Liquidity Model or ILM, and the PdILM library name has been retained for backwards compatibility)
- PdLDM - Logical Data Model
- PdMTM - Merise Model (available in French only)
- PdOOM - Object Oriented Model
- PdPDM - Physical Data Model
- PdPRJ - Project
- 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:

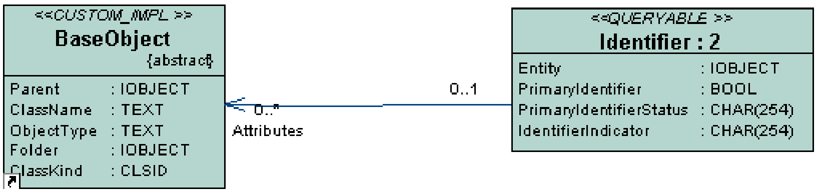
- *Public Names* - Each object in the metamodel has a name and a code corresponding to the public name of the object. The public name is the unique identifier of the object in a model library or package (for example, PdCommon) visible in the Modules diagram in the metamodel. Public names are used in the PowerDesigner XML model files (see *Model Files and the PowerDesigner Metamodel* on page 15) and in the GTL (see the *Chapter 5, Customizing Generation with GTL* on page 263).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 *Model Files and the PowerDesigner Metamodel* on page 15).
- *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.

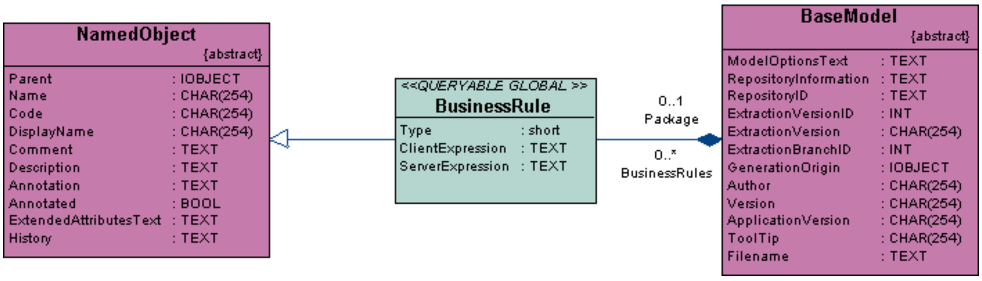
Navigating in the Metamodel

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 9).

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. 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. 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. 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.
Object operations	Operations in metamodel classes correspond to object methods used in VBS. 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. Example: RepositoryGroup

For more information about public names and other metamodel concepts, see *Metamodel concepts* on page 9.

For detailed information about using VB Script with PowerDesigner, see *Chapter 7, Scripting PowerDesigner* on page 323.

Accessing 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 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.

The following calculated attributes are metamodel extensions specific to GTL:

Metaclass	Attributes
PdCommon.BaseObject	• isSelected (boolean) - True if the object is part of the selection in the generation dialog • 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) • RoleAMaxMultiplicity (string) • RoleBMinMultiplicity (string) • RoleBMaxMultiplicity (string)

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

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

Model Files and the PowerDesigner Metamodel

PowerDesigner models are made up of objects, the properties and interactions of which are explained in the public metamodel. Models can be saved in either binary or XML file formats. Binary files are smaller and significantly quicker to open and save, but XML model files can be edited by hand or programatically (and DTDs are provided for each model type in the `DTD` folder in the installation directory).

Warning! You can modify an XML model file using a text or XML editor, but you should take care, as even a minor syntax error may render the file unusable. If you create an object in an XML file by copy and paste, make sure that you remove the duplicated OID. PowerDesigner will automatically assign an OID to the new object when next you open the model.

The following elements are used in PowerDesigner XML files:

- `<o:`*`object`*`>` - A PowerDesigner model object. The first time the object is mentioned in a collection, PowerDesigner assigns it an id using the `<o:`*`object`*` Id="`*`XYZ`*`">` syntax (where *XYZ* is a unique identifier automatically assigned to an object when it is found for the first time) or references it with the `<o:object Ref="`*`XYZ`*`"/>` syntax. Object definition is only used in composition collections, where the parent object owns the children in the association.
- `<c:`*`collection`*`>` - A collection of objects linked to another object. You can use the PowerDesigner metamodel to visualize the collections of an object. For example `<c:Children>`.
- `<a:`*`attribute`*`>` - An object is made up of a number of attributes each of which you can modify independently. For example `<a:ObjectID>`.

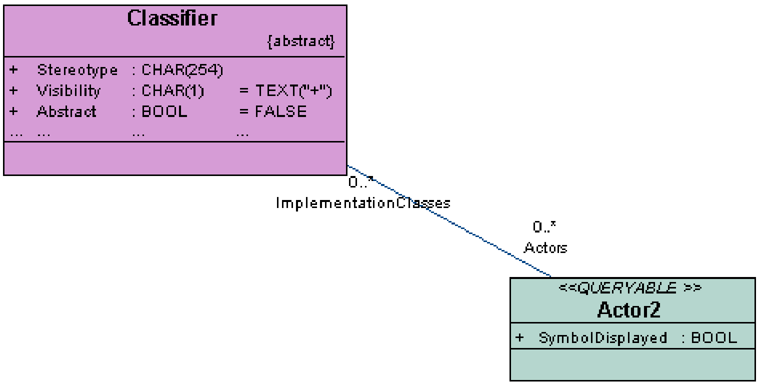
PowerDesigner XML model files have an `<o:model>` element at their root, which contains collections defined in the PowerDesigner metamodel. The model object and all the other object elements that it contains define their attributes and collections in sub-elements. The definition of an object implies the definition of its attributes and its collections. 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.

You can search for an object in the metamodel using its object name in the XML file in order to better understand its definition. Once you have found an object in the metamodel you can read the following information:

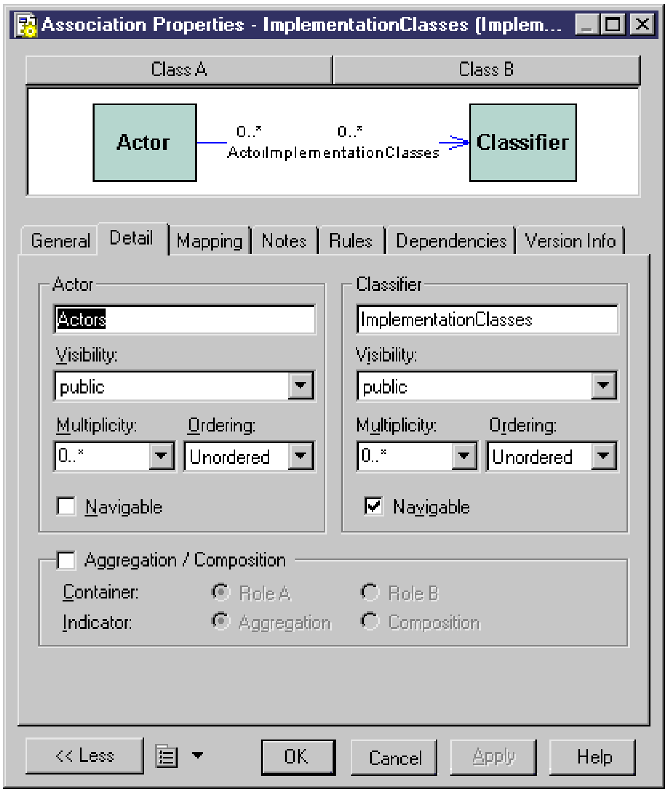
- Each PowerDesigner object can have several collections corresponding to other objects to interact with, these collections are represented by the associations existing between objects. The *roles* of the associations (aggregations and compositions included) correspond to the collections of an object. 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.



Actor: Actors
Visibility: public
Multiplicity: 0..*
Ordering: Unordered
[ ] Navigable

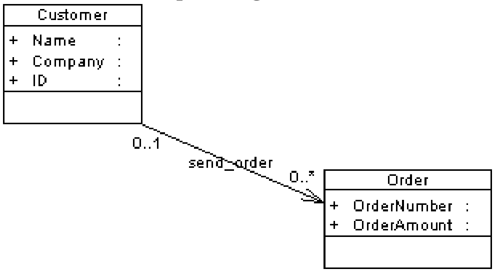
Classifier: ImplementationClasses
Visibility: public
Multiplicity: 0..*
Ordering: Unordered
[✓] Navigable

[ ] Aggregation / Composition
Container: [✓ Role A] Role B
Indicator: [✓ Aggregation] Composition

- 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>
```

CHAPTER 2 Extension Files

Extensions files (`*.xem`) provide means for customizing and extending PowerDesigner metaclasses, parameters and generation. You can use extensions to define additional properties for existing object types or entirely new kinds of objects, to modify the PowerDesigner interface (reorganizing and adding property sheet tabs, Toolbox tools and menu items), and to define additional generation targets and options.

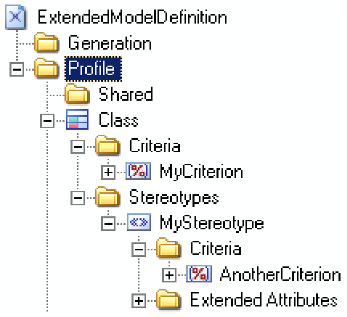
PowerDesigner provides a number of predefined extension files and you can also create your own. Each extension file contains two first-level categories:

- *Generation* - used to develop or complement the default PowerDesigner object generation (for BPM, OOM, and XSM models) or for separate generation. For more information, see *Generation Category* on page 122.
- *Profile* - 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 extend the metamodel in the following ways:
    - Add or sub-classify new kinds of objects:
        - Metaclasses – drawn from the metamodel as a basis for extension.
        - Stereotypes [for metaclasses and stereotypes only] – to sub-classify objects.
        - Criteria – to evaluate conditions to sub-classify objects.
        - Extended objects, sub-objects, and links – to create new kinds of objects.
    - Provide new ways of viewing connections between objects:
        - Dependency matrices – to show connections between two types of objects.
        - Extended collections and compositions – to enable manual linking between objects.
        - Calculated collections – to automate linking between objects.
    - Add new properties to objects and display them:
        - Extended attributes – to provide extra metadata.
        - Forms – to display custom property tabs or dialog boxes.
        - Custom symbols – to help you visually distinguish objects.

- Add constraints and validation rules to objects:
  - Custom checks – to provide data testing.
  - Event handlers – to invoke methods when triggered by an event.
- Execute commands on objects:
  - Methods – to be invoked by other profile extensions such as menus and form buttons (written in VBScript).
  - Menus [for metaclasses and stereotypes only] – to customize PowerDesigner menus.
- Generate objects in new ways:
  - Templates and generated files – to customize generation.
  - Transformations and Transformation profiles – to automate changes to objects at generation or on demand.

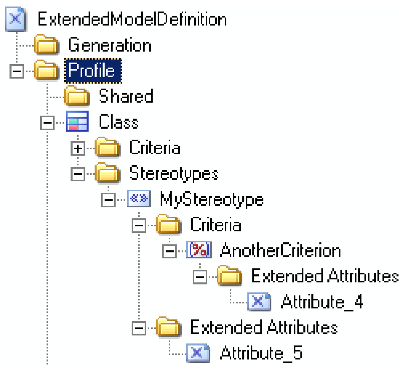
You can review and edit the profile in a resource file by opening it in the Resource Editor and expanding the top-level Profile category. You can add extensions to a metaclass (a type of object, such as Class in an OOM or Table in a PDM), or to a stereotype or criterion, which has previously been defined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

Note: Since you can attach several resource files to a model (for example, a target language and one or more extension files) 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 extension file extension usually prevails. When two XEMs are in conflict, priority is given to the one highest in the list.

Creating, Attaching, and Embedding Extension Files

Extensions can be in independent `*.xem` files that are attached to models or can be embedded in model files. Independent extension files can be referenced by multiple models, and any changes made to such a file are shared by all models that attach it. Changes made to extensions embedded in a model file affect only that model.

Creating an Extension File

You can create an extension file from the list of extension files or directly embedded in your model.

Note: For information about creating an extension file from the list of extension files, see *Creating and Copying Resource Files* on page 6.

1. Open your model, and then select **Model** > **Extensions** to open the List of Extensions.
2. Click the **Add a Row** tool and enter a name for the new extension file.
3. Click the **Properties** tool to open the new extension file in the Resource Editor, and create any appropriate extensions.
4. When you have finished, click **OK** to save your changes and return to the List of Extensions.

The new XEM is initially embedded in your model, and cannot be shared with any other model. For information about exporting your extensions and making them available for sharing, see *Exporting an Embedded Extension File for Sharing* on page 25.

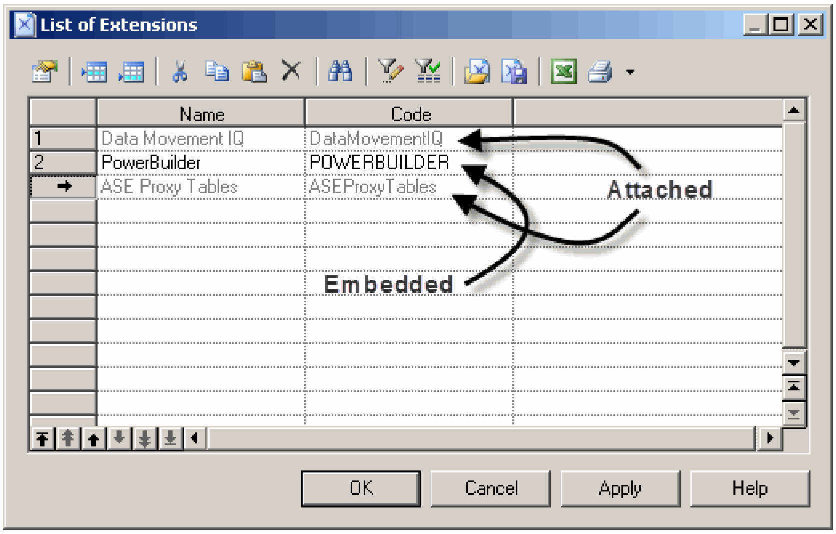
Attaching Extensions to a Model

You can attach an extension file (.xem) to your model at the creation of the model by clicking the **Select Extensions** button on the New Model dialog. You can subsequently attach an extension file to your model at any time from the List of Extensions.

Note: You should never modify the original extensions shipped with PowerDesigner. To create a copy of the file to modify, open the List of Extensions, click the **New** tool, specify a name for the new file, and then select the .xem that you want to modify in the **Copy from** field.

1. Select **Model** > **Extensions** to open the List of Extensions.
2. Click the **Import** tool to open the Select Extensions dialog.
3. Review the different sorts of extensions available by clicking the sub-tabs and select one or more to attach to your model.

   By default, PowerDesigner creates a link in the model to the specified file. To copy the contents of the extension file and save it in your model file, click the **Embed Resource in Model** button in the toolbar. Embedding a file in this way enables you to make changes specific to your model without affecting any other models that reference the shared resource.
4. Click **OK** to return to the List of Extensions.



	Name	Code
1	Data Movement IQ	DataMovementIQ
2	PowerBuilder	POWERBUILDER
➜	ASE Proxy Tables	ASEProxyTables

Extension files listed in grey are attached to the model, while those in black are embedded in the model.

Note: If you import an extension file and embed it in the model, the name and code of the extension may be modified in order to make it respect the naming conventions of the Other Objects category in the Model Options dialog.

Exporting an Embedded Extension File for Sharing

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

1. Select **Model** > **Extensions** to open the List of Extensions.
2. Select an extension in the list.
3. Click the **Export an Extension** tool.
4. Type a name and select a directory for the extension file.
5. Click Save.

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

Extension Properties

All extension filess have the same basic category structure.

The root node of each file contains the following properties:

Property	Description
Name	Specifies the name of the extension. This name must be unique in a model for generic or specific XEMs.
Code	Specifies the code of the extension. This code must be unique in a model for generic or specific XEMs.
File Name	[read-only] Specifies the path to the extension 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 Select Extensions dialog. Extensions having the same category cannot be generated simultaneously. If you do not specify a category, the XEM is displayed in the General Purpose category and is treated as a generation target.

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. 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. Note that PowerBuilder does not support XEMs for complementary generation.
Comment	Provides a descriptive comment for the XEM.

The following categories are also available:

- Generation - Contains Generation commands, options, and tasks to define and activate a generation process (see *Generation Category* on page 122).
- Transformation Profile - 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 *Transformations and Transformation Profiles (Profile)* on page 110. For information about invoking transformations, see *Core Features Guide > Linking and Synchronizing Models > Generating Models and Model Objects > Generating Models > Model Generation Options Window > Applying Model Transformations*

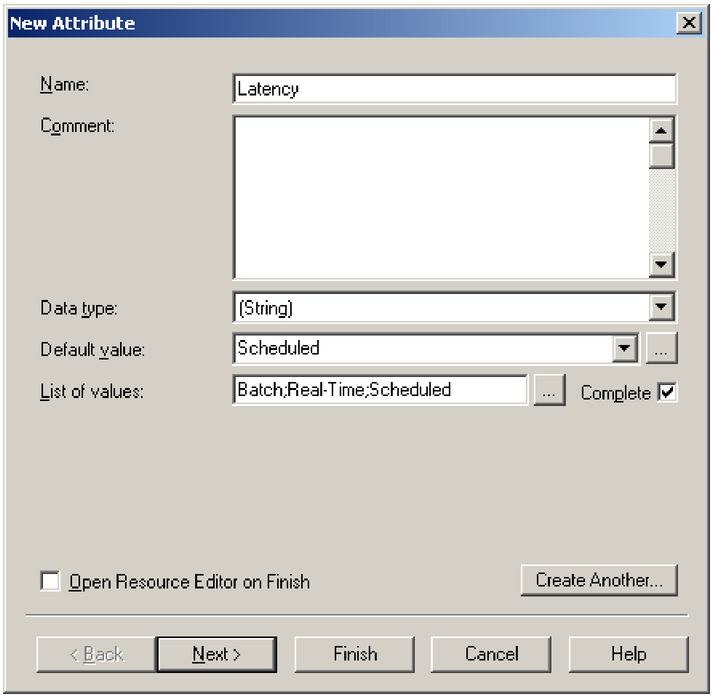
Example: Adding a New Attribute from a Property Sheet

In this example, we will quickly add a new attribute directly from the property sheet of an object. PowerDesigner will manage the creation of the extension file and creation of all the necessary extensions.

1. Click on the **Property Sheet Menu** button at the bottom-left of the property sheet, to the right of the **More/Less** button, and select **New Attribute**.
2. In the New Attribute dialog, enter `Latency` in the **Name** field, select `String` for the data type.
3. Click the ellipsis button to the right of the **List of values** field, enter the following list of predefined values, and then click **OK**:
   - Batch

- Real-Time
- Scheduled

4. [optional] Select `Scheduled` in the **Default value** field.
5. [optional] Click **Next** to specify the property sheet page where you want the new attribute to appear. Here, we'll leave the default, so its inserted on the **General** tab.



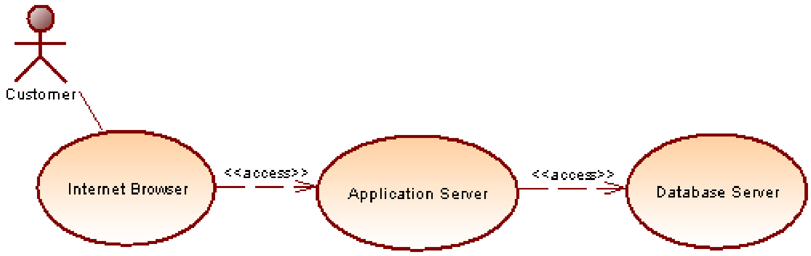
Name: Latency
Comment:
Data type: (String)
Default value: Scheduled
List of values: Batch;Real-Time;Scheduled
Complete ✓

Example: Creating Robustness Diagram Extensions

In this example, we will recreate the Robustness extension file delivered with PowerDesigner to extend the OOM communication diagram to enable robustness analysis. Robustness diagrams sit between use case and sequence diagram analysis, and allow you to bridge the gap between what the system has to do, and how it is actually going to accomplish it.

In order to support the robustness diagram, we will need to define new objects by applying stereotypes to a metaclass, specify custom tools and symbols for them, as well as defining custom checks for instance links and producing a file to output a description of messages exchanged between objects.

Creating the robustness extensions will enable us to verify use cases like the following, which represents a basic Web transaction:

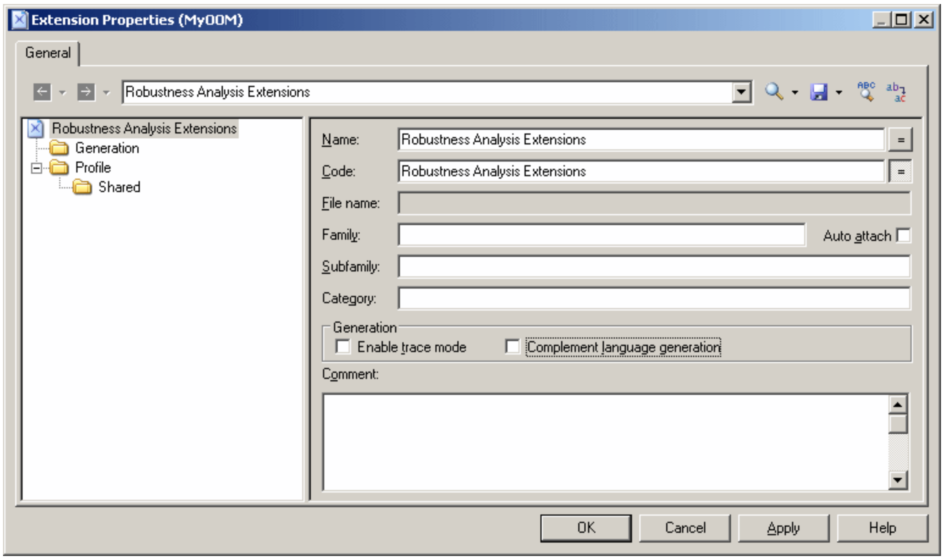


A customer wants to know the value of his stocks in order to decide to sell or not, and sends a stock value query from his Internet Browser, which is transferred from his browser to the database server via the application server.

Creating a New Extension File in your Model

The first step in defining extensions, is to create an extension file (.xem) to keep them in. To begin this process, you should create or open an OOM.

1. Select **Model** > **Extensions** to open the list of extensions attached to the model.
2. Click the **Add a Row** tool to create a new extension file, and then click the **Properties** tool to open it in the Resource Editor.
3. Enter `Robustness Analysis Extensions` in the **Name** field, and clear the **Complement language generation** check box, as these extensions do not belong to any object language family and will not be used to complement any object language generation.
4. Expand the Profile category, in which we will create the extensions:



Robustness Analysis Extensions

Name: Robustness Analysis Extensions

Code: Robustness Analysis Extensions

File name:

Family:

Auto attach ☐

Subfamily:

Category:

Generation

☐ Enable trace mode ☐ Complement language generation

Comment:

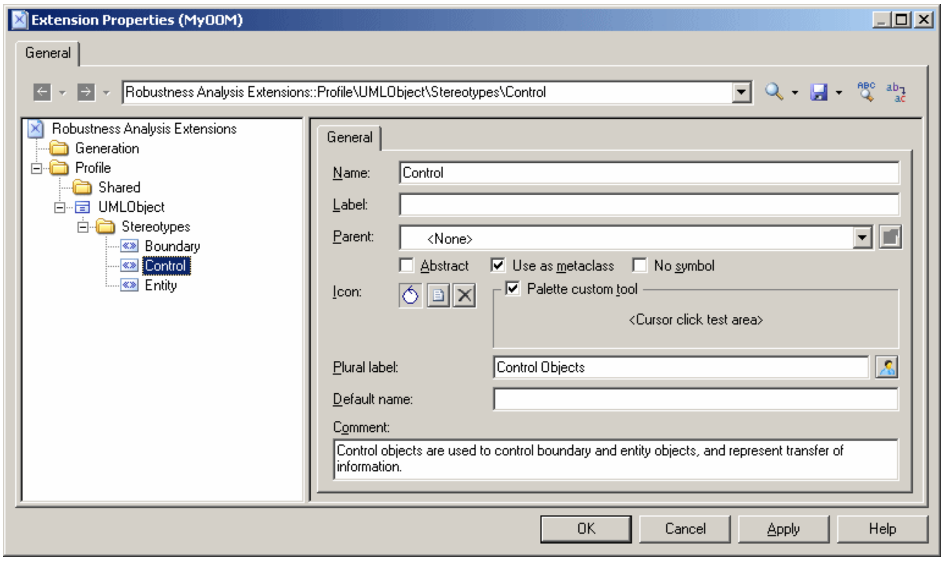
Creating New Objects with Stereotypes

To implement robustness analysis in PowerDesigner, we need to create three new types of objects (boundary, entity, and control objects), which we will define in the Profile category by extending the `UMLObject` metaclass through stereotypes.

1. Right-click the Profile category and select **Add Metaclasses** to open the Metaclass Selection dialog.
2. Select `UMLObject` on the PdOOM tab and click **OK** to add this metaclass to the extension file.

   Note: Click the **Find in Metamodel Objects Help** tool to the right of the **Name** field (or click **Ctrl+F1**) to obtain information about this metaclass and see where it is situated in the PowerDesigner metamodel.
3. Right-click the `UMLObject` category and select **New** > **Stereotype** to create a stereotype to extend this metaclass.
4. Enter `Boundary` in the **Name** field, and `Boundary objects are used by actors when communicating with the system; they can be windows, screens, dialog boxes or menus.` in the **Comment** field.
5. Select the **Use as metaclass** check box to promote the object type in the interface so that it has its own object list and Browser category.
6. Click the **Select Icon** tool to open the PowerDesigner image library dialog, select the **Search Images** tab, enter `boundary` in the **Search for** field, and click the **Search** button.
7. Select the `Boundary.cur` image in the results, and click OK to assign it to represent boundary objects in the Browser and other interface elements. Click the **Toolbox custom tool** check box to create a tool with the same icon for creating the new object in the Toolbox.
8. Repeat these steps to create the following stereotypes and icons:

Stereo-type	Comment	Image file
Entity	Entity objects represent stored data like a database, database tables, or any kind of transient object such as a search result.	entity.cur
Control	Control objects are used to control boundary and entity objects, and represent transfer of information.	control.cur



Name: Control

Parent: <None>

☐ Abstract ☑ Use as metaclass ☐ No symbol

☑ Palette custom tool

Plural label: Control Objects

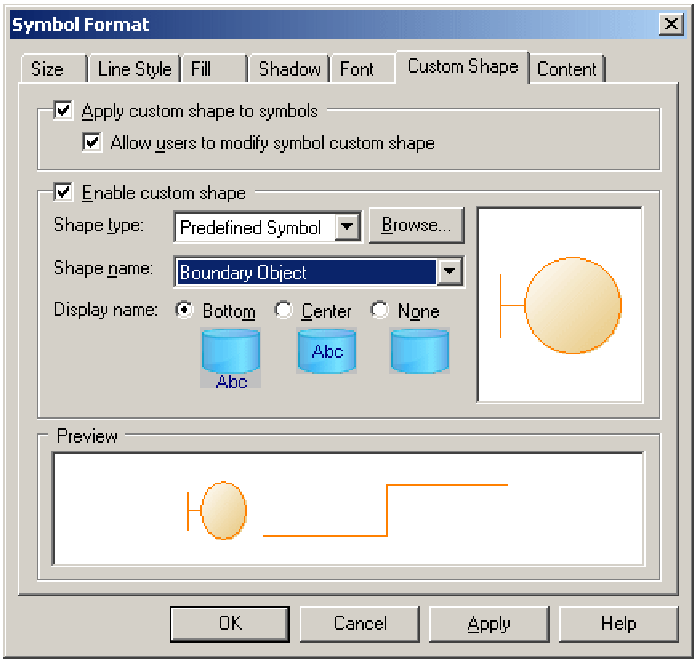
Comment: Control objects are used to control boundary and entity objects, and represent transfer of information.

9. Click **Apply** to save your changes before continuing.

Specifying Custom Symbols for Robustness Objects

We will specify diagram symbols for each of our new robustness diagram objects by adding custom symbols to our new stereotypes.

1. Right-click `Boundary` stereotype and select **New** > **Custom Symbol** to create a custom symbol under the stereotype.
2. Click the Modify button to open the Symbol Format dialog, and select the **Custom Shape** tab.
3. Select the **Enable custom shape** check box, and select Boundary Object in the **Shape name** list.

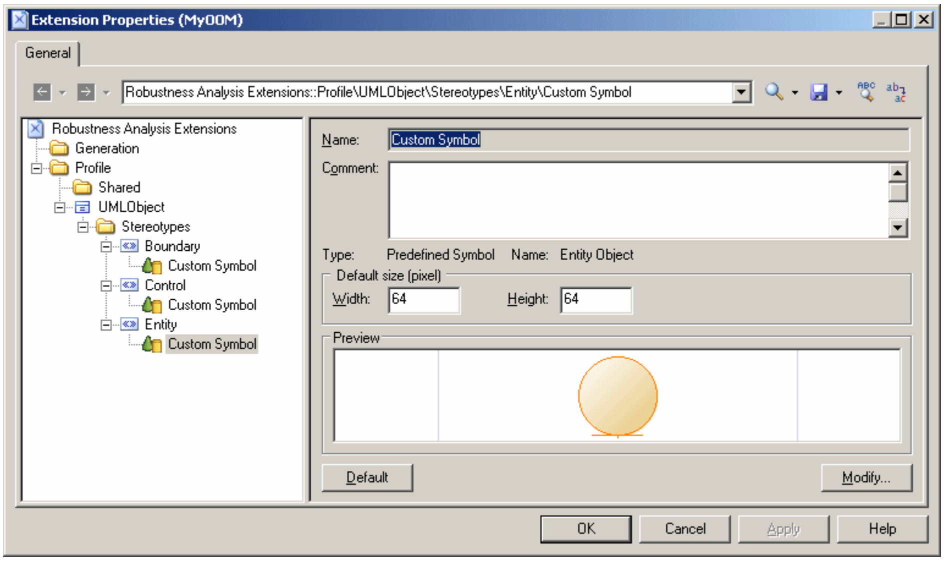


[✓] Apply custom shape to symbols
[✓] Allow users to modify symbol custom shape
[✓] Enable custom shape
Shape type: Predefined Symbol
Shape name: Boundary Object
Display name: [● Bottom] Center None

4. Click **OK** to complete the definition of the custom symbol and return to the Resource Editor.

5. Repeat these steps for the other stereotypes:

Stereotype	Shape Name
Entity	Entity Object
Control	Control Object



Name: Custom Symbol

Comment:

Type: Predefined Symbol Name: Entity Object

Default size (pixel)

Width: 64 Height: 64

Preview

6. Click **Apply** to save your changes before continuing.

Creating Custom Checks on Instance Links

We will now create three custom checks on the instance links that will connect the various robustness objects. These checks, which are written in VB, do not prevent users from creating diagrams not supported by the robustness methodology, but define rules that will be verified by the Check Model function.

For more information on VBS syntax, see *Chapter 7, Scripting PowerDesigner* on page 323.

1. Right-click the Profile category, select **Add Metaclasses** to open the Metaclass Selection dialog, select `InstanceLink` on the PdOOM tab and click **OK** to add it to the extension file.
2. Right-click the `InstanceLink` category and select **New** > **Custom Check** to create a check under the metaclass.
3. Enter the following values for the properties on the **General** tab:

Field	Value
Name	`Incorrect Actor Collaboration`
Comment	`This check verifies if actors are linked to boundary objects. Linking actors to control or entity objects is not allowed in the robustness analysis.`

Field	Value
Help message	`This check ensures that actors only communicate with boundary objects.`
Output message	`The following instance links are incorrect:`
Default severity	`Error`
Execute the check by default	[selected]

4. Select the **Check Script** tab and enter the following script in the text field:

```
Function %Check%(link)
   ' Default return is True
   %Check% = True

   ' The object must be an instance link
   If link is Nothing then
      Exit Function
   End if
   If 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
   ElseIf 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
```

5. Select the **Global Script** tab (where you store functions and static attributes that may be reused among different functions) and enter the following script in the text field:

```
' 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 it's target object
   If Obj.IsShortcut() Then
      CompareObjectKind = CompareObjectKind(Obj.TargetObject,
Kind)
      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 it's stereotype value
Function CompareStereotype(Obj, Kind, Value)
   ' Default return is false
   CompareStereotype = False

   ' Check object
   If Obj is Nothing then
      Exit Function
   End If
   if (not Obj.IsShortcut() and not
Obj.HasAttribute("Stereotype")) Then
      Exit Function
   End If
   ' Shortcut specific case, ask to it's target object
   If Obj.IsShortcut() Then
      CompareStereotype = CompareStereotype(Obj.TargetObject,
Kind, Value)
      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

' This global function copy the standard attribute
' from source to target
Function Copy (src, trgt)
   trgt.name = src.name
   trgt.code  = src.code
   trgt.comment = src.comment
   trgt.description = src.description
   trgt.annotation = src.annotation
   Dim b, d
   for each b in src.AttachedRules
      trgt.AttachedRules.insert -1,b
   next
   for each d in src.RelatedDiagrams
      trgt.RelatedDiagrams.insert -1,d
   next
   output " "
   output trgt.Classname & " " & trgt.name & " has been created."
   output " "
End Function
```

6. Repeat these steps to create a second check by entering the following values:

Field	Value
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]

Field	Value

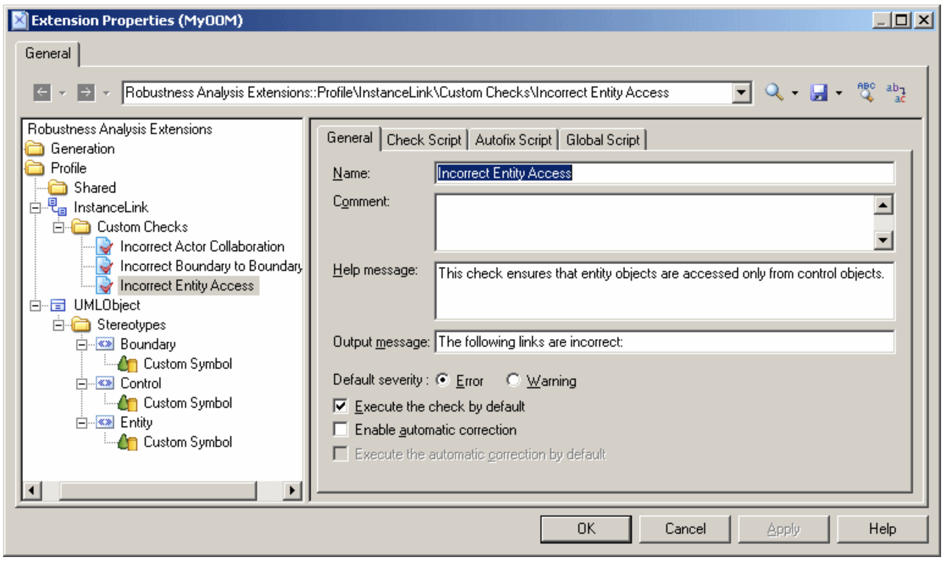
Check Script	
	```
	Function %Check%(link)
	   ' Default return is True
	   %Check% = True

	   ' The object must be an instance link
	   If link is Nothing then
	      Exit Function
	   End if
	   If 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, "Boun-
	dary") Then
	      If CompareStereotype(dst, PdOOM.Cls_UMLObject,
	"Boundary") Then
	         %Check% = False
	      End If
	   End If
	End Function
	```

7. Repeat these steps to create a third check by entering the following values:

Field	Value
Name	`Incorrect Entity Access`
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]

Field	Value

Check Script	Function %Check%(link)
	   ' Default return is True
	   %Check% = True
	
	   ' The object must be an instance link
	   If link is Nothing then
	      Exit Function
	   End if
	   If 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, "Enti-
	ty") Then
	      ' Check if destination is an UML Object with "Con-
	trol" Stereotype
	     If not CompareStereotype(dst, PdOOM.Cls_UMLObject,
	"Control") Then
	         %Check% = False
	      End If
	   ElseIf 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



Robustness Analysis Extensions::Profile\InstanceLink\Custom Checks\Incorrect Entity Access

- Robustness Analysis Extensions
  - Generation
  - Profile
    - Shared
    - InstanceLink
      - Custom Checks
        - Incorrect Actor Collaboration
        - Incorrect Boundary to Boundary
        - Incorrect Entity Access
    - UMLObject
      - Stereotypes
        - Boundary
          - Custom Symbol
        - Control
          - Custom Symbol
        - Entity
          - Custom Symbol

Name: Incorrect Entity Access

Comment:

Help message: This check ensures that entity objects are accessed only from control objects.

Output message: The following links are incorrect:

Default severity : [✓ Error] Warning

[✓] Execute the check by default

[ ] Enable automatic correction

[ ] Execute the automatic correction by default

8. Click **Apply** to save your changes before continuing.

Defining Templates to Extract Message Descriptions

We are going to generate a textual description of the messages in the diagram, giving for each message, the names of the sender, message, and receiver. To do so, we will need to define PowerDesigner Generation Template Language (GTL) templates to extract the information and a generated file to contain and display the extracted information.

To generate this textual description, we will need to extract information from the following metaclasses:

- *Message* - to extract the message sequence number, name, sender, and receiver
- *CommunicationDiagram* - to gather all the messages from each diagram and sort them

1. Right-click the Profile category, select **Add Metaclasses** to open the Metaclass Selection dialog, select `CommunicationDiagram` and `Message` on the PdOOM tab and click **OK** to add them to the extension file.
2. Right-click the `Message` category and select **New** > **Template** to create a template under the metaclass.
3. Enter `description` in the **Name** field, and then enter the following GTL code in the text area:

```
.set_value(_tabs, "", new)
.foreach_part(%SequenceNumber%, '.')
   .set_value(_tabs, "   %_tabs%")
.next
%_tabs%%SequenceNumber%) %Sender.ShortDescription% sends message
"%Name%" to %Receiver.ShortDescription%
```

The first line of the template initializes the `_tabs` variable, and the `foreach_part` macro calculates an appropriate amount of indentation by looping through each sequence number, and adding 3 spaces whenever a dot is found. The last line uses this variable to indent, format, and display information extracted for each message.

For detailed information about the PowerDesigner Generation Template Language (GTL), see *Chapter 5, Customizing Generation with GTL* on page 263.

4. Right-click the `CommunicationDiagram` category and select **New** > **Template** to create a template under the metaclass.
5. Enter `compareCbMsgSymbols` in the **Name** field, and then enter the following GTL code in the text area:

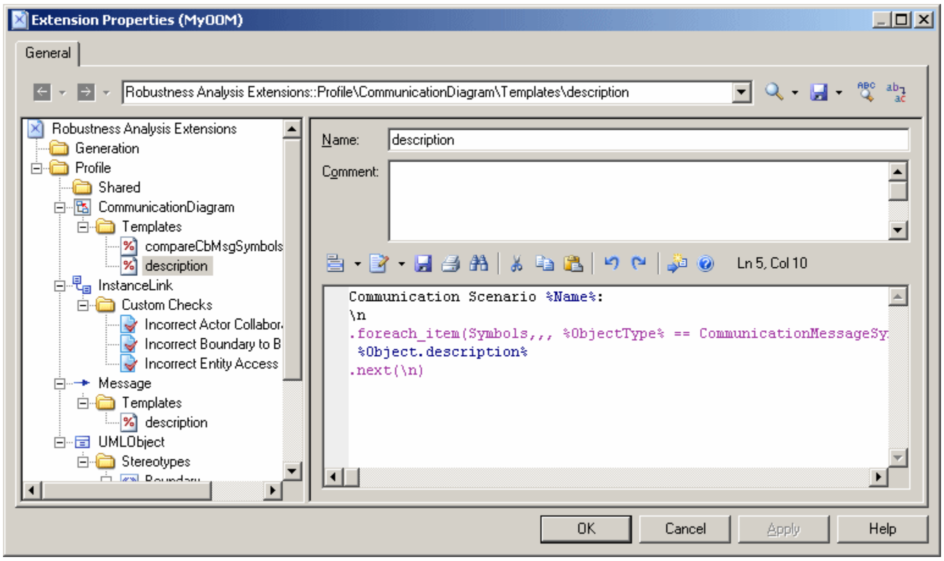
```
.bool (%Item1.Object.SequenceNumber% >=
%Item2.Object.SequenceNumber%)
```

This template resolves to a boolean value to determine if one message number is greater than another, and the result will be used in a second template.

6. Right-click the `CommunicationDiagram` category and select **New** > **Template** to create a second template, enter `description` in the **Name** field, and then enter the following GTL code in the text area:

```
Collaboration Scenario %Name%:
\n
.foreach_item(Symbols,,, %ObjectType% ==
CollaborationMessageSymbol, %compareCbMsgSymbols%)
  %Object.description%
.next(\n)
```

The first line of this template generate the title of the scenario from the name of the communication diagram. Then the `.foreach_item` macro loops on each message symbol, and calls on the other templates to format and output the message information.



Name: description

```
Communication Scenario %Name%:
\n
.foreach_item(Symbols,,, %ObjectType% == CommunicationMessageSy
 %Object.description%
.next(\n)
```

7. Click **Apply** to save your changes before continuing.

Creating a Generated File for the Message Information

Having created templates to extract information about the messages in the model, we need to create a generated file to contain and display them on the **Preview** tab of the diagram property sheet. We will define the file on the `BasePackage` metaclass, which is the common class for all packages and models, and will have it loop through all the communication diagrams in the model to evaluate the template `description` defined on the `CommunicationDiagram` metaclass.

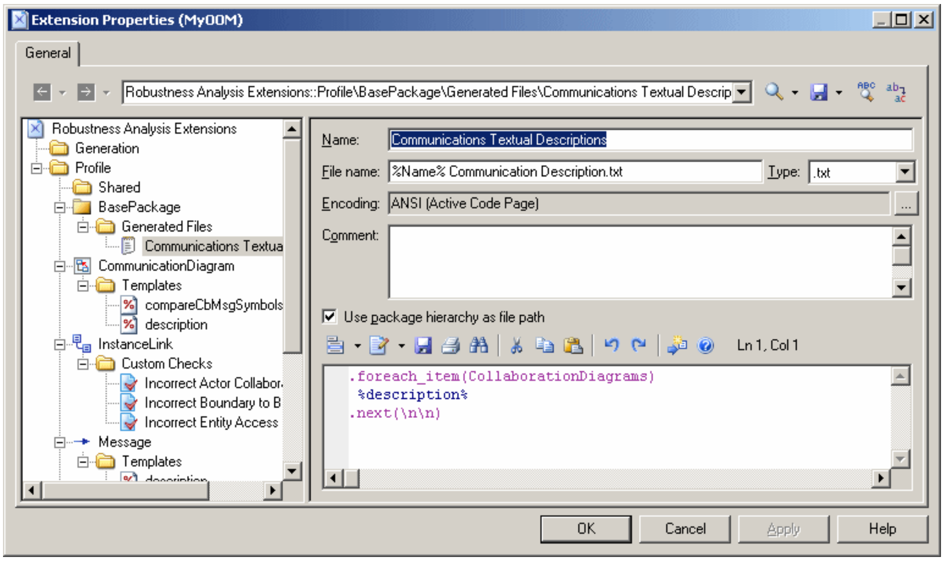
1. Right-click the Profile category, select **Add Metaclasses** to open the Metaclass Selection dialog, click the **Modify Metaclass Filter** tool, select `Show Abstract Modeling Metaclasses`, and click the PdCommon tab.
2. Select `BasePackage` and click **OK** to add it to the extension file.
3. Right-click the BasePackage category and select **New** > **Generated File** to create a file under the metaclass.
4. Enter the following values for the file properties:

Field	Value
Name	`Communications Textual Descriptions`
File name	`%Name% Communication Description.txt`
Encoding	`ANSI`

Field	Value
Use package hierarchy as file path	[unselected]

5. Enter the following code in the text box:

```
.foreach_item(CollaborationDiagrams)
 %description%
.next(\n\n)
```



Name: Communications Textual Descriptions

File name: %Name% Communication Description.txt

Type: .txt

Encoding: ANSI (Active Code Page)

Comment:

☑ Use package hierarchy as file path

```
.foreach_item(CollaborationDiagrams)
 %description%
.next(\n\n)
```

6. Click **Apply** to save your changes, and then **OK** to close the resource editor.
7. Click **OK** to close the List of Extensions.

Testing the Robustness Extensions

To test the extensions we have created, we will create a small robustness diagram to analyze our use case.

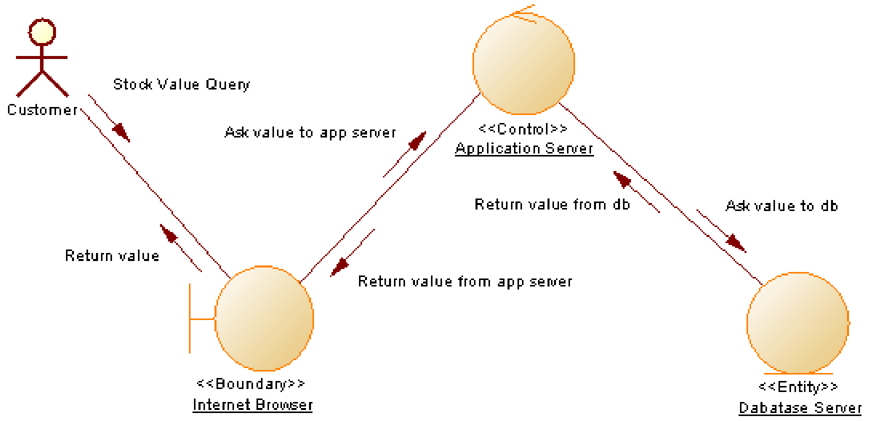
1. Right-click your model node in the Browser, and select **New > Communication Diagram**.

   In addition to the standard Toolbox, a custom toolbox is provided with tools you have defined to create boundary, control, and entity objects.
2. Drag the Customer actor from the Actors category in the Browser into the diagram to create a shortcut. Then create one each of the boundary, control and entity objects, and name them `Internet Browser`, `Application Server`, and `Database Server` respectively.

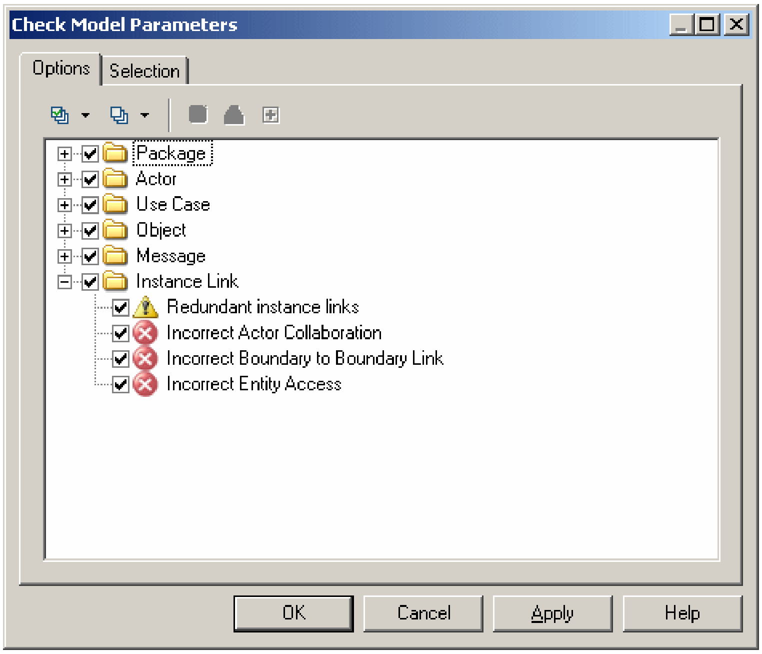
3. Use the Instance Link tool in the standard Toolbox to connect the Customer to the `Internet Browser` to the `Application Server`, to the `Database Server`.

4. Create the following messages on the **Messages** tabs of the instance links property sheets:

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

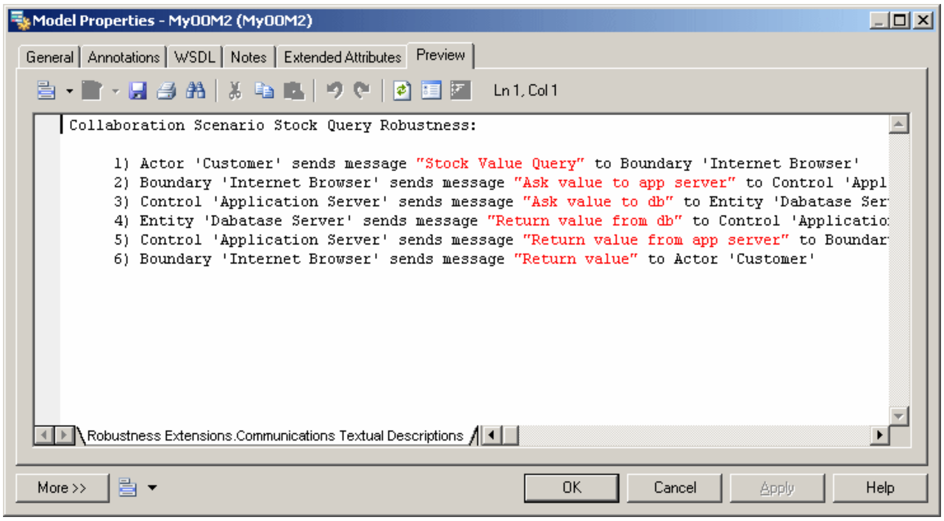


5. Select **Tools > Check Model** to display the Check Model Parameters dialog box, in which the custom checks we have created appear in the Instance Link category:



Click **OK** to test the validity of the instance links we have created.

6. Right-click the model node in the Browser and select **Properties** to open the model property sheet. Click the **Preview** tab to review messages sent for our use case:


Model Properties - MyOOM2 (MyOOM2)
General
Annotations
WSDL
Notes
Extended Attributes
Preview
Ln 1, Col 1
Robustness Extensions.Communications Textual Descriptions
More >>
OK
Cancel
Apply
Help

```
Collaboration Scenario Stock Query Robustness:

    1) Actor 'Customer' sends message "Stock Value Query" to Boundary 'Internet Browser'
    2) Boundary 'Internet Browser' sends message "Ask value to app server" to Control 'Appl
    3) Control 'Application Server' sends message "Ask value to db" to Entity 'Dabatase Ser
    4) Entity 'Dabatase Server' sends message "Return value from db" to Control 'Applicatio
    5) Control 'Application Server' sends message "Return value from app server" to Boundar
    6) Boundary 'Internet Browser' sends message "Return value" to Actor 'Customer'
```

Extending Generation and Creating Separate Generation Targets

Extensions can be used to extend generation and create new generation targets.

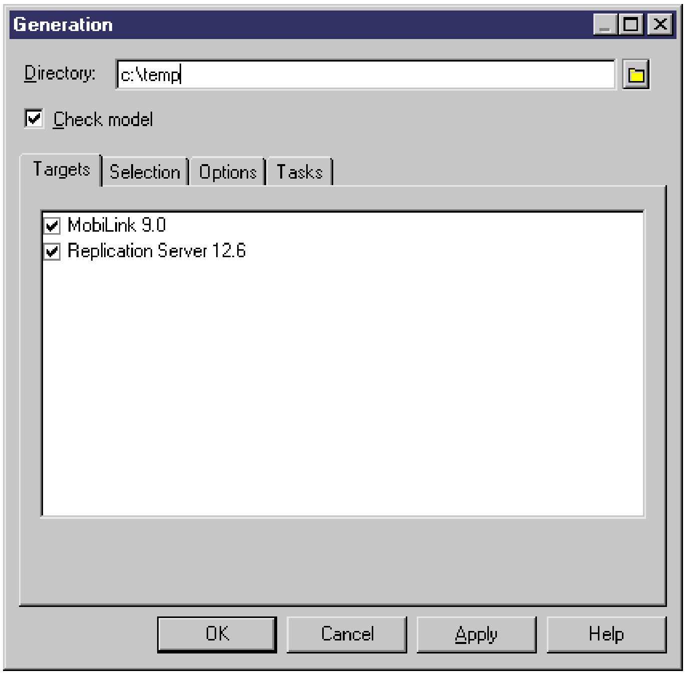
The following table shows how you can customize the standard BPM, OOM, or XSM generation from the Resource Editor:

Generation dialog	Extension
Targets page	The Target page is displayed if the **Complement language generation** property is selected (see *Extension Properties* on page 25) and if the extension contains at least one task or generated file.
Options page	Define options in `Generation\Options`.
Tasks page	Define commands in `Generation\Commands` and reference these commands in tasks.

If you want to create separate generation targets (available from the **Tools > Extended Generation** command), you must respect the following conditions:

- The **Complement Language Generation** check box in the extension property sheet must not be selected
- The extension 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*. If you have several extensions designed for extended generation, these will appear in the Targets page of the extended generation dialog box.



Directory: c:\temp

☑ Check model

☑ MobiLink 9.0
☑ Replication Server 12.6

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 (see *Menus (Profile)* on page 100) under the Model metaclass in the Profile category of the extension, and select the `Tools menu` in the **Location** list
- Create a method (see *Methods (Profile)* on page 97) to invoke extended generation as follows:

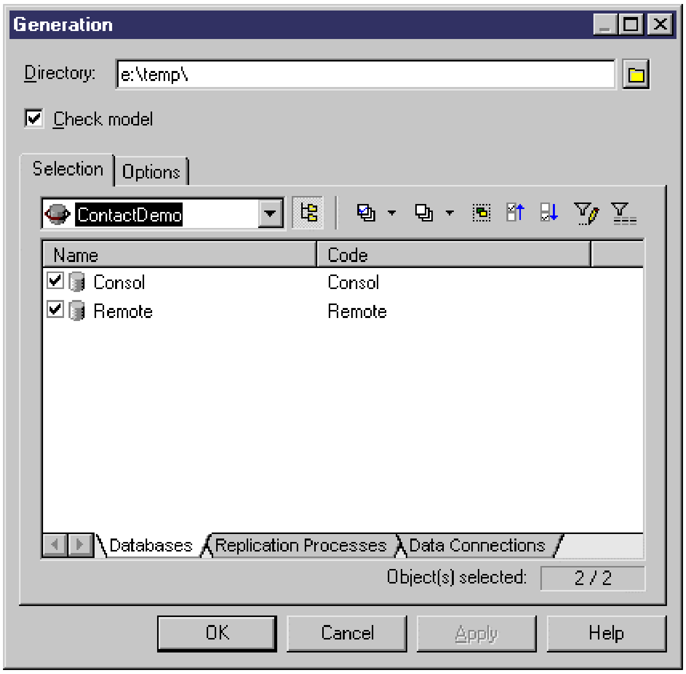
```
Sub %Method%(obj)

 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.

