



**Customizing and Extending
PowerDesigner**

PowerDesigner® 16.5

Windows

DOCUMENT ID: DC38628-01-1650-01

LAST REVISED: January 2013

Copyright © 2013 by Sybase, Inc. All rights reserved.

This publication pertains to Sybase software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase trademarks can be viewed at the Sybase trademarks page at <http://www.sybase.com/detail?id=1011207>. Sybase and the marks listed are trademarks of Sybase, Inc. ® indicates registration in the United States of America.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world.

Java and all Java-based marks are trademarks or registered trademarks of Oracle and/or its affiliates in the U.S. and other countries.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

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

Contents

| | |
|---|-----------|
| CHAPTER 1: PowerDesigner Resource Files | 1 |
| Opening Resource Files in the Editor | 3 |
| Navigating and Searching in Resource Files | 5 |
| Editing Resource Files | 6 |
| Saving Changes | 6 |
| Sharing and Embedding Resource Files | 6 |
| Creating and Copying Resource Files | 7 |
| Comparing Resource Files | 7 |
| Merging Resource Files | 8 |
| | |
| CHAPTER 2: Extension Files | 11 |
| Creating an Extension File | 12 |
| Attaching Extensions to a Model | 12 |
| Exporting an Embedded Extension File for Sharing | 14 |
| Extension File Properties | 14 |
| Example: Adding a New Attribute from a Property Sheet | 15 |
| Example: Creating Robustness Diagram Extensions | 16 |
| Creating New Objects with Stereotypes | 18 |
| Specifying Custom Symbols for Robustness Objects ... | 19 |
| Example: Creating Custom Checks on Instance Links | 21 |
| Example: Defining Templates to Extract Message Descriptions | 27 |
| Example: Creating a Generated File for the Message Information | 29 |
| Example: Testing the Robustness Extensions | 30 |
| Metaclasses (Profile) | 33 |
| Extended Objects, Sub-Objects, and Links (Profile) | 36 |

| | |
|---|-----------|
| Stereotypes (Profile) | 37 |
| Promoting a Stereotype to Metaclass Status | 39 |
| Criteria (Profile) | 40 |
| Extended Attributes (Profile) | 41 |
| Creating an Extended Attribute Type | 45 |
| Specifying Icons for Attribute Values | 46 |
| Linking Objects Through Extended Attributes | 48 |
| Extended Collections and Compositions (Profile) | 48 |
| Calculated Collections (Profile) | 50 |
| Dependency Matrices (Profile) | 52 |
| Specifying Advanced Dependencies | 54 |
| Forms (Profile) | 55 |
| Adding Extended Attributes and Other Controls to Your Form | 57 |
| Example: Creating a Property Sheet Tab | 61 |
| Example: Including a Form in a Form | 64 |
| Example: Opening a Dialog from a Property Sheet | 67 |
| Custom Symbols (Profile) | 70 |
| Custom Checks (Profile) | 71 |
| Example: PDM Custom Check | 73 |
| Example: PDM Autofix | 74 |
| Event Handlers (Profile) | 75 |
| Methods (Profile) | 79 |
| Menus (Profile) | 81 |
| Example: Opening a Dialog Box from a Menu | 82 |
| Templates (Profile) | 84 |
| Generated Files (Profile) | 85 |
| Example: JavaGenerated File and Templates | 87 |
| Generating Your Files in a Standard or Extended Generation | 89 |
| Transformations (Profile) | 92 |
| Transformation Profiles (Profile) | 94 |
| Developing Transformation Scripts | 95 |
| XML Imports (Profile) | 96 |
| XML Import Mappings | 97 |

| | |
|---|------------|
| Metamodel Mapping Properties | 100 |
| Metamodel Object Properties | 101 |
| Object Generations (Profile) | 102 |
| Model-to-Model Generation Mappings | 103 |
| Global Script (Profile) | 105 |
| | |
| CHAPTER 3: Object, Process, and XML Language | |
| Definition Files | 107 |
| Settings Category: Process Language | 109 |
| Settings Category: Object Language | 110 |
| Settings Category: XML Language | 112 |
| Generation Category | 112 |
| Example: Adding a Generation Option | 113 |
| Example: Adding a Generation Command and Task .. | 114 |
| Profile Category (Definition Files) | 117 |
| | |
| CHAPTER 4: DBMS Definition Files | 119 |
| Triggers Templates, Trigger Template Items, and Procedure Templates | 120 |
| Database Generation and Reverse Engineering | 120 |
| Script Generation | 121 |
| Extending Generation with Before and After Statements | 122 |
| Script Reverse Engineering | 124 |
| Live Database Generation | 125 |
| Live Database Reverse Engineering | 125 |
| Creating Queries to Retrieve Additional Attributes | 128 |
| Calling Sub-Queries with the EX Keyword | 128 |
| Live Database Reverse Engineering Physical Options | 129 |
| Live Database Reverse Engineering Function- based Index | 131 |
| Live Database Reverse Engineering Qualifiers . | 132 |

| | |
|--|------------|
| Generating and Reverse-Engineering PDM Extended Objects | 133 |
| Adding Scripts Before or After Generation and Reverse Engineering | 133 |
| General Category (DBMS) | 134 |
| Script/Sql Category (DBMS) | 135 |
| Syntax Category | 135 |
| Format Category | 136 |
| Date and Time Format | 137 |
| File Category | 138 |
| Keywords Category | 140 |
| Script/Objects Category (DBMS) | 142 |
| Common Object Items | 144 |
| Table | 148 |
| Column | 152 |
| Working with Null Values | 159 |
| Index | 160 |
| Pkey | 163 |
| Key | 164 |
| Reference | 166 |
| View | 169 |
| Tablespace | 171 |
| Storage | 171 |
| Database | 172 |
| Domain | 173 |
| Abstract Data Type | 174 |
| Abstract Data Type Attribute | 176 |
| User | 177 |
| Rule | 177 |
| Procedure | 180 |
| Trigger | 181 |
| DBMS Trigger | 184 |
| Join Index | 185 |
| Qualifier | 185 |
| Sequence | 186 |

| | |
|--|------------|
| Synonym | 187 |
| Group | 187 |
| Role | 188 |
| DB Package | 189 |
| DB Package Sub-objects | 190 |
| Parameter | 191 |
| Privilege | 191 |
| Permission | 192 |
| Default | 193 |
| Web Service and Web Operation | 194 |
| Web Parameter | 195 |
| Result Column | 195 |
| Dimension | 196 |
| Extended Object | 197 |
| Script/Data Type Category (DBMS) | 197 |
| Profile Category (DBMS) | 200 |
| Using Extended Attributes During Generation | 200 |
| Modifying the Estimate Database Size Mechanism | 202 |
| Calling the GetEstimatedSize Event Handler on Another Metaclass | 205 |
| Formatting the Database Size Estimation Output | 205 |
| ODBC Category (DBMS) | 206 |
| Physical Options (DBMS) | 207 |
| Simple Physical Options | 207 |
| Composite Physical Options | 209 |
| Adding DBMS Physical Options to Your Forms | 210 |
| PDM Variables and Macros | 212 |
| Testing Variable Values with the [] Operators | 213 |
| Formatting Variable Values | 215 |
| Variables for Tables and Views | 216 |
| Variables for Columns, Domains, and Constraints | 217 |
| Variables for Keys | 219 |
| Variables for Indexes and Index Columns | 220 |
| Variables for References and Reference Columns | 220 |

| | |
|--|-----|
| Variables for Triggers and Procedures | 222 |
| Variables for Rules | 223 |
| Variables for Sequences | 224 |
| Variables for Synonyms | 224 |
| Variables for Tablespaces and Storages | 224 |
| Variables for Abstract Data Types | 224 |
| Variables for Join Indexes (IQ) | 227 |
| Variables for ASE & SQL Server | 227 |
| Variables for Database Synchronization | 227 |
| Variables for DB Packages and Their Child Objects .. | 228 |
| Variables for Database Security | 230 |
| Variables for Defaults | 231 |
| Variables for Web Services | 231 |
| Variables for Dimensions | 232 |
| Variables for Extended Objects | 233 |
| Variables for Reverse Engineering | 234 |
| Variables for Database, Triggers, and Procedures | |
| Generation | 234 |
| .AKCOLN, .FKCOLN, and .PKCOLN Macros | 235 |
| .ALLCOL Macro | 236 |
| .DEFINE Macro | 236 |
| .DEFINEIF Macro | 237 |
| .ERROR Macro | 237 |
| .FOREACH_CHILD Macro | 238 |
| .FOREACH_COLUMN Macro | 239 |
| .FOREACH_PARENT Macro | 240 |
| .INCOLN Macro | 240 |
| .JOIN Macro | 241 |
| .NMFCOL Macro | 242 |
| .CLIENTEXPRESSION and .SERVEREXPRESSION | |
| Macros | 242 |
| .SQLXML Macro | 243 |

CHAPTER 5: Customizing Generation with GTL245

| | |
|--|------------|
| Creating a Template and a Generated File | 245 |
| Extracting Object Properties | 246 |
| Accessing Collections of Sub-Objects or Related Objects | 247 |
| Formatting Your Output | 248 |
| Controlling Line Breaks in Head and Tail Strings | 250 |
| Conditional Blocks | 250 |
| Accessing Global Variables | 251 |
| GTL Operators | 252 |
| Translation Scope | 255 |
| Shortcut Translation | 256 |
| Escape Sequences | 256 |
| Calling Templates | 257 |
| Inheritance and Polymorphism | 257 |
| Passing Parameters to a Template | 260 |
| Recursive Templates | 262 |
| GTL-Specific Metamodel Extensions | 262 |
| GTL Macro Reference | 264 |
| .abort_command Macro | 265 |
| .block Macro | 265 |
| .bool Macro | 266 |
| .break Macro | 266 |
| .change_dir and .create_path Macros | 266 |
| .comment and .// Macro | 267 |
| .convert_name and .convert_code Macros | 267 |
| .delete and .replace Macros | 268 |
| .error and .warning Macros | 269 |
| .execute_command Macro | 269 |
| .execute_vbscript Macro | 270 |
| .foreach_item Macro | 271 |
| .foreach_line Macro | 273 |
| .foreach_part Macro | 274 |
| .if Macro | 275 |
| .log Macro | 277 |
| .lowercase and .uppercase Macros | 277 |

- .object and .collection Macros278
- .set_interactive_mode Macro278
- .set_object, .set_value, and .unset Macros279
- .unique Macro281
- .vbscript Macro281
- GTL Syntax and Translation Errors283**

CHAPTER 6: Translating Reports with Report Language Files287

- Opening a Report Language File288**
- Creating a Report Language File for a New Language . .289**
- Report Language File Properties290**
 - Values Mapping Category291
 - Example: Creating a Mapping Table, and Attaching It to a Specific Model Object292
 - Report Titles Category294
 - Example: Translating the HTML Report Previous Button295
 - All Report Titles Tab297
 - Object Attributes Category298
 - All Classes Tab299
 - All Attributes and Collections Tab300
 - Profile/Linguistic Variables Category300
 - Profile/Report Item Templates Category303

CHAPTER 7: Scripting PowerDesigner305

- Running Scripts in PowerDesigner307**
 - VBScript File Samples309
- Manipulating Models, Collections, and Objects (Scripting)312**
 - Creating and Opening Models (Scripting)313
 - Browsing and Modifying Collections (Scripting)314
 - Accessing and Modifying Objects and Properties (Scripting)316

| | |
|--|------------|
| Creating Objects (Scripting) | 318 |
| Displaying, Formatting, and Positioning Symbols (Scripting) | 320 |
| Deleting Objects (Scripting) | 321 |
| Creating an Object Selection (Scripting) | 321 |
| Controlling the Workspace (Scripting) | 322 |
| Creating Shortcuts (Scripting) | 323 |
| Creating Mappings Between Objects (Scripting) | 324 |
| Creating and Generating Reports (Scripting) | 325 |
| Manipulating the Repository (Scripting) | 325 |
| Generating a Database (Scripting) | 326 |
| Reverse Engineering a Database (Scripting) | 328 |
| Creating and Accessing Extensions (Scripting) | 329 |
| Accessing Metadata (Scripting) | 330 |
| OLE Automation and Add-Ins | 332 |
| Creating an ActiveX Add-in | 334 |
| Creating an XML File Add-in | 335 |
| Launching Scripts and Add-Ins from Menus | 338 |
| Adding Commands to the Tools Menu | 339 |
| | |
| CHAPTER 8: The PowerDesigner Public Metamodel | 343 |
| Navigating in the Metamodel | 344 |
| Using the Metamodel Objects Help File | 347 |
| PowerDesigner Model File Format | 348 |
| Example: Simple OOM XML File | 351 |
| | |
| Index | 355 |

Contents

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, based on or extending the PowerDesigner public metamodel are provided:









- *Definition file*: customize the metamodel to define the objects available for a specific DBMS or language:
 - *DBMS definition files* (.xdb) - define a specific DBMS in the PDM (see *Chapter 4, DBMS Definition Files* on page 119).
 - *Process, object, and XML language definition files* (.xpl, .xol, and .xsl) – define a specific language in the BPM, OOM, or XSM (see *Chapter 3, Object, Process, and XML Language Definition Files* on page 107).
- *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 11).
- *Report templates* (.rtp) - specify the structure of a report. Editable within the Report Template Editor (see *Core Features Guide > Storing, Sharing and Reporting on Models > Reports*).
- *Report language files* (.xrl) – translate the headings and other standard text in a report (see *Chapter 6, Translating Reports with Report Language Files* on page 287).
- *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 > Using Profiles to Control the PowerDesigner Interface*).
- *User profiles* (.upf) - store preferences for model options, general options, display preferences, etc (see *Core Features Guide > Modeling with PowerDesigner > 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 > Modeling with PowerDesigner > Objects > Naming Conventions*).



CHAPTER 1: PowerDesigner Resource Files

You can review all the available resource files from the lists of resource files, available by selecting **Tools > Resources > type**.

Note: To comply with recent Microsoft recommendations, PowerDesigner no longer allows you to save resource files inside the Program Files folder, and will propose an alternative location if you try to do so, adding the selected directory to the list of paths for that type of resource file. Resource files previously saved inside Program Files may no longer be available, as Windows Vista or Windows 7 actually store them in a virtual mirror at, for example, C:\Users\username\AppData\Local\VirtualStore\Program Files\Sybase\PowerDesigner 16\Resource Files\DBMS. To restore these files to your lists, optionally move them to a more convenient path, and add their location to your list using the **Path** tool.

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 <i>Creating and Copying Resource Files</i> on page 7). |
|  | Save - Saves the selected resource file. |
|  | Save All - Saves all the resource files in the list. |
|  | <p>Path - Specifies the directories that contain resource files to populate this list and other places in the PowerDesigner interface where resources of this type can be selected. By default, only the directory containing the resource files delivered in the PowerDesigner installation is specified, but you can add as many additional directories as necessary.</p> <p>If you plan to modify the delivered resource files or create your own, we recommend that you store these files in a directory outside the PowerDesigner installation directory.</p> <p>The root of the library belonging to your most recent repository connection is implicitly included at the head of the list, and is scanned recursively (see <i>Core Features Guide > Administering PowerDesigner > Deploying an Enterprise Glossary and Library</i>).</p> <hr/> <p>Note: In rare cases, when seeking resource files to resolve broken references in models, the directories in the list are scanned in order, and the first matching instance of the required resource is used.</p> |
|  | Compare - Selects two resource files for comparison. |
|  | Merge - Selects two resource files for merging. |
|  | 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 <i>Core Features Guide > Administering PowerDesigner > Deploying an Enterprise Glossary and Library</i> . |

| Tool | Description |
|---|--|
|  | 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:

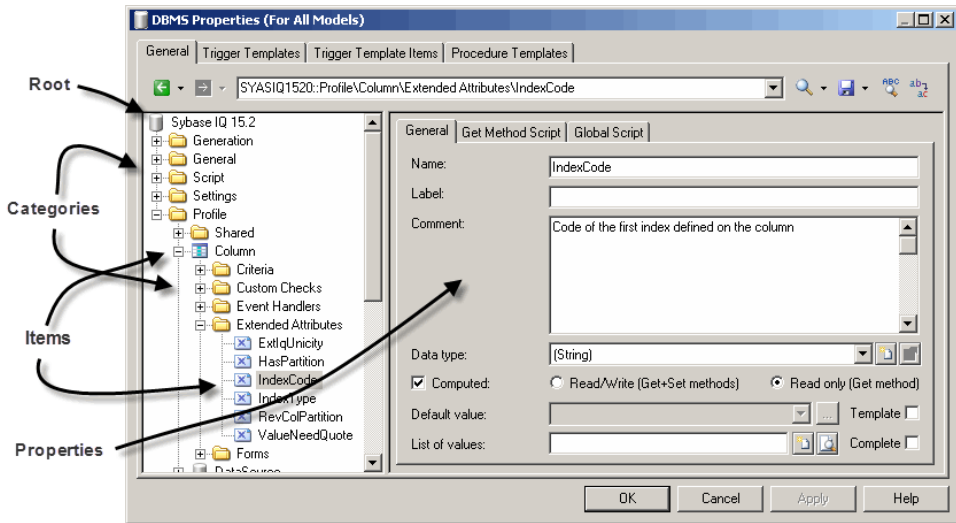
- In a PDM, select **Database > Edit Current DBMS**.
- In a BPM, select **Language > Edit Current Process Language**.
- In an OOM, select **Language > Edit Current Object Language**.
- In an XSM, select **Language > Edit Current Language**.

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:

CHAPTER 1: PowerDesigner Resource Files



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 7).

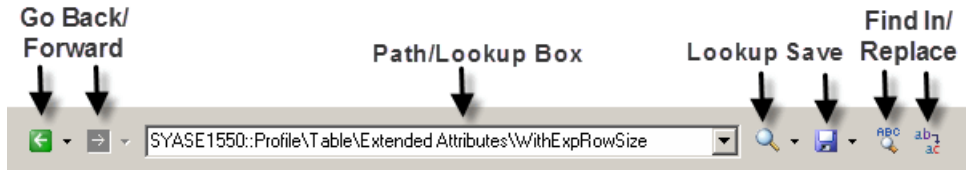
Each entry is a part of the definition of a resource file, and entries are organized into logical categories. For example, the Script category in a DBMS language file collects together all the entries relating to database generation and reverse engineering.

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: <ul style="list-style-type: none"> [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 <code>is*</code> to retrieve all extensions called <code>is...</code> 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 another template (see *Templates (Profile)* on page 84) or other extension, place your cursor between the percent signs and press **F12**. If an extension overrides another item, right-click it and select **Go to super-definition** to go to the overridden item.

Editing Resource Files

You can add items in the resource editor by right-click a category or an entry in the tree view.

The following editing options are available:

| 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 12.

Comparing Resource Files

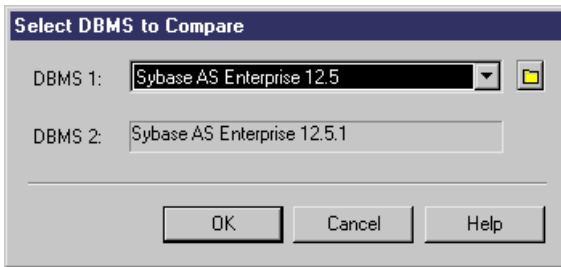
You can select two resource files and compare them to highlight the differences between them.

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.



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 > Modeling with PowerDesigner > 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

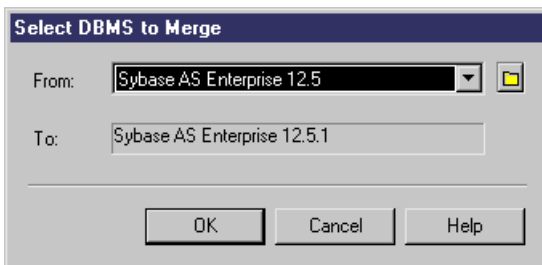
You can select two resource files of the same kind and merge them. Merge is performed from left to right, the resource file in the right pane is compared to the resource file in the left pane, differences are highlighted and merge actions are proposed in the right hand resource file.

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.



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 > Modeling with PowerDesigner > Comparing and Merging Models*.

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

CHAPTER 2 **Extension Files**

Extensions files (* .xem) allow you to customize and extend the PowerDesigner metamodel to support your exact modeling needs. You can define additional properties for existing objects or specify entirely new types of objects, modify the PowerDesigner interface (reorganizing and adding property sheet tabs, Toolbox tools and menu items), and define additional generation targets and options.

Extension files have an .xem extension and are located in `install_dir/Resource Files/Extended Model Definitions`. To view the list of extensions, select **Tools > Resources > Extensions > model type**. For information about the tools available in resource file lists, see *Chapter 1, PowerDesigner Resource Files* on page 1.

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 112.
- *Profile* - used for extending the metaclasses in the PowerDesigner metamodel. You can:
 - Create or sub-classify new kinds of objects:
 - Metaclasses – drawn from the metamodel as a basis for extension.
 - Stereotypes [for metaclasses and stereotypes only] – sub-classify metaclasses by stereotype.
 - Criteria – sub-classify metaclasses by evaluating conditions.
 - Add new properties and collections to objects and display them:
 - Extended attributes – to add metadata.
 - Extended collections and compositions – to enable manual linking between objects.
 - Calculated collections – to automate linking between objects.
 - Dependency matrices – to show dependencies between two types of objects.
 - Forms – to modify property sheets and add custom dialogs.
 - Custom symbols – to change the appearance of objects in diagrams.
 - Add constraints and validation rules to objects:
 - Custom checks – to test the validity of your models on demand
 - Event handlers – to perform validation or invoke methods automatically.
 - Execute commands on objects:
 - Methods – VBScripts to be invoked by menus or form buttons.
 - Menus [for metaclasses and stereotypes only] – to add commands to PowerDesigner menus.
 - Generate objects in new ways:

- Templates – to extract text from object properties.
- Generated Files - to assemble templates for preview and generation of files
- Transformations – to automate changes to objects at generation or on demand.
- Map correspondences between different metamodels:
 - Object generations - to define mappings between different modules in the PowerDesigner metamodel for model-to-model generation.
 - XML imports - to define mappings between an XML schema and a PowerDesigner module to import XML files as models.

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 of Extensions.

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 7.

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 14.

Attaching Extensions to a Model

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.

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

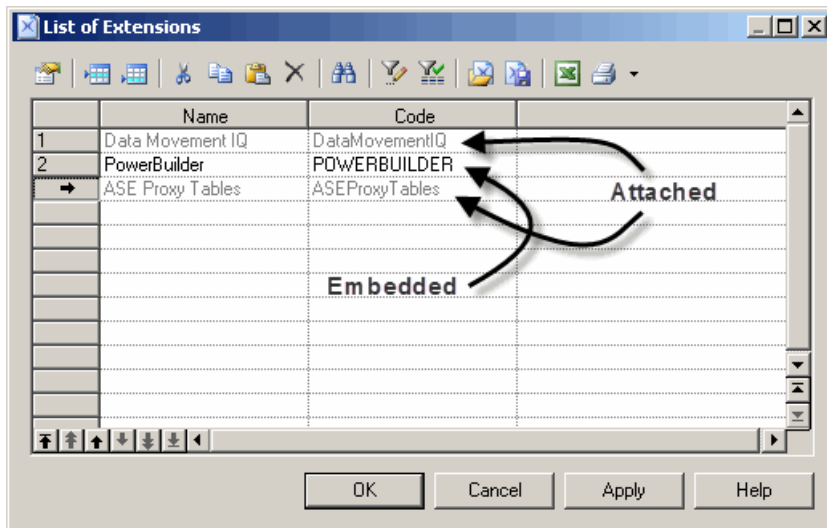
name for the new file, and then select the .xem that you want to modify in the **Copy from** field.

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.

1. Select **Model > Extensions** to open the List of Extensions.
2. Click the **Attach an Extension** 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.



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

Note: If you embed an extension file 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 can now be accessed by and attached to or embedded in other models.

Extension File Properties

All extension files have the same basic category structure.

The root node of each file contains the following properties:

| Property | Description |
|---------------------|---|
| Name / Code | Specify the name and code of the extension file, which must be unique in a model. |
| 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 / Sub-family | Restricts the availability of the XEM to a particular target family and subfamily. For example, when an XEM has the family Java, it is available only for use with targets in the Java object language family. 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. |
| Enable Trace Mode | Lets you preview the templates used during generation (see <i>Templates (Profile)</i> on page 84). Before starting the generation, click the Preview page of the relevant object, and click the Refresh tool to display the templates. When you double-click on a trace line from the Preview page, the Resource Editor opens to the corresponding template definition. |

| Property | Description |
|--------------------------------|--|
| Complement language generation | Specifies that the XEM is used to complement the generation of an DBMS or language definition, so that items to be generated for the language are merged with those of the XEM before generation, and all generated files specified by both the language definition and the XEMs are generated (see <i>Generated Files (Profile)</i> on page 85). If two generated files have identical names, the file in the XEM overrides the one defined in the target. Note: PowerBuilder does not support XEMs for complementary generation. |
| Comment | Provides a descriptive comment for the XEM. |

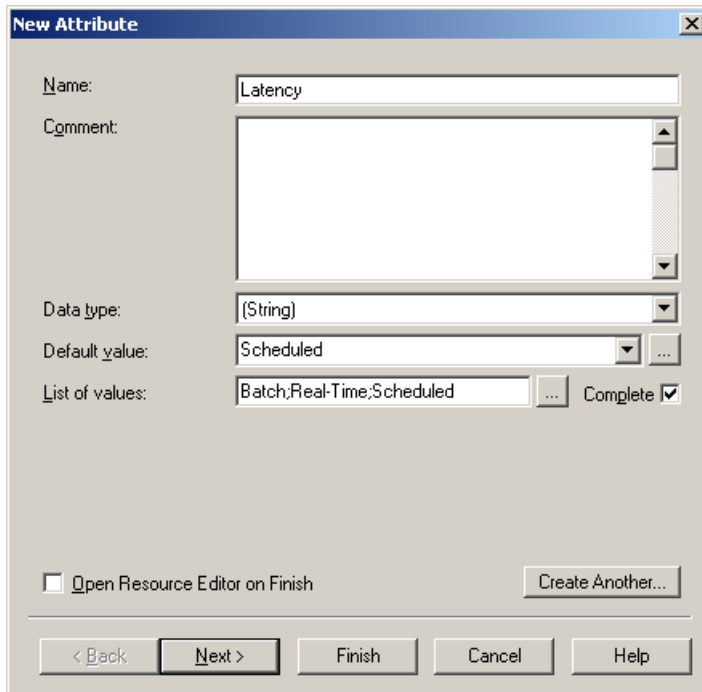
The following categories are also available:

- Generation - Contains Generation commands, options, and tasks to define and activate a generation process (see *Generation Category* on page 112).
- Transformation Profile - Groups transformations for application at model generation time or on demand (see *Transformations (Profile)* on page 92).

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.

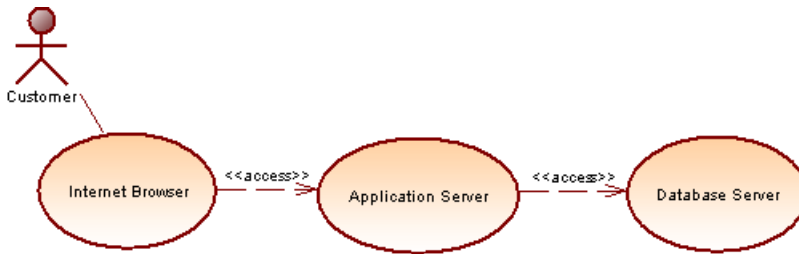


Example: Creating Robustness Diagram Extensions

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.

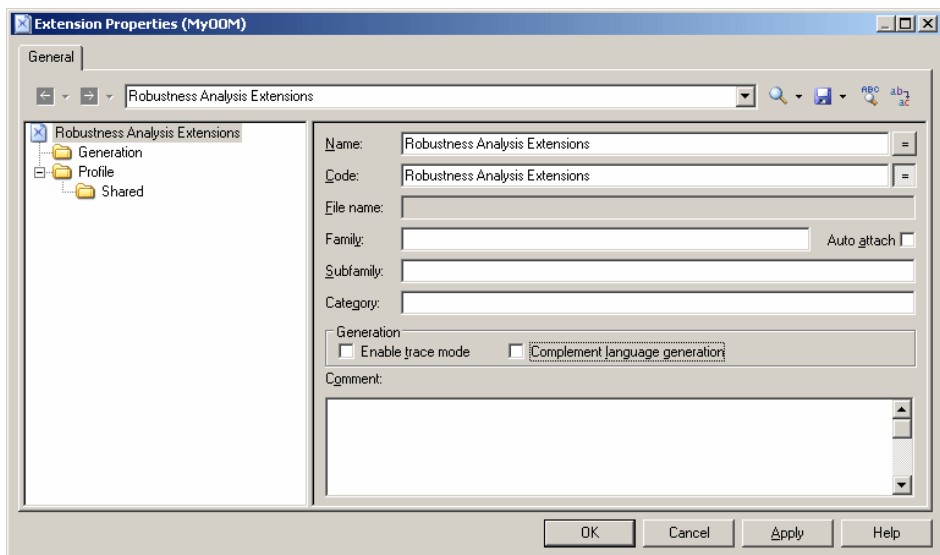
Creating the robustness extensions will enable us to verify use cases like the following, which represents a basic Web transaction:



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.

The first step in defining extensions, is to create an extension file (.xem) to keep them in:

1. Create or open an OOM and 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:



For detailed information about creating extension files, see *Creating an Extension File* on page 12.

Creating New Objects with Stereotypes

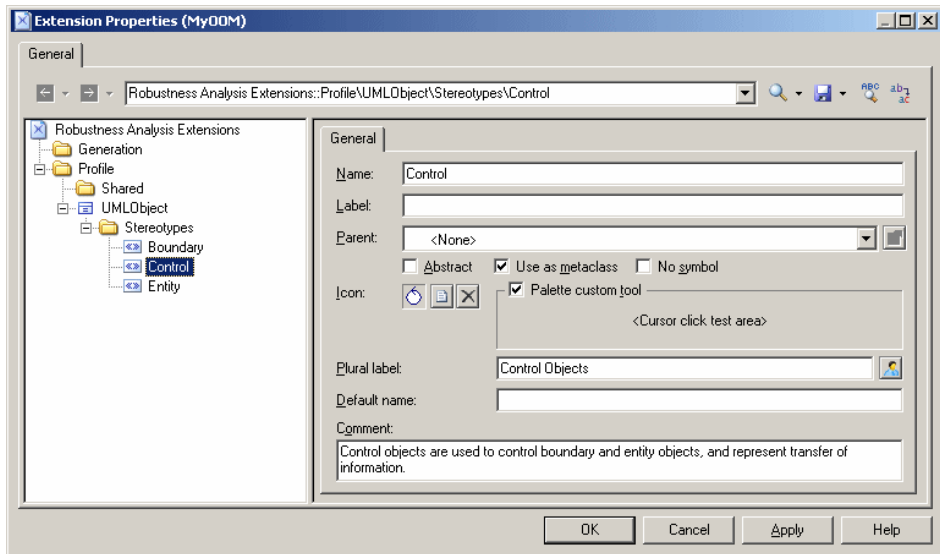
To implement robustness analysis in PowerDesigner, we need to create three new types of objects (boundary, entity, and control objects), which we will define in the Profile category by extending the `UMLObject` metaclass through stereotypes.

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 |



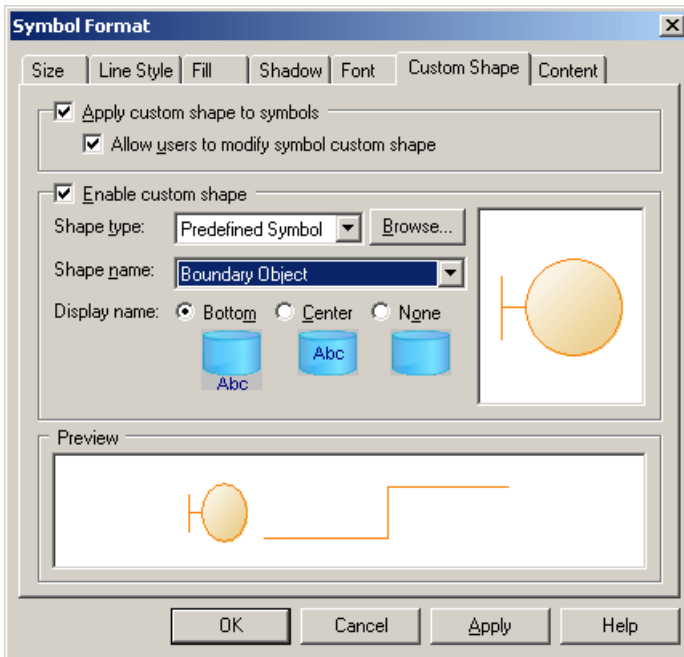
9. Click **Apply** to save your changes before continuing.

For detailed information about creating stereotypes, see *Stereotypes (Profile)* on page 37.

Specifying Custom Symbols for Robustness Objects

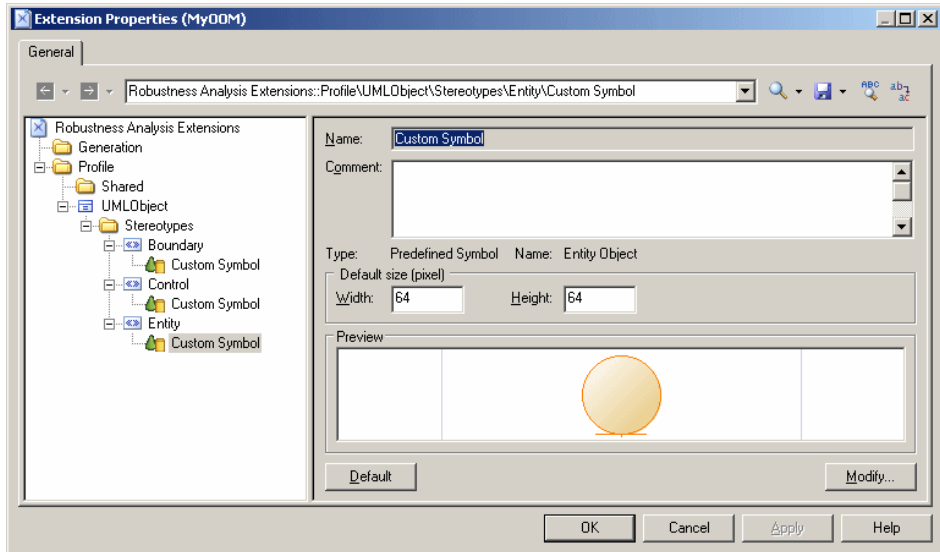
We will specify diagram symbols for each of our new robustness diagram objects by adding custom symbols to our new stereotypes.

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.



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 |



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

For detailed information about creating custom symbols, see *Custom Symbols (Profile)* on page 70.

Example: 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 when you check your model.

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. |
| Help message | This check ensures that actors only communicate with boundary objects. |

| Field | Value |
|------------------------------|---|
| 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 If
End Function
```

Note: For more information on VBScript, see *Chapter 7, Scripting PowerDesigner* on page 305.

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

```

CHAPTER 2: Extension Files

```
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:

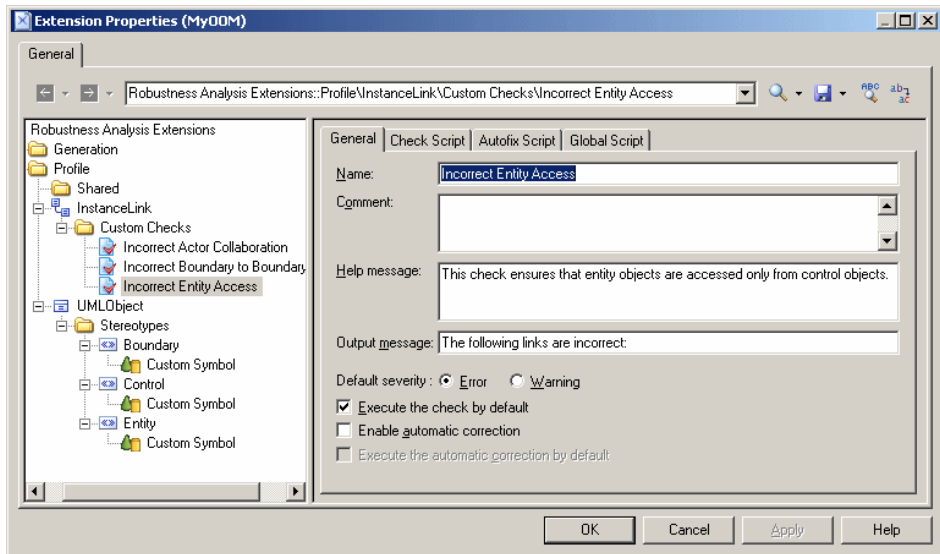
| Field | Value |
|------------------------------|---|
| Name | Incorrect Boundary to Boundary Link |
| Help message | This check ensures that an instance link is not defined between two boundary objects. |
| Output message | The following links between boundary objects are incorrect: |
| Default severity | Error |
| Execute the check by default | [selected] |

| Field | Value |
|--------------|---|
| Check Script | <pre> 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, "Boundary") Then If CompareStereotype(dst, PdOOM.Cls_UMLObject, "Boundary") Then %Check% = False End If End If End Function </pre> |

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 | <pre> 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 If End Function </pre> |



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

For detailed information about creating custom checks, see *Custom Checks (Profile)* on page 71.

Example: 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 `Message` metaclass (to extract the message sequence number, name, sender, and receiver) and the `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%
```

CHAPTER 2: Extension Files

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.

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:

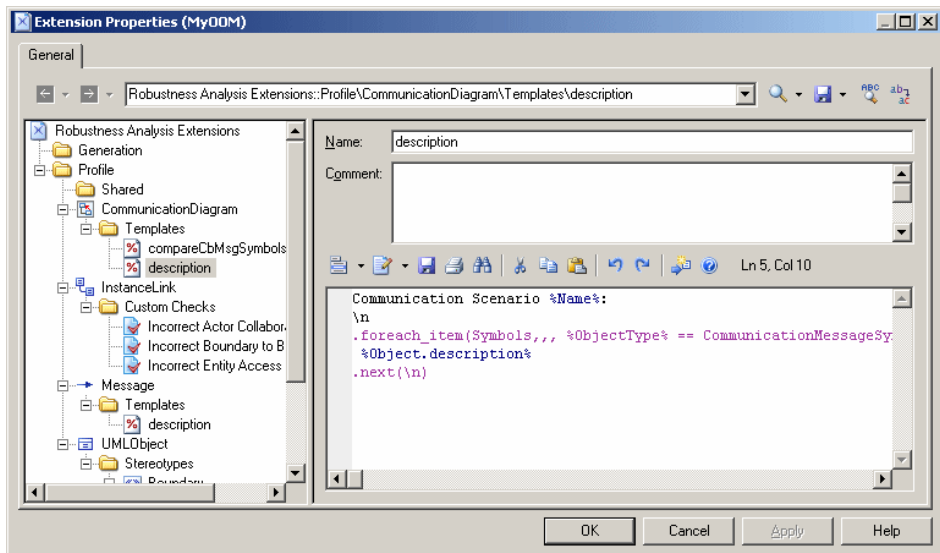
```
.bool (%Item1.Object.SequenceNumber% >=
%Item2.Object.SequenceNumber%)
```

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.



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

For detailed information about templates and GTL, see *Templates (Profile)* on page 84 and *Chapter 5, Customizing Generation with GTL* on page 245.

Example: Creating a Generated File for the Message Information

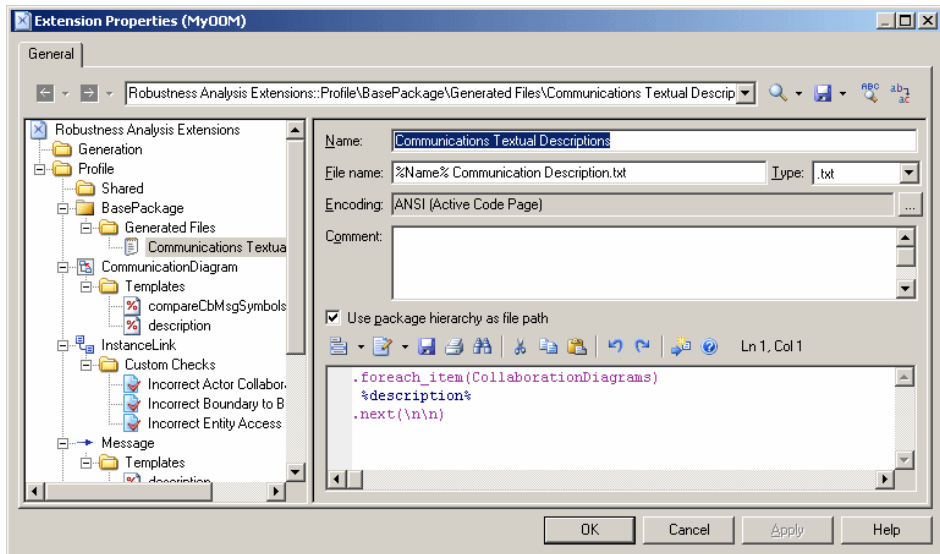
Having created templates to extract information about the messages in the model, we need to create a generated file to contain and display them on the **Preview** tab of the diagram property sheet. We will define the file on the `BasePackage` metaclass, which is the common class for all packages and 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 |
| Use package hierarchy as file path | [unselected] |