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

- Save the extension

The new command is displayed in the **Tools** menu.



Generation

Directory: e:\temp\

☑ Check model

Selection | Options

ContactDemo

Name	Code
☑ Consol	Consol
☑ Remote	Remote

Databases | Replication Processes | Data Connections

Object(s) selected: 2 / 2

OK | Cancel | Apply | Help

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

Metaclasses (Profile)

Metaclasses are classes drawn from the PowerDesigner metamodel, and appear at the top level of the Profile category. You add a metaclass to a profile when you want to extend it.

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.

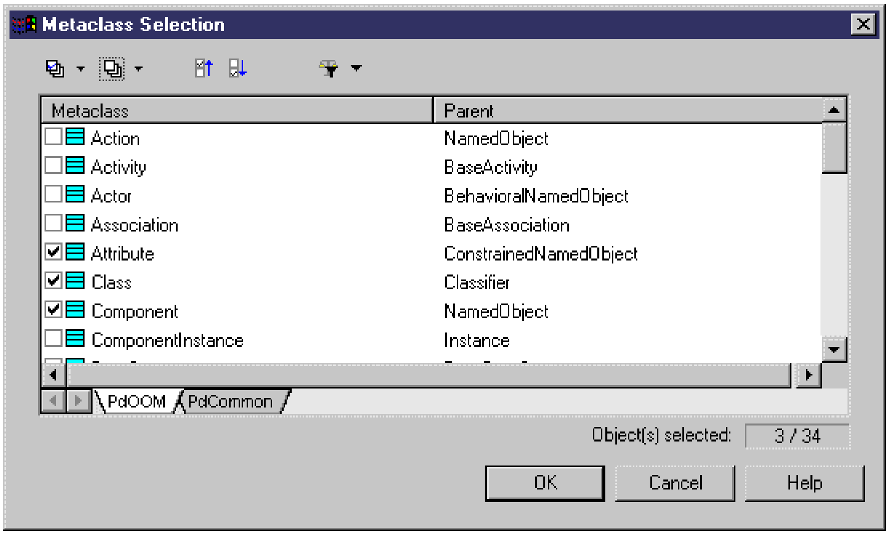
For information about viewing and navigating among metaclasses in the metamodel, see *The PowerDesigner Public Metamodel* on page 8.

If you do not want to extend an existing metaclass, but rather create an entirely new kind of modeling object, you should use the extended object metaclass (see *Extended Objects, Sub-Objects, and Links (Profile)* on page 54).

Adding a Metaclass to a Profile

You add a metaclass to a profile in order to define extensions for it.

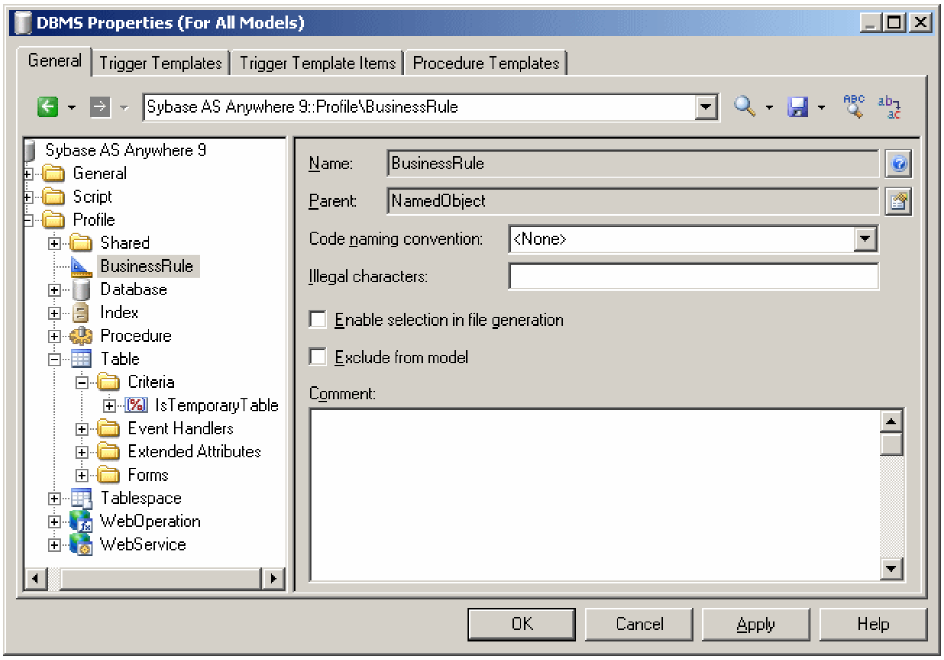
1. Right-click the Profile category and select Add Metaclasses from the contextual menu to open the Metaclass Selection dialog box:



Metaclass	Parent
☐ Action	NamedObject
☐ Activity	BaseActivity
☐ Actor	BehavioralNamedObject
☐ Association	BaseAssociation
✓ Attribute	ConstrainedNamedObject
✓ Class	Classifier
✓ Component	NamedObject
☐ ComponentInstance	Instance

Object(s) selected: 3 / 34

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:



Name: BusinessRule

Parent: NamedObject

Code naming convention: <None>

Illegal characters:

☐ Enable selection in file generation

☐ Exclude from model

Comment:

Metaclass Properties

You specify the properties for a metaclass by selecting its entry in the resource editor.

Property	Description
Name	[read-only] Specifies the name of the metaclass. Click the button to the right of this field to open the Metamodel Objects Help for the metaclass.
Parent	[read-only] Specifies the parent of the metaclass. 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 in target files only] Specifies the default format to initialize the name to code conversion script for instances of the metaclass. The following formats are available: • `firstLowerWord` - First word in lowercase, then other first letters of other words in uppercase • `FirstUpperChar` - First character of all words in uppercase • `lower_case` - All words in lowercase and separated by an underscore • `UPPER_CASE` - All words in uppercase and separated by an underscore For more information on conversion scripts and naming conventions, see *Core Features Guide > The PowerDesigner Interface > Objects > Object Properties > Naming Conventions*.

Property	Description
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: `"/!=<>""'()"` 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 120).
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, Toolbox, 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. 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 toolbox tool. For more information, see *Promoting a Stereotype to Metaclass Status* on page 51.

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.

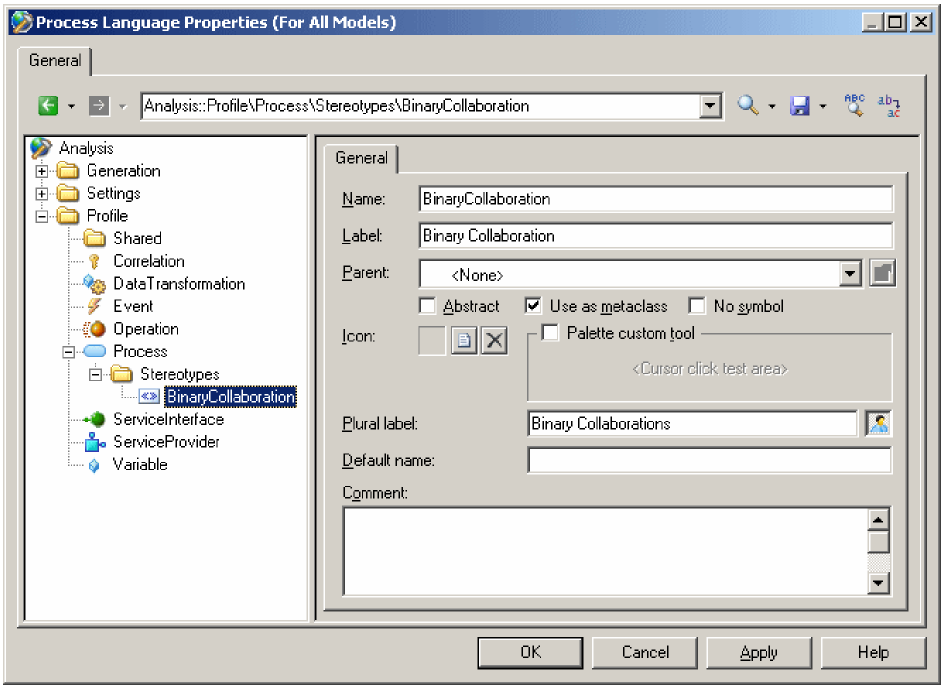
Creating a Stereotype

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**.

   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.



Name: BinaryCollaboration
Label: Binary Collaboration
Parent: <None>
☐ Abstract ☑ Use as metaclass ☐ No symbol
Icon:
☐ Palette custom tool
Plural label: Binary Collaborations
Default name:

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

You specify the properties for a stereotype by selecting its entry in the resource editor.

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.

Property	Description
Use as meta-class	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 *Promoting a Stereotype to Metaclass Status* on page 51.
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 Toolbox custom tool option is disabled when this option is selected.
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.
Toolbox custom tool	Associates a tool in a toolbox 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 *Specifying an Icon and Custom Tool for a Stereotype* on page 52.
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 **Toolbox 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. 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.

Promoting a Stereotype to Metaclass Status

You can promote a stereotype to metaclass status by selecting **Use as Metaclass** in the stereotype property page. Stereotypes promoted in this way have their own Browser folders and entries in the Model menu and New contextual menu.

You can use such stereotypes 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.

1. In the Stereotype property page, select **Use as metaclass**.

   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 under **New**, when you right-click the model or a package.
- Property sheet titles based on the metaclass stereotype.

2. [optional] Specify an icon and tool to create instances of the metaclass stereotype (see *Specifying an Icon and Custom Tool for a Stereotype* on page 52).

3. Click **Apply** to save the changes.

Specifying an Icon and Custom Tool for a Stereotype

You can specify an icon for a stereotype to allow users to identify instances of the metaclass bearing the stereotype in the Browser, property sheets, and elsewhere in the interface. You can also specify a tool to allow user to create instances from the Toolbox.

1. In the Stereotype property page, click the **Select Icon** tool to open the Select Image dialog. Select an appropriate image to use as an icon in the interface and click **OK** to associate it with the stereotype.

Note: The icon is used to identify objects in the Browser and elsewhere in the interface, but is not used as a diagram symbol. To specify a custom diagram symbol, see *Custom Symbols (Profile)* on page 86.

2. [optional] To enable a tool to create instances of the metaclass bearing the stereotype from the Toolbox, select **Toolbox custom tool**. If you do not select this option, users are only able to create instances of the metaclass bearing the stereotype from the Model menu or by right-clicking the model. Custom tools appear in a separate Toolbox group named after the resource file in which they are defined.

Note: If you have not specified an icon, the tool will use a hammer icon by default.

3. [optional] Click inside the `<Cursor Click Test Area>` to preview the look of the tool cursor.

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.

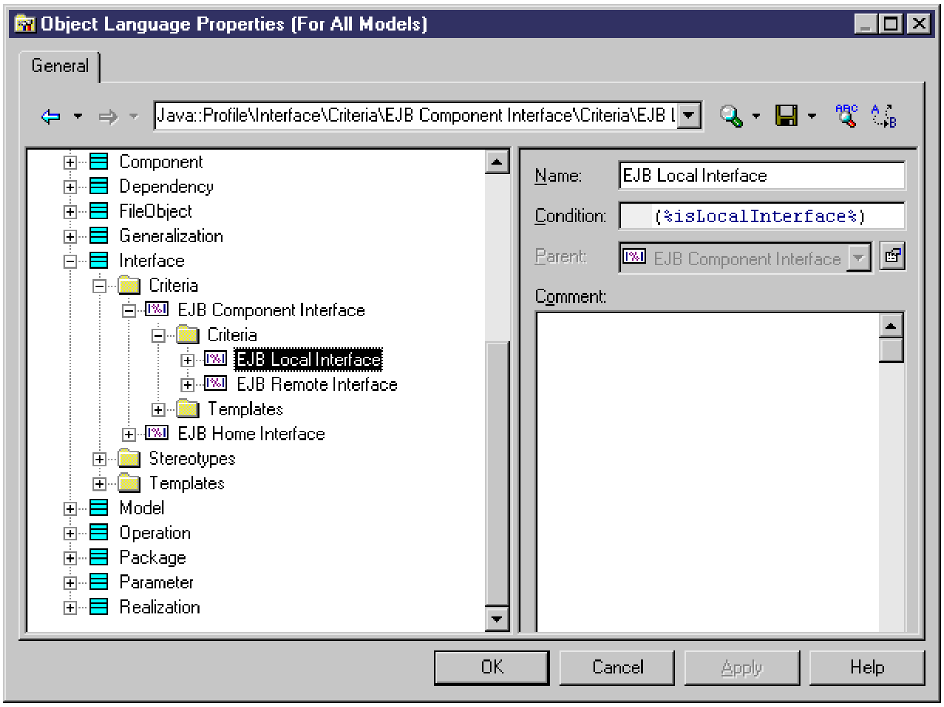
Creating a Criterion

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 297).



Object Language Properties (For All Models)

General

Java::Profile\Interface\Criteria\EJB Component Interface\Criteria\EJB L

- Component
- Dependency
- FileObject
- Generalization
- Interface
  - Criteria
    - EJB Component Interface
      - Criteria
        - EJB Local Interface
        - EJB Remote Interface
      - Templates
    - EJB Home Interface
  - Stereotypes
  - Templates
- Model
- Operation
- Package
- Parameter
- Realization

Name: EJB Local Interface

Condition: (%isLocalInterface%)

Parent: EJB Component Interface

Comment:

3. Click Apply to save your changes.

Criterion Properties

You specify the properties for a criterion by selecting its entry in the resource editor.

Property	Description
Name	Specifies the name of the criterion.

Property	Description
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 *Chapter 5, Customizing Generation with GTL* on page 263. You can reference the extended attributes defined at the metaclass level in the condition, but not those defined in the criterion itself. 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: `(%DimensionalType% == "1")` `%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	Specifies additional information about the criterion.

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 46. 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 65)
- Extended links – can be defined to link extended objects

Adding Extended Objects, Sub-Objects, and Links to a Profile

Extended objects, sub-objects, and links do not appear, by default, in models other than the free model unless you add them to an extension or other resource file.

1. Right-click the **Profile** category, select **Add Metaclasses**, and click the **PdCommon** sub-tab in the dialog to display the list of objects common to all models.
2. Select one or more of `ExtendedLink`, `ExtendedSubObject`, and `ExtendedObject` and click **OK** to add them to your profile.

Note: To make the tools for creating extended objects and extended links available in the Toolbox of models other than the free model, you must add them via the customization dialog available at **Tools > Customize Menus and Tools**.

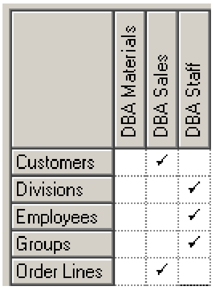
3. [optional] To create your own object add a stereotypes (see *Stereotypes (Profile)* on page 49 and define appropriate extensions under the stereotype. To have your object appear in the PowerDesigner interface as a standard metaclass, with its own tool, Browser category and model list, select **Use as metaclass** in the stereotype definition (see *Promoting a Stereotype to Metaclass Status* on page 51).
4. Click **Apply** to save the changes.

Dependency Matrices (Profile)

Dependency matrices allow you to review and create links between any kind of objects. You specify one metaclass for the matrix rows, and the same or another metaclass for the columns. The contents of 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



	DBA Materials	DBA Sales	DBA Staff
Customers		✓	
Divisions			✓
Employees			✓
Groups			✓
Order Lines		✓	

- PDM Tables and OOM Classes – connected by extended dependencies

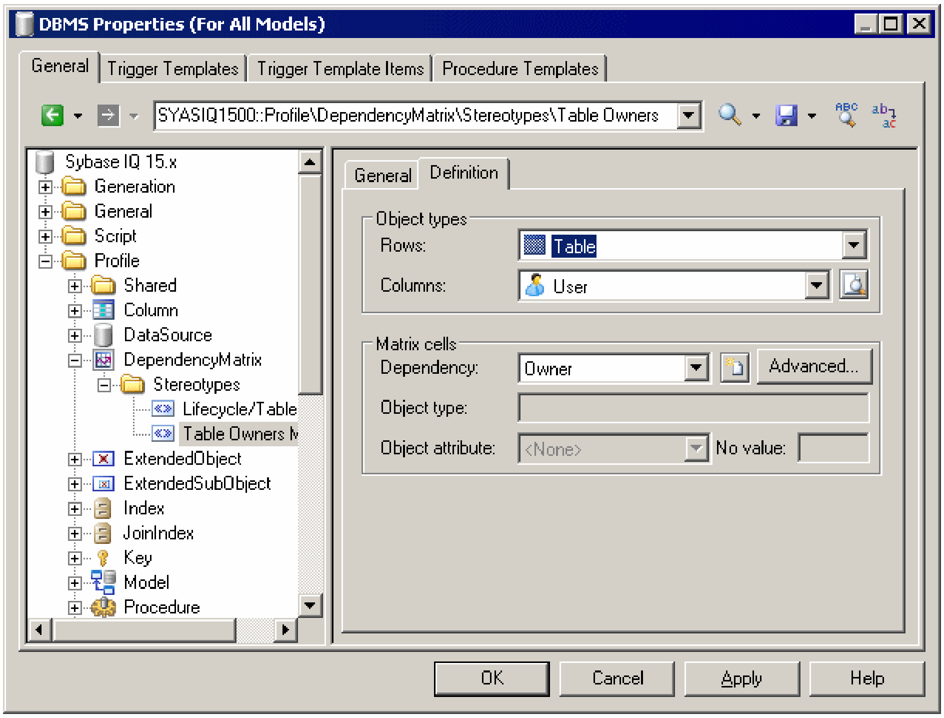
Creating a Dependency Matrix

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 57).



Object types
Rows: Table
Columns: User

Matrix cells
Dependency: Owner
Object type:
Object attribute: <None>
No value:

6. For certain dependencies, the **Object type** on which the dependency is based will be displayed, and you can select an **Object attribute** to display in the matrix cells along with the **No value** symbol, which is displayed if that attribute is not set in any particular instance.
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 *Core Features Guide > The PowerDesigner Interface > Diagrams, Matrices, and Symbols > Dependency Matrices.*

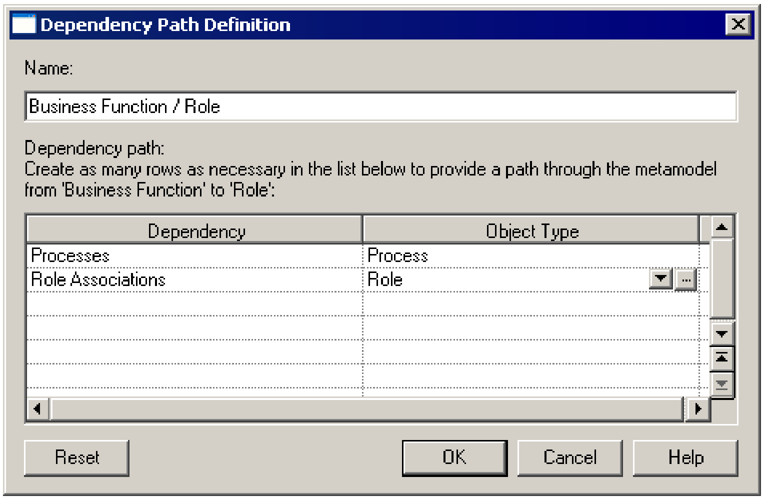
Specifying Advanced Dependencies

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.
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:



Name:
Business Function / Role

Dependency path:
Create as many rows as necessary in the list below to provide a path through the metamodel from 'Business Function' to 'Role':

Dependency	Object Type
Processes	Process
Role Associations	Role

Dependency Matrix Properties

You specify the properties for a dependency matrix by selecting its entry in the resource editor.

Dependency matrices are based on stereotypes. For information about the properties on the **General** tab, see *Stereotype properties* on page 50. The following properties are available on the **Dependency Matrix Definition** tab:

Property	Description
Rows	Specifies the object type with which to populate your matrix rows.
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.
Matrix Cells	Specifies how the rows and columns of your matrix will be associated. You must specify a **Dependency** from the list, which includes all the collections and links available to the object. For certain dependencies, the **Object type** on which the dependency is based will be displayed, and you can select an **Object attribute** to display in the matrix cells along with the **No value** symbol, which is displayed if that attribute is not set in any particular instance. Click the **Create** button to the right of the list to create a new extended collection (see *Extended Collections and Compositions (Profile)* on page 65) connecting your objects, or the **Advanced** button to specify a complex dependency path (see *Specifying Advanced Dependencies* on page 57).

Extended Attributes (Profile)

Extended attributes allow you to define additional metadata for your objects.

They can be defined for metaclasses, stereotypes, and criteria, in order to:

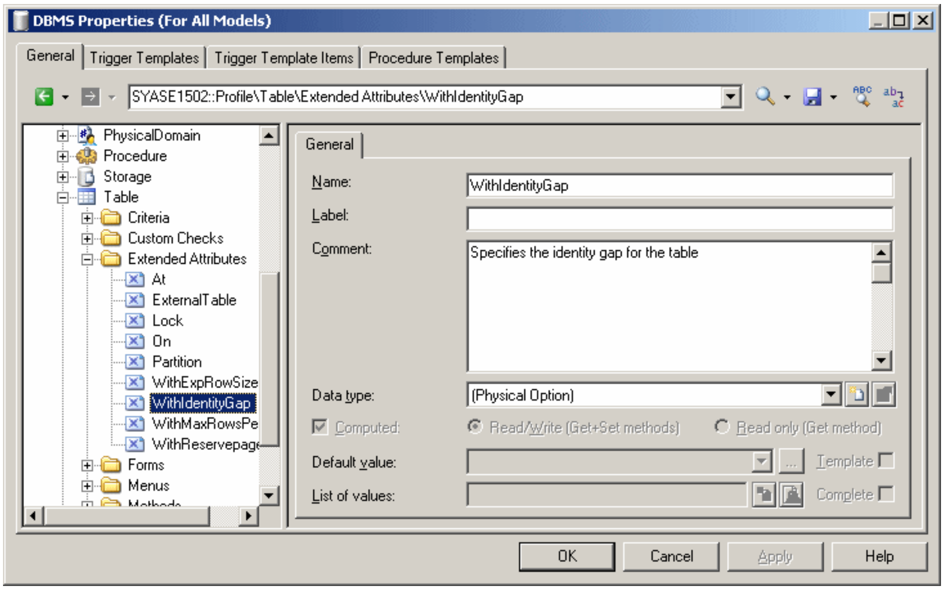
- *Control generation* for a given generation target. In this case, extended attributes are defined in the target language or DBMS of the model. For example, in the Java object language, several metaclasses have extended attributes used for generating Javadoc comments.
- *Further define model objects* in extensions. For example, in the extension 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 70). If all the extended attributes are allocated to forms, the generic page will not be displayed.

Creating an Extended Attribute

You can create an extended attribute for a metaclass, stereotype, or criterion.

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



SYASE1502::Profile\Table\Extended Attributes\WithIdentityGap

General

Name: WithIdentityGap

Label:

Comment: Specifies the identity gap for the table

Data type: (Physical Option)

[✓ Computed] [✓ Read/Write (Get+Set methods)] Read only (Get method)

Default value: Template

List of values: Complete

3. Click **Apply** to save your changes.

Extended Attribute Properties

You specify the properties for an extended attribute by selecting its entry in the resource editor.

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.

Property	Description
Data type	Specifies the form of the data to be held by the extended attribute. You can choose from: • Boolean • Color • Date or Time • File or Path • Integer, Float, or Hex • Font, Font Name, Font Size, or Font Style • Object - Specify the **Object type**, **Object stereotype** (if appropriate), and **Inverse collection name** in the properties below. For more information, see *Linking Objects Through Extended Attributes* on page 65. • String (single line) or Text (multi-line) To create your own data type, click the **Create Extended Attribute Type** tool to the right of the field (see *Creating an Extended Attribute Type* on page 63).

Computed	Specifies that the extended attribute is calculated from other values using VBScript on the **Get Method Script**, **Set Method Script**, and **Global Script** tabs. When you select this checkbox, you must choose between:
	• Read/Write (Get+Set methods)
	• Read only (Get method)
	In the following example script, the `FileGroup` computed extended attribute gets its value from and sets the value of the `filegroup` physical option of the object:
	`Function %Get%(obj)`
	`%Get% = obj.GetPhysicalOptionValue("on/<filegroup>")`
	`End Function`
	
	`Sub %Set%(obj, value)`
	`obj.SetPhysicalOptionValue "on/<filegroup>", value`
	`End Sub`

Default value	[if not `computed`] Specifies a default value for the attribute. You can specify the value in any of the following ways: • Enter the value directly in the list. • [predefined data types] Click the Ellipsis button to open a dialog listing possible values. For example, if the data type is set to Color, the Ellipsis button opens a palette window. • [user-defined data types] Select a value from the list.

Property	Description
Template	[if not `computed`] Specifies that the value of the attribute is to be evaluated as a GTL template at generation time. For example, if the value of the attribute is set to `%Code%`, it will be generated as the value of the code attribute of the relevant object. By default (when this checkbox is not selected), the attribute is evaluated literally, and a value of `%Code%` will be generated as the string `%Code%`.

List of values	Specifies a list of possible values for the attribute in one of the following ways:
	
	• Enter a static list of semi-colon-delimited values directly in the field.
	• Use the tools to the right of the list to create or select a GTL template to generate the list dynamically.
	If the attribute type is `Object`, and you do not want to filter the list of available objects in any way, you can leave this field blank.
	To perform a simple filter of the list of objects, use the `.collection` macro (see *.collection Macro* on page 289). In the following example, only tables with the `Generated` attribute set to true will be available for selection:
	`.collection(Model.Tables, %Generated%==true)`
	
	For more complex filtering, use the `foreach_item` macro (see *.foreach_item Macro* on page 293):
	`.foreach_item (Model.Tables)`
	`   .if %Generated%`
	`   .// (or more complex criteria)`
	`      %ObjectID%`
	`   .endif`
	`.next (\n)`
	
	If the attribute is based on an extended attribute type (see *Creating an Extended Attribute Type* on page 63), this field is unavailable since the values of the extended attribute type will be used.

Complete	Specifies that all possible values for the attribute are defined in the **List of values**, and that the user may not enter any other value.

Property	Description
Edit method	[if not `Complete`] Specifies a method to override the default action associated with the tool to the right of the field. This method is often used to apply a filter defined in the **List of values** field in the object picker. In the following example, only tables with the `Generated` attribute set to true will be available for selection:

```
Sub %Method%(obj)

   Dim Mdl
   Set Mdl = obj.Model

   Dim Sel
   Set Sel = Mdl.CreateSelection

   If not (Sel is nothing) Then
      Dim table
      For Each table in Mdl.Tables
          if table.generated then
              Sel.Objects.Add table
          end if
      Next

      ' Display the object picker on the selection
      Dim selObj
      set selObj = Sel.ShowObjectPicker
      If Not (selObj is Nothing) Then
         obj.SetExtendedAttribute "Storage-For-Each",
selObj
       End If

      Sel.Delete
   End If

End Sub
```

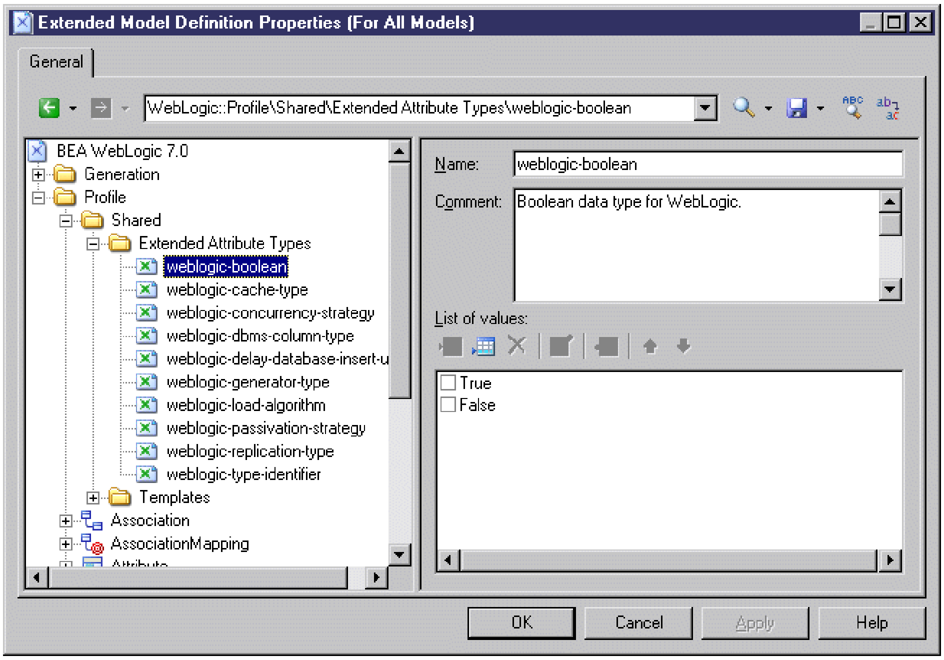
Property	Description
Text format	[for `Text` data types only] Specifies the language contained within the text attribute. If you select any value other than plain `Text`, then an editor toolbar and (where appropriate) syntax coloring are provided in the associated form fields.
Object type	[for `Object` data types only] Specifies the type of the object that the attribute contains (for example, User, Table, Class).
Object stereotype	[for `Object` data types only] Specifies the stereotype that objects of this type must bear to be selectable.

Property	Description
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. 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 76.

Creating an Extended Attribute Type

You can create extended attribute types 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.

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


Extended Model Definition Properties (For All Models)
General
WebLogic::Profile\Shared\Extended Attribute Types\weblogic-boolean
BEA WebLogic 7.0
Generation
Profile
Shared
Extended Attribute Types
weblogic-boolean
weblogic-cache-type
weblogic-concurrency-strategy
weblogic-dbms-column-type
weblogic-delay-database-insert-u
weblogic-generator-type
weblogic-load-algorithm
weblogic-passivation-strategy
weblogic-replication-type
weblogic-type-identifier
Templates
Association
AssociationMapping
List of values:
OK
Cancel
Apply
Help

Name: weblogic-boolean

Comment: Boolean data type for WebLogic.

List of values:
- True
- False

3. Click **Apply** to save your changes.

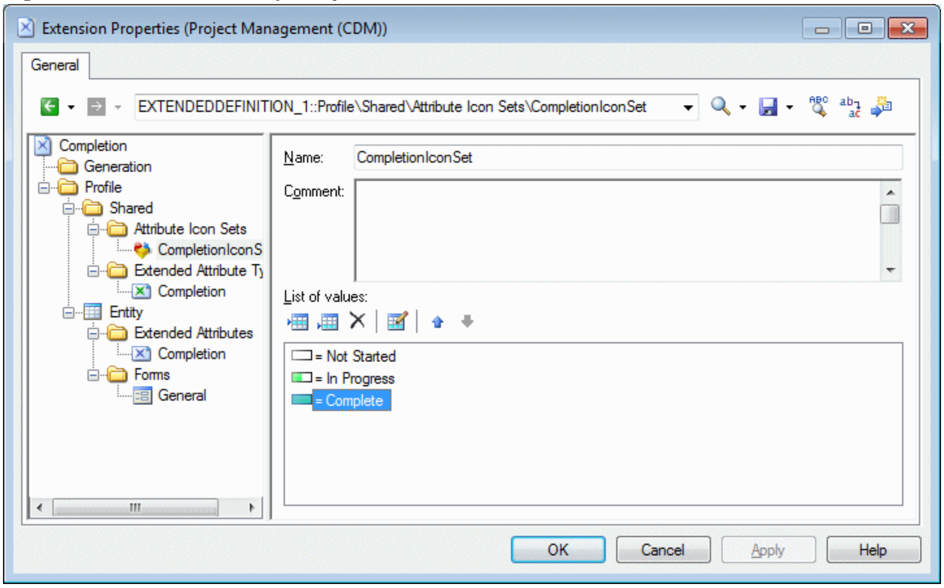
The new shared type is available to any extended attribute in the **Data Type** field. You can also define a list of values for a given extended attribute directly in this field (see *Extended Attribute Properties* on page 59).

Specifying Icons for Attribute Values

You can specify icons to display on object symbols in place of extended attribute values by creating an attribute icon set with individual attribute value icons for each possible value.

1. Create an extended attribute and select a standard data type or an extended attribute type (see *Creating an Extended Attribute Type* on page 63).
2. If appropriate, specify a list of possible values and a default value.
3. Click the **Create** tool to the right of the **Icon set** list to create a new icon set

   A new icon set is created at **Profile** > **Shared** > **Attribute Icon Sets** initialized with the possible values and an empty icon which matches any value for which another icon has not been defined (`=*`).
4. For each value in the list, double-click it, and click the **Icon** tool to select an icon to represent this value on object symbols:



EXTENDEDDEFINITION_1::Profile\Shared\Attribute Icon Sets\CompletionIconSet

Name: CompletionIconSet

Comment:

List of values:

= Not Started
= In Progress
= Complete

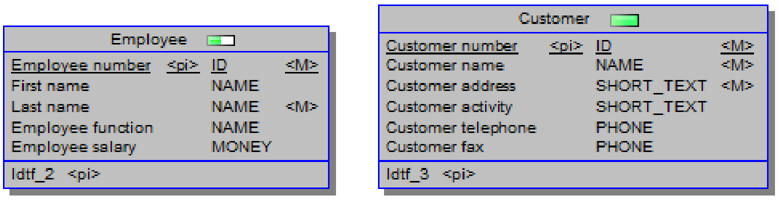
Note: By default, the **Filter operator** field is set to `=`, and each icon matches exactly one possible value. To have a single icon match multiple values, use the `Between` or another operator together with a suitable **Filter value**. For example, in an icon set paired with a `progress` attribute for which the user can enter any value between 0 and 100% progress, you could use three icons:

- Not Started - `= 0`

- In Progress - `Between 1,99`
- Completed - `= 100`

5. If appropriate, add the attribute to a form (see *Forms (Profile)* on page 70), to enable users to modify its value.
6. Click **OK** to save your changes and return to the model.
7. To enable the display of the icon on your object symbol, select **Tools** > **Display Preferences**, select your object type, and click the **Advanced** button to add your attribute to the symbol. For detailed information about working with display preferences, see *Core Features Guide > The PowerDesigner Interface > Diagrams, Matrices, and Symbols > Display Preferences*.

   Your attribute is now displayed on object symbols. In the following example, the `Employee` and `Team` entities are `In Progress`, while the `Customer` entity is `Completed`:



Employee			
Employee number	<pi>	ID	<M>
First name		NAME	
Last name		NAME	<M>
Employee function		NAME	
Employee salary		MONEY	
Idtf_2 <pi>			

Customer			
Customer number	<pi>	ID	<M>
Customer name		NAME	<M>
Customer address		SHORT_TEXT	<M>
Customer activity		SHORT_TEXT	
Customer telephone		PHONE	
Customer fax		PHONE	
Idtf_3 <pi>			

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".

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.

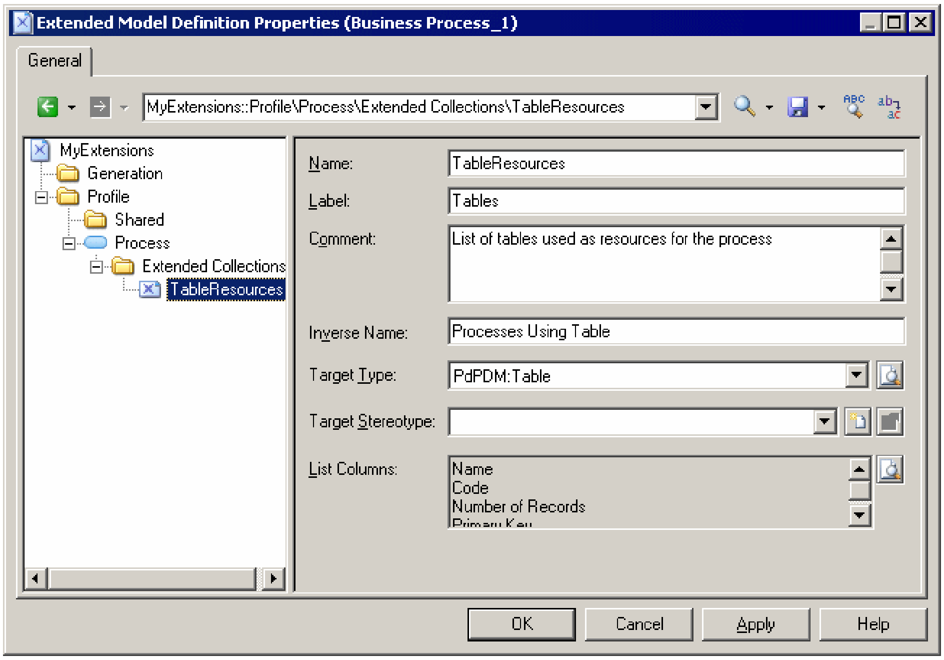
Note: If you create an extended collection or composition on a stereotype or criterion, the corresponding tab is displayed only if the metaclass instance bears the stereotype or meets the criterion.

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

Creating Extended Collections and Compositions

You can create an extended collection for a metaclass, stereotype, or criterion.

1. Right-click a metaclass, stereotype, or criterion and select **New > Extended Collection** or **Extended Composition**.
2. Enter a scripting **Name** and display **Label** to use as the name of the tab associated with the collection in the parent 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 **Target 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. [optional] Click the **Customize Default Columns** tool to modify the columns that will appear by default when the user opens the property sheet tab associated with the collection.


Extended Model Definition Properties (Business Process_1)
General
MyExtensions::Profile\Process\Extended Collections\TableResources
MyExtensions
Generation
Profile
Shared
Process
Extended Collections
TableResources
OK
Cancel
Apply
Help

Name: TableResources
Label: Tables
Comment: List of tables used as resources for the process
Inverse Name: Processes Using Table
Target Type: PdPDM:Table
Target Stereotype:
List Columns: Name
Code
Number of Records
Primary Key

7. Click **Apply** to save your changes.

   You can view the tab associated with the collection by opening the property sheet of a metaclass instance. The tab contains an **Add Objects** (and, if the metaclass belongs to the same type of model, **Create an Object**) tool, to populate the collection.

Extended Collection/Composition Properties

You specify the properties for an extended collection or composition by selecting its entry in the resource editor.

Property	Description
Name	Specifies the name of the extended collection.
Label	Specifies the display name of the collection, which will appear in the PowerDesigner interface.
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. For extended compositions, only the ExtendedSubObject is available, and you must specify a stereotype for it.
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.
List Columns	Specifies the property columns that will be displayed by default in the parent object property sheet tab associated with the collection. Click the Customize Default Columns tool to the right of this field to add or remove columns.

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 65).

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 *Core Features Guide > The PowerDesigner Interface > Reports > The Report Editor > Adding Items to a Report > Modifying the Collection of an Item*).
- Improve GTL generation since you can loop on user-defined calculated collections.

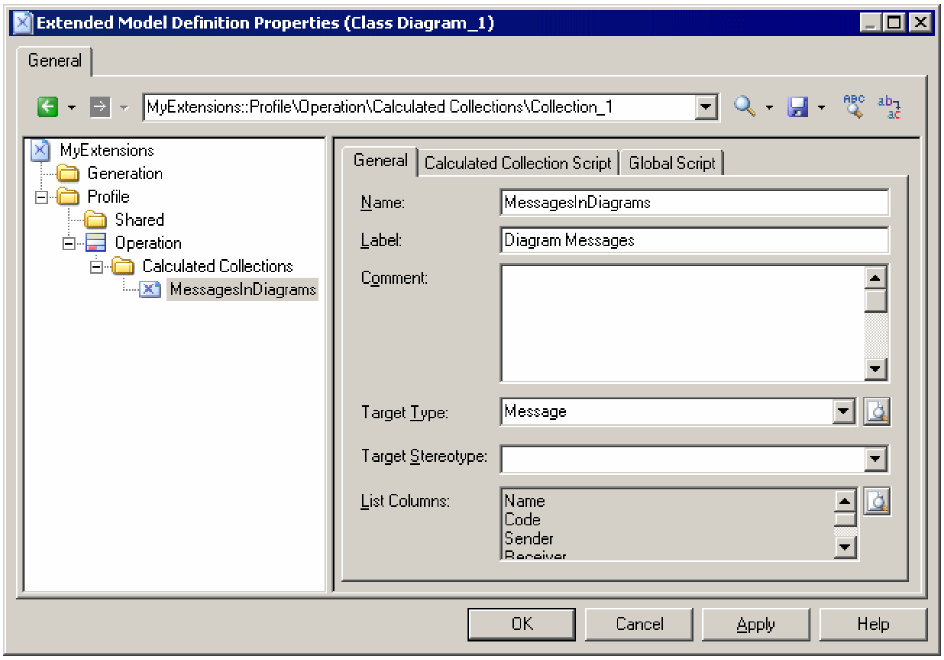
For example, in an OOM, you may need to create a list of sequence diagrams using an operation, and can create a calculated collection on the operation metaclass that retrieves this information.

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.

Creating a Calculated Collection

You can create a calculated collection for a metaclass, stereotype, or criterion.

1. Right-click a metaclass, stereotype, or criterion and select **New** > **Calculated Collection**.
2. Enter a scripting **Name** and display **Label** to use as the name of the tab associated with the collection in the parent object property sheet.
3. [optional] Enter a **Comment** to describe the collection.
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 **Target 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.



Extended Model Definition Properties (Class Diagram_1)
General
MyExtensions::Profile\Operation\Calculated Collections\Collection_1
MyExtensions
Generation
Profile
Shared
Operation
Calculated Collections
MessagesInDiagrams
General
Calculated Collection Script
Global Script
OK
Cancel
Apply
Help

Name: MessagesInDiagrams
Label: Diagram Messages
Comment:
Target Type: Message
Target Stereotype:
List Columns: Name, Code, Sender, Receiver

7. Click **Apply** to save your changes.

   You can view the tab associated with the collection by opening the property sheet of a metaclass instance.

Calculated Collection Properties

You specify the properties for an extended collection by selecting its entry in the resource editor.

Property	Description
Name	Specifies the name of the calculated collection.
Label	Specifies the display name of the collection, which will appear in the PowerDesigner interface.
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. 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 88 and *Using the global script* on page 91.

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.

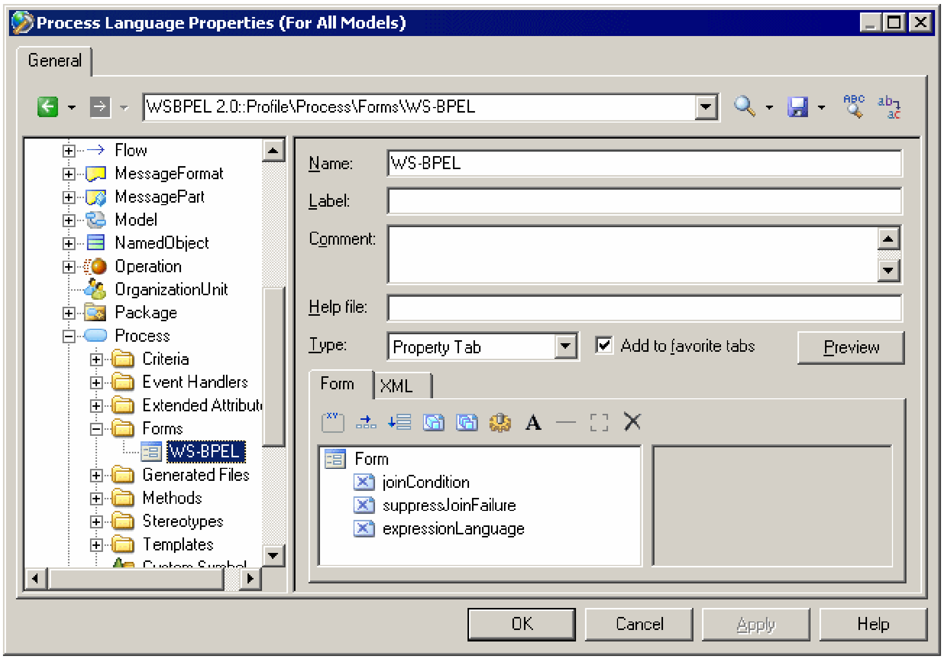
By default, extended attributes are listed alphabetically on the Extended Attributes tab of the object's property sheet. By creating your own form, you can make these attributes more visible and easy to use, by organizing them logically, grouping related ones, and emphasizing those that are most important. 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.

Creating a Form

You can create a form in a profile to create a new property tab or dialog box in the PowerDesigner interface, or to replace a standard property sheet tab.

1. Right-click a metaclass, stereotype or criterion and select **New > Form** to create an empty form.



Name: WS-BPEL
Label:
Comment:
Help file:
Type: Property Tab
☑ Add to favorite tabs

Form
joinCondition
suppressJoinFailure
expressionLanguage

2. Enter a scripting **Name** and display **Label** for the form, select a **Type** and enter any other appropriate properties (see *Form Properties* on page 72). 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. Insert controls as necessary in your form using the toolbar on the **Form** tab (see *Adding Extended Attributes and Other Controls to Your Form* on page 72).
4. Click the **Preview** button to review the layout of your form and, when satisfied, click **Apply** to save your changes.

Form Properties

You specify the properties for a form by selecting its entry in the resource editor.