5. Enter the following code in the text box:

```
.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.

For detailed information about creating generated files, see *Generated Files (Profile)* on page 85.

Example: Testing the Robustness Extensions

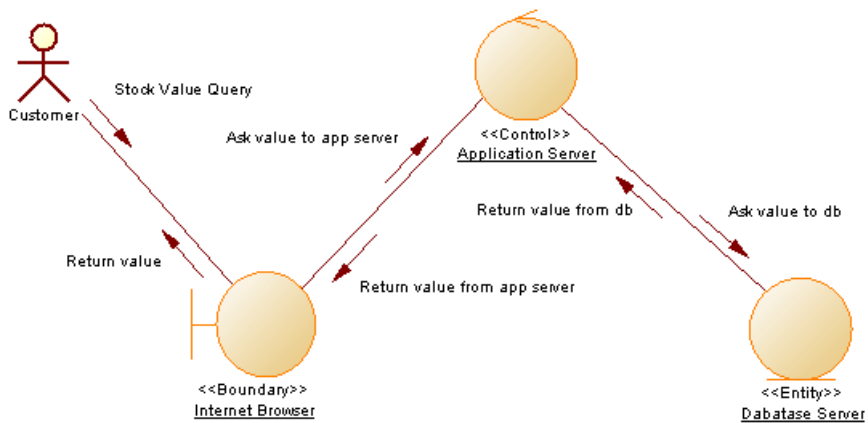
To test the extensions we have created, we will create a small robustness diagram to analyze our use case.

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

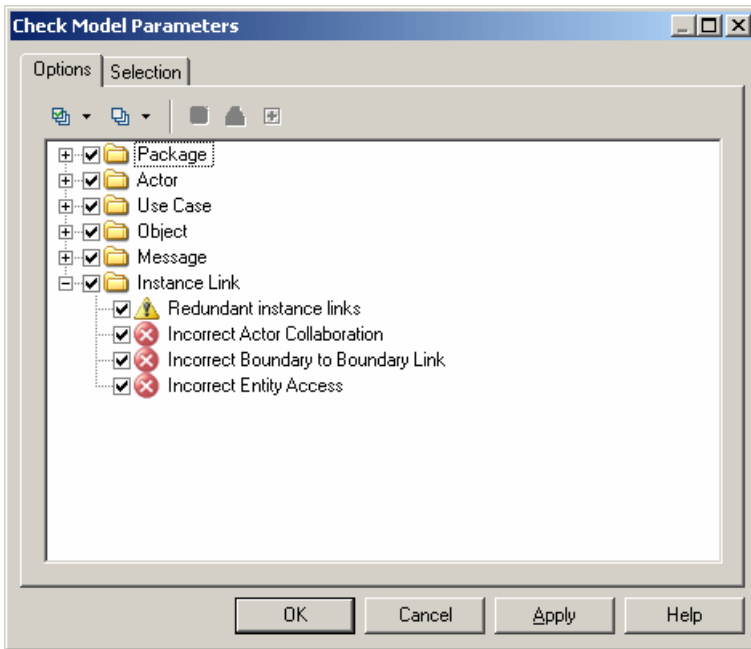
In addition to the standard Toolbox, a custom toolbox is provided with tools you have defined to create boundary, control, and entity objects.

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 |

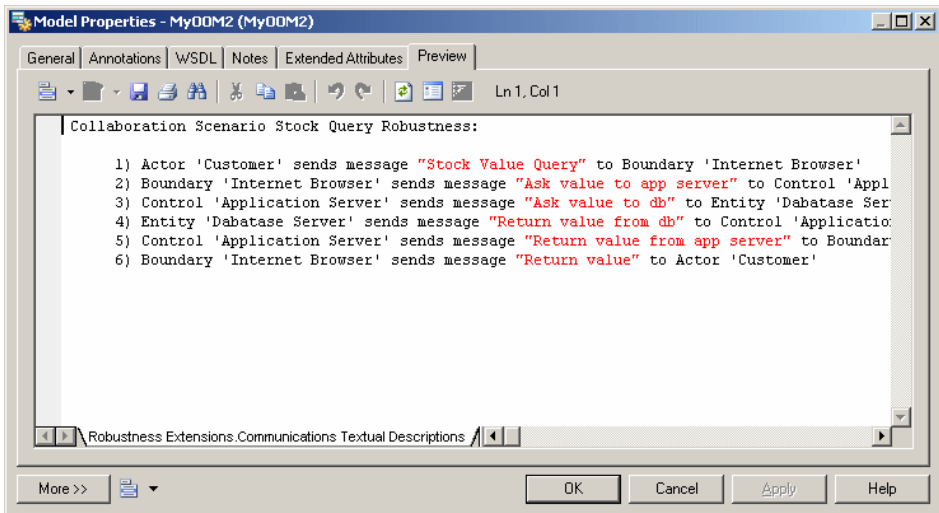


5. Select **Tools > Check Model** to display the Check Model Parameters dialog box, in which the custom checks we have created appear in the Instance Link category:



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:



Metaclasses (Profile)

Metaclasses are defined in the PowerDesigner metamodel and provide the basis for your extensions. You add a metaclass to the Profile category when you want to extend it in some way by modifying its behavior, adding new properties, changing its property sheet or symbol, or even excluding it from your models.

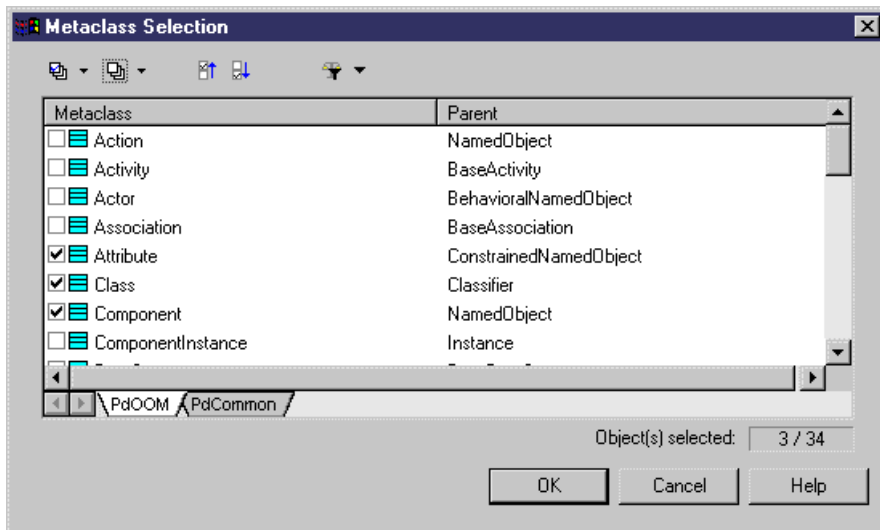
You can either make extensions to an existing type of object or create an entirely new kind of modeling object by adding the `ExtendedObject`, `ExtendedSubObject` or `ExtendedLink` metaclass (see *Extended Objects, Sub-Objects, and Links (Profile)* on page 36).

In the following example, the `FederationController` is an entirely new type of object created by adding the `ExtendedObject` metaclass and defining a stereotype on it. Various specializations of the `Table` metaclass are defined through criteria and stereotypes:

| | |
|--|---|
| | <p>Extensions are inherited, so that any extensions made to a metaclass are available to its stereotyped children, and those that are subject to criteria. The various extended attributes defined under the table metaclass will be available to table instances according to the following rules:</p> <ul style="list-style-type: none"> • <code>SecurityLevel</code> - All tables. • <code>EncryptionKey</code> - Tables for which the <code>SecureTable</code> criterion evaluates to true. • <code>ReplicationPath</code> - Tables for which both the <code>SecureTable</code> and <code>Replicated</code> criteria evaluate to true. • <code>ExternalLogin</code> - Tables bearing either the <code>FederatedTable</code> or <code>PriorityTable</code> stereotype. • <code>Availability</code> - Tables bearing the <code>PriorityTable</code> stereotype. <p>For example, a table bearing the <code>FederatedTable</code> stereotype, and for which the <code>SecureTable</code> criteria evaluates to true, would display the <code>SecurityLevel</code>, <code>EncryptionKey</code>, and <code>ExternalLogin</code> attributes, while a table bearing the <code>PriorityTable</code> stereotype, and for which both the <code>SecureTable</code> and <code>Replicated</code> criteria evaluate to true, would display these attributes and, additionally, the <code>ReplicationPath</code> and <code>Availability</code> attributes.</p> |
|--|---|

1. Right-click the Profile category and select **Add Metaclasses**:

2. Select one or more metaclasses to add to the profile. The sub-tabs list metaclasses belonging to the present module (for example, the OOM), and standard metaclasses belonging to the PdCommon module.

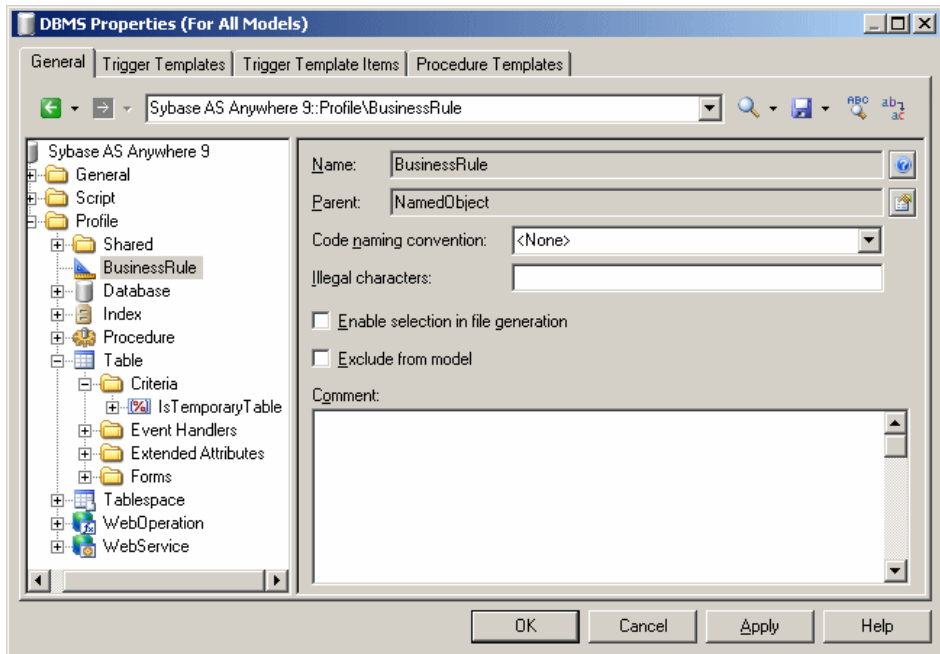


[optional] Use the **Modify Metaclass Filter** tool to display:

- All metaclasses
- Concrete metaclasses - for object types that can be created in a model, such as `Class` or `Interface`.
- Abstract metaclasses - which are never instantiated but are used to define common extensions. For example, add the `Classifier` metaclass to your profile to define extensions that will be inherited by both classes and interfaces.

Note: For information about viewing and navigating among metaclasses in the metamodel, see *Chapter 8, The PowerDesigner Public Metamodel* on page 343.

3. Click **OK** to add the selected metaclasses to your profile:



4. [optional] Enter the following properties as appropriate:

| 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 go to the parent. If the parent is not present in the profile, a message invites you to add it. |
| Code naming convention | <p>[concrete metaclasses in target files] Specifies the default format to initialize the name to code conversion script for instances of the metaclass. The following formats are available:</p> <ul style="list-style-type: none"> • <code>firstLowerWord</code> - First word in lowercase, then other first letters of other words in uppercase • <code>FirstUpperChar</code> - First character of all words in uppercase • <code>lower_case</code> - All words in lowercase and separated by an underscore • <code>UPPER_CASE</code> - All words in uppercase and separated by an underscore <p>For more information on conversion scripts and naming conventions, see <i>Core Features Guide > Modeling with PowerDesigner > Objects > Naming Conventions</i>.</p> |

| Property | Description |
|-------------------------------------|--|
| Illegal characters | <p>[concrete metaclasses in target files] 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:</p> <pre>"/ !=<>" " ' () "</pre> <p>When working with an OOM, this object-specific list overrides any values specified in the <code>IllegalChar</code> property for the object language (see <i>Settings Category: Object Language</i> on page 110).</p> |
| Enable selection in file generation | Specifies that instances of the metaclass will appear in the Selection tab of the extended generation dialog box. |
| Exclude from model | <p>[concrete metaclasses only] Prevents the creation of instances of the metaclass 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, select this check box for the <code>BusinessRule</code> metaclass to hide them in your models.</p> <p>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.</p> |
| Comment | Documents the reason for the presence of the metaclass in this profile. |

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

- Extended objects – define new types of objects that can be created anywhere.
 - Extended sub-objects – define new types of child objects that can only be created in the property sheet of their parent via an extended composition (see *Extended Collections and Compositions (Profile)* on page 48).
 - Extended links – define new types of links between objects.
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 stereotype (see *Stereotypes (Profile)* on page 37 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 39).
4. Click **Apply** to save the changes.

Stereotypes (Profile)

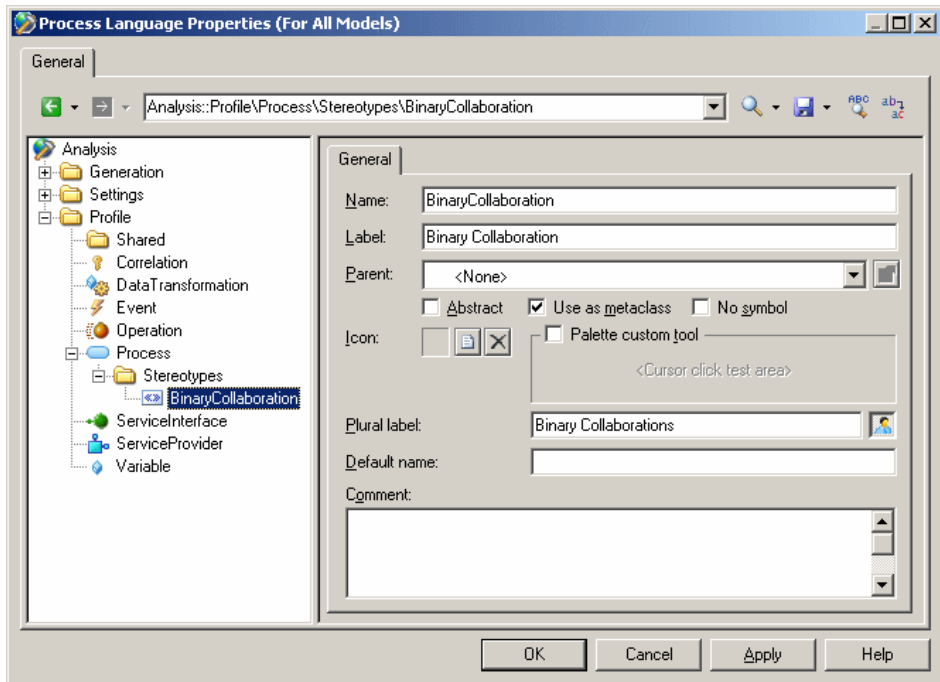
Stereotypes subclassify metaclasses so that extensions are applied to objects only if they bear the stereotype. Stereotypes can be promoted to the status of metaclasses with a specific list, Browser category and custom symbol and Toolbox tool.

Note: You can define more than one stereotype for a given metaclass, but you can only apply a single stereotype to each instance. Like other extensions, stereotypes support *inheritance*, so extensions to a parent stereotype are inherited by child stereotypes.

1. Right-click a metaclass, criterion, or stereotype, and select **New > Stereotype**.
2. Enter the following properties as appropriate:

| Property | Description |
|------------------|---|
| Name | Specifies the internal name of the stereotype, which is 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. Disables the Use as metaclass property. |
| Use as metaclass | Promotes the stereotype to the same status as standard PowerDesigner metaclasses, to give it its own list of objects, Browser category, and its own tab in multi-pane selection boxes such as those used for generation (see <i>Promoting a Stereotype to Metaclass Status</i> on page 39). |
| No Symbol | [available when Use as metaclass is selected] Specifies that instances of the stereotyped metaclass cannot be displayed in a diagram and are visible only in the Browser. Disables the Toolbox custom tool . |

| Property | Description |
|---------------------|---|
| Icon | <p>Specifies an icon for stereotyped instances of the metaclass. Click the tools to the right of this field in order to browse for <code>.cur</code> or <code>.ico</code> files.</p> <hr/> <p>Note: The icon is used to identify objects in the Browser and elsewhere in the interface, but not as a diagram symbol. To specify a custom diagram symbol, see <i>Custom Symbols (Profile)</i> on page 70.</p> |
| Toolbox custom tool | <p>[available for objects supporting symbols] Specifies a Toolbox tool to enable you to create objects in a diagram. If you do not select this option, users are only able to create objects bearing the stereotype from the Browser or Model menu. Custom tools appear in a separate Toolbox group named after the resource file in which they are defined.</p> <hr/> <p>Note: If you have not specified an icon, the tool will use a hammer icon by default.</p> |
| Plural label | <p>[available when Use as metaclass is selected] Specifies the plural form of the display name that will appear in the PowerDesigner interface.</p> |
| Default name | <p>[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.</p> <p>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.</p> |
| Comment | <p>Provides a description or additional information about the stereotype.</p> |



Promoting a Stereotype to Metaclass Status

You can create new types of objects that behave as standard PowerDesigner metaclasses by selecting **Use as Metaclass** in the stereotype property page.

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 promoted to metaclass status.

1. In the Stereotype property page, select **Use as metaclass**.
2. [optional] Specify an icon and tool to create instances of the metaclass stereotype.
3. Click **Apply** to save the changes and then add extended attributes and other appropriate extensions under the stereotype.

In your model, the stereotypes has:

- A separate list in the **Model** menu after the parent metaclass list (and the parent metaclass list will not display objects with the metaclass stereotype). 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.

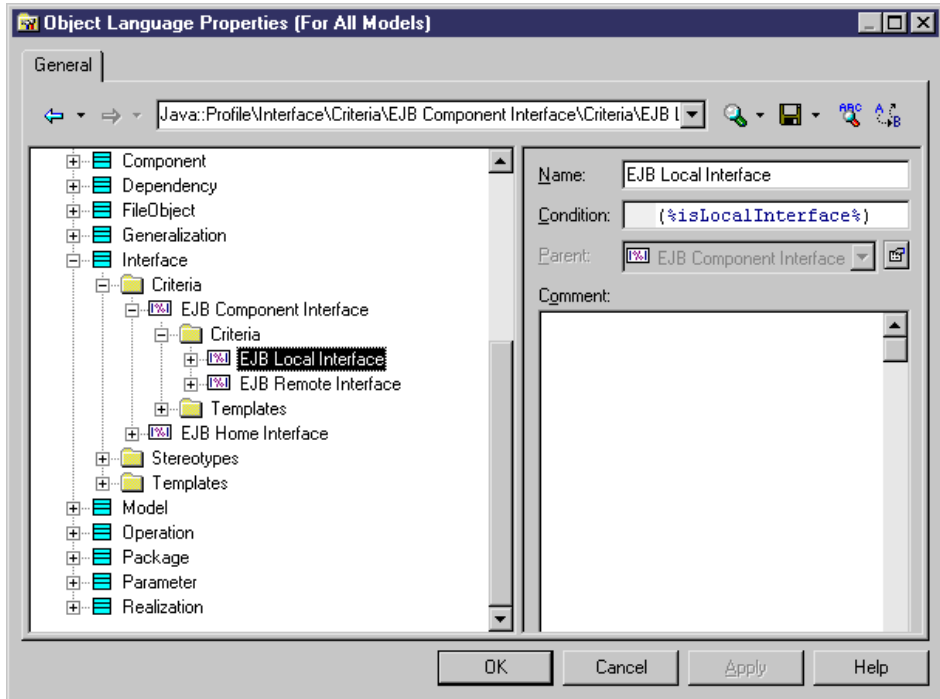
- A separate Browser folder and command under **New**, when you right-click the model or a package.
- Property sheet titles based on the metaclass label.
- Its own tab in multi-pane selection boxes such as those used for generation.

Criteria (Profile)

Criteria subclassify metaclasses so that extensions are applied to objects only if they satisfy conditions. You can test an object instance against multiple criteria, and for sub-criteria, its condition and any conditions specified by its parents must be met for its extensions to be applied to the instance.

1. Right-click a metaclass and select **New > Criterion**.
2. Enter the following properties as appropriate:

| Property | Description |
|-----------|---|
| Name | Specifies the name of the criterion. |
| Condition | <p>Specifies the condition which instances must meet in order to access the criterion extensions. You can use any expressions valid for the PowerDesigner GTL <code>.if</code> macro (see <i>.if Macro</i> on page 275). You can reference any extended attributes defined at the metaclass level in the condition, but not those defined under the criterion itself.</p> <p>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:</p> <pre>(%DimensionalType% == "1")</pre> <p><code>%DimensionalType%</code> is an attribute of the <code>BaseTable</code> 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.</p> |
| Parent | Specifies the parent criterion of the criterion. To move the criterion to under another parent, select the parent in the list. Click the Properties tool to open the parent and view its properties. |
| Comment | Specifies additional information about the criterion. |



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

Extended Attributes (Profile)

Extended attributes define additional metadata to capture for object instances. You can specify a default value, allow users to freely enter numeric, string, or other types of data (or select objects), provide an open or closed list of possible values, or calculate a value.

Note: Extended attributes are listed on a generic **Extended Attributes** tab in the object property sheet, unless you insert them into forms (see *Forms (Profile)* on page 55). If all the extended attributes are allocated to forms, the generic page will not be displayed.

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

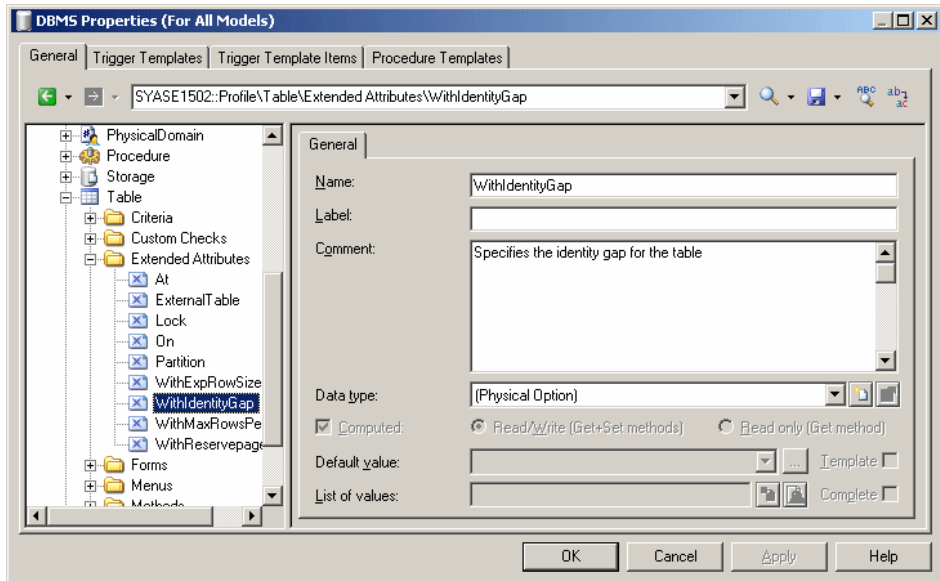
| Property | Description |
|----------|--|
| Name | Specifies the internal name of the attribute, which can be used for scripting. |
| Label | Specifies the display name of the attribute, which will appear in the PowerDesigner interface. |

| Property | Description |
|-----------|--|
| Comment | Provides additional information about the extended attribute. |
| Data type | <p>Specifies the form of the data to be held by the extended attribute. You can choose from:</p> <ul style="list-style-type: none"> • Boolean - TRUE or False. • Color - xxx xxx xxx where x is an integer between 0-255. • Date or Time - your local format as specified in your Windows regional settings • File or Path - cannot contain /// or any of the following characters: ?" < > . • Integer or Float - the appropriate local format. • Hex - a hexadecimal. • Font - font name, font type, font size. • Font Name or Font Style - a 1-50 character string. • Font Size - an integer between 1-400. • Object - an object of the correct type and, if appropriate, with the correct stereotype. When selecting this type you must specify an Object type and, if appropriate, an Object stereotype, and you can also specify an Inverse collection name (see <i>Linking Objects Through Extended Attributes</i> on page 48). • Password - no restrictions. • String (single line) or Text (multi-line) - no restrictions. <p>Select the Validate check box to the right of the list to enforce validation of the values entered for the attribute.</p> <p>To create your own data type, click the Create Extended Attribute Type tool to the right of the field (see <i>Creating an Extended Attribute Type</i> on page 45).</p> |
| Computed | <p>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:</p> <ul style="list-style-type: none"> • Read/Write (Get+Set methods) • Read only (Get method) <p>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:</p> <pre>Function %Get%(obj) %Get% = obj.GetPhysicalOptionValue("on/<filegroup>") End Function Sub %Set%(obj, value) obj.SetPhysicalOptionValue "on/<filegroup>", value End Sub</pre> |

| Property | Description |
|----------------|--|
| Default value | <p>[if not computed] Specifies a default value for the attribute. You can specify the value in any of the following ways:</p> <ul style="list-style-type: none"> • 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. |
| Template | <p>[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.</p> <p>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%.</p> |
| List of values | <p>Specifies a list of possible values for the attribute in one of the following ways:</p> <ul style="list-style-type: none"> • 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. <p>If the attribute type is <code>Object</code>, and you do not want to filter the list of available objects in any way, you can leave this field blank.</p> <p>To perform a simple filter of the list of objects, use the <code>.collection</code> macro (see <i>.object and .collection Macros</i> on page 278). In the following example, only tables with the <code>Generated</code> attribute set to true will be available for selection:</p> <pre>.collection (Model.Tables, %Generated%==true)</pre> <p>For more complex filtering, use the <code>foreach_item</code> macro (see <i>.foreach_item Macro</i> on page 271):</p> <pre>.foreach_item (Model.Tables) .if %Generated% // (or more complex criteria) %ObjectID% .endif .next (\n)</pre> <p>If the attribute is based on an extended attribute type (see <i>Creating an Extended Attribute Type</i> on page 45), this field is unavailable since the values of the extended attribute type will be used.</p> |
| Complete | <p>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.</p> |

| Property | Description |
|--------------------|--|
| Edit method | <p>[if not Complete] Specifies a method to override the default action associated with the tool to the right of the field.</p> <p>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:</p> <pre> 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 </pre> |
| Icon Set | Specifies a set of icons to display on object symbols in place of extended attribute values (see <i>Specifying Icons for Attribute Values</i> on page 46). |
| 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 stereo-type | [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 <i>Adding DBMS Physical Options to Your Forms</i> on page 210. |

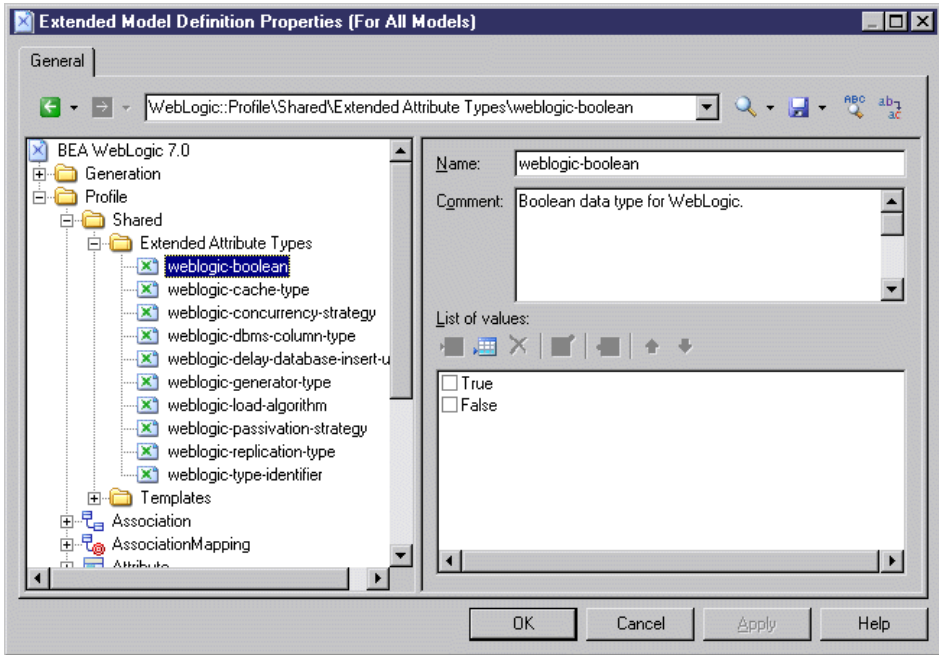


3. Click **Apply** to save your changes.

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.



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 Attributes (Profile)* on page 41).

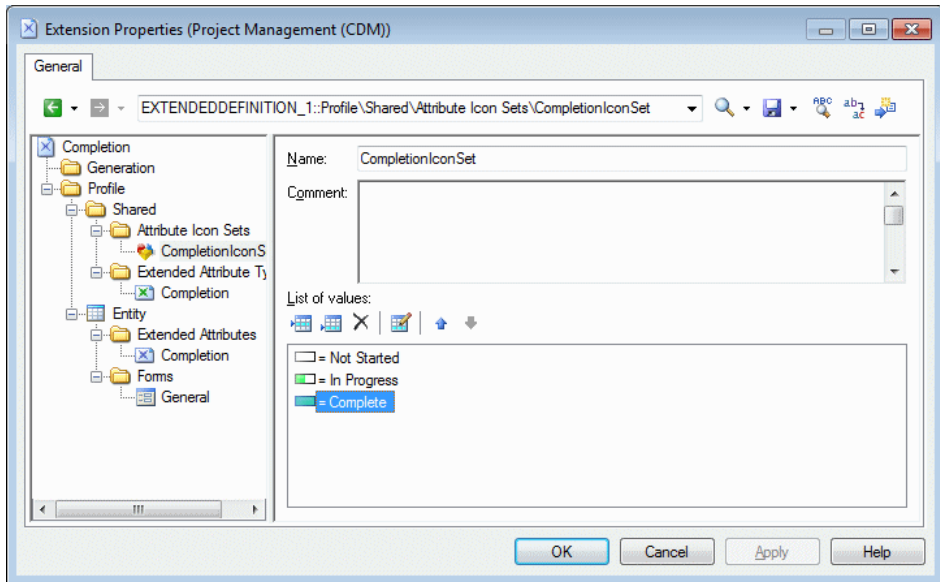
Specifying Icons for Attribute Values

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 (see *Extended Attributes (Profile)* on page 41).
2. Select a standard data type or an extended attribute type (see *Creating an Extended Attribute Type* on page 45).
3. If appropriate, specify a list of possible values and a default value.
4. 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 (=*).

5. 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:



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

6. If appropriate, add the attribute to a form (see *Forms (Profile)* on page 55), to enable users to modify its value.
7. Click **OK** to save your changes and return to the model.
8. 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 > Modeling with PowerDesigner > Diagrams, Matrices, and Symbols > Display Preferences*.

Your attribute is now displayed on object symbols. In the following example, the Employee entity is 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

Specify the [Object] data type to allow users to select another object as the value of the attribute. You must specify an **Object type** (metaclass) to link to, and can optionally specify an **Object stereotype** to filter the objects available for selection and an **Inverse collection** name, which will be displayed on the **Dependencies** tab on the referenced object property sheet.

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. When I set the **Owner** property of a table, the table will be listed on the **Dependencies** tab of the user property sheet, under the inverse collection name of Tables owned.

Extended Collections and Compositions (Profile)

Extended collections define the possibility to associate an object instance with a group of other objects of the specified type. Extended compositions define a parent-child connection between an object instance and a group of sub-objects derived from the ExtendedSubObject metaclass.

For extended collections, the association between the parent and child objects is relatively weak, so that if you copy or move the parent object, the related objects are not copied or moved, but the connection is maintained (using shortcuts if necessary). For example, you could associate documents containing use case specifications with the different packages of a model by creating an extended collection under the Package metaclass and specifying FileObject as the target metaclass.

For extended compositions, the association is stronger. Sub-objects can only be created within the parent object and are moved, copied, and/or deleted along with their parent.

The collection or composition is displayed as a new tab in the object instance property sheet. The property sheets of objects referenced in a collection show the object instance owning the collection on their **Dependencies** tab.

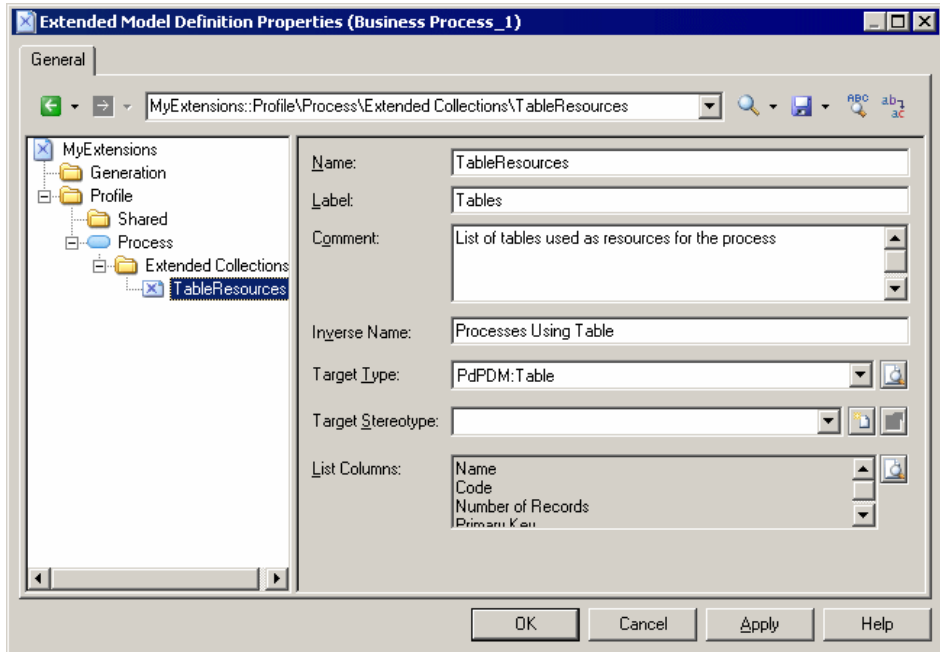
1. Right-click a metaclass, stereotype, or criterion and select **New > Extended Collection** or **Extended Composition**.

Note: If you define the collection or composition under a stereotype or criterion, its tab is displayed only if the metaclass instance bears the stereotype or meets the criterion.

2. Enter the following properties as appropriate:

| Property | Description |
|-------------------|--|
| Name | Specifies the name of the extended collection or composition. |
| Label | Specifies the display name of the collection, which will appear as the name of the tab associated with the collection in the parent object property sheet. |
| Comment | [optional] 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. |
| 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 click the Create tool to the right of this field to create 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. |

3. 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.

Note: 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)

Calculated collections define a read-only connection between an object instance and a group of other objects of the specified type. The logic of the collection is defined using VBScript.

Calculated collections, unlike extended collections (see *Extended Collections and Compositions (Profile)* on page 48) cannot be modified by the user.

You can 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.
- 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 > Storing, Sharing and Reporting on Models > 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.

1. Right-click a metaclass, stereotype, or criterion and select **New > Calculated Collection**.
2. Enter the following properties as appropriate:

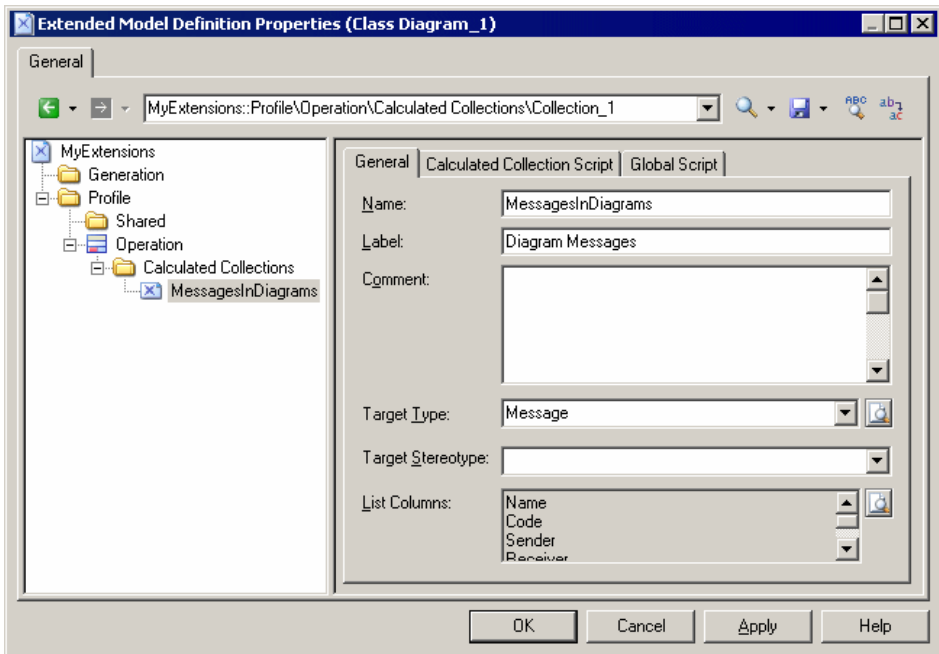
| Property | Description |
|-------------------|---|
| Name | Specifies the name of the calculated collection for use in scripts. |
| Label | Specifies the display name of the collection, which will appear as the name of the tab associated with the collection in the parent object property sheet. |
| Comment | [optional] 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 | [optional] 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 columns displayed by default on the collection property sheet tab. |

3. 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, which 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 *Example: PDM Custom Check* on page 73 and *Global Script (Profile)* on page 105.

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



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

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.

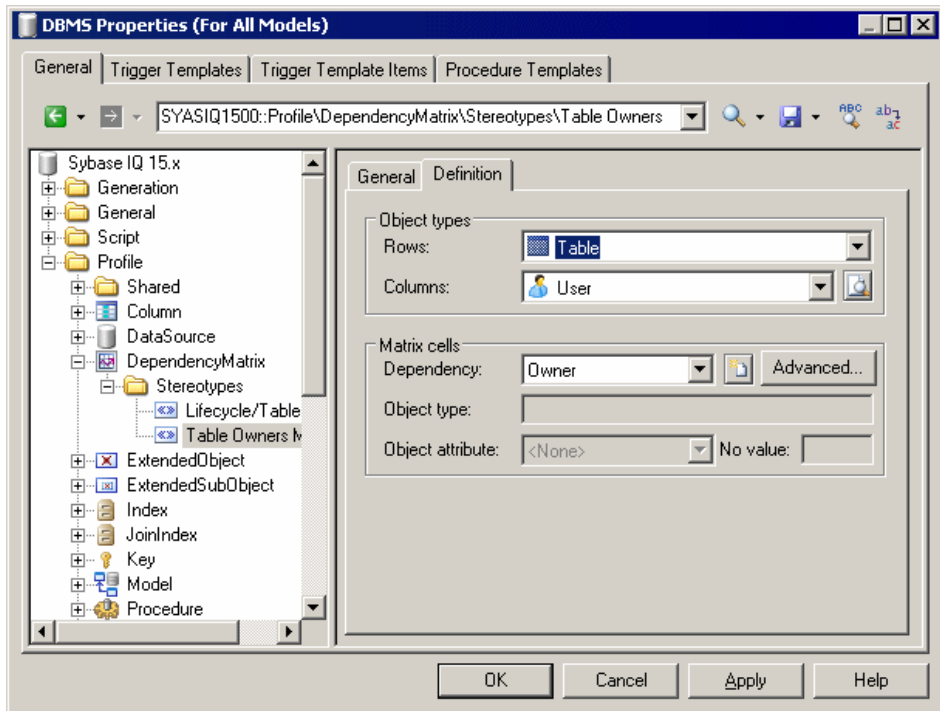
For example, you could create dependency matrices that show links between:

- 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
1. Right-click the **Profile** category and select **Add Dependency Matrix** to add the `DependencyMatrix` metaclass to the profile and create a stereotype under it, in which you will define the matrix properties.
 2. On the **General** tab, 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 and how they are associated using the following properties.

| Property | Description |
|--------------|---|
| Rows | Specifies the object type with which to populate your matrix rows. |
| Columns | Specifies the object type with which 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 | <p>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.</p> <p>Click the Create button to the right of the list to create a new extended collection (see <i>Extended Collections and Compositions (Profile)</i> on page 48) connecting your objects, or the Advanced button to specify a complex dependency path (see <i>Specifying Advanced Dependencies</i> on page 54).</p> <p>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.</p> |



4. 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 > Modeling with PowerDesigner > 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:

Dependency Path Definition

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 |
| | |
| | |
| | |

Reset OK Cancel Help

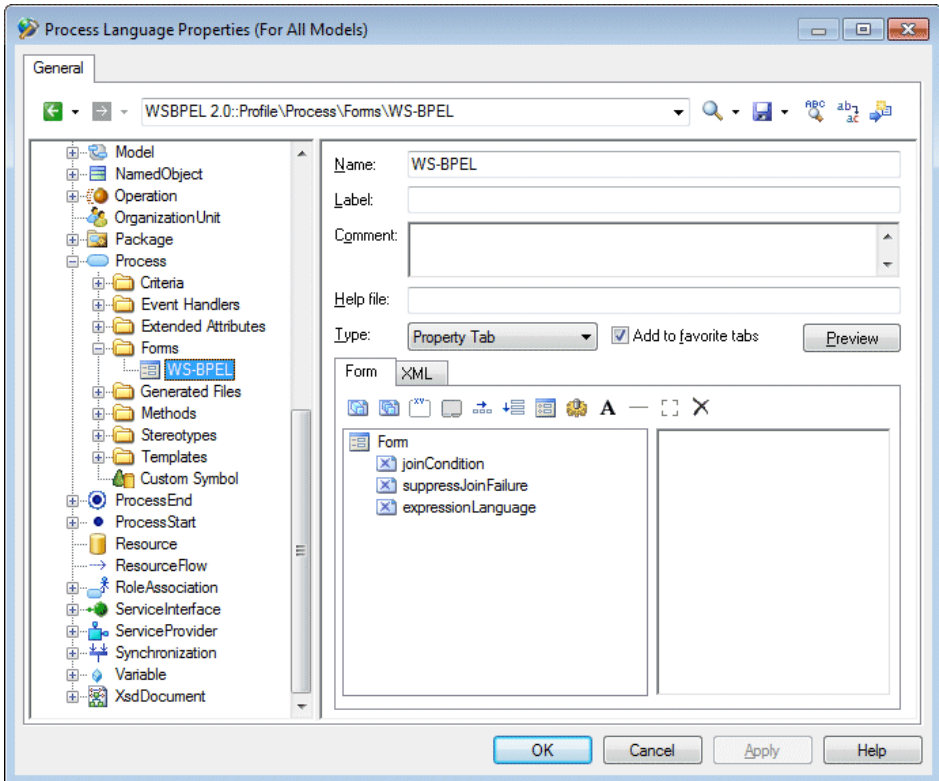
Forms (Profile)

Forms present standard and extended attributes and collections as property sheet tabs or can be used to create dialog boxes launched from menus or property sheet buttons.

Note: Unless you add them to a form, 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. If you associate all of your extended attributes with a form, the **Extended Attributes** tab is not displayed.

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

Note: If you define a property tab under a stereotype or criterion, it is displayed only when the metaclass instance bears the stereotype or meets the criterion.



2. Enter the appropriate following properties:

| 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 display in the tab of the property tab or in the title bar of the dialog box. |
| Comment | Provides additional information about the form. |











| Property | Description |
|----------------------|--|
| Help file | <p>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.</p> <p>The action can be the display of a help file (.hlp, .chm or .html), and can specify a specific topic. For instance:</p> <pre>C:\PD1500\pddoc15.chm 26204</pre> <p>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:</p> <pre>notepad.exe C:\Temp\Readme.txt</pre> |
| Type | <p>Specifies the kind of form. You can choose from the following:</p> <ul style="list-style-type: none"> • 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 <i><standard></i> 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. |

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 57).
4. Click the **Preview** button to review the layout of your form and, when satisfied, click **Apply** to save your changes.

Adding Extended Attributes and Other Controls to Your Form

You insert controls into your form using the tools in the Form tab toolbar. You can reorder controls in the form control tree by dragging and dropping them. To place a control inside a container control (group box or horizontal or vertical layout), drop it onto the container. For example, if you want the extended attributes GUID, InputGUID, and OutputGUID to be displayed in a GUI group box, you should create a group box, name it GUI and drag and drop all three extended attributes under the GUI group box.

The following tools are available:

| Tool | Description |
|---|---|
|  | <p>Add Attribute / Collection – opens a selection box in which you select standard or extended attributes or collections belonging to the metaclass to insert into the form. If you do not enter a label, the attribute or collection name is used as its form label. If you have entered a comment, it is displayed as a tooltip.</p> <p>The type of control associated with an 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. Collections are displayed as standard grids with all the appropriate tools.</p> |
|  | <p>Add Group Box - inserts a group box, intended to contain other controls within a named box.</p> |
|  | <p>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.</p> |
|  | <p>Add Horizontal / Vertical Layout - inserts a horizontal or vertical layout. To arrange controls to display side by side, drag them onto a horizontal layout in the list. To arrange attributes to display one under the other, drag them onto a vertical layout in the list. Vertical and horizontal layouts are often used together to provide columns of controls.</p> |
|  | <p>Include Another Form - inserts a form defined on this or another metaclass in the present form (see <i>Example: Including a Form in a Form</i> on page 64).</p> |
|  | <p>Add Method Push Button - opens a selection box in which you select one or more methods belonging to the metaclass to associate with the form via buttons. Clicking the button invokes the method. If you do not enter a label, the method name is used as the button label. If you have entered a comment, it is displayed as a tooltip.</p> |
|  | <p>Add Edit / Multi-Line Edit Field [dialog boxes only] inserts an edit or multi-line edit field.</p> |
|  | <p>Add Combo Box / List Box / Check Box [dialog boxes only] - inserts a combo box, list box, or check box.</p> |
|  | <p>Add Text / Separator Line / Spacer - inserts the appropriate decorative control. The separator line is vertical when its parent control is a vertical layout.</p> |
|  | <p>Delete – deletes the currently selected control.</p> |

Select a control to specify properties to control its 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 <i>Example: Opening a Dialog Box from a Menu</i> on page 82). |

| Property | Definition |
|-------------|---|
| Label | <p>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.</p> <p>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).</p> |
| Attribute | <p>[included forms] Specifies the object on which the form to be included is defined. The list is populated with all attributes of type <code>object</code> and the following objects:</p> <ul style="list-style-type: none"> • <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 |
| Form name | <p>[included forms] Specifies the name of the form that will be included. You can:</p> <ul style="list-style-type: none"> • 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 | <p>[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.</p> |

| Property | Definition |
|------------------------------|--|
| <p>Label space</p> | <p>[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.</p> <p>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.</p> <p>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.</p> <div data-bbox="400 522 1174 1013" style="border: 1px solid gray; padding: 5px;"> </div> |
| <p>Show control as label</p> | <p>[group boxes] Use the first control contained within the group box as its label.</p> |
| <p>Show Hidden Attribute</p> | <p>[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.</p> |
| <p>Value</p> | <p>[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 <i>Extended Attributes (Profile)</i> on page 41).</p> |
| <p>List of Values</p> | <p>[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 <i>Extended Attributes (Profile)</i> on page 41).</p> |
| <p>Exclusive</p> | <p>[combo boxes] Specifies that only the values defined in the List of values can be entered in the combo box.</p> |

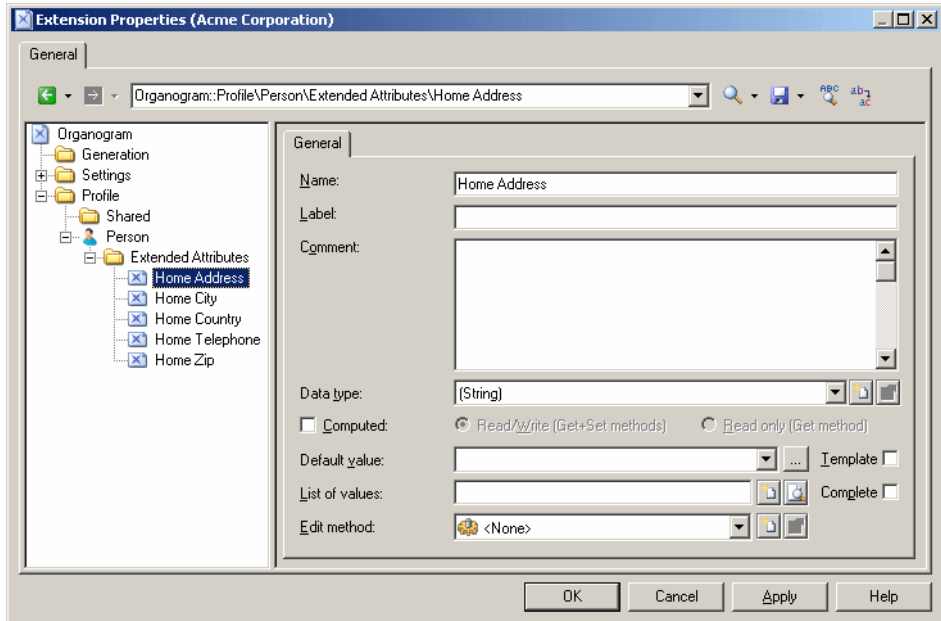
| Property | Definition |
|------------------------------|--|
| Minimum Size (chars) | Specifies the minimum width (in characters) to which the control may be reduced when the window is resized. |
| Minimum Line Number | Specifies the minimum number of lines to which a multiline control may be reduced when the window is resized. |
| Horizontal / Vertical Resize | Specifies that the control may be resized horizontally or, for multiline controls, vertically, when the property sheet or dialog 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 a check box or vertical or horizontal radio buttons, while 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. |

Example: Creating a Property Sheet Tab

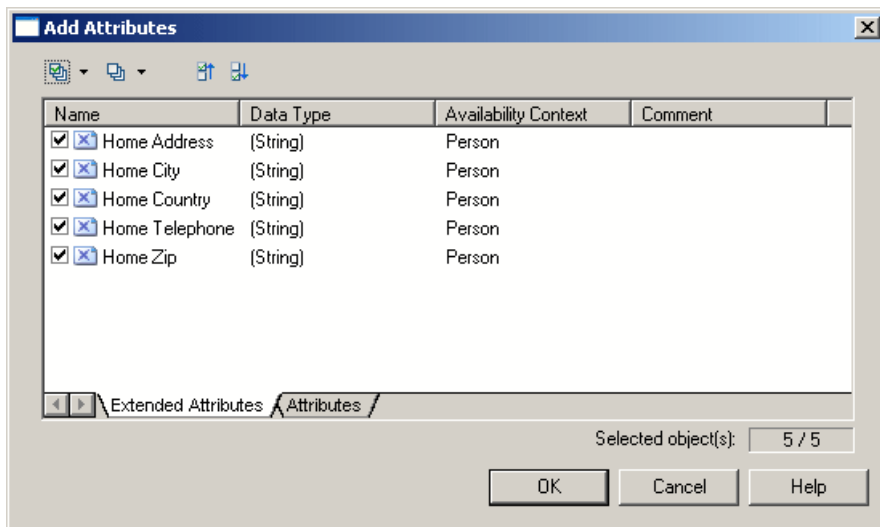
In this example, we will create a new property tab for the EAM Person metaclass to display extended attributes we define to store personal information.

1. Create a new extension file (see *Creating an Extension File* on page 12) in an EAM, add the `Person` metaclass (see *Metaclasses (Profile)* on page 33), and define five extended attributes (see *Extended Attributes (Profile)* on page 41) to contain home contact details:

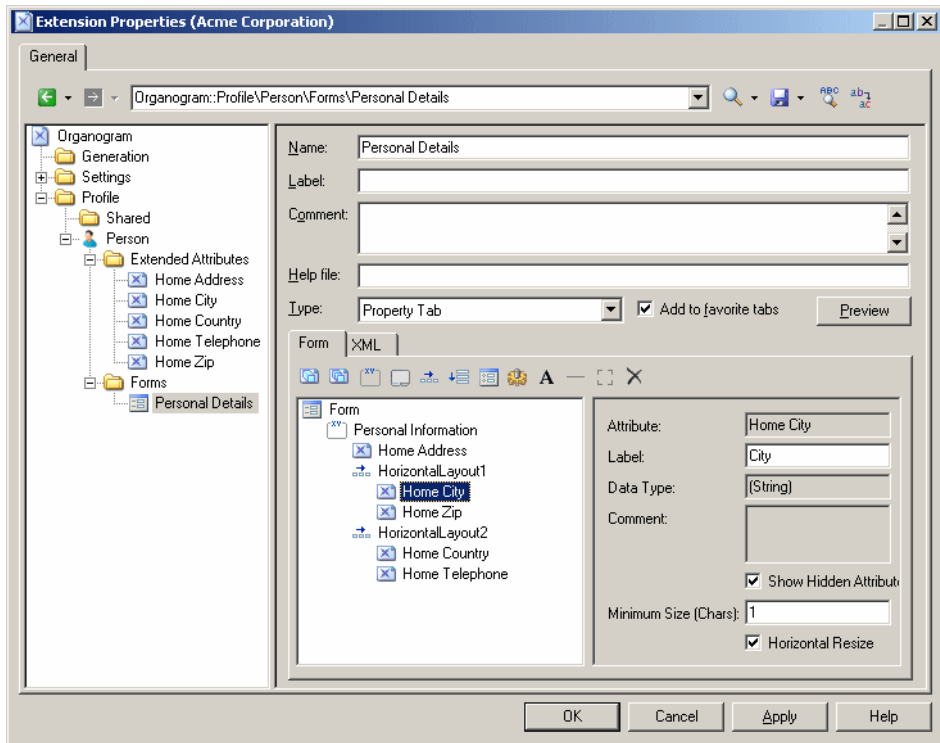
CHAPTER 2: Extension Files



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:



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 override the default name of the attribute in the form for brevity:



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:

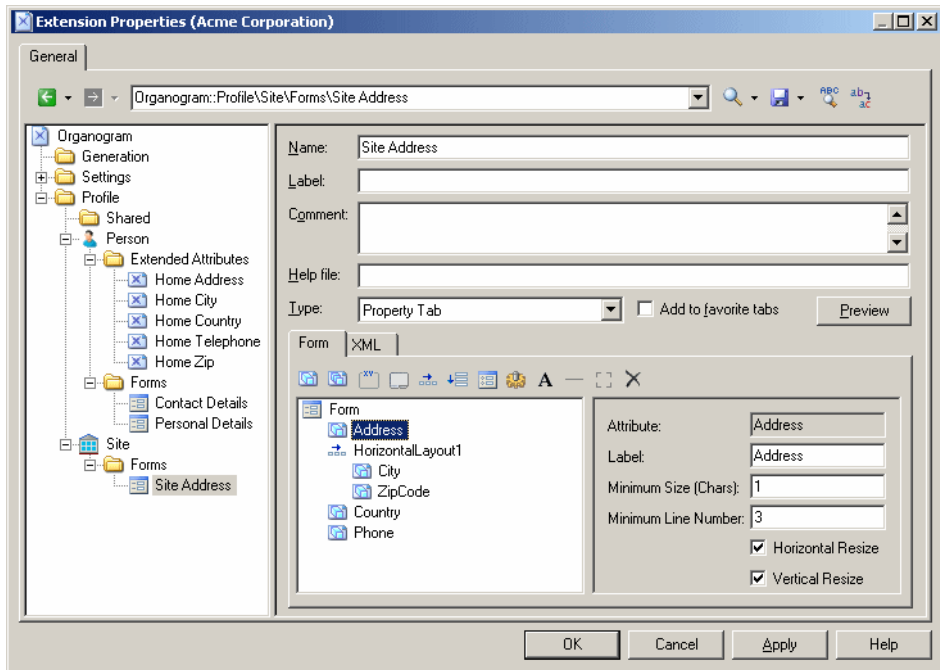
The screenshot shows a dialog box titled "Person Properties - James Jones (james_jones)". It has four tabs: "General", "Roles", "Personal Details", and "Notes". The "Personal Details" tab is active, displaying a "Personal Information" section. This section contains several text input fields: "Home address:" with the value "69 Oldbury Gardens", "City:" with "Chippingham", "Zip:" with "SW16 9AL", "Country:" with "UK", and "Tel:" with "0208-555-9876". At the bottom of the dialog, there are five buttons: "More >>", a printer icon, "OK", "Cancel", "Apply", and "Help".

Example: Including a Form in a Form

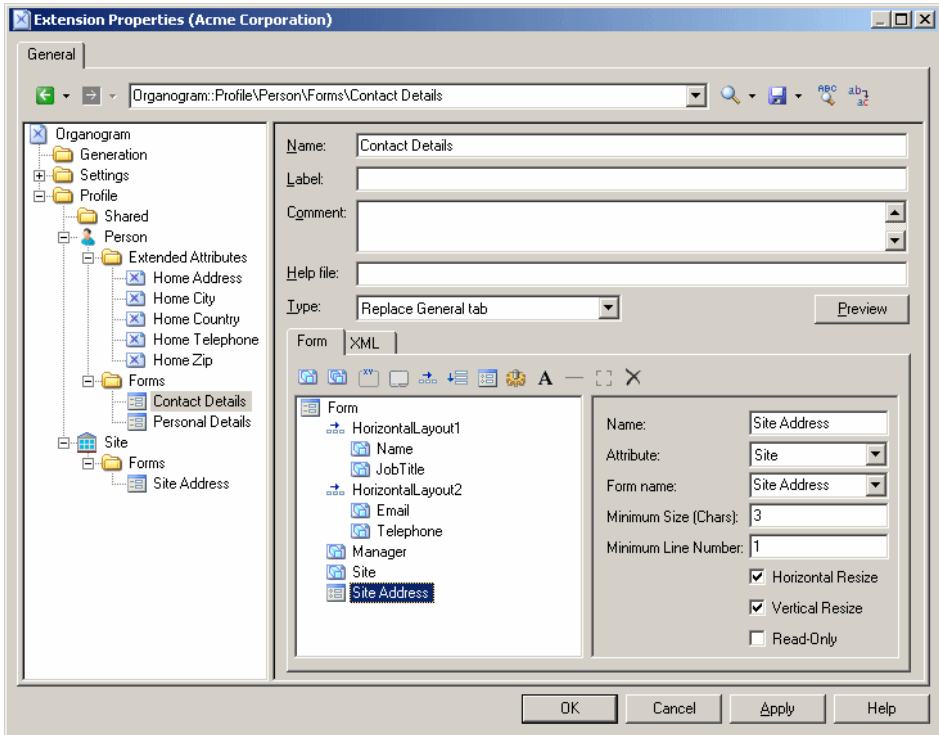
In this example, we will replace the General tab of the EAM Person metaclass by a form which includes properties from the person and from the site to which she is assigned by including a form defined on the Site metaclass as a read-only control in a form defined on the Person metaclass.

This example builds on the extension file created in *Example: Creating a Property Sheet Tab* on page 61.

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:



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:



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

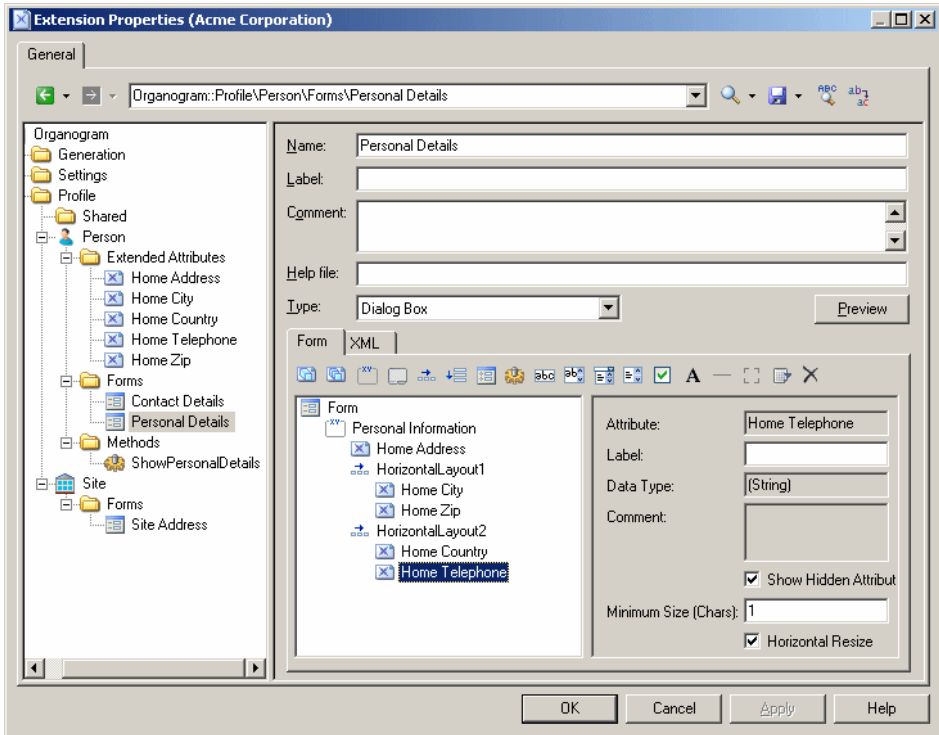
More >> OK Cancel Apply Help

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 64.

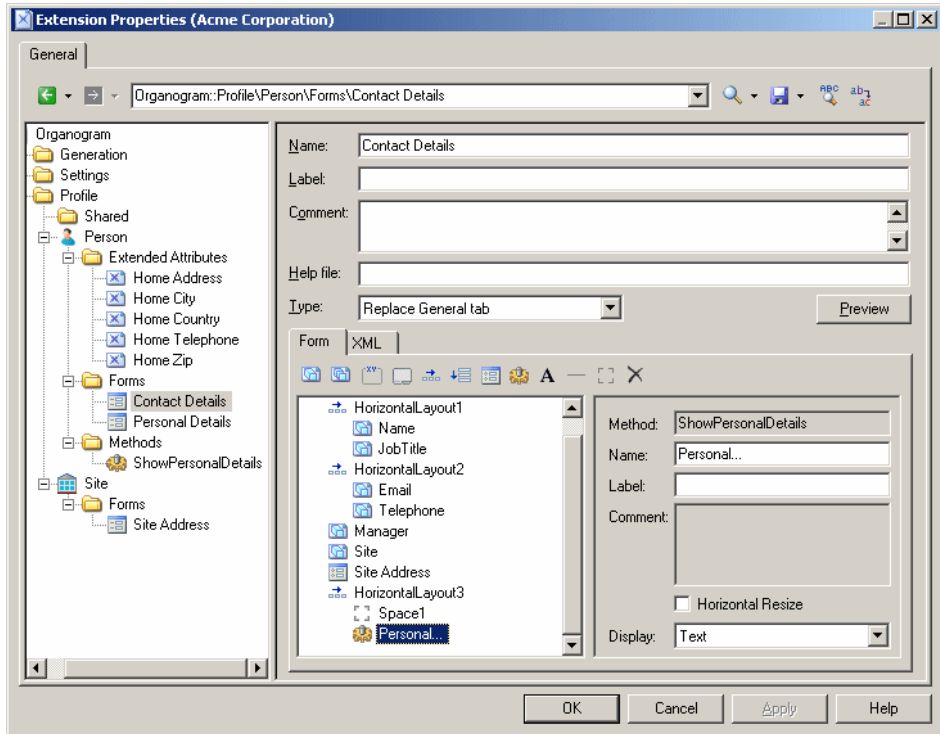
1. Open the `Personal Details` form under the `Person` metaclass, and select `Dialog Box` in the **Type** field, to transform it from a property sheet tab into an independent dialog:



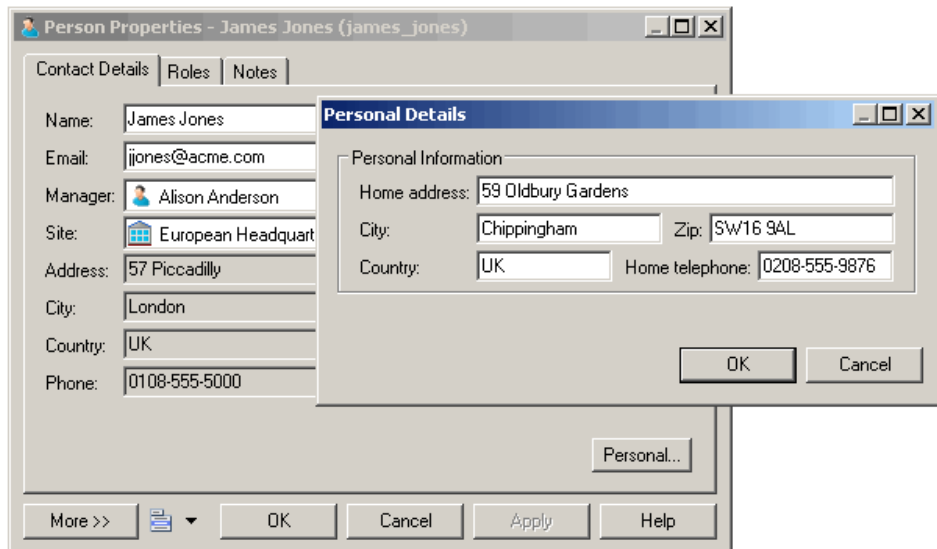
2. Right-click the Person metaclass and select **New > Method**. Enter the name ShowPersonalDetails, and then click the **Method Script** tab and enter the following script:

```
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:



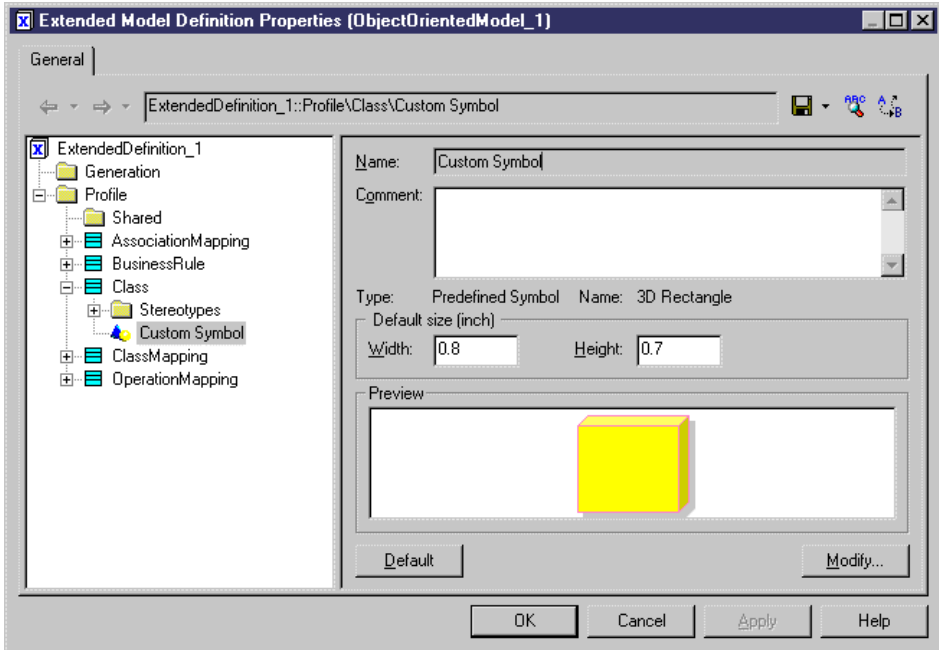
4. Enter `Personal...` in the **Label** field, and then click **OK** to save your changes and return to the model. Now when you open the property sheet of a person, the **Contact Details** tab contains a **Personal...** button which opens the **Personal Information** dialog:



Custom Symbols (Profile)

Custom symbols modify the appearance of object symbols in diagrams along with the content displayed on them. You can choose to enforce certain aspects of the symbol format and content, while allowing users some liberty to change others.

1. Right-click a metaclass, stereotype, or criterion and select **New > Custom Symbol**.



2. Specify a default **Width** and **Height** for the symbol and then click the **Modify** button to open the Symbol Format dialog, and set appropriate properties on the various tabs.

Note: If you customize the line style and arrows of a link symbol (such as a PDM reference), your styles will override those selected in the Display Preferences dialog, and may cause confusion and inconsistency in the model. To ensure coherence in a model governed by a notation, select **Notation** for the **Style** and **Arrows** properties on the **Line Style** tab.

For more information on the Symbol Format dialog (including the 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 > Modeling with PowerDesigner > Diagrams, Matrices, and Symbols > Symbols > Symbol Format Properties*.

3. Click **OK** to return to the resource editor and view your changes in the **Preview** field.

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

Custom Checks (Profile)

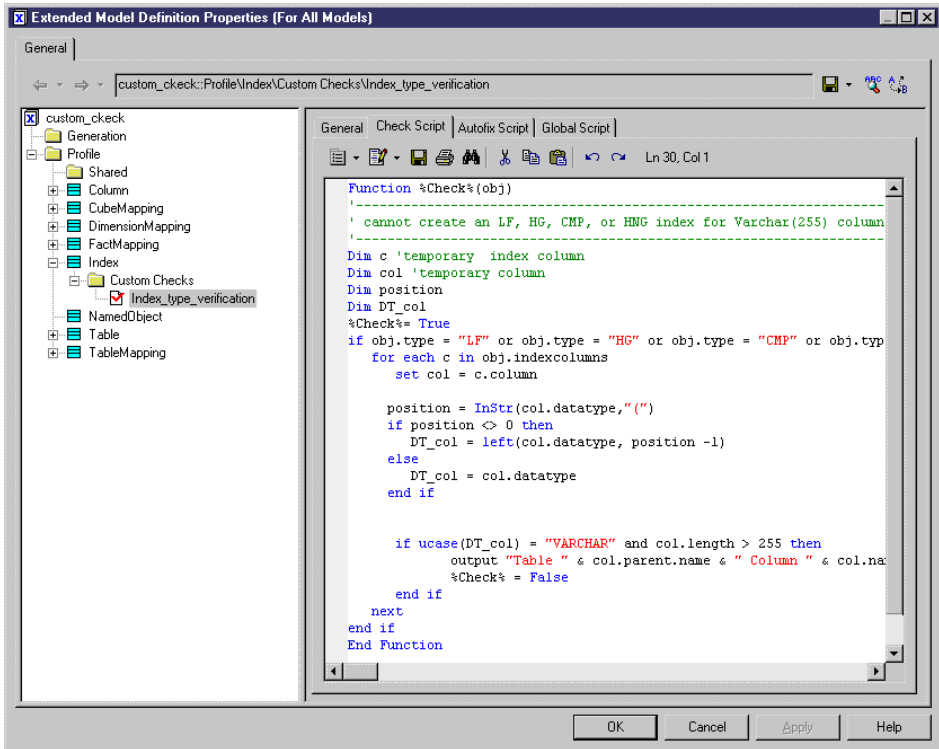
Custom checks define additional rules to validate the content of your models. The logic of the check is defined using VBScript. Custom checks appear alongside standard checks in the **Check Model** dialog.