Property	Description
Name	Specifies the internal name of the form, which can be used for scripting.
Label	Specifies the display name of the form, which will appear in the PowerDesigner interface.
Comment	Provides additional information about the form.

Help file	Enables the display of a Help button and specifies an action that will be performed when the button is clicked or F1 is pressed when in the context of the form.
	The action can be the display of a help file (.hlp, .chm or .html), and can specify a specific topic. For instance:
	`C:\PD1500\pddoc15.chm 26204`
	If no help file extension is found, the string will be treated as a shell command to execute. For instance, you could instruct PowerDesigner to open a simple text file:
	`notepad.exe C:\Temp\Readme.txt`

Type	Specifies the kind of form. You can choose from the following: • Dialog Box – creates a dialog box that can be launched from a menu or via a form button • 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.
Add to favorite tabs	[property tabs only] Specifies that the tab is displayed by default in the object property sheet.

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.

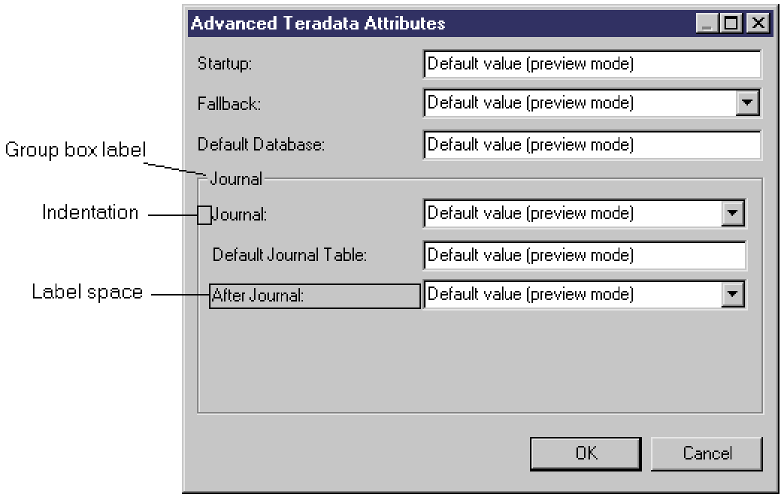
Tool	Description
	Add Group Box - inserts a group box, intended to contain other controls within a named box.
	Add Tab Window - inserts a sub-tab layout, in which each child control appears, by default, in its own sub-tab. To place multiple controls on a single sub-tab, use a horizontal or vertical layout.
	Add Horizontal Layout - inserts a horizontal layout. To arrange controls to display side by side, drag them onto the horizontal layout in the list.
	Add Vertical Layout - inserts a vertical layout. To arrange attributes to display one under the other, drag them onto the vertical layout in the list. Vertical layouts are often used in conjunction with a horizontal layout, to provide columns of controls.
	Include Another Form - inserts a form defined on this or another metaclass in the present form (see *Example: Including a Form in a Form* on page 80).
	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. 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. 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. Collections are displayed as standard grids with all the appropriate tools. 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. 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.

Tool	Description
	Add List Box [dialog boxes only] - inserts a list box.
✓	**Add Check Box** [dialog boxes only] - inserts a check box.
A	**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.
	Add Spacer – inserts an area of blank space.
✕	**Delete** – deletes the currently selected control.

Form Control Properties

When you add controls to your form, you can specify properties to control their format and contents.

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: Opening a Dialog Box from a Menu* on page 102).
Label	Specifies a label for 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. You can insert line breaks with `\n`. To create keyboard shortcuts to navigate among controls, prefix the letter that will serve as the shortcut with an ampersand. If you do not specify a shortcut key, PowerDesigner will choose one by default. To display an ampersand in a label, you must escape it with a second ampersand (for example: `&Johnson && Son` will display as **Johnson & Son**.
Attribute	[included forms] Specifies the object on which the form to be included is defined. The list is populated with all attributes of type `object` and the following objects: • <None> - the present metaclass • Generation Origin - for example, the CDM entity from which a PDM table was generated • Model - the parent model • Parent - the immediate parent object for sub-objects (for example, the table containing a column • Parent Folder - the immediate parent object for composite objects (for example BPM processes that contain other processes) • Parent Package - the immediate parent package

Property	Definition
Form name	[included forms] Specifies the name of the form that will be included. You can: • Select a standard property sheet tab name from the list. • Enter the name of a custom form defined in the extension file. • Enter the name of a GTL template to generate XML to define the form.
Indentation	[container controls] 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] Specifies the space in pixels reserved for displaying the labels of child controls between the indentation of the container and the control fields. To align controls with the controls in a previous container, enter a negative value. For example, if you have two group boxes, and want all controls in both to be aligned identically, set an appropriate indentation in the first group box and set the indentation of the second group box to `-1`. If a child control label is larger than the specified value, the label space property is ignored; to display this label, you need to type a number of pixels greater than 50.  Group box label Indentation Label space Advanced Teradata Attributes OK Cancel Startup: Default value (preview mode) Fallback: Default value (preview mode) Default Database: Default value (preview mode) Journal Journal: Default value (preview mode) Default Journal Table: Default value (preview mode) After Journal: Default value (preview mode)
Show control as label	[group boxes] Use the first control contained within the group box as its label.
Show Hidden Attribute	[extended attributes] 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] Specifies a default value for the control. For extended attributes, default values must be specified in the attribute's properties (see *Extended Attribute Properties* on page 59).

Property	Definition
List of Values	[combo and list boxes] Specifies a list of possible values for the control. For extended attributes, lists of values must be specified in the attribute's properties (see *Extended Attribute Properties* on page 59).
Exclusive	[combo boxes] 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.
Read-Only	[included forms and dialog box entry fields] Specifies that the control is read-only, and will be greyed in the form.
Left Text	[booleans] Places the label text to the left of the checkbox.
Display	[booleans and methods] Specifies the form in which the boolean options or method button are displayed. For booleans, you can choose between: • Check box • Vertical radio buttons • Horizontal radio buttons For methods, you can choose from a range of standard icons or **Text**, which prints the text specified in the **Label** field on the button.
Width/ Height	[spacers] Specify the width and height, in pixels, of the spacer.

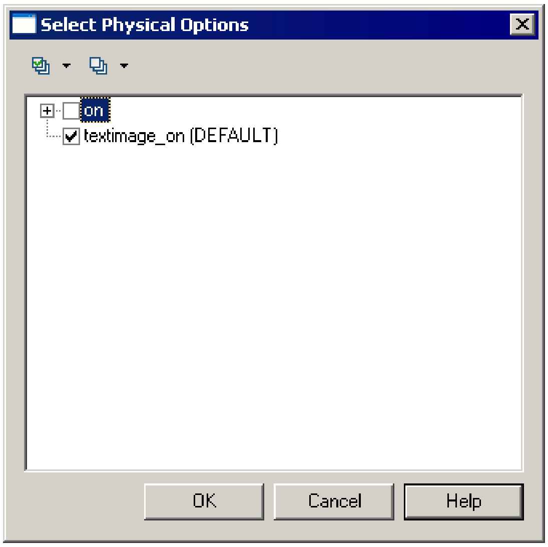
Adding DBMS Physical Options to Your Forms

Many DBMSs use *physical options* as part of the definition of their objects. The most commonly-used physical options are displayed on a form, **Physical Options (Common)**, defined under the appropriate metaclass. You can edit this form, or add physical options to your own forms.

Note: PowerDesigner displays all of the available options for an object (defined at `Script/Objects/`*`object`*`/Options` category) on the **Physical Options** tab (see *Physical Options* on page 218).

For a physical option to be displayed in a form, it must be associated with an extended attribute with the type `physical option`.

1. Right-click the metaclass and select **New Extended Attribute from Physical Options** to open the Select Physical Options dialog:



Note: This dialog will be empty if no physical options are defined at `Script/Objects/`*`object`*`/Options`.

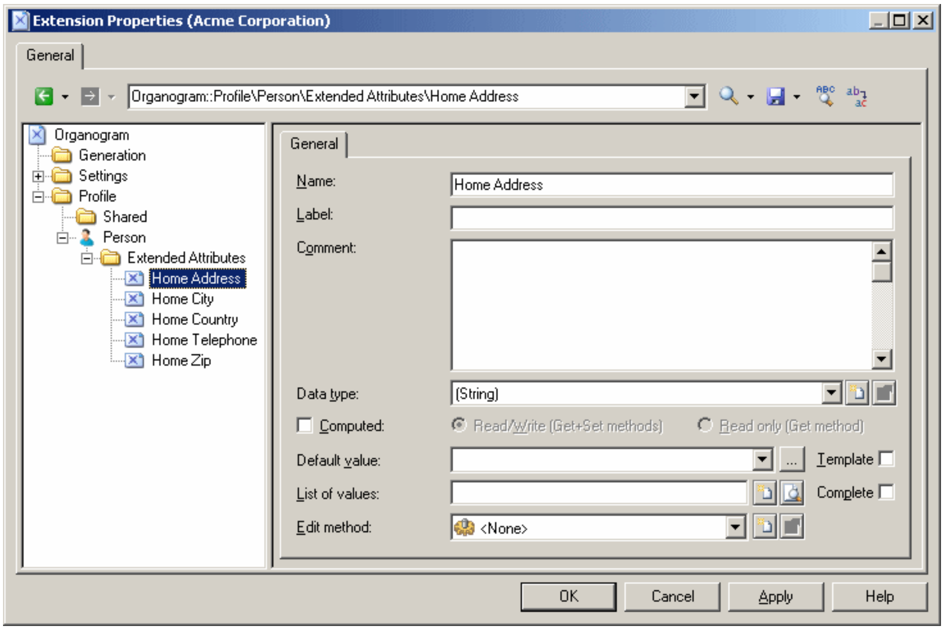
2. Select the physical option required and click **OK** to create an extended attribute associated with it.
3. Specify any other appropriate properties.
4. Select the form in which you want to insert the physical option and click the Add Attribute tool to insert it as a control (see *Adding Extended Attributes and Other Controls to Your Form* on page 72).

Note: To change the physical option associated with an extended attribute, click the ellipsis to the right of the **Physical Options** field in the Extended Attribute property sheet.

Example: Creating a Property Sheet Tab

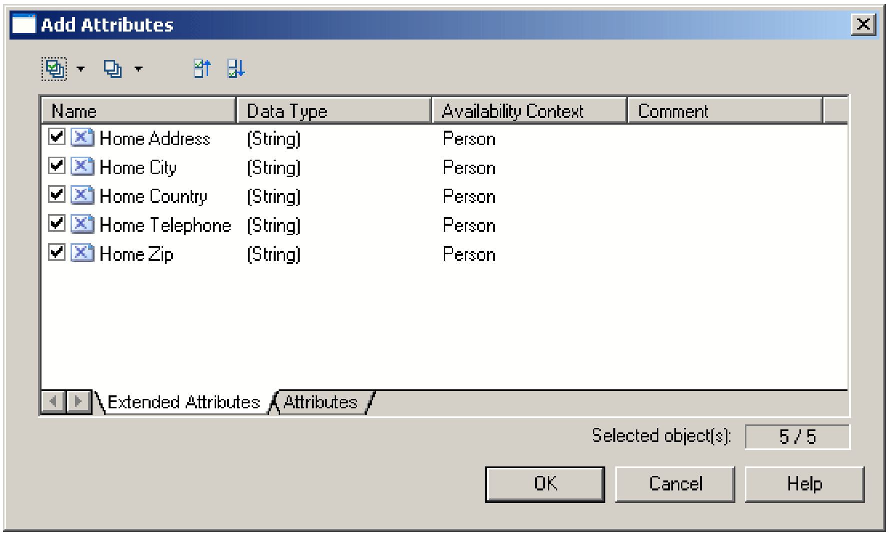
In this example, we will create a new property tab for the EAM Person metaclass to display extended attributes we define to store personal information.

1. Create a new extension file (see *Creating an Extension File* on page 23) in an EAM, add the `Person` metaclass (see *Adding a Metaclass to a Profile* on page 47), and define five extended attributes (see *Creating an Extended Attribute* on page 59) to contain home contact details:



Name: Home Address
Label:
Comment:
Data type: (String)
Default value:
List of values:
Edit method: <None>

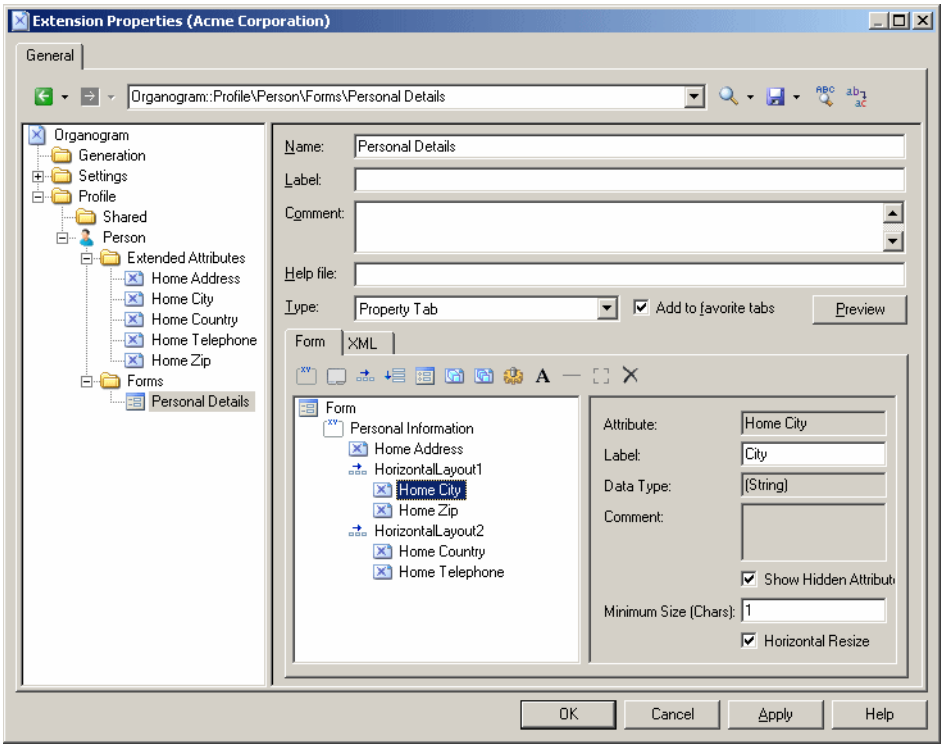
2. Right-click the `Person` metaclass and select **New > Form**, enter `Personal Details` in the **Name** field, select `Property Tab` in the **Type** list, and click the **Add Attribute** tool to select all the new extended attributes for inclusion in the form:



Name	Data Type	Availability Context	Comment
✓ Home Address	(String)	Person	
✓ Home City	(String)	Person	
✓ Home Country	(String)	Person	
✓ Home Telephone	(String)	Person	
✓ Home Zip	(String)	Person	

Selected object(s): 5 / 5

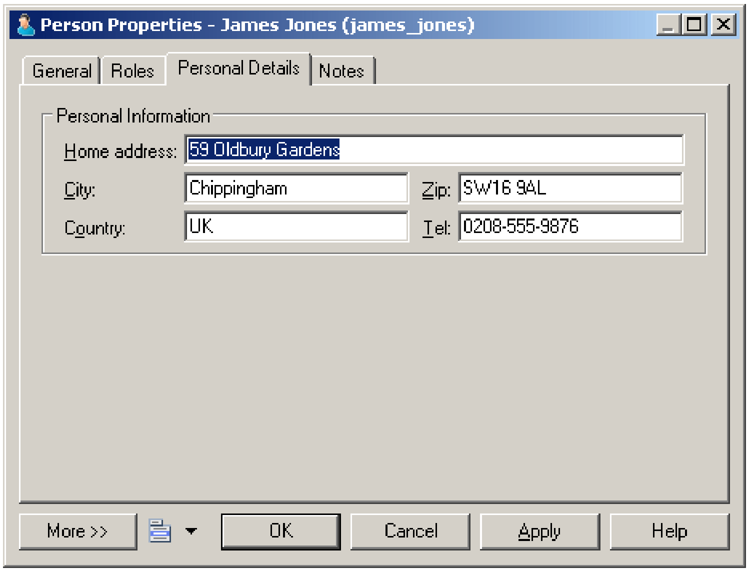
3. Click **OK** to add the attributes to the form, and arrange them in a group box, using horizontal layouts to align them neatly. Here, I'm using the **Label** field to overide the default name of the attribute in the form for brevity:



Name: Personal Details
Label:
Comment:
Help file:
Type: Property Tab
☑ Add to favorite tabs

Attribute: Home City
Label: City
Data Type: (String)
Comment:
☑ Show Hidden Attribut
Minimum Size (Chars): 1
☑ Horizontal Resize

4. Click **OK** to save your changes and return to the model. When you next open the property sheet of a person, a new **Personal Details** tab is available containing the extended attributes:


Person Properties - James Jones (james_jones)

Personal Information

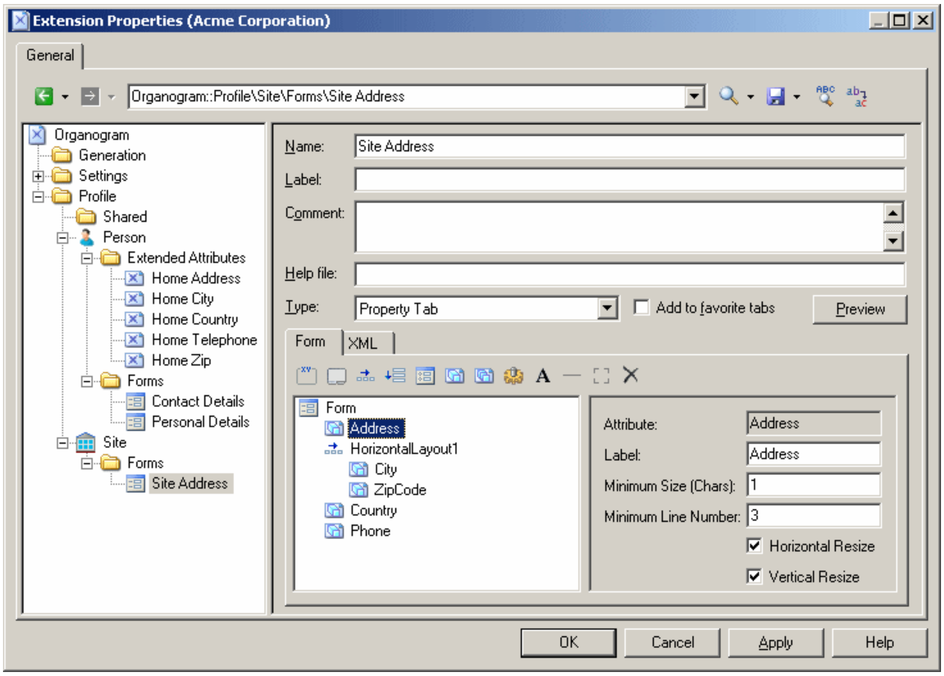
Home address:	59 Oldbury Gardens		
City:	Chippingham	Zip:	SW16 9AL
Country:	UK	Tel:	0208-555-9876

Example: Including a Form in a Form

In this example, we will replace the General tab of the EAM Person metaclass by a form which includes properties from the person and from the site to which she is assigned by including a form defined on the Site metaclass as a read-only control in a form defined on the Person metaclass.

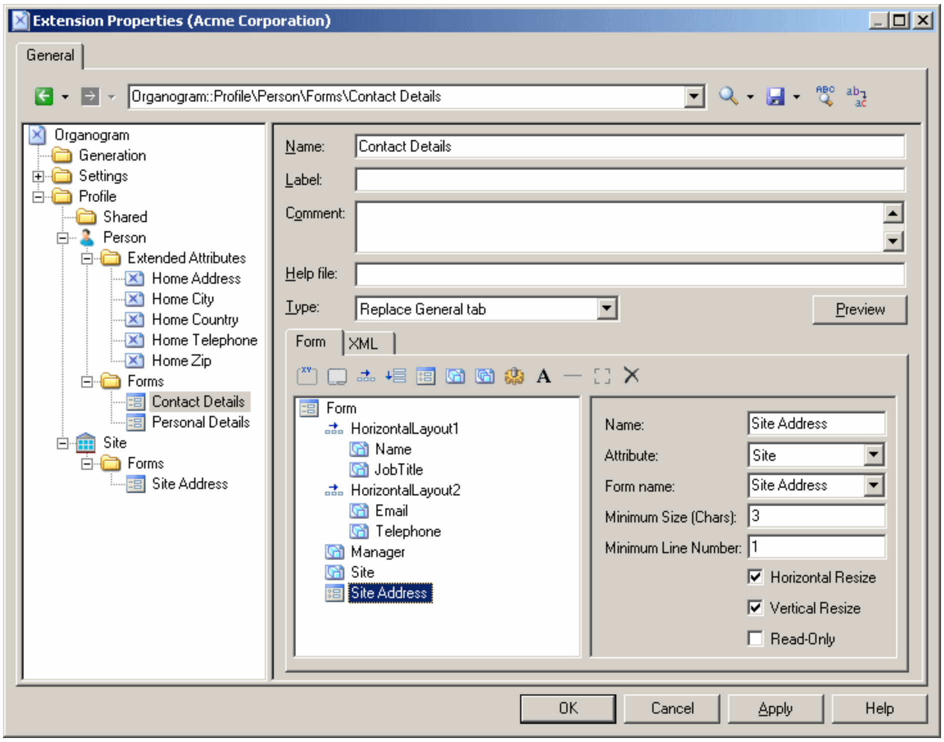
This example builds on the extension file created in *Example: Creating a Property Sheet Tab* on page 77.

1. Add the `Site` metaclass and create a form called `Site Address`. Select `Property Tab` from the **Type** list and unselect the **Add to favorite tabs** option (as we do not want this form, which duplicates standard site properties displayed in site property sheets).
2. Populate the form with standard attributes to display the complete address of the site:



Name: Site Address
Label:
Comment:
Help file:
Type: Property Tab
Add to favorite tabs
Attribute: Address
Label: Address
Minimum Size (Chars): 1
Minimum Line Number: 3
☑ Horizontal Resize
☑ Vertical Resize

3. Create a form under the `Person` metaclass, select `Replace General tab` from the **Type** list, and change the name to `Contact Details`.
4. Delete unwanted attributes from the list, and arrange the remaining attributes you want to display, including the `Site` attribute (which is of type `Object`, and which will enable us to pull in the appropriate properties from the associated site form) using horizontal and vertical layouts.
5. Click the **Include Another Form** tool, select `Site` in the **Attribute** field, and enter `Site Address` in the **Form name** field. Select the **Read-Only** check box to prevent editing of the included form from the person's property sheet:


Extension Properties (Acme Corporation)

General

Organogram::Profile\Person\Forms\Contact Details

- Organogram
  - Generation
  - Settings
  - Profile
    - Shared
    - Person
      - Extended Attributes
        - Home Address
        - Home City
        - Home Country
        - Home Telephone
        - Home Zip
      - Forms
        - Contact Details
        - Personal Details
    - Site
      - Forms
        - Site Address

Name: Contact Details

Label:

Comment:

Help file:

Type: Replace General tab

Preview

Form | XML

- Form
  - HorizontalLayout1
    - Name
    - JobTitle
  - HorizontalLayout2
    - Email
    - Telephone
  - Manager
  - Site
  - Site Address

Name: Site Address

Attribute: Site

Form name: Site Address

Minimum Size (Chars): 3

Minimum Line Number: 1

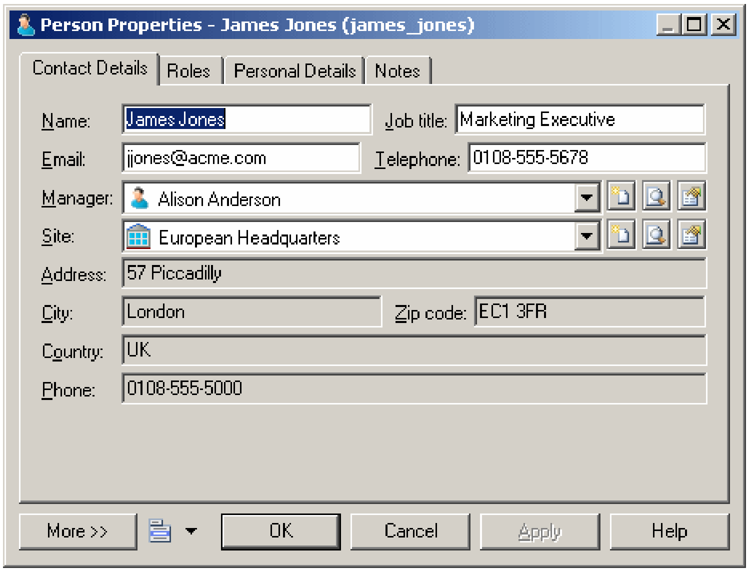
[✓] Horizontal Resize

[✓] Vertical Resize

[ ] Read-Only

OK | Cancel | Apply | Help

6. Click **OK** to save the extensions, and return to your model. When you next open the property sheet of a person, the **General** tab is replaced by the custom **Contact Details** tab, and when the person is assigned to a site, the site's address details are displayed as read-only in the lower part of the form:


Person Properties - James Jones (james_jones)

Contact Details | Roles | Personal Details | Notes

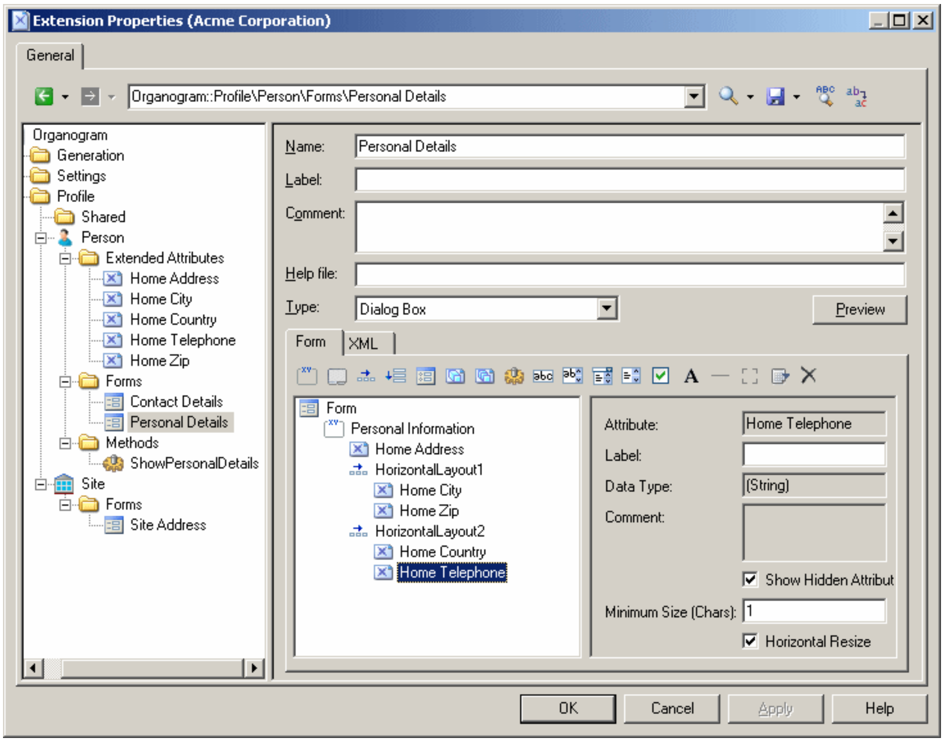
Name:	James Jones	Job title:	Marketing Executive
Email:	jjones@acme.com	Telephone:	0108-555-5678
Manager:	Alison Anderson		
Site:	European Headquarters		
Address:	57 Piccadilly		
City:	London	Zip code:	EC1 3FR
Country:	UK		
Phone:	0108-555-5000		

Example: Opening a Dialog from a Property Sheet

In this example, we will add a button to a property sheet tab, to open a dialog box, allowing you to enter additional personal details for a person.

This example builds on the extension file developed in *Example: Including a Form in a Form* on page 80.

1. Open the `Personal Details` form under the `Person` metaclass, and select `Dialog Box` in the **Type** field, to transform it from a property sheet tab into an independent dialog:


Extension Properties (Acme Corporation)

General

Organogram::Profile\Person\Forms\Personal Details

Organogram
- Generation
- Settings
- Profile
  - Shared
  - Person
    - Extended Attributes
      - Home Address
      - Home City
      - Home Country
      - Home Telephone
      - Home Zip
    - Forms
      - Contact Details
      - Personal Details
    - Methods
      - ShowPersonalDetails
  - Site
    - Forms
      - Site Address

Name: Personal Details
Label:
Comment:
Help file:
Type: Dialog Box

Form | XML

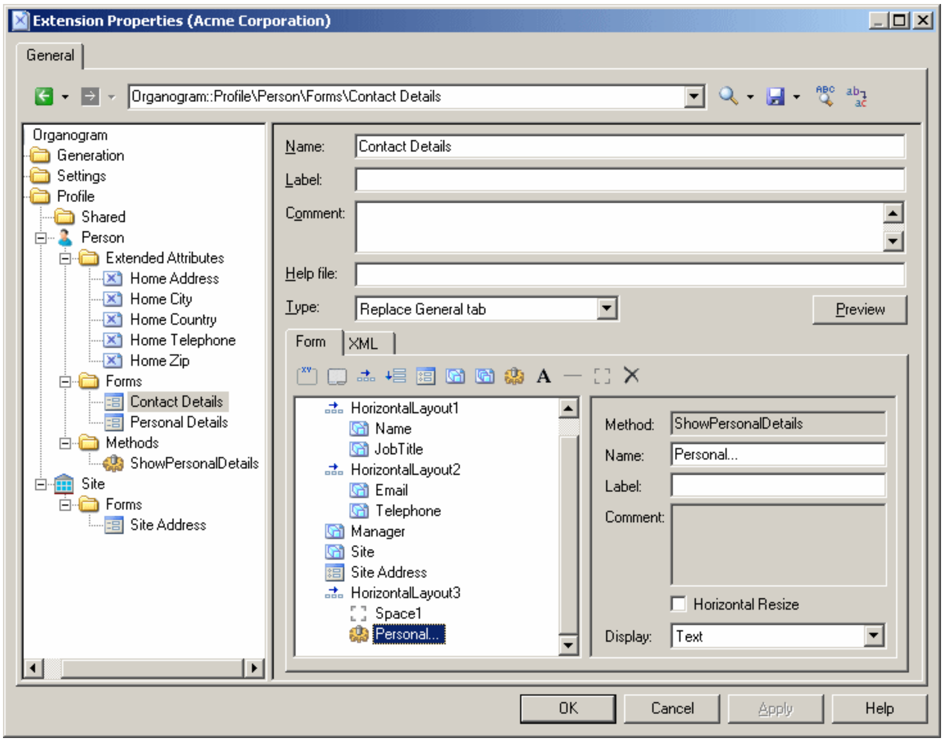
- Form
  - Personal Information
    - Home Address
    - HorizontalLayout1
      - Home City
      - Home Zip
    - HorizontalLayout2
      - Home Country
      - Home Telephone

Attribute: Home Telephone
Label:
Data Type: (String)
Comment:
Show Hidden Attribut: ✓
Minimum Size (Chars): 1
Horizontal Resize: ✓

2. Right-click the Person metaclass and select **New** > **Method**. Enter the name `ShowPersonalDetails`, 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%.Personal
Details")
 If not dlg is Nothing Then
  dlg.ShowDialog()
 End If
End Sub
```

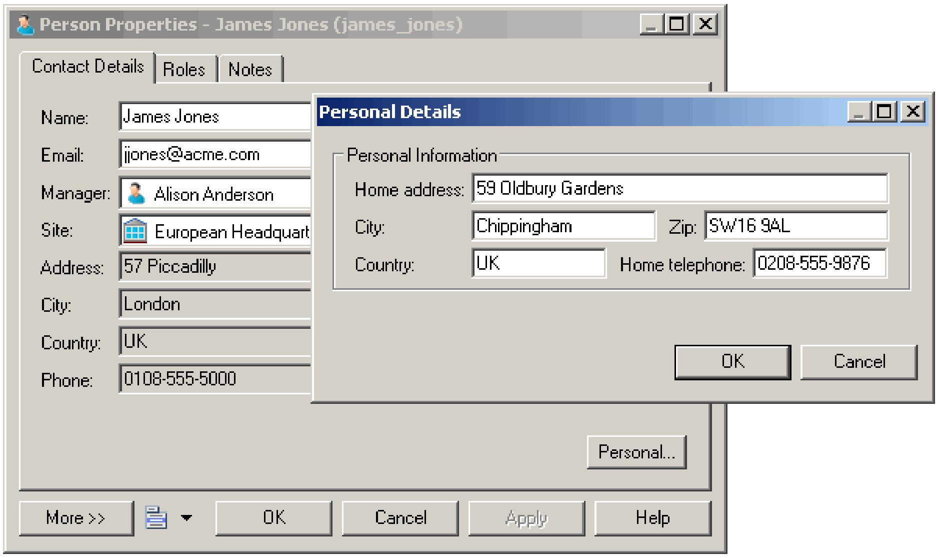
3. Select the `Contact Details` form, and click the **Add Method Push Button** tool, select the `ShowPersonalDetails` method, and then click **OK** to add it to the form. Here, I use a horizontal layout and spacer to align the button with the right edge of the form:



Name: Contact Details
Label:
Comment:
Help file:
Type: Replace General tab

Method: ShowPersonalDetails
Name: Personal...
Label:
Comment:
Display: Text

4. Enter `Personal...` in the **Label** field, and then click **OK** to save your changes and return to the model. Now when you open the property sheet of a person, the **Contact Details** tab contains a **Personal...** button which opens the **Personal Information** dialog:



Name: James Jones
Email: jjones@acme.com
Manager: Alison Anderson
Site: European Headquart
Address: 57 Piccadilly
City: London
Country: UK
Phone: 0108-555-5000

Home address: 59 Oldbury Gardens
City: Chippingham Zip: SW16 9AL
Country: UK Home telephone: 0208-555-9876

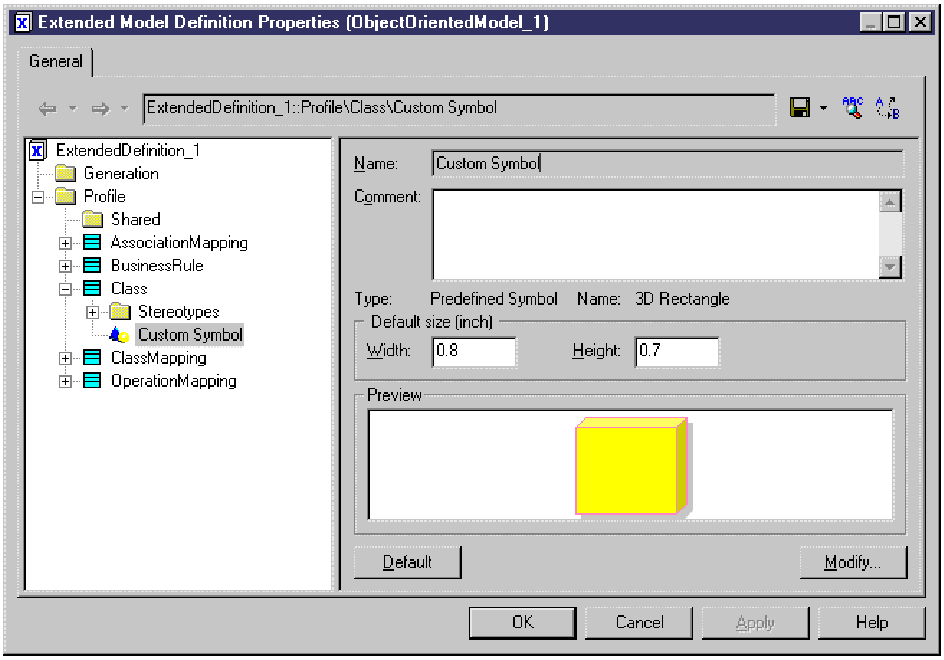
Custom Symbols (Profile)

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.



Name: Custom Symbol
Comment:
Type: Predefined Symbol Name: 3D Rectangle
Default size (inch)
Width: 0.8 Height: 0.7
Preview

2. Specify a default **Width** and **Height** for the symbol and then 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 (and on the additional custom symbol options that let you control the default format options for the symbol, and whether users can edit them, on a per-tab basis) see *Core Features Guide > The PowerDesigner Interface > Diagrams, Matrices, and Symbols > Symbols > Symbol Format Properties*.

3. Click **OK** to return to the resource editor, where you can view your changes in the Preview field.
4. 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 *Chapter 7, Scripting PowerDesigner* on page 323.

Custom Check Properties

You specify the properties for a custom check by selecting its entry in the resource editor.

Parameter	Description
Name	Name of the custom check. This name is displayed under the selected object category in the **Check Model Parameters** dialog. 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.
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.
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 88.
Autofix Script	This tab contains the autofix script. See *Defining the script of an autofix* on page 89
Global Script	This tab is used for sharing library functions and static attributes in the resource file. See *Using the global script* on page 91.

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.

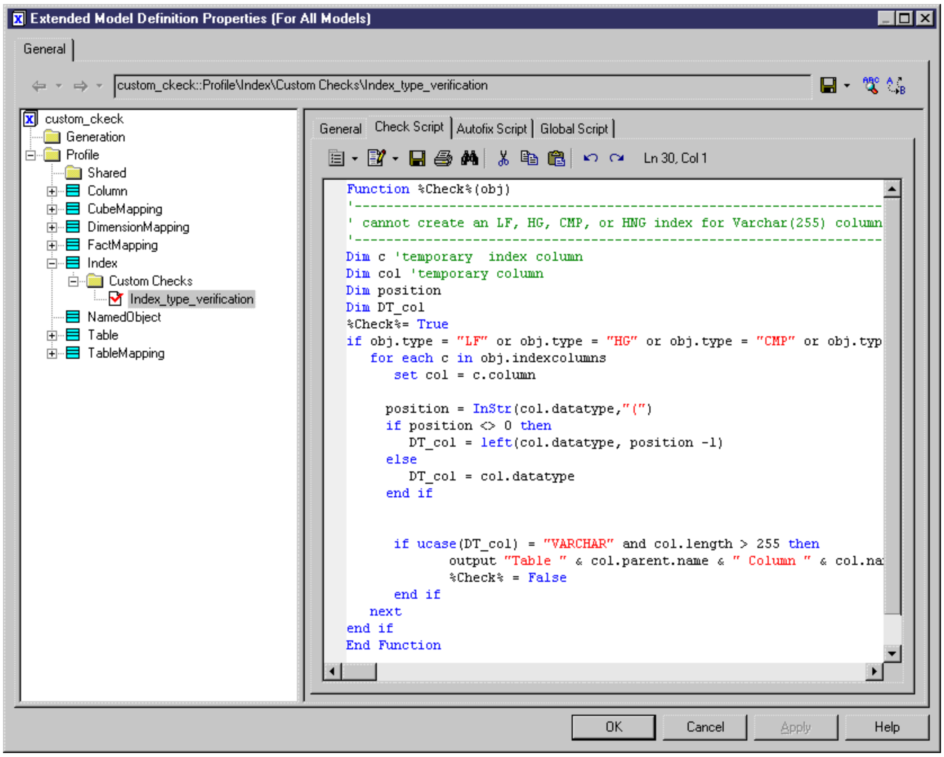
1. Right-click a metaclass, stereotype or a criterion under Profile, and select **New > Custom Check**.
2. Click the Check Script tab in the custom check properties to display the script editor.

   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
```



```
Function %Check%(obj)
'---------------------------------------------------------------------------
' cannot create an LF, HG, CMP, or HNG index for Varchar(255) column
'---------------------------------------------------------------------------
Dim c 'temporary  index column
Dim col 'temporary column
Dim position
Dim DT_col
%Check%= True
if obj.type = "LF" or obj.type = "HG" or obj.type = "CMP" or obj.typ
   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.na
             %Check% = False
      end if
   next
end if
End Function
```

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.

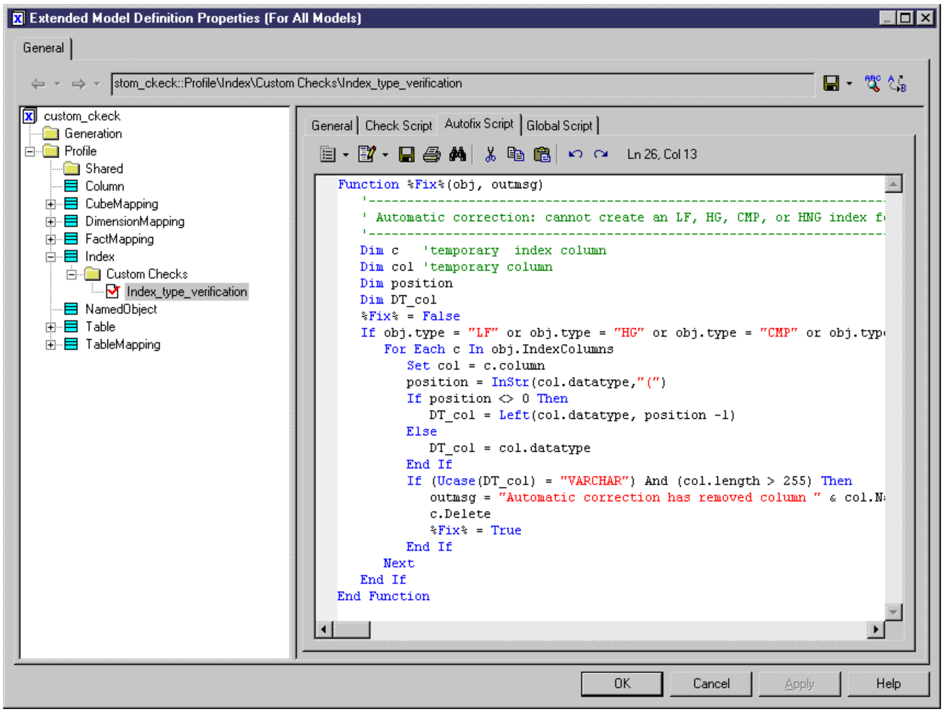
1. Click the Autofix Script tab in the custom check properties.

   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
```



stom_ckeck::Profile\Index\Custom Checks\Index_type_verification

```
Function %Fix%(obj, outmsg)
   '-----------------------------------------------------------------
   ' Automatic correction: cannot create an LF, HG, CMP, or HNG index f
   '-----------------------------------------------------------------
   Dim c   'temporary  index column
   Dim col 'temporary column
   Dim position
   Dim DT_col
   %Fix% = False
   If obj.type = "LF" or obj.type = "HG" or obj.type = "CMP" or obj.typ
      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.N
            c.Delete
            %Fix% = True
         End If
      Next
   End If
End Function
```

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 event handlers can be defined on items in the Profile category:

Event handler	Description

CanCreate	[all objects] 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:
	`Function %CanCreate%(parent)`
	`  if parent is Nothing or`
	`  parent.IsKindOf(PdBpm.Cls_Process) then`
	`  %CanCreate% = False`
	`  else`
	`  %CanCreate% = True`
	`  end if`
	`End Function`
	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 Toolbox, from the Browser or in a list.
	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.

Event handler	Description

Initialize	[all objects] 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.
	```
	...
	' Search for an existing requesting activity
	     symbol
	 Dim ReqSym
	 Set ReqSym = Nothing
	 If Not ReqBizAct is Nothing Then
	  If ReqBizAct.Symbols.Count > 0 Then
	   Set ReqSym = ReqBizAct.Symbols.Item(0)
	  End If
	 End If

	 ' Create a requesting activity if not found
	 If ReqBizAct is Nothing Then
	   Set ReqBizAct =
	      BizTrans.Processes.CreateNew
	   ReqBizAct.Stereotype =
	     "RequestingBusinessActivity"
	   ReqBizAct.Name = "Request"
	 End If
	...
	```
	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 Toolbox, 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 Toolbox, from the Browser, in a list or in a property sheet.
	
	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 attach an extension to the model.

Event handler	Description

Validate	[all objects] Runs when you change tabs or click **OK** or **Apply** in an object property sheet. Used to validate changes to object properties and to implement cascade updates.
	
	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.
	
	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.
	
	`Function %Validate%(obj, ByRef message)`
	` if obj.comment = "" then`
	`  %Validate% = False`
	`  message = "Comment cannot be empty"`
	` else`
	`  %Validate% = True`
	` end if`
	`End Function`

CanLinkKind	[link objects only] Runs when you create a link with a Toolbox tool or modify link ends in a property sheet. Used to restrict the kind and stereotype of the objects that can be linked together.
	
	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.
	
	In the following example, the source of the extended link must be a start object:
	
	`Function %CanLinkKind%(sourceKind,`
	`  sourceStereotype, destinationKind,`
	`  destinationStereotype)`
	` if sourceKind = cls_Start Then`
	` %CanLinkKind% = True`
	` end if`
	`End Function`

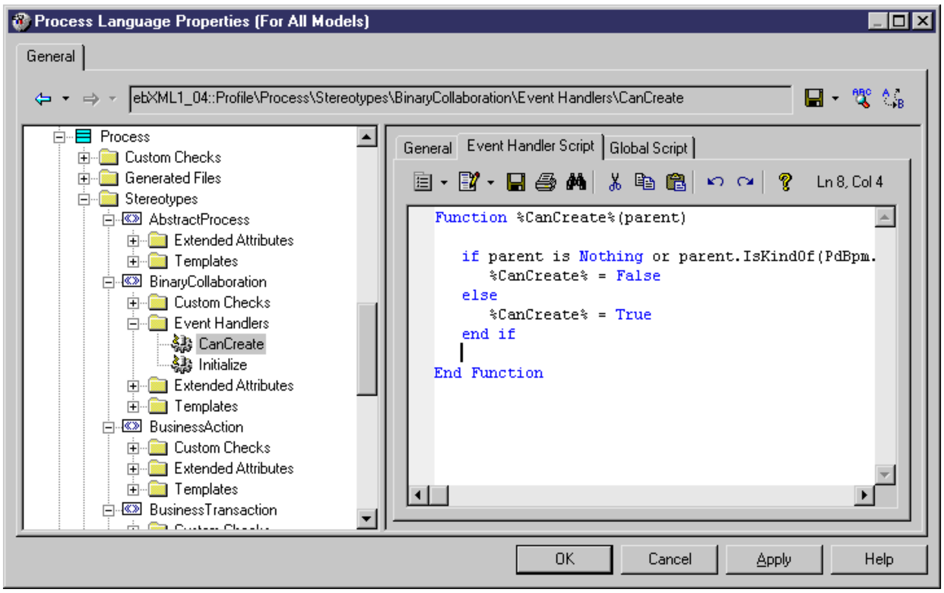
OnModelOpen	[models only] Runs immediately after a model is opened.
OnModelSave	[models only] Runs immediately before a model is saved.
OnModelClose	[models only] Runs immediately before a model is closed.
OnLanguageChangeRequest	[models only] Runs immediately before the model's DBMS or language definition file is changed. If the event handler returns false, then the language change is canceled.

Event handler	Description
OnLanguageChanging	[models only] Runs immediately after the model's DBMS or language definition file is changed, but before any transformations are applied to objects to make them conform with the new language definition.
OnLanguageChanged	[models only] Runs immediately after the model's DBMS or language definition file is changed and the object transformations are applied.
OnNewFromTemplate	[models only] Runs immediately after a model or a project is created from a model or project template.
BeforeDatabaseGenerate	[PDM models only] Runs immediately before generating a database.
AfterDatabaseGenerate	[PDM models only] Runs immediately after generating a database.
BeforeDatabaseReverseEngineer	[PDM models only] Runs immediately before reverse-engineering a database.
AfterDatabaseReverseEngineer	[PDM models only] Runs immediately after reverse-engineering a database.
GetEstimatedSize	[PDM only] Runs when the Estimate Database Size mechanic is called. For detailed information, see *Modifying the Estimate Database Size Mechanism* on page 214.

Adding 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:



```
Function %CanCreate%(parent)

   if parent is Nothing or parent.IsKindOf(PdBpm.
      %CanCreate% = False
   else
      %CanCreate% = True
   end if

End Function
```

5. Click Apply to save your changes.

Event Handler Properties

You specify the properties for an event handler by selecting its entry in the resource editor.

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. For more information on defining a script and using the **Global Script** tab, see *Defining the script of a custom check* on page 88 and *Using the global script* on page 91.

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 100) or form buttons (see *Forms (Profile)* on page 70).

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 102).

```
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 *Chapter 7, Scripting PowerDesigner* on page 323.

Creating a Method

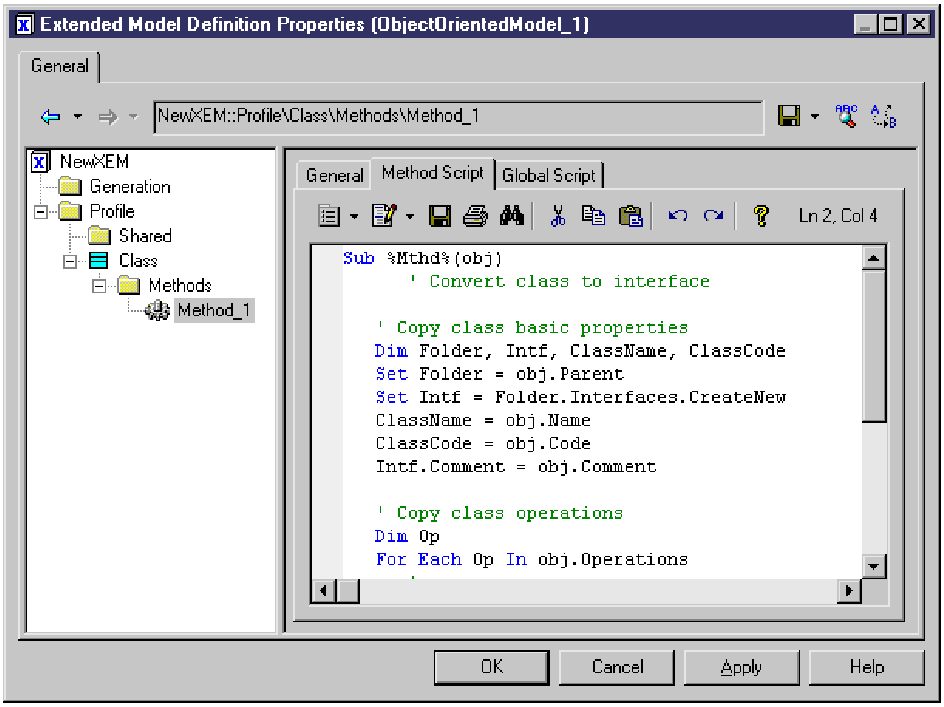
You can create a method in a profile.