Custom checks appear with standard model checks in the **Check Model Parameters** dialog (see *Core Features Guide > Modeling with PowerDesigner > Objects > Checking Models*).

1. Right-click a metaclass, stereotype, or criterion, and select **New > Custom Check**.
2. Enter the following properties as appropriate:

| Parameter | Description |
|---|--|
| Name | Specifies the name of the custom check, which 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 | Provides a description of the custom check. |
| Help Message | Specifies text to display in the message box that opens when the user right-clicks the check and selects Help . |
| Output message | Specifies text to display in the Output window during check execution. |
| Default severity | Specifies whether the check is designated by default as an error (major problem that stops generation) or a warning (minor problem or just recommendation). |
| Execute the check by default | Specifies that the check is selected by default in the Check Model Parameters dialog. |
| Enable automatic correction | Specifies that an autofix is available for the check (see <i>Example: PDM Autofix</i> on page 74). |
| Execute the automatic correction by default | Specifies that the autofix is executed by default. |

3. Click the **Check Script** tab and enter your script (see *Example: PDM Custom Check* on page 73). You can access shared library functions and static attributes defined for reuse in the resource file from the **Global Script** tab (see *Global Script (Profile)* on page 105).



4. If you want to define an autofix, click the **Autofix Script** tab and enter your script (see *Example: PDM Autofix* on page 74).
5. Click **Apply** to save your changes.

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. Your custom checks are displayed in the **Check Model Parameters** dialog box alongside the standard model checks. If there are errors in your custom check scripts, the user will be prompted with the following options:

- Ignore- Skip the problematic script and continue with the other checks.
- Ignore All - Skip this and any future scripts with problems and continue with the other checks.
- Abort - Stop the model checking.
- Debug - Stop the model checking and open the Resource Editor on the script line with the problem.

Example: PDM Custom Check

You enter the script of the custom check in the **Check Script** tab using VBScript. In this example, we will write a script to verify that Sybase IQ indexes of type HG, HNG, CMP, or LF are not linked with columns with a data type of VARCHAR with a length higher than 255.

The script is initialized with the following line, which must not be altered:

```
Function %Check%(obj)
```

At run-time the variable %Check% is replaced by concatenating the names of the resource file, metaclass, any stereotypes or criteria, and the name of the check itself from the **General** tab, with any spaces replaced by an underscore. The parameter obj contains the object being checked.

We begin by defining a certain number of variables after the default function definition:

```
Dim c 'temporary index column
Dim col 'temporary column
Dim position
Dim DT_col
```

Next, we enter the function body, which starts by setting the %Check% to true (meaning that the object passes the test) and then iterates over each of the columns associated with the index and tests their datatype. If a column has a varchar longer than 255, the script outputs a message and sets the check to false (the object fails the test):

```
%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
```

For more information about using VBScript in PowerDesigner, see *Chapter 7, Scripting PowerDesigner* on page 305.

Example: PDM Autofix

If the custom check you have defined supports an automatic correction, you enter its script on the **Autofix Script** tab using VBScript. In this example, we will write a script to fix a Sybase IQ index linked with columns with an invalid data type.

The script is initialized with the following line, which must not be altered:

```
Function %Fix%(obj, outmsg)
```

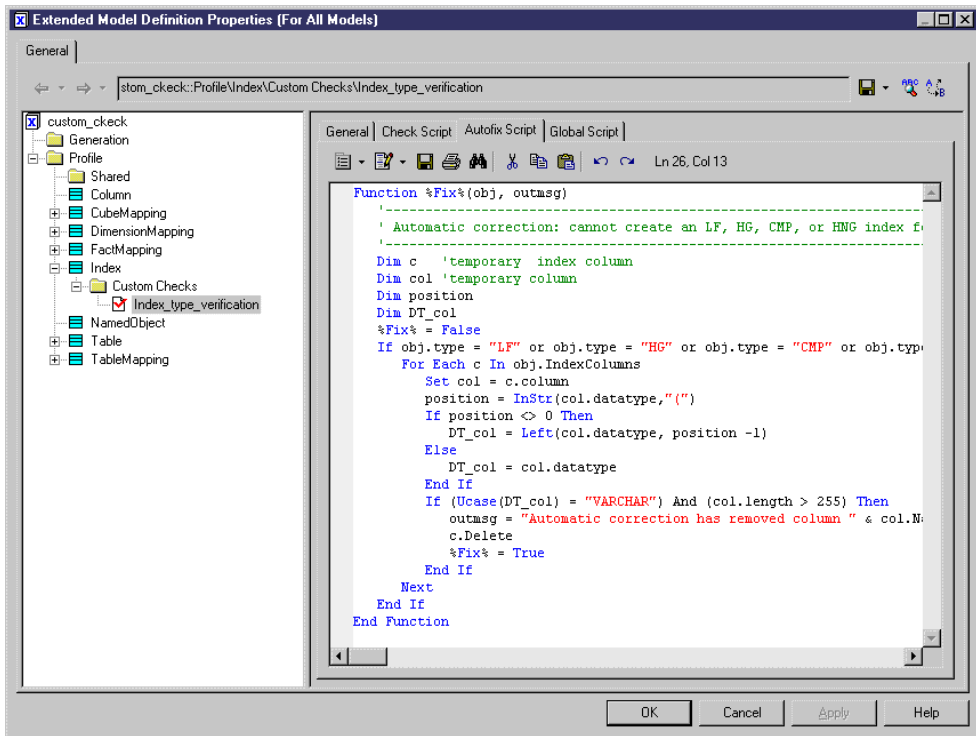
At run-time the variable %Fix% is replaced by the name of the fix. The parameter obj contains the object being checked and outmsg, the message to be output.

We begin by defining a certain number of variables after the default function definition:

```
Dim c 'temporary index column
Dim col 'temporary column
Dim position
Dim DT_col
```

Next, we enter the function body, which starts by setting the %Fix% to false (meaning that it does nothing) and then iterates over each of the columns associated with the index and tests their datatype. If a column has a varchar longer than 255, the script outputs a message, deletes the column from the collection of columns associated with the index, and sets the fix to true (it has made a correction):

```
%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
```



Event Handlers (Profile)

Event handlers define validation rules or other scripts to run when an event occurs on an object. The logic of the event handler is defined using VBScript. Criteria do not support event handlers.

1. Right-click a metaclass or a stereotype and select **New > Event Handler** to open a selection box, listing the available types of event handlers:

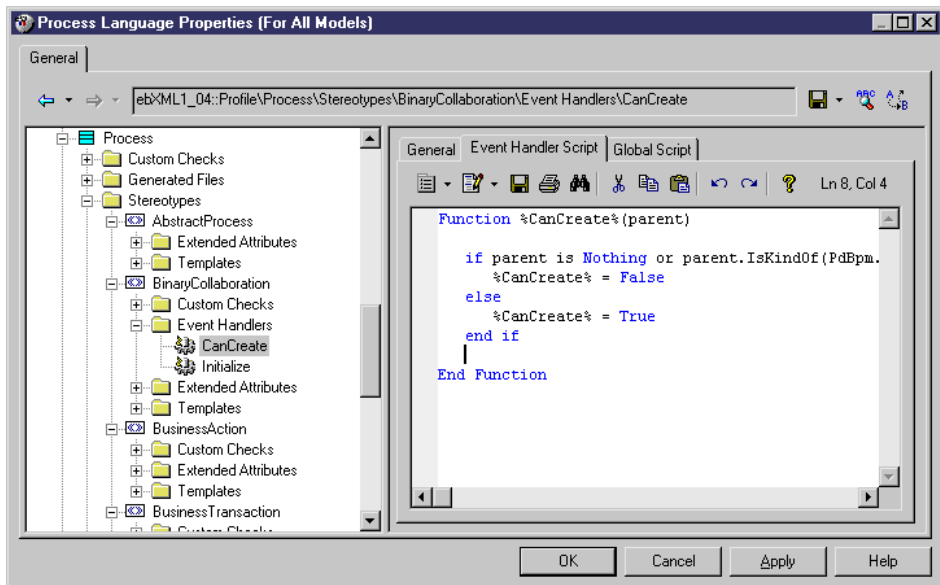
| Event handler | Description |
|---------------|--|
| CanCreate | <p>Implements a validation rule to prevent objects from being created in an invalid context. For example, in a BPM for ebXML, a process with a Business Transactions stereotype can only be created under a process with a Binary Collaboration stereotype. The script of the CanCreate event handler associated with the Business Transaction process stereotype is the following:</p> <pre data-bbox="481 404 1180 604"> Function %CanCreate%(parent) if parent is Nothing or parent.IsKindOf(PdBpm.Cls_Process) then %CanCreate% = False else %CanCreate% = True end if End Function </pre> <p>If the event handler returns True on a stereotype, then you can use the custom tool to create the stereotyped object and the stereotype is available in the Stereotype list on the object property sheet. If it returns True on a metaclass, then you can create the object from the Toolbox, from the Browser or in a list.</p> <p>Note: CanCreate event handlers are ignored during model import or reverse-engineering, since they could modify the model and make it diverge from the source.</p> |

| Event handler | Description |
|---------------|---|
| Initialize | <p>Instantiates objects with a predefined template. 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 script fragment is from the Initialize event handler for a Business Transaction.</p> <pre data-bbox="485 401 1180 907"> ... ' 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 ... </pre> <p>If the event handler returns True on a stereotype, then 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 returns True on a metaclass, then it 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 returns true on a model, then it will be 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.</p> |

| Event handler | Description |
|--|---|
| Validate | <p>Validates changes to object properties or triggers cascade updates when you change tabs or click OK or Apply in an object property sheet. You can define an error message to appear when the condition is not satisfied by filling the message variable and setting the %Validate% variable to False.</p> <p>In this example, the event handler verifies that a comment is added to the definition of an object:</p> <pre>Function %Validate%(obj, ByRef message) if obj.comment = "" then %Validate% = False message = "Comment cannot be empty" else %Validate% = True end if End Function</pre> |
| CanLinkKind | <p>[link objects] Validates the kind and stereotype of the objects that can be linked together as the source and destination extremities when you create a link with a Toolbox tool or modify link ends in a property sheet. The sourceStereotype and destinationStereotype parameters are optional.</p> <p>In this example, the source of the extended link must be a start object:</p> <pre>Function %CanLinkKind%(sourceKind, sourceStereotype, destinationKind, destinationStereotype) if sourceKind = cls_Start Then %CanLinkKind% = True end if End Function</pre> |
| OnModelOpen, OnModelSave, and OnModelClose | <p>[models] Run immediately after a model is opened, saved, or closed.</p> |
| OnLanguageChangeRequest, OnLanguageChanging, and OnLanguageChanged | <p>[models] Run immediately:</p> <ul style="list-style-type: none"> • Before the model's DBMS or language definition file is changed. If the event handler returns false, then the language change is canceled. • After the language change, but before any transformations are applied to objects to make them conform with the new language definition. • After the model's DBMS or language definition file is changed and the object transformations are applied. |
| OnNewFromTemplate | <p>[models] Runs immediately after a model or a project is created from a model or project template.</p> |

| Event handler | Description |
|--|---|
| BeforeDatabaseGenerate, AfterDatabaseGenerate, BeforeDatabaseReverseEngineer, and AfterDatabaseReverseEngineer | [PDM models] Run immediately before or after generating or reverse-engineering a database (see <i>Adding Scripts Before or After Generation and Reverse Engineering</i> on page 133). |
| GetEstimatedSize | [PDM only] Runs when the Estimate Database Size mechanic is called (see <i>Modifying the Estimate Database Size Mechanism</i> on page 202). |

2. Select one or more event handlers and click **OK** to add them.
3. Enter a name and comment to identify and document the event handler.
4. Click the **Event Handler Script** tab and enter a script to define the event handler. You can access shared library functions and static attributes defined for reuse in the resource file from the **Global Script** tab (see *Global Script (Profile)* on page 105).



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

Methods (Profile)

Methods are written in VBScript and perform actions on objects when they are invoked by other extensions, such as menu items or form buttons.

1. Right-click a metaclass, stereotype, or criterion and select **New > Method**.

2. Enter the following properties as appropriate:

| Property | Description |
|----------|---|
| Name | Specifies the name of the method. |
| Comment | Provides additional information about the method. |

3. Click the **Method Script** tab, and enter the VBScript. If appropriate, you can reuse functions on the **Global Script** tab.

For more information on defining a script and using the **Global Script** tab, see *Example: PDM Custom Check* on page 73 and *Global Script (Profile)* on page 105.

The following example, created under the `Class` metaclass, converts classes into interfaces by copying basic class properties and operations, deleting the class (to avoid namespace problems), and creating the new interface.

```
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
```

Note: 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: Opening a Dialog Box from a Menu* on page 82).

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





Menus (Profile)

Menus specify commands to appear in the standard PowerDesigner **File**, **Tools**, and **Help** menus or in contextual menus.

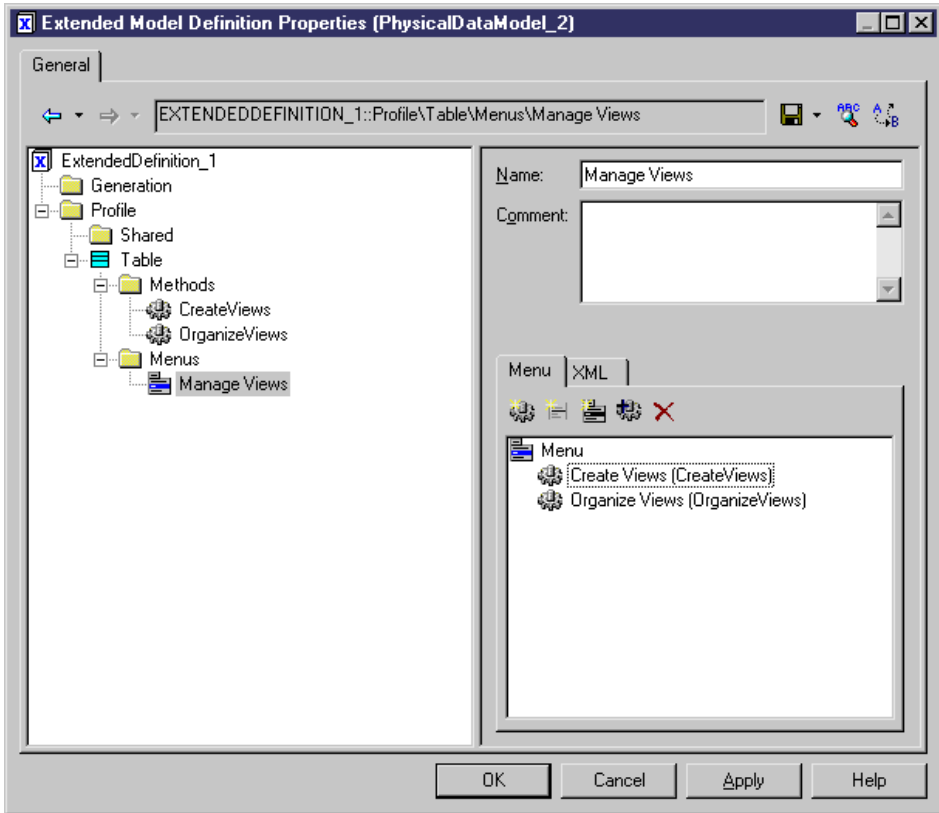
1. Right-click a metaclass, stereotype, or criterion and select **New > Menu**.
2. Enter the following properties as appropriate:

| 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: <ul style="list-style-type: none"> • 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. |

3. Use the tools on the **Menu** sub-tab to create the items in your menu:

| Tool | Function |
|---|---|
|  | <p>Add Command - Opens a selection dialog listing methods (see <i>Methods (Profile)</i> on page 79) and transformations (see <i>Transformations (Profile)</i> on page 92) defined in the current metaclass and its parents to add to the menu as commands. Select one or more and click OK.</p> <p>The items are added to your menu in the format:</p> <p>MenuEntry (Method/TransformationName)</p> <p>You can modify the MenuEntry (and define a shortcut key by adding an ampersand before the shortcut letter) but you must not edit the Method/TransformationName.</p> <p>Note: If you modify the name of a method or transformation, you must update any commands using the method or transformation by hand, because the name is not automatically synchronized. You can use the Replace in Items tool to locate and update these commands.</p> |
|  | Add Separator -Creates a menu separator under the selected item. |
|  | Add Submenu - Creates a submenu under the selected item. |
|  | Delete - Deletes the selected item. |

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.



4. [optional] Click the **XML** sub-tab to review the XML generated from the **Menu** sub-tab.
5. Click **Apply** to save your changes.

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 12) in a PDM and add the `Table` metaclass (see *Metaclasses (Profile)* on page 33).
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")

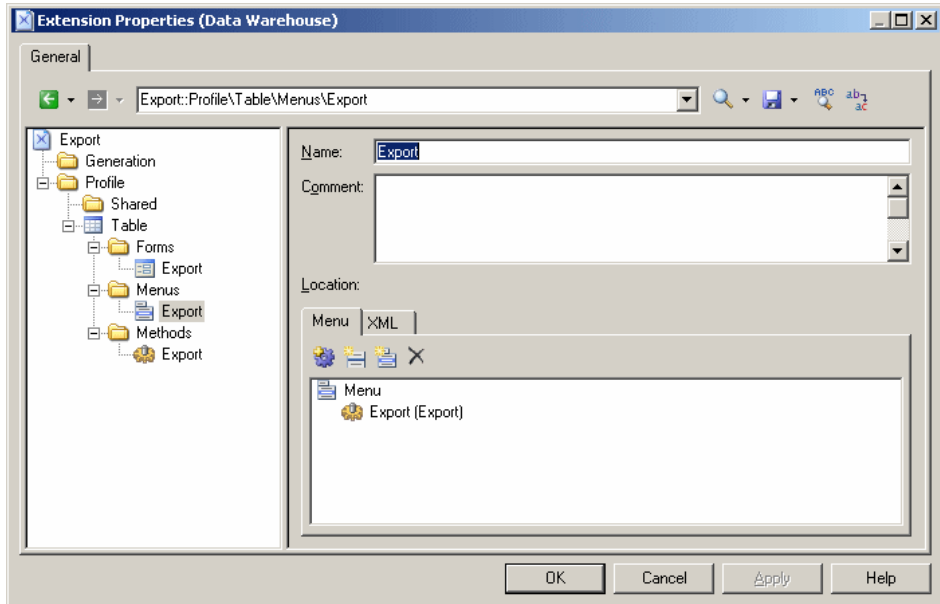
    ' Process the export algorithm...
    ' (Actual export code not included in this example)

    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:



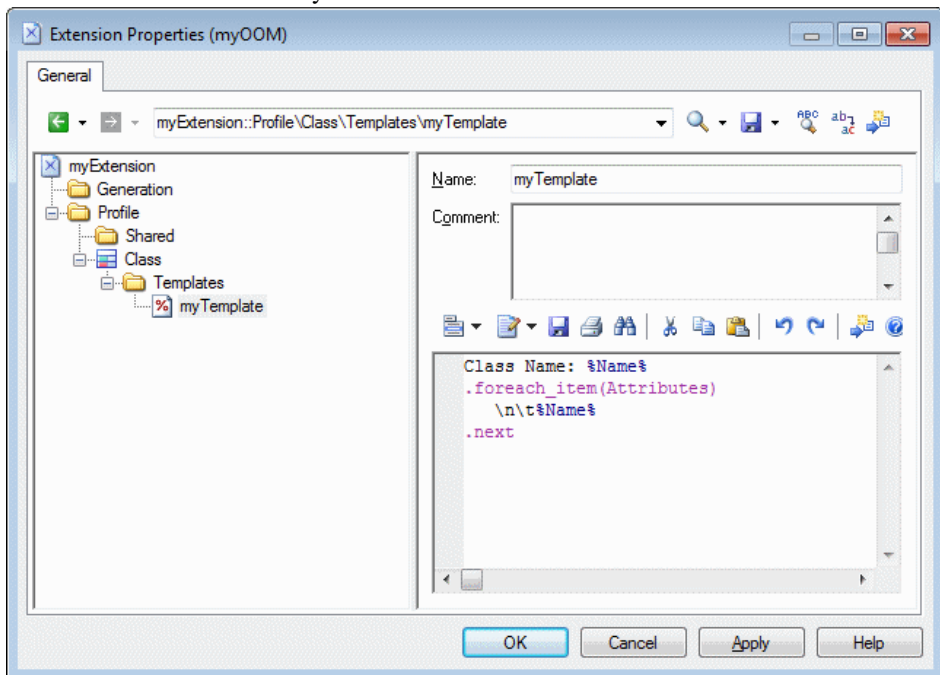
6. Click **OK** to save your changes and return to your model. When you next right-click a table in a diagram or the browser, the **Export** command is available in the contextual menu.

Templates (Profile)

GTL templates extract text from PowerDesigner property values for use in generated files or other contexts.

1. Right-click a metaclass, a stereotype, or a criterion (or the Shared category, if the template applies to all metaclasses) and select **New > Template** to create a template.
2. Enter a name for the template. You should not use spaces in the name and, by convention, templates are named in headless camelcase (for example `myTemplate`).
3. [optional] Enter a comment to explain the use of the template.
4. Enter GTL code (see *Chapter 5, Customizing Generation with GTL* on page 245) in the text box.

In this example, `myTemplate` is defined on the `Class` metaclass, and will generate the name of the class followed by a list of its attributes:



Generated Files (Profile)

Generated files assemble GTL templates for generation as files or for previewing on the object property sheet **Preview** tab.

1. Right-click a metaclass, stereotype, or criterion, and select **New > Generated File**.

Only objects, such as tables or classes, support file generation. However, you can still create generated files for sub-objects, such as columns and attributes, to preview code generated for them on their property sheet **Preview** tab.

2. Enter the following properties as appropriate:

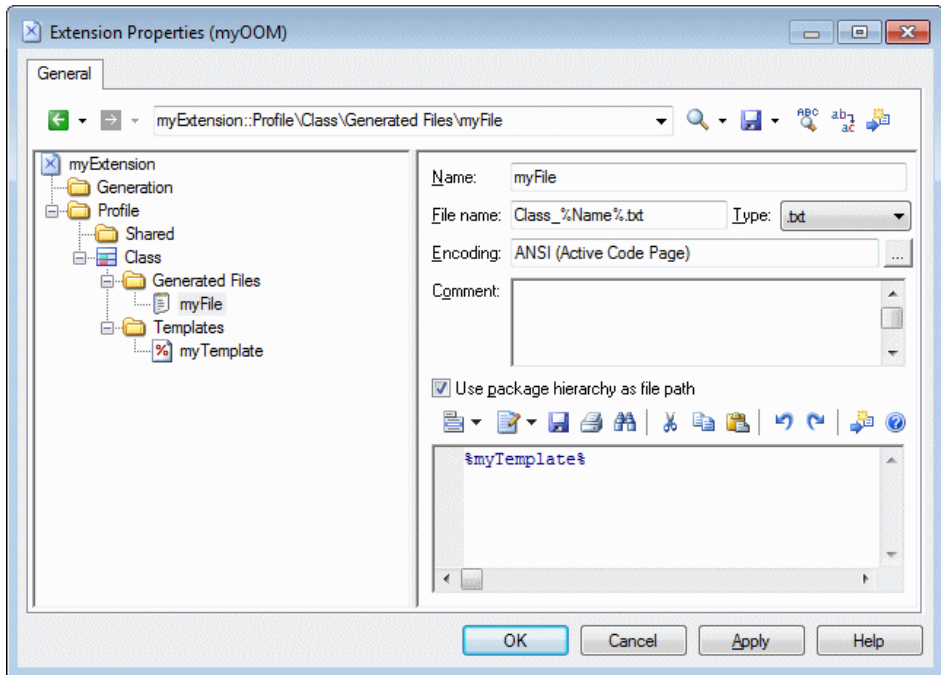
| Property | Description |
|------------------------------------|---|
| Name | Specifies a name for the generated file item in the resource editor. If an extension attached to the model contains a generated file name identical to one defined in the main resource file, then only the extension generated file will be generated. |
| 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>%code%.xml</code> . 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: <ul style="list-style-type: none"> • 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 a hierarchy of file directories. |

3. Enter GTL code (see *Chapter 5, Customizing Generation with GTL* on page 245) or the name of a template to populate the file in the text zone.

In the following example, a generated file is defined for OOM classes. A file will be generated for each class in the model with a name derived from the class `%Name%`, and

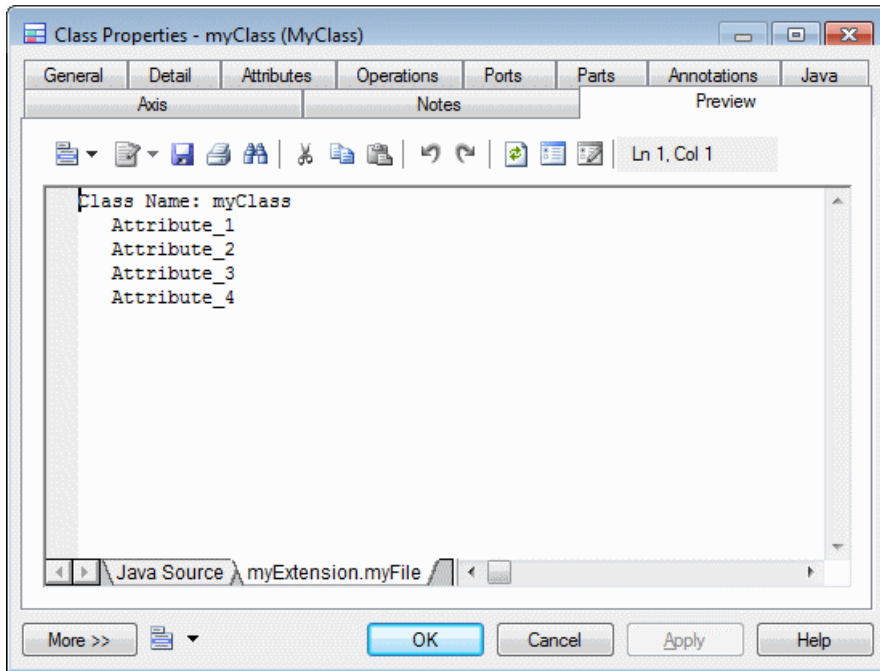
CHAPTER 2: Extension Files

containing the contents generated from the `%myTemplate%` template (see *Templates (Profile)* on page 84):



4. Click **OK** to save your changes and close the resource editor.

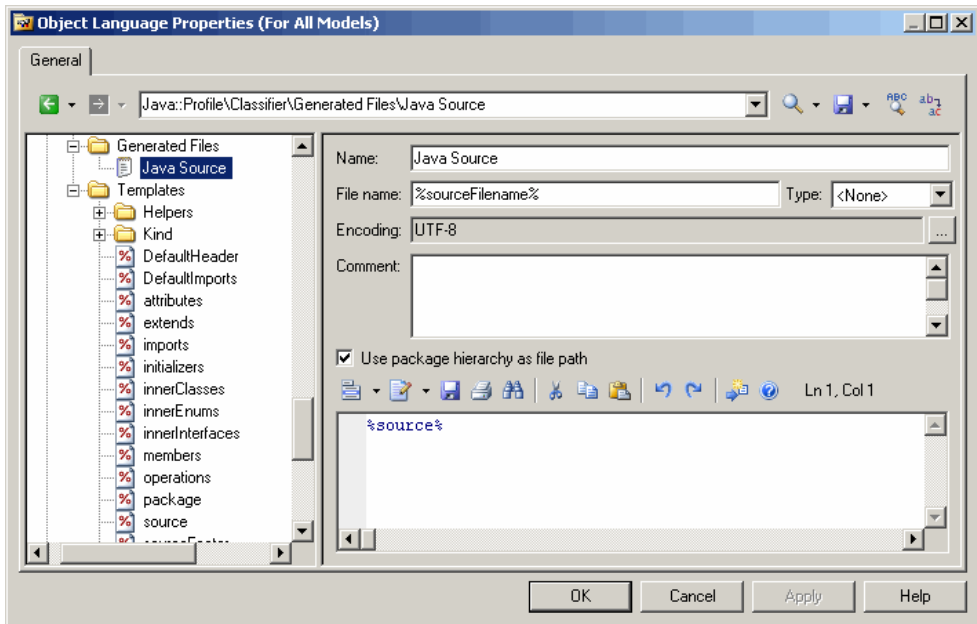
The file is immediately available as a sub-tab on the **Preview** tab of the object property sheet:



Example: JavaGenerated File and Templates

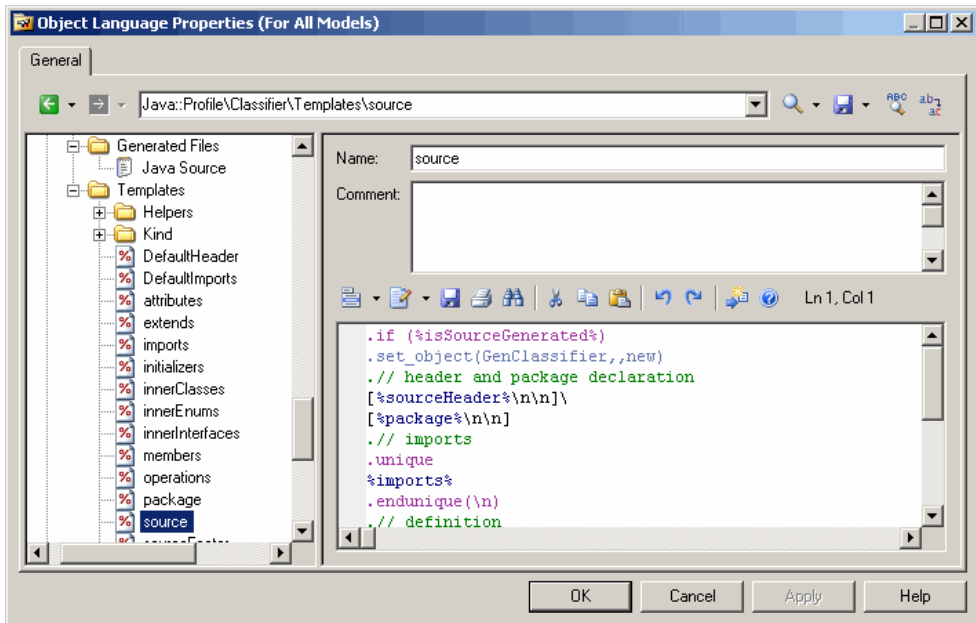
Templates contain GTL code used to generate text fragments from PowerDesigner property values, while generated files are used to assemble templates for generation as files or for previewing on the object property sheet **Preview** tab.

In this example, a generated file called `Java Source` is defined for classifiers. A file will be generated for each classifier in the model with a name derived from the `%sourceFilename%` template specified in the **File name** field, and containing the contents generated from the `%source%` template:



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, to a **Results** dialog in which you select the template to navigate to.

The referenced template, `source`, contains GTL code, including references to further templates called `%isSourceGenerated%`, `%sourceHeader%`, `%package%`, and `%imports%`:



Generating Your Files in a Standard or Extended Generation

You can use generated files to extend the standard generation for objects from OOMs, BPMs, and XSMs or to create a separate extended generation for any type of model. For extended generations, you can define a custom menu command.

To extend the standard BPM, OOM, or XSM generation from the Resource Editor:

1. Select the **Complement language generation** property in the root of the extension file (see *Extension File Properties* on page 14) to have the extension file appear for selection on the **Generation** dialog **Targets** tab.
2. Define generated files as appropriate.
3. [optional] Define options in `Generation\Options` (see *Example: Adding a Generation Option* on page 113) to have them appear on the **Generation** dialog **Options** tab.
4. [optional] Define commands in `Generation\Commands` and reference these commands in tasks (see *Example: Adding a Generation Command and Task* on page 114) to have them appear on the **Generation** dialog **Tasks** tab.

Alternatively, to define separate file generations apart from the standard language generation for a PDM or any type of model and make them available via the **Tools > Extended Generation** command

1. [OOM, BPM, and XSM only] Deselect the **Complement language generation** property in the root of the extension file (see *Extension File Properties* on page 14).

CHAPTER 2: Extension Files

2. Define generated files as appropriate.

The generation is immediately available on the **Targets** tab of the **Generation** dialog when you select **Tools > Extended Generation**.

3. [optional] Create a command in the **Tools** menu to directly access your extended generation in its own dialog:

- a. Create a method in `Profile\Model` with the name you want to give to your command, and enter the following code (where `extension` is the code of the extension file):

```
Sub %Method%(obj)

    Dim selection ' as ObjectSelection

    ' Create a new selection
    set selection = obj.CreateSelection

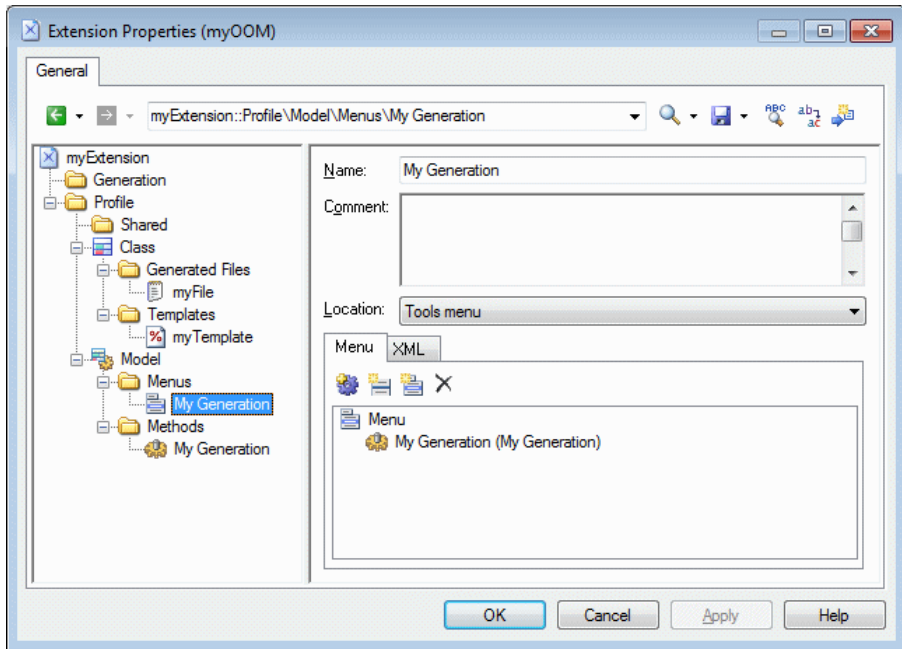
    ' Add object of the active selection in the created selection
    selection.AddActiveSelectionObjects

    ' Generate scripts for specific target
    InteractiveMode = im Dialog
    obj.GenerateFiles "", selection, "extension"

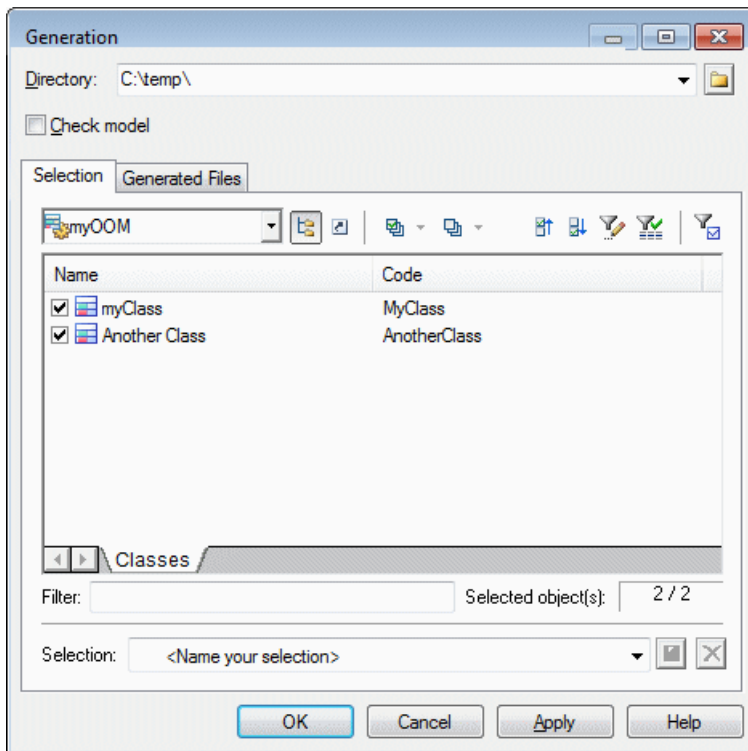
End Sub
```

For more information about methods, see *Methods (Profile)* on page 79.

- b. Create a menu in `Profile\Model` and select the `Tools` menu in the **Location** list (see *Menus (Profile)* on page 81).
- c. Add the method to the menu using the **Add Command** tool:



- d. Select the command specified (for example, **Tools > My Generation**) to open a custom **Generation** dialog, which does not have a **Targets** tab:



Transformations (Profile)

Transformations define sets of actions to modify objects either before or after a model generation or on request. Transformations are commonly grouped together in transformation profiles.

Transformations can be used to:

- Implement *Model Driven Architecture (MDA)*, which uses UML modeling to describe an application at different levels of detail. PowerDesigner allows you to create an initial *platform-independent model (PIM)* (modeling the basic business logic and functionality) and refine it progressively in different models containing increasing levels of implementation and technology-dependent information through to a *platform-specific model (PSM)*. You can define transformations that will generate a more refined version of a model, based on the desired target platform, and changes made to the PIM can be cascaded down to the generated models.
- Apply design patterns to your model objects.
- 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 for 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 invoked:

- On demand, by selecting **Tools > Apply Transformations**.
 - Before or after model generation (see *Core Features Guide > Linking and Synchronizing Models > Generating Models and Model Objects*).
 - Via a user-defined menu command (see *Menus (Profile)* on page 81).
1. Right-click a metaclass or stereotype, and select **New > Transformation**.
 2. Enter an appropriate **Name** and, optionally, a **Comment** to explain its purpose.
 3. On the **Transformation Script** tab, enter a VBScript to perform the transformation.

In this example, which is created in an extension attached to a CDM under the `DataItem` metaclass, the script tests to see whether the data item has a list of values defined and, if this is the case (and a domain with this same list of values does not already exist in the CDM), creates a new domain with the list of values:

```
Sub %Transformation%(obj, trfm)

    Dim list
    list = obj.ListOfValues
    if not list = "" then
        output "transforming " & cstr(obj)

        ' Check if such a domain already exist
        Dim domn, found
        found = false
        for each domn in obj.Model.Domains
            if domn.ListOfValues = list then
                found = true
            end if
        next

        ' Create a new domain
        if not found then
            set domn = obj.Model.Domains.CreateNew()
            domn.SetNameAndCode obj.Name, obj.Code
            domn.ListOfValues = list
        end if
    end if

End Sub
```

This transformation can be added to a transformation profile as a:

- Pre-generation transformation - The transformation is called from the Generation Options dialog. The domains are created temporarily in the CDM before generation and then are generated to the target model (for example, to a PDM).
 - Post-generation transformation - The transformation can be called from the Generation Options dialog (for a CDM-CDM generation). The domains are created in the target CDM after generation. Alternatively, the transformation can be called at any time by selecting **Tools > Apply Transformations** to create the domains in the existing model.
4. [optional] Review the **Global Script** tab (see *Global Script (Profile)* on page 105), which provides access to definitions shared by all VBscript functions defined in the profile, and the **Dependencies** tab, which lists the transformation profiles in which the transformation is used.

Transformation Profiles (Profile)

A transformation profile groups transformations together, and makes them available during model generation or by selecting **Tools > Apply Transformations**.

1. [if the Transformation Profiles category is not present] Right-click the root node, select **Add Items**, select `Transformation Profiles`, and click **OK** to create this folder.
2. Right click the `Transformation Profiles` folder, and select **New** to create a transformation profile.
3. Enter the following properties as appropriate:

| Property | Description |
|----------------------------------|---|
| Name / Comment | Specify the name of the transformation profile and provide an explanation of what it is intended to do. |
| Model Type / Family / Sub-family | [optional] Specify the type of model with which the transformation profile can be used during generation and (if the type supports a language definition file) the family and subfamily. If one or more of these fields is completed, the profile will only be displayed if the model to be generated conforms to them. For example, if you define the transformation in a PDM or PDM extension and specify <code>Object-Oriented Model</code> and <code>Java</code> , then the profile will only be available when you select to generate the PDM into a Java OOM. |

4. Click the **Pre-generation** tab and click the **Add Transformations** tool to add transformations to perform prior to generation.
 These transformations are executed before generation on the objects in your source model. If objects are created by these transformations then they are automatically added to the list of objects to be generated. Any changes to existing objects or new objects created by these transformations are reversed after generation, so that your model returns to its previous state.
5. Click the **Post-generation** tab and click the **Add Transformations** tool to add transformations to perform after generation. Transformations added on this tab are also

made available to apply outside of the context of a generation by selecting **Tools > Apply Transformations**.

These transformations are executed on the objects generated in your target model.

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

Developing Transformation Scripts

Transformation scripts are written in VBScript using a certain number of special methods. Transformation scripts do not require as many checks as standard scripts, because they are always implemented in a new, empty, temporary model, which is merged with the generation target model.

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.

The following methods are available when writing a transformation script:

- `CopyObject (source [, tag])`

Duplicates an existing object, sets a source for the duplicated object, and returns a copy of the new object.

- `SetSource (source, target [, tag])`

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.

- `GetSource (target [, tag])`

Retrieves the source object of a generated object.

- `GetTarget (source [, tag])`

Retrieves the target object of a source object.

Internal transformation objects are preserved when the transformations are used via the **Apply Transformations** or a custom menu command, so that they can be re-executed if you subsequently update (regenerate) the model. For example, you generate a CDM entity A to an OOM class B and then apply a transformation to class B in order to create class C. If you make changes to entity A and repeat the generation to update the OOM, class B is updated and the transformation is automatically reapplied to update class C.

XML Imports (Profile)

XML imports allow you to define mappings between an XML schema and the PowerDesigner metamodel (and any extensions) to enable the import of XML files complying with the schema. You can specify initialization and post-processing scripts to manage complexities in the import.

For an overview of creating, deploying, and using XML imports, see *Core Features Guide > Modeling with PowerDesigner > Objects > Importing Objects from XML Files*.

1. [if the XML Imports category is not present] Right-click the root node, select **Add Items**, select XML Imports, and click **OK** to create this folder.
2. Right click the XML Imports folder, and select **New** to create an XML import.
3. Enter the following properties as appropriate:

| Property | Description |
|------------------------|--|
| Name | Specifies the name of the import, which will be used as the name of the import command under File > Import . |
| First diagram | Specifies the first diagram that should be initialized in the model created from the imported file. |
| Create default symbols | Specifies to create symbols for the imported objects in the diagram. |
| File extension | Specifies the file extension that identifies XML documents that conform to the schema. |
| Comment | Provides an explanation of the import or other additional information. |

4. Click the **Schema** tab and click the **Import** tool to copy the schema, with any imports and includes resolved, to the extension file for mapping.

Warning! If the selected schema is too permissive and allows for too many possible object hierarchies it may not be possible to display it fully in the Mapping Editor. If you have an example XML data file to import, you can import this in place of the schema by clicking the **Import from Sample** tool and PowerDesigner will deduce a partial schema from it. Note that while a schema obtained in this way may successfully import the sample data file, other documents based on the same schema may not be complete if they contain other types of objects (or attributes or collections) that, though valid for the schema, were not in the first document.

You can click the **View as Model** tool to open the schema as an XML schema model.

5. [optional] Click the **Extensions** tab and select extension files containing extensions to the standard PowerDesigner metamodel to provide additional metaclasses (see *Extended Objects, Sub-Objects, and Links (Profile)* on page 36), attributes (see *Extended Attributes*

(*Profile*) on page 41), and collections (see *Extended Collections and Compositions (Profile)* on page 48) to map your XML schema to.

Attaching extension files in this way allow you to reuse previously defined extensions in your imports or to share extensions between imports. You can also define extensions under the Profile category in the resource file containing the XML import definition, or create them dynamically when creating your import mappings.

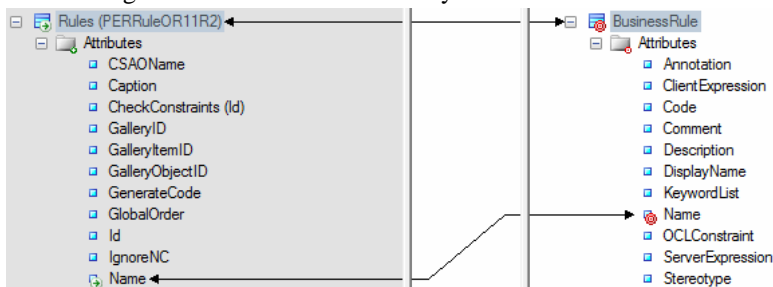
6. [optional] Click the **Initialization** tab and enter VBScript to run at model creation time before the importing of any objects. You can access shared library functions and static attributes defined for reuse in the resource file from the **Global Script** tab (see *Global Script (Profile)* on page 105).
7. [optional] Click the **Post-Process** tab and enter VBScript to run after all the objects have been imported.
8. Click the **General** tab and click the **Mappings** button to define mappings from the metaclasses identified in your XML schema to those in the PowerDesigner metamodel in the Mapping Editor (see *XML Import Mappings* on page 97).
9. Click **Apply** to save your changes.

XML Import Mappings

You control how elements defined in an XML schema are imported by mapping them and their attributes, compositions, and aggregations to objects in the PowerDesigner metamodel. The XML schema is analyzed and presented as a list of metaclasses on the left side of the Mapping Editor and the PowerDesigner metamodel (and any extensions) are displayed on the right side.

Note: It is not necessary to map all metaclasses (or all their contents), but only those with which you want to work. If the PowerDesigner metamodel does not contain appropriate metaclasses, attributes, compositions, or aggregations to map against, you can create them dynamically here or save any existing mappings, close the Mapping Editor, define or attach appropriate extensions, and then reopen the Mapping Editor to map to them.

1. Drag and drop an external metaclass to a PowerDesigner metaclass to create an import mapping. Any external attributes and collections are automatically mapped to PowerDesigner attributes with which they share a name:



By default, the Mapping Editor lists the standard attributes and collections of metaclasses, which are normally displayed in object property sheets. To display all available properties,

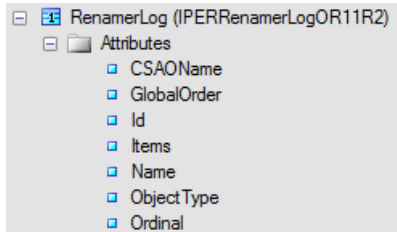
click the **Filter Properties** tool, and select *Show All Properties*. You can also filter the tree by using the **Filter Mappings** and **Filter Objects** tools.

Note: If no suitable metaclass exists, to create and map to a new extended metaclass based on the `ExtendedObject` metaclass, drag and drop the external metaclass onto the PowerDesigner metamodel root.

2. Drag and drop additional attributes under the metaclass to PowerDesigner attributes with compatible data types to create mappings for them. Attributes are contained in a folder under the metaclass and represent individual properties such as Name, Size, DimensionalType, which have boolean, textual, numeric, or object ID values:

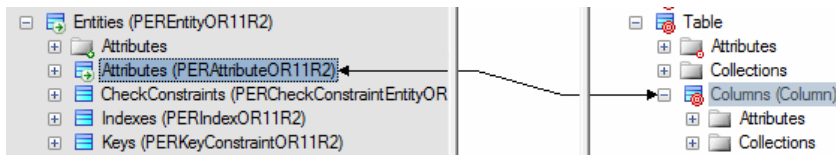


PowerDesigner identifies sub-object metaclasses in the schema that are limited to a single instance and displays a 1 overlay on their icons. Attributes under such metaclasses are treated as belonging to the parent metaclass and can be mapped to attributes under the PowerDesigner object with which the parent is mapped:



Note: If no suitable attribute exists, to create and map to a new extended attribute, drag and drop the external attribute onto the PowerDesigner metaclass to which its parent is mapped.

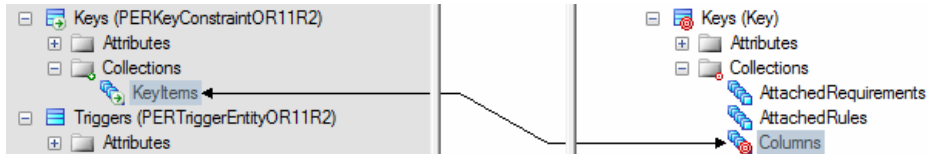
3. Drag and drop external sub-object metaclasses (compositions) under the metaclass to PowerDesigner compositions to create mappings between them:



Any attributes under the sub-object metaclass are automatically mapped to PowerDesigner attributes with which they share a name. Map other sub-object attributes as necessary.

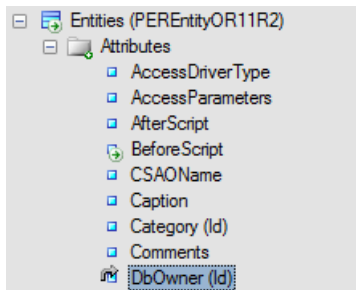
Note: In certain circumstances, it may be appropriate to map an external sub-object metaclass to a PowerDesigner object metaclass, and so such mappings are also permitted.

4. Drag and drop external collections (aggregations) under the metaclass to PowerDesigner collections to create mappings between them:

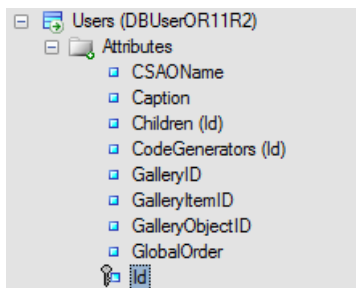


5. In certain schemas, it may be necessary to identify attributes as references and identifiers to link one metaclass to another through aggregation:

- a) Right-click an attribute and select **Declare as Object Reference** to specify that it acts as a pointer to another object. Such attributes often have a type of GUID, Token, or NCName (PowerDesigner automatically identifies attributes of type IDRef as references). A rounded arrow overlay is added to the attribute icon:



- b) Open the metaclass that the object reference points to, select its identifying attribute, right-click it, and select **Declare as Unique Identifier**. A key overlay is added to the attribute icon:



- c) The object reference attribute can now be mapped to a PowerDesigner attribute of type object (which also bears a rounded arrow overlay):



6. [optional] Select a metaclass and enter an initialization or post-processing script to modify the objects at or after creation (see *Metamodel Mapping Properties* on page 100).

- [optional] Click the target model (root node) to display the global list of mappings in the **Mappings** pane at the bottom of the dialog and use the arrows at the bottom of the list to change the order in which objects are imported to ensure that dependencies are respected.

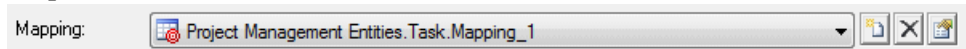
Note: To control the order in which attributes, compositions, and aggregations are imported within objects, select the target metaclass to display its mappings in the **Mappings** pane, and use the arrows at the bottom of the lists on the **Attribute Mappings**, **Collection Mappings**, and **Sub-Object Mappings** sub-tabs.

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

Metamodel Mapping Properties

Metamodel mappings are mappings between metamodel objects, which control how objects are imported or generated. Metamodel mappings are sub-objects of the PowerDesigner metamodel object on which they are defined.

To open a metamodel mapping property sheet, select the mapping from the list at the top of the Mapping Editor **Mappings** pane or parent object property sheet **Mapping** tab and click the **Properties** tool.



The tabs available on a particular mapping property sheet depend on the objects being mapped. The **General** tab contains the following properties:

| Property | Description |
|-----------------|--|
| Source object | Specifies the metamodel object being mapped to the target object. |
| Target object | Specifies the metamodel object being mapped from the source object. This object is the parent of the mapping itself. |

| Property | Description |
|-----------------------|--|
| Transformation script | <p>[metaattribute mappings] Specifies a script to set the value of the attribute. In the following example, from an XML import, the <code>notnullable</code> attribute is imported to the <code>Mandatory</code> attribute and, because the sense of the attributes is reversed, the boolean value imported is set to the opposite of the source value:</p> <pre data-bbox="387 361 1171 435">Sub %Set%(obj, sourceValue) obj.SetAttribute "Mandatory", not sourceValue End Sub</pre> <p>In the following example, from an object generation, the <code>NumberID</code> attribute is generated to the <code>Comment</code> attribute and a text string is prepended to make clear the origin of the value:</p> <pre data-bbox="387 569 1171 743">Function %AdjustValue%(sourceValue, sourceObject, targetObject) Dim targetValue targetValue = "The original process NumberID is " +cstr(sourceValue) %AdjustValue% = targetValue End Function</pre> |

The following tabs are also available for metaclass mappings:

- **Initialization** - Specifies a script to initialize the metaclass to be created. In the following example, the value of the `Stereotype` attribute is set to `SimpleType`:

```
Sub %Initialize%(obj)
  obj.Stereotype = "SimpleType"
End Sub
```

- **Attribute Mappings** - Lists the mappings of attributes under the metaclass. Select a mapping and click the **Properties** tool to open its property sheet. To control the order in which attributes are created, in order to respect dependencies between them, use the arrows at the bottom of the list.
- **Collection Mappings** - Lists the mappings of collections under the metaclass.
- **Post-Process** - Specifies a script to modify the metaclass after creation and execution of mappings. In the following example, the value of the **Code** attribute is copied to the **Name** attribute:

```
Sub %PostProcess%(obj)
  ' Copy code into name
  obj.Name = obj.Code
End Sub
```

Metamodel Object Properties

To view the properties of metaclasses, metaattributes, and metacollections displayed in the Mapping Editor, double-click the object node in the Mapping Editor or right-click the node and selecting **Properties**.

The **General** tab contains the following properties:

| Property | Description |
|----------------------------|---|
| Parent | [metaattributes and metacollections] Specifies the metaclass to which the metaobject belongs. |
| Parent collection | [sub-objects/compositions] Specifies the name of the composition collection that contains the sub-objects under the parent object. |
| Name | Specifies the name of the metaclass in the PowerDesigner metamodel or XML schema. |
| Data type | [metaattributes] Specifies the data type of the attribute. |
| Identifier | [metaattributes] Specifies that the attribute is used to identify the metaclass for referencing by another metaclass. |
| Reference / Reference path | [metaattributes and metacollections] Specifies that the attribute or collection is used to point to another metaclass to form an aggregation. |
| Singleton | [metaclasses] Specifies that only one instance of the metaclass is possible under each parent object. |
| Comment | Provides additional information about the metaobject. |

The following tabs are also available for metaclasses:

- **Attributes** - Lists the metaattributes belonging to the metaclass. Select an attribute in the list and click the **Properties** tool to open its property sheet.
- **Collections** - Lists the metacollections belonging to the metaclass. Select a collection in the list and click the **Properties** tool to open its property sheet.

Object Generations (Profile)

Object generations allow you to define mappings between one PowerDesigner model type and another based on the two metamodels (and any extensions) to enable the generation of one or more object types.

For an overview of creating, deploying, and using object generations, see *Core Features Guide > Linking and Synchronizing Models > Generating Models and Model Objects > Generating Model Objects > Defining Advanced Object Generations*.

1. [if the Object Generations category is not present] Right-click the root node, select **Add Items**, select **Object Generations**, and click **OK** to create this folder.
2. Right click the **Object Generations** folder, and select **New** to create an object generation.

3. Enter the following properties as appropriate:

| Property | Description |
|-------------------|---|
| Target model type | Specifies the type of model that will be created or updated by the generation. |
| Menu command name | Specifies the name of the command that will appear in the interface under Tools > Generate Objects . This field is initialized when you select a target model type. |
| Comment | Provides a description of the generation or other additional information. |

4. [optional] Click the **Source Extensions** and/or **Target Extensions** tab and select extension files containing extended attributes, collections, or metaclasses to reference in your mappings.

Attaching extension files in this way allow you to reuse previously defined extensions in your generations or to share extensions between generations. You can also define extensions as appropriate under the Profile category in the resource file containing the generation definition.

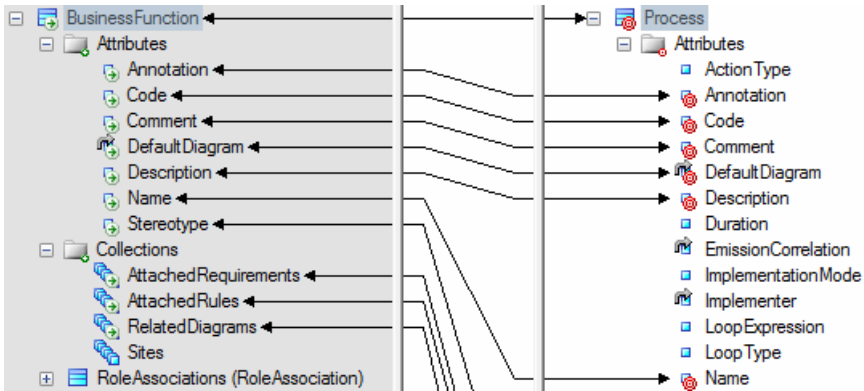
5. Click the **Mappings** button to define mappings from your source to target metaclasses in the Mapping Editor (see *Model-to-Model Generation Mappings* on page 103).
6. Click **Apply** to save your changes.

Model-to-Model Generation Mappings

You control how metaclasses from one PowerDesigner model type will be generated to metaclasses in another model type by mapping them and their attributes and collections in the Mapping Editor. Any extensions defined for the source or target metamodels are displayed and available for mapping.

Note: It is not necessary to map all metaclasses (or all their contents), but only those with which you want to work. If the PowerDesigner metamodel does not contain appropriate metaclasses, attributes, compositions, or aggregations to map against, you should save any existing mappings, close the Mapping Editor, define or attach appropriate extensions, and then reopen the Mapping Editor to map to them.

1. Drag and drop a metaclass from the source pane on the left to a metaclass in the Target pane on the right. Any source attributes are automatically mapped to target attributes with which they share a name:



Note: By default, the Mapping Editor lists the standard attributes and collections of metaclasses, which are displayed, by default, in object property sheets. To display all available properties, click the **Filter Properties** tool, and select **Show All Properties**. You can also filter the tree by using the **Filter Mappings** and **Filter Objects** tools.

2. Drag and drop additional source attributes under the metaclass to target attributes with compatible data types to map them. Attributes are contained in a folder under the metaclass and represent individual properties such as Name, Size, DimensionalType, containing boolean, textual, numeric, or object ID values:
3. Drag and drop source sub-object metaclasses (compositions) under the metaclass to target compositions to create mappings between them:

Any attributes under the source sub-object metaclass are automatically mapped to target attributes with which they share a name. Map other sub-object attributes as necessary.

Note: In certain circumstances, it may be appropriate to map a source sub-object metaclass to a target object metaclass, and so such mappings are also permitted.

4. Drag and drop source collections (aggregations) under the metaclass to target collections to create mappings between them:
5. [optional] Select a metaclass and enter an initialization or post-processing script to modify the objects at or after creation (see *Metamodel Mapping Properties* on page 100).
6. [optional] Click the target model (root node) to display the global list of mappings in the **Mappings** pane at the bottom of the dialog and use the arrows at the bottom of the list to change the order in which objects are generated to ensure that dependencies are respected.

Note: To control the order in which attributes, compositions, and aggregations are generated, select the target metaclass to display its mappings in the **Mappings** pane, and use the arrows at the bottom of the lists on the **Attribute Mappings**, **Collection Mappings**, and **Sub-Object Mappings** sub-tabs.

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

Global Script (Profile)

The profile contains a global script, which you can use to store functions and variables to be reused in your scripts defined for extensions.

For example, we could imagine writing a function for obtaining the data type of an item and reusing it in the scripts for both the custom check and autofix examples (see *Custom Checks (Profile)* on page 71).

The new `DataTypeBase` function is entered on the **Global Script** tab 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
```

The script for the check (see *Example: PDM Custom Check* on page 73 can be rewritten to call the function as follows:

```
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
```

Note: Variables defined on the **Global Script** tab are reinitialized each time they are referenced in another script.

Object, Process, and XML Language Definition Files

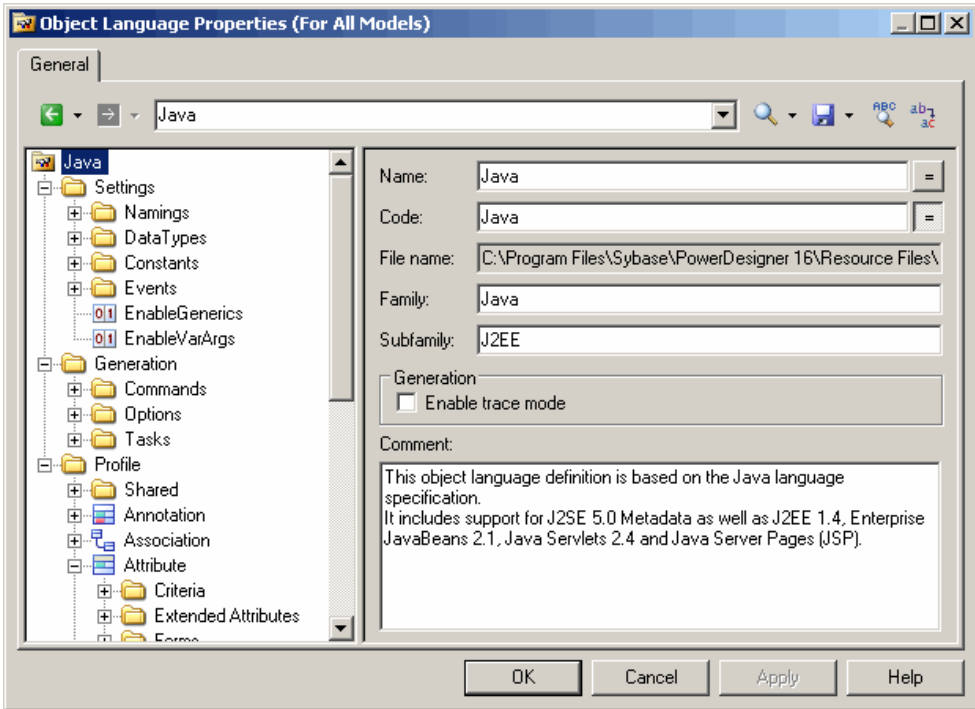
Language definition files provide PowerDesigner with the information necessary to model, reverse-engineer, and generate for a particular object-oriented, business process, or XML language. PowerDesigner provides definition files for many popular languages. You select a language when you create an OOM, BPM, or XSM.

Language definition files have an `.xol`, `.xpl`, or `.xsl` extension and are located in `install_dir/Resource Files`. To view the list of languages, select **Tools > Resources > Object Languages > , Process Languages, or XML Languages**. For information about the tools available in resource file lists, see *Chapter 1, PowerDesigner Resource Files* on page 1.

Note: The PDM uses a different form of definition file (see *Chapter 4, DBMS Definition Files* on page 119), and other model types do not have definition files but can be extended with extension files (see *Chapter 2, Extension Files* on page 11).

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.



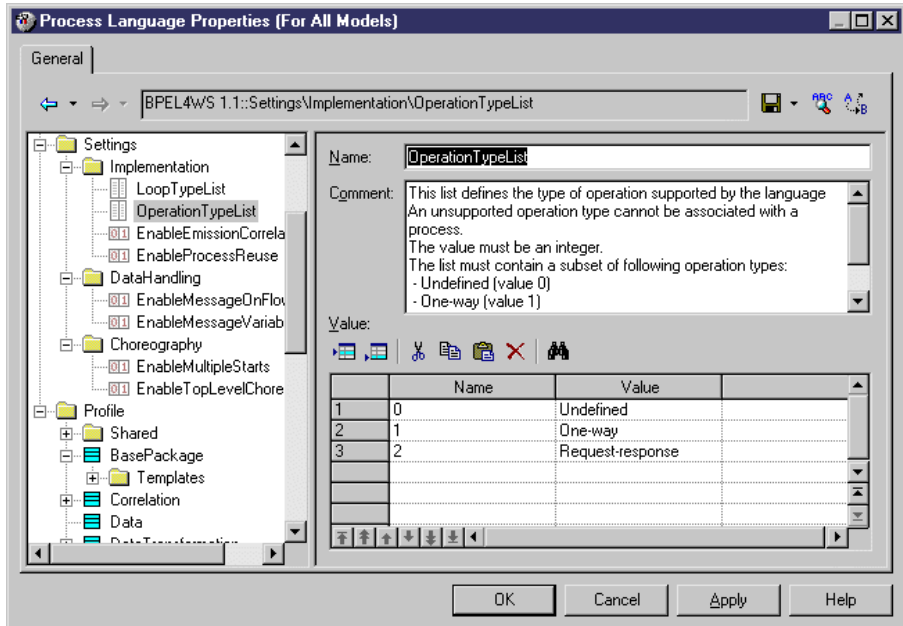
The root node of each file contains the following properties:

| Property | Description |
|---------------------|---|
| Name / Code | Specify the name and code of the language definition file. |
| File Name | [read-only] Specifies the path to the language definition 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 / Sub-family | Specifies the family and subfamily of the language, which may enable certain non-default features in the model. For example, object languages of the Java, XML, IDL and PowerBuilder® families support reverse engineering. |
| Enable Trace Mode | Lets you preview the templates used during generation (see <i>Templates (Profile)</i> on page 84). Before starting the generation, click the Preview page of the relevant object, and click the Refresh tool to display the templates. When you double-click on a trace line from the Preview page, the Resource Editor opens to the corresponding template definition. |
| 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 deduced from the operation type. You can specify:
 - 0 (default) - the action type cannot be deduced and must be specified
 - 1 - the language enforces a Request-Response and a One-Way operation to be received by the process and a Solicit-Response and a Notification operation to be invoked by the process
 - 2 the language ensures that a Solicit-Response and a Notification operation are always received by the process while Request-Response and One-Way operations are always invoked by the process



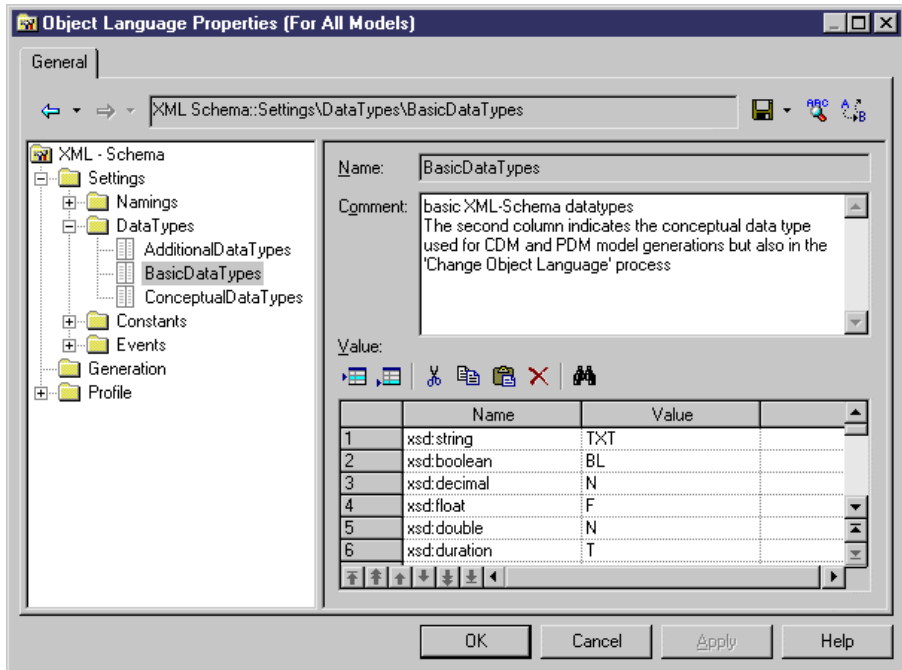
- *DataHandling* - [executable BPM only] Gathers options for managing data in the language. The following constant values are defined by default:
 - *EnableMessageOnFlow* - indicates if a message format can be associated to a flow or not. The default value is Yes
 - *EnableMessageVariable* - enables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

Settings Category: Object Language

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

- *Data Types* - Tables for mapping internal data types with object language data types. The following data types values are defined by default:

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



- *Constants* - contains mapping between the following constants and their default values: Null, True, False, Void, Bool.
- *Namings* - contains parameters that influence what will be included in the files that you generate from an OOM:
 - *GetterName* - Name and value for getter operations
 - *GetterCode* - Code and value for getter operations
 - *SetterName* - Name and value for setter operations
 - *SetterCode* - Code and value for setter operations
 - *IllegalChar* - lists illegal characters for the object language. This list populates the Invalid characters field in **Tools > Model Options > Naming Convention**. For example, " / ! =<> " ' () "
- *Events* - defines standard events on operations. This category may contain default existing events such as constructors and destructors, depending on the object language. An event is linked to an operation, and the contents of the Events category is displayed in the Event list

in operation property sheets to describe the events that can be used by an operation. In PowerBuilder for example, the Events category is used to associate operations with PowerBuilder events.

Settings Category: 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 245), and must be included within tasks to be evoked.

The image shows a screenshot of a text editor containing GTL code. Several annotations with arrows point to specific parts of the code:

- An arrow points to `%$JAVAC%` with the text: "variable defined in General Options > Variables (environment variable)".
- An arrow points to `set_value(_JAVAC, "javac.exe")` with the text: "executable".
- An arrow points to `local` with the text: "local variable".
- An arrow points to `execute_command(%_JAVAC%, %javaFilepath%, cmd_PipeOutput)` with the text: "template for evaluating the path".
- An arrow points to `cmd_PipeOutput` with the text: "blocks until completion and shows the output".

```
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
local
foreach_item(ActiveModel.GeneratedClassifierList)
    .execute_command(%_JAVAC%, %javaFilepath%, cmd_PipeOutput)
next
```

- **Options** – contains options, available on the **Options** tab of the Generation dialog, 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%
```

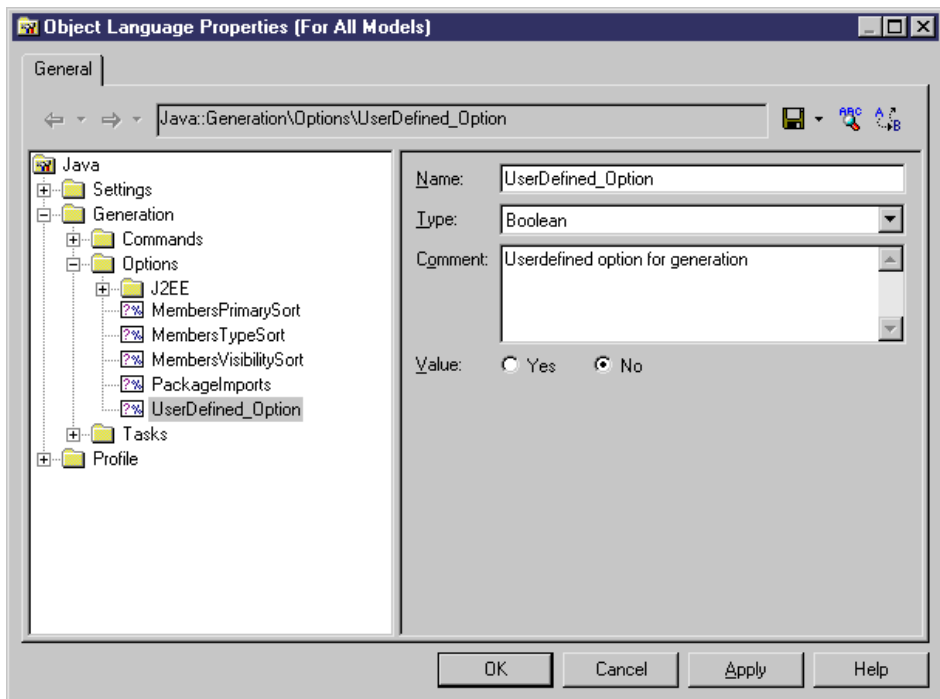
For example, for a boolean option named `GenerateComment`, `%GenOptions.GenerateComment%` will evaluate to either `true` or `false` in a template, depending on the value specified in the Generation dialog **Options** tab.

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

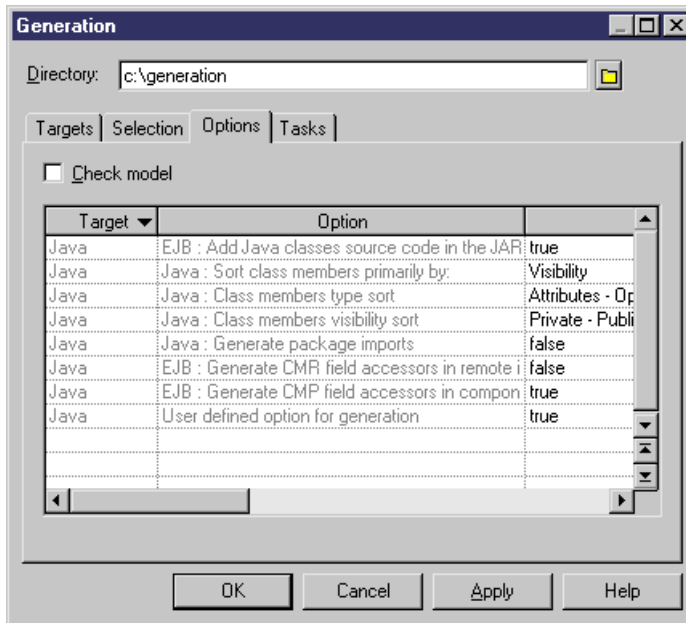
Example: Adding a Generation Option

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**:



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

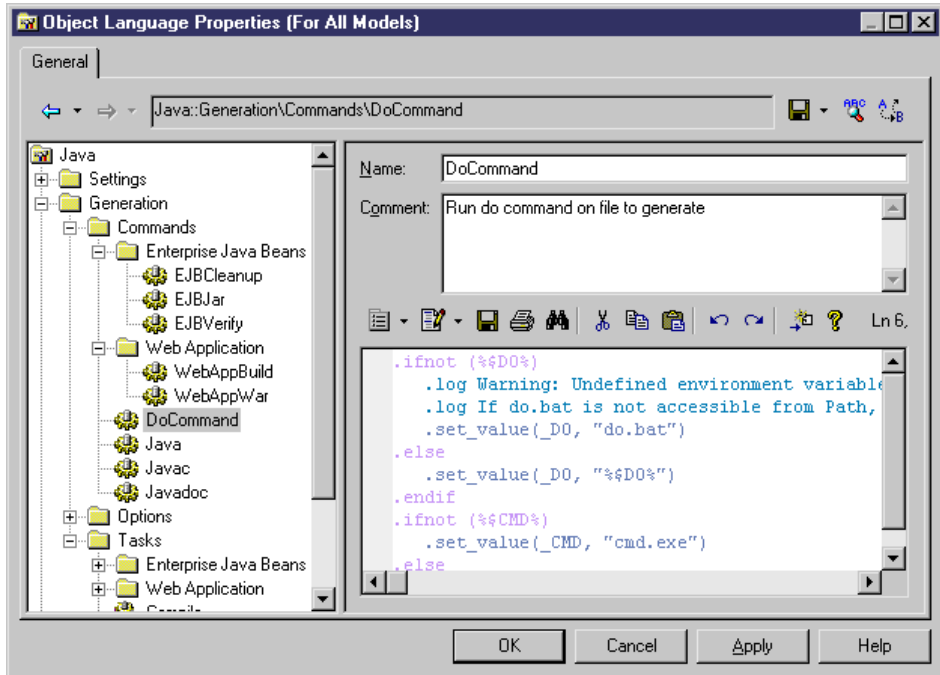


Note: For detailed information about creating and modifying generation templates, see *Chapter 5, Customizing Generation with GTL* on page 245.

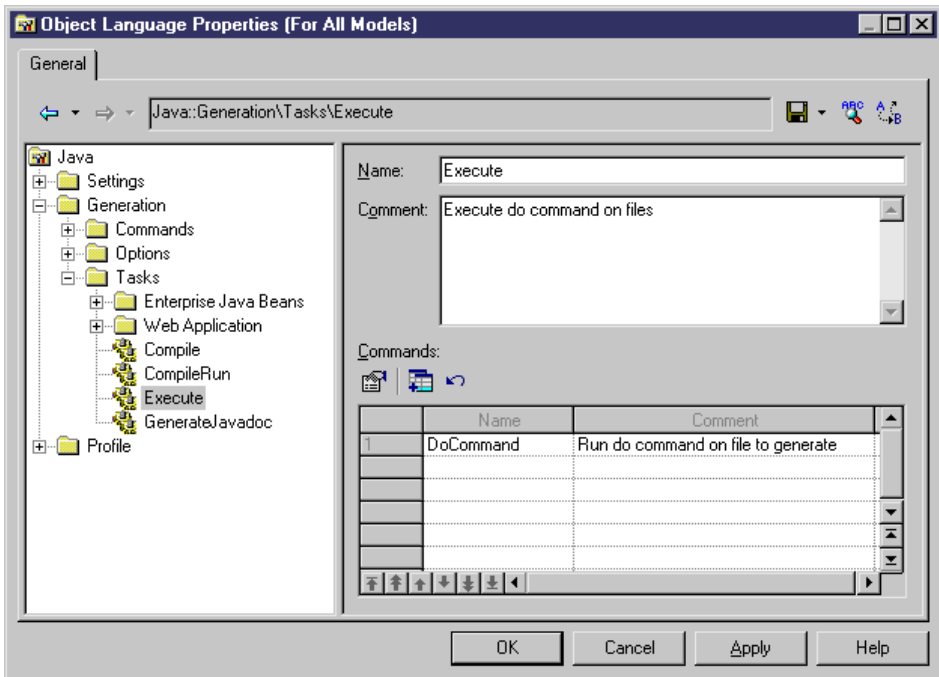
Example: Adding a Generation Command and Task

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

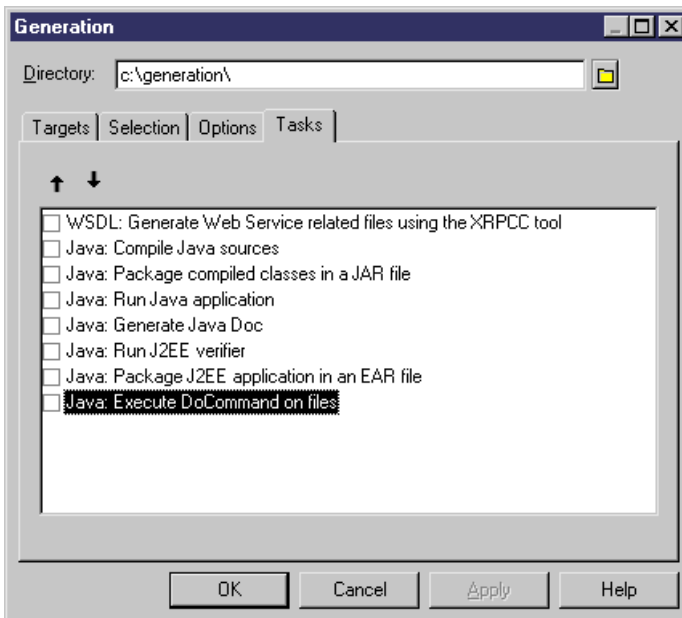
1. Create a new OOM for Java, and then select **Language > Edit Current Object Language**.
2. Expand the Generation category, and then right-click the Commands category and select **New**.
3. Name the command DoCommand and enter an appropriate template:



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



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



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 11.

CHAPTER 4 DBMS Definition Files

DBMS definition files provide PowerDesigner with the information necessary to model, reverse-engineer, and generate for a particular DBMS. PowerDesigner provides definition files for most popular DBMSs. You select a DBMS when you create a PDM.

DBMS definition files have an `.xdb` extension and are located in `install_dir/Resource Files/DBMS`. To view the list of DBMSs, select **Tools > Resources > DBMS**. For information about the tools available in resource file lists, see *Chapter 1, PowerDesigner Resource Files* on page 1.

You can consult or modify the DBMS definition file attached to your PDM in the Resource Editor by selecting **Database > Edit current DBMS**. 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.

Warning! We strongly recommend that you make a back up of the resource files delivered with PowerDesigner before editing them.

Each DBMS file has the following structure:

- *General* - contains general information about the database, without any categories (see *General Category (DBMS)* on page 134). 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 135)
 - *Format* - parameters for allowed characters (see *Format Category* on page 136)
 - *File* - header, footer and usage text items used during generation (see *File Category* on page 138)
 - *Keywords* - the list of SQL reserved words and functions (see *Keywords Category* on page 140)
 - *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 (DBMS)* on page 142).
 - *Data Type* - contains the list of valid data types for the specified DBMS and the corresponding types in PowerDesigner (see *Script/Data Type Category (DBMS)* on page 197).

- *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 (see *Transformations (Profile)* on page 92).
- *Profile* - allows you to define extended attribute types and extended attributes for database objects (see *Profile Category (DBMS)* on page 200).

The following properties are available on the root of a DBMS definition file:

| Property | Description |
|-------------|--|
| Name / Code | Name and code of the DBMS. |
| 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*

Database Generation and Reverse Engineering

PowerDesigner supports generation and reverse engineering of databases through scripts and live connections via SQL statements and queries stored in the `Script/Objects` category. Generation and reverse-engineering of scripts and generation to a live connection all use the same statements, while reverse-engineering from a live connection uses separate queries.

PowerDesigner performs generation and reverse-engineering as follows:

- Generation/Update Database - Each model object selected is applied to the statements in the `Script/Objects` category.
- Reverse engineering:
 - Script - PowerDesigner parses the script and identifies object creation statements by comparing them with the statements in the `Script/Objects` category.
 - Live connection - PowerDesigner uses the queries in the `Script/Objects` category 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.

Script Generation

PowerDesigner can generate a SQL script from a PDM to create or modify a database. The statements that control script generation are available in the `Script/Objects` category.

When generating a SQL script, PowerDesigner takes each object to be created in turn, and applies the appropriate `Create` or other statement to create or modify the object:

- `Create` - Creates a new object.
- `Alter/Modify` - Modifies the attributes of an existing object.
- `Add` - Creates a new sub-object. If keys are defined inside a table, they will be created with an `Add` statement, but if they are created outside the table, then they will be created with a `table Modify` statement.
- `Rename` - Renames an object.
- `Drop` - Drops an object (for use when an `Alter` statement is not possible).
- `ObjectComment` - Adds a comment on the object.
- `Options` - Defines the physical options of an object.
- `ConstName` - Defines the constraint name template for object checks.

For example, in Sybase ASE 15.7, 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 using variables (see *Variables for Tables and Views* on page 216) that extract the necessary information from the object's properties. The `%TABLDEFN%` variable collects the `Add` items in the `Column`, `PKey`, `Key`, and `Reference` categories, and the `AddTableCheck` item in the `Table` category.

Other statements in the object categories are used to customize the PowerDesigner interface and behavior according to database features, such as `Maxlen`, `Permission`, `EnableOwner`, and `AllowedADT`.

Extending Generation with Before and After Statements

You can extend script generation statements to complement generation using the *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.

Extension statements are written in GTL (see *Chapter 5, Customizing Generation with GTL* on page 245). During generation, the statements and variables are evaluated and the result is added to the global script.

Note: We recommend that you avoid using GTL macros (other than `.if`) in generation scripts, as they may not be resolvable when reverse engineering by script. Generating and reverse engineering via a live database connection are not subject to this limitation.

Example - Adding an AfterCreate Statement

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

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

The `.if` macro evaluates variable `%ExtTablePartition%`, which is an extended attribute that contains the number of table partitions. If the value is higher than 1, then `%CreatePartition%`, defined in the `Table` category, will be generated as follows:

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

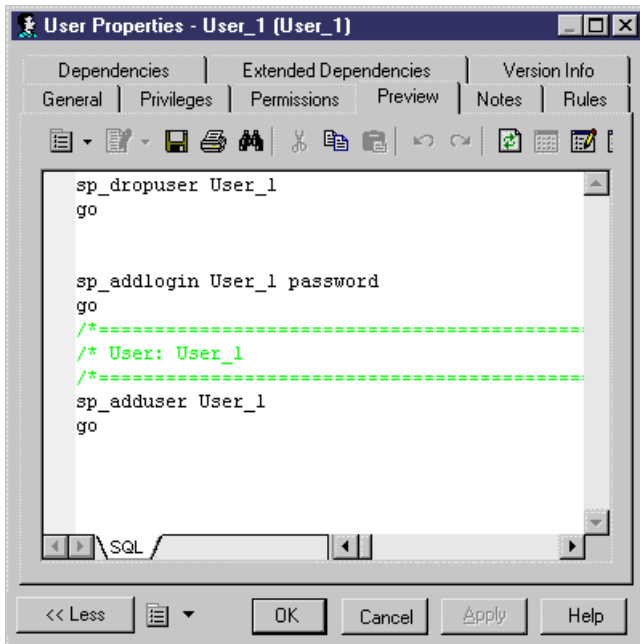
This item generates the statement for creating the number of table partitions specified in `%ExtTablePartition%`.

Example - Adding a BeforeCreate Statement

The extension statement `BeforeCreate` is defined in the `User` category to create the login of a user before the user `Create` statement is executed:

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

The automatically generated login will have the same name as the user, and its password. The `BeforeCreate` statement is displayed before the user creation statement in the **Preview**:



Example - 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
```

Script Reverse Engineering

PowerDesigner can reverse engineer SQL scripts into a PDM. The statements that control script generation are available in the *Script/Objects* category.

When reverse-engineering a SQL script into a PDM, PowerDesigner compares each statement in turn with all of the *Create* statements defined in the DBMS definition file and when it finds a match, extracts all of the available information to create or update PDM objects.

The statements used in script reverse engineering are the same as those for script generation (see *Script Generation* on page 121).

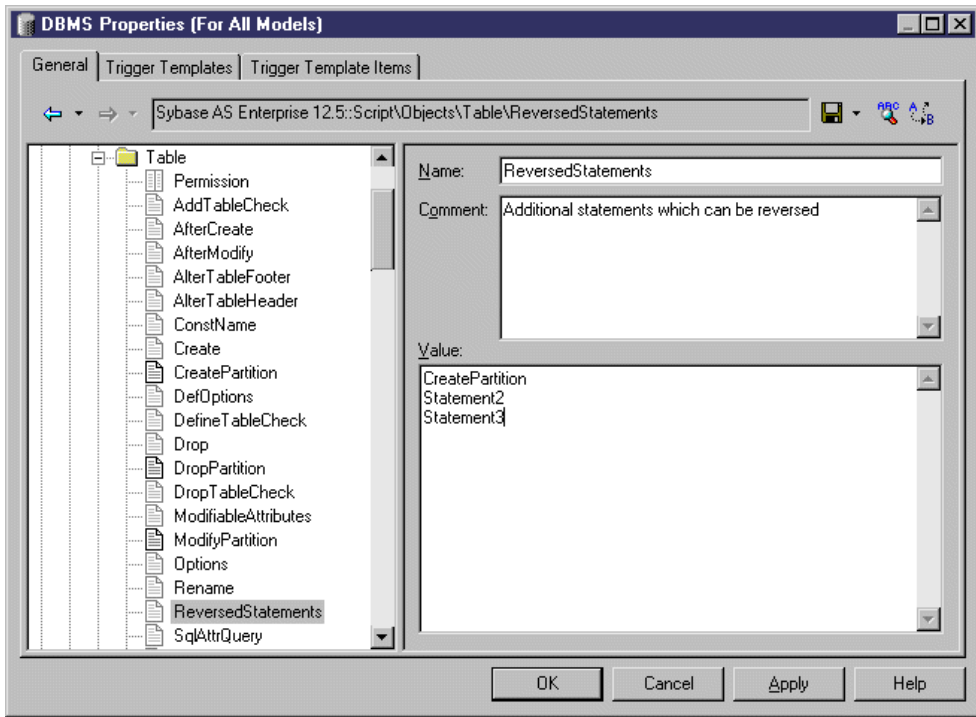
For example, in Sybase IQ v15.2, the *Create* statement in the *Table* category is the following:

```
create[%ExtGlobalTemporaryTable%? global temporary] table
[%QUALIFIER%]%TABLE% (
  %TABLDEFN%
) [.Z:[[%R%? [.O:[in][on]] %DBSpace%:[%DBSpace%?
  in %DBSpaceGeneratedName%]]][
  on commit %OnCommit%][%NotTransactional%? not transactional][
  at %.q:At%][%R%?partition by range %RevPartition%:[%PartitionKey
%?[%hasLifecycle%?:
  partition by range (%PartitionKey.Code%)
  (
    %PartitionDef%
  )]]]
]
```

This statement contains the parameters for creating the table together with its owner and physical options using variables (see *Variables for Tables and Views* on page 216) that extract the necessary information from the object's properties.

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

For example, the extension statement *AfterCreate* uses *CreatePartition*, which must be declared in *ReversedStatements* to be properly reverse engineered:



Live Database Generation

PowerDesigner can generate or modify a database from a PDM to a live connection. The statements that control live generation are available in the *Script/Objects* category, except when the DBMS does not support standard SQL syntax. For example, MS Access, which needs VB scripts to create database objects, has special generation statements defined in the *ODBC* category.

When generating to a live connection, PowerDesigner takes each object to be created in turn, and applies the appropriate `Create` or other statement to create or modify the object.

The statements used in live generation are the same as those for script generation (see *Script Generation* on page 121).

Live Database Reverse Engineering

PowerDesigner can reverse engineer from a live database connection into a PDM. The queries that control live reverse engineering are available in the *Script/Objects* category.

The following queries are used in live reverse engineering:

- `SqlListQuery` - Retrieves a list of available objects to populate the Database Reverse Engineering dialog. This query is memory intensive, and should retrieve the smallest

number of columns possible. If it is not defined, then `SqlAttrQuery` will be used to populate the dialog.

- `SqlAttrQuery` - Retrieves the object attributes to be reverse-engineered. This query is not necessary if the object has few attributes, and the `SqlListQuery` can retrieve all necessary information, as is the case for tablespaces in Sybase SQL Anywhere.
- `SqlOptsQuery` - Retrieves the physical options to be reverse-engineered.
- `SqlListChildrenQuery` - Retrieves lists of child objects (such as columns of an index or key or joins of a reference) to be reverse-engineered.
- `SqlSysIndexQuery` - Retrieves system indexes created by the database.
- `SqlChckQuery` - Retrieves object check constraints.
- `SqlPermQuery` - Retrieves object permissions.

Note: You can also create your own queries (see *Creating Queries to Retrieve Additional Attributes* on page 128).

Each type of query has the same basic structure comprised of a comma-separated list of PowerDesigner variables enclosed in curly braces { } followed by a select statement to extract values to populate these variables. The values of the returned records are stored in these variables, which are then committed as object attribute values.

For example, the `SqlListQuery` in the View category of Oracle 11g R1 extracts values for eight variables:

```
{OWNER, VIEW, VIEWSTYLE, ExtObjViewType,
  ExtObjOIDList, ExtObjSuperView, XMLSCHEMA EX, XMLELEMENT EX}

select
  v.owner,
  v.view_name,
  decode(v.view_type, 'XMLTYPE', 'XML', 'View'),
  v.view_type,
  v.oid_text,
  v.superview_name,
  decode(v.view_type, 'XMLTYPE', '%SqlXMLView.'||v.owner||
v.view_name||'1%', ''),
  decode(v.view_type, 'XMLTYPE', '%SqlXMLView.'||v.owner||
v.view_name||'2%', '')
from sys.all_views v
[where v.owner = %q:SCHEMA%]
```

Each comma-separated part of the header may contain the following:

- Name of variable - [required] can be any standard PDM variable (see *PDM Variables and Macros* on page 212), metamodel public name (see *Navigating in the Metamodel* on page 344) or the name of an extended attribute defined under the metaclass in the Profile (see *Profile Category (DBMS)* on page 200).
- ID - [optional] the variable is part of the identifier.

- . . . - [optional] the variable must be concatenated for all the lines returned by the SQL query that have the same values for the ID columns. The ID and . . . (ellipsis) keywords are mutually exclusive.
- Value pairs - [optional] lists conversions between retrieved values and PowerDesigner values in the following format (where * means all other values):

```
(value1 = PDvalue1, value2 = PDvalue2, * = PDvalue3)
```

Example: Using ID to Define the Identifier

In this script, the identifier is defined as TABLE + ISKEY+ CONSTNAME through the use of the ID keyword:

```
{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 the resulting lines returned by the SQL script, the values of the fourth field are 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, and PowerDesigner will process the contents to remove the last comma.

Example: Converting Value Pairs

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

```
{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
```

Creating Queries to Retrieve Additional Attributes

You can create queries to retrieve additional attributes. These attributes could be added to `SqlAttrQuery`, but retrieving them in a separate query helps to avoid overloading that item. User-created queries are only called during reverse-engineering if their names are added to the `ReversedQueries` item.

To create a new query in a category, right-click the category and select **New > Text Item**. Enter an appropriate name, and then add the name to the `ReversedQueries` item.

For example, in the Oracle family of DBMSs, `SqlColnListQuery` is defined in the View category:

```
{OWNER ID, VIEW ID, VIEWCOLN ...}

select
  c.owner,
  c.table_name,
  c.column_name||', '
from
  sys.all_tab_columns c
where 1 = 1
      [and c.owner=%q:OWNER%]
      [and c.table_name=%q:VIEW%]
order by
  1, 2, c.column_id
```

This query retrieves view columns, and is enabled by adding it to `ReversedQueries` in the View category.

Note: Subqueries that are called with the `EX` keyword from within `SqlAttrQuery` or other queries (see *Calling Sub-Queries with the EX Keyword* on page 128) do not need to be added to `ReversedQueries`.

Calling Sub-Queries with the EX Keyword

DBMS system tables may store information to be reversed in columns with `LONG`, `BLOB`, `TEXT` and other incompatible data types, which `PowerDesigner` cannot directly concatenate 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 are evaluated by sub-queries that 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||%)',
  ...
```

Note: Extended queries are not be added to the `ReversedQueries` item.

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

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

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 `FragDbbsp 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.

`FragDbbsp 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 |
|--------------------------|--|
| <code>FragDbbsp n</code> | Fragment location for fragment number <code>n</code> |
| <code>FragExpr n</code> | Fragment expression for fragment number <code>n</code> |
| <code>FragSprt n</code> | Fragment separator for fragment number <code>n</code> |

`SqlFragQuery` is defined as follows:

```
{A, a(E="expression", R="round robin", H="hash"), B, b, C, c, D,
d(0="", *=",")}
select
  'FragDbbsp' || f.evalpos, f.dbospace,
  'FragExpr' || f.evalpos, f.exprtext,
  'FragSprt' || f.evalpos, f.evalpos
from
  informix.systables t,
  informix.sysfragments f
where
  t.partnum = 0
  and f.fragtype='T'
  and t.tabid=f.tabid
[ and t.owner = '%SCHEMA%']
[ and t.tabname='%TABLE%']
```

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

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

```

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

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

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

Live Database Reverse Engineering Function-based Index

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

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

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

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

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

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

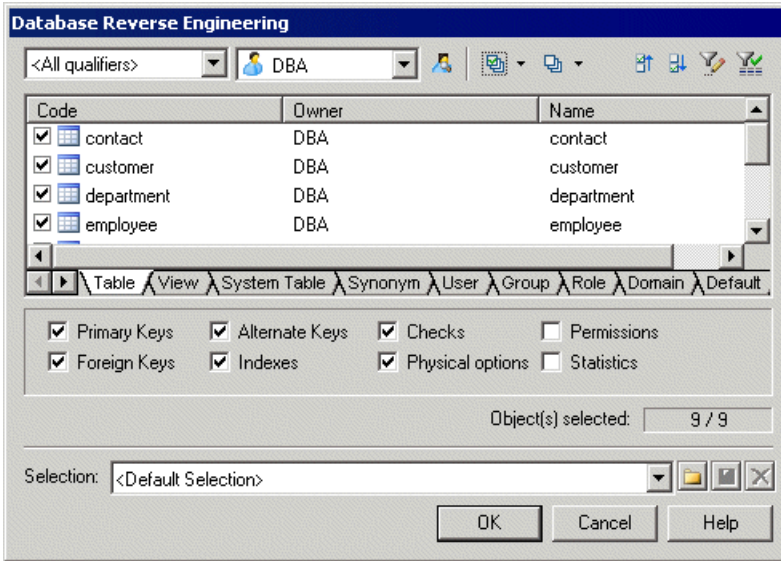
```
{VAR, VAL}

select
  'Xpr'||table_name||index_name||column_position,
  column_expression
from
  all_ind_expressions
where 1=1
[ and table_owner='%SCHEMA%']
[ and table_name='%TABLE%']
```

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

Live Database Reverse Engineering Qualifiers

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



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

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

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

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

Example

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

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

Generating and Reverse-Engineering PDM Extended Objects

Some DBMSs have objects that are not present in the standard PowerDesigner metamodel, and that must be represented as extended objects. PDM extended objects are defined in the *Profile* category, but their generation and reverse-engineering is controlled by statements and queries defined in the *Script/Objects* category.

Note: Before following this procedure, you must create an extended object in the *Profile* category (see *Extended Objects, Sub-Objects, and Links (Profile)* on page 36).

1. Right-click the *Script/Objects* category, select **Add Items**, select your new extended object in the list, and then click OK to add it to the list of objects
2. Right-click the new object entry, and select **Add Items** to add the necessary script items to it. As a minimum, to enable the generation and reverse engineering of the object, you must add the following items:
 - Create
 - Drop
 - AlterStatementList
 - SqlAttrQuery
 - SqlListQuery
3. Click **OK** to add these script items to your object, and enter the appropriate SQL statements and queries. You will need to enter values for each of these items. For guidance on syntax, see *Common Object Items* on page 144.
4. [optional] To control the order in which this and other objects will be generated, use the *Generation Order* item (see *Script/Objects Category (DBMS)* on page 142).

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**.
2. On the PdPDM sub-tab, select *Model* and then click **OK** to add the *Model* item to the *Profile* folder.
3. Right-click the *Model* item, and select **New > Event Handler** (see *Event Handlers (Profile)* on page 75).
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 add the selected event handlers to 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 (DBMS)

The General category is located directly beneath root, and contains high-level items that define the basic behavior of the DBMS.

| Item | Description |
|------------------|--|
| EnableCheck | Specifies whether the generation of check parameters is authorized. The following settings are available. If this item is set to No , no variables linked to check parameters will be evaluated during generation and reverse-engineering. |
| EnableConstName | Specifies whether constraint names are supported by the DBMS. If this item is set to Yes , table and column constraint names are generated in addition to the constraints themselves. |
| EnableIntegrity | Specifies whether integrity constraints are supported by the DBMS. If this item is set to Yes , primary, alternate, and foreign key check boxes are available for database generation and modification |
| EnableMultiCheck | Specifies whether the generation of multiple check parameters for tables and columns is supported by the DBMS. If this item is set to Yes , multiple check parameters are generated, with the first constraint concatenating all the validation business rules, and additional constraints generated for each constraint business rules attached to the object. If this item is set to No , all business rules (validation and constraint) are concatenated into a single constraint expression. |
| SchemaStereotype | Specifies the user stereotype to be used to indicate a schema (object owner). |
| SqlSupport | Specifies whether SQL syntax is supported by the DBMS. If this item is set to Yes , SQL syntax is supported and the SQL Preview is available. |
| UniqConstName | Specifies whether unique constraint names for objects are required by the DBMS. If this item is set to Yes , all constraint names (including index names) must be unique in the database. Otherwise constraint names must be unique only at the object level. |
| UserStereotype | Specifies the user stereotype to be used to indicate a user (permissions grantee). |

Script/Sql Category (DBMS)

The SQL category is located in the **Root > Script** category and contains sub-categories that 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: <pre>/* */</pre> |
| BlockTerminator | Specifies the end of block character, which is used to end expressions for triggers and stored procedures. |
| Delimiter | Specifies the field separation character. |
| IdentifierDelimiter | 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: <pre>%%</pre> |
| 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. |
| 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, <code>BlockTerminator</code> is used instead. |

| Item | Description |
|--------------|--|
| UseBlockTerm | <p>Specifies the use of <code>BlockTerminator</code>. The following settings are available:</p> <ul style="list-style-type: none"> • Yes - <code>BlockTerminator</code> is always used • No - <code>BlockTerminator</code> 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 | <p>Specifies that object codes are systematically enquoted during the generation. The following settings are available:</p> <ul style="list-style-type: none"> • Yes – Quotes are systematically added to object codes during generation • No - Object codes are generated without quotes |
| CaseSensitivityUsingQuote | <p>Specifies if the case sensitivity for identifiers is managed using double quotes. Enable this option if the DBMS you are using needs double quotes to preserve the case of object codes.</p> |
| Date and Time formats | <p>See <i>Date and Time Format</i> on page 137.</p> |
| EnableOwnerPrefix / EnableDtbsPrefix | <p>Specifies that object codes can be prefixed by the object owner (<code>%OWNER%</code>), the database name (<code>%DBPREFIX%</code>), or both (<code>%QUALIFIER%</code>). The following settings are available:</p> <ul style="list-style-type: none"> • Yes – enables the Owner Prefix and/or Database Prefix options in the Database Generation dialog to require one or both prefixes for objects. • No - The Owner Prefix and Database Prefix options are unavailable <p>Note: <code>EnableOwnerPrefix</code> enables the Ignore identifying owner model option for tables and views.</p> |

| Item | Description |
|-------------------------------|--|
| IllegalChar | <p>[generation only] Specifies invalid characters for names. If there is an illegal character in a Code, the code is set between quotes during generation.</p> <p>Example:</p> <pre>+ - * / ! = < > ' " ()</pre> <p>If the name of the table is "SALES+PROFITS", the generated create statement will be:</p> <pre>CREATE TABLE "SALES+PROFITS"</pre> <p>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.</p> |
| LowerCaseOnly / UpperCaseOnly | <p>When generating a script, all objects are generated in lowercase or uppercase independently of the model Naming Conventions and the PDM codes. The following settings are available:</p> <ul style="list-style-type: none"> • Yes - Forces all generated script characters to lowercase or uppercase. • No - Generates all scripts unchanged from the way objects are written in the model. <p>Note: These items are mutually exclusive. If both are enabled, the script is generated in <i>lowercase</i>.</p> |
| MaxScriptLen | Specifies the maximum length of a script line. |

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

CHAPTER 4: DBMS Definition Files

data type in order to know which item to customize in your DBMS. For example, if the columns use the datetime data type in your model, you should customize the `DateTimeFormat` item in your DBMS.

The default date and time format is the following:

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

Where:

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

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

File Category

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

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

| Item | Description |
|-----------------|--|
| EnableMultiFile | <p>Specifies that multiple scripts are allowed. The following settings are available:</p> <ul style="list-style-type: none"> • 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. <p>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.</p> <p>The default name for the global script is CREBAS for database generation, CRETRG for triggers and stored procedures generation, and ALTER for database modification.</p> |
| Footer | Specifies the text for the database generation script footer. |
| Header | Specifies the text for the database generation script header. |
| ScriptExt | <p>Specifies the default script extension when you generate a database or modify a database for the first time.</p> <p>Example:</p> <pre>sql</pre> |
| StartCommand | <p>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.</p> <p>Example (Sybase ASE 11):</p> <pre>isql %NAMESCRIPT%</pre> <p>Corresponds to the %STARTCMD% variable (see <i>PDM Variables and Macros</i> on page 212).</p> |
| TableExt | <p>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.</p> <p>Example:</p> <pre>sql</pre> |
| TrgFooter | Specifies footer text for a triggers and procedures generation script. |
| TrgHeader | Header script for triggers and procedures generation. |
| TrgUsage1 | [when using a single script] Specifies text to display in the Output window at the end of trigger and procedure generation. |

| Item | Description |
|------------|--|
| 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: <code>trg</code> |
| 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: <code>char()</code> <code>charindex()</code> <code>char_length()</code> etc |
| Commit | Specifies a statement for validating the transaction by live connection. |
| ConvertAnyTo-String | Specifies a function to convert any type to a string. |
| ConvertDateTo-Month, ConvertDateToQuarter, ConvertDateToYear | Specifies a function to extract the relevant period from a date. |
| ConvertFunc | Specifies a list of SQL functions to use when converting values between hex and integer and handling strings. Example: <code>convert()</code> <code>hextoint()</code> <code>inttohex()</code> etc |

| Item | Description |
|------------------|---|
| DateFunc | <p>Specifies a list of SQL functions to use with dates.</p> <p>Example:</p> <pre>dateadd() datediff() datename() etc</pre> |
| GroupFunc | <p>Specifies a list of SQL functions to use with group keywords.</p> <p>Example:</p> <pre>avg() count() max() etc</pre> |
| ListOperators | <p>Specifies a list of SQL operators to use when comparing values, boolean, and various semantic operators.</p> <p>Example:</p> <pre>= != not like etc</pre> |
| NumberFunc | <p>Specifies a list of SQL functions to use with numbers.</p> <p>Example:</p> <pre>abs() acos() asin() etc</pre> |
| OtherFunc | <p>Specifies a list of SQL functions to use when estimating, concatenating and SQL checks.</p> <p>Example:</p> <pre>db_id() db_name() host_id() etc</pre> |
| Reserved Default | <p>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.</p> <p>Example (SQL Anywhere® 10) - USER is a reserved default value:</p> <pre>Create table CUSTOMER (Username varchar(30) default USER)</pre> <p>When you run this script, CURRENT DATE is recognized as a reserved default value.</p> |

| 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). |
| StringConcatenationOperator | Specifies the operator used to concatenate two strings. |

Script/Objects Category (DBMS)

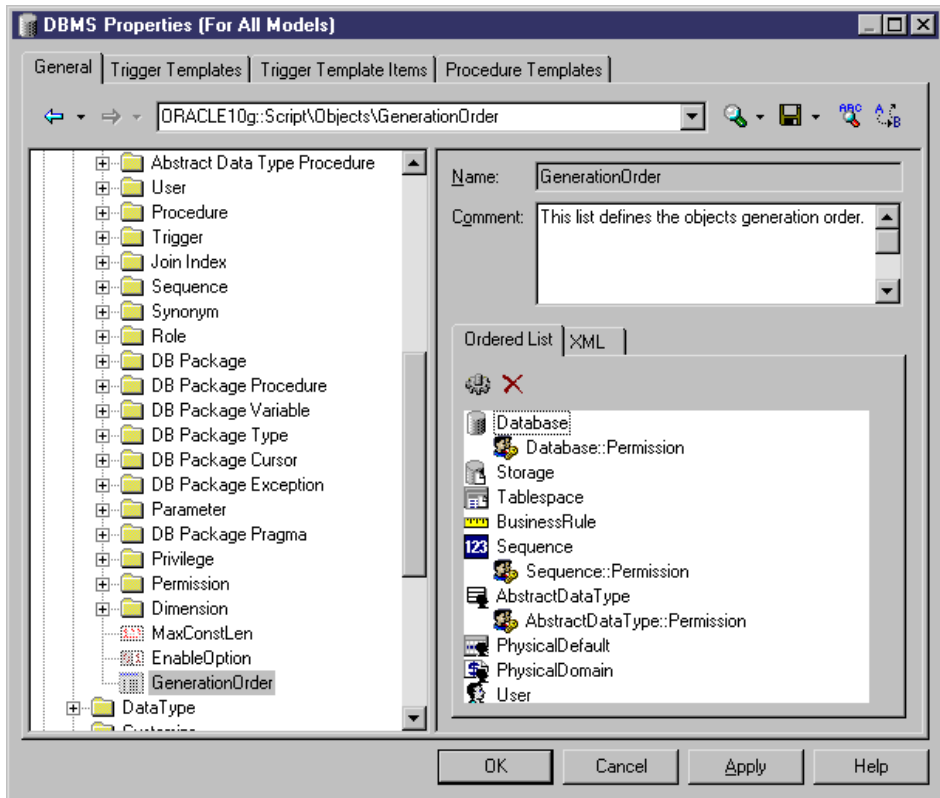
The Objects category is located in the **Root > Script > SQL** category (and, possibly within **Root > ODBC > SQL**), and contains the following items that define the database objects that will be available in your model.

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

- **MaxConstLen** - Specifies the maximum constraint name length supported by the target database for tables, columns, primary and foreign keys. This value is used during model checking and returns 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** - Specifies that physical options are supported by the target DBMS for the model, tables, indexes, alternate keys, and other objects and enables the display of the **Options** tab in object property sheets. For more information, see *Physical Options (DBMS)* on page 207.
- **GenerationOrder** - Specifies the generation order of database objects. Drag and drop entries in the **Ordered List** tab to adjust the order in which objects will be created.



Note: If an object does not appear on the list, it will still be generated, but after the listed objects. You can add and remove items using the tools on the tab. 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. Extended objects (see *Generating and Reverse-Engineering PDM Extended Objects* on page 133) cannot be added to this list, and are generated after all other objects.