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.



```
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
```

4. Click Apply.

Method Properties

You specify the properties for a method by selecting its entry in the resource editor.

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 88 and *Using the global script* on page 91.

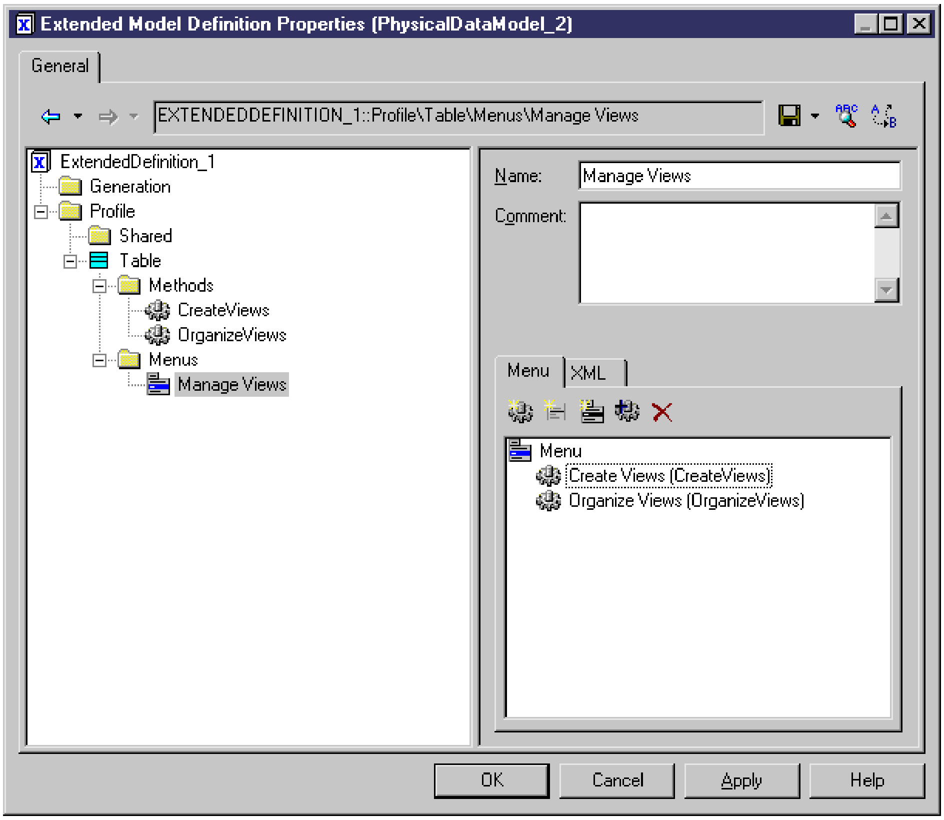
Menus (Profile)

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 97 and *Transformations and Transformation Profiles (Profile)* on page 110.

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 102).


Extended Model Definition Properties (PhysicalDataModel_2)
General
EXTENDEDDEFINITION_1::Profile\Table\Menus\Manage Views
ExtendedDefinition_1
Generation
Profile
Shared
Table
Methods
CreateViews
OrganizeViews
Menus
Manage Views
Name:
Manage Views
Comment:
Menu
XML
Menu
Create Views (CreateViews)
Organize Views (OrganizeViews)
OK
Cancel
Apply
Help

4. Click Apply to save your changes.

Menu Properties

You specify the properties for a menu by selecting its entry in the resource editor.

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: • File > Export menu • Help menu • Object Contextual Menu • Tools menu Menus created on other metaclasses are only available on the contextual menu, and do not display a **Location** field.

Property	Description
Menu	This sub-tab provides tools to add items to your menu (see *Adding commands and other items to your menu* on page 102).
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. When you click OK, each selected method is added to your menu in the format: <Caption> (<Method/Transformation name>). 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. 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.

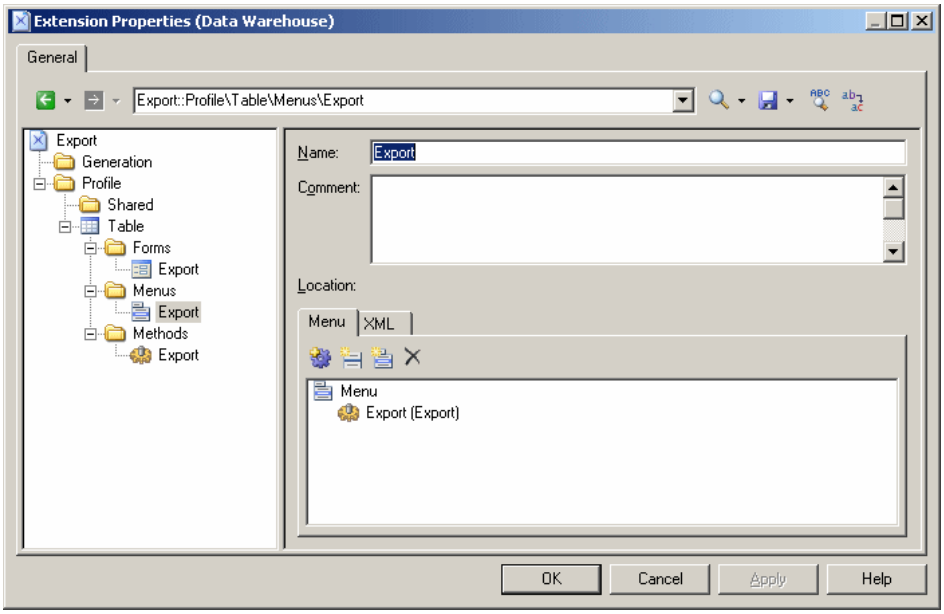
Example: Opening a Dialog Box from a Menu

In this example, we will create a menu command to export object properties to an XML file via a dialog box.

1. Create a new extension file (see *Creating an Extension File* on page 23) in a PDM and add the `Table` metaclass (see *Adding a Metaclass to a Profile* on page 47).
2. Right-click the `Table` metaclass and select **New** > **Form**. Enter `Export` in the **Name** field, and select `Dialog Box` from the **Type** list.
3. Click the **Edit Field** tool to add an edit field control, and call it `Filename`.
4. Right-click the `Table` metaclass and select **New** > **Method**. Enter `Export` in the **Name** field, click the **Method Script** tab and enter the following code:

```
Sub %Method%(obj)
' Exports an object to 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.xml"

    ' Show dialog
    If dlg.ShowDialog() Then
        ' Retrieve customer value for filename control
        Dim filename
        filename = dlg.GetValue("Filename")

        ' Process the export algorithm...
        ' (Actual export code not included in this example)

        Output "Exporting object " + obj.Name + " to file " +
filename
    End If

    ' Free dialog object
    dlg.Delete
    Set dlg = Nothing
End If
End Sub
```

5. Right-click the `Table` metaclass and select **New** > **Menu**. Enter `Export` in the **Name** field, and then click the **Add Command** tool and select the `Export` method:



Name: Export

Menu
    Export (Export)

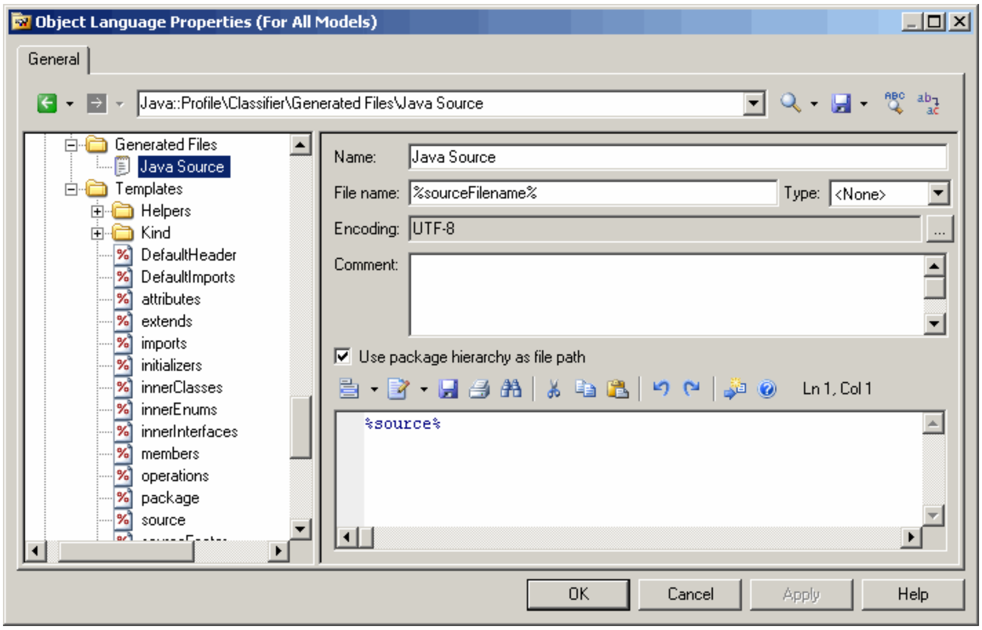
6. Click **OK** to save your changes and return to your model. When you next right-click a table in a diagram or the browser, the **Export** command is available in the contextual menu.

Templates and Generated Files (Profile)

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.

The PowerDesigner Generation Template Language (GTL) is used to generate files from metaclasses and for scripting (see *Chapter 5, Customizing Generation with GTL* on page 263). 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.

In the following example, the Generated Files category for classifiers contains a `Java Source` entry, which contains a reference to the template `%source%`. When the file is generated for a given classifier or for the instances of a classifier with a selected stereotype or criterion, it will have the name specified in the **File name** field, and will contain the contents generated by this template:



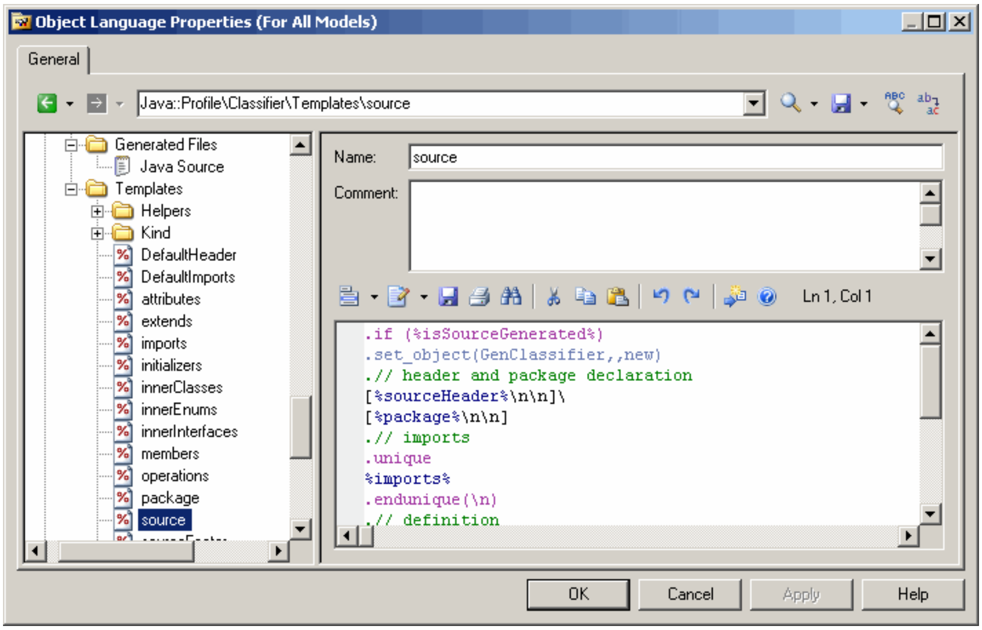
Name: Java Source
File name: %sourceFilename%
Type: <None>
Encoding: UTF-8
Comment:
☑ Use package hierarchy as file path

```
%source%
```

Note: If you position your cursor between the percent signs surrounding this or any other template name and press **F12**, you will either jump directly to the referenced template or, if several templates share the same name, a Results diaglog will open allowing you to select the template you want to navigate to.

The referenced template, `source`, contains GTL code to generate the contents of the file, including references to other templates called:

- `%isSourceGenerated%`
- `%sourceHeader%`
- `%package%`
- `%imports%`



Java::Profile\Classifier\Templates\source

Name: source

Comment:

```
.if (%isSourceGenerated%)
.set_object(GenClassifier,,new)
.// header and package declaration
[%sourceHeader%\n\n]\
[%package%\n\n]
.// imports
.unique
%imports%
.endunique(\n)
.// definition
```

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.

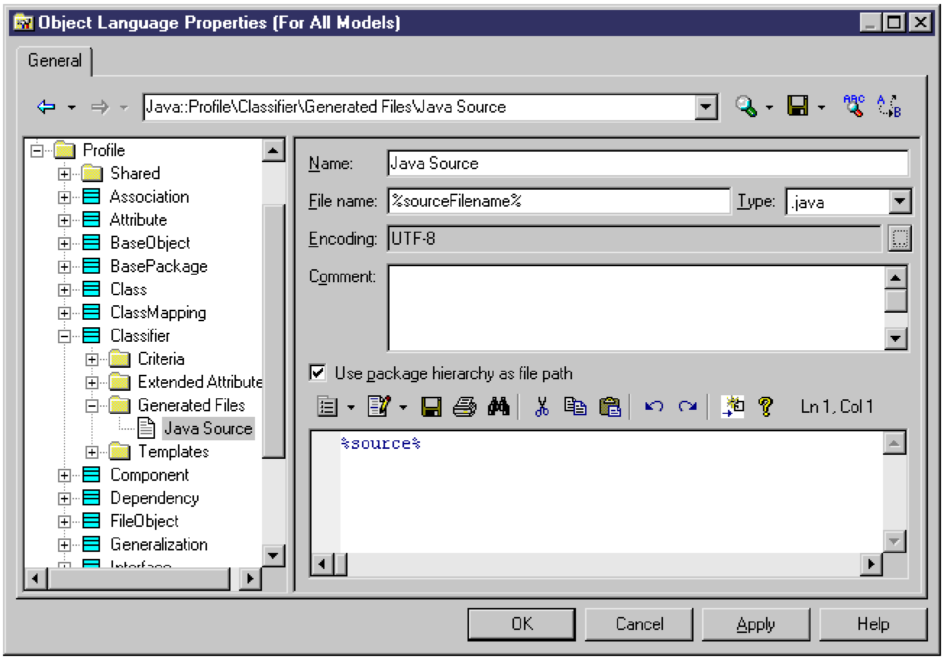
3. [optional] Enter a comment to explain the use of the template.
4. Enter the template body using GTL in the central box.

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 extension attached to the model contains a generated file name identical to one defined in the main resource file, then the extension 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:



Name: Java Source
File name: %sourceFilename%
Type: .java
Encoding: UTF-8
Comment:
[✓] Use package hierarchy as file path

```
%source%
```

4. Click Apply to save your changes.

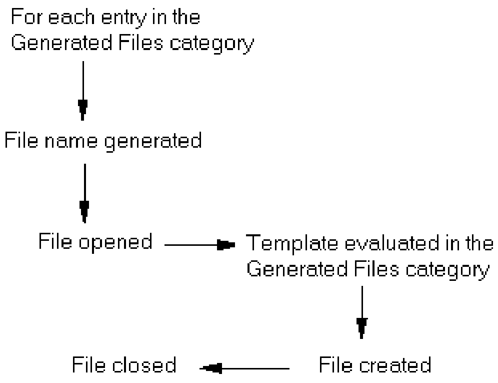
Each generated file has the following properties:

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`. 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. If this field contains a recognized extension, the code is displayed with the corresponding language editor and syntactic coloring.
Type	Specifies the type of file to provide appropriate syntax coloring in the Preview window.
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, where you can specify the following options: • Encoding - Encoding format of the generated file • Abort on character loss - Specifies to stop generation if characters cannot be identified and are to be lost in current encoding
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 *Chapter 5, Customizing Generation with GTL* on page 263).

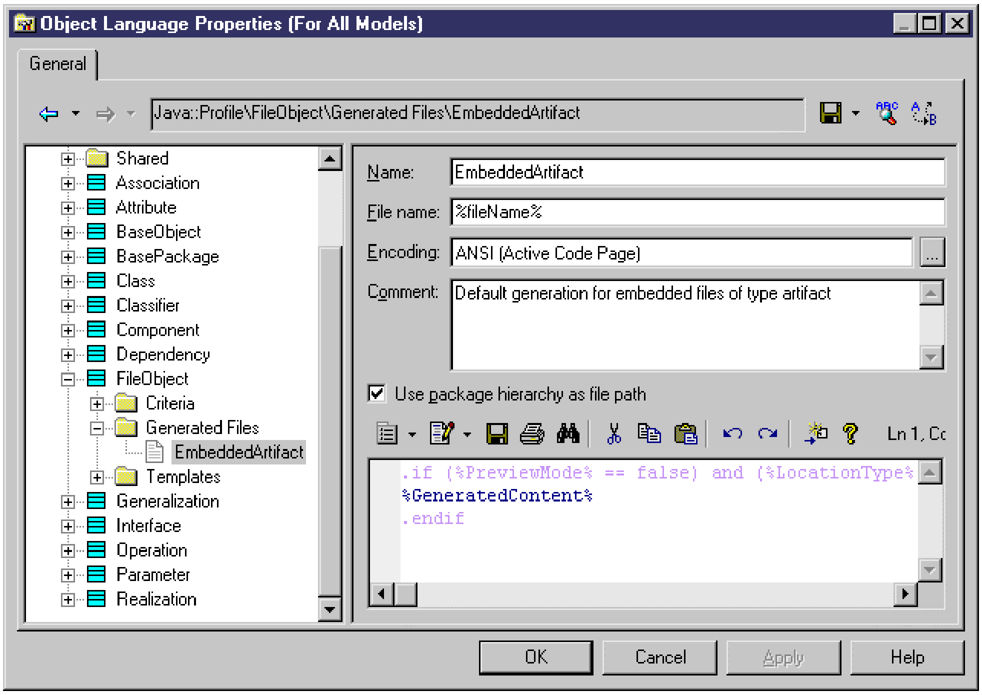
Generated File Examples

A file is generated for each entry in the `Generated File` category under a metaclass, stereotype or criterion when the File name field is not empty. If no file name or only an extension is specified, the content of the generated file can still be viewed in the object's **Preview** tab.

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


For each entry in the Generated Files category
File name generated
File opened
Template evaluated in the Generated Files category
File created
File closed

In the following 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, and whose **File name** field contains the template for the name of the file to be generated. The text field at the bottom displays the code of the template of the file to generate:



Name: EmbeddedArtifact

File name: %fileName%

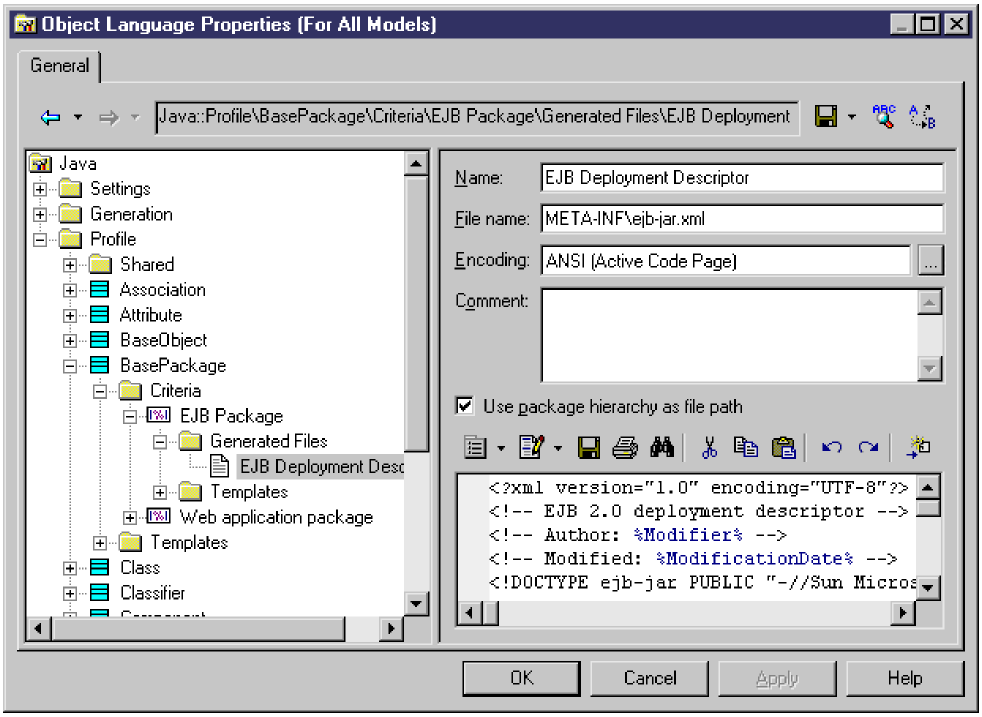
Encoding: ANSI (Active Code Page)

Comment: Default generation for embedded files of type artifact

[✓] Use package hierarchy as file path

```
.if (%PreviewMode% == false) and (%LocationType%
%GeneratedContent%
.endif
```

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



Java::Profile\BasePackage\Criteria\EJB Package\Generated Files\EJB Deployment

Name: EJB Deployment Descriptor

File name: META-INF\ejb-jar.xml

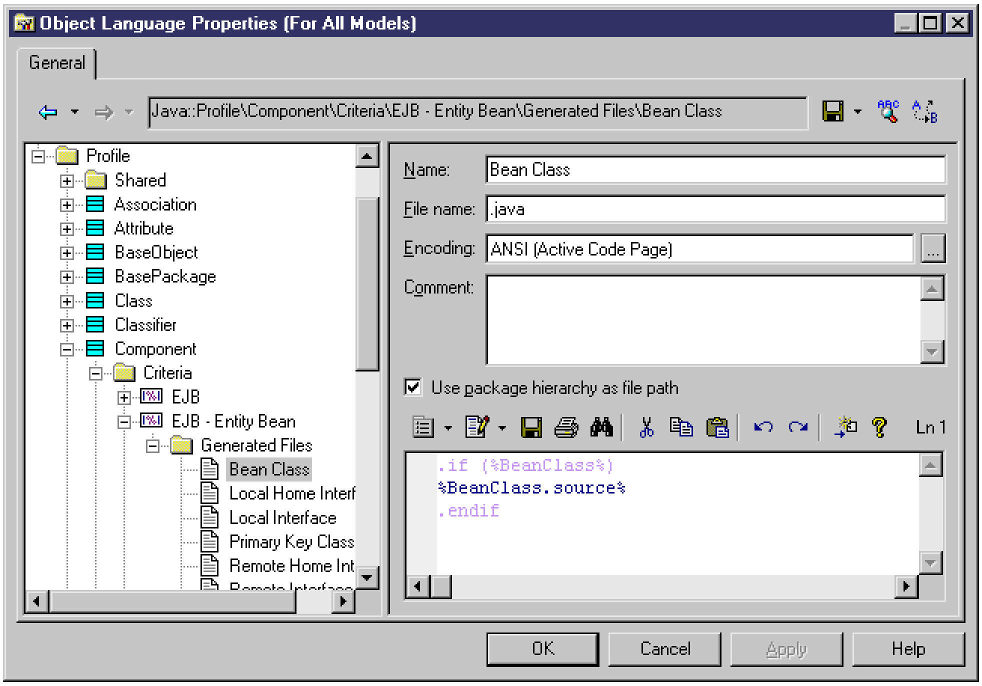
Encoding: ANSI (Active Code Page)

Comment:

[✓] Use package hierarchy as file path

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- EJB 2.0 deployment descriptor -->
<!-- Author: %Modifier% -->
<!-- Modified: %ModificationDate% -->
<!DOCTYPE ejb-jar PUBLIC "-//Sun Micros
```

In this example, there is no file generated since the **File name** field contains only an extension beginning with a . (dot) character. The contents of the file is only available on the **Preview** tab of the component (EJB - Entity Bean) property sheet:



Name: Bean Class
File name: .java
Encoding: ANSI (Active Code Page)
Use package hierarchy as file path ✓

```
.if (%BeanClass%)
%BeanClass.source%
.endif
```

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 extension 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 114) during model generation, or on demand. For more information, see *Core Features Guide > Linking and Synchronizing Models > Generating Models and Model Objects >*

Generating Models > Model Generation Options Window > Applying Model Transformations.
- As a command in a user-defined menu (see *Menus (Profile)* on page 100)

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:

General Tab

The General tab contains 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. Parameters: • source: object to duplicate • tag [optional]: identifier 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. Parameters: • source: source object • target: target object • tag [optional]: identifier Returns:
GetSource	Retrieves the source object of a generated object. Parameters: • target: target object • tag [optional]: identifier Returns: Source object
GetTarget	Retrieves the target object of a source object. Parameters: • source: source object • tag [optional]: identifier Returns: Target object

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, the transformation is used as an internal identifier of the target object.
- If the source object is transformed into several objects, 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
```

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 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 extension (see *Transformations and Transformation Profiles (Profile)* on page 110). Each profile is identified by the model in which the current extension file 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 extension is in a PDM, the transformation profile can be used during PDM to OOM generation or reverse engineering

Property	Description
Family and subfamily	[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 extension file. When the resource file family corresponds to the extension family, it suggests that the extension 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 extension is the source model, and when the constraints defined in the model type, family, and subfamily boxes are met. Any object created during pre-generation is automatically added to the list of objects used in generation. These changes to the source model are temporary and are reversed after generation is complete. 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 extension is the target model. For example, you can define a transformation profile with a transformation that automatically applies the correct naming conventions to the generated model.

CHAPTER 3 Object, Process, and XML Language Definition Files

A separate definition file is supplied for each OOM, BPM, and XSM language supported by PowerDesigner, which defines the syntax and guidelines for generating objects and implementing stereotypes, data types, scripts and constants for the language. You must select a language definition file when you create an OOM, BPM, or XSM.

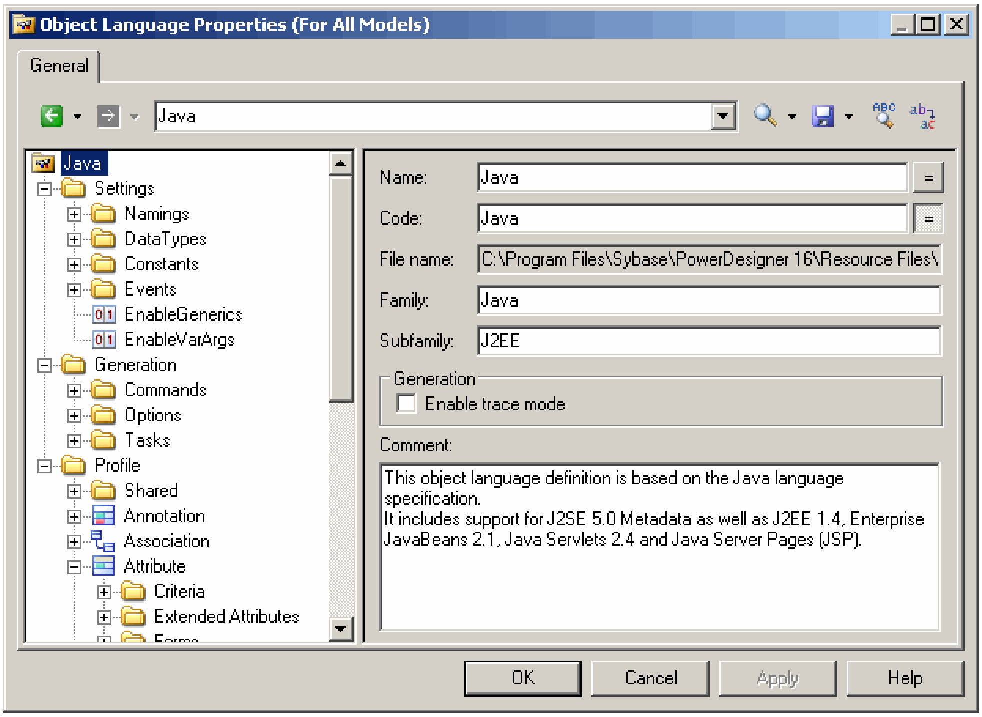
The following types of language definition files are explained in this chapter:

- OOM - Object language definition files (.xol)
- BPM – Business process language definition files (.xpl)
- XSM - XML language definition files (.xsl)

Note: The PDM uses a different form of definition file (see *Chapter 4, DBMS Definition Files* on page 127), and other model types do not have definition files but can be extended with extension files (see *Chapter 2, Extension Files* on page 21).

All target languages have the same basic category structure, but the detail and values of entries differs for each language:

- Settings - 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.
- Generation - contains generation commands, options, and task.
- Profile - contains extensions on metaclasses.


Object Language Properties (For All Models)

General

Java

Name: Java

Code: Java

File name: C:\Program Files\Sybase\PowerDesigner 16\Resource Files\

Family: Java

Subfamily: J2EE

Generation

[ ] Enable trace mode

Comment:

This object language definition is based on the Java language specification.
It includes support for J2SE 5.0 Metadata as well as J2EE 1.4, Enterprise JavaBeans 2.1, Java Servlets 2.4 and Java Server Pages (JSP).

The root node of each file contains the following properties:

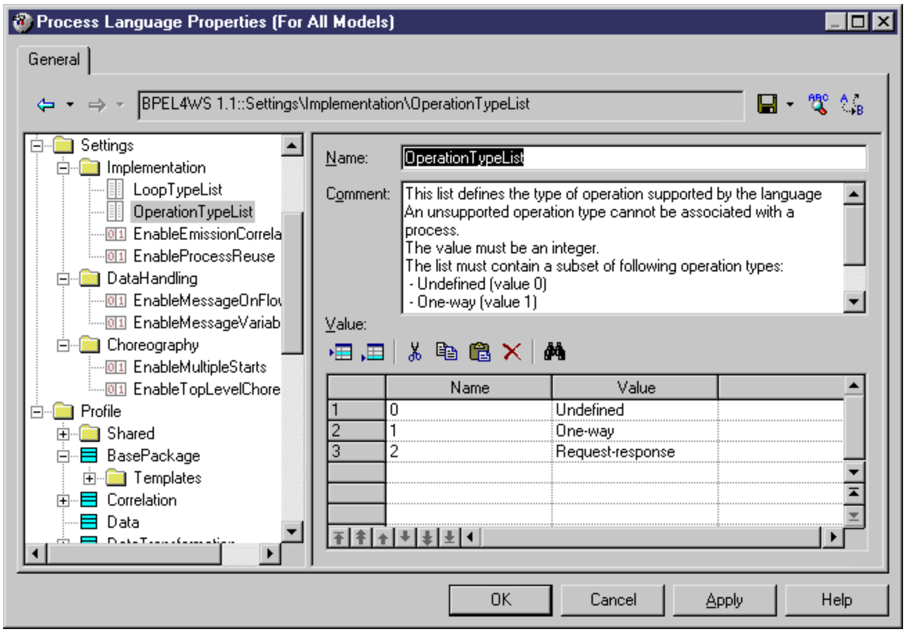
Property	Description
Name	Specifies the name of the target language.
Code	Specifies the code of the target language.
File Name	[read-only] Specifies the path to the .xol, xpl, or .xsl file. If the target language has been copied to your model, this field is empty.
Version	[read-only] Specifies the repository 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. 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 107).
Comment	Specifies additional information about the target language.

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



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.
The list must contain a subset of following operation types:
- Undefined (value 0)
- One-way (value 1)

	Name	Value
1	0	Undefined
2	1	One-way
3	2	Request-response

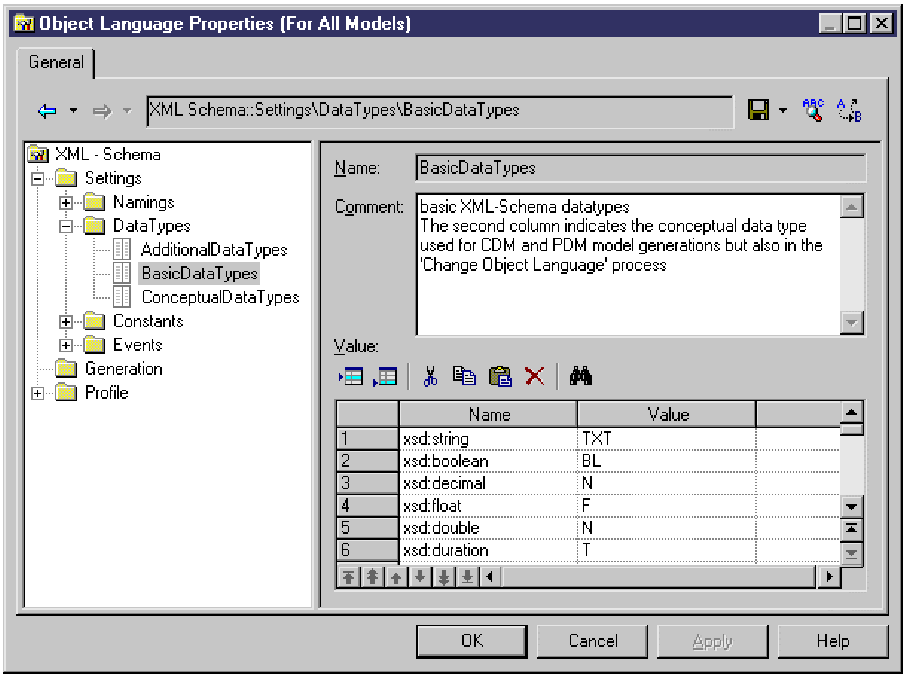
- *DataHandling* - [executable BPM only] Gathers options for managing data in the language. The following constant values are defined by default:
  - *EnableMessageOnFlow* - indicates if a message format can be associated to a flow or not. The default value is Yes
  - *EnableMessageVariable* - enables 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: 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.



Name: BasicDataTypes

Comment: basic XML-Schema datatypes
The second column indicates the conceptual data type used for CDM and PDM model generations but also in the 'Change Object Language' process

Value:

	Name	Value	
1	xsd:string	TXT	
2	xsd:boolean	BL	
3	xsd:decimal	N	
4	xsd:float	F	
5	xsd:double	N	
6	xsd:duration	T	

- *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: 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:

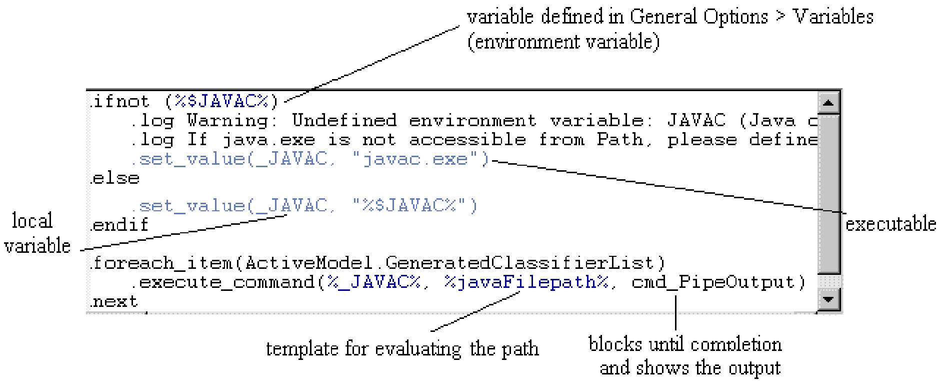
- ConceptualDataTypes - The Value column indicates the XML language data type used for model generations. Conceptual data types are the internal data types of PowerDesigner, and cannot be modified
- XsmDataTypes- Data types for generations from the XML model

Generation Category

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

The following sub-categories are available:

- *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 *Chapter 5, Customizing Generation with GTL* on page 263), and must be included within tasks to be evoked.



```
.ifnot (%$JAVAC%)
    .log Warning: Undefined environment variable: JAVAC (Java c
    .log If java.exe is not accessible from Path, please define
    .set_value(_JAVAC, "javac.exe")
.else
    .set_value(_JAVAC, "%$JAVAC%")
.endif
.foreach_item(ActiveModel.GeneratedClassifierList)
    .execute_command(%_JAVAC%, %javaFilepath%, cmd_PipeOutput)
.next
```

variable defined in General Options > Variables (environment variable)
local variable
executable
template for evaluating the path
blocks until completion and shows the output

- *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> '%'
```

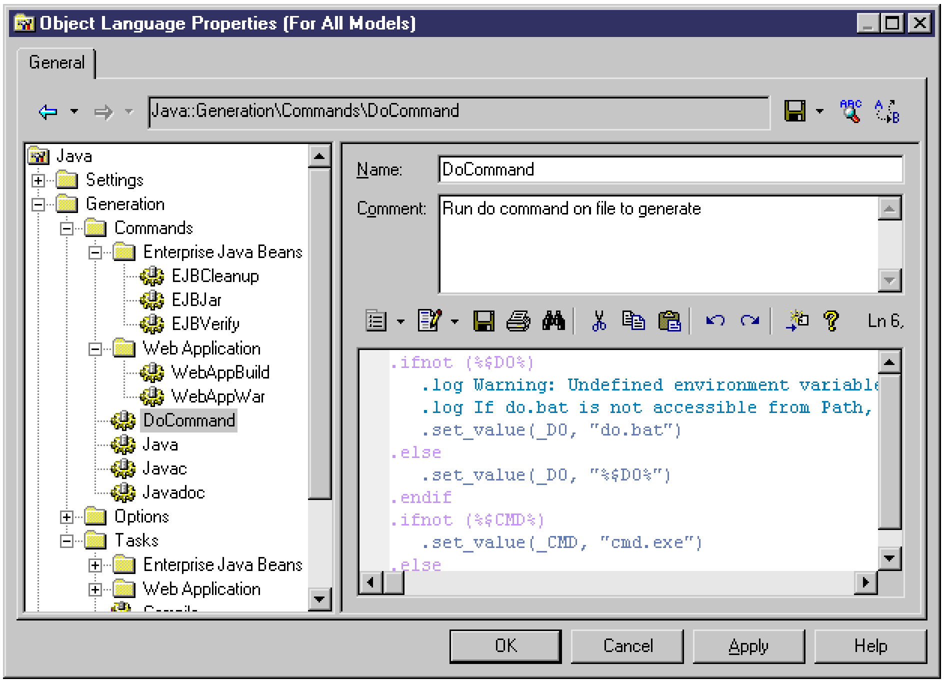
For example, for a boolean option named GenerateComment, %GenOptions.GenerateComment% will evaluate to either true or false in a template, depending on the value specified in the Generation dialog Options tab.

- *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:

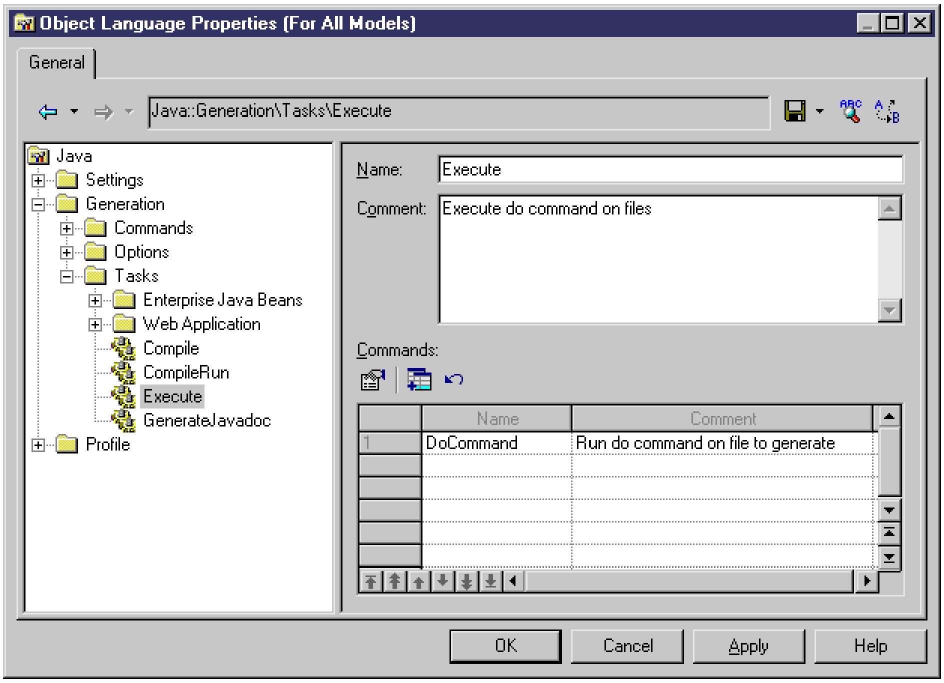


Name: DoCommand

Comment: Run do command on file to generate

```
.ifnot (%$DO%)
   .log Warning: Undefined environment variable
   .log If do.bat is not accessible from Path,
   .set_value(_DO, "do.bat")
.else
   .set_value(_DO, "%$DO%")
.endif
.ifnot (%$CMD%)
   .set_value(_CMD, "cmd.exe")
.else
```

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:



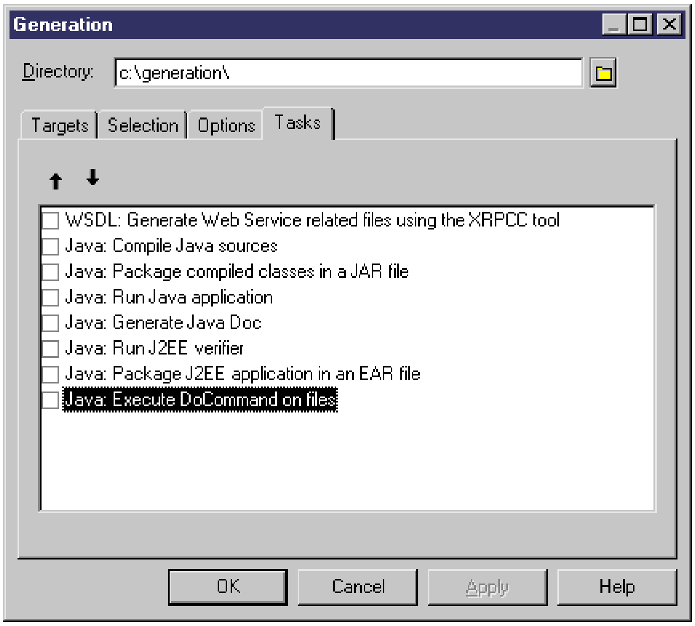
Name: Execute

Comment: Execute do command on files

Commands:

	Name	Comment
1	DoCommand	Run do command on file to generate

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):



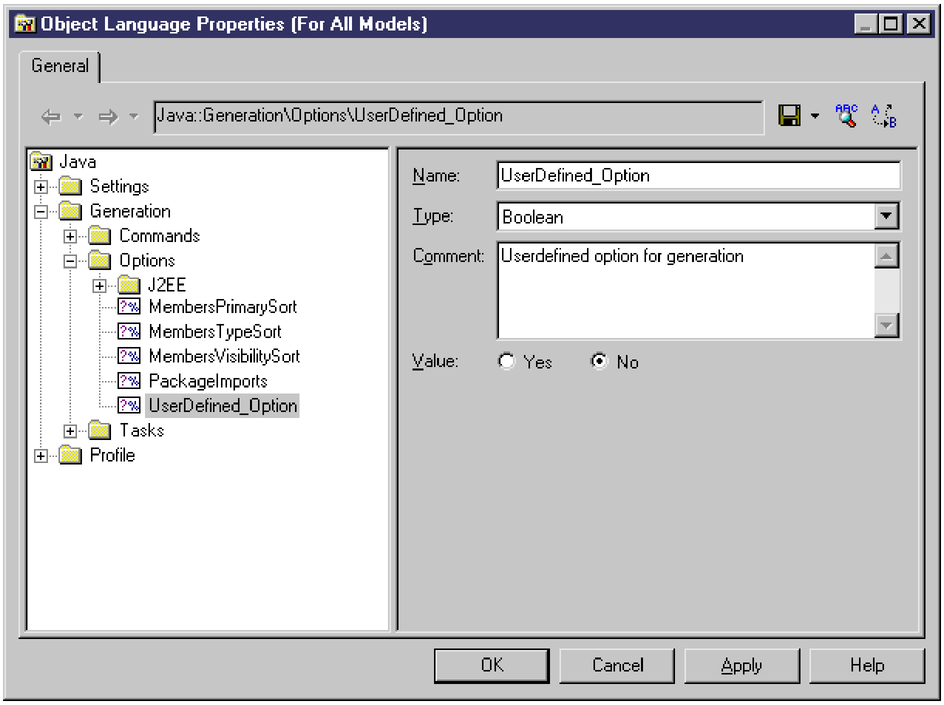
Directory: c:\generation\

- [ ] WSDL: Generate Web Service related files using the XRPCC tool
- [ ] Java: Compile Java sources
- [ ] Java: Package compiled classes in a JAR file
- [ ] Java: Run Java application
- [ ] Java: Generate Java Doc
- [ ] Java: Run J2EE verifier
- [ ] Java: Package J2EE application in an EAR file
- [ ] Java: Execute DoCommand on files

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:



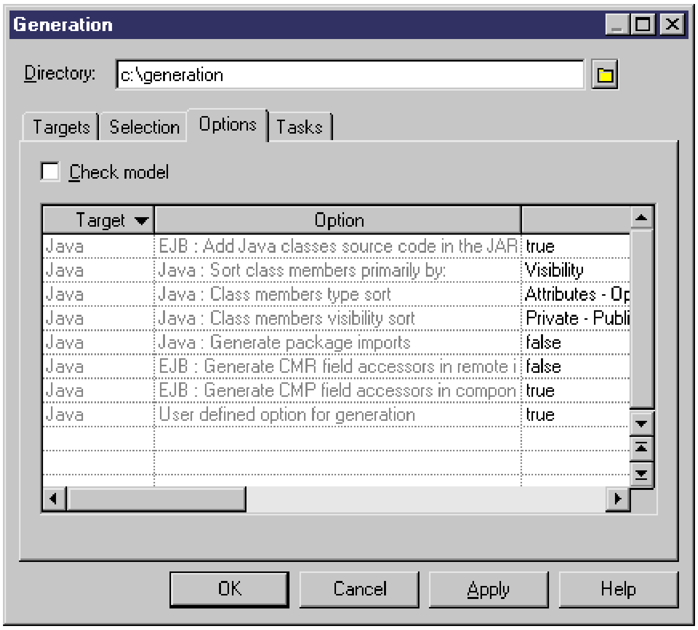
Name: UserDefined_Option

Type: Boolean

Comment: Userdefined option for generation

Value: Yes [✓ No]

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):



Directory: c:\generation

☐ Check model

Target	Option	
Java	EJB : Add Java classes source code in the JAR	true
Java	Java : Sort class members primarily by:	Visibility
Java	Java : Class members type sort	Attributes - Op
Java	Java : Class members visibility sort	Private - Publi
Java	Java : Generate package imports	false
Java	EJB : Generate CMR field accessors in remote i	false
Java	EJB : Generate CMP field accessors in compon	true
Java	User defined option for generation	true

Note: For detailed information about creating and modifying generation templates, see *Chapter 5, Customizing Generation with GTL* on page 263. 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 (Definition Files)

The language definition file Profile category can contain Stereotypes, Extended attributes, Methods and so on, to extend the metaclasses defined in the PowerDesigner metamodel.

In object languages, the `Shared/Extended Attribute Types` category contains various attributes used to control object language support within PowerDesigner. The **Object Container** variable specifies the default container for implementing associations. This attribute has an editable list of possible values for each object language, from which you can select a default value for your language. You can, if necessary, override this default using the **Default association container** model option.

For detailed information about working with the Profile category, see *Chapter 2, Extension Files* on page 21.

CHAPTER 4 DBMS Definition Files

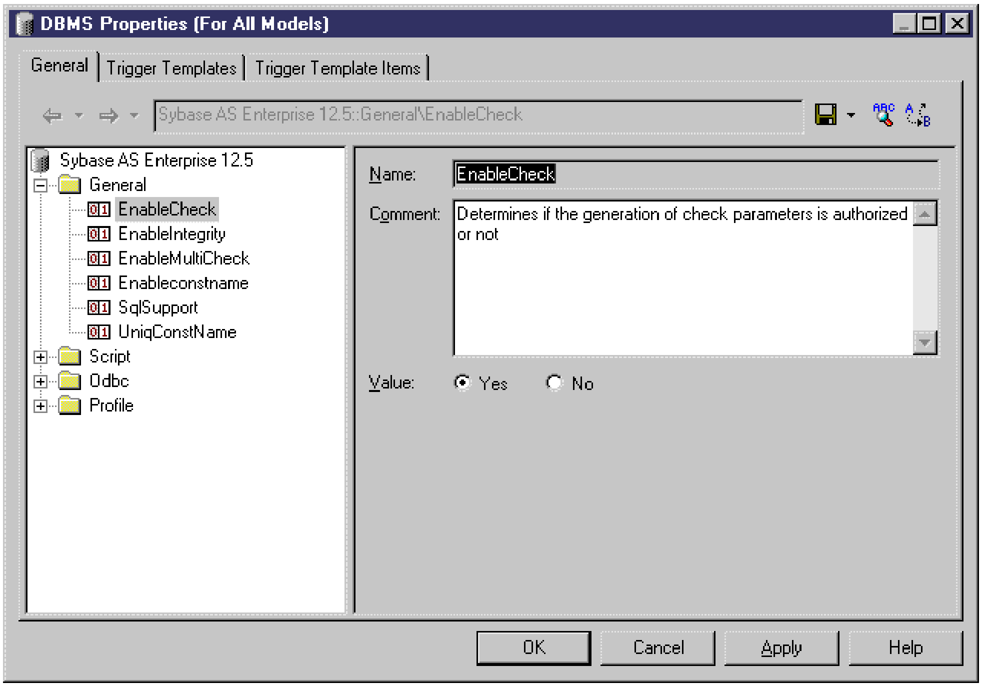
A DBMS definition file is a PowerDesigner resource file that provides PowerDesigner with the information necessary to model, reverse-engineer, and generate a particular DBMS.

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.



Name: EnableCheck

Comment: Determines if the generation of check parameters is authorized or not

Value: [✓ Yes] No

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 *Opening Resource Files in the 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 146). 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 147)
    - *Format* - parameters for allowed characters (see *Format Category* on page 148)
    - *File* - header, footer and usage text items used during generation (see *File Category* on page 150)
    - *Keywords* - the list of SQL reserved words and functions (see *Keywords Category* on page 152)
  - *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 154).
  - *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 209).
  - *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 110 and *Core Features Guide > Linking and Synchronizing Models > Generating Models and*

Model Objects > Generating Models > Model Generation Options Window > Applying Model Transformations.

- *Profile* - allows you to define extended attribute types and extended attributes for database objects. For more information, see *Profile Category (DBMS)* on page 212.

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. Triggers are retained when you change target within the same family. 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. In addition, for Oracle, there is a tab for database package templates.

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

For more information, see *Data Modeling > Building Data Models > Triggers and Procedures*

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 227.

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 135.

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 134.

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 *Chapter 5, Customizing Generation with GTL* on page 263.

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%
```

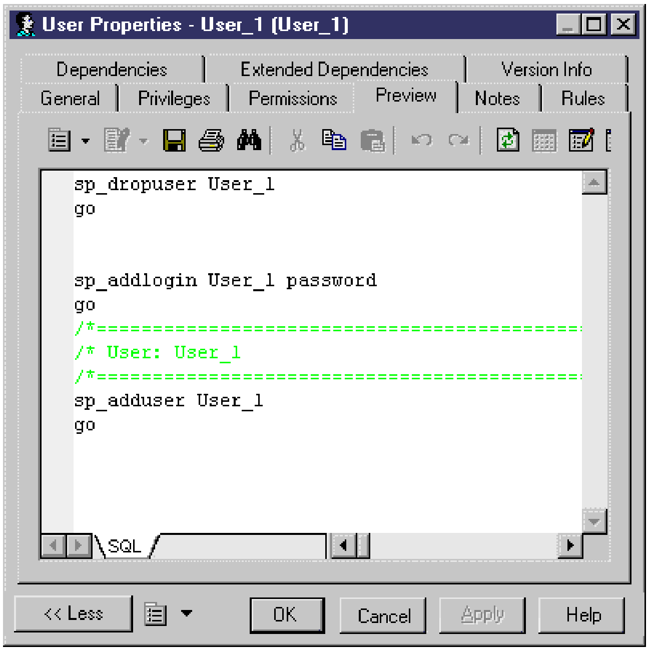
%CreatePartition% generates the statement for creating the number of table partitions specified in %ExtTablePartition%.