Common Object Items

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

| Item | Description |
|--|--|
| Add | <p>Specifies the statement required to add the object inside the creation statement of another object.</p> <p>Example (adding a column):</p> <pre>%20: COLUMN% %30: DATATYPE% [default %DEFAULT%] [%IDENTITY%?identity: [%NULL%] [%NOTNULL%]] [[constraint %CONSTNAME%] check (%CONSTRAINT%)]</pre> |
| AfterCreate/ After-Drop/ AfterModify | Specify extended statements executed after the main Create, Drop or Modify statements. For more information, see <i>Script generation</i> on page 121. |
| Alter | Specifies the statement required to alter the object. |
| AlterDBIgnored | Specifies a list of attributes that should be ignored when performing a comparison before launching an update database. |
| AlterStatementList | Specifies a list of attributes which, when changed, should give rise to an alter statement. Each attribute in the list is mapped to the alter statement that should be used. |
| BeforeCreate/ BeforeDrop/ BeforeModify | Specify extended statements executed before the main Create, Drop or Modify statements. For more information, see <i>Script generation</i> on page 121. |
| ConstName | <p>Specifies a constraint name template for the object. The template controls how the name of the object will be generated.</p> <p>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.</p> <p>Examples (ASE 15):</p> <ul style="list-style-type: none"> • Table: CKT_%.U26:TABLE% • Column: CKC_%.U17:COLUMN%_%.U8:TABLE% • Primary Key: PK_%.U27:TABLE% |
| Create | <p>[generation and reverse] Specifies the statement required to create the object.</p> <p>Example:</p> <pre>create table %TABLE%</pre> |

| Item | Description |
|-----------------|--|
| DefOptions | <p>Specifies default values for physical options (see <i>Physical Options (DBMS)</i> on page 207) that will be applied to all objects. These values must respect SQL syntax.</p> <p>Example:</p> <pre>in default_tablespace</pre> |
| Drop | <p>Specifies the statement required to drop the object.</p> <p>Example (SQL Anywhere 10):</p> <pre>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</pre> |
| Enable | <p>Specifies whether an object is supported.</p> |
| EnableOwner | <p>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:</p> <ul style="list-style-type: none"> • Yes - The Owner list is enabled in the object's property sheet. • No – Owners are not supported for the object. <p>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:</p> <pre>create [%UNIQUE?%UNIQUE% :[%INDEXTYPE%]]index [%QUALIFIER%]%INDEX% on [%CLUSTER?cluster C_%TABLE% : [%TABLQUALIFIER%]%TABLE% (%CIDXLIST%)] [%OPTIONS%]</pre> <p>Where %QUALIFIER% refers to the current object (index) and %TABLQUALIFIER% refers to the parent table of the index.</p> |
| EnableSynonym | <p>Enables support for synonyms on the object.</p> |
| Footer / Header | <p>Specify the object footer and header. The contents are inserted directly after or before each create object statement.</p> |
| MaxConstLen | <p>Specifies the maximum constraint name length supported for the object in the target database, where this value differs from the default specified in MaxConstLen (see <i>.Script/Objects Category (DBMS)</i> on page 142).</p> |

| 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 <i>Script generation</i> on page 121. Example (ASE 12.5): <code>ExtTablePartition</code> |
| Options | Specifies physical options (see <i>Physical Options (DBMS)</i> on page 207) available to apply when creating an object. Example (ASA 6): <code>in %s : category=tablespace</code> |
| 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): <code>SELECT / Sel</code> <code>INSER / Ins</code> <code>DELETE / Del</code> <code>UPDATE / Upd</code> <code>REFERENCES / Ref</code> |
| Reversed Queries | Specifies a list of additional attribute queries to be called during live database reverse engineering. For more information, see <i>Live database reverse engineering</i> on page 125. |
| Reversed Statements | Specifies a list of additional statements that will be reverse engineered. For more information, see <i>Script reverse engineering</i> on page 124. |
| SqlAttrQuery | Specifies a SQL query to retrieve additional information on objects reversed by <code>SQLListQuery</code> . Example (Join Index in Oracle 10g): <code>{OWNER ID, JIDX ID, JIDXWHERE ...}</code> <code>select index_owner, index_name,</code> <code>outer_table_owner '.' outer_table_name '.'</code> <code> outer_table_column '=' inner_table_owner</code> <code> '.' inner_table_name '.' inner_ta-</code> <code>ble_column ','</code> <code>from all_join_ind_columns</code> <code>where 1=1</code> <code>[and index_owner=%.q:OWNER%]</code> <code>[and index_name=%.q:JIDX%]</code> |

| Item | Description |
|--------------|--|
| SqlListQuery | <p>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.</p> <p>Example (Dimension in Oracle 10g):</p> <pre>{ 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</pre> |
| SqlOptsQuery | <p>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.</p> <p>Example (Table in SQL Anywhere 10):</p> <pre>{OWNER, TABLE, OPTIONS} select u.user_name, t.table_name, 'in '+ f.dbpace_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.dbpace_name <> 'SYSTEM' and t.table_type in (1, 3, 4) [and t.table_name = %.q:TABLE%] [and u.user_name = %.q:OWNER%]</pre> |
| SqlPermQuery | <p>Specifies a SQL query to reverse engineer permissions granted on the object.</p> <p>Example (Procedure in SQL Anywhere 10):</p> <pre>{ 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)</pre> |

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] | <p>The following common object items may be defined for tables:</p> <ul style="list-style-type: none"> • 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 <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| AddTableCheck | <p>Specifies a statement for customizing the script to modify the table constraints within an <code>alter table</code> statement.</p> <p>Example (SQL Anywhere 10):</p> <pre>alter table [%QUALIFIER%]%TABLE% add [constraint %CONSTNAME%]check (%.A:CON- STRAINT%)</pre> |
| AllowedADT | <p>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.</p> <p>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.</p> <p>Example (Oracle 10g):</p> <pre>OBJECT</pre> |

| Item | Description |
|-------------------|--|
| AlterTable Footer | <p>Specifies a statement to be placed after <code>alter table</code> statements (and before the terminator).</p> <p>Example:</p> <pre>AlterTableFooter = /* End of alter statement */</pre> |
| AlterTable Header | <p>Specifies a statement to be placed before <code>alter table</code> statements. You can place an alter table header in your scripts to document or perform initialization logic.</p> <p>Example:</p> <pre>AlterTableHeader = /* Table name: %TABLE% */</pre> |
| DefineTable Check | <p>Specifies a statement for customizing the script of table constraints (checks) within a <code>create table</code> statement.</p> <p>Example:</p> <pre>check (%CONSTRAINT%)</pre> |
| DropTable Check | <p>Specifies a statement for dropping a table check in an <code>alter table</code> statement.</p> <p>Example:</p> <pre>alter table [%QUALIFIER%]%TABLE% delete check</pre> |
| InsertIdentityOff | <p>Specifies a statement for enabling insertion of data into a table containing an identity column.</p> <p>Example (ASE 15):</p> <pre>set identity_insert [%QUALIFIER%]%@OBJTCODE% off</pre> |
| InsertIdentityOn | <p>Specifies a statement for disabling insertion of data into a table containing an identity column.</p> <p>Example (ASE 15):</p> <pre>set identity_insert [%QUALIFIER%]%@OBJTCODE% on</pre> |

| Item | Description |
|--------------|---|
| Rename | <p>[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.</p> <p>Example (Oracle 10g):</p> <pre>rename %OLDTABL% to %NEWTABL%</pre> <p>The %OLDTABL% variable is the code of the table before renaming, and the %NEWTABL% variable is the new code.</p> |
| SqlChckQuery | <p>Specifies a SQL query to reverse engineer table checks.</p> <p>Example (SQL Anywhere 10):</p> <pre>{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</pre> |

| Item | Description |
|--------------------|---|
| SqlListRefr Tables | <p>Specifies a SQL query used to list the tables referenced by a table.</p> <p>Example (Oracle 10g):</p> <pre>{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_constraint_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_constraint_name = c.constraint_name and r.r_owner = c.owner) [and c.owner = %q:SCHEMA%] [and c.table_name = %q:TABLE%]</pre> |
| SqlListSchema | <p>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).</p> <p>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.</p> <p>Example (Oracle 10g):</p> <pre>SELECT schema_url FROM dba_xml_schemas</pre> |
| SqlStatistics | <p>Specifies a SQL query to reverse engineer column and table statistics. See SqlStatistics in <i>Column</i> on page 152.</p> |
| SqlXMLTable | <p>Specifies a sub-query used to improve the performance of SqlAttrQuery (see <i>Common object items</i> on page 144).</p> |
| TableComment | <p>[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.</p> <p>Example (Oracle 10g):</p> <pre>comment on table [%QUALIFIER%]TABLE% is %q:COMMENT%</pre> <p>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.</p> |

| 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • Add • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • ConstName • Create, Drop • Enable • Maxlen, MaxConstLen • ModifiableAttributes • Options, DefOptions • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| AddColnCheck | Specifies a statement for customizing the script for modifying column constraints within an alter table statement. Example (Oracle 10g): <pre>alter table [%QUALIFIER%]%TABLE% add [constraint %CONSTNAME%] check (%.A:CONSTRAINT %)</pre> |

| Item | Description |
|-----------------------|--|
| AlterTableAdd Default | <p>Specifies a statement for defining the default value of a column in an alter statement.</p> <p>Example (SQL Server 2005):</p> <pre>[[constraint %ExtDeftConstName%] default %DEFAULT%]for %COLUMN%</pre> |
| AltEnableAdd ColnChk | <p>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:</p> <ul style="list-style-type: none"> • Yes - AddColnChk 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. <p>See also AddColnChk.</p> |
| AltEnableTS Copy | Enables timestamp columns in insert statements. |
| Bind | <p>Specifies a statement for binding a rule to a column.</p> <p>Example (ASE 15):</p> <pre>[%R%?[exec]][execute]sp_bindrule [%R%?['[%QUALIFIER%]%RULE%']][[%QUALIFIER%]%RULE%]:['[%QUALIFIER%]%RULE%']], '%TABLE%.%COLUMN%'</pre> |
| CheckNull | Specifies whether a column can be null. |
| Column Comment | <p>Specifies a statement for adding a comment to a column.</p> <p>Example:</p> <pre>comment on column [%QUALIFIER%]%TABLE%.%COLUMN% is %.q:COMMENT%</pre> |
| DefineColn Check | <p>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%.</p> <p>Example:</p> <pre>[constraint %CONSTNAME%] check (%CONSTRAINT%)</pre> |

| Item | Description |
|------------------------|---|
| DropColnChck | <p>Specifies a statement for dropping a column check in an <code>alter table</code> statement. This statement is used in the database modification script when the check parameters have been removed on a column.</p> <p>If <code>DropColnChck</code> is empty, PowerDesigner copies data to a temporary table before recreating the table with the new constraints.</p> <p>Example (SQL Anywhere 10):</p> <pre>alter table [%QUALIFIER%]%TABLE% drop constraint %CONSTNAME%</pre> |
| DropColnComp | <p>Specifies a statement for dropping a column computed expression in an <code>alter table</code> statement.</p> <p>Example (SQL Anywhere 10):</p> <pre>alter table [%QUALIFIER%]%TABLE% alter %COLUMN% drop compute</pre> |
| DropDefault Constraint | <p>Specifies a statement for dropping a constraint linked to a column defined with a default value</p> <p>Example (SQL Server 2005):</p> <pre>[%ExtDeftConstName%?alter table [%QUALIFIER%]%TABLE% drop constraint %ExtDeftConstName%]</pre> |
| EnableBindRule | <p>Specifies whether business rules may be bound to columns for check parameters. The following settings are available:</p> <ul style="list-style-type: none"> • Yes - The Create and Bind entry of Rule are generated • No - The check is generated inside the column Add order |
| Enable Computed-Coln | <p>Specifies whether computed columns are permitted.</p> |

| Item | Description |
|---------------|--|
| EnableDefault | <p>Specifies whether predefined default values are permitted. The following settings are available:</p> <ul style="list-style-type: none"> • 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. <p>Example (AS IQ 12.6):</p> <p>EnableDefault is enabled and the default value for the column employee function EMPFUNC is Technical Engineer. The generated script is:</p> <pre>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));</pre> |

| Item | Description |
|-----------------------|--|
| EnableIdentity | <p>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:</p> <ul style="list-style-type: none"> • Yes - Enables the Identity check box in the column property sheet. • No - The Identity check box is not available. <p>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:</p> <ul style="list-style-type: none"> • 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 <i>numeric</i>. If the data type of an identity column is changed to an unsupported type, an error is displayed. <p>Note that, during generation, the %IDENTITY% variable contains the value "identity" but you can easily change it, if needed, using the following syntax :</p> <pre>[%IDENTITY%?new identity keyword]</pre> |
| EnableNotNullWithDflt | <p>Specifies whether default values are assigned to columns containing Null values. The following settings are available:</p> <ul style="list-style-type: none"> • 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. |
| ModifyColnChck | <p>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.</p> <p>If AddColnChck is empty, PowerDesigner copies data to a temporary table before recreating the table with the new constraints.</p> <p>Example (AS IQ 12.6):</p> <pre>alter table [%QUALIFIER%]%TABLE% modify %COLUMN% check (%A:CONSTRAINT%)</pre> <p>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.</p> <p>AltEnableAddColnChk must be set to YES to allow use of this statement.</p> |

| Item | Description |
|-----------------|--|
| ModifyColn Comp | <p>Specifies a statement for modifying a computed expression for a column in an alter table.</p> <p>Example (ASA 6):</p> <pre>alter table [%QUALIFIER%]%TABLE% alter %COLUMN% set compute (%COMPUTE%)</pre> |
| ModifyColnDflt | <p>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.</p> <p>If ModifyColnDflt is empty, PowerDesigner copies data to a temporary table before recreating the table with the new constraints.</p> <p>Example (ASE 15):</p> <pre>alter table [%QUALIFIER%]%TABLE% replace %COLUMN% default %DEFAULT%</pre> <p>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.</p> |
| ModifyColnNull | <p>Specifies a statement for modifying the null/not null status of a column in an alter table statement.</p> <p>Example (Oracle 10g):</p> <pre>alter table [%QUALIFIER%]%TABLE% modify %COLUMN% %MAND%</pre> |
| ModifyColumn | <p>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.</p> <p>Example (SQL Anywhere 10):</p> <pre>alter table [%QUALIFIER%]%TABLE% modify %COLUMN% %DATATYPE% %NOTNULL%</pre> |
| NullRequired | <p>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 <i>Working with Null values</i> on page 159.</p> |
| Rename | <p>Specifies a statement for renaming a column within an alter table statement.</p> <p>Example (Oracle 10g):</p> <pre>alter table [%QUALIFIER%]%TABLE% rename column %OLDCOLN% to %NEWCOLN%</pre> |

| Item | Description |
|---------------|---|
| SqlChkQuery | <p>Specifies a SQL query to reverse engineer column check parameters. The result must conform to proper SQL syntax.</p> <p>Example (SQL Anywhere 10):</p> <pre>{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.constraint_id) join sys.systab t on (t.object_id = k.table_object_id) join sys.sysuserperms u on (u.user_id = t.creator) join sys.syscolumn c on (c.object_id = k.ref_object_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</pre> |
| SqlStatistics | <p>Specifies a SQL query to reverse engineer column and table statistics.</p> <p>Example (ASE 15):</p> <pre>[%ISLONGDTP%?{ AverageLength } select [%ISLONGDTP%?[%ISSTRDTP%? 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 distinct_values, [%ISVARDTP%?[%ISSTRDTP%?avg(char_length(%COLUMN%)):avg(datalength(%COLUMN%))]:null] as average_length from [%QUALIFIER%]%TABLE%</pre> |
| Unbind | <p>Specifies a statement for unbinding a rule to a column.</p> <p>Example (ASE 15):</p> <pre>[%R%?[exec]][execute]sp_unbindrule '%TABLE%.%COLUMN%'</pre> |

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_NULL 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] | <p>The following common object items may be defined for indexes:</p> <ul style="list-style-type: none"> • Add • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner • Header, Footer • Maxlen • ModifiableAttributes • Options, DefOptions • ReversedQueries • ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| AddColIndex | <p>Specifies a statement for adding a column in the <code>Create Index</code> statement. This parameter defines each column in the column list of the <code>Create Index</code> statement.</p> <p>Example (ASE 15):</p> <pre style="background-color: #f0f0f0; padding: 5px;">%COLUMN% [%ASC%]</pre> <p><code>%COLUMN%</code> is the code of the column defined in the column list of the table. <code>%ASC%</code> is <code>ASC</code> (ascending order) or <code>DESC</code> (descending order) depending on the <code>Sort</code> radio button state for the index column.</p> |
| AlterIgnoreOrder | <p>Specifies that changes in the order of the collection should not provoke a modify database order.</p> |
| Cluster | <p>Specifies the value to be assigned to the <code>Cluster</code> keyword. If this parameter is empty, the default value of the <code>%CLUSTER%</code> variable is <code>CLUSTER</code>.</p> |
| CreateBefore Key | <p>Controls the generation order of keys and indexes. The following settings are available:</p> <ul style="list-style-type: none"> • Yes – Indexes are generated before keys. • No – Indexes are generated after keys. |

| Item | Description |
|--------------------|---|
| DefIndexType | <p>Specifies the default type of an index.</p> <p>Example (DB2):</p> <pre>Type2</pre> |
| DefineIndex Column | <p>Specifies the column of an index.</p> |
| EnableAscDesc | <p>Enables the Sort property in Index property sheets, which allows sorting in ascending or descending order. The following settings are available:</p> <ul style="list-style-type: none"> • 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. <p>Example (SQL Anywhere 10):</p> <p>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:</p> <pre>create index IX_TASK on TASK (PRONUM asc, TSKNAME desc);</pre> |
| EnableCluster | <p>Enables the creation of cluster indexes. The following settings are available:</p> <ul style="list-style-type: none"> • Yes - The Cluster check box is enabled in index property sheets. • No – Cluster indexes are not supported. |
| EnableFunction | <p>Enables the creation of function-based indexes. The following settings are available:</p> <ul style="list-style-type: none"> • Yes - You can define expressions for indexes. • No – Function-based indexes are not supported. |
| IndexComment | <p>Specifies a Statement for adding a comment to an index.</p> <p>Example (SQL Anywhere 10):</p> <pre>comment on index [%QUALIFIER%]%TABLE%.%INDEX% is % .q:COMMENT%</pre> |

| Item | Description |
|-------------------|---|
| IndexType | <p>Specifies a list of available index types.</p> <p>Example (IQ 12.6):</p> <pre>CMP HG HNG LF WD DATE TIME DTTM</pre> |
| MandIndexType | <p>Specifies whether the index type is mandatory for indexes. The following settings are available:</p> <ul style="list-style-type: none"> • Yes – The index type is mandatory. • No - The index type is not mandatory. |
| MaxColIndex | <p>Specifies the maximum number of columns that may be included in an index. This value is used during model checking.</p> |
| SqlSysIndex Query | <p>Specifies a SQL query used to list system indexes created by the database. These indexes are excluded during reverse engineering.</p> <p>Example (AS IQ 12.6):</p> <pre>{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%]</pre> |
| UniqName | <p>Specifies whether index names must be unique within the global scope of the database. The following settings are available:</p> <ul style="list-style-type: none"> • 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] | <p>The following common object items may be defined for primary keys:</p> <ul style="list-style-type: none"> • Add • ConstName • Create, Drop • Enable • Options, DefOptions • ReversedQueries <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| EnableCluster | <p>Specifies whether clustered constraints are permitted on primary keys.</p> <ul style="list-style-type: none"> • Yes - Clustered constraints are permitted. • No - Clustered constraints are not permitted. |
| PkAutoIndex | <p>Determines whether a <code>Create Index</code> statement is generated for every Primary key statement. The following settings are available:</p> <ul style="list-style-type: none"> • 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 | <p>Specifies a statement for adding a primary key comment.</p> |

| Item | Description |
|--------------|---|
| UseSpPrimKey | <p>Specifies the use of the <code>Sp_primarykey</code> 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 <code>%USE_SP_PKEY%</code> and choose between the creation key in the table or launching a procedure. The following settings are available:</p> <ul style="list-style-type: none"> • Yes - The <code>Sp_primarykey</code> statement is used to generate primary keys. • No - Primary keys are generated separately in an <code>alter table</code> statement. <p>Example (ASE 15):</p> <p>If UseSpPrimKey is enabled the Add entry for Pkey contains:</p> <pre>UseSpPrimKey = YES Add entry of [%USE_SP_PKEY%?[execute] sp_primarykey %TABLE%, %PKEYCOLUMNS% :alter table [%QUALIFIER%]%TABLE% add [constraint %CONSTNAME%] primary key [%IsClustered%] (%PKEYCOLUMNS%) [%OPTIONS%]</pre> |

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] | <p>The following common object items may be defined for keys:</p> <ul style="list-style-type: none"> • Add • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • ConstName • Create, Drop • Enable • MaxConstLen • ModifiableAttributes • Options, DefOptions • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |

| Item | Description |
|---------------------|--|
| AKeyComment | Specifies a statement for adding an alternate key comment. |
| AllowNullable Coln | <p>Specifies whether non-mandatory columns are permitted. The following settings are available:</p> <ul style="list-style-type: none"> • Yes - Non mandatory columns are permitted. • No - Non mandatory column are not permitted. |
| AlterIgnoreOrder | Specifies that changes in the order of the collection should not provoke a modify database order. |
| EnableCluster | <p>Specifies whether clustered constraints are permitted on alternate keys.</p> <ul style="list-style-type: none"> • Yes - Clustered constraints are permitted. • No - Clustered constraints are not permitted. |
| SqlAkeyIndex | <p>Specifies a reverse-engineering query for obtaining the alternate key indexes of a table by live connection.</p> <p>Example (SQL Anywhere 10):</p> <pre>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%]</pre> |
| UniqConstAuto Index | <p>Determines whether a Create Index statement is generated for every key statement. The following settings are available:</p> <ul style="list-style-type: none"> • 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] | <p>The following common object items may be defined for references:</p> <ul style="list-style-type: none"> • Add • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • ConstName • Create, Drop • Enable • MaxConstLen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| CheckOn Commit | <p>Specifies that referential integrity testing is performed only after the COMMIT. Contains the keyword used to specify a reference with the CheckOn-Commit option.</p> <p>Example:</p> <pre>CHECK ON COMMIT</pre> |
| DclDelIntegrity | <p>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:</p> <ul style="list-style-type: none"> • RESTRICT • CASCADE • SET NULL • SET DEFAULT |

| Item | Description |
|-------------------------|---|
| DclUpdIntegrity | <p>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:</p> <ul style="list-style-type: none"> • RESTRICT • CASCADE • SET NULL • SET DEFAULT |
| DefineJoin | <p>Specifies a statement to define a join for a reference. This is another way of defining the contents of the <code>create reference</code> statement, and corresponds to the <code>%JOINS%</code> variable.</p> <p>Usually the <code>create</code> script for a reference uses the <code>%CKEYCOLUMNS%</code> and <code>%PKEYCOLUMNS%</code> variables, which contain the lists of child and parent columns separated by commas.</p> <p>If you use <code>%JOINS%</code>, you can refer to each paired parent and child columns separately. A loop is executed on <code>Join</code> for each paired parent and child columns, allowing to have a syntax mix of PK and FK.</p> <p>Example (Access 2000):</p> <pre>P=%PK% F=%FK%</pre> |
| EnableChange Join-Order | <p>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:</p> <ul style="list-style-type: none"> • 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. |
| EnableCluster | <p>Specifies whether clustered constraints are permitted on foreign keys.</p> <ul style="list-style-type: none"> • Yes - Clustered constraints are permitted. • No - Clustered constraints are not permitted. |
| EnablefKey Name | <p>Specifies the foreign key role allowed during database generation. The following settings are available:</p> <ul style="list-style-type: none"> • Yes - The code of the reference is used as role for the foreign key. • No - The foreign key role is not allowed. |

| Item | Description |
|-----------------------|--|
| FKAutoIndex | <p>Determines whether a <code>Create Index</code> statement is generated for every foreign key statement. The following settings are available:</p> <ul style="list-style-type: none"> • 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. |
| FKKeyComment | Specifies a statement for adding an alternate key comment. |
| SqlListChildren Query | <p>Specifies a SQL query used to list the joins in a reference.</p> <p>Example (Oracle 10g):</p> <pre>{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]</pre> |
| UseSpFornKey | <p>Specifies the use of the <code>Sp_foreignkey</code> statement to generate a foreign key. The following settings are available:</p> <ul style="list-style-type: none"> • Yes - The <code>Sp_foreignkey</code> statement is used to create references. • No - Foreign keys are generated separately in an <code>alter table</code> statement using the <code>Create order of reference</code>. <p>See also <code>UseSpPrimKey</code> (<i>Pkey</i> on page 163).</p> |

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] | <p>The following common object items may be defined for views:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableSynonym • Header, Footer • ModifiableAttributes • Options • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| EnableIndex | <p>Specifies a list of view types for which a view index is available.</p> <p>Example (Oracle 10g):</p> <pre>MATERIALIZED</pre> |
| SqlListSchema | <p>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).</p> <p>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.</p> <p>Example (Oracle 10g):</p> <pre>SELECT schema_url FROM dba_xml_schemas</pre> |
| SqlXMLView | <p>Specifies a sub-query used to improve the performance of SqlAttrQuery.</p> |
| TypeList | <p>Specifies a list of types (for example, DBMS: relational, object, XML) for views. This list populates the Type list of the view property sheet.</p> <p>The XML type is to be used with the SqlListSchema item.</p> |

| Item | Description |
|-------------|---|
| ViewCheck | <p>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.</p> <p>Example (SQL Anywhere 10):</p> <p>If ViewCheck is set to with check option, the generated script is:</p> <pre>create view TEST as select CUSTOMER.CUSNUM, CUSTOMER.CUSNAME, CUSTOMER.CUSTEL from CUSTOMER with check option;</pre> |
| ViewComment | <p>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.</p> <p>Example (Oracle 10g):</p> <pre>[%VIEWSTYLE%=view? comment on table [%QUALIFIER%] %VIEW% is %.q:COMMENT%]</pre> |
| ViewStyle | <p>Specifies a view usage. The value defined is displayed in the Usage list of the view property sheet.</p> <p>Example (Oracle 10g):</p> <pre>materialized view</pre> |

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] | <p>The following common object items may be defined for tablespaces:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • ModifiableAttributes • Options, DefOptions • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| 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] | <p>The following common object items may be defined for storages:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • ModifiableAttributes • Options, DefOptions • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| 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] | <p>The following common object items may be defined for databases:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • ModifiableAttributes • Options, DefOptions • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| BeforeCreate Database | <p>Controls the order in which databases, tablespaces, and storages are generated. The following settings are available:</p> <ul style="list-style-type: none"> • 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 | <p>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.</p> |
| EnableMany Databases | <p>Enables support for multiple databases in the same model.</p> |
| OpenDatabase | <p>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.</p> <p>Example (ASE 15):</p> <pre>use %DATABASE%</pre> <p>The %DATABASE% variable is the code of the database associated with the generated model.</p> |

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] | <p>The following common object items may be defined for domains:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| Bind | <p>Specifies the syntax for binding a business rule to a domain.</p> <p>Example (ASE 15):</p> <pre data-bbox="454 852 1174 927">[%R%?[exec]][execute]sp_bindrule [%R%?['[%QUALIFIER%]%RULE%']][[%QUALIFIER%]%RULE%]:['[%QUALIFIER%]%RULE%']], %DOMAIN%</pre> |
| EnableBindRule | <p>Specifies whether business rules may be bound to domains for check parameters. The following settings are available:</p> <ul style="list-style-type: none"> • Yes - The Create and Bind entry of Rule are generated • No - The check inside the domain Add order is generated |
| EnableCheck | <p>Specifies whether check parameters are generated.</p> <p>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.</p> <p>The following settings are available:</p> <ul style="list-style-type: none"> • Yes - Check parameters are generated • No - Variables linked to check parameters are not evaluated during generation and reverse |

| Item | Description |
|----------------------|---|
| EnableDefault | <p>Specifies whether default values are generated. The following settings are available:</p> <ul style="list-style-type: none"> • 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 | <p>Specifies the syntax for unbinding a business rule from a domain.</p> <p>Example (ASE 15):</p> <pre>[%R%?[exec]][execute]sp_unbindrule %DOMAIN%</pre> |

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] | <p>The following common object items may be defined for abstract data types:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • ModifiableAttributes • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlPermQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| ADTComment | Specifies a statement for adding an abstract data type comment. |
| AllowedADT | <p>Specifies a list of abstract data types which can be used as data types for abstract data types.</p> <p>Example (Oracle 10g):</p> <pre>OBJECT TABLE VARRAY</pre> |

| Item | Description |
|--------------------|---|
| Authorizations | Specifies a list of those users able to invoke abstract data types. |
| CreateBody | <p>Specifies a statement for creating an abstract data type body.</p> <p>Example (Oracle 10g):</p> <pre>create [or replace]type body [%QUALIFIER%]%ADT% [.O:[as][is]] %ADTBODY% end;</pre> |
| EnableAdtOn Coln | <p>Specifies whether abstract data types are enabled for columns. The following settings are available:</p> <ul style="list-style-type: none"> • 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 | <p>Specifies whether abstract data types are enabled for domains. The following settings are available:</p> <ul style="list-style-type: none"> • 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 | <p>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 <code>create</code> statement.</p> <p>Example (SQL Anywhere 10):</p> <pre>install JAVA UPDATE from file %.q:FILE%</pre> |
| JavaData | Specifies a list of available instantiation mechanisms for SQL Java abstract data types. |
| Remove | <p>Specifies a statement for installing a Java class as an abstract data class.</p> <p>Example (SQL Anywhere 10):</p> <pre>remove JAVA class %ADT%</pre> |

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] | <p>The following common object items may be defined for abstract data type attributes:</p> <ul style="list-style-type: none"> • Add • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop, Modify • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlListQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| AllowedADT | <p>Specifies a list of abstract data types which can be used as data types for abstract data type attributes.</p> <p>Example (Oracle 10g):</p> <pre>OBJECT TABLE VARRAY</pre> <p>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.</p> |

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] | <p>The following common object items may be defined for users:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • Maxlen • ModifiableAttributes • Options, DefOptions • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery, SqlPermQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| 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] | <p>The following common object items may be defined for rules:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |

| Item | Description |
|------------------|--|
| ColnDefault Name | <p>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.</p> <p>The corresponding variable is %DEFAULTNAME%.</p> <p>Example (ASE 15):</p> <pre>D_%.19:COLUMN%_%.8:TABLE%</pre> <p>The EMPFUNC column of the EMPLOYEE table has a default value of Technical Engineer. The D_EMPFUNC_EMPLOYEE column default name is created:</p> <pre>create default D_EMPFUNC_EMPLOYEE as 'Technical Engineer' go execute sp_bindefault D_EMPFUNC_EMPLOYEE, "EMPLOYEE.EMPFUNC" go</pre> |
| ColnRuleName | <p>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.</p> <p>The corresponding variable is %RULE%.</p> <p>Example (ASE 15):</p> <pre>R_%.19:COLUMN%_%.8:TABLE%</pre> <p>The TEASPE column of the Team table has a list of values - Industry, Military, Nuclear, Bank, Marketing - defined in its check parameters:</p> <p>The R_TEASPE_TEAM rule name is created and associated with the TEASPE column:</p> <pre>create rule R_TEASPE_TEAM as @TEASPE in ('Industry','Military','Nuclear','Bank','Marketing') go execute sp_bindrule R_TEASPE_TEAM, "TEAM.TEASPE" go</pre> |
| 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 | <p>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.</p> <p>The corresponding variable is %DEFAULTNAME%.</p> <p>Example (ASE 15):</p> <pre>D_%.28:DOMAIN%</pre> <p>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:</p> <pre>create default D_FunctionList as 'Technical Engineer' go</pre> |
| UddtRuleName | <p>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.</p> <p>The corresponding variable is %RULE%.</p> <p>Example (ASE 15):</p> <pre>R_%.28:DOMAIN%</pre> <p>The Domain_speciality domain has to belong to a set of values. This domain check has been defined in a validation rule. The SQL script will generate the rule name following the template defined in the item UddtRuleName:</p> <pre>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</pre> |

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] | <p>The following common object items may be defined for procedures:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner, EnableSynonym • Maxlen • ModifiableAttributes • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlPermQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| CreateFunc | <p>Specifies the statement for creating a function.</p> <p>Example (SQL Anywhere 10):</p> <pre>create function [%QUALIFIER%]%FUNC%[%PROCPRMS%? ([%PROCPRMS%])] %TRGDEFN%</pre> |
| CustomFunc | <p>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.</p> <p>Example (SQL Anywhere 10):</p> <pre>create function [%QUALIFIER%]%FUNC% (<arg> <type>) RETURNS <type> begin end</pre> |
| CustomProc | <p>Specifies the statement for creating a stored procedure.</p> <p>Example (SQL Anywhere 10):</p> <pre>create procedure [%QUALIFIER%]%PROC% (IN <arg> <type>) begin end</pre> |

| Item | Description |
|--------------------|--|
| DropFunc | Specifies the statement for dropping a function. Example (SQL Anywhere 10): <pre>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</pre> |
| 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: <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| DefaultTrigger Name | Specifies a template to define default trigger names. Example (SQL Anywhere 10): <pre>%.TEMPLATE%_%.L:TABLE%</pre> |
| EnableMulti Trigger | Enables the use of multiple triggers per type. |

| Item | Description |
|--------------------|--|
| Event | <p>Specifies a list of trigger event attributes to populate the Event list on the Definition tab of Trigger property sheets.</p> <p>Example:</p> <pre>Delete Insert Update</pre> |
| EventDelimiter | Specifies a character to separate multiple trigger events. |
| ImplementationType | Specifies a list of available trigger template types. |
| Time | <p>Specifies a list of trigger time attributes to populate the Time list on the Definition tab of Trigger property sheets.</p> <p>Example:</p> <pre>Before After</pre> |
| Trigger Comment | Specifies a statement for adding a trigger comment. |
| UniqName | <p>Specifies whether trigger names must be unique within the global scope of the database. The following settings are available:</p> <ul style="list-style-type: none"> • 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 | <p>Specifies a macro for accessing trigger error messages from a message table in your database.</p> <p>Enables the use of the User-defined radio button on the Error Messages tab of the Trigger Rebuild dialog box (see <i>Data Modeling > Building Data Models > Triggers and Procedures > Generating Triggers and Procedures > Creating User-Defined Error Messages</i>).</p> <p>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.</p> <p>Example (ASE 15):</p> <pre data-bbox="454 562 827 739">begin select @errno = %ERRNO%, @errmsg = %MSGTXT% from %MSGTAB% where %MSGNO% = %ERRNO% goto error end</pre> <p>Where:</p> <ul data-bbox="454 812 1188 968" style="list-style-type: none"> • %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 <p>See also UseErrorMsgText.</p> |
| UseErrorMsg Text | <p>Specifies a macro for accessing trigger error messages from the trigger template definition.</p> <p>Enables the use of the Standard radio button on the Error Messages tab of the Trigger Rebuild dialog box.</p> <p>The error number and message defined in the template definition are used.</p> <p>Example (ASE 15):</p> <pre data-bbox="454 1269 827 1390">begin select @errno = %ERRNO%, @errmsg = %MSGTXT% goto error end</pre> <p>See also UseErrorMsgTable.</p> |
| 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] | <p>The following common object items may be defined for DBMS triggers:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Alter, AlterStatementList, AlterDBIgnored • Enable, EnableOwner • Header, Footer • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| 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 <i>scope</i> , 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 | <p>Specifies a list of trigger time attributes to populate the Time list on the Definition tab of Trigger property sheets.</p> <p>Example:</p> <pre>Before After</pre> |
| 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] | <p>The following common object items may be defined for join indexes:</p> <ul style="list-style-type: none"> • Add • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner • Header, Footer • Maxlen • ModifiableAttributes • Options, DefOptions • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlOptsQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| AddJoin | <p>Specifies the SQL statement used to define joins for join indexes.</p> <p>Example:</p> <pre>Table1.coln1 = Table2.coln2</pre> |
| EnableJidxColn | <p>Enables support for attaching multiple columns to a join index. In Oracle 9i, this is called a bitmap join index.</p> |
| JoinIndex Comment | <p>Specifies a statement for adding a join index comment.</p> |

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] | <p>The following common object items may be defined for qualifiers:</p> <ul style="list-style-type: none"> • Enable • ReversedQueries • SqlListQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |

| 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] | <p>The following common object items may be defined for sequences:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner, EnableSynonym • Maxlen • ModifiableAttributes • Options, DefOptions • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlPermQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| Rename | <p>Specifies the command for renaming a sequence.</p> <p>Example (Oracle 10g):</p> <pre>rename %OLDNAME% to %NEWNAME%</pre> |
| 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] | <p>The following common object items may be defined for synonyms:</p> <ul style="list-style-type: none"> • Create, Drop • Enable, EnableSynonym • Maxlen • ReversedQueries • SqlAttrQuery, SqlListQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| 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] | <p>The following common object items may be defined for groups:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlPermQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| Bind | <p>Specifies a command for adding a user to a group.</p> <p>Example (SQL Anywhere 10):</p> <pre>grant membership in group %GROUP% to %USER%</pre> |
| Group Comment | Specifies a statement for adding a group comment. |
| ObjectOwner | Allows groups to be object owners. |

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

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] | <p>The following common object items may be defined for roles:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlPermQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| Bind | <p>Specifies a command for adding a role to a user or to another role.</p> <p>Example (ASE 15):</p> <pre>grant role %ROLE% to %USER%</pre> |

| Item | Description |
|-----------------------|--|
| SqlListChildren Query | <p>Specifies a SQL query for listing the members of a group.</p> <p>Example (ASE 15):</p> <pre>{ 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</pre> |
| 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] | <p>The following common object items may be defined for database packages:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableSynonym • Maxlen • ModifiableAttributes • Permission • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery, SqlPermQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| Authorizations | Specifies a list of those users able to invoke database packages. |

| Item | Description |
|------------|--|
| CreateBody | <p>Specifies a template for defining the body of the database package. This statement is used in the extension statement AfterCreate.</p> <p>Example (Oracle 10g):</p> <pre>create [or replace]package body [%QUALIFIER%] %DBPACKAGE% [.O:[as][is]][%IsPragma% ? pragma seri- ally_reusable] %DBPACKAGEBODY% [begin %DBPACKAGEINIT%]end[%DBPACKAGE%];</pre> |

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] | <p>The following common object items may be defined for database packages:</p> <ul style="list-style-type: none"> • Add • ReversedQueries <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| DBProcedure Body | <p>[database package procedures only] Specifies a template for defining the body of the package procedure in the Definition tab of its property sheet.</p> <p>Example (Oracle 10g):</p> <pre>begin end</pre> |

| Item | Description |
|----------------|--|
| ParameterTypes | <p>[database package procedures and cursors only] Specifies the available types for procedures or cursors.</p> <p>Example (Oracle 10g: procedure):</p> <pre>in in nocopy in out in out nocopy out out nocopy</pre> |

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] | <p>The following common object items may be defined for database packages:</p> <ul style="list-style-type: none"> • Add • ReversedQueries <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |

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] | <p>The following common object items may be defined for privileges:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • ModifiableAttributes • ReversedQueries, ReversedStatements <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |

| Item | Description |
|-----------------|---|
| GrantOption | Specifies the grant option for a privileges statement. Example (Oracle 10g): <code>with admin option</code> |
| 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): <code>CREATE DATABASE</code> <code>CREATE DEFAULT</code> <code>CREATE PROCEDURE</code> <code>CREATE TRIGGER</code> <code>CREATE RULE</code> <code>CREATE TABLE</code> <code>CREATE VIEW</code> |

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: <ul style="list-style-type: none"> • Create, Drop • Enable • ReversedQueries • SqlListQuery For a description of each of these common items, see <i>Common object items</i> on page 144. |
| GrantOption | Specifies the grant option for a permissions statement. Example (ASE 15): <code>with grant option</code> |
| RevokeInherited | Allows you to revoke inherited permissions from groups and roles. |
| RevokeOption | Specifies the revoke option for a permissions statement. Example (ASE 15): <code>cascade</code> |

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] | <p>The following common object items may be defined for defaults:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| Bind | <p>Specifies the command for binding a default object to a domain or a column.</p> <p>When a domain or a column use a default object, a <i>binddefault</i> statement is generated after the domain or table creation statement. In the following example, column Address in table Customer uses default object CITYDFLT:</p> <pre>create table CUSTOMER (ADDRESS char(10) null) sp_binddefault CITYDFLT, 'CUSTOMER.ADDRESS'</pre> <p>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:</p> <pre>ADDRESS char(10) default 'StdAddr' null</pre> |
| PublicOwner | Enables PUBLIC to own public synonyms. |
| Unbind | <p>Specifies the command for unbinding a default object from a domain or a column.</p> <p>Example (ASE 15):</p> <pre>[%R%?[exec]][execute]sp_unbinddefault %.q:BOUND_OBJECT%</pre> |

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] | <p>The following common object items may be defined for web services and web operations:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • Alter • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable, EnableOwner • Header, Footer • MaxConstLen (web operations only) • Maxlen • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| Enable Namespace | Specifies whether namespaces are supported. |
| EnableSecurity | Specifies whether security options are supported. |
| OperationType List | <p>[web operation only] Specifies a list of web service operation types.</p> <p>Example (DB2 UDB 8.x CS):</p> <pre>query update storeXML retrieveXML call</pre> |
| ServiceTypeList | <p>[web service only] Specifies a list of web service types.</p> <p>Example (SQL Anywhere 10):</p> <pre>RAW HTML XML DISH</pre> |
| 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] | <p>The following common object items may be defined for web parameters:</p> <ul style="list-style-type: none"> • Add • Enable <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| 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] | <p>The following common object items may be defined for dimensions:</p> <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • Alter • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • Enable • Header, Footer • Maxlen • ReversedQueries • SqlAttrQuery, SqlListQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| AddAttr Hierarchy | <p>Specifies the syntax for defining a list of hierarchy attributes.</p> <p>Example (Oracle 10g):</p> <pre>child of %DIMNATTRHIER%</pre> |
| AddAttribute | <p>Specifies the syntax for defining an attribute.</p> <p>Example (Oracle 10g):</p> <pre>attribute %DIMNATTR% determines [.O:[(%DIMNDEPCOLN- LIST%)] [%DIMNDEPCOLN%]]</pre> |
| AddHierarchy | <p>Specifies the syntax for defining a dimension hierarchy.</p> <p>Example (Oracle 10g):</p> <pre>hierarchy %DIMNHIER% (%DIMNATTRHIERFIRST% %DIMNATTRHIERLIST%)</pre> |
| AddJoin Hierarchy | <p>Specifies the syntax for defining a list of joins for hierarchy attributes.</p> <p>Example (Oracle 10g):</p> <pre>join key [.O:[(%DIMNKEYLIST%)] [%DIMNKEYLIST%]] references %DIMNPARENTLEVEL%</pre> |

| 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: <ul style="list-style-type: none"> • AfterCreate, AfterDrop, AfterModify • BeforeCreate, BeforeDrop, BeforeModify • Create, Drop • EnableSynonym • Header, Footer • ModifiableAttributes • ReversedQueries, ReversedStatements • SqlAttrQuery, SqlListQuery <p>For a description of each of these common items, see <i>Common object items</i> on page 144.</p> |
| 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 (DBMS)

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 |
|------------------|---|
| AmcdAmcd-Type | <p>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.</p> |
| AmcdDataType | <p>Lists mappings to convert from PowerDesigner (Internal) data types to DBMS (Physical Model) data types.</p> <p>These mappings are used during CDM to PDM generation and with the Change Current DBMS command.</p> <p>Examples (ASE 15):</p> <ul style="list-style-type: none"> • 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 | <p>Lists mappings to convert from DBMS (Physical Model) data types to PowerDesigner (Internal) data types.</p> <p>These mappings are used during PDM to CDM generation and with the Change Current DBMS command.</p> <p>Examples (ASE 15):</p> <ul style="list-style-type: none"> • 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 | <p>Lists the storage sizes of DBMS data types. These values are used when estimating the size of a database.</p> <p>Examples (ASE 15):</p> <ul style="list-style-type: none"> • The ASE 15 smallmoney requires 8 bytes of space. • The ASE 15 smalldatetime requires 4 bytes of space. |
| OdbcPhysDataType | <p>Lists mappings to convert from live database (ODBC) data types to DBMS (Physical Model) data types during database reverse engineering.</p> <p>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.</p> <p>Examples (ASE 15):</p> <ul style="list-style-type: none"> • 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 | <p>Lists mappings of DBMS (Physical Model) data types to database (ODBC) data types for use when updating and reverse engineering a database.</p> <p>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.</p> <p>Examples (ASE 15):</p> <ul style="list-style-type: none"> • A <code>unicar</code> is treated as equivalent to a <code>unicar (1)</code> in an ASE 15 database. • A <code>float (1)</code> is treated as equivalent to a <code>float (4)</code> in an ASE 15 database. |
| PhysLogADT Type | <p>Lists mappings to convert from DBMS (Physical Model) abstract data types to PowerDesigner (Internal) abstract data types.</p> <p>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.</p> <p>Examples (Oracle 11g):</p> <ul style="list-style-type: none"> • The Oracle 11g <code>VARRAY</code> abstract data type is converted to an <code>Array</code> for PowerDesigner. • The Oracle 11g <code>SQLJ_OBJECT</code> datatype is converted to a <code>JavaObject</code> for PowerDesigner. |
| LogPhysADT Type | <p>Lists mappings to convert from PowerDesigner (Internal) abstract data types to DBMS (Physical Model) abstract data types.</p> <p>These mappings are used with the Change Current DBMS command.</p> <p>Examples (Oracle 11g):</p> <ul style="list-style-type: none"> • The PowerDesigner <code>List</code> abstract data type is converted to a <code>TABLE</code> for Oracle 11g. • The PowerDesigner <code>Object</code> abstract data type is converted to an <code>OBJECT</code> for Oracle 11g. |
| AllowedADT | <p>Lists the abstract data types that may be used as types for columns and domains in the DBMS.</p> <p>Example (ASE 15):</p> <ul style="list-style-type: none"> • <code>JAVA</code> |

| Item | Description |
|--------------|---|
| HostDataType | <p>Lists mappings to convert from DBMS data types (Physical Model) to data types permitted as procedure parameters (Trigger).</p> <p>These mappings are used to populate the Data type field in ADT procedure parameter property sheets</p> <p>Examples (Oracle 11g):</p> <ul style="list-style-type: none"> • 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:

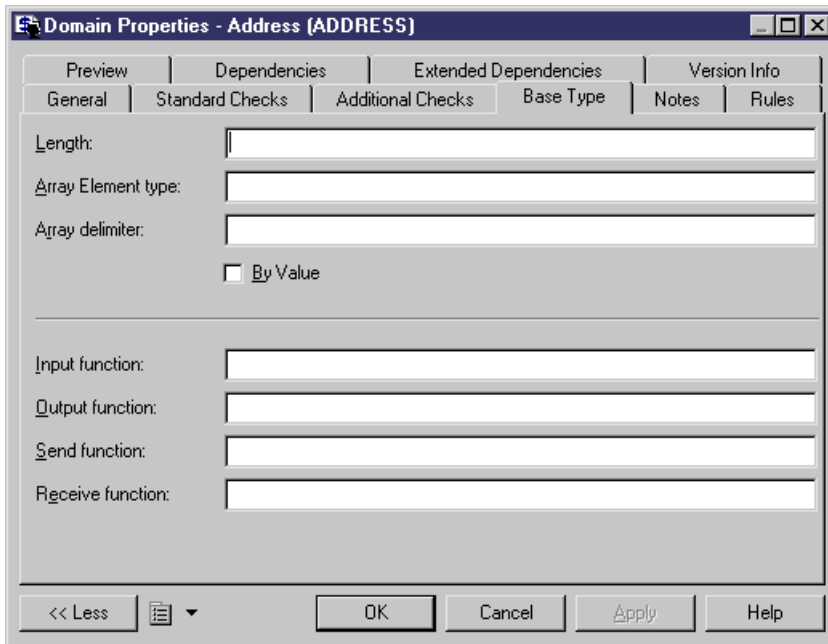
- your DBMS definition file - you should save a backup of this file before editing it.
- a separate extension file - which you attach to your model.

For detailed information about working with profiles, including adding extended attributes and objects, see *Chapter 2, Extension Files* on page 11.

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,
```

```

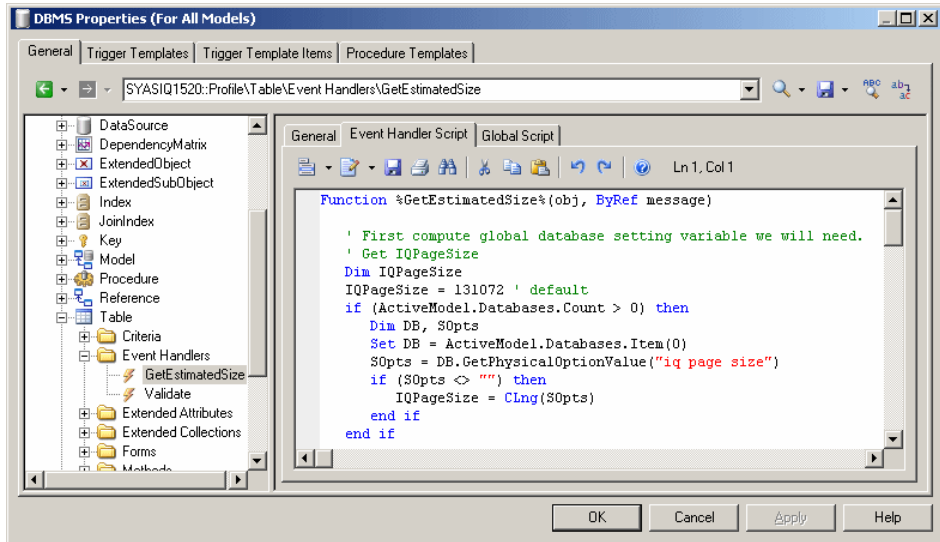
name,
case indextype when '2' then 'type 2' else 'type 1' end,
case uniquerule when 'D' then '' else 'unique' end,
case uniquerule when 'P' then 'primary' when 'U' then 'unique' else
'' end,
case clustering when 'Y' then 'cluster' else '' end,
case uniquerule when 'N' then 'TRUE' else 'FALSE' end
from
sysibm.sysindexes
where l=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.



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 205):

```
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 205). 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 following 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 unnecessary 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:

CHAPTER 4: DBMS Definition Files

```
message = CStr(TableName) & "||" & 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 'customerFKKeyIndex' (976 KB) Index 'descriptionIndex' (1976 KB) |
| | | [...etc...] |

Tables are allocated to the following tablespaces:

| Estimated size | Object |
|----------------|--------------------------|
| 6096 KB | Tablespace 'mainStorage' |
| | [...etc...] |

ODBC Category (DBMS)

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 scripts 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.

Physical Options (DBMS)

For some DBMSs, additional options are used to specify how an object is optimized or stored in a database. In PowerDesigner, these options are called *physical options* and are displayed on the **Physical Options** and **Physical Options (Common)** tabs of object property sheets.

To appear on the **Physical Options** tab, an option must be defined in the `Script\Objects\object\Options` item (see *Common Object Items* on page 144). Default values can be stored in `Options` or in `DefOptions`. To appear on the **Physical Options (Common)** tab (or any other property sheet tab), the physical option must, additionally be associated with an extended attribute (see *Adding DBMS Physical Options to Your Forms* on page 210).