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:



```
sp_dropuser User_1
go


sp_addlogin User_1 password
go
/*==========================================
/* User: User_1
/*==========================================
sp_adduser User_1
go
```

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
```

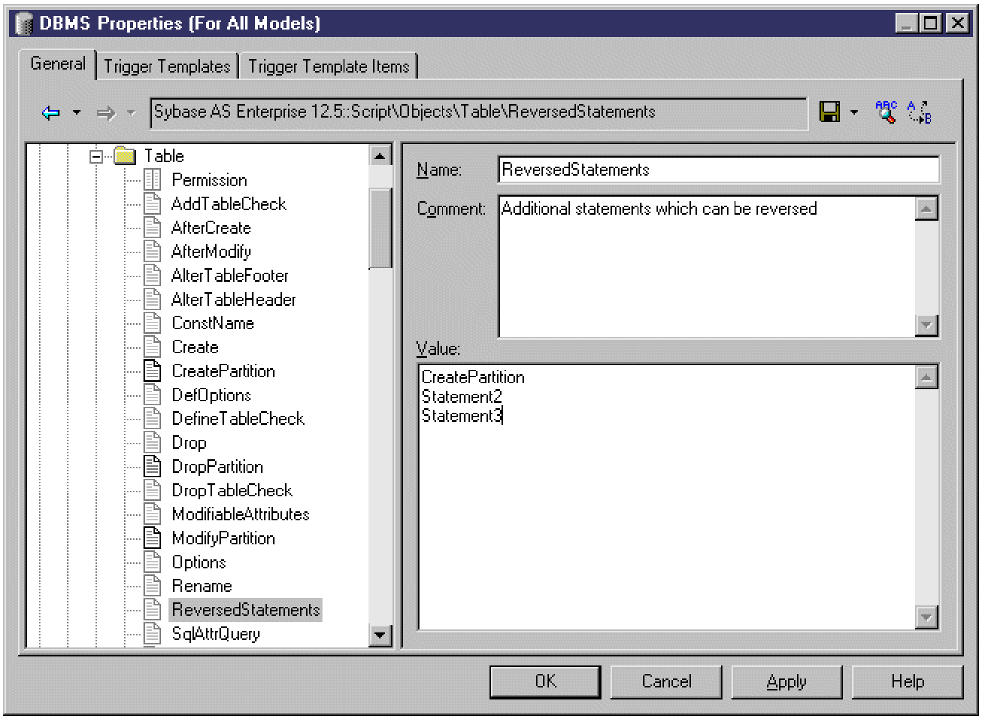
For more information on the PowerDesigner Generation Template Language (GTL), see *Chapter 5, Customizing Generation with GTL* on page 263.

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:



Name: ReversedStatements

Comment: Additional statements which can be reversed

Value:

```
CreatePartition
Statement2
Statement3
```

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

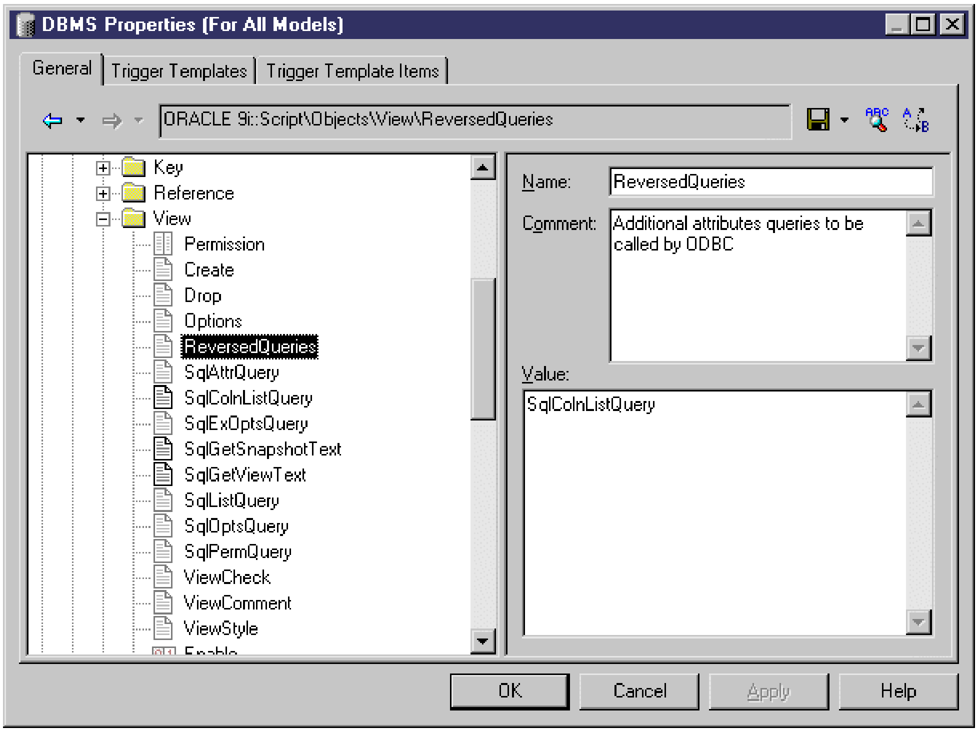
The DBMS contains live database reverse engineering queries for retrieving objects (like Table, Columns, and so on) from the database.

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. If `SqlListQuery` is not defined, standard functions are used to retrieve objects. `SqlAttrQuery, SqlOptsQuery` etc. will then be executed, if defined. `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 138.



Name: ReversedQueries

Comment: Additional attributes queries to be called by ODBC

Value: SqlColnListQuery

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 226.

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:

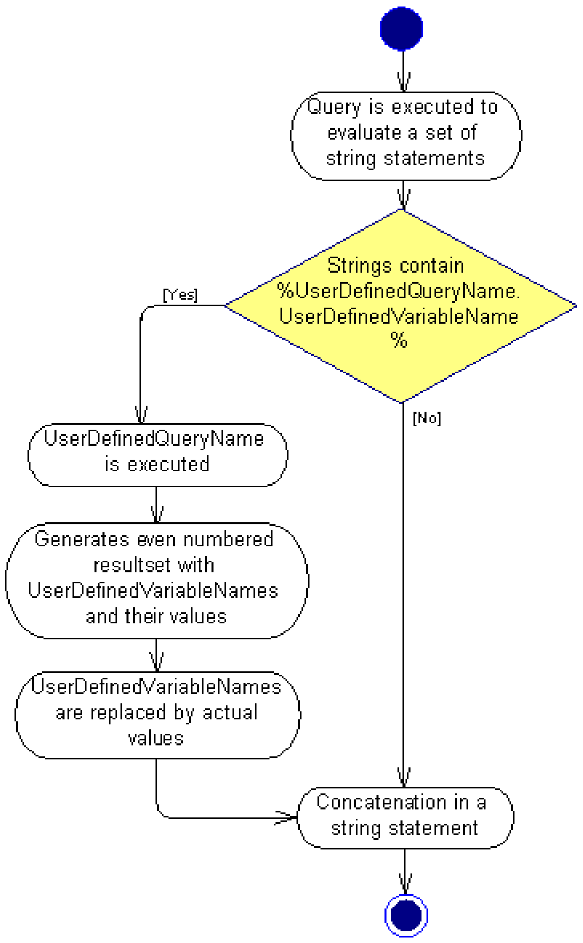
```
%UserDefinedQueryName.UserDefinedVariableName%
```

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:


Query is executed to evaluate a set of string statements
Strings contain %UserDefinedQueryName. UserDefinedVariableName %
[Yes]
[No]
UserDefinedQueryName is executed
Generates even numbered resultset with UserDefinedVariableNames and their values
UserDefinedVariableNames are replaced by actual values
Concatenation in a string statement

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`.

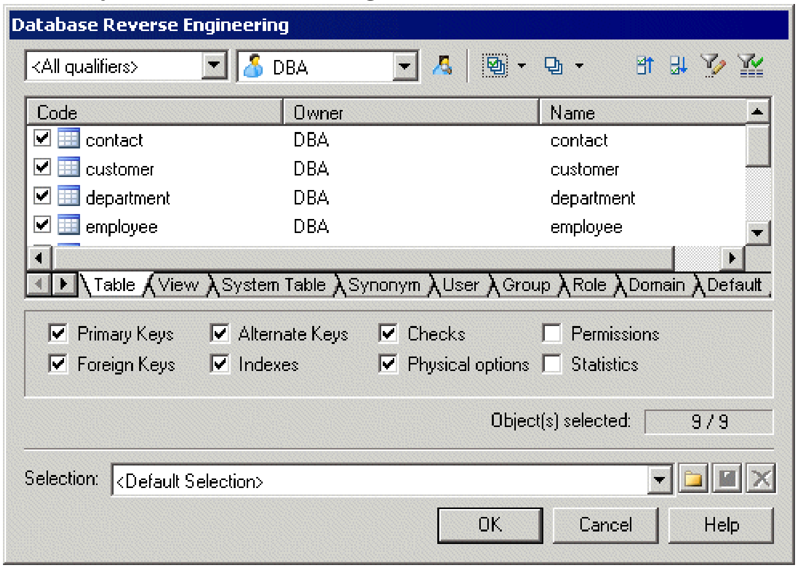
`SqlExpression` is followed by a user-defined variable defined as follow:

```
{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.



<All qualifiers> DBA

Code	Owner	Name
✓ contact	DBA	contact
✓ customer	DBA	customer
✓ department	DBA	department
✓ employee	DBA	employee

- [✓] Primary Keys
- [✓] Alternate Keys
- [✓] Checks
- [ ] Permissions
- [✓] Foreign Keys
- [✓] Indexes
- [✓] Physical options
- [ ] Statistics

Object(s) selected: 9 / 9

Selection: <Default Selection>

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.

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.

Creating 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 *Chapter 2, Extension Files* on page 21.

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

Defining 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 156.
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 154.

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: • Yes - Check parameters generated • 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: • Yes - Constraint names are used during generation • No - Constraint names are not used
EnableIntegrity	Specifies whether there are integrity constraints in the DBMS. The following settings are available: • Yes - Primary, alternate, and foreign key check boxes are available for database generation and modification • 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: • 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 • 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: • Yes - SQL syntax allowed and SQL Preview available • 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: • Yes - All constraint names (including index names) must be unique in the database • No - Constraint names must be unique for an object 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. Example: `/* */`
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. Example: `%%`
Quote	Specifies the character used to enclose string values. 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.

Item	Description
Terminator	Specifies the end of statement character, which is used to terminate create table, view, index, or the open/close database, and other statements. If empty, `BlockTerminator` is used instead.
UseBlockTerm	Specifies the use of `BlockTerminator`. The following settings are available: • Yes - `BlockTerminator` is always used • 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: • Yes – Quotes are systematically added to object codes during generation • No - Object codes are generated without quotes
CaseSensitivity UsingQuote	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 149.
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: • 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. • No - The Owner Prefix and Database Prefix options are unavailable

Item	Description

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.
	Example:
	`+-*/!=<>'"()`
	If the name of the table is "SALES+PROFITS", the generated create statement will be:
	`CREATE TABLE "SALES+PROFITS"`
	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: • Yes - Forces all generated script characters to lowercase • 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: • Yes - Forces all generated script characters to uppercase • No - Generates all script unchanged from the way objects are written in the model 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.

Item	Description
AlterFooter	Specifies footer text for a modify database script.
EnableMulti File	Specifies that multiple scripts are allowed. The following settings are available: • 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. • The One File Only check box is unavailable, and a single script includes all the statements. 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. 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. Example: `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.
	Example (Sybase ASE 11):
	`isql %NAMESCRIPT%`
	Corresponds to the %STARTCMD% variable (see *PDM Variables* on page 226).

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. Example: `sql`
TrgFooter	Specifies footer text for a triggers and procedures generation script.
TrgHeader	Header script for triggers and procedures generation.

Item	Description
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. Example: `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.
	Example:
	`char()`
	`charindex()`
	`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.
	Example:
	`convert()`
	`hextoint()`
	`inttohex() etc`

Item	Description

DateFunc	Specifies a list of SQL functions to use with dates.
	
	Example:
	
	`dateadd()`
	`datediff()`
	`datename() etc`

GroupFunc	Specifies a list of SQL functions to use with group keywords.
	
	Example:
	
	`avg()`
	`count()`
	`max() etc`

ListOperators	Specifies a list of SQL operators to use when comparing values, boolean, and various semantic operators.
	
	Example:
	
	`=`
	`!=`
	`not like etc`

NumberFunc	Specifies a list of SQL functions to use with numbers.
	
	Example:
	
	`abs()`
	`acos()`
	`asin() etc`

OtherFunc	Specifies a list of SQL functions to use when estimating, concatenating and SQL checks.
	
	Example:
	
	`db_id()`
	`db_name()`
	`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.
	
	Example (SQL Anywhere® 10) - USER is a reserved default value:
	
	`Create table CUSTOMER (`
	`Username varchar(30) default USER`
	`)`
	
	When you run this script, CURRENT DATE is recognized as a reserved default value.

Item	Description
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.

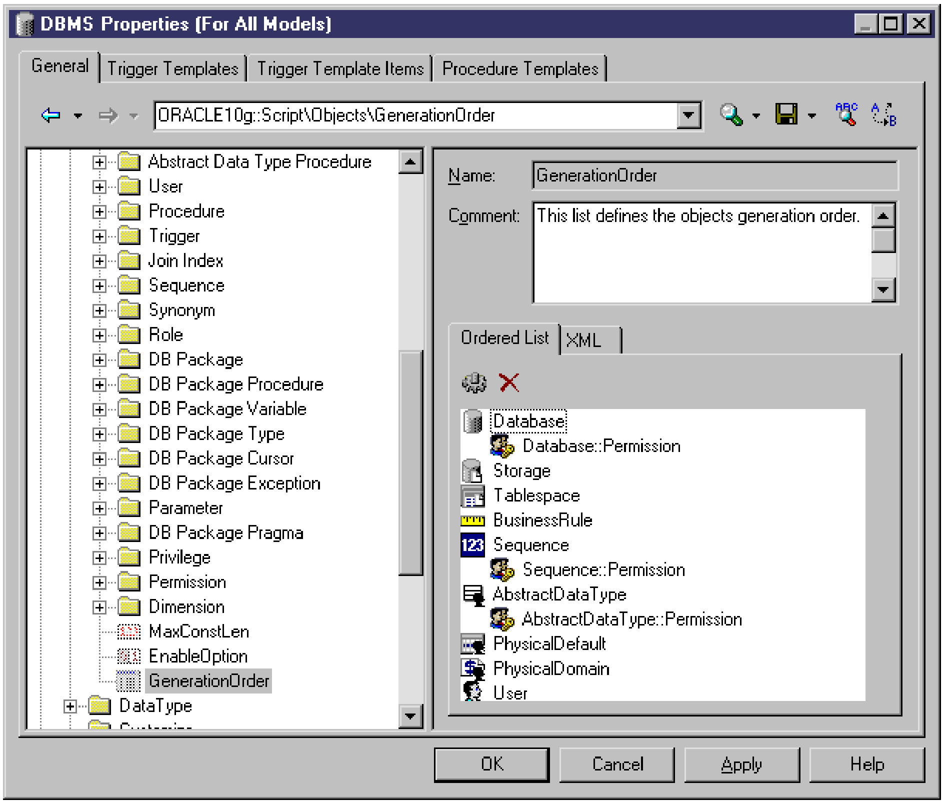
For more information, see *Physical Options* on page 218

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:



Name: GenerationOrder

Comment: This list defines the objects generation order.

Database
    Database::Permission
Storage
Tablespace
BusinessRule
Sequence
    Sequence::Permission
AbstractDataType
    AbstractDataType::Permission
PhysicalDefault
PhysicalDomain
User

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 144) 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.
	
	Example (adding a column):
	
	`%20:COLUMN% %30:DATATYPE% [default %DEFAULT%]`
	`[%IDENTITY%?identity:[%NULL%][%NOTNULL%]]`
	`[[constraint %CONSTNAME%] check (%CONSTRAINT%)]`

AfterCreate/ After-Drop/ AfterModify	Specifies extended statements executed after the main Create, Drop or Modify statements. For more information, see *Script generation* on page 131.
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/ Befor-eDrop/ BeforeModi-fy	Specifies extended statements executed before the main Create, Drop or Modify statements. For more information, see *Script generation* on page 131.
ConstName	Specifies a constraint name template for the object. The template controls how the name of the object will be generated. 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. Examples (ASE 15): • Table: CKT_%.U26:TABLE% • Column: CKC_%.U17:COLUMN%_%.U8:TABLE% • Primary Key: PK_%.U27:TABLE%
Create	[generation and reverse] Specifies the statement required to create the object. Example: `create table %TABLE%`

Item	Description
DefOptions	Specifies default values for physical options that will be applied to all objects. These values must respect SQL syntax. Example: `in default_tablespace` For more information, see *Physical Options* on page 218.

Drop	Specifies the statement required to drop the object.
	
	Example (SQL Anywhere 10):
	
	`if exists( select 1 from sys.systable`
	`  where table_name=%.q:TABLE%`
	`  and table_type in ('BASE', 'GBL TEMP')[%QUALIFIER%?`
	`  and creator=user_id(%.q:OWNER%)]`
	`) then drop table [%QUALIFIER%]%TABLE%`
	`end if`

Enable	Specifies whether an object is supported.

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:
	
	• Yes - The Owner list is enabled in the object's property sheet.
	• No – Owners are not supported for the object.
	
	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:
	
	`create [%UNIQUE%?%UNIQUE% :[%INDEXTYPE% ]]index [%QUALIFIER%]%INDEX% on [%CLUSTER%?cluster C_%TABLE%:[%TABLQUALIFIER%]%TABLE% (`
	` %CIDXLIST%`
	`)]`
	`[%OPTIONS%]`
	
	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 154).

Item	Description
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 131.
	
	Example (ASE 12.5):
	
	`ExtTablePartition`

Options	Specifies physical options for creating an object.
	
	Example (ASA 6):
	
	`in %s : category=tablespace`
	
	For more information, see *Physical Options* on page 218.

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.
	
	Example (table permissions in ASE 15):
	
	`SELECT / Sel`
	`INSER / Ins`
	`DELETE / Del`
	`UPDATE / Upd`
	`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 135.
Reversed Statements	Specifies a list of additional statements that will be reverse engineered. For more information, see *Script reverse engineering* on page 134.

SqlAttrQuery	Specifies a SQL query to retrieve additional information on objects reversed by `SQLListQuery`.
	
	Example (Join Index in Oracle 10g):
	
	`{OWNER ID, JIDX ID,  JIDXWHERE ...}`
	`select index_owner, index_name,`
	`outer_table_owner || '.' || outer_table_name || '.'`
	`|| outer_table_column || '=' || inner_table_owner`
	`|| '.' || inner_table_name || '.' || inner_ta-`
	`ble_column || ','`
	`from all_join_ind_columns`
	`where 1=1`
	`[  and index_owner=%.q:OWNER%]`
	`[  and index_name=%.q:JIDX%]`

Item	Description

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.
	
	Example (Dimension in Oracle 10g):
	
	`{ OWNER, DIMENSION }`
	`select d.owner, d.dimension_name`
	`from sys.all_dimensions d`
	`where 1=1`
	`[  and d.dimension_name=%.q:DIMENSION%]`
	`[  and d.owner=%.q:SCHEMA%]`
	`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.
	
	Example (Table in SQL Anywhere 10):
	
	`{OWNER, TABLE, OPTIONS}`
	`select u.user_name, t.table_name,`
	` 'in '+ f.dbspace_name`
	`from sys.sysuserperms u`
	` join sys.systab t on (t.creator = u.user_id)`
	` join sys.sysfile f on (f.file_id = t.file_id)`
	`where f.dbspace_name <> 'SYSTEM'`
	` and t.table_type in (1, 3, 4)`
	`[  and t.table_name = %.q:TABLE%]`
	`[  and u.user_name = %.q:OWNER%]`

SqlPermQuery	Specifies a SQL query to reverse engineer permissions granted on the object.
	
	Example (Procedure in SQL Anywhere 10):
	
	`{ GRANTEE, PERMISSION}`
	`select`
	`u.user_name grantee, 'EXECUTE'`
	`from sysuserperms u, sysprocedure s, sysprocperm p`
	`where (s.proc_name = %.q:PROC% ) and`
	`(s.proc_id = p.proc_id) and`
	`(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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • ConstName • Create, Drop • Enable, EnableSynonym • Header, Footer • Maxlen, MaxConstLen • ModifiableAttributes • Options, DefOptions • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery For a description of each of these common items, see *Common object items* on page 156.

AddTableCheck	Specifies a statement for customizing the script to modify the table constraints within an `alter table` statement.
	Example (SQL Anywhere 10):
	`alter table [%QUALIFIER%]%TABLE%`
	`  add [constraint %CONSTNAME% ]check (%.A:CON-`
	`STRAINT%)`

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.
	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.
	Example (Oracle 10g):
	`OBJECT`

Item	Description
AlterTable Footer	Specifies a statement to be placed after `alter table` statements (and before the terminator). Example: `AlterTableFooter = /* End of alter statement */`
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. Example: `AlterTableHeader = /* Table name: %TABLE% */`
DefineTable Check	Specifies a statement for customizing the script of table constraints (checks) within a `create table` statement. Example: `check (%CONSTRAINT%)`

DropTable Check	Specifies a statement for dropping a table check in an `alter table` statement.
	Example:
	`alter table [%QUALIFIER%]%TABLE%`
	`  delete check`

InsertIdentityOff	Specifies a statement for enabling insertion of data into a table containing an identity column.
	Example (ASE 15):
	`set identity_insert [%QUALIFIER%]%@OBJTCODE% off`

InsertIdentityOn	Specifies a statement for disabling insertion of data into a table containing an identity column.
	Example (ASE 15):
	`set identity_insert [%QUALIFIER%]%@OBJTCODE% on`

Item	Description

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.
	
	Example (Oracle 10g):
	
	`rename %OLDTABL% to %NEWTABL%`
	
	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.
	
	Example (SQL Anywhere 10):
	
	`{OWNER, TABLE, CONSTNAME, CONSTRAINT}`
	`select u.user_name, t.table_name,`
	` k.constraint_name,`
	` case(lcase(left(h.check_defn, 5))) when 'check' then substring(h.check_defn, 6) else h.check_defn end`
	`from sys.sysconstraint k`
	` join sys.syscheck h on (h.check_id = k.constraint_id)`
	` join sys.systab t on (t.object_id = k.table_object_id)`
	` join sys.sysuserperms u on (u.user_id = t.creator)`
	`where k.constraint_type = 'T'`
	` and t.table_type in (1, 3, 4)`
	`[  and u.user_name = %.q:OWNER%]`
	`[  and t.table_name = %.q:TABLE%]`
	`order by 1, 2, 3`

Item	Description

SqlListRefr Tables	Specifies a SQL query used to list the tables referenced by a table.
	
	Example (Oracle 10g):
	
	`{OWNER, TABLE, POWNER, PARENT}`
	`select c.owner, c.table_name, r.owner,`
	` r.table_name`
	`from sys.all_constraints c,`
	` sys.all_constraints r`
	`where (c.constraint_type = 'R' and c.r_con-`
	`straint_name = r.constraint_name and c.r_owner =`
	`r.owner)`
	`[ and c.owner = %.q:SCHEMA%]`
	`[ and c.table_name = %.q:TABLE%]`
	`union select c.owner, c.table_name,`
	` r.owner, r.table_name`
	`from sys.all_constraints c,`
	` sys.all_constraints r`
	`where (r.constraint_type = 'R' and r.r_con-`
	`straint_name = c.constraint_name and r.r_owner =`
	`c.owner)`
	`[ and c.owner = %.q:SCHEMA%]`
	`[ 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).
	
	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.
	
	Example (Oracle 10g):
	
	`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 164.
SqlXMLTable	Specifies a sub-query used to improve the performance of SqlAttrQuery (see *Common object items* on page 156).

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.
	
	Example (Oracle 10g):
	
	`comment on table [%QUALIFIER%]%TABLE% is`
	`%.q:COMMENT%`
	
	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.

Item	Description
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. 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: • Yes – The table constraint and index names must be different, and this will be tested during model checking • 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: • Add • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • ConstName • Create, Drop • Enable • Maxlen, MaxConstLen • ModifiableAttributes • Options, DefOptions • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery For a description of each of these common items, see *Common object items* on page 156.

AddColnCheck	Specifies a statement for customizing the script for modifying column constraints within an alter table statement.
	
	Example (Oracle 10g):
	
	`alter table [%QUALIFIER%]%TABLE%`
	` add [constraint %CONSTNAME%] check (%.A:CONSTRAINT%)`

Item	Description

AlterTableAdd Default	Specifies a statement for defining the default value of a column in an alter statement.
	
	Example (SQL Server 2005):
	
	`[[ 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: • Yes - `AddColnChck` can be used to modify the column check constraint in an `alter table` statement. • No - PowerDesigner copies data to a temporary table before recreating the table with the new constraints. See also `AddColnChck`.
AltEnableTS Copy	Enables timestamp columns in insert statements.

Bind	Specifies a statement for binding a rule to a column.
	
	Example (ASE 15):
	
	`[%R%?[exec ]][execute ]sp_bindrule [%R%?['[%QUALIFIER%]%RULE%'][[%QUALIFIER%]%RULE%]:['[%QUALIFIER %]%RULE%']], '%TABLE%.%COLUMN%'`

CheckNull	Specifies whether a column can be null.
Column Comment	Specifies a statement for adding a comment to a column. Example: `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%. Example: `[constraint %CONSTNAME%] check (%CONSTRAINT%)`

Item	Description

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.
	If `DropColnChck` is empty, PowerDesigner copies data to a temporary table before recreating the table with the new constraints.
	Example (SQL Anywhere 10):
	`alter table [%QUALIFIER%]%TABLE%`
	` drop constraint %CONSTNAME%`

DropColnComp	Specifies a statement for dropping a column computed expression in an alter table statement.
	Example (SQL Anywhere 10):
	`alter table [%QUALIFIER%]%TABLE%`
	` alter %COLUMN% drop compute`

DropDefault Con-straint	Specifies a statement for dropping a constraint linked to a column defined with a default value
	Example (SQL Server 2005):
	`[%ExtDeftConstName%?alter table [%QUALIFIER%]%TA-`
	`BLE%`
	` drop constraint %ExtDeftConstName%]`

EnableBindRule	Specifies whether business rules may be bound to columns for check param-eters. The following settings are available: • Yes - The Create and Bind entry of Rule are generated • No - The check is generated inside the column Add order
Enable Computed-Coln	Specifies whether computed columns are permitted.

Item	Description

EnableDefault	Specifies whether predefined default values are permitted. The following settings are available:
	• 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
	• No - The default value can not be generated, and the Default Value check box is unavailable.
	
	Example (AS IQ 12.6):
	
	EnableDefault is enabled and the default value for the column employee function `EMPFUNC` is Technical Engineer. The generated script is:
	```
	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)
	);
	```

Item	Description
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: • Yes - Enables the Identity check box in the column property sheet. • No - The Identity check box is not available. 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: • Only one identity column is defined per table • A foreign key cannot be an identity column • 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. Note that, during generation, the %IDENTITY% variable contains the value "identity" but you can easily change it, if needed, using the following syntax : `[%IDENTITY%?new identity keyword]`
EnableNotNull WithDflt	Specifies whether default values are assigned to columns containing Null values. The following settings are available: • 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. • 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.
	
	If `AddColnChck` is empty, PowerDesigner copies data to a temporary table before recreating the table with the new constraints.
	
	Example (AS IQ 12.6):
	
	`alter table [%QUALIFIER%]%TABLE%`
	` modify %COLUMN% check (%.A:CONSTRAINT%)`
	
	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.
	
	`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.
	
	Example (ASA 6):
	
	`alter table [%QUALIFIER%]%TABLE%`
	` 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.
	
	If `ModifyColnDflt` is empty, PowerDesigner copies data to a temporary table before recreating the table with the new constraints.
	
	Example (ASE 15):
	
	`alter table [%QUALIFIER%]%TABLE%`
	` replace %COLUMN% default %DEFAULT%`
	
	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.
	
	Example (Oracle 10g):
	
	`alter table [%QUALIFIER%]%TABLE%`
	` modify %COLUMN% %MAND%`

Item	Description

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.
	
	Example (SQL Anywhere 10):
	
	`alter table [%QUALIFIER%]%TABLE%`
	` 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 171.

Rename	Specifies a statement for renaming a column within an `alter table` statement.
	
	Example (Oracle 10g):
	
	`alter table [%QUALIFIER%]%TABLE%`
	` rename column %OLDCOLN% to %NEWCOLN%`

SqlChckQuery	Specifies a SQL query to reverse engineer column check parameters. The result must conform to proper SQL syntax.
	
	Example (SQL Anywhere 10):
	
	`{OWNER, TABLE, COLUMN, CONSTNAME, CONSTRAINT}`
	`select u.user_name, t.table_name,`
	` c.column_name, k.constraint_name,`
	` case(lcase(left(h.check_defn, 5))) when 'check'`
	`then substring(h.check_defn, 6) else h.check_defn`
	`end`
	`from sys.sysconstraint k`
	` join sys.syscheck h on (h.check_id = k.con-`
	`straint_id)`
	` join sys.systab t on (t.object_id = k.table_ob-`
	`ject_id)`
	` join sys.sysuserperms u on (u.user_id = t.creator)`
	` join sys.syscolumn c on (c.object_id = k.ref_ob-`
	`ject_id)`
	`where k.constraint_type = 'C'`
	`[  and u.user_name=%.q:OWNER%]`
	`[  and t.table_name=%.q:TABLE%]`
	`[  and c.column_name=%.q:COLUMN%]`
	`order by 1, 2, 3, 4`

Item	Description

SqlStatistics	Specifies a SQL query to reverse engineer column and table statistics.
	Example (ASE 15):
	`[%ISLONGDTTP%?{ AverageLength }`
	`select [%ISLONGDTTP%?[%ISSTRDTTP%?`
	`avg(char_length(%COLUMN%)):avg(datalength(%COLUMN`
	`%))]:null] as average_length`
	`from [%QUALIFIER%]%TABLE%`
	`:{ NullValuesRate, DistinctValues, AverageLength }`
	`select`
	`[%ISMAND%?null:(count(*) - count(%COLUMN%)) * 100 /`
	`count(*)]  as null_values,`
	`[%ISMAND%?null:count(distinct %COLUMN%)]  as dis-`
	`tinct_values,`
	`[%ISVARDTTP%?[%ISSTRDTTP%?avg(char_length(%COLUMN`
	`%)):avg(datalength(%COLUMN%))]:null] as aver-`
	`age_length`
	`from [%QUALIFIER%]%TABLE%]`

Unbind	Specifies a statement for unbinding a rule to a column.
	Example (ASE 15):
	`[%R%?[exec ]][execute ]sp_unbindrule '%TABLE%.%COL-`
	`UMN%'`

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: • Add • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner • Header, Footer • Maxlen • ModifiableAttributes • Options, DefOptions • ReversedQueries • ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery For a description of each of these common items, see *Common object items* on page 156.

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.
	
	Example (ASE 15):
	
	`%COLUMN%[ %ASC%]`
	
	%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: • Yes – Indexes are generated before keys. • No – Indexes are generated after keys.

DefIndexType	Specifies the default type of an index.
	
	Example (DB2):
	
	`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: • 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 • No – Index sorting is not supported. Example (SQL Anywhere 10): 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: `create index IX_TASK on TASK (PRONUM asc, TSKNAME desc);`
EnableCluster	Enables the creation of cluster indexes. The following settings are available: • Yes - The Cluster check box is enabled in index property sheets. • No – Cluster indexes are not supported.

Item	Description
EnableFunction	Enables the creation of function-based indexes. The following settings are available: • Yes - You can define expressions for indexes. • No – Function-based indexes are not supported.

IndexComment	Specifies a Statement for adding a comment to an index.
	Example (SQL Anywhere 10):
	`comment on index [%QUALIFIER%]%TABLE%.%INDEX% is %.q:COMMENT%`

IndexType	Specifies a list of available index types.
	Example (IQ 12.6):
	`CMP`
	`HG`
	`HNG`
	`LF`
	`WD`
	`DATE`
	`TIME`
	`DTTM`

MandIndexType	Specifies whether the index type is mandatory for indexes. The following settings are available: • Yes – The index type is mandatory. • 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.

Item	Description

SqlSysIndex Query	Specifies a SQL query used to list system indexes created by the database. These indexes are excluded during reverse engineering.
	
	Example (AS IQ 12.6):
	
	`{OWNER, TABLE, INDEX, INDEXTYPE}`
	`select u.user_name, t.table_name, i.index_name, i.index_type`
	`from sysindex i, systable t, sysuserperms u`
	`where t.table_id = i.table_id`
	`and u.user_id = t.creator`
	`and i.index_owner != 'USER'`
	`[and u.user_name=%.q:OWNER%]`
	`[and t.table_name=%.q:TABLE%]`
	`union`
	`select u.user_name, t.table_name, i.index_name, i.index_type`
	`from sysindex i, systable t, sysuserperms u`
	`where t.table_id = i.table_id`
	`and u.user_id = t.creator`
	`and i.index_type = 'SA'`
	`[and u.user_name=%.q:OWNER%]`
	`[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: • Yes – Index names must be unique within the global scope of the database. • 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: • Add • ConstName • Create, Drop • Enable • Options, DefOptions • ReversedQueries For a description of each of these common items, see *Common object items* on page 156.

Item	Description
EnableCluster	Specifies whether clustered constraints are permitted on primary keys. • Yes - Clustered constraints are permitted. • 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: • 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. • 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:
	
	• Yes - The `Sp_primarykey` statement is used to generate primary keys.
	• No - Primary keys are generated separately in an `alter table` statement.
	
	Example (ASE 15):
	
	If UseSpPrimKey is enabled the Add entry for Pkey contains:
	
	`UseSpPrimKey = YES`
	`Add entry of`
	
	`[%USE_SP_PKEY%?[execute] sp_primarykey %TABLE%,`
	`%PKEYCOLUMNS%`
	`:alter table [%QUALIFIER%]%TABLE%`
	` add [constraint %CONSTNAME%] primary key [%IsClus-`
	`tered%] (%PKEYCOLUMNS%)`
	`  [%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: • Add • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • ConstName • Create, Drop • Enable • MaxConstLen • ModifiableAttributes • Options, DefOptions • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery For a description of each of these common items, see *Common object items* on page 156.
AKeyComment	Specifies a statement for adding an alternate key comment.
AllowNullable Coln	Specifies whether non-mandatory columns are permitted. The following settings are available: • Yes - Non mandatory columns are permitted. • No - Non mandatory column are not permitted.
EnableCluster	Specifies whether clustered constraints are permitted on alternate keys. • Yes - Clustered constraints are permitted. • No - Clustered constraints are not permitted.

SqlAkeyIndex	Specifies a reverse-engineering query for obtaining the alternate key indexes of a table by live connection.
	Example (SQL Anywhere 10):
	`select distinct  i.index_name`
	`from sys.sysuserperms u`
	`  join sys.systable t on`
	`  (t.creator=u.user_id)`
	`  join sys.sysindex i on`
	`  (i.table_id=t.table_id)`
	`where i."unique" not in ('Y', 'N')`
	`[  and t.table_name = %.q:TABLE%]`
	`[  and u.user_name = %.q:SCHEMA%]`

Item	Description
UniqConstAuto Index	Determines whether a `Create Index` statement is generated for every key statement. The following settings are available: • 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. • 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: • Add • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • ConstName • Create, Drop • Enable • MaxConstLen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery For a description of each of these common items, see *Common object items* on page 156.
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. Example: `CHECK ON COMMIT`

Item	Description
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: • RESTRICT • CASCADE • SET NULL • 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: • RESTRICT • CASCADE • SET NULL • 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. 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. 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. Example (Access 2000): `P=%PK% F=%FK%`
EnableChange Join-Order	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: • 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. • No - The auto arrange join order property is unavailable.

Item	Description
EnableCluster	Specifies whether clustered constraints are permitted on foreign keys. • Yes - Clustered constraints are permitted. • No - Clustered constraints are not permitted.
EnablefKey Name	Specifies the foreign key role allowed during database generation. The following settings are available: • Yes - The code of the reference is used as role for the foreign key. • 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: • 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. • 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.
	Example (Oracle 10g):
	`{CKEYCOLUMN, FKEYCOLUMN}`
	`[%ISODBCUSER%?select`
	` p.column_name, f.column_name`
	`from sys.user_cons_columns f,`
	` sys.all_cons_columns p`
	`where f.position = p.position`
	`  and f.table_name=%.q:TABLE%`
	`[ and p.owner=%.q:POWNER%]`
	`  and p.table_name=%.q:PARENT%`
	`  and f.constraint_name=%.q:FKCONSTRAINT%`
	`  and p.constraint_name=%.q:PKCONSTRAINT%`
	`order by f.position`
	`:select p.column_name, f.column_name`
	`from sys.all_cons_columns f,`
	` sys.all_cons_columns p`
	`where f.position = p.position`
	`  and f.owner=%.q:SCHEMA%`
	`  and f.table_name=%.q:TABLE%`
	`[ and p.owner=%.q:POWNER%]`
	`  and p.table_name=%.q:PARENT%`
	`  and f.constraint_name=%.q:FKCONSTRAINT%`
	`  and p.constraint_name=%.q:PKCONSTRAINT%`
	`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: • Yes - The `Sp_foreignkey` statement is used to create references. • No - Foreign keys are generated separately in an `alter table` statement using the Create order of reference. See also `UseSpPrimKey` (*Pkey* on page 175).

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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableSynonym • Header, Footer • ModifiableAttributes • Options • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery For a description of each of these common items, see *Common object items* on page 156.

EnableIndex	Specifies a list of view types for which a view index is available.
	Example (Oracle 10g):
	`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).
	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.
	Example (Oracle 10g):
	`SELECT schema_url FROM dba_xml_schemas`

Item	Description
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. The XML type is to be used with the SqlListSchema item.

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.
	
	Example (SQL Anywhere 10):
	
	If ViewCheck is set to with check option, the generated script is:
	
	`create view TEST as`
	`select CUSTOMER.CUSNUM, CUSTOMER.CUSNAME, CUSTOM-`
	`ER.CUSTEL`
	`from CUSTOMER`
	`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.
	
	Example (Oracle 10g):
	
	`[%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.
	
	Example (Oracle 10g):
	
	`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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • ModifiableAttributes • Options, DefOptions • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery For a description of each of these common items, see *Common object items* on page 156.
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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • ModifiableAttributes • Options, DefOptions • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery For a description of each of these common items, see *Common object items* on page 156.
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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • ModifiableAttributes • Options, DefOptions • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery For a description of each of these common items, see *Common object items* on page 156.
BeforeCreate Data-base	Controls the order in which databases, tablespaces, and storages are generated. The following settings are available: • Yes – [default] Create Tablespace and Create Storage statements are generated before the Create Database statement. • 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 Data-bases	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.
	Example (ASE 15):
	`use %DATABASE%`
	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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery For a description of each of these common items, see *Common object items* on page 156.

Bind	Specifies the syntax for binding a business rule to a domain.
	Example (ASE 15):
	`[%R%?[exec ]][execute ]sp_bindrule [%R%?['[%QUALI-FIER%]%RULE%'][[%QUALIFIER%]%RULE%]:['[%QUALIFIER %]%RULE%']], %DOMAIN%`

EnableBindRule	Specifies whether business rules may be bound to domains for check parameters. The following settings are available: • Yes - The Create and Bind entry of Rule are generated • No - The check inside the domain Add order is generated
EnableCheck	Specifies whether check parameters are generated. 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. The following settings are available: • Yes - Check parameters are generated • No - Variables linked to check parameters are not evaluated during generation and reverse

Item	Description
EnableDefault	Specifies whether default values are generated. The following settings are available: • 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 • 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.
	Example (ASE 15):
	`[%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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • ModifiableAttributes • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlPermQuery For a description of each of these common items, see *Common object items* on page 156.
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.
	Example (Oracle 10g):
	`OBJECT`
	`TABLE`
	`VARRAY`

Item	Description
Authorizations	Specifies a list of those users able to invoke abstract data types.

CreateBody	Specifies a statement for creating an abstract data type body.
	
	Example (Oracle 10g):
	
	`create [or replace ]type body [%QUALIFIER%]%ADT%`
	`[.O:[as][is]]`
	` %ADTBODY%`
	`end;`

EnableAdtOn Coln	Specifies whether abstract data types are enabled for columns. The following settings are available: • Yes - Abstract Data Types are added to the list of column types provided they have the valid type. • 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: • Yes - Abstract Data Types are added to the list of domain types provided they have the valid type • 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.
	
	Example (SQL Anywhere 10):
	
	`install JAVA UPDATE from file %.q:FILE%`

JavaData	Specifies a list of available instantiation mechanisms for SQL Java abstract data types.

Remove	Specifies a statement for installing a Java class as an abstract data class.
	
	Example (SQL Anywhere 10):
	
	`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: • Add • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop, Modify • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlListQuery For a description of each of these common items, see *Common object items* on page 156.

AllowedADT	Specifies a list of abstract data types which can be used as data types for abstract data type attributes.
	Example (Oracle 10g):
	`OBJECT`
	`TABLE`
	`VARRAY`
	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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • Maxlen • ModifiableAttributes • Options, DefOptions • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery For a description of each of these common items, see *Common object items* on page 156.
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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery For a description of each of these common items, see *Common object items* on page 156.

Item	Description

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.
	
	The corresponding variable is %DEFAULTNAME%.
	
	Example (ASE 15):
	
	`D_%.19:COLUMN%_%.8:TABLE%`
	
	The `EMPFUNC` column of the EMPLOYEE table has a default value of `Technical Engineer.` The `D_EMPFUNC_EMPLOYEE` column default name is created:
	
	`create default D_EMPFUNC_EMPLOYEE`
	`as 'Technical Engineer'`
	`go`
	`execute sp_bindefault D_EMPFUNC_EMPLOYEE, "EMPLOY-`
	`EE.EMPFUNC"`
	`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.
	
	The corresponding variable is %RULE%.
	
	Example (ASE 15):
	
	`R_%.19:COLUMN%_%.8:TABLE%`
	
	The TEASPE column of the Team table has a list of values - Industry, Military, Nuclear, Bank, Marketing - defined in its check parameters:
	
	The R_TEASPE_TEAM rule name is created and associated with the TEA-SPE column:
	
	`create rule R_TEASPE_TEAM`
	`as @TEASPE in ('Industry','Military','Nu-`
	`clear','Bank','Marketing')`
	`go`
	`execute sp_bindrule R_TEASPE_TEAM, "TEAM.TEASPE"`
	`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.
	
	The corresponding variable is %DEFAULTNAME%.
	
	Example (ASE 15):
	
	`D_%.28:DOMAIN%`
	
	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:
	
	`create default D_FunctionList`
	`as 'Technical Engineer'`
	`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.
	
	The corresponding variable is %RULE%.
	
	Example (ASE 15):
	
	`R_%.28:DOMAIN%`
	
	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 `Udd-tRuleName`:
	
	`create rule R_Domain_speciality`
	`as (@Domain_speciality in ('Industry','Mili-`
	`tary','Nuclear','Bank','Marketing'))`
	`go`
	`execute sp_bindrule R_Domain_speciality, T_Do-`
	`main_speciality`
	`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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner, EnableSynonym • Maxlen • ModifiableAttributes • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlPermQuery For a description of each of these common items, see *Common object items* on page 156.

CreateFunc	Specifies the statement for creating a function.
	Example (SQL Anywhere 10):
	`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.
	Example (SQL Anywhere 10):
	`create function [%QUALIFIER%]%FUNC% (<arg> <type>)`
	`RETURNS <type>`
	`begin`
	`end`

CustomProc	Specifies the statement for creating a stored procedure.
	Example (SQL Anywhere 10):
	`create procedure [%QUALIFIER%]%PROC% (IN <arg>`
	`<type>)`
	`begin`
	`end`

Item	Description

DropFunc	Specifies the statement for dropping a function.
	Example (SQL Anywhere 10):
	`if exists(select 1 from sys.sysprocedure where proc_name = %.q:FUNC%[ and user_name(creator) = %.q:OWNER%]) then`
	` drop function [%QUALIFIER%]%FUNC%`
	`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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery For a description of each of these common items, see *Common object items* on page 156.

DefaultTrigger Name	Specifies a template to define default trigger names.
	Example (SQL Anywhere 10):
	`%TEMPLATE%_%.L:TABLE%`

EnableMulti Trigger	Enables the use of multiple triggers per type.

Item	Description

Event	Specifies a list of trigger event attributes to populate the Event list on the Definition tab of Trigger property sheets.
	
	Example:
	
	`Delete`
	`Insert`
	`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.
	
	Example:
	
	`Before`
	`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: • Yes – Trigger names must be unique within the global scope of the database. • 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.
	
	Enables the use of the User-defined radio button on the Error Messages tab of the Trigger Rebuild dialog box (see *Data Modeling > Building Data Models > Triggers and Procedures > Generating Triggers and Procedures > Creating and generating user-defined error messages*).
	
	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.
	
	Example (ASE 15):
	
	`begin`
	` select @errno  = %ERRNO%,`
	`    @errmsg = %MSGTXT%`
	` from %MSGTAB%`
	` where  %MSGNO% = %ERRNO%`
	` goto error`
	`end`
	
	Where:
	
	• %ERRNO% - error number parameter to the .ERROR macro
	• %ERRMSG% - error message text parameter to the .ERROR macro
	• %MSGTAB% - name of the message table
	• %MSGNO% - name of the column that stores the error message number
	• %MSGTXT% - name of the column that stores the error message text
	
	See also UseErrorMsgText.

UseErrorMsg Text	Specifies a macro for accessing trigger error messages from the trigger template definition.
	
	Enables the use of the Standard radio button on the Error Messages tab of the Trigger Rebuild dialog box.
	
	The error number and message defined in the template definition are used.
	
	Example (ASE 15):
	
	`begin`
	` select @errno  = %ERRNO%,`
	`    @errmsg = %MSGTXT%`
	` goto error`
	`end`
	
	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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Alter, AlterStatementList, AlterDBIgnored • Enable, EnableOwner • Header, Footer • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery For a description of each of these common items, see *Common object items* on page 156.
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.
	Example:
	`Before`
	`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: • Add • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner • Header, Footer • Maxlen • ModifiableAttributes • Options, DefOptions • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery For a description of each of these common items, see *Common object items* on page 156.
AddJoin	Specifies the SQL statement used to define joins for join indexes. Example: `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: • Enable • ReversedQueries • SqlListQuery For a description of each of these common items, see *Common object items* on page 156.

Item	Description
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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner, EnableSynonym • Maxlen • ModifiableAttributes • Options, DefOptions • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlPermQuery For a description of each of these common items, see *Common object items* on page 156.

Rename	Specifies the command for renaming a sequence.
	Example (Oracle 10g):
	`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: • Create, Drop • Enable, EnableSynonym • Maxlen • ReversedQueries • SqlAttrQuery, SqlListQuery For a description of each of these common items, see *Common object items* on page 156.
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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlPermQuery For a description of each of these common items, see *Common object items* on page 156.