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 and Macros* on page 212) 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%
```

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

Simple Physical Options

Simple physical options must contain a name, and may contain a `%d`, `%s`, or other variable to let the user specify a value, and keywords to specify permitted values and defaults.

Simple physical options are specified on a single line using the following syntax:

```
name [=] %s|%d|%variable% [: keywords]
```

Everything entered before the colon is generated in scripts. The *name* is required by PowerDesigner, but you can place it between carets (<*name*>) if you need to exclude it from the final script. The %d or %s variables require a numeric or string value, and you can also use a PowerDesigner variable or GTL snippet.

| Physical Option | Generates As |
|----------------------|---------------------------------|
| max_rows_per_page=%d | max_rows_per_page= <i>value</i> |
| for instance %s | for instance <i>string</i> |
| <Partition-name> %s | <i>name</i> |

You can insert a colon followed by comma-separated keywords to control your options:

| Keyword | Value and result |
|--------------------------------|---|
| category= <i>meta-class</i> | <p>Allows the user to associate the object with an object of the specified kind. The following settings are available:</p> <ul style="list-style-type: none"> • tablespace • storage <p>Note: In Oracle, the <i>storage</i> composite physical option is used as a template to define all the storage values in a storage entry to avoid having to set values independently each time you need to re-use them same values in a storage clause. For this reason, the Oracle physical option does not include the <i>storage name</i> (%s).</p> <ul style="list-style-type: none"> • <i>qualified metaclass collection</i> - For example: Model.Tables or Table.Columns <pre>on %s : category=storage {</pre> |
| list= <i>value value</i> | Specifies a list of pipe-separated values permitted for the option. |
| default= <i>value</i> | Specifies a default value for the option. |
| dquoted=yes and squoted=yes | Specifies that the value is enclosed in double or single quotes. |

| Keyword | Value and result |
|--|--|
| multi- ple=yes | Specifies that the option is displayed with a <*> suffix in the left pane of the Physical Options tab and can be added to the right pane as many times as necessary. If the option is selected in the right pane and you click the same 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 , a second instance of the option is added to the right pane. |
| enable- dbpre- fix=yes | Specifies that the database name is inserted as a prefix (see tablespace options in DB2 OS/390). |
| pre- vmand=yes and next- mand=yes | Specifies that the previous or next physical option is required for the present option and that if the present option is added to the right pane, then the previous or next option is also added. |

Examples

| Physical Option | Generates As |
|---|---------------|
| ccsid %s : list=ascii ebcdic unicode, default=ascii | ccsid ascii |
| table=%s : category=Model.Ta- bles, dquoted=yes | table="table" |
| <flashback_archive> %s | <i>string</i> |

Composite Physical Options

Composite physical options are specified over multiple lines, and contain one or more dependent options. If you add the composite option to the right pane of the **Physical Options** tab, all the dependant options are added with it. If you add a dependant option, the composite option is added as well to contain it.

Composite physical options are defined with the following syntax:

```
name [=] [%s|%d|%variable%] : composite=yes[, keywords]
{
  sub-option
  [sub-option...]
}
```

Everything entered before the colon is generated in scripts. The *name* is required by PowerDesigner, but you can place it between carets (<name>) if you need to exclude it from the final script. The %d or %s variables require a numeric or string value, and you can also use a PowerDesigner variable or GTL snippet.

CHAPTER 4: DBMS Definition Files

The `composite=yes` keyword is required for composite options, and can be used in conjunction with any of the simple physical option keywords or any of the following:

| Keyword | Value and result |
|------------------------------|--|
| <code>composite=yes</code> | Specifies that the option is a composite option containing dependant options surround by curly braces. |
| <code>separator=yes</code> | Specifies that the dependant options are separated by commas. |
| <code>parenthesis=yes</code> | Specifies that the ensemble of dependant objects are contained between parentheses. |
| <code>chldmand=yes</code> | Specifies that at least one of the dependant options must be set. |

Examples

| Physical Option | Generates As |
|--|---|
| <pre><list> : composite=yes, multiple=yes { <frag-expression> %s in %s : category=storage }</pre> | <pre><i>frag-expression</i> in <i>storage</i> <i>frag-expression2</i> in <i>storage2</i> etc</pre> |
| <pre><using_block> : composite=yes, parenthesis=yes { using vcat %s using stogroup %s : category=storage, composite=yes { priqty %d : default=12 secqty %d erase %s : default=no, list=yes } }</pre> | <pre>(using vcat <i>string</i> using stogroup <i>storage</i> priqty <i>value</i> secqty <i>value</i> erase <i>no</i>)</pre> |

Adding DBMS Physical Options to Your Forms

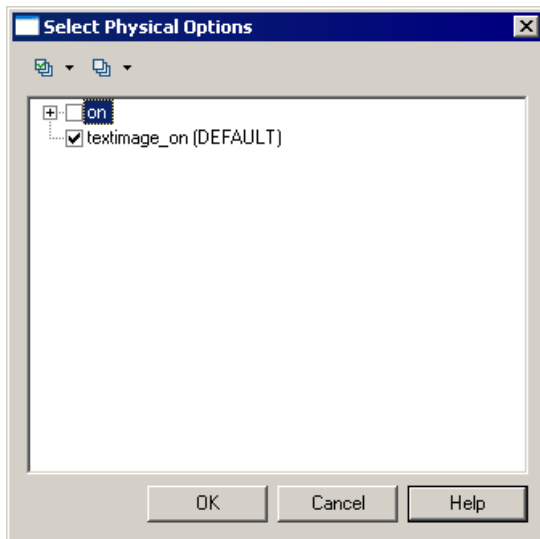
Many DBMSs use *physical options* as part of the definition of their objects. The most commonly-used physical options are displayed on a form, **Physical Options (Common)**,

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 (DBMS)* on page 207).

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 57).

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.

PDM Variables and Macros

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

For example, in the following query, the variable %TABLE% will be replaced by the code of the table being created:

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

Note: You can use these variables freely in your own queries, but you cannot change the method of their evaluation (ie, %TABLE% can only ever evaluate to the code of the table). You can alternately, access any object properties using GTL (see *Chapter 5, Customizing Generation with GTL* on page 245) and the public names available through the PowerDesigner metamodel (see *Chapter 8, The PowerDesigner Public Metamodel* on page 343).

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

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) |
| %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 |

| Variable | Comment |
|-----------------|---|
| %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 |

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 |
|--|---|
| [% <i>variable</i> %] | Tests for the existence of the variable. Generation: Generated only if <i>variable</i> 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. |
| [% <i>variable</i> %? <i>true</i> : <i>false</i>] | Tests for the existence of the variable and allows conditional output. Generation: <i>true</i> is generated if <i>variable</i> exists and is not assigned NO or FALSE. Otherwise, <i>false</i> is generated. Reverse: If the parser detects <i>variable</i> and it is not assigned NO or FALSE, <i>true</i> is reversed. Otherwise, <i>false</i> is reversed. <i>variable</i> is set to True or False as appropriate. |
| [% <i>variable</i> %= <i>constant</i> ? <i>true</i> : <i>false</i>] | Tests the value of the variable and allows conditional output. Generation: If <i>variable</i> equals <i>constant</i> , <i>true</i> is generated. Otherwise, <i>false</i> is generated. Reverse: If the parser detects that <i>variable</i> equals <i>constant</i> , <i>true</i> is reversed. Otherwise, <i>false</i> is reversed. |

| Variable | Generation |
|---|--|
| <code>[.Z: [item1] [item2]...]</code> | <p>Specifies that the <i>items</i> do not have a significant order.</p> <p>Generation: <code>.Z</code> is ignored</p> <p>Reverse: The <i>items</i> can be reversed in any order they are encountered.</p> |
| <code>[.O: [item1] [item2]...]</code> | <p>Specifies that the <i>items</i> are synonyms, only one of which should be output.</p> <p>Generation: Only the first <i>item</i> listed is generated.</p> <p>Reverse: The reverse parser must find one of the <i>items</i> to validate the full statement.</p> |

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:

```
%[[?][-][x][.[-]y][options]:]variable%
```

The variable formatting options are the following:

| Option | Description | | | | | | | | | | | | | | | | | | | | | | | | |
|--|--|-------------------------------|--|--------|--------------------------|-------|--------------------|----------------------------|-------|--------------------|----------------------------|-------|-------------------------------|------------------------------|-------|-------------------------------|-----------------------------|-------|-------------------------------|-------------------------------|-------|-------------------------------|-----------------------------|-------|--------------------|
| ? | Mandatory field, if a null value is returned the translate call fails | | | | | | | | | | | | | | | | | | | | | | | | |
| <code>[-][x].[-]y[M]</code> | <p>Extracts the first <i>y</i> characters or, for <i>-y</i>, the last <i>y</i> characters.</p> <p>If <i>x</i> is specified, and <i>y</i> is lower than <i>x</i>, then blanks or zeros are added to the right of the extracted characters to fill the width up to <i>x</i>. For <i>-x</i>, the blanks or zeros are added to the left and the output is right-justified.</p> <p>If the <i>M</i> option is appended, then the first <i>x</i> characters of the variable are discarded and the next <i>y</i> characters are output.</p> <p>Thus, for an object named <code>abcdefghijklmnopqrstuvwxyz</code> (with parentheses present simply to demonstrate padding):</p> <table border="1"> <thead> <tr> <th>Template</th> <th></th> <th>Output</th> </tr> </thead> <tbody> <tr> <td><code>(%.3:Name%)</code></td> <td>gives</td> <td><code>(abc)</code></td> </tr> <tr> <td><code>(%. -3:Name%)</code></td> <td>gives</td> <td><code>(xyz)</code></td> </tr> <tr> <td><code>(%10.3:Name%)</code></td> <td>gives</td> <td><code>(abc)</code></td> </tr> <tr> <td><code>(%10. -3:Name%)</code></td> <td>gives</td> <td><code>(xyz)</code></td> </tr> <tr> <td><code>(%-10.3:Name%)</code></td> <td>gives</td> <td><code>(abc)</code></td> </tr> <tr> <td><code>(%-10. -3:Name%)</code></td> <td>gives</td> <td><code>(xyz)</code></td> </tr> <tr> <td><code>(%10.3M:Name%)</code></td> <td>gives</td> <td><code>(jkl)</code></td> </tr> </tbody> </table> | Template | | Output | <code>(%.3:Name%)</code> | gives | <code>(abc)</code> | <code>(%. -3:Name%)</code> | gives | <code>(xyz)</code> | <code>(%10.3:Name%)</code> | gives | <code>(abc)</code> | <code>(%10. -3:Name%)</code> | gives | <code>(xyz)</code> | <code>(%-10.3:Name%)</code> | gives | <code>(abc)</code> | <code>(%-10. -3:Name%)</code> | gives | <code>(xyz)</code> | <code>(%10.3M:Name%)</code> | gives | <code>(jkl)</code> |
| Template | | Output | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%.3:Name%)</code> | gives | <code>(abc)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%. -3:Name%)</code> | gives | <code>(xyz)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%10.3:Name%)</code> | gives | <code>(abc)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%10. -3:Name%)</code> | gives | <code>(xyz)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%-10.3:Name%)</code> | gives | <code>(abc)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%-10. -3:Name%)</code> | gives | <code>(xyz)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%10.3M:Name%)</code> | gives | <code>(jkl)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>L[F]</code> , <code>U[F]</code> , and <code>c</code> | Converts the output to lowercase or uppercase. If <i>F</i> is specified, only the first character is converted. <i>c</i> is equivalent to <i>UF</i> . | | | | | | | | | | | | | | | | | | | | | | | | |
| <code>q</code> and <code>Q</code> | Surrounds the variable with single or double quotes. | | | | | | | | | | | | | | | | | | | | | | | | |
| <code>T</code> | Trims leading and trailing whitespace from the variable. | | | | | | | | | | | | | | | | | | | | | | | | |
| <code>H</code> | Converts number to hexadecimal. | | | | | | | | | | | | | | | | | | | | | | | | |

You can combine format codes. For example, the template `(%12.3QMFU:Name%)` applied to object `abcdefghijklmnopqrstuvwxyz` generates `("Lmn")`.

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 |

| Variable | Comment |
|----------|---|
| %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 <i>Working with Null Values</i> on page 159 |
| %PKEYCLUSTER% | CLUSTER keyword for the primary key when it is defined on the same line |

| Variable | Comment |
|-----------------|---|
| %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 |
| %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) |

| Variable | Comment |
|---------------|--|
| %LISTVAL% | SQL constraint associated with List of values. Ex: C1 in (0, 1, 2, 3, 4, 5) |
| %MINMAX% | SQL constraint associated with Min and max values. Ex: (C1 <= 0) AND (C1 >= 5) |
| %ISMAND% | TRUE if Domain or column is mandatory |
| %MAND% | Contains Keywords "null" or "not null" depending on Mandatory attribute |
| %NULL% | Contains Keyword "null" if Domain or column is not mandatory |
| %NOTNULL% | Contains Keyword "not null" if Domain or column is mandatory |
| %IDENTITY% | Keyword "identity" if Domain or Column is identity (Sybase specific) |
| %WITHDEFAULT% | Keyword "with default" if Domain or Column is with default |
| %ISUPPERVAL% | TRUE if the upper-case attribute of standard check has been selected |
| %ISLOWERVAL% | TRUE if the lower-case attribute of standard check has been selected |
| %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) |
| %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 |

| Variable | Comment |
|----------------|--|
| %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 |
| %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 |
| %AK% | Alternate key column code (same as PK) |
| %AKNAME% | Alternate key column name (same as PKNAME) |
| %COLTYPE% | Primary key column data type |
| %COLTYPEOWNER% | 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.

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 |

| Variable | Comment |
|--------------------------|---|
| %MSGTAB% | Name of Table containing user-defined error messages |
| %MSGNO% | Name of Column containing Error numbers in User-defined error table |
| %MSGTXT% | Name of Column containing Error messages in User-defined error table |
| %SCRIPT% | SQL script of trigger or procedure. |
| %TRGBODY% | Trigger body (only for Oracle live database reverse engineering) |
| %TRGDESC% | Trigger description (only for Oracle live database reverse engineering) |
| %TRGDEFN% | Trigger definition |
| %TRGSCOPE% | Trigger scope (keywords: database, schema, all server) |
| %TRGSCOPEOWNER % | Trigger scope owner |
| %TRGSCOPEQUALI- FIER% | 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 |
| %RULECEXP% | Rule client expression |
| %RULESEXP% | 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 |

| Variable | Comment |
|-----------------------|--|
| %TYPE% | Type of Abstract data type. It contains keywords like "array", "list", ... |
| %SIZE% | Abstract data type size |
| %FILE% | Abstract data type Java file |
| %ISARRAY% | TRUE if Abstract data type is of type array |
| %ISLIST% | TRUE if Abstract data type is of type list |
| %ISSTRUCT% | TRUE if Abstract data type is of type structure |
| %ISOBJECT% | TRUE if Abstract data type is of type object |
| %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 |
| %SUPERQUALIFI- ER% | 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 |

CHAPTER 4: DBMS Definition Files

The following variables are available for abstract data type attributes:

| Variable | Comment |
|-----------------|--|
| %ADTATTR% | Generated code of Abstract data type attribute |
| %ATTRJAVA-NAME% | 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 |
| %PROCNOTFINAL% | Procedure final |
| %PROCSTATIC% | Procedure member |
| %PROCABSTRACT% | Procedure instantiable |
| %SUPERPROC% | Procedure super-procedure |
| %ISCONSTRUCTOR% | 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) |
| %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 |

| Variable | Comment |
|-----------|----------------------|
| %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 DB 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 |

| Variable | Comment |
|-----------------|----------------------|
| %DBPKPROCPARAM% | Procedure parameters |

The following variables are available for database package variables:

| Variable | Comment |
|----------------|-------------------------------------|
| %DBPFVAR% | Variable code |
| %DBPFVARTYPE% | Variable type |
| %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 |
|------------------------|-------------------------|
| %DBPKPARAM% | Parameter code |
| %DBPKPARMTYPE% | Parameter type |
| %DBPKPARMDTTP% | Parameter data type |
| %DBPKPARAMDEFAULT % | 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 |

| Variable | Comment |
|----------------|--|
| %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 |
| %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 Web services.

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 |
| %WEBSERVICESTSQL% | 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% | Web operation type |
| %WEBOPERATIONSQ% | SQL statement |
| %WEBOPERATIONPARAM% | Web operation parameters list |

The following variables are available for web service security:

| Variable | Comment |
|---------------------|--|
| %WEBUSER% | Connection user required for web service |
| %WEBCONNECTSECURED% | 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 |
| %WEBPARAMDTP% | 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 |

| Variable | Comment |
|---------------------|---------------------------------|
| %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 |
| %DIMNATTRHIERFIRST% | First attribute of hierarchy |
| %DIMNATTRHIERLIST% | List of attributes of hierarchy |
| %DIMNPARENTLEVEL% | Parent level for hierarchy |
| %DIMNDEPATTR% | Dimension dependant attribute |
| %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) |

| Variable | Comment |
|-----------------|--|
| %EXTOBJECTDEFN% | Complete body of the extended object definition. Contains definition of extended collection listed in DefinitionContent DBMS item. |

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 |

| Variable | Comment |
|---------------|---|
| %ISUPPER% | TRUE if upper case generation option is set |
| %ISLOWER% | TRUE if lower case generation option is set |
| %DBMSNAME% | Name of DBMS associated with Generated model |
| %DATABASE% | Code of Database associated with Generated model |
| %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, .FKCOLN, and .PKCOLN Macros

Repeat a statement for each alternate, foreign, or primary key column in a table.

Syntax

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

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

```
.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 |
| 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:

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

generates the following trigger script:

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

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

generates the following trigger script:

CHAPTER 4: DBMS Definition Files

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

- `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, these macros only accept the `%COLUMN%` variable.

.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")
```

.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 |
| 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 |
| 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 |
| 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);
```

.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:

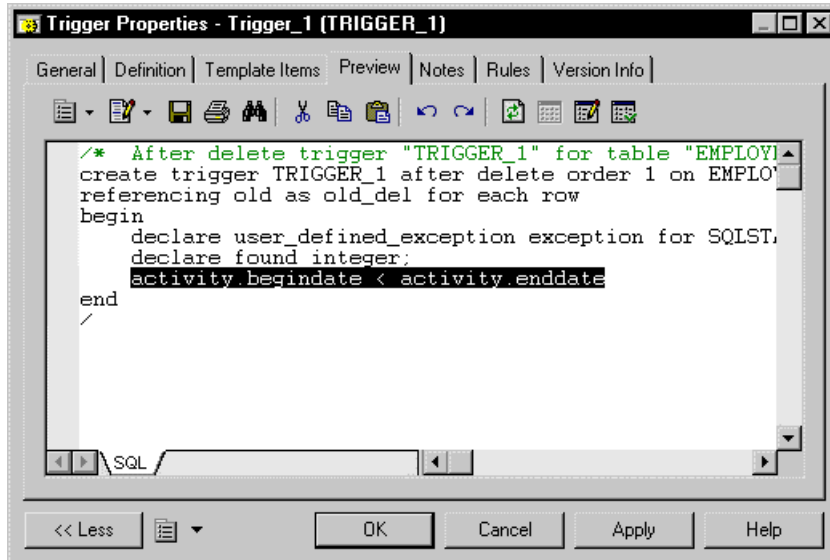
```
activity.begindate < activity.enddate
```

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
```



.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.

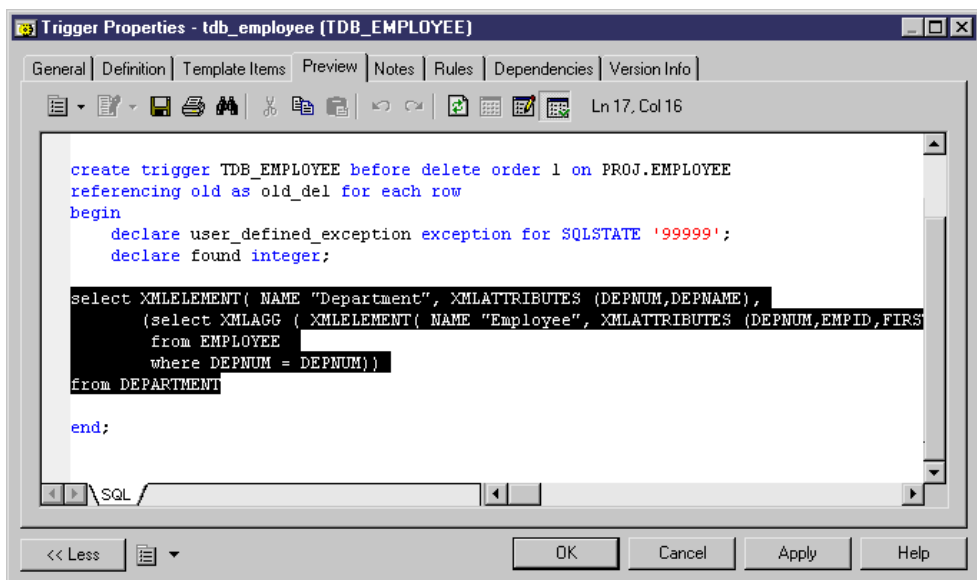
Example

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
```



Customizing Generation with GTL

The PowerDesigner Generation Template Language (GTL) is used to extract model object properties as text. GTL is written in *templates* and *generated files* defined under metaclasses in language definition and extension files. It powers generation of code for business process, object-oriented and XML languages, and can be used to define new generations for any model.

When you launch a generation from a model, PowerDesigner generates a file for each instance of each metaclass for which you have defined a generated file (see *Generated Files (Profile)* on page 85) by evaluating the templates it calls and resolving any variables.

GTL is object-oriented, supporting inheritance and polymorphism for reusability and maintainability, and provides macros 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

Note: Though GTL can be used to extend generation in a PDM, the standard generation is primarily defined using a different mechanism (see *Database Generation and Reverse Engineering* on page 120).

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. Open your language definition or extension file in the resource editor (see *Opening Resource Files in the Editor* on page 3).
2. If necessary, add a metaclass to the Profile category (see *Metaclasses (Profile)* on page 33) and then right-click it and select **New > Template** (see *Templates (Profile)* on page 84).
3. Enter `helloWorld` as the name of the template and enter the following code in the text box:

```
Hello World!  
This template is being generated for the %Name% object.
```

Note: We recommend that you 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.

4. Right-click the metaclass again, and select **New > Generated File** (see *Generated Files (Profile)* on page 85).
5. Enter `myFile` as the name of the generated file, and enter the following code in the text box to call your template:


```
%helloWorld%
```
6. Click **OK** to save your changes in the resource file and return to your model.
7. Create an instance of the metaclass on which you defined the template and generated file, open its property sheet, and click the **Preview** tab.
8. Select the **myFile** sub-tab to preview what would be generated for this object.

Extracting Object Properties

Object properties are referenced as variables and enclosed between percent signs: `%variable%`. Variable names are case sensitive, and property names are, by convention, defined in CamelCase.

Properties are extracted as the following types:

- String - returns text.
- Boolean - returns `true` or `false`.
- Object - returns the object ID or `null`.

Example

```
This file is generated for %Name%, which is a %Color% %Shape%.
```

Result:

```
This file is generated for MyObject, which is a Red Triangle.
```

Standard properties defined in the PowerDesigner public metamodel (see *Chapter 8, The PowerDesigner Public Metamodel* on page 343) are referenced using their public names, which are written in CamelCase. You can infer public names for many properties from their labels in object property sheets, but in case of doubt, click the **Property Sheet Menu** button at the bottom of the property sheet and select **Find in Metamodel Objects Help** to review all available properties for the object.

Extended attributes (see *Extended Attributes (Profile)* on page 41) are referenced by their **Name** defined in the resource editor.

Note: To access an extended attribute defined in another extension file attached to the model, prefix the name with the `.D` formatting option. For example:

```
%.D:MyExtAtt%
```

Accessing Collections of Sub-Objects or Related Objects

An OOM contains a collection of classes and classes contain collections of attributes and operations. To iterate over a collection, use the `.foreach_item` macro.

Example

```
%Name% contains:
.foreach_item(Widgets)
    \n\t%Name% (%Color% %Shape%)
.next
```

Result:

```
MyObject contains:
    Widget1 (Red Triangle)
    Widget2 (Yellow Square)
    Widget3 (Green Circle)
```

Standard collections defined in the PowerDesigner public metamodel (see *Chapter 8, The PowerDesigner Public Metamodel* on page 343) are referenced using their public names, which are written in CamelCase. You can infer public names for many collections from their labels in object property sheet tabs, but in case of doubt, click the **Property Sheet Menu** button at the bottom of the property sheet and select **Find in Metamodel Objects Help** to review all available collections for the object.

Extended collections (see *Extended Collections and Compositions (Profile)* on page 48 and *Calculated Collections (Profile)* on page 50) are referenced by their **Name**.

You can use the following keywords to access information about a collection:

| Name | Description |
|---------|---|
| First | (object) Returns the first element of the collection. |
| IsEmpty | (boolean) Returns <code>True</code> if the collection is empty, or <code>false</code> if it contains one or more members. |
| Count | (integer) Returns the number of elements in the collection. You can use this keyword for defining criteria based on collection size, for example <code>Attributes.Count>=10</code> . |

| Example |
|--|
| <pre>%Name% is associated with %AttachedRules.Count% business rules, of which the first is %AttachedRules.First.Name%.</pre> |
| <p>Result:</p> <pre>myClass is associated with 3 business rules, of which the first is myRule.</pre> |

Formatting Your Output

You can change the formatting of variables by embedding formatting options in variable syntax. New lines and tabs are specified using the `\n` and `\t` escape sequences respectively.

```
%[[-] [x] [. [-] y] [options] :] variable%
```

The following variable formatting options are available:

| Option | Description | | | | | | | | | | | | | | | | | | | | | | | | |
|--|--|---------------------|-------|--------|--------------------------|-------|--------------------|---------------------------|-------|--------------------|----------------------------|-------|---------------------|-----------------------------|-------|---------------------|-----------------------------|-------|---------------------|------------------------------|-------|---------------------|-----------------------------|-------|--------------------|
| <code>[-] [x] . [-] y [M]</code> | <p>Extracts the first <i>y</i> characters or, for <code>-y</code>, the last <i>y</i> characters.</p> <p>If <i>x</i> is specified, and <i>y</i> is lower than <i>x</i>, then blanks or zeros are added to the right of the extracted characters to fill the width up to <i>x</i>. For <code>-x</code>, the blanks or zeros are added to the left and the output is right-justified.</p> <p>If the <i>M</i> option is appended, then the first <i>x</i> characters of the variable are discarded and the next <i>y</i> characters are output.</p> <p>Thus, for an object named <code>abcdefghijklmnopqrstuvwxy</code> (with parentheses present simply to demonstrate padding):</p> <table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">Template</th> <th style="text-align: left;">gives</th> <th style="text-align: left;">Output</th> </tr> </thead> <tbody> <tr> <td><code>(%.3:Name%)</code></td> <td>gives</td> <td><code>(abc)</code></td> </tr> <tr> <td><code>(%.-3:Name%)</code></td> <td>gives</td> <td><code>(xyz)</code></td> </tr> <tr> <td><code>(%10.3:Name%)</code></td> <td>gives</td> <td><code>(abc)</code></td> </tr> <tr> <td><code>(%10.-3:Name%)</code></td> <td>gives</td> <td><code>(xyz)</code></td> </tr> <tr> <td><code>(%-10.3:Name%)</code></td> <td>gives</td> <td><code>(abc)</code></td> </tr> <tr> <td><code>(%-10.-3:Name%)</code></td> <td>gives</td> <td><code>(xyz)</code></td> </tr> <tr> <td><code>(%10.3M:Name%)</code></td> <td>gives</td> <td><code>(jkl)</code></td> </tr> </tbody> </table> | Template | gives | Output | <code>(%.3:Name%)</code> | gives | <code>(abc)</code> | <code>(%.-3:Name%)</code> | gives | <code>(xyz)</code> | <code>(%10.3:Name%)</code> | gives | <code>(abc)</code> | <code>(%10.-3:Name%)</code> | gives | <code>(xyz)</code> | <code>(%-10.3:Name%)</code> | gives | <code>(abc)</code> | <code>(%-10.-3:Name%)</code> | gives | <code>(xyz)</code> | <code>(%10.3M:Name%)</code> | gives | <code>(jkl)</code> |
| Template | gives | Output | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%.3:Name%)</code> | gives | <code>(abc)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%.-3:Name%)</code> | gives | <code>(xyz)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%10.3:Name%)</code> | gives | <code>(abc)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%10.-3:Name%)</code> | gives | <code>(xyz)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%-10.3:Name%)</code> | gives | <code>(abc)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%-10.-3:Name%)</code> | gives | <code>(xyz)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>(%10.3M:Name%)</code> | gives | <code>(jkl)</code> | | | | | | | | | | | | | | | | | | | | | | | |
| <code>L [F] , U [F] , and c</code> | Converts the output to lowercase or uppercase. If <i>F</i> is specified, only the first character is converted. <i>c</i> is equivalent to <code>UF</code> . | | | | | | | | | | | | | | | | | | | | | | | | |
| <code>q</code> and <code>Q</code> | Surrounds the variable with single or double quotes. | | | | | | | | | | | | | | | | | | | | | | | | |
| <code>A</code> | Removes indentation and aligns text on the left border. | | | | | | | | | | | | | | | | | | | | | | | | |
| <code>T</code> | Trims leading and trailing whitespace from the variable. | | | | | | | | | | | | | | | | | | | | | | | | |
| <code>H</code> | Converts number to hexadecimal. | | | | | | | | | | | | | | | | | | | | | | | | |

| Option | Description |
|--------|--|
| D | <p>Returns the human-readable value of an attribute used in the PowerDesigner interface when this value differs from the internal representation.</p> <p>For example, the value of the <code>Visibility</code> attribute is stored internally as <code>+</code>, but is displayed as <code>public</code> in the property sheet. The template <code>%Visibility%</code> generates as <code>+</code>, but <code>%.D:Visibility%</code> generates as <code>public</code>.</p> <p>Note: You can access extended attributes defined in another extension file by prefixing them with the <code>.D</code> option (see <i>Extracting Object Properties</i> on page 246).</p> |
| X | Escapes XML forbidden characters. |
| E | [deprecated – use the <code>!</code> power evaluation operator instead, see <i>GTL Operators</i> on page 252]. |

| Examples |
|---|
| <pre>This file is generated for %.UQ:Name%. It has the form of a %.L:Color% %.L:Shape%.</pre> |
| <pre>This file is generated for "MYGADGET". It has the form of a red triangle.</pre> |
| <p>The following template is applied to object abcdefghijklmnopqrstuvwxyz</p> <pre>%12.3QMFU:Name%</pre> <p>Result:</p> <pre>"Lmn"</pre> |

Controlling Line Breaks in Head and Tail Strings

The head and tail strings in a macro block are only generated when necessary. If the block returns nothing then the head and tail strings do not appear, which can help to control the creation of new lines.

Example

The text and new lines in the head and tail of each `.foreach_item` loop are only printed if the collection is not empty. When this template is applied to a class with attributes but no operations, the text `// Operations` and the new lines specified before and after the operations list will not be printed:

```
class "%Code%" {
  .foreach_item(Attributes, // Attributes\n,\n\n)
  %DataType% %Code%
  .if (%InitialValue%)
  = %InitialValue%
  .endif
  .next(\n)
  .foreach_item(Operations, // Operations\n,\n\n)
  %ReturnType% %Code%(...)
  .next(\n)
<Source>
}
```

Result:

```
class "C1" { // Attributes
  int a1 = 10
  int a2
  int a3 = 5
  int a4
<Source>
}
```

Note: To print a blank space between the curly brace and the string `// Attributes`, you must enclose the head string in double-quotes:

```
.foreach_item(Attributes, " // Attributes\n", \n)
```

Conditional Blocks

Place text containing a variable between square brackets to have it appear only if the variable resolves to a non-null value.

You can also use a form similar to C and Java ternary expressions to print a string if the variable is true or not null:

```
[variable ? ifNotNull]
```

You can optionally include a string to print if the variable is evaluated to false, null, or the empty string:

```
[variable ? ifNotNull :ifNull]
```

| Examples |
|--|
| <pre>Attribute %Code%[= %InitialValue%];</pre> <p>Result:</p> <pre>Attribute A1 =0; Attribute A2 =100; Attribute A3; Attribute A4 =10;</pre> |
| <pre>The class %Name% is [%Abstract%?Abstract:Concrete].</pre> <p>Result if the Abstract property is selected:</p> <pre>The class myClass is Abstract.</pre> <p>Result if the Abstract property is not selected:</p> <pre>The class myClass is Concrete.</pre> |

Accessing Global Variables

You can insert information such as your user name and the current date with global variables.

| Name | Description |
|---------------|---|
| %ActiveModel% | (object) Returns the UID of the model. Use %ActiveModel.Name% to obtain the name of the model. |
| %GenOptions% | (struct) Returns the model generation options. |
| %PreviewMode% | (boolean) Returns <code>true</code> in the Preview tab, <code>false</code> when generated to a file. |
| %CurrentDate% | (string) Returns the current system date and time formatted using local settings. |
| %CurrentUser% | (string) Returns the current user login. |
| %NewUUID% | (string) Returns a new universally unique identifier. |

Example

```
This file was generated from %ActiveModel.Name% by %CurrentUser% on
%CurrentDate%.
```

Result:

```
This file was generated from My Model by jsmith on Tuesday, Novem-
ber 06, 2012 4:06:41 PM.
```

GTL Operators

GTL supports standard arithmetic and logical 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 |
|-------------|---|
| = | Assignment operator. |
| == and != | Equal to and not equal to operators. |
| > and < | Greater than and less than operators. |
| >= and <= | Greater than or equal to and less than or equal to operators. |
| && and | Logical AND and logical OR operators. |
| %+ (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% | Base number= 4 |
| Number+1= %+ (Number, 1) % | Number+1= 5 |
| Number-1= %- (Number, 1) % | Number-1= 3 |
| Number*2= %* (Number, 2) % | Number*2= 8 |
| Number/2= %/ (Number, 2) % | Number/2= 2 |
| Number&1= %& (Number, 1) % | Number&1= 0 |

The following advanced template operators are also supported:

| Operator | Description |
|----------|---|
| <p>*</p> | <p>Dereferencing operator - Corresponds to a double evaluation, returning a template instead of text, using the syntax:</p> <pre data-bbox="397 331 744 354">%*template [(P1,P2...)]%</pre> <p>For information about template parameters, see <i>Passing Parameters to a Template</i> on page 260.</p> <p>In the following example, a local variable is returned normally and in a dereferenced form:</p> <pre data-bbox="397 531 666 600">.set_value(C, Code) %C% %*C%</pre> <p>Result:</p> <pre data-bbox="397 666 485 713">Code %Code%</pre> |
| <p>!</p> | <p>Power evaluation operator - Evaluates the results of the evaluation of the variable as a template.</p> <p>In the following example, a local variable is returned normally and in a power-evaluated form:</p> <pre data-bbox="397 904 825 973">.set_value(C, %%MyAttribute%%) %C% %!C%</pre> <p>Result:</p> <pre data-bbox="397 1039 585 1086">%MyAttribute% Red</pre> <p>The ! operator may be applied any number of times. For example:</p> <pre data-bbox="397 1152 471 1175">%!!t%</pre> <p>This outputs the results of the evaluation of the evaluation of the evaluation of template t.</p> |

| Operator | Description |
|----------|--|
| ? | <p>Existence operator - Tests whether a template, local variable, or property is present, and returns <code>false</code> if it is not.</p> <p>For example:</p> <pre data-bbox="397 343 1180 444">.set_value (myVariable, 20, new) %myVariable?% .unset (myVariable) %myVariable?%</pre> <p>Result:</p> <pre data-bbox="397 513 1180 565">true false</pre> |
| + | <p>Visibility operator - Tests whether an object property is visible in the interface, and returns <code>false</code> if it is not.</p> <p>For example, to test if the Type field is displayed in the General tab of a database property sheet in a DMM (meaning that a Replication Server extension file is attached to the model), enter the following:</p> <pre data-bbox="397 782 1180 808">%Database.Type+%</pre> |

Translation Scope

The initial scope of a template is always the metaclass on which it is defined. All standard and extended attributes, collections, and templates defined on the active object metaclass and its parents are visible, but only one object is active at any given time.

Examples

The following template is applied to a package P1, which contains a class C1, which contains operations O1 and O2, which each contain parameters P1 and P2. The scope changes, affecting the value of the %Name% variable, as each collection is traversed. The `Outer` keyword is used to return temporarily to previous scopes:

```
%Name%
.foreach_item(Classes)
  \n\t*%Name% in %Outer.Name%
  .foreach_item(Operations)
    \n\t*%Name% in %Outer.Name% in %Outer.Outer.Name%
    .foreach_item(Parameters)
      \n\t\t*%Name% in %Outer.Name% in %Outer.Outer.Name% in
%Outer.Outer.Outer.Name%
  .next
.next
.next
```

Result:

```
P1
 *C1 in P1
 *O1 in C1 in P1
   *P1 in O1 in C1 in P1
   *P2 in O1 in C1 in P1
 *O2 in C1 in P1
   *P1 in O2 in C1 in P1
   *P2 in O2 in C1 in P1
```

The `Outer` scope is restored when you leave a `.foreach_item` block. Nested scopes form a hierarchy that can be viewed as a tree, with the top level scope being the root. Use `Parent` instead of `Outer` to climb above the scope of the original object. For example, nothing will be output if the following template is applied to the parameter P1:

```
%Name% in %Outer.Name% in %Outer.Outer.Name%
```

However, this template will produce output:

```
%Name% in %Parent.Name% in %Parent.Parent.Name%
```

Result:

```
P1 in O1 in C1
```

Shortcut Translation

Shortcuts are dereferenced during translation, so that the scope of the target object replaces the scope of the shortcut. This is different from VB Script where shortcut translation retrieves the shortcut itself. You can use the `%IsShortcut%` variable to test whether an object is a shortcut, and the `Shortcut` keyword to access the properties of the shortcut itself.

| Template |
|--|
| <p>In this example, the template is applied to an OOM package P1 containing two classes and two shortcuts to classes in P2:</p> <pre>.foreach_item(Classes) \n*Class %Code% [%IsShortcut% ? From package %Package.Name% : Local Object] .next</pre> <p>Result:</p> <pre>*Class C1 Local Object *Class C2 Local Object *Class C3 From package P2 *Class C4 From package P2</pre> |

Note: If your model contains shortcuts to objects in another model that is not open, a dialog box invites you to open the target model. You can use the `.set_interactive_mode` macro to change this behavior (see *.set_interactive_mode Macro* on page 278).

Escape Sequences

GTL supports a number of escape sequences to simplify the layout of your templates and generated files, and to make reserved characters accessible.

The following escape sequences can be used inside templates:

| Escape sequence | Description |
|-------------------------------------|---|
| <code>\n</code> | New line. For examples of using new lines in macro blocks, see <i>Controlling Line Breaks in Head and Tail Strings</i> on page 250. |
| <code>\t</code> | Tab |
| <code>\\</code> | Backslash |
| <code>\</code> at end of line | Continuation character (ignores the new line) |
| <code>.</code> at beginning of line | Comment. Ignores the line. |

| Escape sequence | Description |
|-------------------------|--------------------------------------|
| .. at beginning of line | Dot character (to generate a macro). |
| %% | Percent character. |

Calling Templates

You can call a template from a generated file or from another template by entering its name surrounded by percentage signs. Object properties, collections, and local and global variables are called in the same way. At generation time, a template call is replaced by the template content, which is then resolved to its final textual value.

Examples:

- `%Name%` - Calls the object's **Name** property
- `%myTemplate%` - Calls the `%myTemplate%` template
- `%CurrentDate%` - Calls the `%CurrentDate%` global variable (see *Accessing Global Variables* on page 251)

Breaking templates into concise units and calling them at generation time helps with readability and reuse. For example, you can define a commonly-used condition in one template and reference it in multiple other templates:

| Example |
|--|
| <p>The <code>%isInner%</code> template is defined as:</p> <pre>.bool (%ContainerClassifier!=null)</pre> |
| <p>The <code>%QualifiedCode%</code> template calls the <code>%isInner%</code> template to test if the class is an inner class:</p> <pre>.if (%isInner%) %ContainerClassifier.QualifiedCode::%Code% .else %Code% .endif</pre> |
| <p>Result:</p> <pre>C2::C1</pre> |
| <p>The <code>%QualifiedCode%</code> template is applied to the <code>C1</code> class, which is an inner class to <code>C2</code>.</p> |

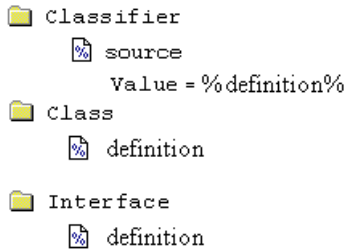
Inheritance and Polymorphism

Templates are defined on a particular metaclass in a language definition file or extension and are inherited by and available to the children of the metaclass. For example, a template defined

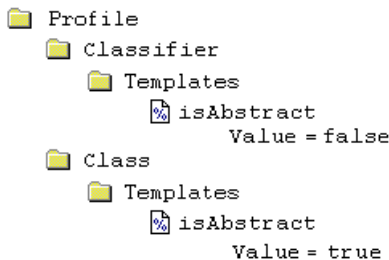
on the Classifier metaclass is available to templates or generated files defined on the Class and Interface metaclasses.

GTL supports the following OO concepts as part of inheritance:

- Polymorphism* - The choice of the template to be evaluated is made at translation-time. A template defined on a classifier can access templates defined on its children (class, interface). In the following example, the content of %definition% depends on whether a class or an interface is being processed:



- Template overriding* - A template defined on a given metaclass can be overridden by a template of the same name defined on a child class. In the following example the template defined on the Classifier metaclass is overridden by the one defined on the Class metaclass:



You can view the overridden parent by right-clicking the child template and selecting **Go to Super-Definition**. You can specify the use of the parent template by prefixing the template call with the :: qualifying operator. For example:

```
%Classifier::isAbstract%
```

- Template overloading* - You can overload your template definitions and test for different conditions. Templates can also be defined under criteria (see *Criteria (Profile)* on page 40) or stereotypes (see *Stereotypes (Profile)* on page 37), 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>
  
```

You can define the same template multiple times in the hierarchy of a language definition file and extensions files, and PowerDesigner will resolve it using inheritance rules. For example, the `myLang` OOM language definition file and the `myExtension` extension file each contain a template `%t%` defined on each of the `Classifier` and `Class` metaclasses:

| myLang Language Definition File | myExtension Extension File |
|---|--|
| <ul style="list-style-type: none"> • Classifier: <ul style="list-style-type: none"> • <code>myFile</code> generated file • <code>%t%</code> template • Class: <ul style="list-style-type: none"> • <code>%t%</code> template | <ul style="list-style-type: none"> • Classifier: <ul style="list-style-type: none"> • <code>myOtherFile</code> generated file • <code>%t%</code> template • Class: <ul style="list-style-type: none"> • <code>%t%</code> template |

The `Class` and `Interface` metaclasses both inherit from the `Classifier` metaclass, and each will generate a `myFile` and a `myOtherFile`.

The following template calls are possible in `myLang/Classifier/myFile` (which cannot access the templates in `myExtension`):

| Template Call in myFile | Template Called |
|---|----------------------------------|
| <code>%t%</code> or <code>%myLang::t%</code> | <code>myLang/Class/t</code> |
| <code>%Classifier::t%</code> or <code>%myLang::Classifier::t%</code> | <code>myLang/Classifier/t</code> |

The following template calls are possible in `myExtension/Classifier/myOtherFile` (which can access both its own templates and those in `myLang`):

| Template Call in myOtherFile | Template Called |
|--|---------------------------------------|
| <code>%t%</code> or <code>%myExtension::t%</code> | <code>myExtension/Class/t</code> |
| <code>%Classifier::t%</code> or <code>%myExtension::Classifier::t%</code> | <code>myExtension/Classifier/t</code> |
| <code>%myLang::t%</code> or <code>%myLang::Class::t%</code> | <code>myLang/Class/t</code> |
| <code>%myLang::Classifier::t%</code> | <code>myLang/Classifier/t</code> |

Note: For an extension file to reach templates defined in a language definition file, the **Complement language generation** property in the extension must be selected (see *Extension File Properties* on page 14).

Passing Parameters to a Template

You can pass parameters to a template, using the syntax: `%t (p1, p2 . . .) %`.

Parameter values cannot contain any `%` characters (you cannot pass a template), and are separated by commas. They are retrieved in the template using local variables with the names `@1, @2, . . .`.

Examples

The following template call:

```
%myTemplate (fine, sunny, 24, 12) %
```

calls `%myTemplate%`:

```
The weather today is %@1% and %@2%, with a high of %@3% and a low of %@4%.
```

Result:

```
The weather today is fine and sunny, with a high of 24 and a low of 12.
```


Examples

The template %Attributes% is defined as follows:

```
.foreach_item(Attributes)
  .if (%Visibility% == %@1%)
    %DataType% %Code%
  .endif
.next(\n)
```

The template %AttributeList% calls %Attributes% three times, passing a different visibility value each time to loop over only the attributes that have this visibility:

```
Class "%Code%" attributes:
// Public
%attributes(+)%

// Protected
%attributes(#)%

// Private
%attributes(-)%
```

Result:

```
Class "C1" attributes :
// Public
  int height
  int width

// Protected
  int shape

// Private
  int cost
  int price
```

Recursive Templates

A template can call itself, but such a template should contain some kind of criteria or scope change to avoid an infinite loop.

| Example |
|--|
| <p>Class C1 is inner to class C2, which is in turn inner to C3. The template %topContainerCode% tests whether the present classifier is inner to another, and if so, calls itself on the container classifier to perform the same test until it reaches a classifier that is not inner, at which point it prints the code of the top container:</p> <pre>.if (%isInner%) %ContainerClassifier.topContainerCode% .else %Code% .endif</pre> <p>Result:</p> <p>C3</p> |

GTL-Specific Metamodel Extensions

A number of calculated attributes and collections are provided as GTL-specific extensions to the metamodel.

The following calculated attributes are metamodel extensions specific to GTL:

| Metaclass | GTL-Specific Attributes |
|---------------------|---|
| PdCommon.BaseObject | <ul style="list-style-type: none"> isSelected (boolean) - True if the object is part of the selection in the generation dialog isShortcut (boolean) - True if the object was accessed by dereferencing a shortcut |
| PdCommon.BaseModel | <ul style="list-style-type: none"> GenOptions (struct) - Gives access to user-defined generation options |
| PdOOM.* | <ul style="list-style-type: none"> ActualComment (string) - Cleaned-up comment (with /**, /*, */ and // removed) |
| PdOOM.Association | <ul style="list-style-type: none"> RoleAMinMultiplicity (string) RoleAMaxMultiplicity (string) RoleBMinMultiplicity (string) RoleBMaxMultiplicity (string) |

| Metaclass | GTL-Specific Attributes |
|-----------------|--|
| PdOOM.Attribute | <ul style="list-style-type: none"> MinMultiplicity (string) MaxMultiplicity (string) Overridden (boolean) DataTypeModifierPrefix (string) DataTypeModifierSuffix (string) @<tag> [Java-specific] (string) - Javadoc@<tag> extended attribute with additional formatting |
| PdOOM.Class | <ul style="list-style-type: none"> MinCardinality (string) MaxCardinality (string) SimpleTypeAttribute [XML-specific] @<tag> [Java-specific] (string) - Javadoc@<tag> extended attribute with additional formatting |
| PdOOM.Interface | <ul style="list-style-type: none"> @<tag> [Java-specific] (string) - Javadoc@<tag> extended attribute with additional formatting |
| PdOOM.Operation | <ul style="list-style-type: none"> 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 | <ul style="list-style-type: none"> DataTypeModifierPrefix (string) DataTypeModifierSuffix (string) |

The following calculated collections are metamodel extensions specific to 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.BaseClassifier-Mapping | SourceLinks |
| PdCommon.BaseAssociation-Mapping | SourceLinks |

GTL Macro Reference

GTL supports macros to express template logic, and to loop on object collections. Macro keywords are prefixed by a . (dot) character, which must be the first non-blank character in the line, and you must respect the use of line breaks in the macro syntax.

Note: Macro parameters can be delimited by double quotes, and this is required if the parameter value includes commas, braces, leading or trailing blanks. The escape sequence for double quotes inside a parameter value is `\`". When the macro parameters specify that a parameter is of type *simple template*, this means that it can contain text, variables, and conditional blocks, but no macros. Parameters of type *complex template* can additionally include macros.

The following macros are available:

- *Conditional and loop / iterative macros:*
 - *.if Macro* on page 275 - evaluates conditions.
 - *.foreach_item Macro* on page 271 – iterates on object collections.
 - *.foreach_line Macro* on page 273 – iterates on lines of a multi-line text block.
 - *.foreach_part Macro* on page 274 – iterates on parts of a string.
 - *.break Macro* on page 266 – breaks a loop.
- *Formatting and string manipulation macros:*
 - *.lowercase and .uppercase Macros* on page 277 - change the case of a text block.
 - *.convert_name and .convert_code Macros* on page 267 - convert codes into names or names into codes.
 - *.delete and .replace Macros* on page 268 - perform operations on substrings.
 - *.unique Macro* on page 281 - filters redundant lines from a text block.
 - *.block Macro* on page 265 - adds a header and a footer to a text block.
- *Generation command macros* - for use when writing GTL in the context of the execution of a generation command:
 - *.vbscript Macro* on page 281 - embed VB script code inside a template.
 - *.execute_vbscript Macro* on page 270 - launch vbscripts.
 - *.execute_command Macro* on page 269 - launch executables.
 - *.abort_command Macro* on page 265 - stop command execution.
 - *.change_dir and .create_path Macros* on page 266 - change directory or create a path.
 - *.log Macro* on page 277 - write log messages.
- *Miscellaneous macros:*
 - *.set_object, .set_value, and .unset Macros* on page 279 - create local objects or variables.
 - *.comment and .// Macro* on page 267 - inserts a comment in a template.

- *.object and .collection Macros* on page 278 - returns a collection of objects based on the specified scope and condition.
- *.object and .collection Macros* on page 278 - return an object or collection based on the specified scope and condition.
- *.bool Macro* on page 266 - evaluates a condition.
- *.set_interactive_mode Macro* on page 278 – defines whether the GTL execution must interact with the user.
- *.error and .warning Macros* on page 269

.abort_command Macro

This macro stops a generation command.

Example

```
.if %_JAVAC%
  .execute_command (%_JAVAC%, %FileName%)
.else
  .abort_command
.endif
```

For information about generation commands, see *Generation Category* on page 112.

.block Macro

This macro wraps a block of output with a header and/or a footer, if the output is not empty.

```
.block [ (head) ]
      block-input
.endblock [ (tail) ]
```

The following parameters are available:

| Parameter | Description |
|--------------------|--|
| head | [optional] Generated only if block-input is not empty. Type: Simple template |
| block-input | Specifies the text to output between the head and tail. Type: Complex template |
| tail | [optional] Generated only if block-input is not empty. Type: Simple template |

| Example | Result |
|---|---|
| <pre>.block (%Comment% .endblock ()</pre> | <pre>My comment is in bold!</pre> <p>Note: The tags would not be generated if no comment were entered for a particular object.</p> |

.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 | Specifies the condition to be evaluated. Type: Condition |

| Example | Result |
|------------------------------------|-----------------|
| <pre>.bool (%.3:Code%= =ejb)</pre> | <pre>true</pre> |

.break Macro

This macro can be used to break out of .foreach loops.

| Example |
|--|
| <pre>.set_value(_hasMain, false, new) .foreach_item(Operations) .if (%Code% == main) .set_value(_hasMain, true) .break .endif .next %_hasMain%</pre> |

.change_dir and .create_path Macros

These macros change the current directory or create the specified path as part of a generation command.

```
.change_dir (path)
```

```
.create_path (path)
```

The following parameters are available:

| Parameter | Description |
|-------------|--|
| path | Specifies the directory to go to or to create. Type: Simple template (escape sequences ignored) |

| Example | Result |
|--|--|
| <code>.change_dir(C:\temp)</code> | Changes the path to write to to C:\temp. |
| <code>.create_path(C:\temp\mydir)</code> | Creates the new directory C:\temp\mydir. |

For information about generation commands, see *Generation Category* on page 112.

.comment and .// Macro

These macros are used to insert comments in a template. Lines starting with `.//` or `.comment` are ignored during generation.

| Example |
|--|
| <code>.// This is a comment</code> <code>.comment This is also a comment</code> |

.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" [, "delimiters" ], case )
```

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

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

The following parameters are available:

| Parameter | Description |
|-------------------|--|
| expression | Specifies the text to be converted. For <code>.convert_name</code> , this is generally the <code>%Name</code> <code>%</code> variable and may include a suffix or prefix. Type: Simple template |
| separator | [optional] Character generated each time a separator declared in <code>delimiters</code> is found in the code. For example, <code>"_"</code> (underscore). Type: Text |

| Parameter | Description |
|-------------------|--|
| delimiters | [optional] Specifies the different delimiters likely to exist in the input code or name, and which will be replaced by separator . You can declare several separators, for example "_" and "-" Type: Text |
| case | [optional for <code>.convert_name</code> only] Specifies the case into which to convert the code. You can choose between: <ul style="list-style-type: none"> <code>firstLowerWord</code> - First word in lowercase, first letters of subsequent words in uppercase <code>FirstUpperChar</code> - First character of all words in uppercase <code>lower_case</code> - All words in lowercase and separated by an underscore <code>UPPER_CASE</code> - All words in uppercase and separated by an underscore |

.delete and .replace Macros

These macros delete or replace all instances of the given string in the text input.

```
.delete (string)
    block-input
.enddelete
```

```
.replace (string, new-string)
    block-input
.endreplace
```

The following parameters are available:

| Parameter | Description |
|--------------------|--|
| string | Specifies the string to be deleted. Type: Text |
| new-string | [.replace only] Specifies the string with which to replace string . Type: Text |
| block-input | Specifies the text to be parsed for instances of the string to delete or replace. Type: Complex template |

| Examples | Result |
|---|--------------|
| <code>.delete (Get)</code> <code> GetCustomerName</code> <code>.enddelete</code> | CustomerName |

| Examples | Result |
|---|-----------------|
| <code>.replace(Get, Set) GetCustomerName .endreplace</code> | SetCustomerName |
| <code>.replace(" ", _) Customer Name .endreplace</code> | Customer_Name |

.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.

```
.error message
```

```
.warning message
```

The following parameters are available:

| Parameter | Description |
|----------------|---|
| <i>message</i> | Specifies the text of the message. Type: Simple template |

Example

```
.error no initial value supplied for attribute %Code% of class  
%Parent.Code%
```

.execute_command Macro

This macro is launches executables as part of a generation command. If there is a failure for any reason (executable not found or output sent to `stderr`), then command execution is stopped.

```
.execute_command (cmd [, args [, mode]])
```

The following parameters are available:

| Parameter | Description |
|------------|--|
| cmd | Specifies the path to the executable Type: Simple template (escape sequences ignored) |

| Parameter | Description |
|-------------|---|
| args | [optional] Specifies arguments for the executable. Type: Simple template (escape sequences ignored) |
| mode | [optional] Specifies the execution mode. You can choose from: <ul style="list-style-type: none"> cmd_ShellExecute - runs as an independent process cmd_PipeOutput - blocks until completion, and shows the executable output in the output window |

| Example |
|---|
| <code>.execute_command(notepad, file1.txt, cmd_ShellExecute)</code> |

For information about generation commands, see *Generation Category* on page 112.

.execute_vbscript Macro

This macro is used to execute a VB script specified in a separate file as part of a generation command.

```
.execute_vbscript (vbs-file [, script-parameter])
```

The following parameters are available:

| Parameter | Description |
|-------------------------|---|
| vbs-file | Specifies the path to the VB script. Type: Simple template (escape sequences ignored) |
| script-parameter | [optional] Passed to the script through the ScriptInputParameters global property. Type: Simple template |

| Example |
|---|
| <code>.execute_vbscript(C:\samples\vbs\login.vbs, %username%)</code> |
| The result of the script is available in the ScriptResult global property (see <i>Manipulating Models, Collections, and Objects (Scripting)</i> on page 312). The active object of the current translation scope can be accessed through the ActiveSelection collection as ActiveSelection.Item(0). |

For information about generation commands, see *Generation Category* on page 112.

.foreach_item Macro

This macro iterates over a collection of sub-objects or related objects.

```
.foreach_item (collection [, head [, tail [, filter [, order]]]])
  output
.next [(separator)]
```

The following parameters are available:

| Parameter | Description |
|-------------------|---|
| collection | Specifies the collection over which to iterate. Type: Simple template |
| head | [optional] Specifies text to be generated before the output, unless the collection is empty. Type: Text |
| tail | [optional] Specifies text to be generated after the output, unless the collection is empty. Type: Text |
| filter | [optional] Specifies a filter to apply to the collection before iteration. Type: Simple condition |
| order | [optional] Specifies the order in which the collection will be iterated in the format: %Item1. property % <= %Item2. property % When the comparison evaluates to true, %Item1% will be placed after %Item2%. By default, the collection is ordered alphabetically by name. Type: Simple condition |
| output | Specifies the text to output for each item in the collection. Type: Complex template |
| separator | [optional] Specifies text to be generated between each instance of output . Type: Text |

Note: If parameter values contain commas, braces, or leading or trailing blanks, they must be delimited with double-quotes. To escape double-quotes inside a parameter value, use \".

Examples

Simple list:

```
.foreach_item(Attributes)
  *%Code% (%DataType%) [ = %InitialValue%];
.next(\n)
```

Result:

```
*available (boolean) = true;
*actualCost (int);
*baseCost (int);
*color (String);
*height (int) = 10;
*width (int) = 5;
*name (int);
```

With head and tail:

```
.foreach_item(Attributes,Attributes:\n,\n\nEnd of Attribute List)
  *%Code% (%DataType%) [ = %InitialValue%];
.next(\n)
```

Result:

```
Attributes:
*available (boolean) = true;
*actualCost (int);
*baseCost (int);
*color (String);
*height (int) = 10;
*width (int) = 5;
*name (int);
```

End of Attribute List

With filter:

```
.foreach_item(Attributes,,,%1:Code%==a)
  *%Code% (%DataType%) [ = %InitialValue%];
.next(\n)
```

Result:

```
*available (boolean) = true;
*actualCost (int);
```

Examples

With reverse alphabetical ordering:

```
.foreach_item(Attributes,,,, %Item1.Code% <= %Item2.Code% )
  *%Code% (%DataType%) [ = %InitialValue%];
.next(\n)
```

Result:

```
*width (int) = 5;
*name (int);
*height (int) = 10;
*color (String);
*baseCost (int);
*available (boolean) = true;
*actualCost (int);
```

.foreach_line Macro

This macro iterates over the lines of the multiline block of text using the special `%CurrentLine%` local variable.

```
.foreach_line (input [, head [, tail]])
  output
.next [(separator)]
```

The following parameters are available:

| Parameter | Description |
|------------------|---|
| input | Specifies the text over which to iterate. Type: Simple template |
| head | [optional] Specifies text to be generated before the output, unless there is no output. Type: Text |
| tail | [optional] Specifies text to be generated after the output, unless there is no output. Type: Text |
| output | Specifies the text to output for each line in the input. Type: Complex template |
| separator | [optional] Specifies text to be generated between each line of output . Type: Text |

| Example |
|--|
| <pre>.foreach_line(%Comment%, "**\n", "\n*") * %CurrentLine% .next("\n")</pre> |
| <p>Result:</p> <pre>/** * This is my comment. * It is a Java style documentation comment. * It spans several lines. */</pre> |

.foreach_part Macro

This macro iterates over the parts of a string divided by a delimiter using the special `%CurrentPart%` local variable.

```
.foreach_part (input [,"delimiter" [, head [, tail ]]])
    output
.next[(separator)]
```

The following parameters are available:

| Parameter | Description |
|------------------|--|
| input | Specifies the text over which to iterate. Type: Simple template |
| delimiter | Specifies the sub-string that divides the input into parts. You can specify multiple characters including ranges. For example <code>[A-Z]</code> specifies that any capital letter acts as a delimiter. By default, the delimiter is set to <code>' - _ , \t '</code> (space, dash, underscore, comma, or tab). Note: The delimiter must be surrounded by single quotes if it contains a space. Type: Text |
| head | [optional] Specifies text to be generated before the output, unless there is no output. Type: Text |
| tail | [optional] Specifies text to be generated after the output, unless there is no output. Type: Text |
| output | Specifies the text to output for each part in the input. Type: Complex template |

| Parameter | Description |
|------------------|--|
| separator | [optional] Specifies text to be generated between each part of output . Type: Text |

For example:

| Examples |
|--|
| <p>This template is applied to My class:</p> <pre>.foreach_part (%Name%) %.FU:CurrentPart% .next</pre> <p>Result:</p> <pre>MyClass</pre> |
| <p>This template is applied to My class:</p> <pre>.foreach_part (%Name%, ' -_', tbl_) %.L:CurrentPart% .next(_)</pre> <p>Result:</p> <pre>tbl_my_class</pre> |
| <p>This template is applied to MyClass:</p> <pre>.foreach_part (%Name%, [A-Z]) %.L:CurrentPart% .next(-)</pre> <p>Result:</p> <pre>my-class</pre> |

.if Macro

This macro is used for conditional generation.

```
.if[not] condition
  output
  [ (.elseif[not] condition
    output)* ]
  [.else
    output]
.endif [ (tail) ]
```

The following parameters are available:

| Parameter | Description |
|------------------|---|
| condition | <p>Specifies the condition to evaluate, in the form: variable [operator comparison]</p> <p>Where <i>comparison</i> may be :</p> <ul style="list-style-type: none"> • Text, or a simple template • true or false • null or notnull <p>If no operator and condition are specified, the condition evaluates to true unless the value of the variable is false, null, or the empty string.</p> <p>If variable and comparison are not integers, the operators perform a string comparison that takes into account embedded numbers. For example: Class_10 > Class_2</p> <p>You can chain conditions together using the <code>and</code> or <code>or</code> logical operators.</p> <p>Type: Simple template</p> |
| output | <p>Specifies the output if the condition is true.</p> <p>Type: Complex template</p> |
| tail | <p>[optional] Specifies text to be generated after the output, unless the output is empty.</p> <p>Type: Text</p> |

| Examples |
|--|
| <p>Simple <code>.if</code> block:</p> <pre>.if %Abstract% This class is abstract. .endif</pre> <p>Result (if the Abstract property is selected):</p> <pre>This class is abstract.</pre> |
| <p>With two conditions and an <code>.else</code> clause:</p> <pre>.if (%Abstract%==false) && (%Visibility%=="") This class is public and concrete. .else This is not a public, concrete class. .endif</pre> <p>Result (if the Abstract property is not selected and the Visibility property is set to Public):</p> <pre>This class is public and concrete.</pre> |

Examples

With an `.elseif` clause:

```
.if (%Abstract%==false) && (%Visibility%=="")
    This class is public and concrete.
.elseif (%Visibility%=="")
    This class is public.
.else
    This is not a public, concrete class.
.endif
```

.log Macro

This macro logs a message to the **Output** window **Generation** tab as part of a generation command.

```
.log message
```

Example

```
.log undefined environment variable: JAVAC
```

For information about generation commands, see *Generation Category* on page 112.

.lowercase and .uppercase Macros

These macros convert text blocks to the specified case.

```
.lowercase
    block-input
.endlowercase
```

```
.uppercase
    block-input
.enduppercase
```

The following parameters are available:

| Parameter | Description |
|--------------------|--|
| <i>block-input</i> | Specifies the text to convert. Type: Complex template |

| Example | Result |
|---|--|
| <pre>.lowercase %Comment% .endlowercase</pre> | <p>Applied to</p> <pre>This is my comment.</pre> <p>Produces:</p> <pre>this is my comment.</pre> |

.object and .collection Macros

These macro return a single object OID or a collection of objects as a concatenation of semi-colon terminated OIDs, and are generally used to create templates returning objects for use by other templates.

```
.collection (scope [, filter])
```

```
.object (scope [, filter])
```

The following parameters are available:

| Parameter | Description |
|---------------|---|
| scope | Specifies the collection over which to iterate. Type: simple-template returning a collection scope |
| filter | [optional] Specifies a filter condition to filter the collection. Type: simple-template |

| Examples |
|---|
| <pre>.object(Attributes, (%.1:Code%>= a) and (%.1:Code% <= e))</pre> <p>Result:</p> <pre>C73C03B7-CD73-466A-B323-0B90B67E82FC</pre> |
| <pre>.collection(Attributes, (%.1:Code%>= a) and (%.1:Code% <= e))</pre> <p>Result:</p> <pre>C73C03B7-CD73-466A-B323-0B90B67E82FC;77E3F55C- CF24-440F-84E7-5AA7B3399C00;F369CD8C-0C16-4896-9C2D-0CD2F80D6980;0 0ADD959-0705-4061-BF77-BB1914EDC018;</pre> |

.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 parameters are available:

| Parameter | Description |
|-------------|--|
| mode | <p>Specifies the level of interaction required. You can choose between:</p> <ul style="list-style-type: none"> <code>im_Batch</code> - Suppresses dialog boxes and always uses default values. For example, if your model contains external shortcuts and the target model for the shortcuts is closed, this mode will automatically open the model without user interaction. <code>im_Dialog</code> - Displays information and confirmation dialog boxes that require user interaction for the execution to keep running. <code>im_Abort</code> - Suppresses dialog boxes and aborts execution if a dialog is encountered. |

.set_object, .set_value, and .unset Macros

These macros are used to define a local variable of object (local object) or value type or to unset them.

Use the following syntax to create a local object:

```
.set_object ( [scope.] name [, object-ref [, mode]])
```

Use the following syntax to create a local variable:

```
.set_value ( [scope.] name, value [, mode])
```

Use the following syntax to remove a local object or variable:

```
.unset ( [scope.] name)
```

The following parameters are available:

| Parameter | Description |
|-------------------|--|
| scope | <p>[optional] Specifies the qualifying scope. If no scope is set, then the scope is the object with the current scope. Use the <code>this</code> keyword to explicitly give a scope of the current object, or <code>Parent</code> to give a scope of the parent object.</p> <p>Type: Simple-template returning an object or a collection scope</p> |
| name | <p>Specifies the name of the object or variable, which you can reference elsewhere in the template in the form of <code>%name%</code>.</p> <p>Type: Simple-template</p> |
| object-ref | <p>[.set_object only - optional] Specifies an object reference. If no reference is specified or an empty string is given, the variable is a reference to the active object in the current translation scope.</p> <p>Type: <code>[scope.]object-scope</code></p> |

| Parameter | Description |
|--------------|--|
| value | [.set_value only] Specifies the value to give to the variable. Type: Simple template (escape sequences ignored) |
| mode | [optional] Specifies the mode of creation. You can choose between: <ul style="list-style-type: none"> new - Forces the (re)-definition of the variable in the current scope. Recommended when a variable with the same name may already be defined in a previous 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. |

Examples:

| Examples |
|--|
| <pre>.set_object(Attribut1, Attributes.First) .set_value(FirstAttributeCode, %Attributes.First.Code%) %FirstAttributeCode% (OID: %Attribut1%)</pre> <p>Result:</p> <pre>a1 (OID: 63442F85-48DF-42C8-92C1-0591F5D34525)</pre> |
| <pre>.set_value(this.key, %Code%-%ObjectID%)</pre> <p>Result:</p> <pre>C1-40D8F396-EE29-4B7B-8C78-E5A0C5A23325</pre> |
| <pre>.set_value(i, 1, new) %i?% .unset(i) %i?%</pre> <p>Result:</p> <pre>true false</pre> <p>The first call to %i?% outputs true as the variable i is defined, and the second outputs false, because it has been unset.</p> |

Note: You can use the dereferencing operator, * (see *GTL Operators* on page 252), to convert the value of a variable set with the .set_value macro to a template name. For example, the following code is equivalent to %Code%.:

```
.set_value(i, Code)
%*i%
```

.unique Macro

This macro outputs a block in which each line of the text generated is unique, and is often used for calculating imports, includes, typedefs, or forward declarations in languages such as Java, C++ or C#.

```
.unique
  block-input
.endunique [ (tail) ]
```

The following parameters are available:

| Parameter | Description |
|--------------------|---|
| block-input | Specifies the text block to be processed. Type: Complex template |
| tail | [optional] Specifies text to be generated after the output, unless the collection is empty. Type: Text |

Example

```
.unique
  import java.util.*;
  import java.lang.String;
  %imports%
.endunique
```

.vbscript Macro

This macro is used to embed VBScript code inside a template as part of a generation command. The result of the script is available as the `ScriptResult` array

```
.vbscript [ (script-param-list) ]
  block-input
.endvbscript [ (tail) ]
```

The following parameters are available:

| Parameter | Description |
|--------------------------|--|
| script-param-list | Specifies the parameters to pass to the script through the <code>ScriptInputArray</code> table. Type: List of simple-template arguments separated by commas |
| block-input | Specifies the VBScript to run. Type: Text |

| Parameter | Description |
|-------------|---|
| tail | Appended to the output, if there is one Type: Text |

Examples

This simple script accepts the two words `hello` and `world` as input parameters, and returns them as a single string with a space in between them:

```
.vbscript(hello, world)
ScriptResult = ScriptInputArray(0) + " " + ScriptInputArray(1)
.endvbscript
```

Result:

```
hello world
```

Examples

This script accepts an attribute code, reviews it against all the attribute codes in the current model, and appends a 1 to it if it matches any other code:

```
.set_value(_code,%@1%,new)
.vbscript(%_code%)
  Dim attrCode
  attrCode = ScriptInputArray(0)

  While (attrFound(attrCode))
    attrCode = attrCode + "1"
  Wend

  Function attrFound(attrCode)
    Dim found, attr
    found = False
    For Each attr in ActiveSelection.Item(0).Attributes
      If attr.Code = attrCode Then
        found = True
        Exit For
      End If
    Next

    For Each attr in ActiveSelection.Item(0).InheritedAttributes
      If attr.Code = attrCode Then
        found = True
        Exit For
      End If
    Next
    attrFound = found
  End Function

  ScriptResult = attrCode
.endvbscript
```

Note: The active object of the current translation scope is accessed as `ActiveSelection.Item(0)` (see *Manipulating Models, Collections, and Objects (Scripting)* on page 312).

For information about generation commands, see *Generation Category* on page 112.

GTL Syntax and Translation Errors

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
```

CHAPTER 5: Customizing Generation with GTL

You may encounter the following syntax errors:

| Syntax error message | Description and correction |
|---|--|
| condition parsing error | Syntax error in a boolean expression |
| expecting .endif .else with no matching .if .endif with no matching .if | Add an .endif or .if (see <i>.if Macro</i> on page 275). |
| expecting .next .next with no matching .foreach | Add an appropriate .next or .foreach to the collection block (for example, see <i>.foreach_item Macro</i> on page 271). |
| expecting .end%s | Add an appropriate .end to the macro block (for example, see <i>.unique Macro</i> on page 281). |
| .end%s with no matching .%s | Add an appropriate <i>.macro</i> to the <i>.endmacro</i> (for example, see <i>.vbscript Macro</i> on page 281). |
| missing or mismatched parentheses | Correct any mismatched parentheses. |
| unexpected parameters: <i>extra-params</i> | Remove any unnecessary parameters |
| unknown macro | Replace with a valid macro (see <i>GTL Macro Reference</i> on page 264). |
| .execute_command incorrect syntax | The correct syntax is displayed in the Preview tab, or in the Output window (see <i>.execute_command Macro</i> on page 269). |
| Change_dir incorrect syntax | See <i>.change_dir and .create_path Macros</i> on page 266. |
| convert_name incorrect syntax convert_code incorrect syntax | See <i>.convert_name and .convert_code Macros</i> on page 267. |
| set_object incorrect syntax set_value incorrect syntax | See <i>.set_object, .set_value, and .unset Macros</i> on page 279. |
| execute_vbscript incorrect syntax | See <i>.execute_vbscript Macro</i> on page 270. |

Translation errors are evaluation errors on a variable when evaluating a template:

| Translation error message | Description and correction |
|--|--|
| unresolved collection: <i>collection</i> | Unknown collection (see <i>Accessing Collections of Sub-Objects or Related Objects</i> on page 247). |

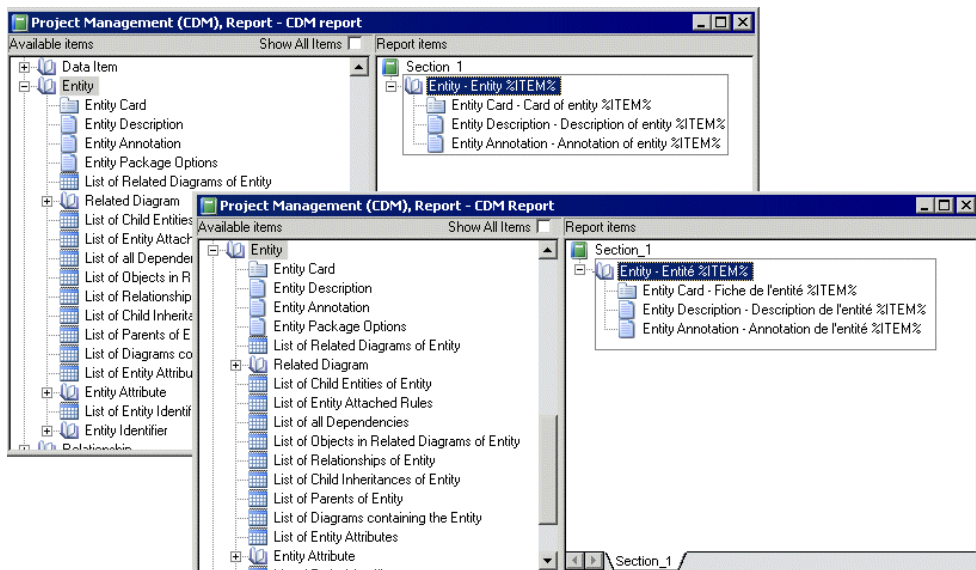
| Translation error message | Description and correction |
|---|---|
| unresolved member: <i>member</i> null object expecting object variable: <i>object</i> | Unknown member, null object member, or expecting a string instead of an object (see <i>Extracting Object Properties</i> on page 246). |
| no outer scope | Invalid use of the <code>Outer</code> keyword (see <i>Translation Scope</i> on page 255). |
| VBScript execution error | VB script error (see <i>.vbscript Macro</i> on page 281). |
| Deadlock detected | Deadlock due to an infinite loop. |

Translating Reports with Report Language Files

When you create a report, you select a report language, which contains all the framing text used in the generation of the report for the selected language, such as report section titles, types of model objects, and their properties. PowerDesigner ships with support for English (default), French, and simplified and traditional Chinese. You can edit these files, or use them as the basis for creating your own files for translations into other languages.

Report language files have an .xrl extension and are stored in `install_dir/Resource Files/Report Languages`. To view the list of report languages, select **Tools > Resources > Report Languages**. For information about the tools available in resource file lists, see *Chapter 1, PowerDesigner Resource Files* on page 1.

In the following example, Entity Card, Entity Description, and Entity Annotation are shown in English and French as they will appear in the Report items pane:



The report language files use GTL templates (see *Chapter 5, Customizing Generation with GTL* on page 245) 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

CHAPTER 6: Translating Reports with Report Language Files

elements in a report and dramatically reduce (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 303) is translated as:

```
Liste des %@Value% %ParentMetaClass.OFTHECLSSNAME% %%PARENT%%
```

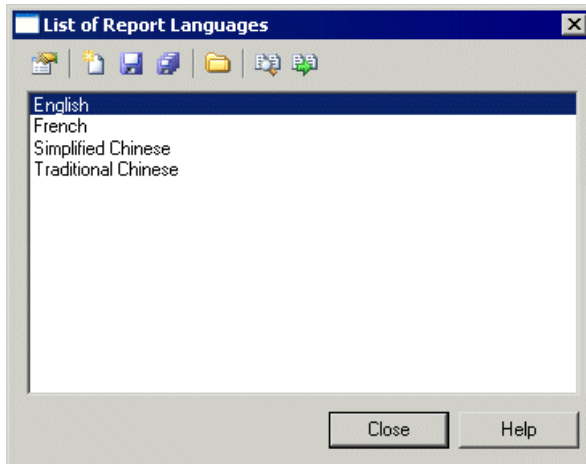
in which the following variables are resolved:

- `%@Value%` - resolves to the object type of the metaclass (see *Object Attributes Category* on page 298), `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 300), `l'entité`.
- `%%PARENT%%` - resolves to the name of the specific object (see *Object Attributes Category* on page 298), `MyEntity`.

Opening a Report Language File

You can review and edit report language 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, PowerDesigner Resource Files* on page 1.

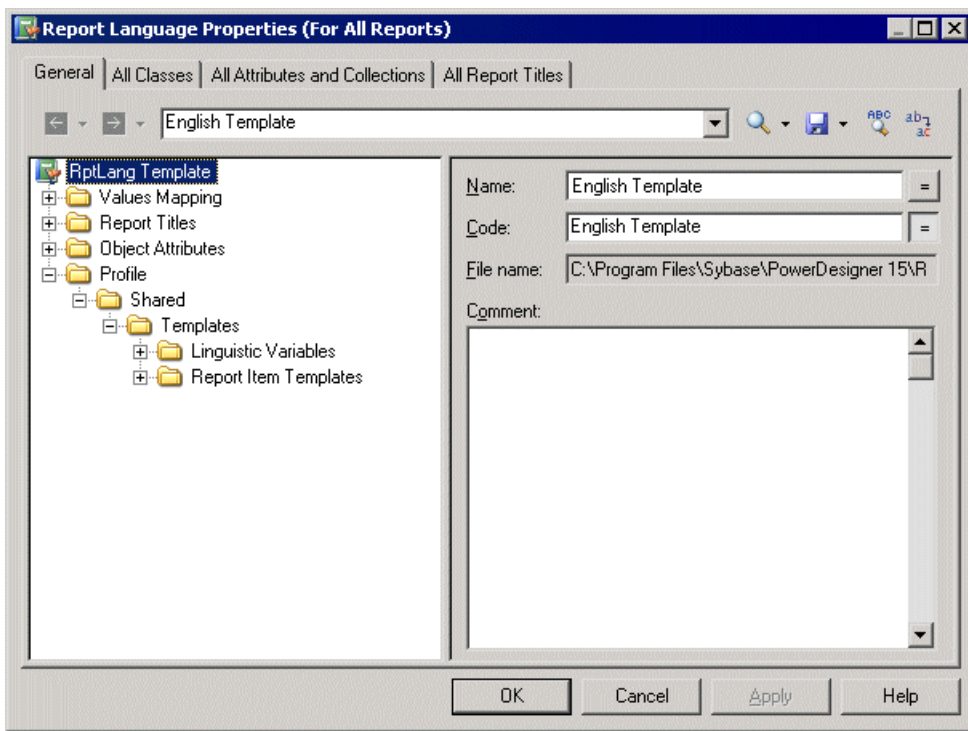
Creating a Report Language File for a New Language

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, 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 (see *Values Mapping Category* on page 291).
6. Open the **Profile > Linguistic Variables** category to create the grammar rules necessary for the correct evaluation of the report item templates (see *Profile/Linguistic Variables Category* on page 300).
7. Open the **Profile > Report Items Templates** category, and translate the various templates (see *Profile/Report Item Templates Category* on page 303). As you translate, you may discover additional linguistic variables that you should create.
8. Click the **All Classes** tab to view a sortable list of all the metaclasses available in the PowerDesigner metamodel (see *All Classes Tab* on page 299). Translate each of the metaclass names.
9. Click the **All Attributes and Collections** tab to view a sortable list of all the attributes and collections available in the PowerDesigner metamodel (see *All Attributes and Collections Tab* on page 300). Translate each of the attribute and collection names.
10. Click the **All Report Titles** tab, and review the automatically generated report titles (see *All Report Titles Tab* on page 297). 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 file is now ready to be attached to a report.

Report Language File Properties

All report language files can be opened in the Resource Editor, and have the same basic category structure.

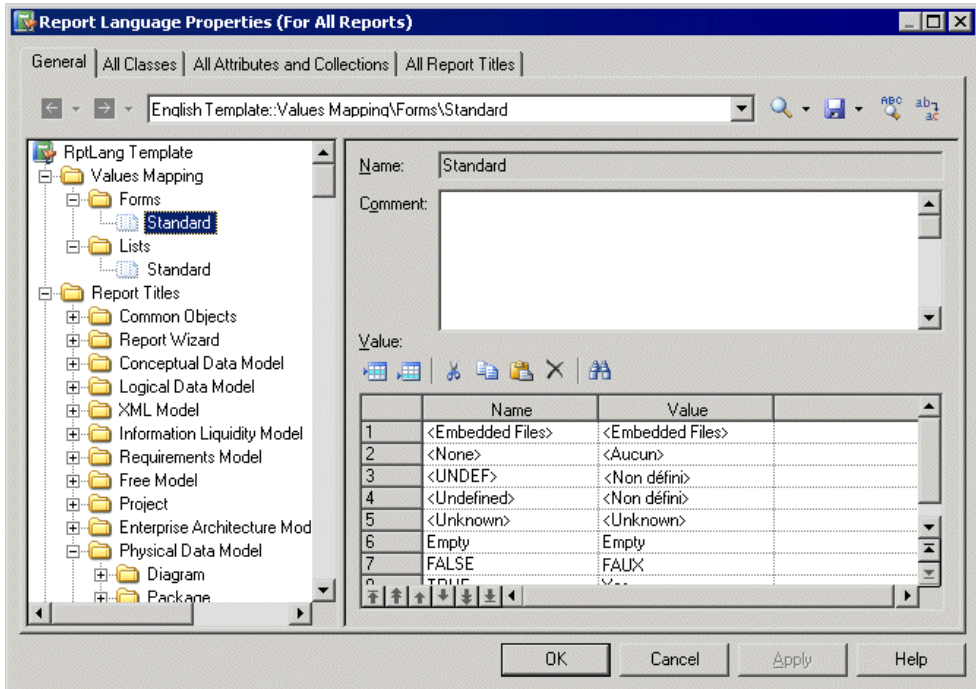


The root node of each file contains the following properties:

| 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:



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.

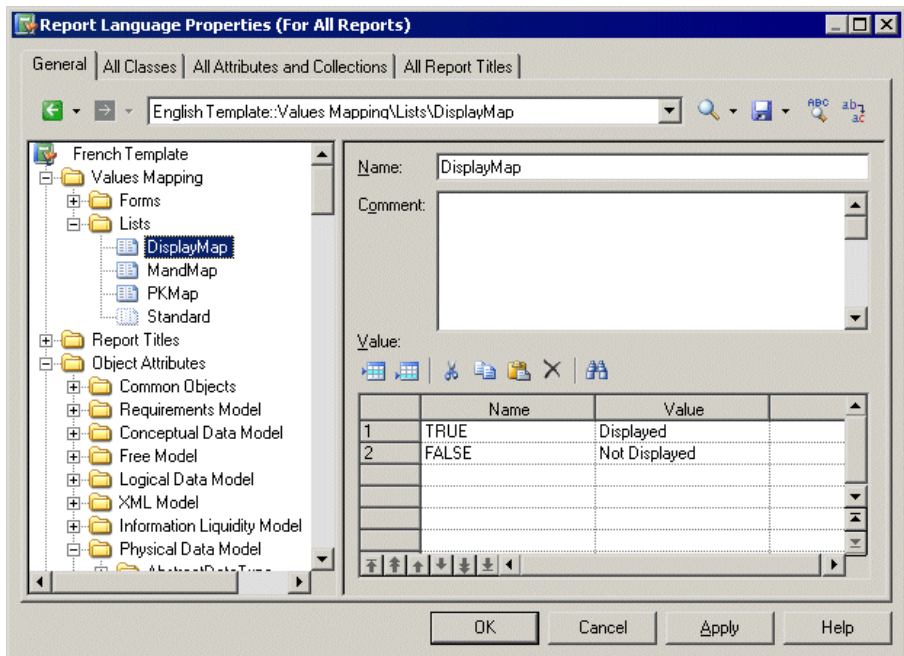
Example: Creating a Mapping Table, and Attaching It to a Specific Model Object

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.

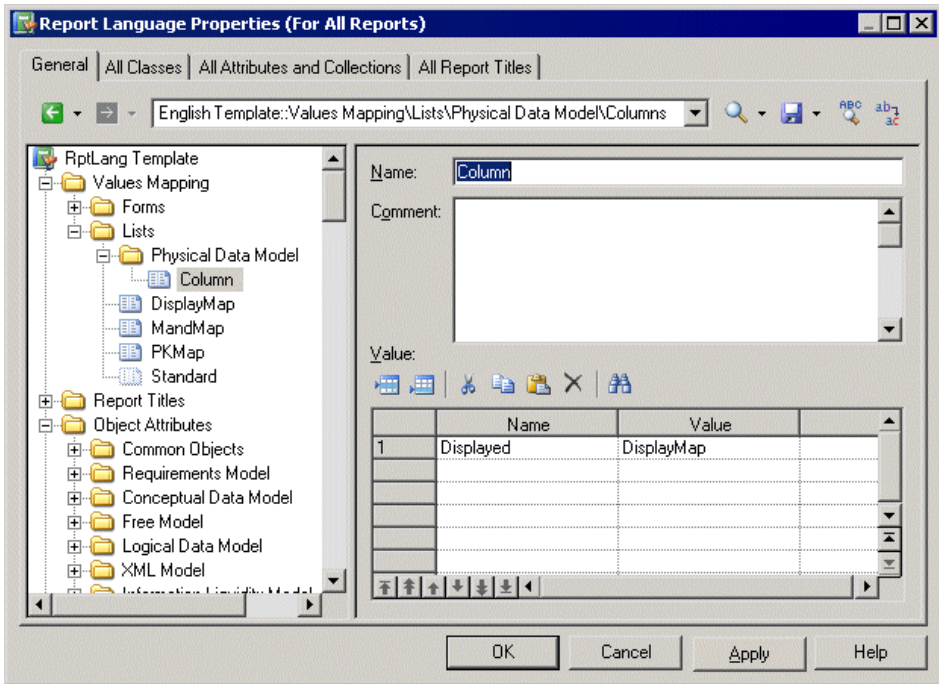
In the following example, the DisplayMap mapping table is used to override the Standard mapping table for PDM columns to provide custom values for the Displayed property, which controls the display of the selected column in the table symbol. This situation can be summarized as follows:

| 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.



4. Right-click the Lists category, select **New > Category**, name the category Physical Data Model, and click Apply.
5. To complete the recreation of the PDM Object Attributes tree, right-click the new Physical Data Model category, select **New > Map Item**, name the category Column, and click Apply.
6. Click the Name column to create a value and enter Displayed, which is the name of the PDM column attribute (property).
7. Click the Value column and enter DisplayMap to specify the mapping table to use for that attribute.



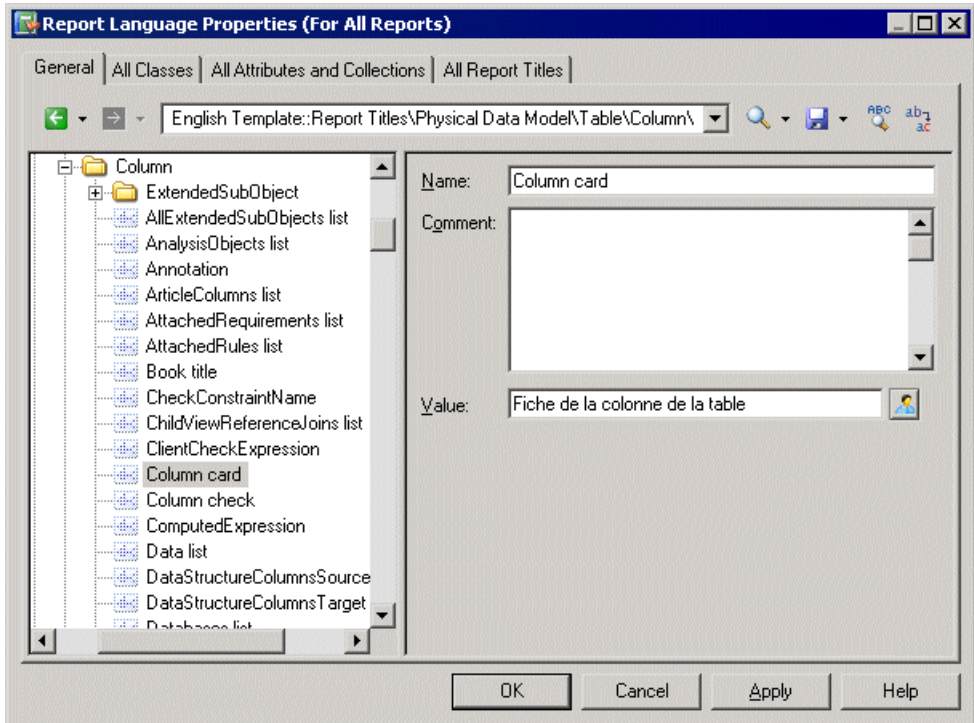
8. Click Apply to save your changes. When you generate a report, the Displayed property will be shown using the specified values:

1 List of table columns

| <i>Name</i> | <i>Code</i> | <i>Displayed</i> |
|-------------|-------------|------------------|
| id | id | Displayed |
| name | name | Not Displayed |
| size | size | Not Displayed |
| supplier | supplier | Not Displayed |
| quantity | quantity | Displayed |
| unit_price | unit_price | Displayed |

Report Titles Category

The Report Titles category contains translations for all the possible report titles that appear in the Available Items pane in the Report Editor, those that are generated with the Report Wizard, and other miscellaneous text items.



This category contains the following sub-categories:

| Sub-category | Description |
|----------------|--|
| Common Objects | Contains the text items available to all models. You must provide translations of these items here. 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] | <p>Contain the report titles and other text items available to each model. These are automatically generated, but you can override the default values.</p> <p>Example: DataTransformationTasks list provides the text for the data transformation tasks list of a given transformation process in the Data Movement Model.</p> |

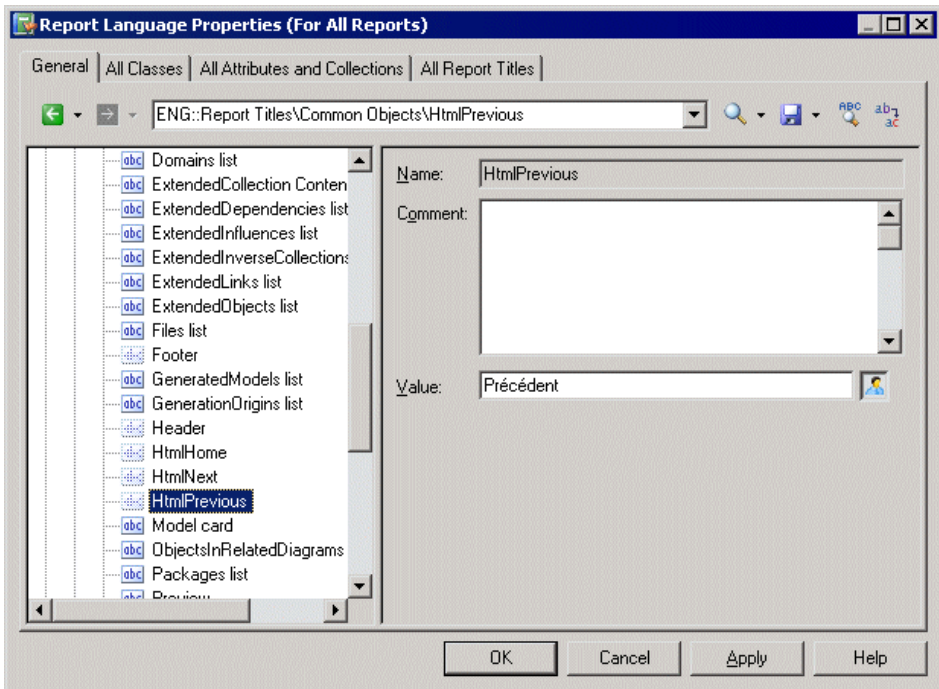
By default (with the exception of the Common Objects and Report Wizard sub-categories) these translations are automatically generated from the templates in the Profile category (see *Profile/Report Item Templates Category* on page 303). You can override the automatically generated values by entering your own text in the **Localized name** field, which will depress the **User-Defined** button to indicate that the value is no longer generated.

Note: The **All Report Titles** tab (see *All Report Titles Tab* on page 297) 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.

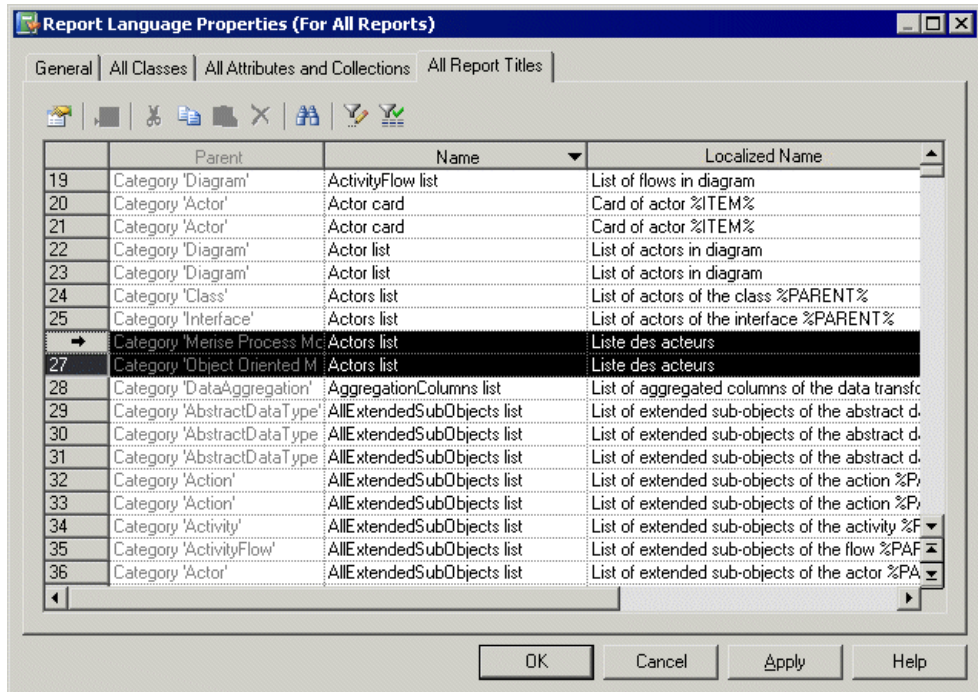
1. Open the **Report Titles > Common Objects** category.
2. Click the `HtmlPrevious` entry to display its properties, and enter a translation in the **Value** box. The **User-Defined** button is depressed to indicate that the value is no longer generated.



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

All Report Titles Tab

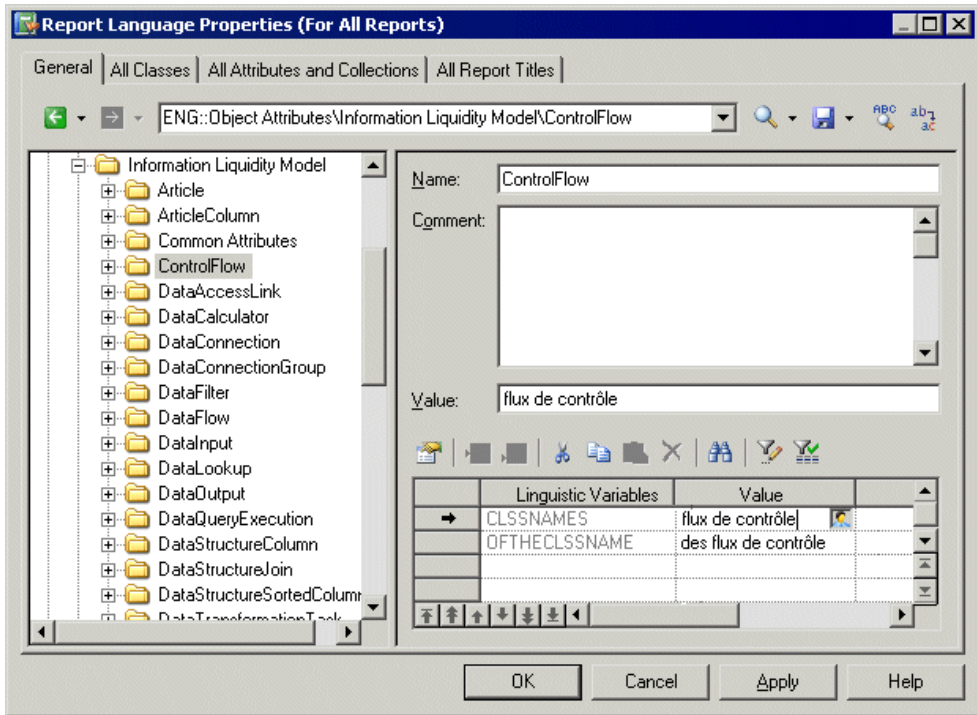
The Report Titles tab lists all the report titles and other miscellaneous text items available in the Report Titles category on the General tab, but the flat structure makes it more convenient to work with.



For 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.

Object Attributes Category

The Object Attributes category contains all the metaclasses, collections and attributes available in the PowerDesigner metamodel, organized in tree form:



This category contains the following sub-categories:

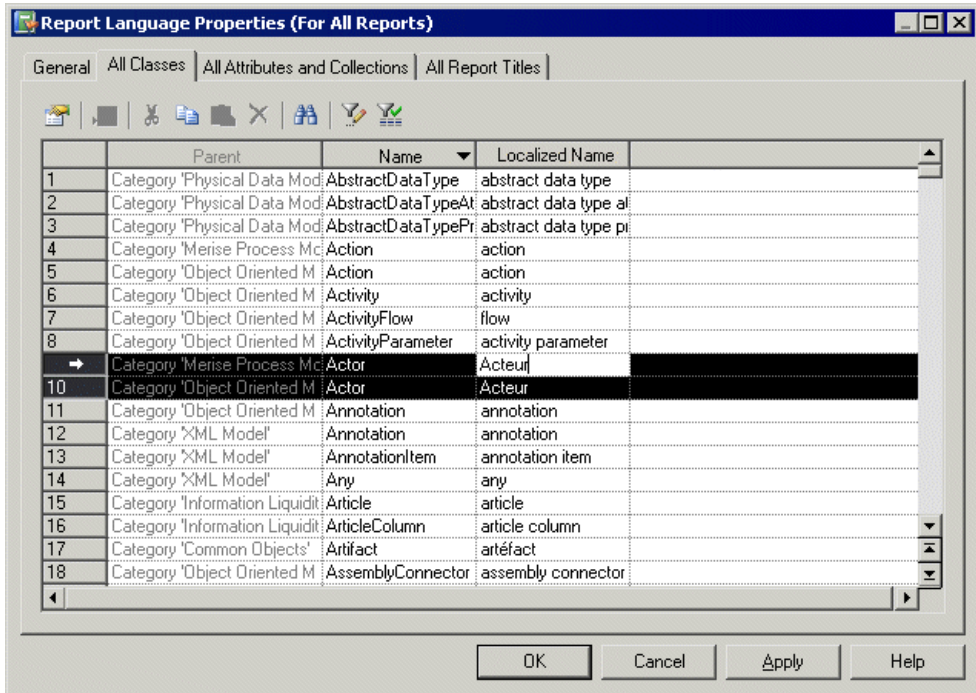
| Sub-category | Description |
|----------------|--|
| [Models] | Contain text items for metaclasses, collections and attributes available to each model, for which you must provide translations. Example: Action provides the text for an attribute of a process in the Business Process Model. |
| Common 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 294).

For metaclasses only, the linguistic variables you have specified (see *Profile/Linguistic Variables Category* on page 300) 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, which will depress the **User-Defined** button to indicate that the value is no longer generated.

All Classes Tab

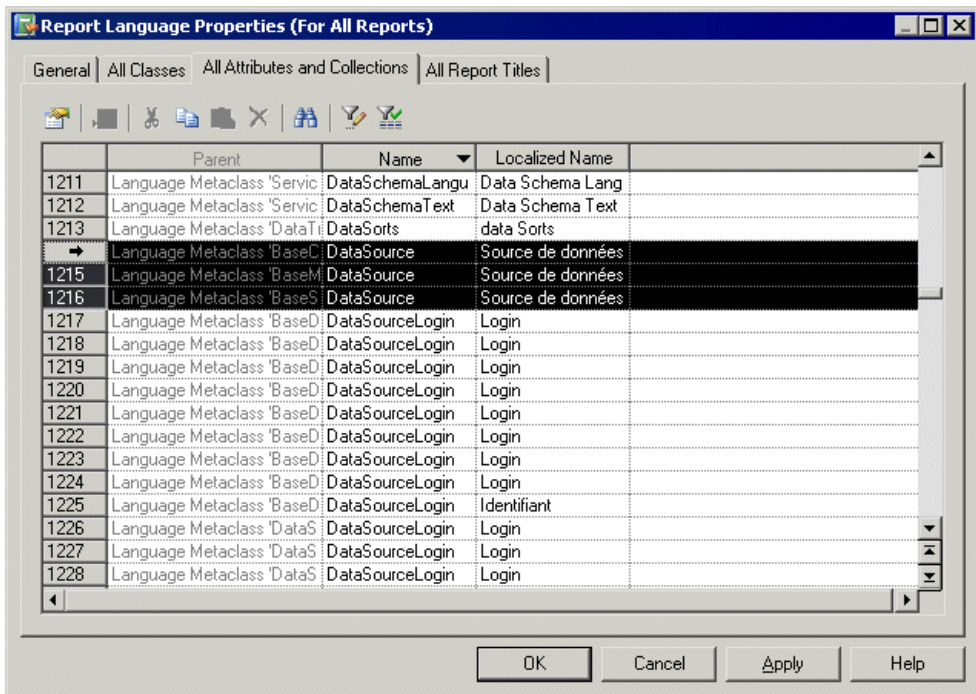
The All Classes tab lists all the metaclasses available in the Object Attributes category on the General tab but the flat structure makes it more convenient to work with.



For 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.

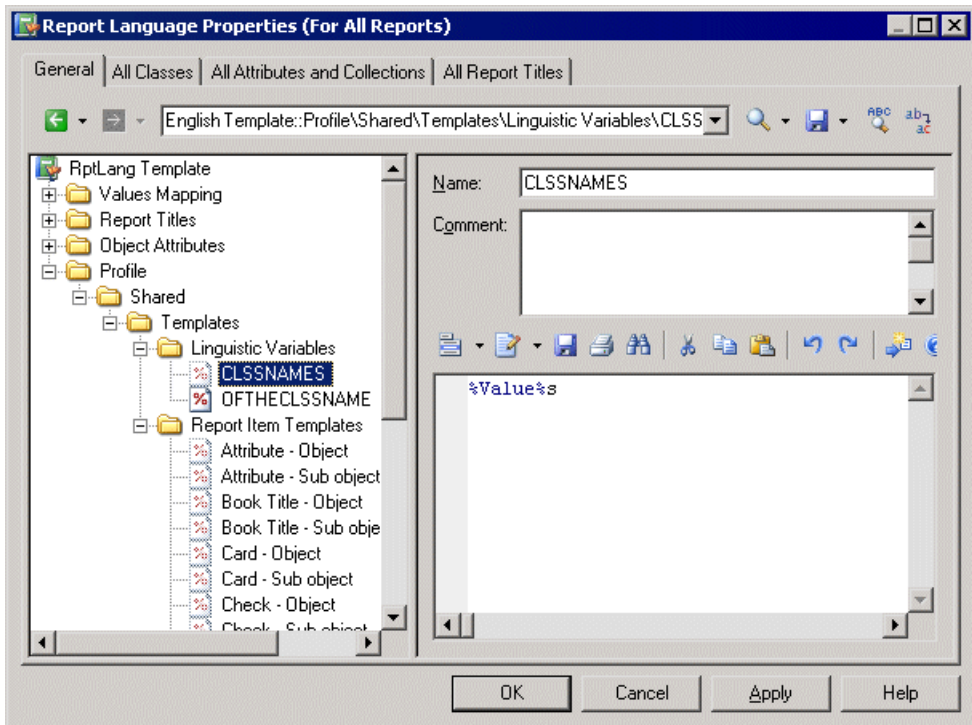


For each attribute or collection listed in the **Name** column, you must enter a translation in the **Localized Name** column. You can sort the list to group similarly-named objects, and translate identical items together by selecting multiple lines.

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 (see *Profile/Report Item Templates Category* on page 303).



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 298).

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.

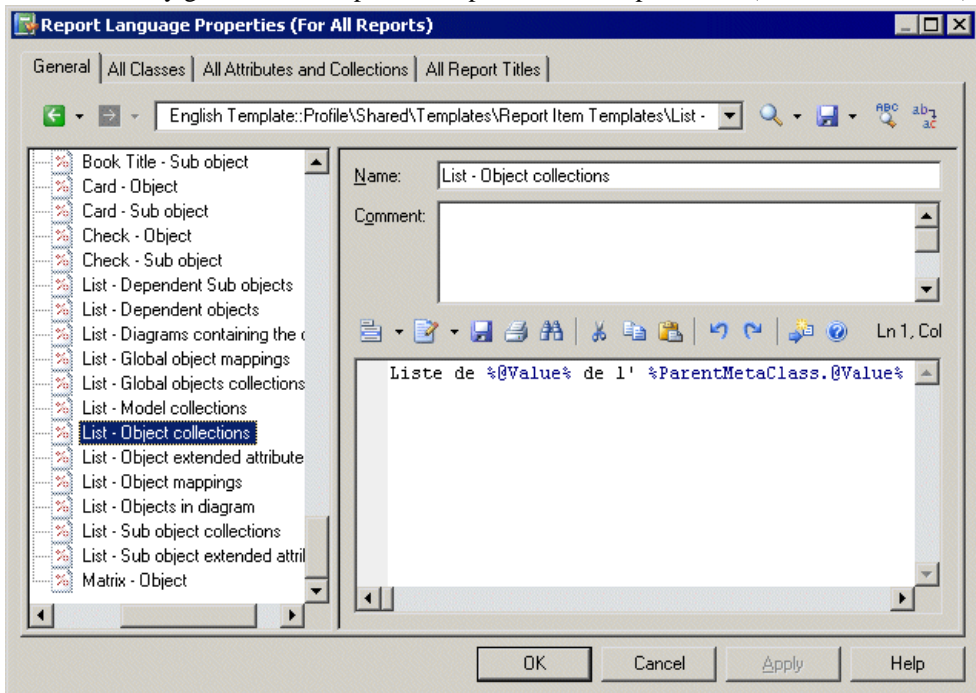
- **OFCLSSNAME** – Inserts the preposition "d' " before the metaclass name %Value%, if it begins with a vowel, otherwise "de".

```
.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.)



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 300).

CHAPTER 6: Translating Reports with Report Language Files

This template will be reused to create report titles for all the lists of sub-objects contained within a collection belonging to an object.

Note: You cannot create or delete templates.

When working with large or multiple models, it can 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.

You can access and modify any PowerDesigner object using Java, VBScript, C#, or many other languages. In this chapter, we focus primarily on writing VBScript to execute in PowerDesigner's Edit/Run Script dialog, but you can also call add-ins from PowerDesigner menus (see *Launching Scripts and Add-Ins from Menus* on page 338) or script the PowerDesigner application via OLE automation (see *OLE Automation and Add-Ins* on page 332).

The following script illustrates the basic syntax of VBScript applied to manipulating PowerDesigner models and objects, including:

- Declaration of local variable
- Assignment of value to a local variable (with the specific case of object)
- Condition operator: If Then/Else/End If
- Iteration on a list: For Each/Next
- Definition and call of a procedure: Sub
- Definition and call of a function: Function
- Error handling using On Error statements

```
' This is a VBScript comment.
Dim var ' Declaration of a local variable
var = 1 ' Value assignment for simple type
Set var = ActiveModel ' Value assignment for an object. ActiveModel
is a PowerDesigner global property
If not var is Nothing Then ' Condition on an object, testing if it is
'null'
    Dim objt ' Declaration of another local variable
    For Each objt In ActiveModel.Children ' Loop on the Children
object collection
        DescribeObject objt ' Procedure call with objt as a parameter
(without parentheses). The procedure is defined below.
    Next
Else
    output "There is no active model" ' Output is a PowerDesigner
procedure that writes text to the Output window
End If

' This is a procedure - a method that does not return a value
Sub DescribeObject(objt)
    Dim desc ' A variable declaration inside the procedure
    desc = ComputeObjectLabel(objt) ' A function call with objt as
parameter (with parentheses). The function is defined below.
```

```

' We retrieve the value returned by the
function in the variable desc
  output desc ' Displays the object description in the output
End Sub

' This is a function - a method that returns a value
Function ComputeObjectLabel(objt)
  Dim label ' Declare a local variable to store the object label
  label = "" ' Initialize the label variable with a default value
  If objt is nothing then
    label = "There is no object"
  ElseIf objt.IsShortcut() then ' IsShortcut is a PowerDesigner
function available on objects
    label = objt.Name & " (shortcut)" ' Concatenation of two strings
  Else
    On Error Goto 0 ' Disables script execution abort on error
    label = objt.Name ' Assigns the object's Name property to the
local variable
    On Error Resume Next ' Reactivates script execution error
  End If
  ComputeObjectLabel = label ' The value is returned by assigning an
implicit variable with same name than the function
End Function

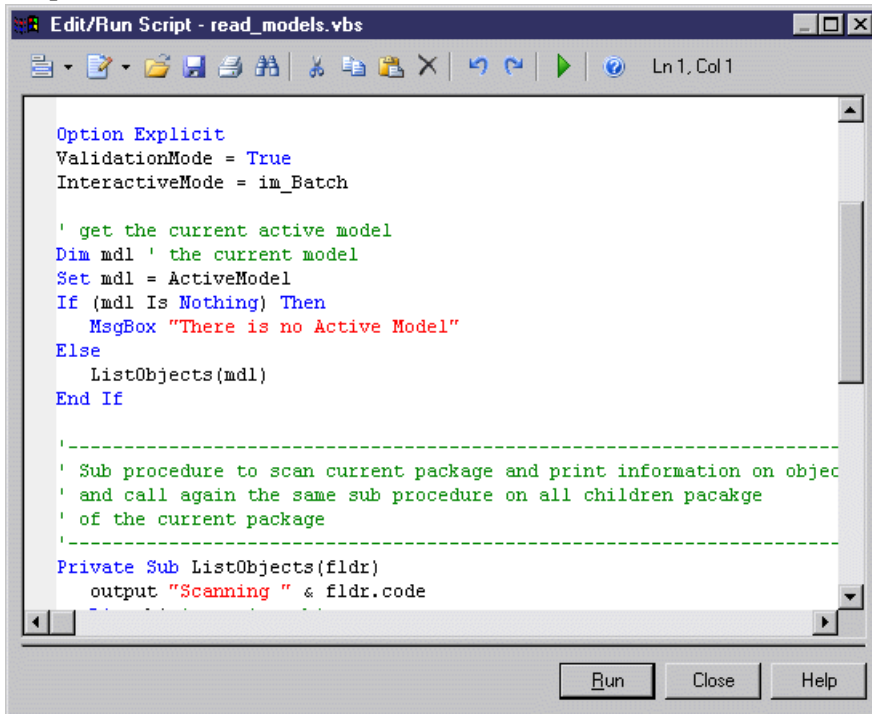
```

Note: VBScript can also be used to create custom checks, event handlers, transformations, and methods in an extension file (see *Chapter 2, Extension Files* on page 11) and embedded in or called from GTL templates (see *.execute_vbscript Macro* on page 270 and *.vbscript Macro* on page 281).









The examples in this chapter are intended to introduce the basic concepts and techniques for controlling PowerDesigner by script. For complete documentation of the PowerDesigner metamodel, select **Help > Metamodel Objects Help**. For full documentation of VBScript, see the *Microsoft MSDN site*.



Running Scripts in PowerDesigner

You can run VBScript scripts in your PowerDesigner client by selecting **Tools > Execute Commands** to open the **Edit/Run Script** dialog. Output from the script is printed to the **Output** window.



The following tools are available on the **Edit/Run Script** dialog toolbar:

| Tools | Description |
|---|--|
|  | <p>Editor Menu [Shift+F11] - Contains the following commands:</p> <ul style="list-style-type: none"> • New [Ctrl+N] - Reinitializes the field by removing all the existing content. • Open... [Ctrl+O] - Replaces the content of the field with the content of the selected file. • Insert... [Ctrl+I] - Inserts the content of the selected file at the cursor. • Save [Ctrl+S] - Saves the content of the field to the specified file. • Save As... - Saves the content of the field to a new file. • Select All [Ctrl+A] - Selects all the content of the field. • Find... [Ctrl+F] - Opens a dialog to search for text in the field. • Find Next... [F3] - Finds the next occurrence of the searched for text. • Find Previous... [Shift+F3] - Finds the previous occurrence of the searched for text. • Replace... [Ctrl+H] - Opens a dialog to replace text in the field. • Go To Line... [Ctrl+G] - Opens a dialog to go to the specified line. • Toggle Bookmark [Ctrl+F2] Inserts or removes a bookmark (a blue box) at the cursor position. Note that bookmarks are not printable and are lost if you refresh the tab • Next Bookmark [F2] - Jumps to the next bookmark. • Previous Bookmark [Shift+F2] - Jumps to the previous bookmark. |
|  | <p>Edit With [Ctrl+E] - Opens the previewed code in an external editor. Click the down arrow to select a particular editor or Choose Program to specify a new editor. Editors specified here are added to the list of editors available at Tools > General Options > Editors.</p> |
|  | <p>Save [Ctrl+S] - Saves the content of the field to the specified file.</p> |
|  | <p>Print [Ctrl+P] - Prints the content of the field.</p> |
|  | <p>Find [Ctrl+F] - Opens a dialog to search for text.</p> |
|  | <p>Cut [Ctrl+X], Copy [Ctrl+C], and Paste [Ctrl+V] - Perform the standard clipboard actions.</p> |
|  | <p>Clear - Deletes the script in the dialog.</p> |
|  | <p>Undo [Ctrl+Z] and Redo [Ctrl+Y] - Move backward or forward through edits.</p> <p>Multiple levels of Undo and Redo are supported but , 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.</p> |

| Tools | Description |
|---|--|
|  | <p>Run [F5] - Runs the script. Output is printed to the Output window.</p> <p>If a compilation error occurs, a message box is displayed, a brief error description appears in the dialog's Result pane, and the cursor is set at the error position.</p> <p>You can catch errors using the <code>On Error Resume Next</code> statement, unless the script is called in the <code>im_Abort</code> interactive mode (see <i>.set_interactive_mode Macro</i> on page 278).</p> |
|  | <p>Find in Metamodel Objects Help [Ctrl+F1] - Opens the PowerDesigner metamodel objects help file, which provides detailed information about all the attributes, collections, and methods available for each metaclass.</p> |

VBScript File Samples

PowerDesigner ships with a set of script samples, that you can use as a basis to create your own scripts, and which are located in the `VB Scripts` folder of the PowerDesigner installation directory. These scripts are intended to show you the range of tasks you can perform on PowerDesigner models using VBScript.

Warning! You should always make a backup copy of the sample script before making changes to it.