Bind	Specifies a command for adding a user to a group.
	
	Example (SQL Anywhere 10):
	
	`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.
	
	Example (ASE 15):
	
	`{GROUP ID, MEMBER}`
	`select g.name, u.name`
	`from`
	` [%CATALOG%.]dbo.sysusers u, [%CATALOG%.]dbo.sysus-`
	`ers g`
	`where`
	` u.suid > 0 and`
	` u.gid = g.gid and`
	` g.gid = g.uid`
	`order by 1, 2`

Unbind	Specifies a command for removing a user from a group.
	
	Example (SQL Anywhere 10):
	
	`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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlPermQuery For a description of each of these common items, see *Common object items* on page 156.

Bind	Specifies a command for adding a role to a user or to another role.
	
	Example (ASE 15):
	
	`grant role %ROLE% to %USER%`

Item	Description

SqlListChildren Query	Specifies a SQL query for listing the members of a group.
	Example (ASE 15):
	`{ ROLE ID, MEMBER }`
	`SELECT r.name, u.name`
	`FROM`
	` master.dbo.sysloginroles l,`
	` [%CATALOG%.]dbo.sysroles s,`
	` [%CATALOG%.]dbo.sysusers u,`
	` [%CATALOG%.]dbo.sysusers r`
	`where`
	` l.suid = u.suid`
	` and s.id   =l.srid`
	` and r.uid = s.lrid`

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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableSynonym • Maxlen • ModifiableAttributes • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlPermQuery For a description of each of these common items, see *Common object items* on page 156.
Authorizations	Specifies a list of those users able to invoke database packages.

Item	Description

CreateBody	Specifies a template for defining the body of the database package. This statement is used in the extension statement AfterCreate.
	
	Example (Oracle 10g):
	
	`create [or replace ]package body [%QUALIFIER%]`
	`%DBPACKAGE% [.O:[as][is]][%IsPragma% ? pragma seri-`
	`ally_reusable]`
	` %DBPACKAGEBODY%`
	`[begin`
	` %DBPACKAGEINIT%`
	`]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: • Add • ReversedQueries For a description of each of these common items, see *Common object items* on page 156.

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.
	
	Example (Oracle 10g):
	
	`begin`
	`end`

Item	Description

ParameterTypes	[database package procedures and cursors only] Specifies the available types for procedures or cursors.
	
	Example (Oracle 10g: procedure):
	
	`in`
	`in nocopy`
	`in out`
	`in out nocopy`
	`out`
	`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: • Add • ReversedQueries For a description of each of these common items, see *Common object items* on page 156.

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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • ModifiableAttributes • ReversedQueries, ReversedStatements For a description of each of these common items, see *Common object items* on page 156.

Item	Description

GrantOption	Specifies the grant option for a privileges statement.
	
	Example (Oracle 10g):
	
	`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.
	
	Example (ASE 15):
	
	`CREATE DATABASE`
	`CREATE DEFAULT`
	`CREATE PROCEDURE`
	`CREATE TRIGGER`
	`CREATE RULE`
	`CREATE TABLE`
	`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: • Create, Drop • Enable • ReversedQueries • SqlListQuery For a description of each of these common items, see *Common object items* on page 156.

GrantOption	Specifies the grant option for a permissions statement.
	
	Example (ASE 15):
	
	`with grant option`

RevokeInherited	Allows you to revoke inherited permissions from groups and roles.

RevokeOption	Specifies the revoke option for a permissions statement.
	
	Example (ASE 15):
	
	`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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery For a description of each of these common items, see *Common object items* on page 156.

Bind	Specifies the command for binding a default object to a domain or a column.
	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:
	`create table CUSTOMER (`
	`  ADDRESS  char(10)  null`
	`)`
	`sp_bindefault CITYDFLT, 'CUSTOMER.ADDRESS'`
	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:
	`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.
	Example (ASE 15):
	`[%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: • AfterCreate, AfterDrop, AfterModify • Alter • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner • Header, Footer • MaxConstLen (web operations only) • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery For a description of each of these common items, see *Common object items* on page 156.
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.
	Example (DB2 UDB 8.x CS):
	`query`
	`update`
	`storeXML`
	`retrieveXML`
	`call`

ServiceTypeList	[web service only] Specifies a list of web service types.
	Example (SQL Anywhere 10):
	`RAW`
	`HTML`
	`XML`
	`DISH`

UniqName	Specifies whether web service operation names must be unique in the database.

Item	Description
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: • Add • Enable For a description of each of these common items, see *Common object items* on page 156.
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: • AfterCreate, AfterDrop, AfterModify • Alter • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • Header, Footer • Maxlen • ReversedQueries • SqlAttrQuery, SqlListQuery For a description of each of these common items, see *Common object items* on page 156.

AddAttr Hierarchy	Specifies the syntax for defining a list of hierarchy attributes.
	Example (Oracle 10g):
	`child of %DIMNATTRHIER%`

AddAttribute	Specifies the syntax for defining an attribute.
	Example (Oracle 10g):
	`attribute %DIMNATTR% determines [.O:[(%DIMNDEPCOLN-LIST%)][%DIMNDEPCOLN%]]`

AddHierarchy	Specifies the syntax for defining a dimension hierarchy.
	Example (Oracle 10g):
	`hierarchy %DIMNHIER% (`
	`%DIMNATTRHIERFIRST% %DIMNATTRHIERLIST%)`

AddJoin Hierarchy	Specifies the syntax for defining a list of joins for hierarchy attributes. Example (Oracle 10g): join key [.O:[(%DIMNKEYLIST%)][%DIMNKEYLIST%]] references %DIMNPARENTLEVEL%

Item	Description
AddLevel	Specifies the syntax for dimension level (attribute). Example (Oracle 10g): 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: • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • EnableSynonym • Header, Footer • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery For a description of each of these common items, see *Common object items* on page 156.
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 provides mappings to allow PowerDesigner to handle DBMS-specific data types correctly.

The following variables are used in many of the entries:

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

Item	Description
AmcdAmcdType	Lists mappings to convert from specialized data types (such as `XML`, `IVL`, `MEDIA`, etc) to standard PowerDesigner data types. These mappings are used to help conversion from 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 to convert from PowerDesigner (**Internal**) data types to DBMS (**Physical Model**) data types. These mappings are used during CDM to PDM generation and with the **Change Current DBMS** command. Examples (ASE 15): • The PowerDesigner `A%n` datatype is converted to a `char(%n)` for ASE 15. • The PowerDesigner `VA%n` datatype is converted to a `varchar(%n)` for ASE 15.
PhysDataType	Lists mappings to convert from DBMS (**Physical Model**) data types to PowerDesigner (**Internal**) data types. These mappings are used during PDM to CDM generation and with the **Change Current DBMS** command. Examples (ASE 15): • The ASE 15 `sysname` datatype is converted to a `VA30` for PowerDesigner. • The ASE 15 `integer` datatype is converted to a `I` for PowerDesigner.
PhysDttpSize	Lists the storage sizes of DBMS data types. These values are used when estimating the size of a database. Examples (ASE 15): • The ASE 15 `smallmoney` requires 8 bytes of space. • The ASE 15 `smalldatetime` requires 4 bytes of space.
OdbcPhysDataType	Lists mappings to convert from live database (**ODBC**) data types to DBMS (**Physical Model**) data types during database reverse engineering. These mappings are used when data types are stored differently in the database (often due to the inclusion of a default size) than in the DBMS notation. Examples (ASE 15): • A `float(8)` in an ASE 15 database is reversed as a `float`. • A `decimal(30,6)` in an ASE 15 database is reversed as a `decimal`.

Item	Description
PhysOdbcData Type	Lists mappings of DBMS (**Physical Model**) data types to database (**ODBC**) data types for use when updating and reverse engineering a database. These mappings are used when data types that are functionally equivalent but different to those specified in the PDM are found in an existing database to avoid the display of unnecessary and irrelevant differences in the Merge dialog. Examples (ASE 15): • A `unichar` is treated as equivalent to a `unichar(1)` in an ASE 15 database. • A `float(1)` is treated as equivalent to a `float(4)` in an ASE 15 database.
PhysLogADT Type	Lists mappings to convert from DBMS (**Physical Model**) abstract data types to PowerDesigner (**Internal**) abstract data types. These mappings are used to populate the **Type** field and display the appropriate properties in abstract data type property sheets and with the **Change Current DBMS** command. Examples (Oracle 11g): • The Oracle 11g `VARRAY` abstract data type is converted to an `Array` for PowerDesigner. • The Oracle 11g `SQLJ_OBJECT` datatype is converted to a `JavaObject` for PowerDesigner.
LogPhysADT Type	Lists mappings to convert from PowerDesigner (**Internal**) abstract data types to DBMS (**Physical Model**) abstract data types. These mappings are used with the **Change Current DBMS** command. Examples (Oracle 11g): • The PowerDesigner `List` abstract data type is converted to a `TABLE` for Oracle 11g. • The PowerDesigner `Object` abstract data type is converted to an `OBJECT` for Oracle 11g.
AllowedADT	Lists the abstract data types that may be used as types for columns and domains in the DBMS. Example (ASE 15): • JAVA

Item	Description
HostDataType	Lists mappings to convert from DBMS data types (**Physical Model**) to data types permitted as procedure parameters (**Trigger**). These mappings are used to populate the **Data type** field in ADT procedure parameter property sheets Examples (Oracle 11g): • The Oracle 11g `DEC` data type is converted to a `number`. • The Oracle 11g `SMALLINT` datatype is converted to an `integer`.

Profile Category (DBMS)

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, and add new objects by creating and stereotyping extended objects and sub-objects.

You can add extensions in either:

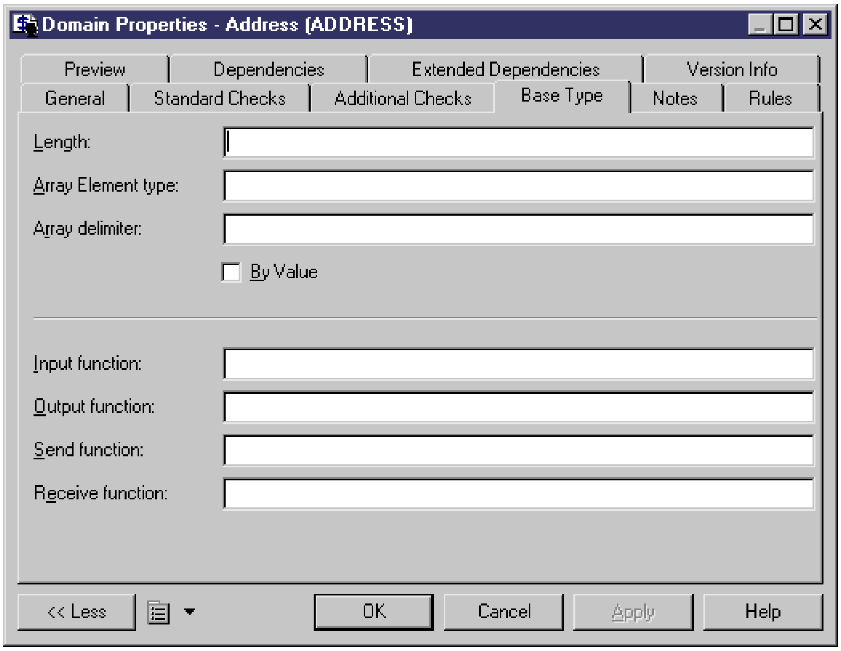
- your DBMS definition file - you should save a backup of this file before editing it.
- a separate extension file - which you attach to your model.

For detailed information about working with profiles, including adding extended attributes and objects, see *Chapter 2, Extension Files* on page 21.

Using Extended Attributes During Generation

Extended attributes can be taken into account during generation. Each extended attribute value can be used as a variable that can be referenced in the scripts defined in the Script category.

Some DBMSs 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

For 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.

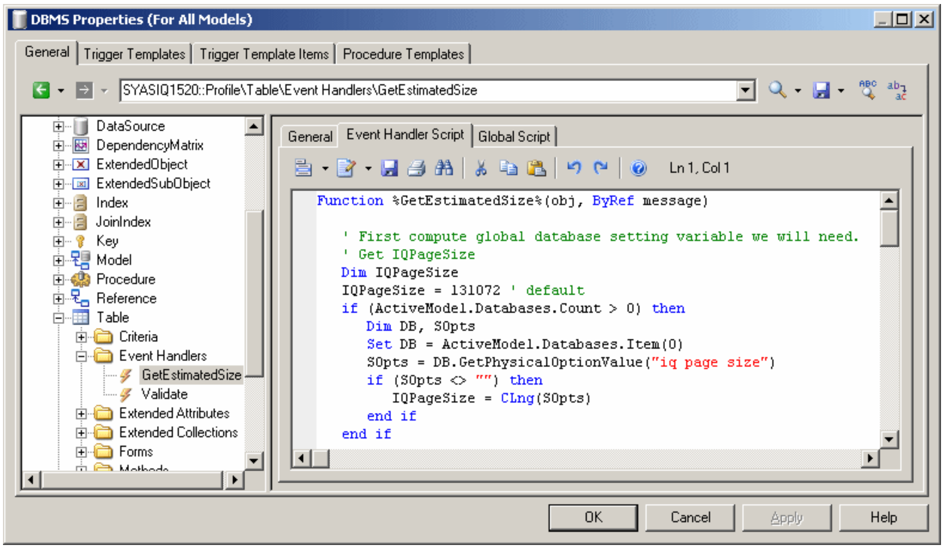
In the `SqlListQuery` item:

```
{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
```

Modifying the Estimate Database Size Mechanism

By default, the Estimate Database Size mechanism uses standard algorithms to calculate the sizes of tablespaces, tables, columns, and indexes and adds them together to provide an indication of the size that the database will require. You can override the algorithm for one or more of these types of objects or include additional objects in the calculation by adding the `GetEstimatedSize` event handler to the appropriate object in the `Profile` category and entering a script to calculate its size.

1. Select **Database > Edit Current DBMS** to open the DBMS definition file, and expand the profile category.
2. Right-click the metaclass for which you want to provide a script to calculate the object size, select **New > Event Handler** to open a selection dialog, select the `GetEstimatedSize` event handler, and then click **OK** to add it under the metaclass.
3. Click the **Event Handler Script** tab in the right pane and enter appropriate code to calculate the size of your chosen object.



```
Function %GetEstimatedSize%(obj, ByRef message)

   ' First compute global database setting variable we will need.
   ' Get IQPageSize
   Dim IQPageSize
   IQPageSize = 131072 ' default
   if (ActiveModel.Databases.Count > 0) then
      Dim DB, SOpts
      Set DB = ActiveModel.Databases.Item(0)
      SOpts = DB.GetPhysicalOptionValue("iq page size")
      if (SOpts <> "") then
         IQPageSize = CLng(SOpts)
      end if
   end if
```

In the following example, we look at extracts of a `GetEstimatedSize` event handler defined on the `Table` metaclass to estimate the size of the database by calculating the size of each table as the total size of all its columns plus the total size of all its indexes.

Note: For examples of the `GetEstimatedSize` event handler in use on the Table and other metaclasses, see the Sybase IQ v15.2 and HP Neoview R2.4 DBMS definition files.

In this first extract from the script, the `GetEstimatedSize` function opens and the size of each table is obtained by looping through the size of each of its columns. The actual work of calculating the column size is done by the line:

```
ColSize = C.GetEstimatedSize(message, false)
```

, which calls the `GetEstimatedSize` event handler on the `Column` metaclass (see *Calling the GetEstimatedSize Event Handler on Another Metaclass* on page 217):

```
Function %GetEstimatedSize%(obj, ByRef message)

' First compute global database setting variable we will need.

' Get table size and keep column size for future use
  Dim ColSizes, TblSize, ColSize, C
  Set ColSizes = CreateObject("Scripting.Dictionary")

  TblSize = 0 ' May be changed to take into account table
definition initial size.

 for each C in obj.Columns

    ' Start browsing table columns and use event handler defined
on column metaclass (if it exists).
      ColSize = C.GetEstimatedSize(message, false)
```

```
        ' Store column size in the map for future use in indexes.
      ColSizes.Add C, ColSize

        ' Increase the table global size.
      TblSize = TblSize + ColSize
   next
   Dim RawDataSize
   RawDataSize = BlockSize * int(obj.Number * TblSize / BlockSize)
    ' At this point, the RawDataSize is the size of table in
database.
```

Next the size of the table indexes is calculated directly in the script without making a call to an event handler on the Index metaclass, the line outputting index sizes is formatted and the size of the indexes added to the total database size:

```
' Now calculate index sizes. Set up variables to store indexes
sizes.
   Dim X, XMsg, XDataSize
   XMsg = ""
   for each X in obj.Indexes
      XDataSize = 0
      ' Browsing index columns and get their size added in
XDataSize
      For each C in X.IndexColumns
         XDataSize = XDataSize + ColSizes.Item(C.Column)
      next
      XDataSize = BlockSize * int(obj.Number * XDataSize /
BlockSize)

        ' Format the display message in order to get size
information in output and result list.
      XMsg = XMsg & CStr(XDataSize) & "|" & X.ObjectID & vbCrLf

      ' Add the index size to table size.
      RawDataSize = RawDataSize + XDataSize
   next
```

Finally the size information is formatted for output (see *Formatting the Database Size Estimation Output* on page 217). Each table is printed on a separate line in both the Output and Result List windows, and its total size including all columns and indexes is given:

```
    ' set the global message to table size and all indexes
(separate with carriage return).
   message = CStr(RawDataSize) & "||" & obj.ShortDescription &
vbCrLf & XMsg

   %GetEstimatedSize% = RawDataSize

End Function
```

Once all the tables have been processed, PowerDesigner calculates and prints the total estimated size of the database.

Calling the GetEstimatedSize Event Handler on Another Metaclass

You can call a `GetEstimatedSize` event handler defined on another metaclass to use this size in your calculation. For example, you may define `GetEstimatedSize` on the `Table` metaclass, and make a call to `GetEstimatedSize` defined on the `Column` and `Index` metaclasses to use these sizes to calculate the total size of the table.

The syntax of the function is as follows, where *message* is the name of your variable containing the results to print:

```
GetEstimatedSize(message[,true|false])
```

In general, we recommend that you use the function in the folllowing form:

```
GetEstimatedSize(message, false)
```

The use of the `false` parameter (which is the default, but which is shown here for clarity) means that we call the `GetEstimatedSize` event handler on the other metaclass, and use the default mechanism only if the event handler is not available.

Setting the parameter to true will force the use of the default mechanism for calculating the size of objects (only possible for tables, columns, and join indexes):

```
GetEstimatedSize(message, true)
```

Formatting the Database Size Estimation Output

You can format the output for your database size estimation. Sub-objects (such as columns and indexes) contained in a table are offset, and you can print additional information after the total.

The syntax for the output is as follows:

```
[object-size][:compartment]|[ObjectID][|label]
```

where:

- *object-size* - is the size of the object.
- *compartment* - is a one-digit number, which indicates that the size of the object should be excluded from the total size of the database and should be printed after the database size has been calculated. For example, you may include the size of individual tables in your calculation of the database size and print the sizes of tablespaces separately after the calculation.
- `ObjectID` - is unneccessary for objects, such as tables, but required for sub-objects if you want to print them to the Result List.
- *label* - is any appropriate identifying string, and is generally set to `ShortDescription`, which prints the type and name of the selected object.

For example, in the event handler defined on the `Table` metaclass (having calculated and stored the size of a table, the size of all the columns of type LONG contained in the table, and the size of each index in the table), we create a message variable to print this information. We begin by printing a line giving the size of a table:

```
message = CStr(TableSize) & "||" & objTable.ShortDescription & vbCrLf
```

We then add a line printing the total size of all the columns of type LONG in the table:

```
message = message & CStr(LongSize) & "||Columns of type LONG" &
vbCrLf
```

We then add a line printing the size of each index in the table:

```
message = message & CStr(IndexSize) & "|" & objIndex.ObjectID &
vbCrLf
```

In the event handler defined on the `Tablespace` metaclass (having calculated and stored the size of a tablespace), we create a message variable to print this information after the database size calculation has been printed.

We begin by overriding the default introduction to this second compartment:

```
message = ":1||Tables are allocated to the following tablespaces:"
```

We then add a line printing the size of each tablespace in the table

```
message = message + CStr(tablespaceSize) & ":1||" &
objTablespace.ShortDescription
```

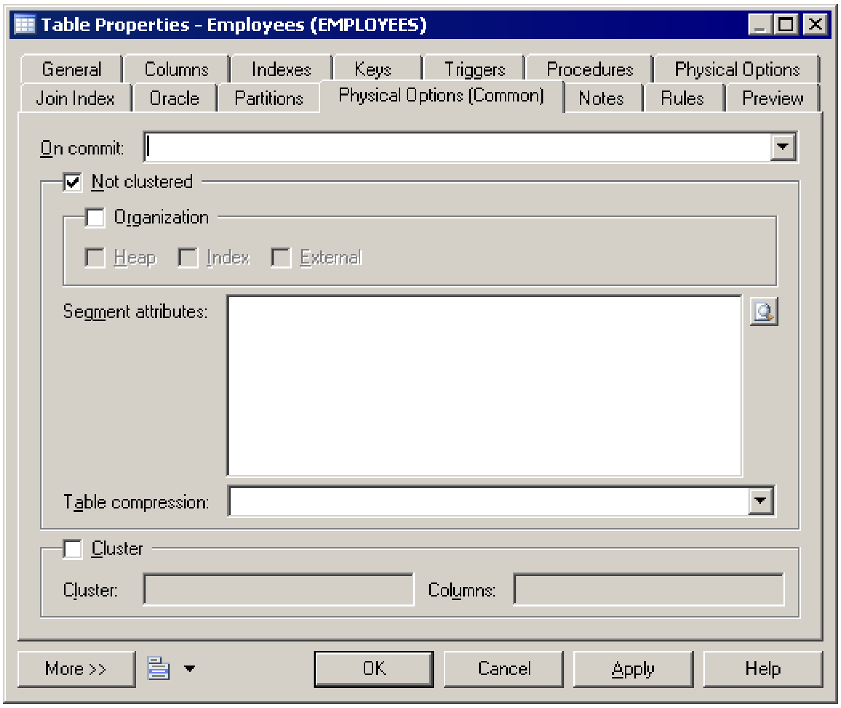
The result gives the following output:

```
Estimate of the size of the Database "Sales"...

 Number    Estimated size      Object
-------    --------------      ------------------------------------

 10,000           6096 KB      Table 'Invoices'
                                 Columns of type LONG (35 KB)
                                 Index 'customerFKeyIndex' (976 KB)
                                 Index 'descriptionIndex' (1976 KB)

                           [...etc...]

 Tables are allocated to the following tablespaces:

           Estimated size      Object
           --------------      ------------------------------------
                  6096 KB      Tablespace 'mainStorage'

                           [...etc...]
```

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:



On commit:

[✓] Not clustered

[ ] Organization

[ ] Heap [ ] Index [ ] External

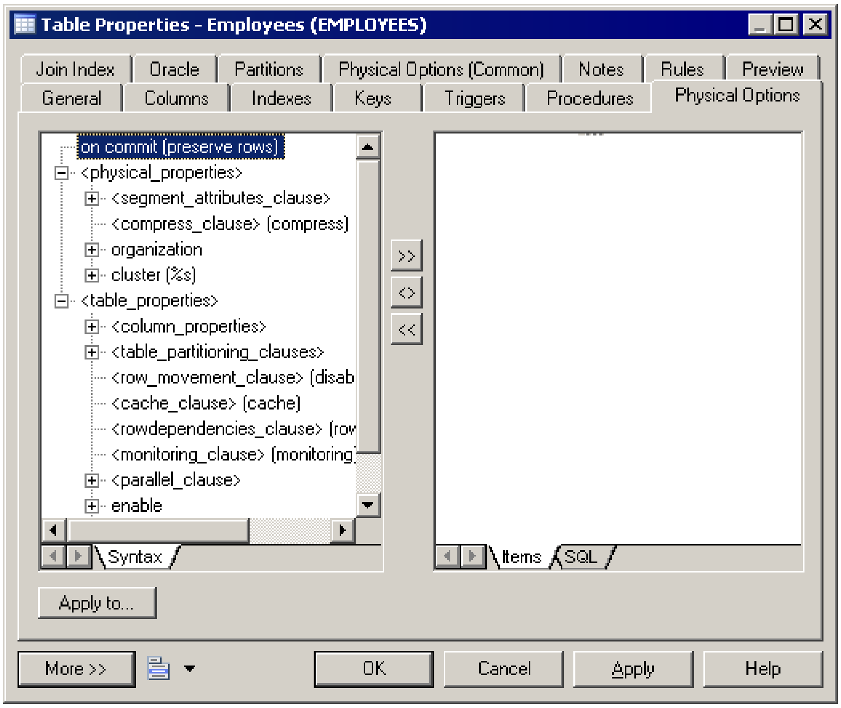
Segment attributes:

Table compression:

[ ] Cluster

Cluster: Columns:

- 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 76.

For information about setting physical options, see *Data Modeling > Building Data Models > Physical Implementation > Physical Options*.

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 156. 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 226) 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: • yes - brackets can be used to define a composite physical option • no – brackets cannot be used
separator	The following settings are available: • yes - options are separated by a comma • no [default] - options have no separator character
parenthesis	The following settings are available: • yes - the composite option is delimited by parenthesis, including all the other options, for example: with (max_row_per_page=0, ignore_dup_key) • no [default] - nothing delimits the composite option

Keyword	Value and result
nextmand	The following settings are available: • yes - the next line in the physical option is mandatory. • no - you will not be able to generate/reverse the entire composite physical option
prevmand	The following settings are available: • yes - the previous line in the physical option is mandatory • no - you will not be able to generate/reverse the entire composite physical option
chldmand	The following settings are available: • yes - at least one child line is mandatory • no – children are not mandatory

category	The following settings are available:
	• tablespace - the item is linked to a tablespace
	• storage - the item is linked to a storage
	`storage : category=storage, composite=yes, separa-`
	`tor=no, parenthesis=yes`
	`{`
	Note: In Oracle, the `storage` composite physical option 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):

list	List in which values are separated by a pipe (|)
dquoted	The following settings are available: • yes - the value is enclosed in double quotes ("" "") • no - the value is not enclosed in double quotes ("" "")
squoted	The following settings are available: • yes - the value is enclosed in single quotes (' ') • no - the value is not enclosed in single quotes (' ')
enabledbprefix	The following settings are available: • yes - the database name is used as prefix (see tablespace options in DB2 OS/390) • 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
```

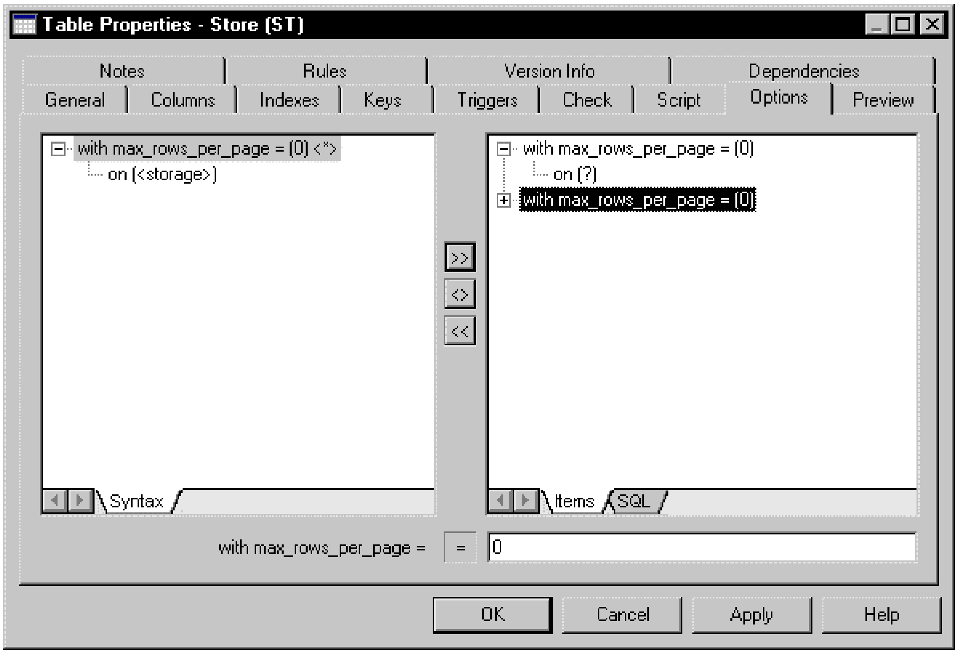
For example, the Informix fragmentation options can be repeated *n* times as follows:

```
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 <*>
```



with max_rows_per_page = (0) <*>
on (<storage>)

with max_rows_per_page = (0)
on (?)
with max_rows_per_page = (0)

with 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 and Macros

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 *Chapter 5, Customizing*

Generation with GTL on page 263 and *Chapter 1, Working with PowerDesigner Resource Files* on page 1.

Testing Variable Values with the [ ] Operators

You can use square brackets [ ] to test for the existence or value of a variable.

You can use square brackets to

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

Variable	Generation
`[%variable%]`	Tests for the existence of the variable. Generation: Generated only if *variable* exists and is not assigned `NO` or `FALSE`. Reverse: Evaluated if the parser detects a SQL statement corresponding to the variable and it is not assigned `NO` or `FALSE`.
`[%variable%? true : false]`	Tests for the existence of the variable and allows conditional output. Generation: *true* is generated if *variable* exists and is not assigned `NO` or `FALSE`. Otherwise, *false* is generated. Reverse: If the parser detects *variable* and it is not assigned `NO` or `FALSE`, *true* is reversed. Otherwise, *false* is reversed. *variable* is set to `True` or `False` as appropriate.
`[%variable%=constant? true : false]`	Tests the value of the variable and allows conditional output. Generation: If *variable* equals *constant*, *true* is generated. Otherwise, *false* is generated. Reverse: If the parser detects that *variable* equals *constant*, *true* is reversed. Otherwise, *false* is reversed.
`[.Z: [item1] [item2]...]`	Specifies that the *items* do not have a significant order. Generation: `.Z` is ignored Reverse: The *items* can be reversed in any order they are encountered.

Variable	Generation
`[.O: [`*`item1`*`] [`*`item2`*`]...]`	Specifies that the *items* are synonyms, only one of which should be output. Generation: Only the first *item* listed is generated. Reverse: The reverse parser must find one of the *items* to validate the full statement.

Examples

- `[%OPTIONS%]`

  If `%OPTIONS%` (physical options for the objects visible in the object property sheet) exists and is not assigned `NO` or `FALSE`, it is generated to the value of `%OPTIONS%`.
- `[default %DEFAULT%]`

  If the statement `default 10` is found during reverse engineering, `%DEFAULT%` is assigned the value `10`, but the statement is not mandatory and reversing continues even if it is absent. In script generation, if `%DEFAULT%` has a value of `10`, it is generated as `default 10` otherwise nothing is generated for the block.
- `[%MAND%? not null : null ]`

  If `%MAND%` is evaluated as true or contains a value other than `False` or `NO`, it is generated as `not null`. Otherwise it is generated as `null`.
- `[%DELCONST%=RESTRICT?:[on delete %DELCONST%]]`

  If `%DELCONST%` contains the value `RESTRICT`, it is generated as `on delete RESTRICT`.
- `%COLUMN% %DATATYPE%[.Z: [%NOTNULL%][%DEFAULT%]]`

  Because of the presence of the `.Z` variable, both of the following statements will be reversed correctly even though the column attributes are not in the same order:
  - `Create table abc (a integer not null default 99)`
  - `Create table abc (a integer default 99 not null)`
- `[.O:[procedure][proc]]`

  This statement will generate `procedure`. During reverse engineering, the parser will match either `procedure` or `proc` keywords.
- **Note:** A string between square brackets is always generated. For reverse engineering, placing a string between square brackets means that it is optional and its absence will not cancel the reversing of the statement.

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

  A script containing either `create` or `create or replace` will be correctly reversed because `or replace` is optional.

Formatting Variable Values

You can specify a format for variable values. For example, you can force values to lowercase or uppercase, truncate the length of values, or place values between quotes.

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.

Examples

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. Example: `PK_%.L:TABLE%` Result: `PK_customer_priority`
.U*n*	Upper-case characters + left justify variable text to fixed length where *n* is the number of characters. Example: `PK_%.U12:TABLE%` Result: `PK_CUSTOMER_PRI`
.T	Trim the leading and trailing white space from the variable. Example: `PK_%.T:TABLE%` Result: `PK_customer_priority`
.*n*	Maximum length where *n* is the number of characters. Example: `PK_%.8:TABLE%` Result: `PK_Customer`
-*n*	Pad the output with blanks to the right to display a fixed length where *n* is the number of characters. Example: `PK_%-20:TABLE%` Result: `PK_ Customer_priority`
M	Extract a portion of a variable. Example: `PK%3.4M:TABLE%` Result: `PK_CUST`

Common Variables for Objects

These variables can be used for all objects supporting these concepts.

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)

Variable	Comment
%OPTIONS%	SQL text defining physical options for Object
%OPTIONSEX%	The parsed SQL text defining physical options of the 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)
%TABLQUALIFIER%	Parent table qualifier (database prefix + owner prefix)
%TABLOWNER%	The generated code of the user owning the parent table

The following variables are available for all named objects:

Variable	Comment
%@OBJTNAME%	Name of Object
%@OBJTCODE%	Code of Object
%@OBJTLABL%	Comment of Object
%@OBJTDESC%	Description of Object

The following metadata variables are available:

Variable name	Comment
@CLSSNAME	Localized name for an object class. For example: Table, View, Column, Index
@CLSSCODE	Object class code. For example: TABL, VIEW, COLN, INDX

Variables for Tables and Views

PowerDesigner can use variables in the generation and reverse-engineering of tables and views.

The following variables are available for tables:

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
%TABLDEFN%	Complete body of Table definition. It contains definition of columns, checks and keys
%CLASS%	Abstract data type name
%CLASSOWNER%	Owner of the class object
%CLASSQUALIFIER%	Qualifier of the class object
%CLUSTERCOLUMNS%	List of columns used for a cluster
%INDXDEFN%	Table indexes definition
%TABLTYPE%	Table type

The following variables are available for views:

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
%VIEWSTYLE%	Style of view: view, snapshot, materialized view
%ISVIEW%	True is it is a view (and not a snapshot)
%USAGE%	Read-only=0, Updatable=1, Check option=2

The following variables are available for tables and views:

Variable	Comment
%XMLELEMENT%	Element contained in the XML schema
%XMLSCHEMA%	XML schema

Variables for Columns, Domains, and Constraints

PowerDesigner can use variables in the generation and reverse-engineering of columns, domains, and constraints. Parent table variables are also available.

The following variables are available for columns:

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 a primary key
%ISAKEY%	TRUE if Column is part of an alternate key
%FOREIGN%	TRUE if Column is part of a foreign key
%COMPUTE%	Compute constraint text
%PREVCOLN%	Code of the previous column in the list of columns of the table
%NEXTCOLN%	Code of the next column in the list of columns of the table
%NULLNOTNULL%	Mandatory status of a column. This variable is always used with Null-Required item, see *Working with Null Values* on page 171
%PKEYCLUSTER%	CLUSTER keyword for the primary key when it is defined on the same line
%AKEYCLUSTER%	CLUSTER keyword for the alternate key when it is defined on the same line
%AVERAGELENGTH%	Average length
%ISVARDTTP%	TRUE if the column datatype has a variable length
%ISLONGDTTP%	TRUE if the column datatype is a long datatype but not an image or a blob

Variable	Comment
%ISBLOBDTTP%	TRUE if the column datatype is an image or a blob
%ISSTRDTTP%	TRUE if the column datatype contains characters

The following variables are available for domains:

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

The following variables are available for constraints:

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

Variable	Comment
%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
%UPPER%	SQL constraint associated with upper only values
%LOWER%	SQL constraint associated with lower only values
%CASE%	SQL constraint associated with cases (upper, lower, first word capital, etc)

Variables for Keys

PowerDesigner can use variables in the generation and reverse-engineering of keys.

Variable	Comment
%COLUMNS% or %COLNLIST%	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 Indexes and Index Columns

PowerDesigner can use variables in the generation and reverse-engineering of indexes and index columns.

The following variables are available for indexes:

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)

Variable	Comment
%INDEXNAME%	Index name
%INDEXCODE%	Index code
%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

The following variables are available for index columns:

Variable	Comment
%ASC%	Contains keywords "ASC" or "DESC" depending on sort order
%ISASC%	TRUE if index column sort is ascending

Variables for References and Reference Columns

PowerDesigner can use variables in the generation and reverse-engineering of references and reference columns.

The following variables are available for references:

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.

The following variables are available for reference columns:

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

Variable	Comment
%AK%	Alternate key column code (same as PK)
%AKNAME%	Alternate key column name (same as PKNAME)
%COLTYPE%	Primary key column data type
%COLTYPENOOWNER%	Primary column owner
%DEFAULT%	Foreign key column default value
%HOSTCOLTYPE%	Primary key column data type used in procedure declaration. For example: without length

Variables for Triggers and Procedures

PowerDesigner can use variables in the generation and reverse-engineering of triggers and procedures. Parent table variables are also available.

The following variables are available for triggers:

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
%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)

Variable	Comment
%TRGDEFN%	Trigger definition
%TRGSCOPE%	Trigger scope (keywords: database, schema, all server)
%TRGSCOPEOWNER%	Trigger scope owner
%TRGSCOPEQUALIFIER%	Trigger scope owner plus dot

The following variables are available for procedures:

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)
%PROCPRMS%	List of parameters of the procedure

Variables for Rules

PowerDesigner can use variables in the generation and reverse-engineering of rules.

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

Variables for Sequences

PowerDesigner can use variables in the generation and reverse-engineering of sequences.

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

Variables for Synonyms

PowerDesigner can use variables in the generation and reverse-engineering of synonyms.

Variable	Comment
%SYNONYM%	Generated code of the synonym
%BASEOBJECT%	Base object of the synonym
%BASEOWNER%	Owner of the base object
%BASEQUALIFIER%	Qualifier of the base object
%VISIBILITY%	Private (default) or public
%SYNMTYPE%	Synonym of alias (DB2 only)
%ISPRIVATE%	True for a private synonym
%ISPUBLIC%	True for a public synonym

Variables for Tablespaces and Storages

PowerDesigner can use variables in the generation and reverse-engineering of tablespaces and storages.

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

Variables for Abstract Data Types

PowerDesigner can use variables in the generation and reverse-engineering of abstract data types and their child objects.

The following variables are available for abstract data types:

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

Variable	Comment
%ISSTRUCT%	TRUE if Abstract data type is of type structure
%ISOBJECT%	TRUE if Abstract data type is of type object
%ISJAVAOBJECT%	TRUE if Abstract data type is of type JAVA object
%ISJAVA%	TRUE if Abstract data type is of type JAVA class
%ADTDEF%	Contains Definition of Abstract data type
%ADTBODY%	Abstract data type body
%SUPERADT%	Abstract data type supertype
%ADTNOTFINAL%	Abstract data type final
%ADTABSTRACT%	Abstract data type instantiable
%ADTHEADER%	Abstract data type body with ODBC
%ADTTEXT%	Abstract data type spec with ODBC
%SUPERQUALIFIER%	Abstract data type supertype qualifier
%SUPEROWNER%	Abstract data type supertype owner
%ADTAUTH%	Abstract data type authorization
%ADTJAVANAME%	Abstract data type JAVA name
%ADTJAVADATA%	Abstract data type JAVA data
%ADTATTRDEF%	Attributes part of abstract data type definition
%ADTMETHDEF%	Methods part of abstract data type definition

The following variables are available for abstract data type attributes:

Variable	Comment
%ADTATTR%	Generated code of Abstract data type attribute
%ATTRJAVANAME%	Abstract data type attribute JAVA name

The following variables are available for abstract data type procedures:

Variable	Comment
%ADTPROC%	Procedure code
%PROCTYPE%	Procedure type (constructor, order, map)
%PROCFUNC%	Procedure type (procedure, function)
%PROCDEFN%	Procedure body (begin... end)
%PROCRETURN %	Procedure return type
%PARAM%	Procedure parameters
%PROCNOTFI-NAL%	Procedure final
%PROCSTATIC%	Procedure member
%PROCAB-STRACT%	Procedure instantiable
%SUPERPROC%	Procedure super-procedure
%ISCONSTRUC-TOR%	True if the procedure is a constructor
%PROCJAVA-NAME%	Procedure JAVA name
%ISJAVAVAR%	True if procedure is mapped to a static JAVA variable
%ISSPEC%	True in specifications, undefined in body

Variables for Join Indexes (IQ)

PowerDesigner can use variables in the generation and reverse-engineering of IQ join indexes.

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)
%FACTQUALIFIER%	Qualifier for the fact table
%JIDXFACT%	Fact (base table)

Variable	Comment
%JIDXCOLN%	List of columns
%JIDXFROM%	From clause
%JIDXWHERE%	Where clause

Variables for ASE & SQL Server

PowerDesigner can use variables in the generation and reverse-engineering of objects for ASE and SQL Server.

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 Database Synchronization

PowerDesigner can use variables in the generation and reverse-engineering of objects during database synchronization.

Variable	Comment
%OLDOWNER%	Old owner name of Object. See also OWNER
%NEWOWNER%	New owner name of Object. See also OWNER
%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 Packages and Their Child Objects

PowerDesigner can use variables in the generation and reverse-engineering of database packages and their child objects.

The following variables are available for database packages:

Variable	Comment
%DBPACKAGE%	Generated code of the database package
%DBPACKAGECODE%	Initialization code at the end of the package
%DBPACKAGESPEC%	Database package specification
%DBPACKAGEBODY%	Database package body
%DBPACKAGEINIT%	Database package initialization code
%DBPACKAGEPRIV%	Database package authorization (old privilege)
%DBPACKAGEAUTH%	Database package authorization
%DBPACKAGEPUBLIC%	True for public sub-object
%DBPACKAGETEXT%	Database package body with ODBC
%DBPACKAGEHEADER%	Database package spec with ODBC

The following variables are available for database package procedures:

Variable	Comment
%DBPKPROC%	Procedure code
%DBPKPROCTYPE%	Procedure type (procedure, function)
%DBPKPROCCODE%	Procedure body (begin... end)
%DBPKPROCRETURN%	Procedure return type
%DBPKPROCPARAM%	Procedure parameters

The following variables are available for database package variables:

Variable	Comment
%DBPFVAR%	Variable code
%DBPFVARTYPE%	Variable type

Variable	Comment
%DBPFVARCONST%	Variable of constant type
%DBPFVARVALUE%	Variable default value for constant

The following variables are available for database package types:

Variable	Comment
%DBPKTYPE%	Type code
%DBPKTYPEVAR%	List of variables
%DBPKISSUBTYPE%	True if type is a subtype

The following variables are available for database package cursors:

Variable	Comment
%DBPKCURSOR%	Cursor code
%DBPKCURSORRE-TURN%	Cursor return type
%DBPKCURSORQUERY%	Cursor query
%DBPKCURSORPARAM%	Cursor parameter

The following variables are available for database package exceptions:

Variable	Comment
%DBPKEXEC%	Exception code

The following variables are available for database package parameters:

Variable	Comment
%DBPKPARM%	Parameter code
%DBPKPARMTYPE%	Parameter type
%DBPKPARMDTTP%	Parameter data type
%DBPKPARMDEFAULT%	Parameter default value

The following variables are available for database package pragmas:

Variable	Comment
%DBPKPRAGMA%	Pragma directive
%DBPKPRAGMAOBJ%	Pragma directive on object
%DBPKPRAGMAPARAM%	Pragma directive parameter

Variables for Database Security

PowerDesigner can use variables in the generation and reverse-engineering of database security objects.

Variable	Comment
%PRIVLIST%	List of privileges for a grant order
%REVPRIVLIST%	List of privileges for a revoke order
%PERMLIST%	List of permissions for a grant order
%REVPERMLIST%	List of permissions for a revoke order
%COLNPERMISSION%	Permissions on a specific list of columns
%BITMAPCOLN%	Bitmap of specific columns with permissions
%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)
%PERMISSION%	SQL grant/revoke order for a database object
%PRIVILEGE%	SQL grant/revoke order for an ID (user, group, or role)
%GRANTOPTION%	Option for grant: with grant option / with admin option
%REVOKEOPTION%	Option for revoke: with cascade
%GRANTOR%	User that grants the permission
%MEMBER%	Member of a group or member with a role
%GROUPS%	List of groups separated by the delimiter
%MEMBERS%	List of members (users or roles) of a group or role separated by the delimiter

Variable	Comment
%ROLES%	List of parent roles of a user or role
%SCHEMADEFN%	Schema definition

Variables for Defaults

PowerDesigner can use variables in the generation and reverse-engineering of defaults.

Variable	Comment
%BOUND_OBJECT%	Binded object

Variables for Web Services

PowerDesigner can use variables in the generation and reverse-engineering of sequences.

The following variables are available for web services:

Variable	Comment
%WEBSERVICENAME%	Only generated code of the web service
%WEBSERVICE%	Generated code of the web service and local path
%WEBSERVICETYPE%	Web service type
%WEBSERVICESQL%	SQL statement
%WEBSERVICELOCAL-PATH%	Local path

The following variables are available for web service operations:

Variable	Comment
%WEBOPERATION-NAME%	Only generated code of the web operation
%WEBOPERATION%	Generated code of the operation, service, and local path
%WEBOPERATIONTYPE %	We operation type
%WEBOPERATIONSQL %	SQL statement
%WEBOPERATIONPAR-AM%	Web operation parameters list

The following variables are available for web service security:

Variable	Comment
%WEBUSER%	Connection user required for web service
%WEBCNCTSECURED%	Connection secured
%WEBAUTHREQUIRED%	Authorization required

The following variables are available for web service parameters:

Variable	Comment
%WEBPARAM%	List of web parameters
%WEBPARAMNAME%	Web parameter name
%WEBPARAMTYPE%	Web parameter type
%WEBPARAMDTTP%	Web parameter data type
%WEBPARAMDEFAULT%	Web parameter default value

Variables for Dimensions

PowerDesigner can use variables in the generation and reverse-engineering of dimensions.

Variable	Comment
%DIMENSION%	Generated code of dimension
%DIMNDEF%	Dimension definition
%DIMNATTR%	Dimension attribute (level)
%DIMNOWNERTABL%	Level table owner
%DIMNTABL%	Level table
%DIMNCOLN%	Level column
%DIMNCOLNLIST%	Level columns list
%DIMNHIER%	Dimension hierarchy
%DIMNKEY%	List of child key columns
%DIMNKEYLIST%	List of child key columns
%DIMNLEVELLIST%	Level list for hierarchy
%DIMNATTRHIER%	Attribute of hierarchy

Variable	Comment
%DIMNATTRHIERFIRST%	First attribute of hierarchy
%DIMNATTRHIERLIST%	List of attributes of hierarchy
%DIMNPARENTLEVEL%	Parent level for hierarchy
%DIMNDEPATTR%	Dimension attribute with dependent
%DIMNDEPCOLN%	Dependent column
%DIMNDEPCOLNLIST%	List of dependent columns

Variables for Extended Objects

PowerDesigner can use variables in the generation and reverse-engineering of extended objects.

Variable	Comment
%EXTENDEDOBJECT%	Generated code for extended object
%EXTENDEDSUBOBJECT%	Generated code for extended sub-object
%EXTSUBOBJTPARENT%	Generated code for parent of extended sub-object
%EXTSUBOBJTPARENTOWNER%	Generated code for owner of extended sub-object
%EXTSUBOBJTPARENTQUALIFIER%	Parent object qualifier (database prefix and owner prefix)
%EXTOBJECTDEFN%	Complete body of the extended object definition. Contains definition of extended collection listed in DefinitionContent DBMS item.

Variables for Metadata

PowerDesigner can use variables in the generation and reverse-engineering of metadata.

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

Variables for Reverse Engineering

PowerDesigner can use variables during the reverse engineering of objects.

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, Triggers, and Procedures Generation

PowerDesigner can use variables in the generation of databases, triggers, and procedures.

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

Variable	Comment
%DATABASE%	Code of Database associated with Generated model
%DATASOURCE%	Name of the data source associated with the generated script
%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)

.AKCOLN Macro

Repeats a statement for each alternate key in a table

Syntax

.AKCOLN("*statement*","*prefix*","*suffix*","*last_suffix*", "condition")

Argument	Description
statement	Statement to repeat for each column
prefix	Prefix for each new line
suffix	Suffix for each new line
last suffix	Suffix for the last line
condition	Alternate key code (if condition argument is left empty the macro returns a statement for each alternate key in the table)

Example

In a trigger for the table TITLEAUTHOR, the following macro:

```
message .AKCOLN("'%COLUMN% is an alternate key column'","", "", "",
"AKEY1")
```

generates the following trigger script:

```
message 'TA_ORDER is an alternate key column',
```

Note: For columns, the macro AKCOLN only accepts the variable %COLUMN%.

.ALLCOL Macro

Repeats a statement for each column in a table

Syntax

.ALLCOL("*statement*","*prefix*","*suffix*","*last_suffix*")

Argument	Description
statement	Statement to repeat for each column
prefix	Prefix for each new line
suffix	Suffix for each new line
last suffix	Suffix for the last line

Example

In a trigger for the table AUTHOR, the following macro:

```
.ALLCOL("%COLUMN% %COLTYPE%","",",",";")
```

generates the following trigger script:

```
AU_ID char(12),
AU_LNAME varchar(40),
AU_FNAME varchar(40),
AU_BIOGRAPH long varchar,
AU_ADVANCE numeric(8,2),
AU_ADDRESS varchar(80),
CITY varchar(20),
STATE char(2),
POSTALCODE char(5),
AU_PHONE char(12);
```

.DEFINE Macro

Defines a variable and initializes its value

Syntax

.DEFINE "*variable***" "***value***"**

Argument	Description
variable	Variable name (without % signs)
value	Variable value (may include another variable surrounded by % signs)

Example

In a trigger for the table AUTHOR, the following macro:

```
.DEFINE "TRIGGER" "T_%TABLE%"
message 'Error: Trigger(%TRIGGER%) of table %TABLE%'
```

generates the following trigger script:

```
message 'Error: Trigger(T_AUTHOR) of table AUTHOR';
```

.DEFINEIF Macro

Defines a variable and initializes its value if the test value is not null

Syntax

.DEFINEIF "*`test_value`***" "***`variable`***" "***`value`***"**

Argument	Description
test_value	Value to test
variable	Variable name (without % signs)
value	Variable value (may include another variable surrounded by % signs)

Example

For example, to define a variable for a default data type:

```
%DEFAULT%
.DEFINEIF "%DEFAULT%" "_DEFLT"" "%DEFAULT%"
Add %COLUMN% %DATATYPE% %_DEFLT%
```

.ERROR Macro

Handles errors.

Syntax

.ERROR (*`errno`***, "***`errmsg`***")**

Argument	Description
errno	Error number
errmsg	Error message

Example

```
.ERROR(-20001, "Parent does not exist, cannot insert child")
```

.FKCOLN Macro

Repeats a statement for each foreign key column in a table.

Syntax

.FKCOLN("*`statement`***","***`prefix`***","***`suffix`***","***`last_suffix`***")**

Argument	Description
statement	Statement to repeat for each column
prefix	Prefix for each new line
suffix	Suffix for each new line
last suffix	Suffix for the last line

Example

In a trigger for the table TITLEAUTHOR, the following macro:

```
message .FKCOLN("'%COLUMN% is a foreign key column'","",",",";")
```

generates the following trigger script:

```
message 'AU_ID is a foreign key column,
TITLE_ISBN is a foreign key column;'
```

Note: For columns, the macro FKCOLN only accepts the variable %COLUMN%.

.FOREACH_CHILD Macro

Repeats a statement for each parent-to-child reference in the current table fulfilling a condition.

Syntax

.FOREACH_CHILD ("*condition***")**

"*statement***"**

.ENDFOR

Argument	Description
condition	Reference condition (see below)
statement	Statement to repeat

Condition	Selects
UPDATE RESTRICT	Restrict on update
UPDATE CASCADE	Cascade on update
UPDATE SETNULL	Set null on update
UPDATE SETDEFAULT	Set default on update
DELETE RESTRICT	Restrict on delete

Condition	Selects
DELETE CASCADE	Cascade on delete
DELETE SETNULL	Set null on delete
DELETE SETDEFAULT	Set default on delete

Example

In a trigger for the table TITLE, the following macro:

```
.FOREACH_CHILD("DELETE RESTRICT")
--  Cannot delete parent "%PARENT%" if children still exist in
"%CHILD%"
.ENDFOR
```

generates the following trigger script:

```
--  Cannot delete parent "TITLE" if children still exist in
"ROYSCHED"
--  Cannot delete parent "TITLE" if children still exist in "SALE"
--  Cannot delete parent "TITLE" if children still exist in
"TITLEAUTHOR"
```

.FOREACH_COLUMN Macro

Repeats a statement for each column in the current table fulfilling a condition.

Syntax

.FOREACH_COLUMN ("*condition*")

"*statement*"

.ENDFOR

Argument	Description
condition	Column condition (see below)
statement	Statement to repeat

Condition	Selects
empty	All columns
PKCOLN	Primary key columns
FKCOLN	Foreign key columns
AKCOLN	Alternate key columns

Condition	Selects
NMFCOL	Non-modifiable columns (columns that have Cannot Modify selected as a check parameter)
INCOLN	Triggering columns (primary key columns, foreign key columns; and non-modifiable columns)

Example

In a trigger for the table TITLE, the following macro:

```
.FOREACH_COLUMN("NMFCOL")
-- "%COLUMN%" cannot be modified
.ENDFOR
```

generates the following trigger script:

```
-- "TITLE_ISBN" cannot be modified
-- "PUB_ID" cannot be modified
```

.FOREACH_PARENT Macro

Repeats a statement for each child-to-parent reference in the current table fulfilling a condition.

Syntax

.FOREACH_PARENT ("*`condition`***")**

"*`statement`***"**

.ENDFOR

Argument	Description
condition	Reference condition (see below)
statement	Statement to repeat

Condition	Selects references defined with ...
empty	All references
FKNULL	Non-mandatory foreign keys
FKNOTNULL	Mandatory foreign keys
FKCANTCHG	Non-modifiable foreign keys

Example

In a trigger for the table SALE, the following macro:

```
.FOREACH_PARENT("FKCANTCHG")
--  Cannot modify parent code of "%PARENT%" in child "%CHILD%"
.ENDFOR
```

generates the following trigger script:

```
--  Cannot modify parent code of "STORE" in child "SALE"
--  Cannot modify parent code of "TITLE" in child "SALE"
```

.INCOLN Macro

Repeats a statement for each primary key column, foreign key column, alternate key column, or non-modifiable column in a table.

Syntax

```
.INCOLN("statement","prefix","suffix","last_suffix")
```

Argument	Description
statement	Statement to repeat for each column
prefix	Prefix for each new line
suffix	Suffix for each new line
last suffix	Suffix for the last line

Example

In a trigger for the table TITLE, the following macro:

```
.INCOLN("%COLUMN% %COLTYPE%","",",",";")
```

generates the following trigger script:

```
TITLE_ISBN char(12),
PUB_ID char(12);
```

.JOIN Macro

Repeats a statement for column couple in a join.

Syntax

```
.JOIN("statement","prefix","suffix","last_suffix")
```

Argument	Description
statement	Statement to repeat for each column
prefix	Prefix for each new line
suffix	Suffix for each new line

Argument	Description
last suffix	Suffix for the last line

Example

In a trigger for the table TITLE, the following macro:

```
.FOREACH_PARENT()
where .JOIN("%PK%=%FK%", " and", "", ";")
message 'Reference %REFR% links table %PARENT% to %CHILD%'
 .ENDFOR
```

generates the following trigger script:

```
message 'Reference TITLE_PUB links table PUBLISHER to TITLE
```

Note: For columns, the macro JOIN only accepts the variables %PK%, %AK%, and %FK %.

.NMFCOL Macro

Repeats a statement for each non-modifiable column in a table. Non-modifiable columns have Cannot Modify selected as a check parameter.

Syntax

```
.NMFCOL("statement","prefix","suffix","last_suffix")
```

Argument	Description
statement	Statement to repeat for each column
prefix	Prefix for each new line
suffix	Suffix for each new line
last suffix	Suffix for the last line

Example

In a trigger for the table TITLE, the following macro:

```
.NMFCOL("%COLUMN% %COLTYPE%","",",",";")
```

generates the following trigger script:

```
TITLE_ISBN char(12),
PUB_ID char(12);
```

.PKCOLN Macro

Repeats a statement for each primary key column in a table.

Syntax

```
.PKCOLN("statement","prefix","suffix","last_suffix")
```

Argument	Description
statement	Statement to repeat for each column
prefix	Prefix for each new line
suffix	Suffix for each new line
last suffix	Suffix for the last line

Example

In a trigger for the table TITLEAUTHOR, the following macro:

```
message .PKCOLN("'%COLUMN% is a primary key column'","",",",";")
```

generates the following trigger script:

```
message 'AU_ID is a primary key column',
    'TITLE_ISBN is a primary key column';
```

Note: For columns, the macro PKCOLN only accepts the variable %COLUMN%.

.CLIENTEXPRESSION and .SERVEREXPRESSION Macros

Uses the client and/or server expression of a business rule in the trigger template, template item, trigger, and procedure script.

Syntax

.CLIENTEXPRESSION(code of the business rule)

.SERVEREXPRESSION(code of the business rule)

Example

The business rule ACTIVITY_DATE_CONTROL has the following server expression:

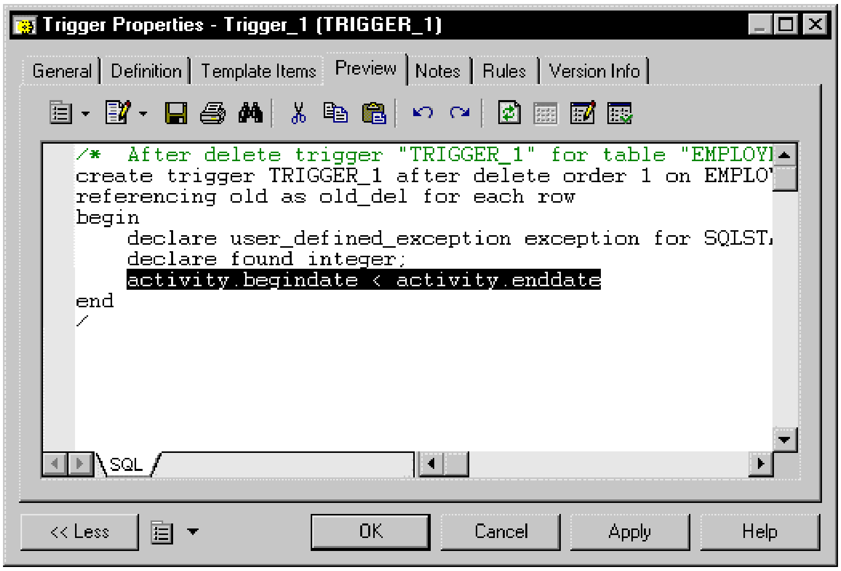
```
activity.begindate < activity.enddate
```

In a trigger based on template AfterDeleteTrigger, you type the following macro in the Definition tab of the trigger:

```
.SERVEREXPRESSION(ACTIVITY_DATE_CONTROL)
```

This generates the following trigger script:

```
activity.begindate < activity.enddate
end
```



```
/*  After delete trigger "TRIGGER_1" for table "EMPLOYI
create trigger TRIGGER_1 after delete order 1 on EMPLO'
referencing old as old_del for each row
begin
    declare user_defined_exception exception for SQLST.
    declare found integer;
    activity.begindate < activity.enddate
end
/
```

.SQLXML Macro

Represents a SQL/XML query in the definition of a trigger, a procedure or a function.

Use one of the following tools:

- The *Insert SQL/XML Macro* tool opens a selection dialog box where you choose a global element from an XML model. The XML model must be open in the workspace, mapped to a PDM, and have the SQL/XML extension file attached. Click OK in the dialog box and the SQLXML macro is displayed in the definition code, with the code of the XML model (optional) and the code of the global element.
- The *Macros* tool, where you select *.SQLXML( )* in the list. The SQLXML macro is displayed empty in the definition code. You must fill the parentheses with the code of an XML model (optional), followed by :: and the code of a global element. The XML model, from which you choose a global element, must be open in the workspace, mapped to a PDM, and have the SQL/XML extension file attached.

After generation, the SQLXML macro is replaced by the SQL/XML query of the global element.

Syntax

```
.SQLXML(code of an XML model::code of a global element)
```

Note: the code of an XML model is optional.

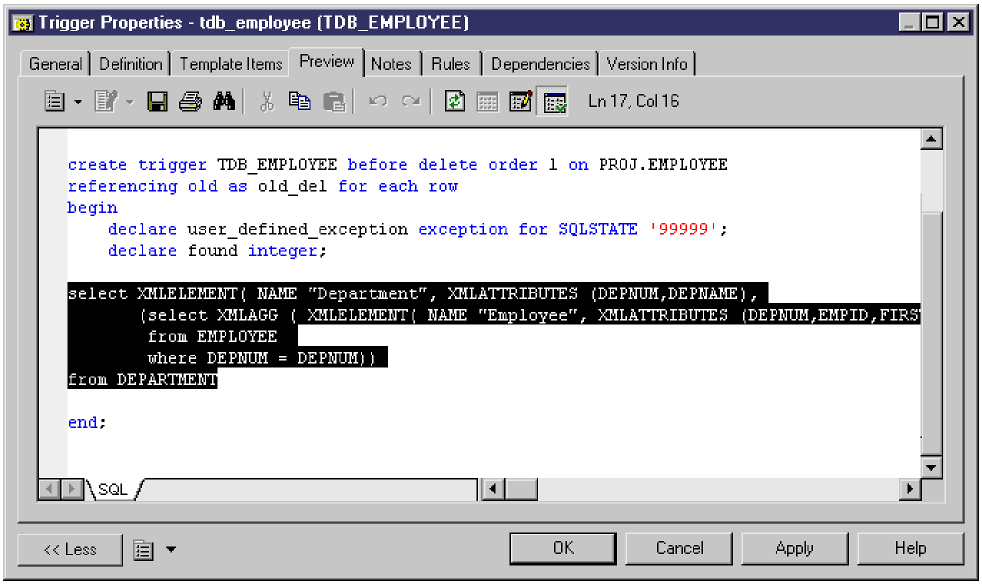
Example

In a trigger for the table EMPLOYEE, the following macro:

```
.SQLXML(CorporateMembership::DEPARTMENT)
```

generates the following trigger script:

```
select XMLELEMENT( NAME "Department", XMLATTRIBUTES
(DEPNUM,DEPNAME),
   (select XMLAGG ( XMLELEMENT( NAME "Employee", XMLATTRIBUTES
(DEPNUM,EMPID,FIRSTNAME,LASTNAME)) )
    from EMPLOYEE
    where DEPNUM = DEPNUM))
from DEPARTMENT
```



```
create trigger TDB_EMPLOYEE before delete order 1 on PROJ.EMPLOYEE
referencing old as old_del for each row
begin
    declare user_defined_exception exception for SQLSTATE '99999';
    declare found integer;

select XMLELEMENT( NAME "Department", XMLATTRIBUTES (DEPNUM,DEPNAME),
       (select XMLAGG ( XMLELEMENT( NAME "Employee", XMLATTRIBUTES (DEPNUM,EMPID,FIRS
        from EMPLOYEE
        where DEPNUM = DEPNUM))
from DEPARTMENT

end;
```

CHAPTER 5

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 (such as 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. For example, 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.

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.

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 templates - which 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 - which 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.

1. In the resource editor, right-click a metaclass in the Profile category, and select **New > Template** from the contextual menu.

   The convention is to name your templates using headless camelCase, (starting with a lowercase letter), in order to avoid clashes with property and collection names which, by convention use full CamelCase.
2. Right-click the metaclass again, and select **New > Generated File** from the contextual menu.
3. Insert the name of the template in the generated file 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 268.

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 271.

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 269.

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 269.

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 269.

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 65.

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 *Chapter 1, Working with PowerDesigner Resource Files* 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
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

Name	Type	Description
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 301. 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.
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 272]

You can combine format codes. For example, %.U8:CHILD% formats the first eight characters of the code of the CHILD table in uppercase letters.

GTL Operators

GTL supports standard arithmetic operators along with some advanced template operators.

The following standard arithmetical and logical operators are supported, where *x* and *y* can be numbers or templates resolving to numbers:

Operator	Description
`%+(x,y)%`	Addition operator
`%-(x,y)%`	Subtraction operator
`%*(x,y)%`	Multiplication operator
`%/(x,y)%`	Division operator
`%&(x,y)%`	Logical bitfield `and` operator

In this example, the template in the left column produces the output on the right:

Template	Results

`Base number=   %Number%`
`Number+1=      %+(Number,1)%`
`Number-1=      %-(Number,1)%`
`Number*2=      %*(Number,2)%`
`Number/2=      %/(Number,2)%`
`Number&1=      %&(Number,1)%`
	`Base number=   4`
	`Number+1=      5`
	`Number-1=      3`
	`Number*2=      8`
	`Number/2=      2`
	`Number&1=      0`

The following advanced template operators are also supported:

Operator	Description
*	Dereferencing operator - The syntax [*]+*local-value* [(*param-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). 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).

Operator	Description
!	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. The ! operator may be applied any number of times. For example: `%!!template%` 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.
	
	For example, if custname is defined whereas custid is not, then the template:
	
	`.set_value(foo, tt)`
	`%custname?%`
	`%custid?%`
	
	outputs:
	
	`true`
	`false`

+	The + operator is used to test if an object property is visible in the interface. For example, you could test if the Type box is displayed in the General tab of a database property sheet in a DMM, and thus that a Replication Server XEM is attached to the model. 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.

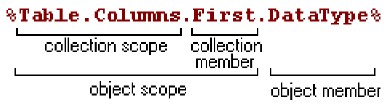
The initial scope is always the metaclass on which the template is defined. All metamodel attributes and collections defined on the active object metaclass and its parents are visible, as well as the corresponding extended attributes and templates.

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.

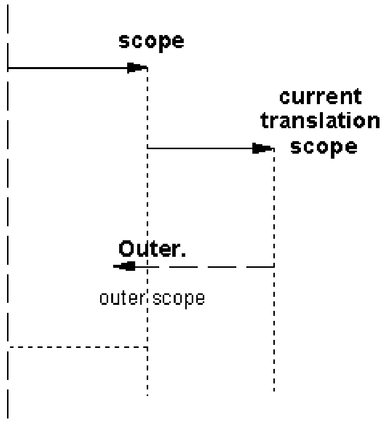
The following types of scope are available:

- 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 *Chapter 1, Working with PowerDesigner Resource Files* on page 1.

  For example:

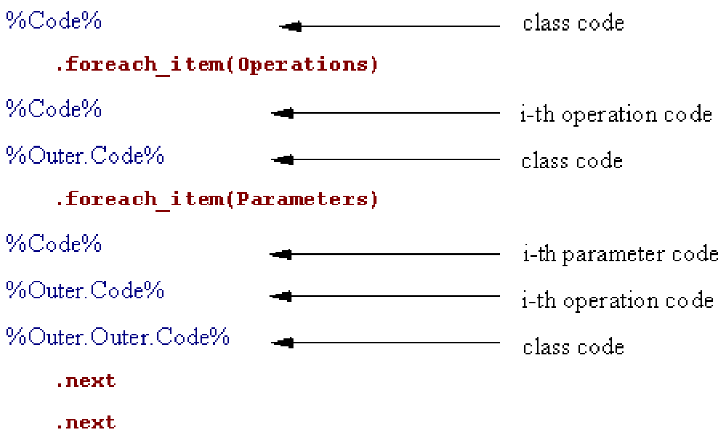
- Outer scope - accessed using the Outer keyword. The following rules apply:
  - When a scope is created, the old scope becomes the outer scope.
  - When a scope is exited, the outer scope is restored as the current translation scope



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 293) that allows for iteration on collections defines a new scope, and the foreach_line macro (see *.foreach_line macro* on page 295). 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.

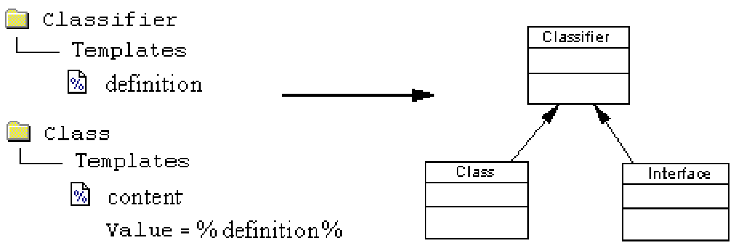
The following example shows the scope mechanism using a Class template:



```
%Code%
    .foreach_item(Operations)
%Code%
%Outer.Code%
    .foreach_item(Parameters)
%Code%
%Outer.Code%
%Outer.Outer.Code%
    .next
    .next
```

Inheritance and Polymorphism

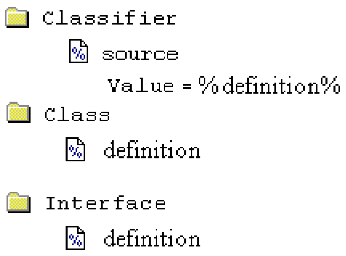
Templates are defined with respect to a given metaclass and are inherited by and available to the children of the metaclass. In the following example, the definition template defined on the parent metaclass is available to, and used in the evaluation of the content template on the child metaclass.



GTL supports the following OO concepts as part of inheritance:

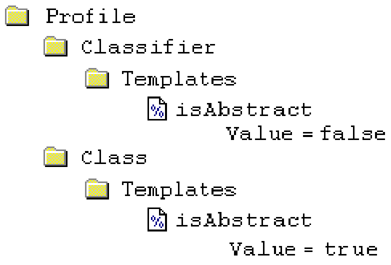
- *Polymorphism* - Templates are dynamically bound; the choice of the template to be evaluated is made at translation-time. Polymorphism allows template code defined on a classifier to use templates defined on its children (class, interface), the template being used does not have to be defined on the 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:



```
Classifier
    source
        Value = %definition%
Class
    definition

Interface
    definition
```

- *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 52 and *Stereotypes (Profile)* on page 49), 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 301.

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

Escape sequence	Description
\ 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 279.

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, .endre-place, ...)
.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_com-mand(executable [,*arguments*[,{cmd_ShellExecute|cmd_PipeOutput}]])

Syntax error message	Description and correction
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}])
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 297
  - *.foreach_item macro* on page 293 – iterates on object collections
  - *.foreach_line macro* on page 295 – iterates on lines
  - *.foreach_part macro* on page 295 – iterates on parts
  - *.break macro* on page 288 – 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 301
  - *.unset macro* on page 303
- *Output and error reporting macros*:
  - *.log macro* on page 298
  - *.error and .warning macros* on page 291
- *Command macros* - only available in the context of the execution of a generic command:
  - *.vbscript macro* on page 303 - embed VB script code inside a template
  - *.execute_vbscript macro* on page 293 - launch vbscripts
  - *.execute_command macro* on page 292 - launch executables
  - *.abort_command macro* on page 287 - stop command execution
  - *.change_dir macro* on page 288 - changing a directory
  - *.create_path macro* on page 290 - creating a specified path
- *Formatting macros*:
  - *.lowercase and .uppercase macros* on page 299
  - *.convert_code and .convert_code macros* on page 289 – converts codes into names
- *String manipulation macros*:
  - *.replace macro* on page 300
  - *.delete macro* on page 291
  - *.unique macro* on page 302

    - *.block macro* on page 287 - adds a header and a footer to a text block
- *Miscellaneous macros*:
    - *.comment and .// macro* on page 289 - inserts a comment in a template
    - *.collection macro* on page 289 - returns a collection of objects based on the specified scope and condition
    - *.object macro* on page 299 - returns an object based on the specified scope and condition
    - *.bool macro* on page 288 - evaluates a condition
    - *.set_interactive_mode macro* on page 301 – 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. Type: Simple template
block-input	Parameter used to input text Type: Complex template
tail	[optional] Appended to the output, if there is one 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 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 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. Type: <simple-template> returning a collection scope
filter	[optional] Filter condition 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_name and .convert_code Macros

These macros convert the object name to its code (or vice versa).

Use the following syntax to convert a name to a code:

```
.convert_name (expression[,"separator"[,"separator_pattern"],case])
```

Use the following syntax to convert a code to a name:

```
.convert_code (expression[,"separator"[,"separator_pattern"]])
```

The following parameters are available:

Parameter	Description
expression	Specifies the text to be converted. For .convert_name, this is generally the `%Name%` variable and may include a suffix or prefix. Type: Simple template
separator	[optional] Character generated each time a separator declared in **pattern-separator** is found in the code. For example, "_" (underscore). Type: Text
pattern-separator	[optional] Declaration of the different separators likely to exist in a code, and which will be replaced by **separator**. You can declare several separators, for example "_ " and "-" Type: Text
case	[optional for `.convert_name` only] Specifies the case into which to convert the code. You can choose between: • `firstLowerWord` - First word in lowercase, first letters of subsequent words in uppercase • `FirstUpperChar` - First character of all words in uppercase • `lower_case` - All words in lowercase and separated by an underscore • `UPPER_CASE` - All words in uppercase and separated by an underscore

In the following example, the `.convert_name` macro is 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)
```

Note: These macros can also be used to perform conversions to apply naming conventions in your model. For more information, see *Core Features Guide > The PowerDesigner Interface > Objects > Object Properties > Naming Conventions*.

.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 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 *Core Features Guide > The PowerDesigner Interface > Objects > Object Properties > Naming Conventions*).

The following parameters are available:

Parameter	Description
del-string	String to be deleted in the input block Type: Text
delete-block-input	Parameter used to input text 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 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 Type: Simple template (escape sequences ignored)
args	[optional] Arguments for the executable Type: Simple template (escape sequences ignored)
mode	[optional] You can choose one of the following: • cmd_ShellExecute - runs as an independent process • 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 Type: Simple template (escape sequences ignored)
script-parameter	[optional] Passed to the script through the ScriptInputParameters global property. 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 328.

.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 Type: Simple template

Parameter	Description
head	[optional] Generated before output, if there is one Type: Text
tail	[optional] Appended to the output, if there is one Type: Text
condition	[optional] If specified, only objects satisfying the given condition are considered during the iteration 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 Type: Simple condition
complex-template	Template to apply to each item. Type: Complex template
separator	[optional] Generated between non empty evaluations of <complex-template> 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 Type: Simple template
head	[optional] Generated before output, if there is one Type: Text
tail	[optional] Appended to the output, if there is one Type: Text
complex-template	Template to apply to each line. Type: Complex template
separator	[optional] Generated between non empty evaluations of *complex-template* Type: Text

Example:

```
.foreach_line(%Comment%)
// %CurrentLine%
.next(\n)
```

.foreach_part Macro

This macro iterates through and transforms the parts of the input template, with the parts delimited by a separator pattern.

```
.foreach_part (expression [,"separator" [,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 often used in applying naming conventions (see *Core Features Guide > The PowerDesigner Interface > Objects > Object Properties > Naming Conventions*).

The following parameters are available:

Parameter	Description
input	Input text over which iteration is performed Type: Simple template
separator-pattern	Char and word separators • Any character specified in the pattern can be used as separator • [<c1> - <c2>] specifies a character within the range defined between both characters <c1> and <c2> 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). By default, the <separator-pattern> is initialized with the pattern (). If the specified pattern is empty, the pattern is initialized using the default value. 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. There are two kinds of separator: • Char separator - for each char separator, the separator specified in the next statement of the macro is returned (even for consecutive separators) • 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 Default: " -_,\t" Type: Text
head	[optional] Generated before output, if there is one Type: Text
tail	[optional] Appended to the output, if there is one Type: Text
simple-template	Template to apply to each part. Type: Complex template
separator	[optional] Generated between non empty evaluations of *complex-template* 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: `variable [operator comparison]` 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). Where *comparison* may be any of: • A simple template • "*text*" • true • false • null • notnull 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. You can chain conditions together using the and or or logical operators. Type: Simple template
complex-tem-plate	The template to apply is the condition is true. Type: Complex template
tail	Appended to the output, if there is one 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 Type: Simple template

Example:

```
.log undefined environment variable: JAVAC
```

.lowercase and .uppercase Macros

These macros convert text blocks to the specified case.

```
.lowercase
    block-input
.endlowercase
```

and

```
.uppercase
    block-input
.enduppercase
```

These macros are particularly useful when working with naming conventions (see *Core Features Guide > The PowerDesigner Interface > Objects > Object Properties > Naming Conventions*).

The following parameters are available:

Parameter	Description
block-input	Parameter used to input text 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 Type: Simple template returning either an object or a collection scope
simple-tem-plate	Template to be evaluated. Type: Simple template

Parameter	Description
filter	Filter condition Type: condition

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))
```

In the following example, template `::myPackage2` is defined as:

```
.object(ActiveModel.Packages, %Name% == MyPackage2)
```

and template `OOM.Model::MyTemplate` is defined as:

```
.foreach_item(myPackage2.Classes)
%Code%
.next(\n)
```

In `OOM.Model M = { OOM.Package MyPackage1, OOM.Package MyPackage2 { OOM.Class C1, OOM.Class C2} }`, and template `OOM.Model::MyTemplate` evaluates to:

```
C1
C2
```

In the following example, this template in a DMM returns the first read-only data connection for the process associated with the current publication:

```
.object(Process.DataConnections, %AccessType% == "RO")
```

.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 *Core Features Guide > The PowerDesigner Interface > Objects > Object Properties > Naming Conventions*.

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. Type: Text

Parameter	Description
new-string	String which replaces *old-string*. Type: Text
block-input	Parameter used to input text. 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. Type: Simple-template returning an object or a collection scope
name	Variable name 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 Type: [*scope*.]*object-scope*]
value [.set_value only]	Value. Type: Simple template (escape sequences ignored)
mode	[optional] Specifies the mode of creation. You can choose between: • new - (Re)define the variable in the current scope • update – [default] If a variable with the same name already exists, update the existing variable otherwise define a new one • 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 Type: Complex template
tail	[optional] Appended to the output, if there is one 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. Type: Simple-template returning an object or a collection scope
name	Local variable or volatile attribute name. 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. Type: List of simple-template arguments separated by commas
block-input	VB script text Type: Text
tail	Appended to the output, if there is one 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 328) as ActiveSelection.Item(0).

CHAPTER 6

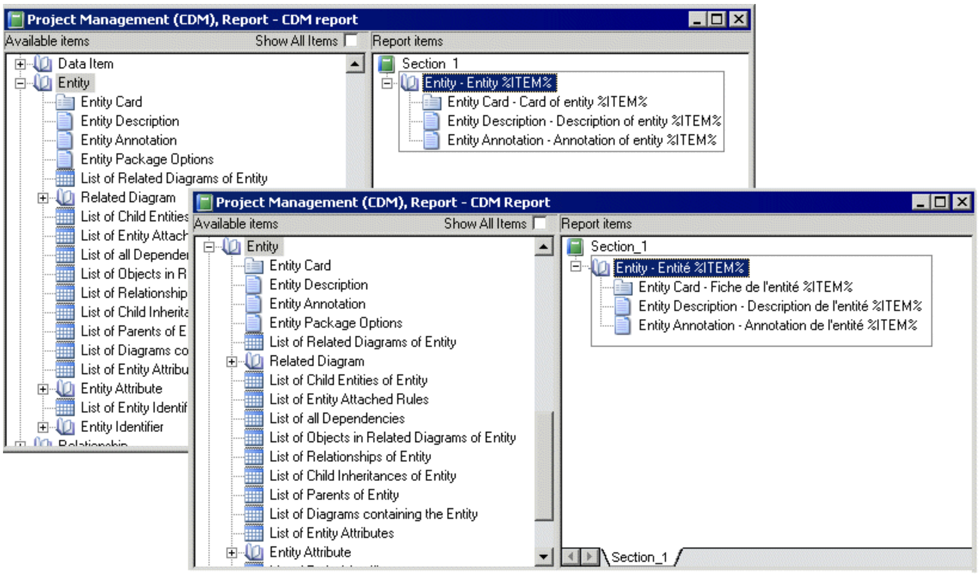
Translating Reports with Report Language Resource Files

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 *Core Features Guide > The PowerDesigner Interface > Reports*.

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:



Project Management (CDM), Report - CDM report
Available items
Show All Items
Report items
Data Item
Entity
Entity Card
Entity Description
Entity Annotation
Entity Package Options
List of Related Diagrams of Entity
Related Diagram
Section 1
Entity - Entity %ITEM%
Entity Card - Card of entity %ITEM%
Entity Description - Description of entity %ITEM%
Entity Annotation - Annotation of entity %ITEM%
Project Management (CDM), Report - CDM Report
Available items
Show All Items
Report items
Entity
Entity Card
Entity Description
Entity Annotation
Entity Package Options
List of Related Diagrams of Entity
Related Diagram
List of Child Entities of Entity
List of Entity Attached Rules
List of all Dependencies
List of Objects in Related Diagrams of Entity
List of Relationships of Entity
List of Child Inheritances of Entity
List of Parents of Entity
List of Diagrams containing the Entity
List of Entity Attributes
Entity Attribute
Section_1
Entity - Entité %ITEM%
Entity Card - Fiche de l'entité %ITEM%
Entity Description - Description de l'entité %ITEM%
Entity Annotation - Annotation de l'entité %ITEM%
Section_1

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 319) is translated as:

```
   Liste des %@Value% %ParentMetaClass.OFTHECLSSNAME% %%PARENT%%
```

in which the following variables are resolved:

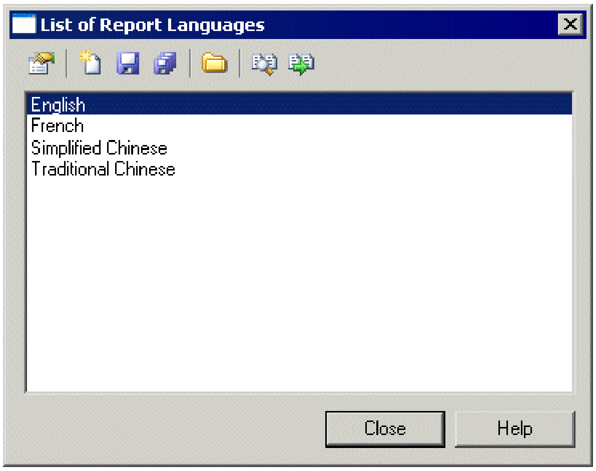
- %@Value% - resolves to the object type of the metaclass (see *Object Attributes category* on page 316). In this case, données.
- %ParentMetaClass.OFTHECLSSNAME% %%PARENT%% - resolves to the object type of the parent metaclass, as generated by the OFTHECLSSNAME linguistic variable (see *Profile/Linguistic Variables category* on page 317). In this case, l'entité.
- %%PARENT%% - resolves to the name of the specific object (see *Object Attributes category* on page 316). In this case, MyEntity.

For more information about templates, see *Chapter 5, Customizing Generation with GTL* on page 263.

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 *Chapter 1, Working with PowerDesigner Resource Files* on page 1.

Creating a Report Language Resource File for a New Language

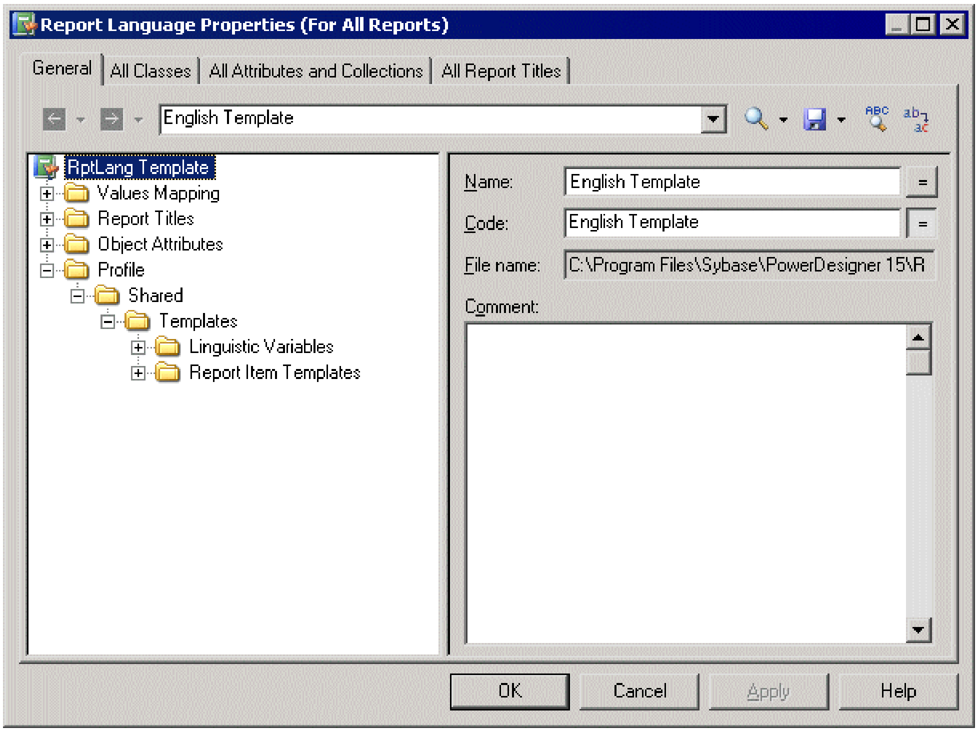
You can translate reports and other text items used to generate PowerDesigner reports into a new language.

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 310.
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 317.

7. Open the **Profile > Report Items Templates** category, and translate the various templates. For more information, see *Profile/Report Item Templates category* on page 319. 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 320.
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 321.
10. Click the All Report Titles tab, and review the automatically generated report titles. For more information, see *All Report Titles tab* on page 321. 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:



Name: English Template
Code: English Template
File name: C:\Program Files\Sybase\PowerDesigner 15\R
Comment:

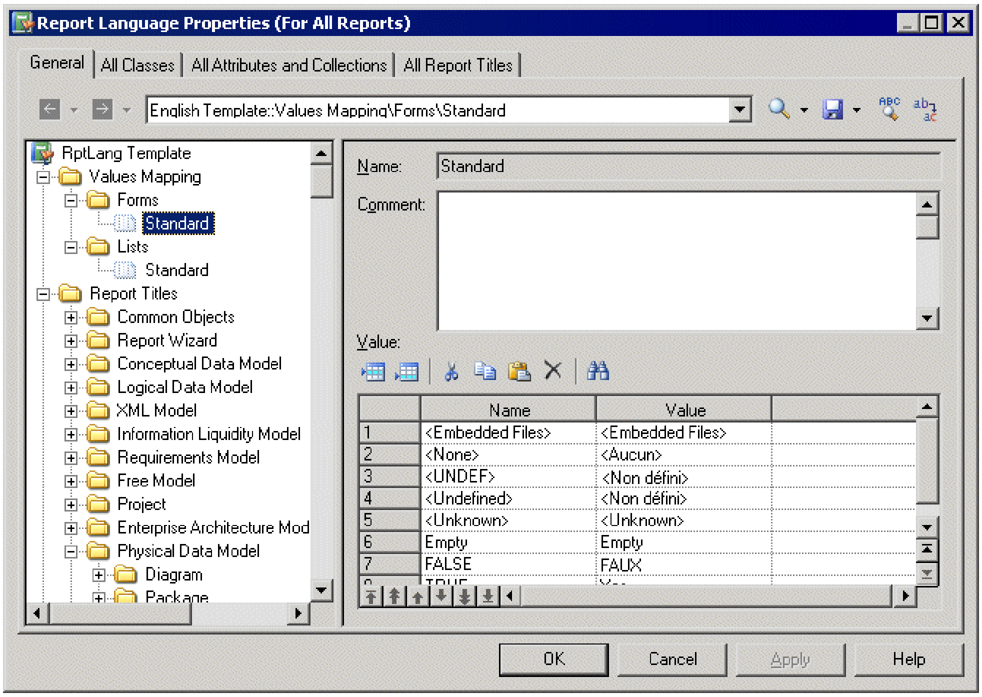
For more information about using the Resource Editor, see *Opening Resource Files in the Editor* on page 2.

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.
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:


Report Language Properties (For All Reports)

General | All Classes | All Attributes and Collections | All Report Titles

English Template::Values Mapping\Forms\Standard

- RptLang Template
  - Values Mapping
    - Forms
      - Standard
    - Lists
      - Standard
  - Report Titles
    - Common Objects
    - Report Wizard
    - Conceptual Data Model
    - Logical Data Model
    - XML Model
    - Information Liquidity Model
    - Requirements Model
    - Free Model
    - Project
    - Enterprise Architecture Mod
    - Physical Data Model
      - Diagram
      - Package

Name: Standard

Comment:

Value:

	Name	Value
1	<Embedded Files>	<Embedded Files>
2	<None>	<Aucun>
3	<UNDEF>	<Non défini>
4	<Undefined>	<Non défini>
5	<Unknown>	<Unknown>
6	Empty	Empty
7	FALSE	FAUX

OK | Cancel | Apply | Help

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. 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. Example: True.

You can create new mapping tables containing keywords values specific to particular types of model objects.

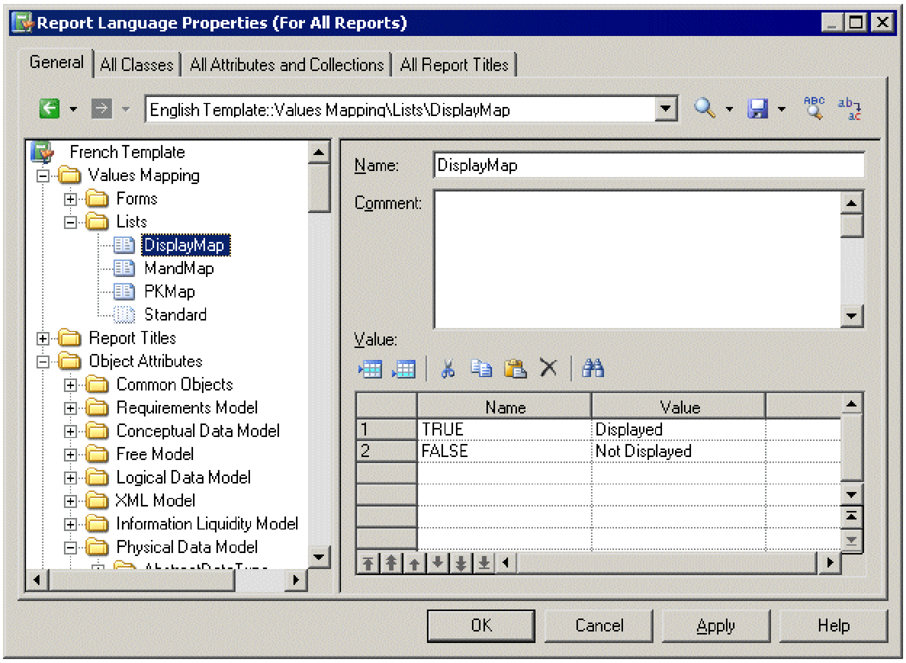
Example: Creating a Mapping Table, and Attaching It to a Specific Model Object

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.



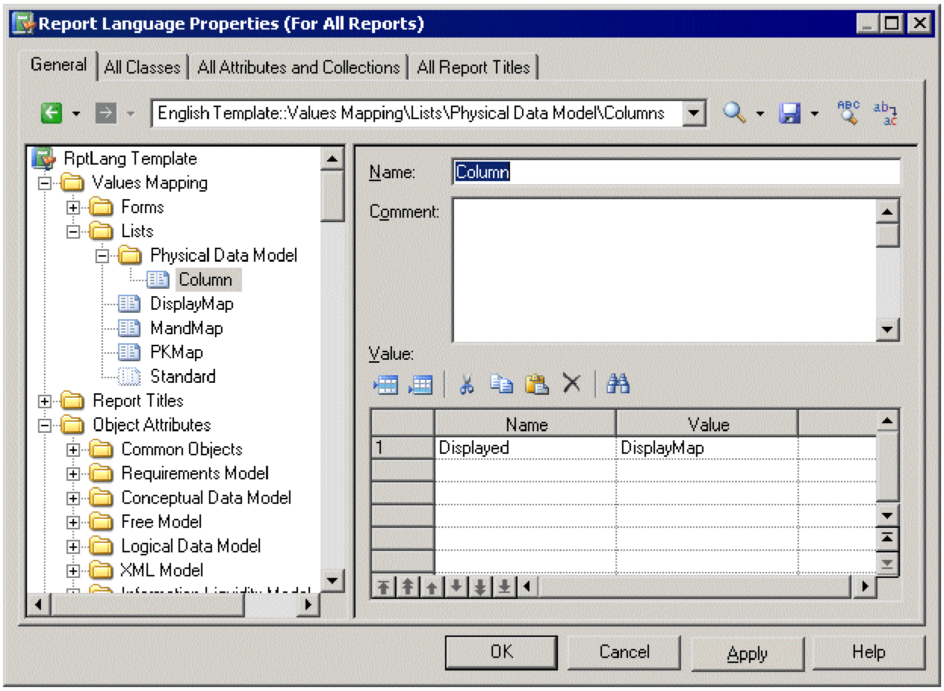
Name: DisplayMap

Comment:

Value:

	Name	Value
1	TRUE	Displayed
2	FALSE	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.



Name: Column

	Name	Value
1	Displayed	DisplayMap

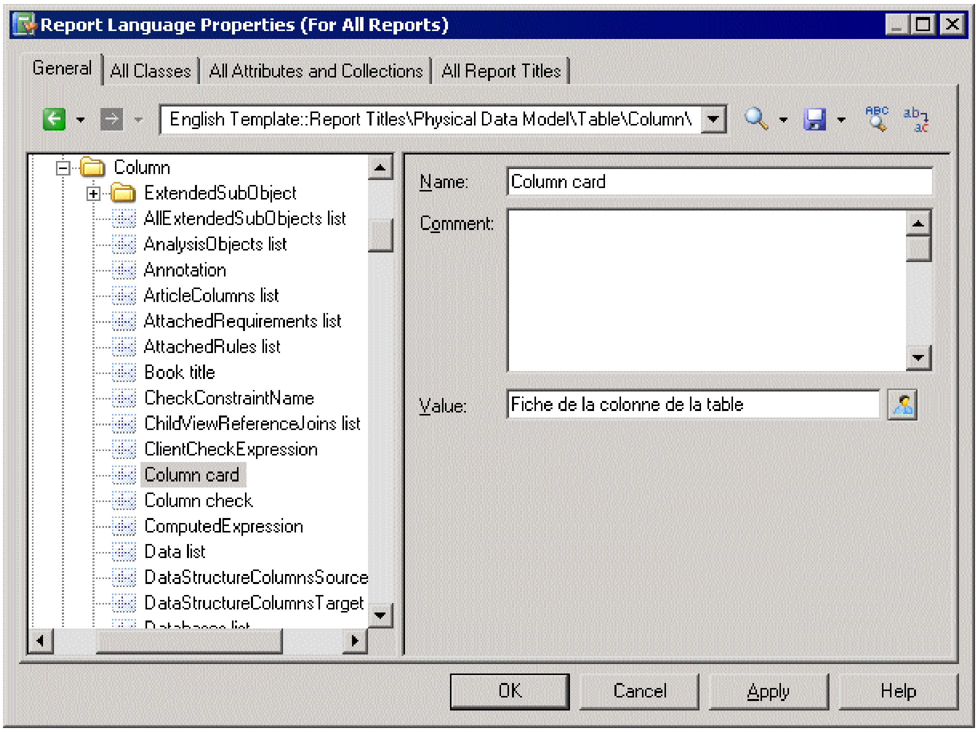
8. Click Apply to save your changes. When you generate a report, the Displayed property will be shown using the specified values:

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.


Report Language Properties (For All Reports)
General
All Classes
All Attributes and Collections
All Report Titles
English Template::Report Titles\Physical Data Model\Table\Column\
Column
ExtendedSubObject
AllExtendedSubObjects list
AnalysisObjects list
Annotation
ArticleColumns list
AttachedRequirements list
AttachedRules list
Book title
CheckConstraintName
ChildViewReferenceJoins list
ClientCheckExpression
Column card
Column check
ComputedExpression
Data list
DataStructureColumnsSource
DataStructureColumnsTarget
OK
Cancel
Apply
Help

Name: Column card

Comment:

Value: Fiche de la colonne de la table

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. 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. Example: Short description title provides the text for a short description section when you generate a report with the Report Wizard.

Sub-category	Description
[Models]	Contain the report titles and other text items available to each model. These are automatically generated, but you can override the default values. Example: DataTransformationTasks list provides the text for the data transformation tasks list of a given transformation process in the Data Movement Model.

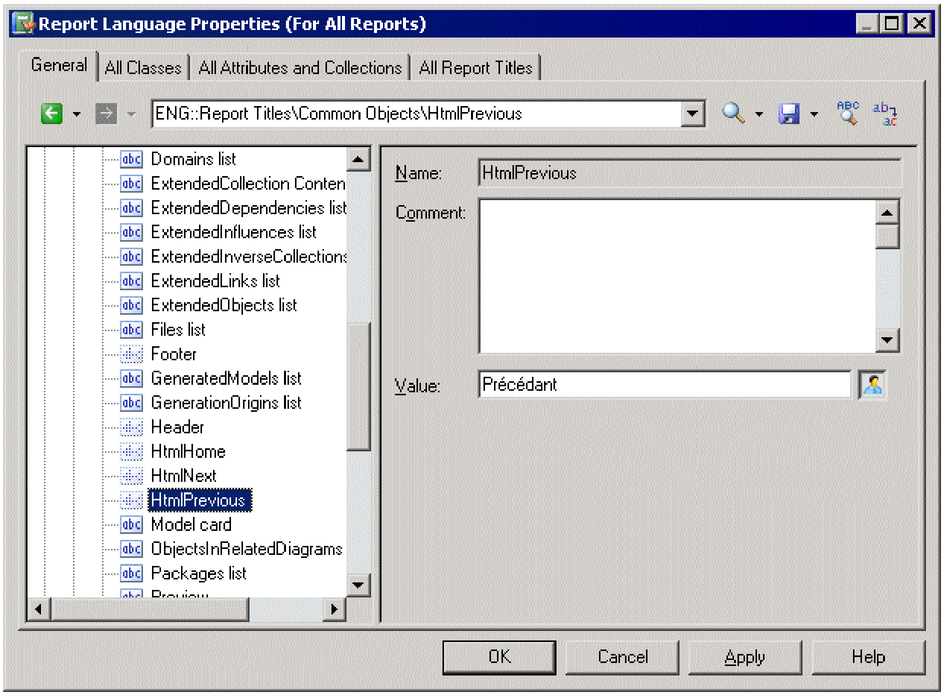
By default (with the exception of the Common Objects and Report Wizard sub-categories) these translations are automatically generated from the templates in the Profile category (See *Profile/Report Item Templates category* on page 319). You can override the automatically generated values by entering 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 321) 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.



ENG::Report Titles\Common Objects\HtmlPrevious

Name: HtmlPrevious

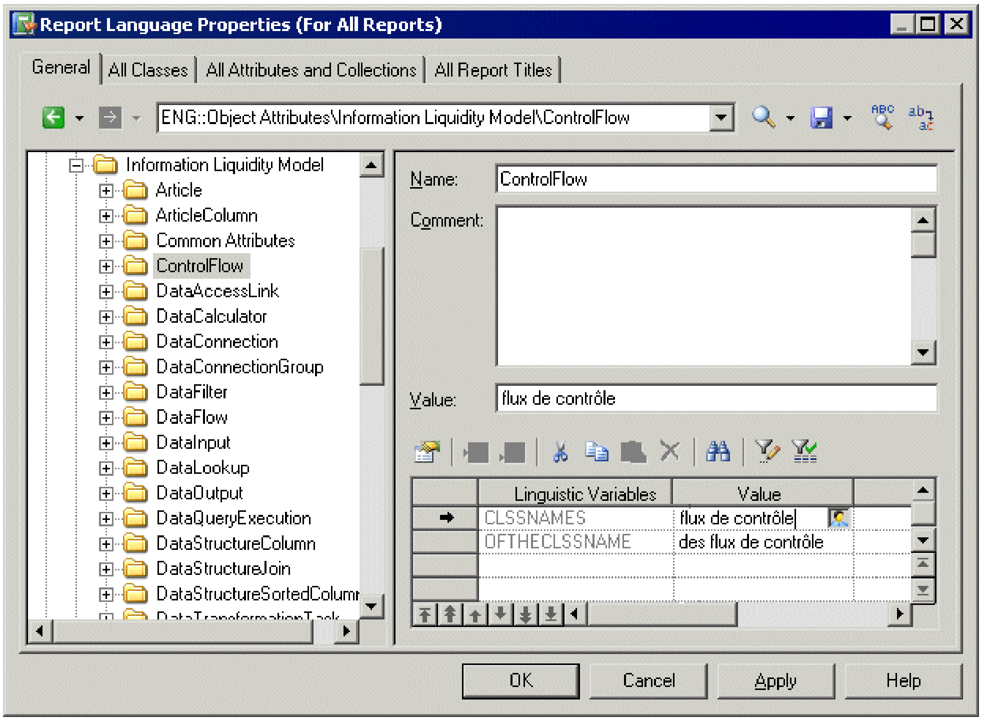
Comment:

Value: Précédant

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:


Report Language Properties (For All Reports)
General
All Classes
All Attributes and Collections
All Report Titles
OK
Cancel
Apply
Help

ENG::Object Attributes\Information Liquidity Model\ControlFlow

- Information Liquidity Model
  - Article
  - ArticleColumn
  - Common Attributes
  - ControlFlow
  - DataAccessLink
  - DataCalculator
  - DataConnection
  - DataConnectionGroup
  - DataFilter
  - DataFlow
  - DataInput
  - DataLookup
  - DataOutput
  - DataQueryExecution
  - DataStructureColumn
  - DataStructureJoin
  - DataStructureSortedColumn

Name: ControlFlow

Comment:

Value: flux de contrôle

Linguistic Variables	Value
CLSSNAMES	flux de contrôle
OFTHECLSSNAME	des flux de contrôle

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. 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. 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 313).

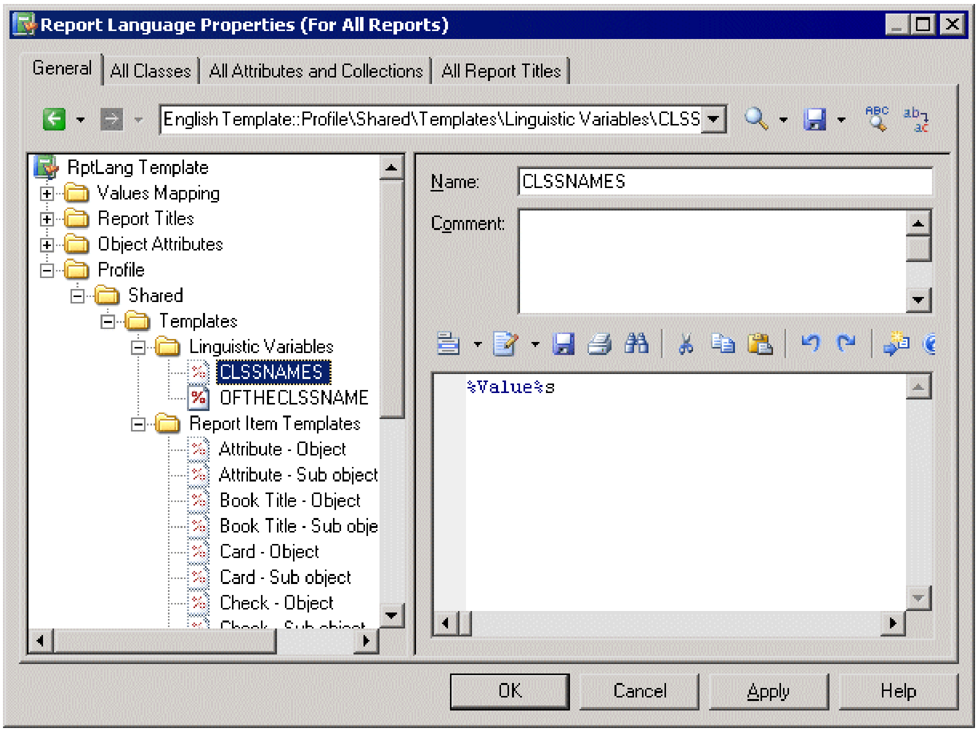
For metaclasses only, the linguistic variables you have specified (see *Profile/Linguistic Variables category* on page 317) are listed along with the results of their application to the translations given in the Localized name field. If necessary, you can override the automatically generated values by 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 320 and *All Attributes and Collections tab* on page 321).

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 319.



Name: CLSSNAMES

Comment:

```
%Value%s
```

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 316).

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".

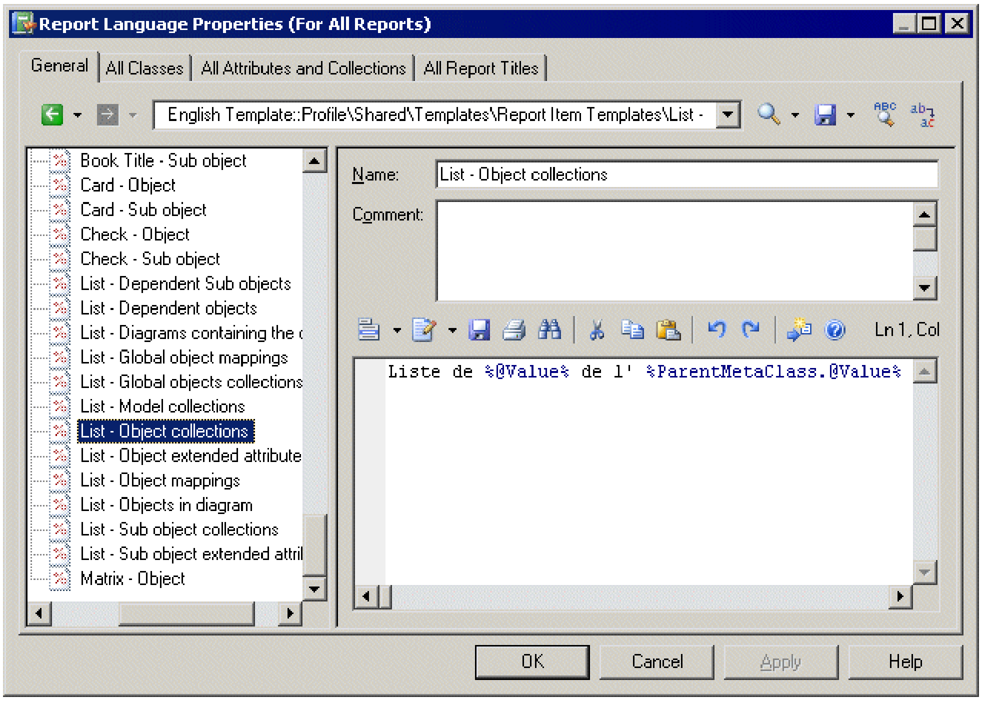
```
.if (%.1U:@Value% == A) or (%.1U:@Value% == E) or (%.1U:@Value% == I)
or (%.1U:@Value% == O) or (%.1U:@Value% == U)
d'%@Value%
.else
de %@Value%
.endif
```

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 316.


Report Language Properties (For All Reports)
General
All Classes
All Attributes and Collections
All Report Titles
English Template::Profile\Shared\Templates\Report Item Templates\List -
Book Title - Sub object
Card - Object
Card - Sub object
Check - Object
Check - Sub object
List - Dependent Sub objects
List - Dependent objects
List - Diagrams containing the
List - Global object mappings
List - Global objects collections
List - Model collections
List - Object collections
List - Object extended attribute
List - Object mappings
List - Objects in diagram
List - Sub object collections
List - Sub object extended attri
Matrix - Object
Name:
List - Object collections
Comment:
Ln 1, Col
OK
Cancel
Apply
Help

```
Liste de %@Value% de l' %ParentMetaClass.@Value%
```

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 317).

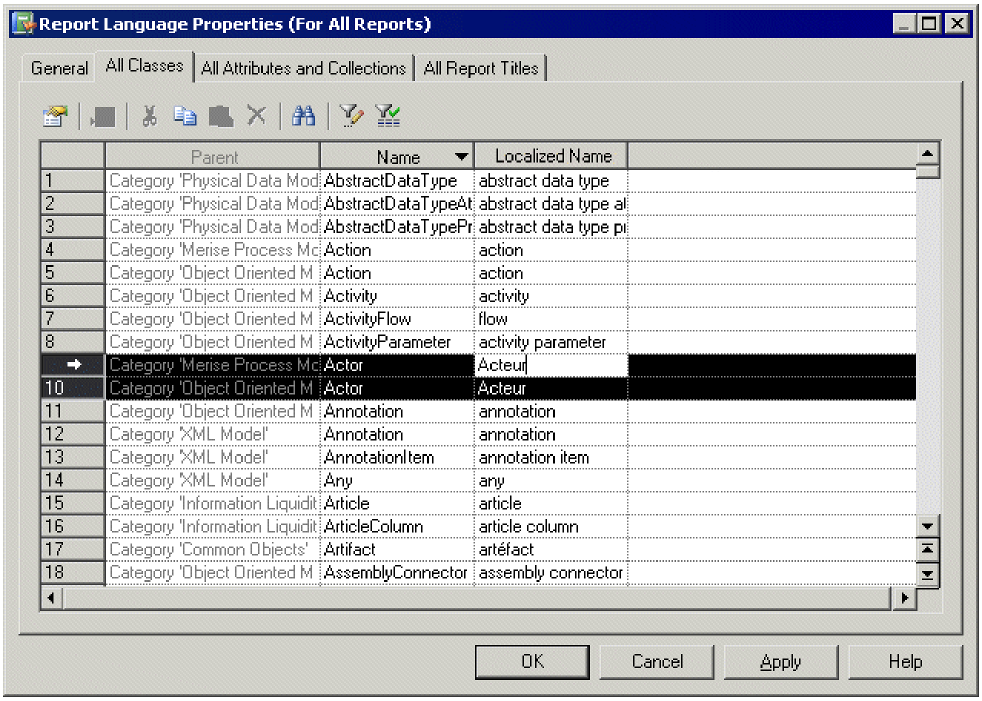
This template will be reused to create report titles for all the lists of sub-objects contained within a collection belonging to an object.

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 316.


Report Language Properties (For All Reports)
General
All Classes
All Attributes and Collections
All Report Titles
OK
Cancel
Apply
Help

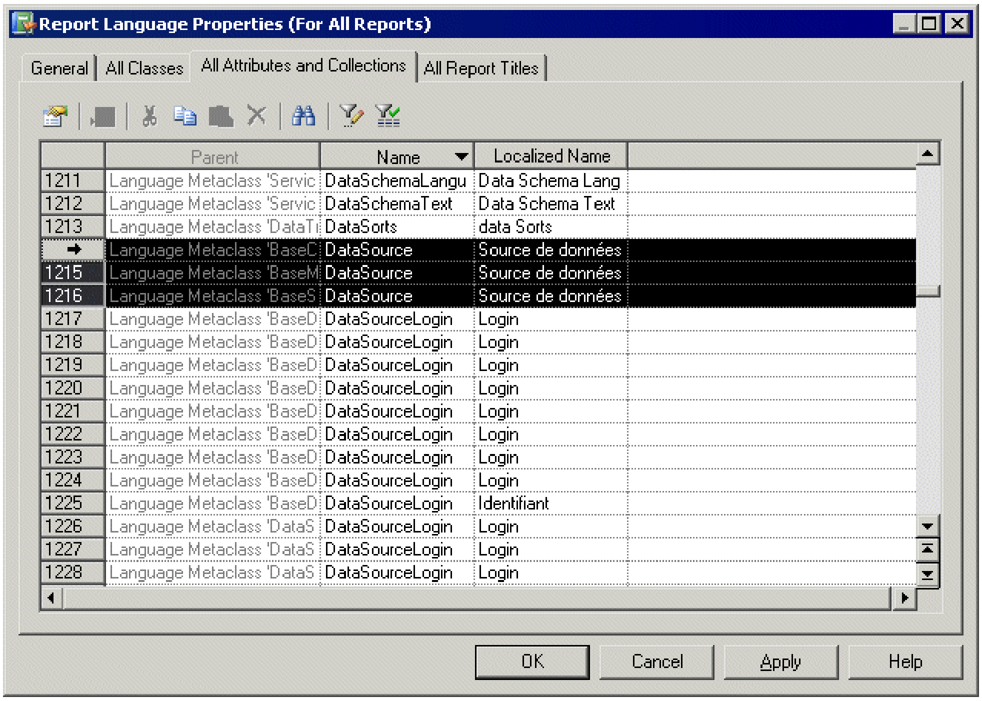
	Parent	Name	Localized Name
1	Category 'Physical Data Mod	AbstractDataType	abstract data type
2	Category 'Physical Data Mod	AbstractDataTypeAt	abstract data type al
3	Category 'Physical Data Mod	AbstractDataTypePr	abstract data type pı
4	Category 'Merise Process Mc	Action	action
5	Category 'Object Oriented M	Action	action
6	Category 'Object Oriented M	Activity	activity
7	Category 'Object Oriented M	ActivityFlow	flow
8	Category 'Object Oriented M	ActivityParameter	activity parameter
→	Category 'Merise Process Mc	Actor	Acteur
10	Category 'Object Oriented M	Actor	Acteur
11	Category 'Object Oriented M	Annotation	annotation
12	Category 'XML Model'	Annotation	annotation
13	Category 'XML Model'	AnnotationItem	annotation item
14	Category 'XML Model'	Any	any
15	Category 'Information Liquidit	Article	article
16	Category 'Information Liquidit	ArticleColumn	article column
17	Category 'Common Objects'	Artifact	artéfact
18	Category 'Object Oriented M	AssemblyConnector	assembly connector

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 316.


Report Language Properties (For All Reports)
General
All Classes
All Attributes and Collections
All Report Titles
OK
Cancel
Apply
Help

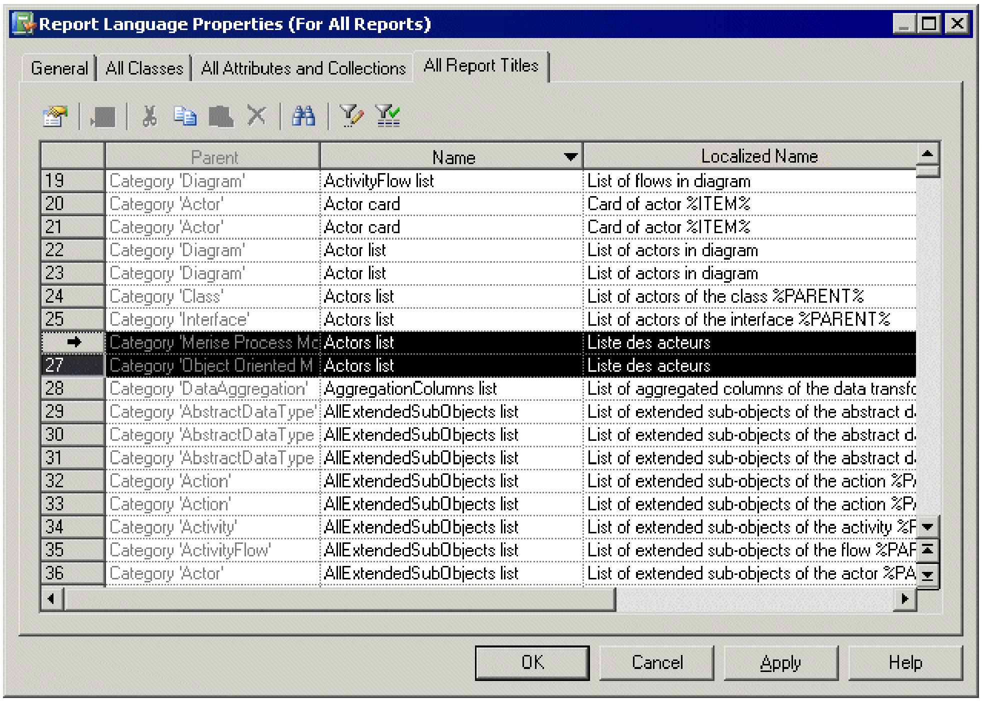
	Parent	Name	Localized Name
1211	Language Metaclass 'Servic	DataSchemaLangu	Data Schema Lang
1212	Language Metaclass 'Servic	DataSchemaText	Data Schema Text
1213	Language Metaclass 'DataTi	DataSorts	data Sorts
→	Language Metaclass 'BaseC	DataSource	Source de données
1215	Language Metaclass 'BaseM	DataSource	Source de données
1216	Language Metaclass 'BaseS	DataSource	Source de données
1217	Language Metaclass 'BaseD	DataSourceLogin	Login
1218	Language Metaclass 'BaseD	DataSourceLogin	Login
1219	Language Metaclass 'BaseD	DataSourceLogin	Login
1220	Language Metaclass 'BaseD	DataSourceLogin	Login
1221	Language Metaclass 'BaseD	DataSourceLogin	Login
1222	Language Metaclass 'BaseD	DataSourceLogin	Login
1223	Language Metaclass 'BaseD	DataSourceLogin	Login
1224	Language Metaclass 'BaseD	DataSourceLogin	Login
1225	Language Metaclass 'BaseD	DataSourceLogin	Identifiant
1226	Language Metaclass 'DataS	DataSourceLogin	Login
1227	Language Metaclass 'DataS	DataSourceLogin	Login
1228	Language Metaclass 'DataS	DataSourceLogin	Login

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 316.


Report Language Properties (For All Reports)
General
All Classes
All Attributes and Collections
All Report Titles
OK
Cancel
Apply
Help

	Parent	Name	Localized Name
19	Category 'Diagram'	ActivityFlow list	List of flows in diagram
20	Category 'Actor'	Actor card	Card of actor %ITEM%
21	Category 'Actor'	Actor card	Card of actor %ITEM%
22	Category 'Diagram'	Actor list	List of actors in diagram
23	Category 'Diagram'	Actor list	List of actors in diagram
24	Category 'Class'	Actors list	List of actors of the class %PARENT%
25	Category 'Interface'	Actors list	List of actors of the interface %PARENT%
➜	Category 'Merise Process Mc	Actors list	Liste des acteurs
27	Category 'Object Oriented M	Actors list	Liste des acteurs
28	Category 'DataAggregation'	AggregationColumns list	List of aggregated columns of the data transfc
29	Category 'AbstractDataType'	AllExtendedSubObjects list	List of extended sub-objects of the abstract d
30	Category 'AbstractDataType	AllExtendedSubObjects list	List of extended sub-objects of the abstract d
31	Category 'AbstractDataType	AllExtendedSubObjects list	List of extended sub-objects of the abstract d
32	Category 'Action'	AllExtendedSubObjects list	List of extended sub-objects of the action %P
33	Category 'Action'	AllExtendedSubObjects list	List of extended sub-objects of the action %P
34	Category 'Activity'	AllExtendedSubObjects list	List of extended sub-objects of the activity %F
35	Category 'ActivityFlow'	AllExtendedSubObjects list	List of extended sub-objects of the flow %PAF
36	Category 'Actor'	AllExtendedSubObjects list	List of extended sub-objects of the actor %PA

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.

CHAPTER 7 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 *Chapter 2, Extension Files* on page 21)
- Communicate with PowerDesigner from another application (see *Communicating With PowerDesigner Using OLE Automation* on page 381)
- Customize PowerDesigner menus by adding your own menu items (see *Customizing PowerDesigner Menus Using Add-Ins* on page 386).
- Create VBscript macros and embed VBscript code inside a template for generation (see *GTL Macro Reference* on page 285).

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 *Chapter 1, Working with PowerDesigner Resource Files* 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 328.

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 325.

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

Property or Method	Use
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
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 ActivePackage As BaseObject 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

Type	Global property	Use
	ActiveWorkspace As BaseObject	Retrieves the Application active Work-space
	MetaModel As BaseObject	Retrieves the Application MetaModel
	Models As ObjectSet	Read-only collection that lists opened models
	RepositoryConnection As BaseOb-ject	Retrieves the current repository connec-tion, which is the object that manages the connection to the repository server and then provides access to documents and ob-jects 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 follow-ing way: • It returns True if the application main window is visible and not minimized • False otherwise
	Locked As Boolean	Can be set to True to ensure that the appli-cation 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

Constant	Code	Description
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 (modelkind As Long, filename As String = "", flags As Long =omf_Default) As BaseObject	Creates a new model
CreateModelFromTemplate (filename As String, flags As Long =omf_Default) As BaseObject	Creates a new model using given model file as template
OpenModel (filename As String, flags As Long =omf_Default) As BaseObject	Opens an existing model
Output (message As String = "")	Writes a message in the Script tab of the Output window of PowerDesigner main window
NewPoint (X As Long = 0, Y As Long = 0) As APoint	Creates a point to position a symbol
NewRect (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

Global functions	Use
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 (cmd As String, Optional arglist As String, Optional mode As Long) As String	Opens an external application
Rtf2Ascii (rtf As String) As String	Removes RTF (Rich-Text-File) tags from an RTF formatted text
ConvertToUTF8 (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 file-names must be different
ConvertToUTF16 (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 <OutputFi-leName>. The two filenames must be dif-ferent
EvaluateNamedPath (FileName As String, QueryIfUnknown As Boolean = True, FailOnError As Boolean = False) As String	Replaces a variable in a path by the corre-sponding named path
MapToNamedPath (FileName As String) As String	Replaces the path of a file by the corre-sponding 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/CreateModelFromTemplate
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/CreateModelFromTemplate

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

Global constants	Use
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

Libraries are available for each type of model and for shared PowerDesigner features.

- 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.
- PdEAM - Enterprise Architecture Model
- PdFRM - Free Model
- PdILM - Data Movement Model (the DMM was previously named Information Liquidity Model or ILM, and the PdILM library name has been retained for backwards compatibility)
- PdLDM - Logical Data Model
- PdMTM - Merise Model (available in French only)
- PdOOM - Object Oriented Model
- PdPDM - Physical Data Model
- PdPRJ - Project
- PdRMG - Repository
- PdRQM - Requirements Model
- PdXSM - XML Model
- PdWSP - Workspace

For each library, you can browse a list of:

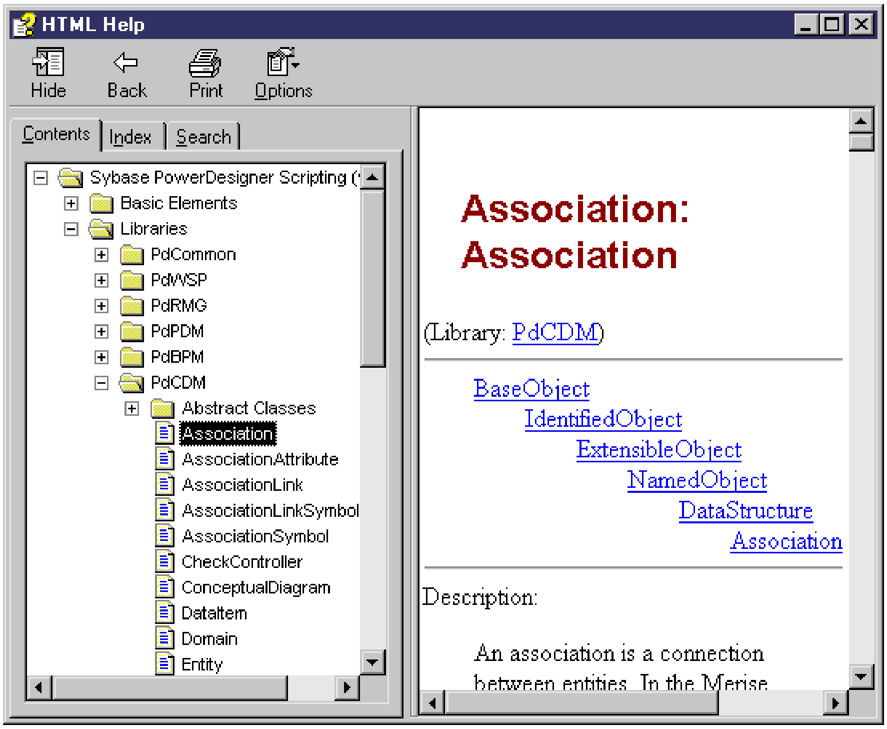
- *Abstract classes* (located in the Abstract Classes expanded node). They are general classes that are used to factorize attributes and behaviors. They are not visible in PowerDesigner. Instantiable classes inherit from abstract classes
- *Instantiable classes* (located directly at the root of each 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 *Chapter 1, Working with PowerDesigner Resource Files* 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.

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:



Association: Association

(Library: PdCDM)

BaseObject
IdentifiedObject
ExtensibleObject
NamedObject
DataStructure
Association

Description:

An association is a connection between entities. In the Merise

You can expand the following nodes from the tree view:

Nodes	What you can find...
Basic Elements	General information on: • Read-only, ordered, and unordered collections (see *Collections* on page 325) • Structured Types (points, rectangles, lists of points) • Global properties (see *Global Properties* on page 328), constants (see *Global Constants* on page 333), and functions (see *Global functions* on page 331)
Libraries	Librairies for common features and for each model (see *Libraries* on page 335)
Appendix	Hierarchical representation of the PowerDesigner metamodel List of constants used to identify objects of each library

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:

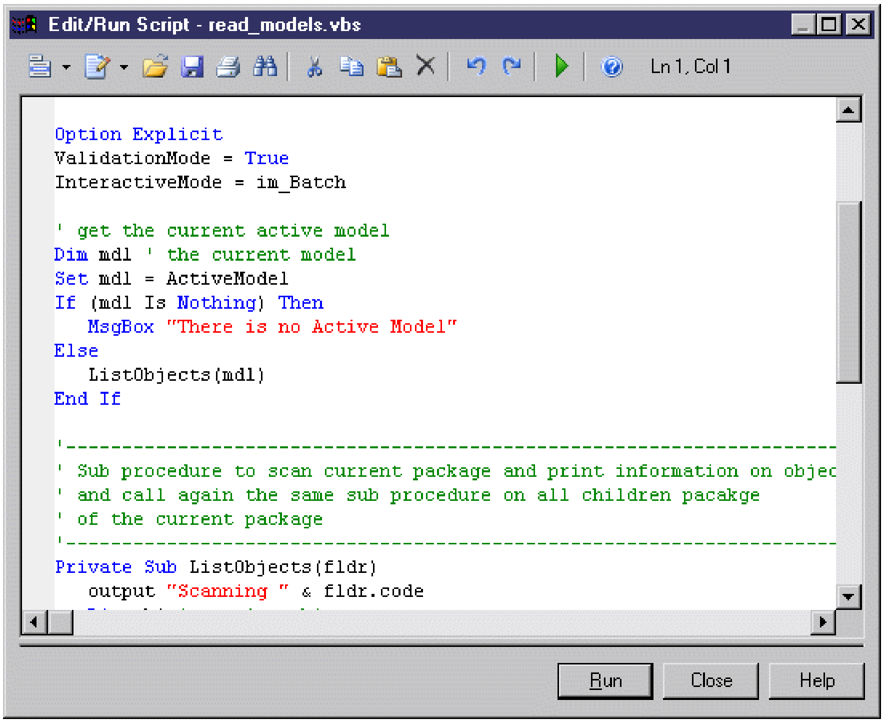
- Properties (Example: Name, Data Type, Transport)
- Read-only, ordered, and unordered collections (Example: Symbols, Columns of a table)
- Methods (Example: Delete (), UpadateNamingOpts())

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:



```
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
   ListObjects(mdl)
End If

'-----------------------------------------------------------------------------
' Sub procedure to scan current package and print information on objec
' and call again the same sub procedure on all children pacakge
' of the current package
'-----------------------------------------------------------------------------
Private Sub ListObjects(fldr)
   output "Scanning " & fldr.code
```

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 wild-cards in the search expression.	**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**
▶	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 *Core Features Guide > The PowerDesigner Interface > Customizing Your Modeling Environment > General Options > Specifying Text Editors*.

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.

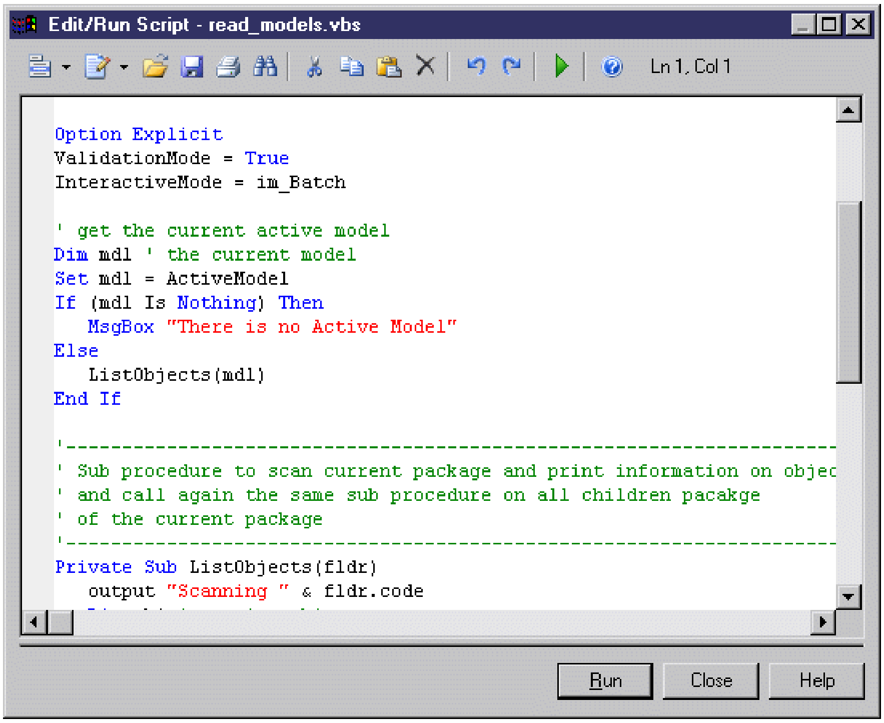
The Edit/Run Script editor lets you:

- 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.



```
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
   ListObjects(mdl)
End If

'-----------------------------------------------------------------------
' Sub procedure to scan current package and print information on objec
' and call again the same sub procedure on all children pacakge
' of the current package
'-----------------------------------------------------------------------
Private Sub ListObjects(fldr)
   output "Scanning " & fldr.code
```

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 386.

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
```

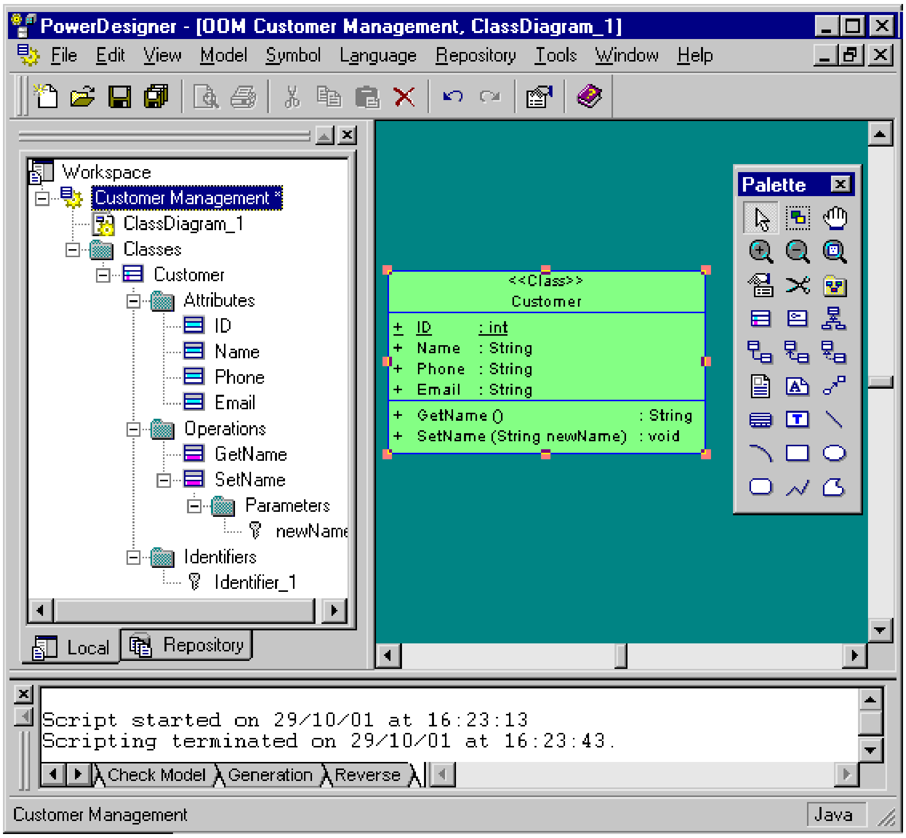
Model Creation Sample

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:



```
Script started on 29/10/01 at 16:23:13
Scripting terminated on 29/10/01 at 16:23:43.
```

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 extension file
- 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 extension
 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 extension"
   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 extension"
    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 extension"
    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
extension 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
extension 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 Extension by Script

Like any other PowerDesigner object, extensions can be read, created and modified using scripting.

For detailed information about extensions, see *Chapter 2, Extension Files* on page 21.

The following script illustrates how you can *access* an existing extension, *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 Extension Properties dialog box.

Example

```
Dim M
Set M = ActiveModel
'Retrieve first extension 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 *Core Features Guide > Linking and Synchronizing Models > Object Mappings*.

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 root using the ChildObjects collection (containing both documents and folders) and any subfolders.

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 Base-Object	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
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 meta-models

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 Merge-Mode 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

Value	Description
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)
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 368.

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 ClassMapping 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
<<notScriptable>> stereotype	Objects that do not support VB scripting have the <<notScriptable>> stereotype	CheckModelInternalMessage 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

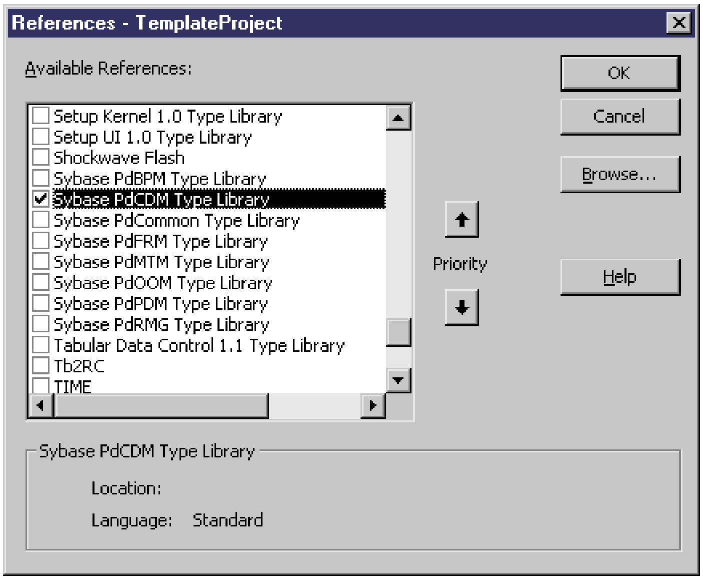
You must add references to the PowerDesigner type libraries you want to use, for example Sybase PdCommon, Sybase PdOOM, Sybase PdPDM, etc. for programs like VB, VBA, VB .NET and C#.

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:



References - TemplateProject

Available References:

- [ ] Setup Kernel 1.0 Type Library
- [ ] Setup UI 1.0 Type Library
- [ ] Shockwave Flash
- [ ] Sybase PdBPM Type Library
- [✓] Sybase PdCDM Type Library
- [ ] Sybase PdCommon Type Library
- [ ] Sybase PdFRM Type Library
- [ ] Sybase PdMTM Type Library
- [ ] Sybase PdOOM Type Library
- [ ] Sybase PdPDM Type Library
- [ ] Sybase PdRMG Type Library
- [ ] Tabular Data Control 1.1 Type Library
- [ ] Tb2RC
- [ ] TIME

Sybase PdCDM Type Library

Location:

Language: Standard

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 387.
- 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 100.
- 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 394.
- 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 396.

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 396.

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.

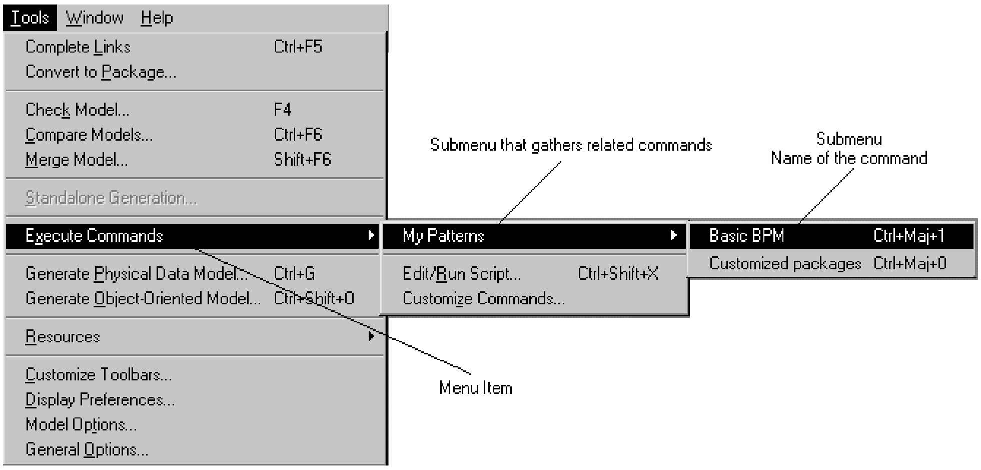
Defining a Customized Command

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)
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

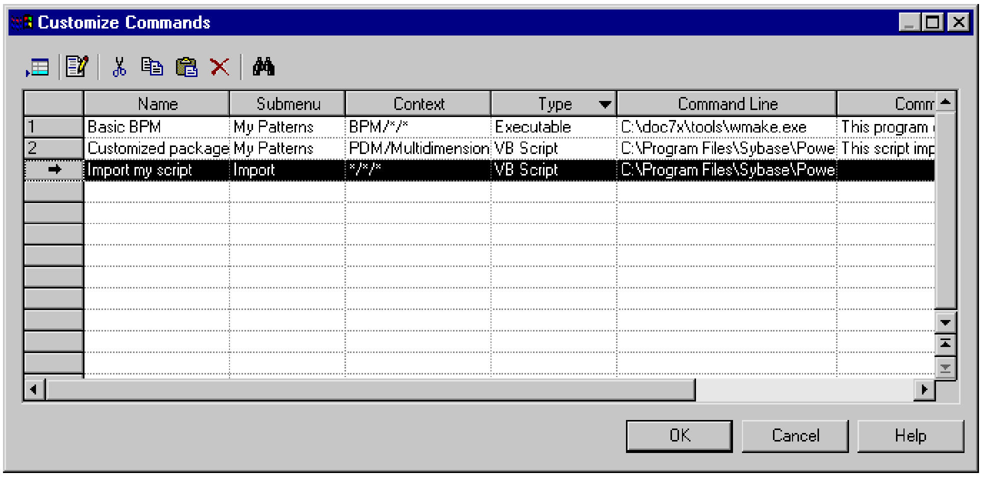
Here are some examples of context definitions as they display in the Context column of the Customize Commands dialog box:

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
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

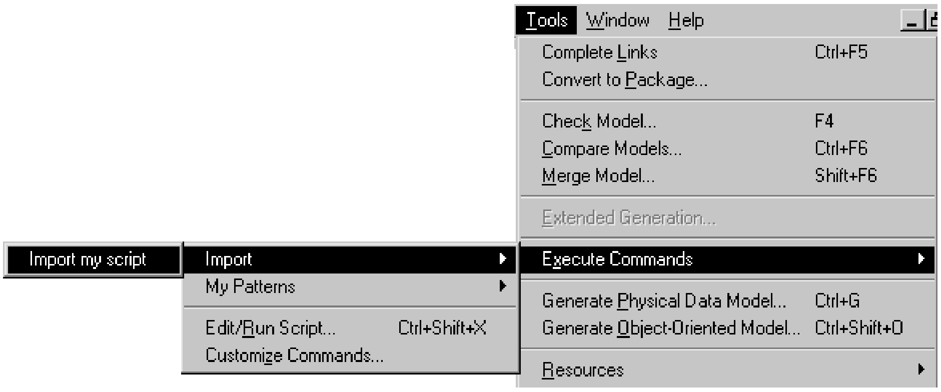
When you select Import or Export in the Submenu list of the Customize Commands dialog box, the command you defined is displayed not only as a submenu entry of the Execute Commands menu item of the Tools menu but also as a submenu entry of the Import or Export menu items of the File menu.

For example you defined the following command in the Customize Commands dialog box:

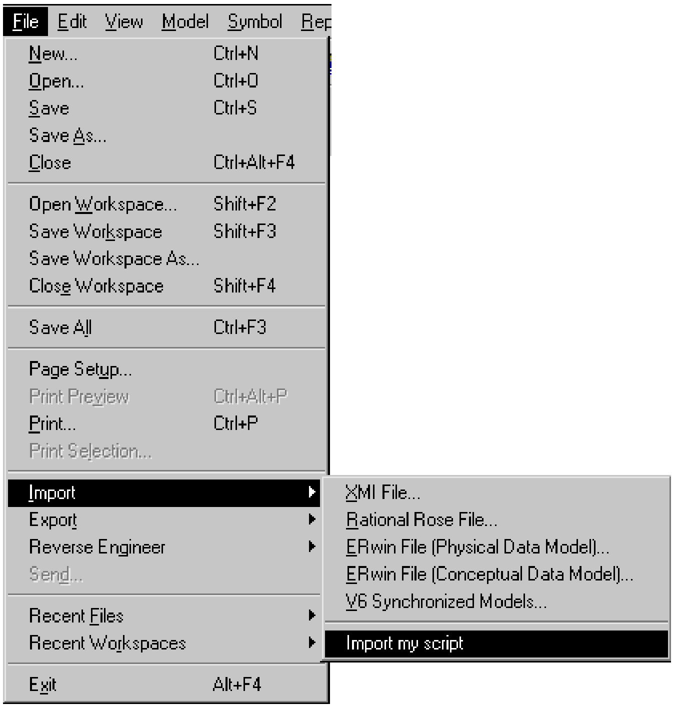


	Name	Submenu	Context	Type	Command Line	Comm
1	Basic BPM	My Patterns	BPM/*/*	Executable	C:\doc7x\tools\wmake.exe	This program
2	Customized package	My Patterns	PDM/Multidimension	VB Script	C:\Program Files\Sybase\Powe	This script imp
➔	Import my script	Import	*/*/*	VB Script	C:\Program Files\Sybase\Powe	

The command is displayed as follows in the Tools menu:



The command is displayed as follows in the File menu:



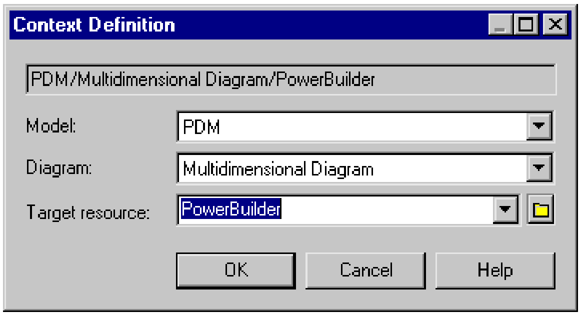
Defining a Customized Command

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.

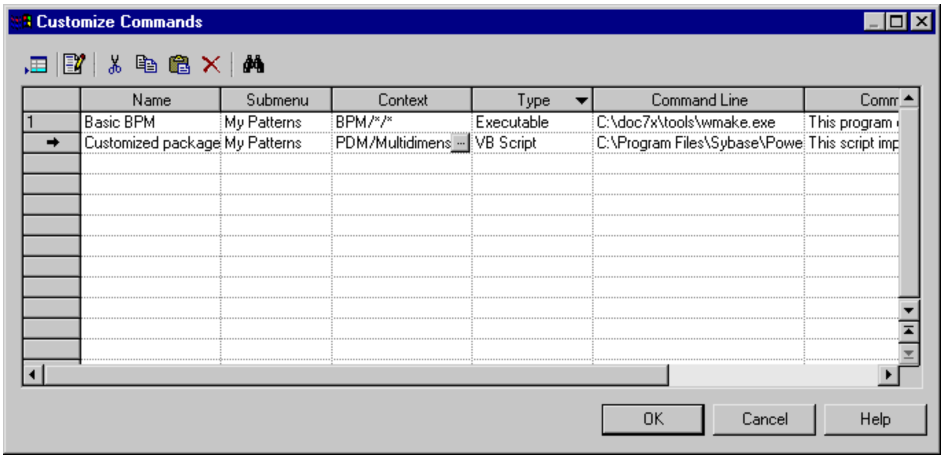


Context Definition

PDM/Multidimensional Diagram/PowerBuilder

Model:	PDM
Diagram:	Multidimensional Diagram
Target resource:	PowerBuilder

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.



Customize Commands

	Name	Submenu	Context	Type	Command Line	Comm
1	Basic BPM	My Patterns	BPM/*/*	Executable	C:\doc7x\tools\wmake.exe	This program
➜	Customized package	My Patterns	PDM/Multidimens	VB Script	C:\Program Files\Sybase\Powe	This script imp

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.

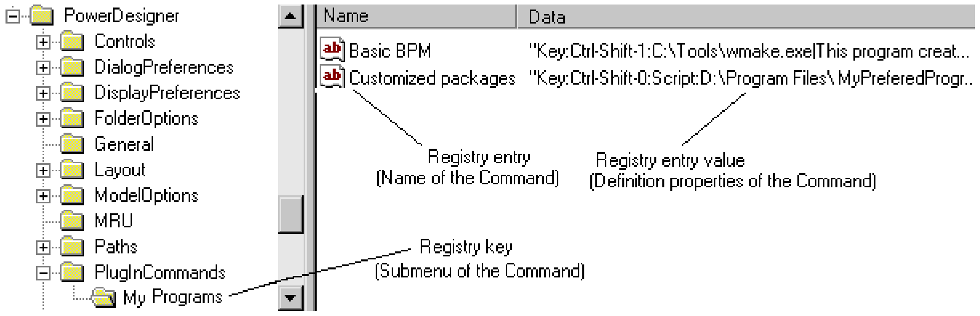
If you define values in the LOCAL MACHINE Registry, customized commands are available for any user of the machine. Thus, when you remove a customized command defined in that Registry from the Customize Commands dialog box, you only remove the line from the list but not the corresponding Registry entry. 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)



Name	Data
Basic BPM	"Key:Ctrl-Shift-1:C:\Tools\wmake.exe|This program creat...
Customized packages	"Key:Ctrl-Shift-0:Script:D:\Program Files\ MyPreferedProgr..

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

Syntax Keyword	Description
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: • **Ctrl-** for CONTROL flag • **Shift-** for SHIFT flag 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.

Note: 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 *Core Features Guide > The PowerDesigner Interface > Customizing Your Modeling Environment > General Options > Managing Add-Ins*).

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 102.

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:

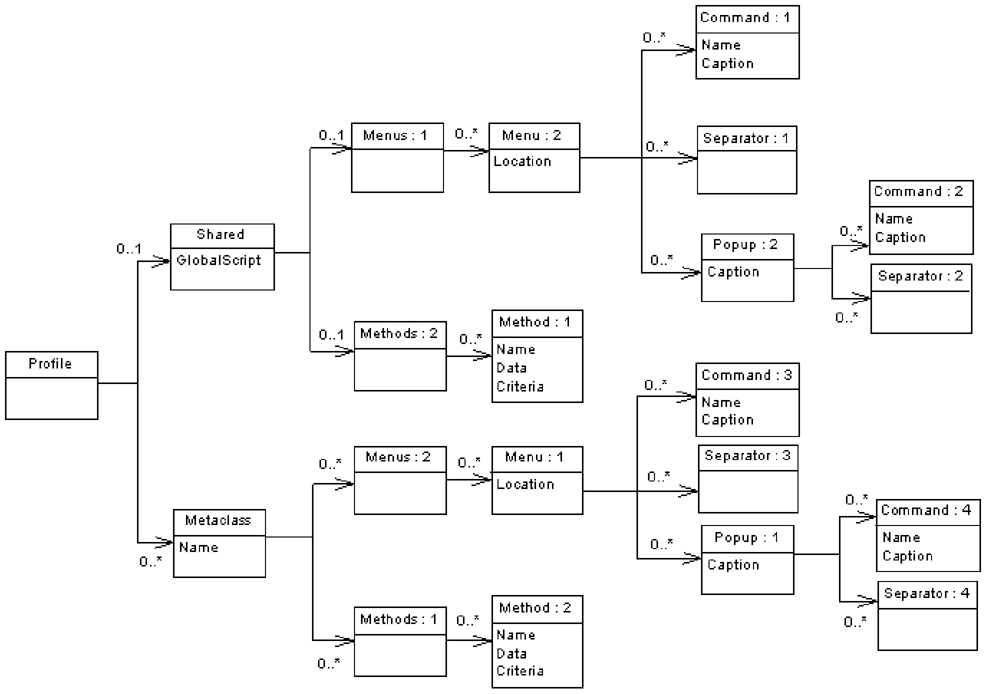
```
DoCommand ("Object", pModel, "CheckIn")
```

Creating an XML File Add-in

You can create your own menu items in PowerDesigner menus by using an XML file.

Note: 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 *Core Features Guide > The PowerDesigner Interface > Customizing Your Modeling Environment > General Options > Managing Add-Ins*).

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