Model Scan Sample

The following script browses any model, looping through any packages and listing the objects contained in them:

```
Option Explicit ' Forces each variable to be declared
'before assignment
InteractiveMode = im_Batch ' Supresses the display of dialogs
' get the current active model
Dim diag
Set diag = ActiveDiagram ' the current diagram
If (diag Is Nothing) Then
  MsgBox "There is no Active Diagram"
Else
  Dim fldr
  Set Fldr = diag.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 child packages
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 script creates a new OOM model, then creates a class with attributes and operations:

```

ValidationMode = True 'Forces PowerDesigner to validate
' actions and return errors in the event of a forbidden action
InteractiveMode = im_Batch ' Supresses PowerDesigner dialogs
' 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

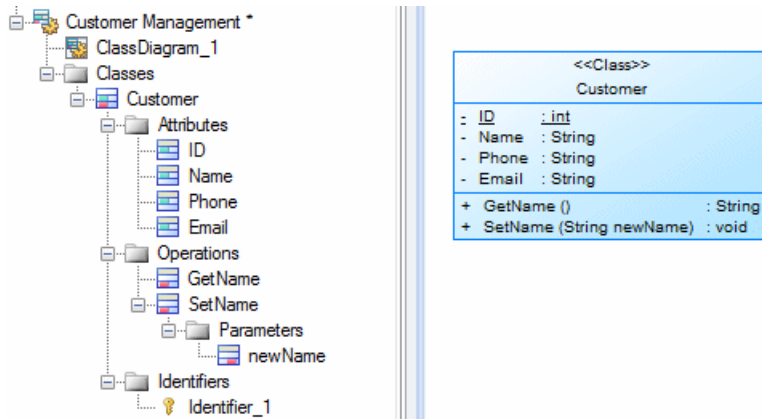
' Create operations function
Function CreateOperations(cls)
    Dim oper
    Set oper = cls.CreateObject(PdOOM.cls_Operation)
    oper.Name = "GetName"
    oper.Code = "GetName"
    oper.ReturnType = "String"
    Dim body
    body = "{" + vbCrLf
    body = body + " return Name;" + vbCrLf
    body = body + "}"
    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

```



Manipulating Models, Collections, and Objects (Scripting)

You can manipulate the contents of a model by creating or opening it and then descending from the model root through collections of objects. A number of global properties, functions, and constants are available in any context and provide entry points for your scripts.

The following global properties provide access to the Workspace and models it contains:

- `ActiveWorkspace` - Retrieves the current Workspace.
- `ActiveModel`, `ActivePackage`, and `ActiveDiagram` - Retrieves the model, package, or diagram with current focus.
- `ActiveSelection` - Read-only collection of the objects selected in the active diagram.
- `Models` - Read-only collection of models open in the current Workspace.
- `RepositoryConnection` - Retrieves the current repository connection (see *Manipulating the Repository (Scripting)* on page 325).

The following global functions are commonly used to create or open models and perform actions upon them:

- `CreateModel ()` and `OpenModel ()` - Create and open a model (see *Creating and Opening Models (Scripting)* on page 313).
- `Output ()` - Prints text to the **Script** tab of PowerDesigner's **Output** window.
- `IsKindOf ()` - Tests the metaclass of the object.
- `ExecuteCommand ()` - Launches an external application

- `EvaluateNamedPath()` and `MapToNamedPath()` - Manage named paths in model files.
- `BeginTransaction()`, `CancelTransaction()`, and `EndTransaction()` - Start, cancel, and commit transactions.

The following global constants provide information about the instance of PowerDesigner:

- `UserName` - Retrieves the user login name.
- `Version` - Returns the PowerDesigner version.
- `HomeDirectory` - Returns the application home directory.
- `RegistryHome` - Returns the application registry home path.
- `Viewer` - Returns `True` if the running application is a Viewer version that has limited features.
- `ValidationMode` - By default, PowerDesigner performs various checks to validate your actions and gives an error in the case of a forbidden action. You can set `ValidationMode = False` (which turns off validation rules such as name uniqueness or link extremities) to improve performance or if your algorithm temporarily requires an invalid state.
- `InteractiveMode` - Specifies the level of interaction required. You can choose between:
 - `im_Batch` [default] - Suppresses dialog boxes and always uses default values. For example, if your model contains external shortcuts and the target model for the shortcuts is closed, this mode will automatically open the model without user interaction.
 - `im_Dialog` - Displays information and confirmation dialog boxes that require user interaction for the execution to keep running.
 - `im_Abort` - Suppresses dialog boxes and aborts execution if a dialog is encountered.
- `ShowMode` [OLE-specific] - Checks or changes the visibility status of the main application window. Returns `True` if the application main window is visible and not minimized.
- `Locked` [OLE-specific] - When set to `True`, ensures that PowerDesigner continues to run even after an OLE client disconnects.

For detailed information about all the global properties, constants, and functions, select **Help** > **MetaModel Objects Help** and navigate to `Basic Elements`.

Creating and Opening Models (Scripting)

You create models and open existing models using the `CreateModel()` and `OpenModel()` global functions. The model with the current focus is accessible via the `ActiveModel` global property, and the models currently open in the workspace are available from the `Models` global collection.

This script creates a new OOM targeting the Analysis language, creates some classes in it, displays them in the diagram, and then saves the model and closes it:

CHAPTER 7: Scripting PowerDesigner

```
Dim NewModel
set NewModel = CreateModel(PdOOM.Cls_Model, "Language=Analysis|
Diagram=ClassDiagram|Copy")
If NewModel is Nothing then
  msgbox "Failed to create UML Model", vbOkOnly, "Error" ' Display an
error message
Else
  output "The UML model has been created" ' Display a message in
Output
  NewModel.SetNameAndCode "MyOOM", "MyOOM" 'Initialize model name and
code
  For idx = 1 to 12 'Create classes and display them
    Set obj=NewModel.Classes.CreateNew()
    obj.SetNameAndCode "C" & idx, "C" & idx
    Set sym=ActiveDiagram.AttachObject (obj)
  Next
  ActiveDiagram.AutoLayoutWithOptions(2)
  NewModel.Save "c:\temp\MyOOM.oom" ' Save the model
  NewModel.Close ' Close the model
  Set NewModel = Nothing ' Release last reference to object to free
memory
End If
```

This script verifies that the previously created model exists, and then opens it in the workspace:

```
Dim MyModel, FileName
FileName = "c:\temp\MyOOM.oom"
On Error Resume Next ' Avoid generic scripting error message
Set MyModel = OpenModel(FileName)
If MyModel is nothing then ' Display an error message box
  msgbox "Failed to open Model:" + vbCrLf + FileName, vbOkOnly,
"Error"
Else ' Display a message in Output
  output "The OOM has been opened."
End If
```

Browsing and Modifying Collections (Scripting)

Most metamodel navigation is performed by descending from the model root through collections of objects to collections of sub-objects or associated objects. An OOM contains a collection of classes and classes contain collections of attributes and operations. You can obtain information about and browse the members of a collection through scripting, as well as adding, removing, and moving objects in the collection.

To browse the members of a collection, navigate to the parent object and then use a For each loop. This script prints the names of all the tables in an open PDM:

```
Dim MyModel
Set MyModel=ActiveModel
For each t in MyModel.Tables
  Output "*" & t.Name
Next
```

When you browse a collection, both full objects in the collection and any shortcuts will be retrieved.

Note: For information about accessing collections defined in extensions, see *Creating and Accessing Extensions (Scripting)* on page 329.

The following kinds of collections appear in the metamodel:

- **Compositions** - contain objects that will be deleted if the parent is deleted. For example, the PdPDM/Tables and PdPDM/Table/Columns collections are compositions.
- **Aggregations** - reference objects that will continue to exist if the parent is deleted. For example, the PdCommon/NamedObject/AttachedRules collection (inherited by most objects) is an aggregation.
- **Unordered collections** - contain objects with no significant order. For example, the PdCDM/Entity/Relationships collection is unordered.
- **Ordered collections** - contain objects where the user chooses the order. For example, the PdPDM/Table/Columns collection is ordered.
- **Read-only collections** - can only be browsed. For example, the global Models collection (all open models) is read-only.

The following properties are available for all collections:

- **Count** - Retrieves the number of objects in the collection.
- **Item [(index)]** - Retrieves the specified item in the collection as an object. Item (0) is the first object (and the default) and Item (-1) is the last object.
- **MetaCollection** - Retrieves the metadefinition of the collection as an object.
- **Kind** - Retrieves the type of objects the collection can contain.
- **Source** - Retrieves the object on which the collection is defined.

The following methods are available for modifying writeable collections:

- **CreateNew ([kind] and CreateNewAt (index [, kind])** - [compositions only] Creates a new object at the end of the collection or at the specified **index** (default, -1). The **kind** parameter (for example, PdPDM.cls_Table) is only needed if the collection supports multiple kinds of objects.
- **Add (object)** - Inserts the specified **object** at the end of the collection.
- **Insert ([index] [, object])** - Inserts the specified **object** in the collection at the specified **index** position (default, -1).
- **Move (index2, index1)** - Moves the object at position **index1** to position **index2** in the collection.
- **Remove (object [, delete = y|n]) and RemoveAt ([index] [, delete = y|n])** - Removes the specified **object** or the object at the specified **index** (default, -1) from the collection. For aggregations, you can additionally specify to delete the object (objects removed from a composition are always deleted).

CHAPTER 7: Scripting PowerDesigner

- `Clear([delete = y|n])` - Removes all objects from the collection and optionally deletes them.

The following script:

- Creates a PDM,
- Creates objects in the model's `Tables` and `BusinessRules` unordered composition collections, and
- Adds some objects to table `T1`'s `AttachedRules` ordered aggregation collection and then manipulates that collection:

```
Dim MyModel, t, r, sym
set MyModel = CreateModel(PdPDM.Cls_Model,"DBMS=SYASA12")
MyModel.SetNameAndCode "MyPDM" , "MyPDM"
'Create tables and rules
For idx = 1 to 12
    Set t=MyModel.Tables.CreateNew()
    t.SetNameAndCode "T" & idx, "T" & idx
    Set sym=ActiveDiagram.AttachObject (t)
    Set r=MyModel.BusinessRules.CreateNew()
    r.SetNameAndCode "BR" & idx, "BR" & idx
Next
ActiveDiagram.AutoLayoutWithOptions(2)
'Attach rules to Table 1
Dim MyTable
Set MyTable=MyModel.FindChildByName("T1",cls_table)
For idx = 1 to 10
    MyTable.AttachedRules.Add(MyModel.FindChildByName("BR" &
(idx),cls_businessrule))
Next
'Print list of rules attached to Table 1
Output "Rules Attached to T1 (" & MyTable.AttachedRules.Count & ")"
For each r in MyTable.AttachedRules
    Output "*" & r.Name
Next
'Modify attached rules by insertion, move and removal
MyTable.AttachedRules.Insert 3,
MyModel.FindChildByName("BR12",cls_businessrule)
MyTable.AttachedRules.Move 5,0
MyTable.AttachedRules.Remove(MyModel.FindChildByName("BR6",cls_busi
nessrule))
'Print modified list of rules
Output "Modified Rules Attached to T1 (" &
MyTable.AttachedRules.Count & ")"
For each r in MyTable.AttachedRules
    Output "*" & r.Name
Next
```

Accessing and Modifying Objects and Properties (Scripting)

You can access and modify any PowerDesigner object and its properties by script. Objects include not only standard design objects (such as tables, classes, processes, and columns), but also diagrams and symbols and functional objects (such as a report or repository). An object

belongs to a metaclass of the PowerDesigner metamodel and inherits properties, collections and methods from its metaclass.

Root objects, such as models, are accessed using global properties and functions (see *Manipulating Models, Collections, and Objects (Scripting)* on page 312), while standard objects are accessed by browsing collections (see *Browsing and Modifying Collections (Scripting)* on page 314) or individually through the following methods:

- `FindChildByName ("Name" , Kind [, OptionalParams]`
- `FindChildByCode ("Code" , Kind [, OptionalParams]`
- `FindChildByPath ("Path" , Kind [, OptionalParams]`

The following parameters are available:

| Parameter | Description |
|---------------------------|--|
| Name / Code / Path | Specifies the name or code of, or the path to the object. For example, to find the column <code>Address</code> in the table <code>Customer</code> in the package <code>Sales</code> from the context of the model node, you could search by name <code>Address</code> or by path <code>Sales/Customer/Address</code> . |
| Kind | Specifies the metaclass of the object to find in the form <code>cls_PublicName</code> . For example, to find a column, select <code>cls_Column</code> . These metaclass ids are unique within their model library but, in cases such as packages, which appear in multiple types of models, you must prefix the id with the name of the module (<code>PdOOM.cls_Package</code>). When you create a model, you must use the module prefix (for example <code>PdPDM.cls_Model</code>). |
| OptionalParams | The following parameters are optional: <ul style="list-style-type: none"> • "Stereotype" - Specifies that the object to find must bear the specified stereotype. • "LastFound" - Specifies to begin the search after this object. This parameter is used when several objects have the same path value, and can be used to launch a find in a while loop that uses the previous match as the last found parameter. • CaseSensitive=y n - [default: y] Specifies that the search is case sensitive. • IncludeShortcuts - [default: n] Specifies that shortcuts can be found. • UseCodeInPlaceOfName - [ByPath, default: n] Specifies that the object can be found by its code (Default=n). • PathSeparator - [ByPath, default=/, \, or ::] Specifies the character to separate nodes in the path. |

You can get standard attribute values using the dot notation (**object . attribute**) or using the following methods:

- `GetAttribute ("attribute")` - retrieves the value stored for the attribute
- `GetAttributeText ("attribute")` - retrieves the value displayed for the attribute

You can set attribute values using the dot notation (**object . attribute=value**) or using the following methods:

- SetAttribute "attribute", value
- SetAttributeText "attribute", "value"

Note: For information about getting and setting extended attribute values see *Creating and Accessing Extensions (Scripting)* on page 329

The following script opens a sample OOM, finds a class by name and a parameter by path, and then prints and modifies some of their properties:

```
Dim MyModel, C, P
'Open model file
Set MyModel=OpenModel(EvaluateNamedPath("%_EXAMPLES%" & "UML2
Sample.oom"))
'Obtain class and parameter
Set C=MyModel.FindChildByName("OrderManager",cls_Class)
Set P=MyModel.FindChildByPath("SecurityManager/CheckPassword/
login",PdOOM.cls_Parameter)

'Print initial values
Output "Initial Values:"
PrintProperties C, P
'Modify values
C.Comment="This class controls orders."
C.SetAttributeText "Visibility", "private"
P.Name="LoginName"
'Print revised values
Output "Revised Values:"
PrintProperties C, P

'Procedure for printing values
Sub PrintProperties(MyClass, MyParam)
output "Class: " & MyClass.Name
output vbTab & "Comment: " & MyClass.Comment
output vbTab & "Visibility: " &
MyClass.GetAttributeText("Visibility")
output vbTab & "Persisted as: " &
MyClass.GetAttributeText("PersistentGenerationMode")
output "Parameter: " & MyParam.Parent & "." & MyParam.Name
output vbTab & "Data type: " & MyParam.DataType
output vbTab & "Parameter type: " &
MyParam.GetAttributeText("ParameterType")
End Sub
```

Creating Objects (Scripting)

You should generally create objects via the collection under the parent object using the CreateNew() method. The CreateObject(kind) method is also available on model objects.

This script creates a class in an OOM, sets some of its properties, and then creates an attribute under the class, in each case creating the objects inside collections:

```

Dim MyModel
Set MyModel = ActiveModel
Dim MyClass
' Create a class
Set MyClass = MyModel.Classes.CreateNew()
If MyClass is nothing Then
    ' Display an error message box
    msgbox "Fail to create a class", vbOkOnly, "Error"
Else
    output "The class has been created."
    ' Set Name, Code, Comment, Stereotype and Final attributes
    MyClass.SetNameAndCode "Customer", "cust"
    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 attribute has been created."
        MyAttr.SetNameAndCode "Name", "custName"
        MyAttr.DataType = "String"
        ' Reset the variable in order to avoid memory leaks
    End If
End If

```

You can also create objects using the `CreateObject` (*kind*) method. This script creates a class inside an OOM and sets some of its properties:

```

Dim MyModel
Set MyModel = ActiveModel
Dim MyClass
' Create a class
Set MyClass = MyModel.CreateObject(cls_Class)
MyClass.SetNameAndCode "Another Class", "Class2"
MyClass.Comment = "Created by CreateObject"

```

When creating a link object, you must define its extremities. This script creates two classes and joins them by an association link:

```

Dim MyModel
Set MyModel = ActiveModel
Dim MyFirstClass, MySecondClass, MyAssociation
' Create classes
Set MyFirstClass = MyModel.Classes.CreateNew()
MyFirstClass.SetNameAndCode "Class1", "C1"
Set MySecondClass = MyModel.Classes.CreateNew()
MySecondClass.SetNameAndCode "Class2", "C2"
' Create association
Set MyAssociation = MyModel.Associations.CreateNew()
MyAssociation.Name = "A1"
' Define its extremities
Set MyAssociation.Object1 = MyFirstClass
Set MyAssociation.Object2 = MySecondClass

```

Displaying, Formatting, and Positioning Symbols (Scripting)

When you create an object, it will not appear in a diagram unless you use the `AttachObject()` or `AttachLinkObject()` method. Symbols are objects in their own right that can be accessed via collections on the parent object or diagram. You can position a symbol using the `Position()` method and change its format using the `LineWidth` and other formatting attributes.

The following script creates two classes, joins them by an association link, and displays all three symbols in the active diagram:

```
Dim MyModel, MyDiagram, C1, C2, A1
Set MyModel = ActiveModel
Set MyDiagram = ActiveDiagram
' Create classes
Set C1 = MyModel.Classes.CreateNew()
C1.SetNameAndCode "C1", "C1"
Set C2 = MyModel.Classes.CreateNew()
  C2.SetNameAndCode "C2", "C2"
' Display class symbols
MyDiagram.AttachObject(C1)
MyDiagram.AttachObject(C2)
' Create association
Set A1 = MyModel.Associations.CreateNew()
A1.SetNameAndCode = "A1", "A1"
' Define its extremities
Set A1.Object1 = C1
Set A1.Object2 = C2
' Display Association symbol
MyDiagram.AttachLinkObject(A1)
```

The following script creates an EAM and four architecture areas, aligns them in a square, and formats the top-left area:

```
Dim NewModel, idx, obj, sym
set NewModel = CreateModel(PdEAM.Cls_Model,
"Diagram=CityPlanningDiagram")
NewModel.SetNameAndCode "MyEAM" , "MyEAM"
For idx = 1 to 4
  Set obj=NewModel.ArchitectureAreas.CreateNew()
  obj.SetNameAndCode "A" & idx, "A" & idx
  Set sym=ActiveDiagram.AttachObject(obj)
  sym.width=30000
  sym.height=20000
Next
dim A1, A2, A3, A4, X1, Y1
set A1 =
NewModel.FindChildByName("A1",cls_architecturearea).Symbols.Item(0)
set A2 =
NewModel.FindChildByName("A2",cls_architecturearea).Symbols.Item(0)
set A3 =
NewModel.FindChildByName("A3",cls_architecturearea).Symbols.Item(0)
set A4 =
NewModel.FindChildByName("A4",cls_architecturearea).Symbols.Item(0)
```

```

X1 = A1.Position.X
Y1 = A1.Position.Y
' Move symbols for them to be adjacent
A2.Position = NewPoint(X1 + A1.Width, Y1)
A3.Position = NewPoint(X1, Y1 - A1.Height)
A4.Position = NewPoint(X1 + A1.Width, Y1 - A1.Height)
A1.DashStyle = 2
A1.LineWidth = 3

```

Deleting Objects (Scripting)

You can delete objects using the Delete method.

The following script creates a new CDM, populates it with entities and relationships, and then deletes entity E5 and relationship R8:

```

Dim MyModel, obj, sym, idx
set MyModel = CreateModel(PdCDM.Cls_Model)
MyModel.SetNameAndCode "MyCDM" , "MyCDM"
'Create entities
For idx = 1 to 12
    Set obj=MyModel.Entities.CreateNew()
    obj.SetNameAndCode "E" & idx, "E" & idx
    Set sym=ActiveDiagram.AttachObject (obj)
Next
'Create relationships
For idx = 2 to 11
    Set obj=MyModel.Relationships.CreateNew()
    obj.SetNameAndCode "R" & idx-1, "R" & idx-1
    Set obj.Object1 = MyModel.FindChildByName("E" &
(idx-1),cls_entity)
    Set obj.Object2 = MyModel.FindChildByName("E" & (idx
+1),cls_entity)
    Set sym=ActiveDiagram.AttachLinkObject (obj)
Next
ActiveDiagram.AutoLayoutWithOptions(2)
'Delete objects
MyModel.FindChildByName("E5",cls_entity).Delete
MyModel.FindChildByName("R8",cls_relationship).Delete

```

Creating an Object Selection (Scripting)

You can create a selection of objects using the CreateSelection() method. You can perform actions on the selection such as changing properties or format or moving them to another package.

The following script creates a PDM, populates it with tables and then makes a selection of tables and moves them into a package:

```

Dim MyModel, obj, sym
set MyModel = CreateModel(PdPDM.Cls_Model,"DBMS=SYASA12")
MyModel.SetNameAndCode "MyPDM" , "MyPDM"
'Create tables
For idx = 1 to 12
    Set obj=MyModel.Tables.CreateNew()

```

```

    obj.SetNameAndCode "T" & idx, "T" & idx
    Set sym=ActiveDiagram.AttachObject (obj)
Next
ActiveDiagram.AutoLayoutWithOptions (2)
'Create package
Dim MyPackage
Set MyPackage=MyModel.Packages.CreateNew()
MyPackage.SetNameAndCode "P1", "P1"
ActiveDiagram.AttachObject (MyPackage)
'Create selection
Dim MySelection
Set MySelection = ActiveModel.CreateSelection
For idx = 1 to 5
    MySelection.Objects.Add(MyModel.FindChildByName("T" &
(idx*2),cls_table))
Next
'Move selection to package
MySelection.MoveToPackage(MyPackage)

```

To add all the tables to the selection, use the `AddObjects` method:

```
MySelection.AddObjects MyModel,cls_table
```

To remove an object from the selection, use the `Remove` method:

```
MySelection.Objects.Remove(MyModel.FindChildByName("T6",cls_table))
```

Controlling the Workspace (Scripting)

You can access the current workspace using the `ActiveWorkspace` global property, open, save, and close workspaces, and add folders and documents to it.

The following script constructs a simple folder structure in a workspace and adds and creates several models in it:

```

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
    ElseIf item.IsKindOf(PdWsp.cls_WorkspaceModel) Then
    ElseIf 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"

```

For more information about properties and methods available on the workspace, select **Help > MetaModel Objects Help** and navigate to **Libraries/PdWSP/Workspace**.

Creating Shortcuts (Scripting)

You create a shortcut in a model using the `CreateShortcut()` method.

The following script acts on an OOM and creates a shortcut of the class C1 from package P1 in package P2:

```

Dim obj, shortcut, recipient
' Get class to shortcut
Set obj = ActiveModel.FindChildByPath("P1/C1",cls_Class)
' Get package to create shortcut in
Set recipient = ActiveModel.FindChildByPath("P2",PdOOM.cls_Package)
' Create shortcut
Set shortcut = obj.CreateShortcut(recipient)
If not shortcut is nothing then
    output "The class shortcut has been successfully created"
End If

```

The following script creates a shortcut of the class C1 from model O1 package P1 directly under model O2:

```

Dim targetmodel, usingmodel, obj, shortcut
For each m in Models
    Output m.Name
    If m.Name="O1" then 'Get model with object to shortcut
        Set targetmodel=m
    End If
    If m.Name="O2" then 'Get model to create shortcut in
        Set usingmodel=m
    End If
Next
' Get object to shortcut
Set obj = targetmodel.FindChildByPath("P1/C1",cls_Class)
' Create shortcut
Set shortcut = obj.CreateShortcut(receivingmodel)
If not shortcut is nothing then
    output "The class shortcut has been successfully created"
End If

```

Creating Mappings Between Objects (Scripting)

You can create data sources in a model and from there create mappings from source objects in other models to objects in the first model using scripts.

The following script creates an OOM and a PDM, populates them with classes and tables, then creates a data source in the OOM, associates the PDM with it and creates mappings:

```

Dim MyOOM, MyPDM
'Create an OOM and a PDM
set MyOOM = CreateModel(PdOOM.Cls_Model, "|Language=Analysis|
Diagram=ClassDiagram|Copy")
MyOOM.SetNameAndCode "MyOOM", "OOM"
set MyPDM = CreateModel(PdPDM.Cls_Model, "|DBMS=Sybase SQL Anywhere
12|Copy")
MyPDM.SetNameAndCode "MyPDM", "PDM"
'Create classes and tables
For idx = 1 to 6
    Set c=MyOOM.Classes.CreateNew()
    c.SetNameAndCode "Class" & idx, "C" & idx
    Set t=MyPDM.Tables.CreateNew()
    t.SetNameAndCode "Table" & idx, "T" & idx
Next
'Create a data source in the OOM and add the PDM as its source
Dim ds, m1
Set ds = MyOOM.DataSources.CreateNew()
ds.SetNameAndCode "MyPDM", "PDM"
ds.AddSource MyPDM

'Create a mapping between C1 and T6
set m1 = ds.CreateMapping(MyOOM.FindChildByName("Class1",cls_class))
m1.AddSource MyPDM.FindChildByName("Table6",cls_table)
' Retrieve mappings for each class in the OOM
For each c in MyOOM.Classes
    Dim m, sc
    set m = ds.GetMapping(c)

```



```

If not m is nothing then
    Output c.Name & vbtab & "Mapped to: "
    for each sc in m.SourceClassifiers
        output vbtab & vbtab & "- " & sc.Name
    next
Else
    Output c.Name & vbtab & "No mapping defined."
End if
Next

```

For more information about objects mapping, see *Core Features Guide > Linking and Synchronizing Models > Object Mappings*.

Creating and Generating Reports (Scripting)

You can create a report, browse its contents, and generate it as HTML or RTF using scripting.

To create a report, use the `CreateReport()` method on a model. For example:

```

Dim model
Set model = ActiveModel
model.CreateReport("MyReport")

```

To browse the reports in a model, use the `Reports` collection. For example:

```

Dim model
Set model = ActiveModel
For each m in model.Reports
    Output m.Name
Next

```

To generate a report as RTF or HTML, use the `GenerateRTF()` or `GenerateHTML()` method:

```

set m = ActiveModel
For each r in m.Reports
    filename = "C:\temp\" & r.Name & ".htm"
    r.GenerateHTML (filename)
Next

```

Manipulating the Repository (Scripting)

You can connect to the repository and check documents into and out of it by script and iterate on the latest versions of repository documents through the `RepositoryConnection` object. You can manage repository folders and branches and LDAP and SMTP servers and the repository password policy, but you cannot manipulate repository users and groups.

The following script opens a repository connection, creates a new PDM and checks it in, and then loops over the creation of tables, and further consolidations, before closing the connection:

```
Dim rc
Set rc = RepositoryConnection
rc.Open "REPOSITORYNAME", "USER", "PW", "DBUSER", "DBPW"
Output "Before consolidation"
ListChildren rc
Dim NewModel
Set NewModel = CreateModel(PdPDM.Cls_Model, "|Language=SYASIQ1540")
NewModel.Name = "My PDM"
NewModel.ConsolidateNew rc
For i = 1 to 5
    For j = 1 to 5
        NewModel.Tables.CreateNew()
    Next
    NewModel.Consolidate
Next
Output "After consolidation"
ListChildren rc
rc.Close

Sub ListChildren(rc)
For each c in rc.ChildObjects
    Output c.Name & "(Modified: " & c.ModificationDateInRepository &
    ") "
Next
End Sub
```

To check out a model, use the `CheckOut` method.

For detailed information about the members, collections, and methods available for scripting the repository, select **Help > MetaModel Objects Help** and navigate to `Libraries/PdRMG`.

Generating a Database (Scripting)

You can generate a PDM as a SQL script or directly to a live database connection using the `GenerateDatabase()` method. You can generate test data with the `GenerateTestData()` method.

The following script fragment opens an example PDM and then calls procedures to generate various scripts:

```
Dim GenDir, MyModel
GenDir = "C:\temp\"
Set MyModel=OpenModel(EvaluateNamedPath("%_EXAMPLES%" &
"project.pdm"))

GenerateDatabaseScripts MyModel 'Generate a SQL script to create the
database
ModifyModel MyModel 'Modify each table in the model
GenerateAlterScripts MyModel - Generate alter scripts to modify the
database
GenerateTestDataScript MyModel - generate test data to load into the
database
```

This procedure generates a SQL script to create the database:

```
Sub GenerateDatabaseScripts(m)
    Dim opts
    Set opts = m.GetPackageOptions()
    InteractiveMode = im_Batch ' Avoid displaying generation window
    opts.GenerateODBC = False ' Force sql script generation rather
than ODBC
    opts.GenerationPathName = GenDir
    opts.GenerationScriptName = "MyScript.sql"
    m.GenerateDatabase ' Launch the Generate Database feature
End Sub
```

To generate to a live database connection, you would connect to the database (using the `ConnectToDatabase()` method) and then set the `GenerateODBC` property to true.

Note: For more information about the generation options, select **Help > MetaModel Objects Help** and navigate to `Libraries/PdPDM/BasePhysicalPackageOptions`.

This procedure modifies the model by adding a new column to each table:

```
Sub ModifyModel(m)
    dim pTable, pCol
    For each pTable in m.Tables
        Set pCol = pTable.Columns.CreateNew()
        pCol.SetNameAndCode "az" & pTable.Name, "AZ" & pTable.Code
        pCol.Mandatory = False
    Next
End Sub
```

This procedure generates an alter script to modify the database:

```
Sub GenerateAlterScripts(m)
    Dim pOpts
    Set pOpts = m.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 = "MyAlterScript.sql"
    m.ModifyDatabase ' Launch the Modify Database feature
End Sub
```

This procedure generates test data to load to the database:

```
Sub GenerateTestDataScript(m)
    Dim pOpts
    Set pOpts = m.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 = "MyTestData.sql"
m.GenerateTestData ' Launch the Generate Test Data feature
End Sub
```

Reverse Engineering a Database (Scripting)

You can connect to a database using the `ConnectToDatabase()` method, and reverse engineer the schema to a PDM using `ReverseDatabase()`.

To connect to a database via a user or system data source, define a constant in the form "ODBC:**datasourcename**". For example:

```
Const cnxDSN = "ODBC:ASA 9.0 sample"
```

To use a data source file, define a constant with the full path to the DSN file. For example:

```
Const cnxDSN = "\\romeo\public\DATABASES\_filedsn
\sybase_asa9_sample.dsn"
```

This script creates a new PDM, connects to a database via a system data source, sets reverse options and reverses all objects to the PDM:

```
' Define ODBC data source and PDM file
Const cnxDSN = "ODBC:MyDatabase"
Const cnxUSR = "MyUser"
Const cnxPWD = "MyPassword"
Const filename = "C:\temp\MyReversedDB.pdm"

Dim pModel, pOpt
' Create model with appropriate DBMS
Set pModel=CreateModel(PdPDM.cls_Model, "|DBMS=Sybase SQL Anywhere
12")
' Hide dialogs
InteractiveMode = im_Batch

' Connect to the database
pModel.ConnectToDatabase cnxDSN, cnxUSR, cnxPWD
' Set reverse options to reverse all listed objects via ODBC
Set pOpt = pModel.GetPackageOptions()
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

' Reverse database to model and then save model
```

```
pModel.ReverseDatabase
pModel.save(filename)
```

Creating and Accessing Extensions (Scripting)

You can create extensions by script to define additional properties, new metaclasses, forms, and any other type of extension to the standard metamodel.

The following example creates an EAM, then creates an extension inside it, defines a new type of object called `tablet` derived from the `MobileDevice` metaclass, and creates an extended attribute and new custom form for it:

```
Dim MyModel, MyExt, MyStype, MyExAtt, MyForm, FormDef
set MyModel =
CreateModel(PdEAM.Cls_Model, "Diagram=TechnologyInfrastructureDiagram")
MyModel.SetNameAndCode "MyEAM" , "MyEAM"
'Create extension
Set MyExt = MyModel.ExtendedModelDefinitions.CreateNew()
MyExt.Name = "MyExtension"
MyExt.Code = "MyExtension"
'Create stereotype
Set MyStype = MyExt.AddMetaExtension(PdEAM.Cls_MobileDevice,
Cls_StereotypeTargetItem)
MyStype.Name = "Tablet"
MyStype.UseAsMetaClass = true
'Create extended attribute
Set MyExAtt =
MyStype.AddMetaExtension(Cls_ExtendedAttributeTargetItem)
MyExAtt.Name = "TabletType"
MyExAtt.Label = "Type"
MyExAtt.DataType = "l2" ' (String) For a full list of values,
' see ExtendedAttributeTargetItem in the Metamodel objects help
MyExAtt.ListOfValues = "iPad;Android;Playbook;Windows8"
MyExAtt.Value = "iPad"
'Create form to replace General tab
Set MyForm = MyStype.AddMetaExtension(Cls_FormTargetItem)
MyForm.Name = "ReplaceGeneral"
MyForm.FormType = "GENERAL"
'Assemble form definition
FormDef = "<Form><StandardNameAndCode Attribute=""NameAndCode"" />"
& vbCrLf
FormDef = FormDef + "<StandardAttribute Attribute=""Comment"" />" &
vbCrLf
FormDef = FormDef + "<ExtendedAttribute Attribute=""TabletType"" />"
& vbCrLf
FormDef = FormDef + "<StandardAttribute Attribute=""KeywordList"" /"
></Form>"
MyForm.Value = FormDef
```

You can get and set extended attribute values using the following methods:

- `GetExtendedAttribute("resource.attribute")`

- `GetExtendedAttributeText("resource.attribute")`
- `SetExtendedAttribute "resource.attribute" "value"`
- `SetExtendedAttributeText "resource.attribute" "value"`

You can access collections defined in an extension using the following methods:

- `GetCollectionByStereotype("stereotype")` - for new types of objects defined in an target or extension file (see *Promoting a Stereotype to Metaclass Status* on page 39).
- `GetExtendedCollection("resource.collection")` - for extended collections and compositions (see *Extended Collections and Compositions (Profile)* on page 48).
- `GetCalculatedCollection("resource.collection")` - for calculated collections (see *Calculated Collections (Profile)* on page 50).
- `GetCollectionByName("resource.collection")` -for any kind of collection.

The following script uses the `GetCollectionByStereotype()` method to access the collection of tablets and the `SetExtendedAttribute` method to set the tablet type:

```
Dim col, obj
'The collection of tablets is not directly accessible
set col = ActiveModel.GetCollectionByStereotype("Tablet")
'Create an array to hold the values to assign to tablet properties
Dim myArray(3)
myArray(0) = "Tablet1, T1, PlayBook"
myArray(1) = "Tablet2, T2, Android"
myArray(2) = "Tablet3, T3, iPad"
myArray(3) = "Tablet4, T4, iPad"
CreateObjects col, myArray

'Procedure to assign values to properties
Sub CreateObjects(compColl, dataArray)
  For Each line In dataArray
    Dim myProps
    myProps = split(line, ",")
    set obj = compColl.CreateNew()
    obj.Name = myProps(0)
    obj.Code = myProps(1)
    'Special syntax for setting extended attribute
    obj.SetExtendedAttribute "MYEXT.TabletType", myProps(2)
  Next
End Sub
```

Accessing Metadata (Scripting)

You can explore the structure of the PowerDesigner metamodel as a standalone model or starting from object instances in your model.

For general information about accessing and navigating in the metamodel, see *Chapter 8, The PowerDesigner Public Metamodel* on page 343. Metaclasses (such as `CheckModelInternalMessage` and `FileReportItem`) that are not accessible by

script are visible in Metamodel.oom, but bear the <<notScriptable>> stereotype and are not listed in the Metamodel Object Help file.

You can access metaclasses, metaattributes, and metacollections by iterating over collections descending from the MetaModel root or individually through the following methods:

- GetMetaClassByPublicName (**name**) - to access a metaclass by its public name.
- GetMetaMemberByPublicName (**name**) - to access a metaattribute or a metacollection by its public name

The following script traverses the metamodel by library and lists each concrete class:

```
for each l in MetaModel.Libraries
  for each c in l.Classes
    if c.Abstract = false then
      Output l.PublicName + "." + c.PublicName
    end if
  next
next
```

The following script locates the BaseClass root and shows the first two levels of inheritance under it:

```
set root = MetaModel.GetMetaClassByPublicName("PdCommon.BaseObject")
for each c in root.Children
  output c.PublicName
  for each cc in c.Children
    output "    " + cc.PublicName
  next
next
```

The following script obtains a table in a PDM, and then shows the metaclass of which the object is an instance, the parent metaclass and metalibrary to the metaclass, and all the attributes and collections that are available on that metaclass:

```
Dim object
Set object = ActiveModel.FindChildByName("myTable", cls_Table)
Output "Object: " + object.Name

Dim metaclass
Set metaclass = object.MetaClass
Output "Metaclass: " + metaclass.PublicName
Output "Parent: " + metaclass.Parent.PublicName
Output "Metalibrary: " + metaclass.Library.PublicName
Output "Attributes:"
For each attr in metaclass.attributes
  Output " - " + attr.PublicName
Next
Output "Collections:"
For each coll in metaclass.collections
  Output " - " + coll.PublicName
Next
```

Properties and collections are read-only for all metamodel objects.

OLE Automation and Add-Ins

OLE Automation provides a way to communicate with PowerDesigner from another application using the COM architecture. You can write a program using any language that supports COM, such as Word or Excel macros, VB, C++, or PowerBuilder. You can create executables that call PowerDesigner or add-ins that are called by PowerDesigner.

VBScript programs that run from within PowerDesigner and OLE Automation programs are very similar, but OLE requires you to work through a PowerDesigner application object, and to use stronger typing. You must:

- Create an instance of the PowerDesigner Application object and release it when your script terminates:

```
Dim PD As PdCommon.Application
Set PD = CreateObject("PowerDesigner.Application")
'Enter script here
'Once script is finished, release PD object
Set PD = Nothing
```

If PowerDesigner is currently running, this instance will be used; otherwise a new instance will be launched. If you do not specify a version number, the most recent version is used. To specify a specific version, use the syntax:

```
Set PD = CreateObject("PowerDesigner.Application.version")
```

- Prefix all global properties and functions (see *Manipulating Models, Collections, and Objects (Scripting)* on page 312) with the PowerDesigner Application object. For example, to access the model with focus using a PowerDesigner application object called PD, use the following syntax:

```
PD.ActiveModel
```

- Specify object types whenever possible. For example, instead of simply using `Dim cls`, you should use:

```
Dim cls as PdOOM.Class
```

If your model contains shortcuts, we recommend that you use the following syntax to avoid runtime errors when the target model is closed:

```
Dim obj as PdCommon.IdentifiedObject
```

- Adapt the object class ID syntax to the language when you create object. For VBScript, VBA and VB and other languages that support enumeration defined outside a class, you can use the syntax:

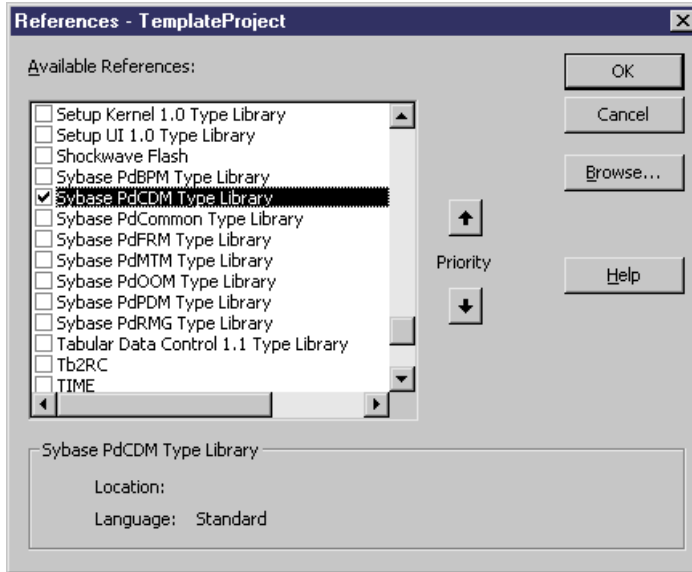
```
Dim cls as PdOOM.Class
Set cls = model.CreateObject(PdOOM.cls_Class)
```

For C# and VB.NET, you can use the following syntax (where `PdOOM_Classes` is the name of the enumeration):

```
Dim cls As PdOOM.Class
Set cls = model.CreateObject(PdOOM.PdOOM_Classes.cls_Class)
```


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.

- Add references to the object type libraries you need to use. For example, in a VBA editor, select **Tools > References**:



This script is launched from outside PowerDesigner, creates an instance of the PowerDesigner Application object, and then uses it to create two OOMs through OLE Automation:

```
'* 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."
ElseIf 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

```

OLE Automation samples for different languages are provided in the OLE Automation directory within your PowerDesigner installation directory.

Creating an ActiveX Add-in

You can create ActiveX add-ins to provide additional features to PowerDesigner, and call them through menu items.

To operate as a PowerDesigner add-in, the ActiveX add-in must implement the `IPDAddIn` interface, which defines the following methods, invoked by PowerDesigner to dialog with menus and execute the commands defined by the add-in:

- `HRESULT Initialize([in] IDispatch * pApplication)` and `HRESULT Uninitialize()` - The `Initialize()` method initializes communication between PowerDesigner and the add-in. PowerDesigner provides a pointer to its application object, defined in the `PdCommon` type library, which allows you to access the PowerDesigner environment (output window, active model etc.). The `Uninitialize()` method is called when PowerDesigner is closed to release all global variables and clean all references to PowerDesigner objects.
- `BSTR ProvideMenuItems([in] BSTR sMenu, [in] IDispatch *pObj)` - is invoked each time PowerDesigner needs to display a menu, and returns an XML text that describes the menu items to display. It is called once without an object parameter at the initialization of PowerDesigner to fill the **Import** and **Reverse** menus. 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 DTD for menu definition is as follows:

```

<!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>

```

For example:

```
ProvideMenuItems ("Object", pModel)
```

returns the following text:

```
<Menu>
<Popup Caption="&Perforce">
  <Command Name="CheckIn" Caption="Check &In"/>
  <Separator/>
  <Command Name="CheckOut" Caption="Check &Out"/>
</POPUP>
</MENU>
```

- `BOOL IsCommandSupported([in] BSTR sMenu, [in] IDispatch * pObject, [in] BSTR sCommandName)` - allows you to dynamically disable commands defined in a menu. The method must return true to enable a command and false to disable it.
- `HRESULT DoCommand(in BSTR sMenu, in IDispatch *pObj, in BSTR sCommandName)` - implements the execution of a command designated by its name. For example:

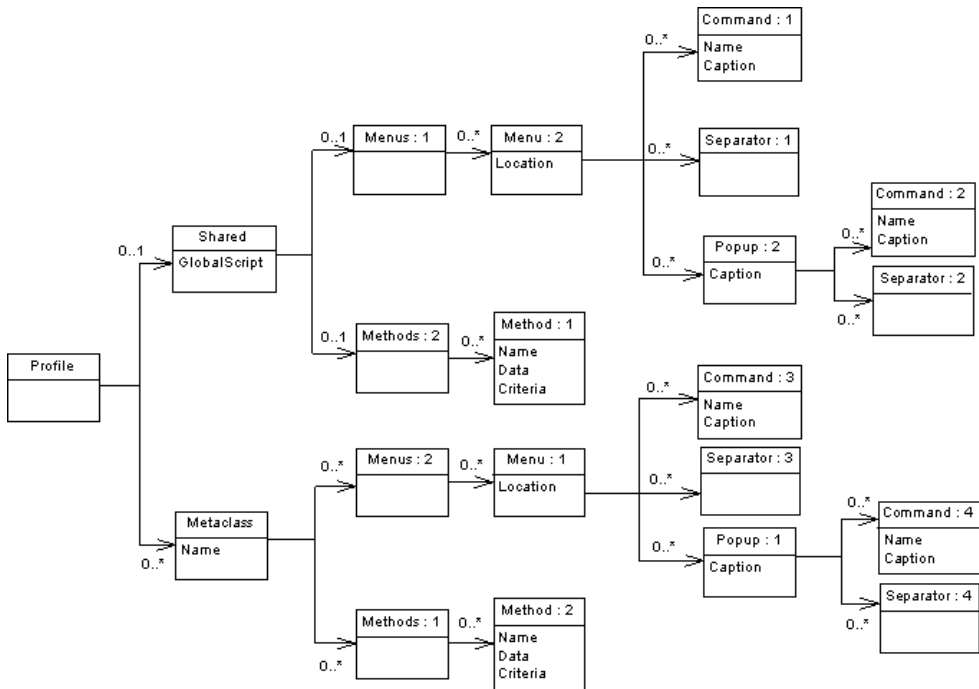
```
DoCommand ("Object", pModel, "CheckIn")
```

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 > Modeling with PowerDesigner > Customizing Your Modeling Environment > General Options > Add-Ins*).

Creating an XML File Add-in

You can create XML add-ins to group multiple commands for calling executable programs or VB scripts and add them to PowerDesigner menus.

The following illustration helps you understand the XML file structure:



Note: The DTD is available at `PD_install_dir\Add-ins\XMLAddins.dtd`.

The `Profile` is the root element of the XML file add-in descriptor and can contain:

- A `Shared` element - which defines the menus that are always available and their associated methods, along with a `GlobalScript` attribute, which can contain a global script for shared functions.
- One or more `Metaclass` elements - which define commands and menus for a specific metaclass, identified by its public name prefixed by its Type Library public name.

Both these elements can contain sub-elements as follows:

- `Menus` contains `Menu` elements that specify a location, which can be one of:
 - `FileImport` - shared only
 - `FileExport` - metaclass only
 - `FileReverse` - shared only
 - `Tools`
 - `Help`
 - `Object` - metaclasses only (default)

Each `Menu` element can contain:

- A `Command` element - whose `Name` must be equal to the name of a `Method`, and whose `Caption` defines the name of the command that appears in the menu.

- A Separator element - which indicates that you want to insert a line in the menu.
- A Popup element - which defines a sub-menu item that may in turn contain commands, separators, and popups.
- Methods contains Method elements, which define the methods used in the menus, and which are defined by a name and a VBScript. A method defined under a metaclass has the current object as a parameter. Inheritance is taken into account, so that a menu defined on the metaclass PdCommon.NamedObject will be available on PdOOM.Class.

The following example defines two menu items for the Perforce repository and the methods that are called by them:

```
<?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>
```

The following example defines a global script which is referenced by a method defined under a metaclass:

```
<?xml version="1.0" encoding="UTF-8"?>
<Profile>
  <Shared>
    <GlobalScript>
      Option Explicit
      Function Print (obj)
        Output obj.classname & " " & 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>

```

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 > Modeling with PowerDesigner > Customizing Your Modeling Environment > General Options > Add-Ins*).

Launching Scripts and Add-Ins from Menus

You can extend PowerDesigner menus to add commands to call scripts defined in resource files or externally and to launch executables and ActiveX add-ins. XML add-ins can be used to group and organize multiple commands. You can extend the **File**, **Tools**, and **Help** menus, and the contextual menus available on objects in the Browser and diagrams.

You can modify PowerDesigner menus in the following ways:

- Custom commands - are defined directly in PowerDesigner and can call executable programs or VB scripts (see *Adding Commands to the Tools Menu* on page 339).
- Menu and method extensions – are specified in a DBMS or language definition or extension file and define commands for a specific target or model type (see *Menus (Profile)* on page 81).
- ActiveX Add-Ins – are written in languages such as VB, C#, C++ or any language supporting COM, and permit more complex interactions with PowerDesigner, such as enabling and disabling menu items based on object selection, and interaction with the windows display environment (see *Creating an ActiveX Add-in* on page 334).

Note: The XML syntax used to define menus in an ActiveX or XML add-in is the same as that used in the creation of a menu extension, and you can use the resource editor menu XML page (see *Menus (Profile)* on page 81) to help you construct the syntax for your add-ins.

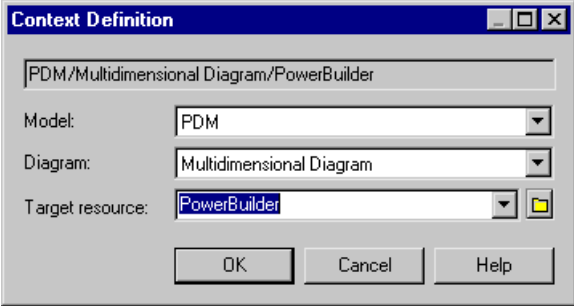
- XML Add-Ins – define multiple commands to call executable programs or VB scripts. Commands linked to the same applications (for example, ASE, IQ etc.) should be gathered into the same XML file (see *Creating an XML File Add-in* on page 335).

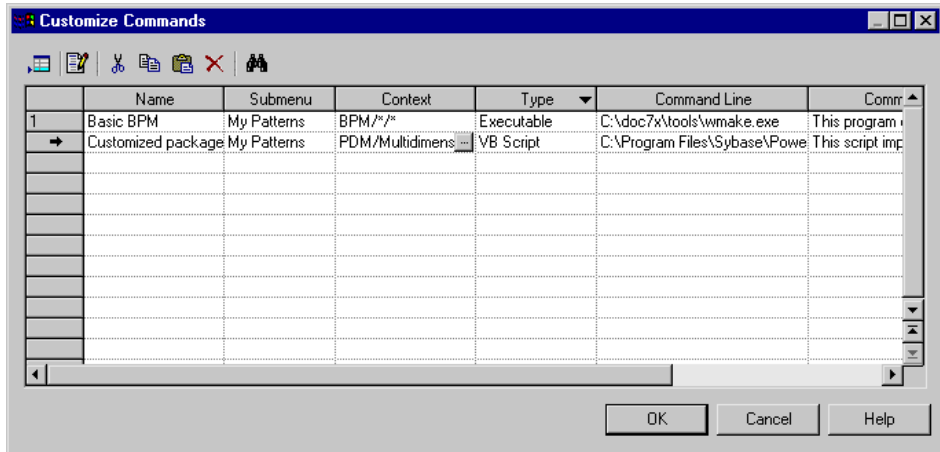
Adding Commands to the Tools Menu

You can create your own menu items in the PowerDesigner Tools menu to access PowerDesigner objects using your own scripts or executable programs. You can define up to 256 commands in the **Customize Commands** dialog, and control the contexts (model, diagram, and target type) in which they appear.

1. Select **Tools > Execute Commands > Customize Commands** and click the **Add a row** tool.
2. Enter the following properties:

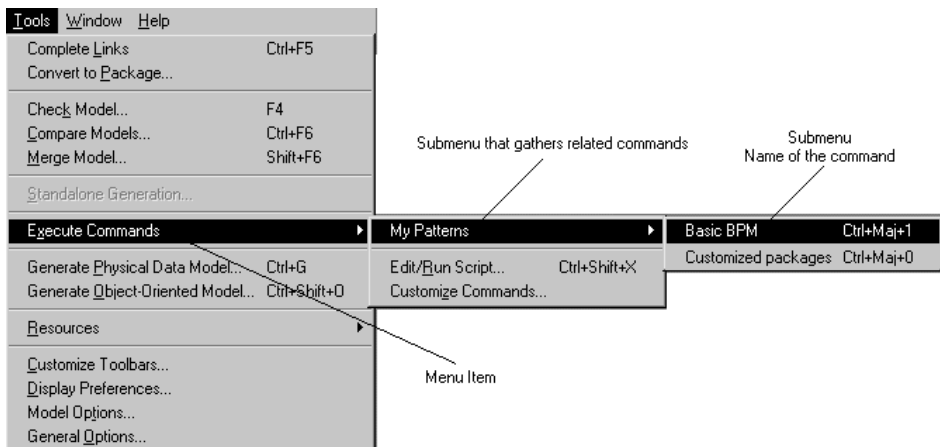
| Property | Description |
|----------|--|
| Name | Specifies the name of the command that will appear in the menu. Names must be unique and can contain a pick letter (&Generate Java will appear as Generate Java) |
| Submenu | Specifies a submenu in which to place the command. You can enter your own or select one of: <ul style="list-style-type: none"> • <None> - directly under Tools > Execute Commands • Check Model • Export • Generation • Import - also appears under File > Import • Reverse - also appears under File > Reverse-Engineer |

| Property | Description |
|-----------------|---|
| Context | <p>Specifies when the command is available. By default the command is available at all times (*/*/*). Click the ellipsis button to restrict the display of the command to a specific:</p> <ul style="list-style-type: none"> • Model type - for example OOM/*/* • Model and Diagram type - for example OOM/Class diagram/* • Model, Diagram, and Target type - for example OOM/Class diagram/Java. By default, the list contains extensions available for the chosen model type. Click the Path tool to navigate to another folder containing extensions or DBMS or language definition files.  |
| Type | Specifies whether the command will launch an executable or VBScript. |
| Command Line | Specifies the path to the executable or script file to run. Click the ellipsis button to navigate to a file. If your file is a VBScript, you can review or edit the script by clicking the Edit With tool in the toolbar. |
| Comment | Specifies text that is displayed in the status bar when you select the command. |
| [S]how in Menu | Specifies that the command should be displayed. Deselect this field to hide the command while retaining its definition. |
| Accelerator Key | Associates one of ten reserved keyboard shortcuts Ctrl-Shift-0 to Ctrl-Shift-9 with the command. |



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

Your command is now available under **Tools > Execute Commands**.



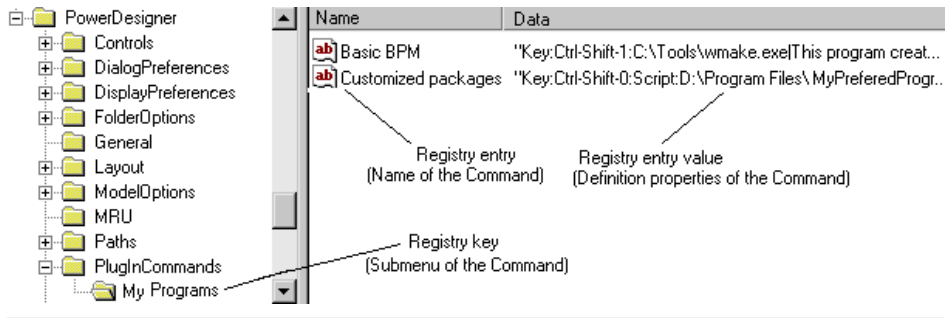
Note: Customized Commands are saved by default in the Registry at `HKEY_CURRENT_USER\Software\Sybase\PowerDesigner\PlugInCommands\submenu` and are available only to the user defining them. To make them available to all users, create an entry at the same location under `HKEY_LOCAL_MACHINE`.

The name of the entry is the name of the command, and its value takes the following syntax, in which only the **commandline** parameter is mandatory and must be terminated by a | (pipe) character

```
[Hide:] [Key: accelerator:] [Script:] commandline [ | comment]
```

If you want to insert a pipe within a command, you must escape it with a second pipe.

CHAPTER 7: Scripting PowerDesigner



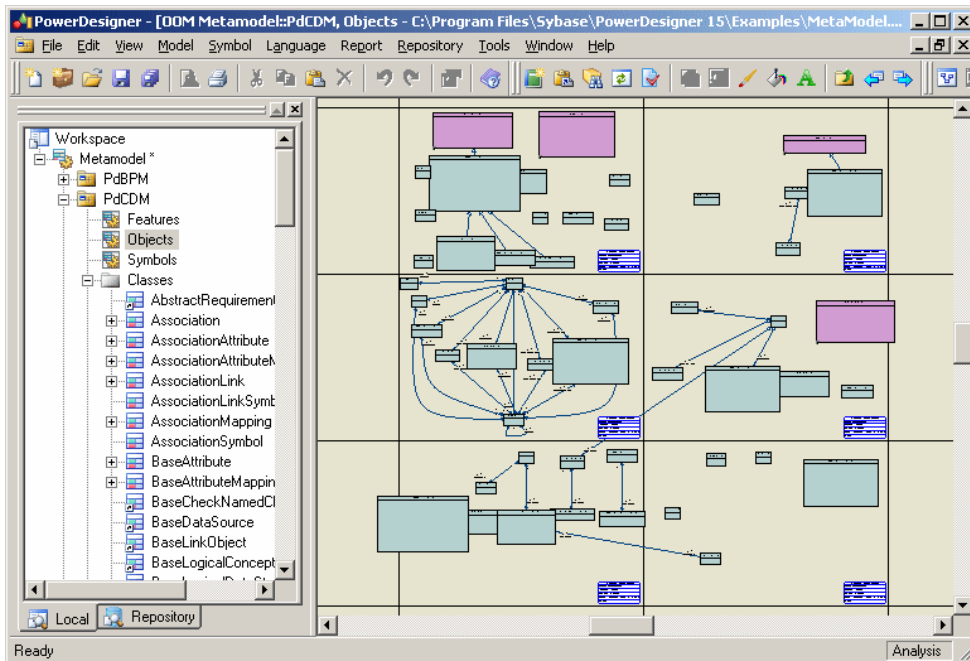
The PowerDesigner Public Metamodel

The PowerDesigner public metamodel is an abstraction of the metadata for all the PowerDesigner models, describing the elements of a model, and the syntax and semantics of their manipulation.

You can review the public metamodel in PowerDesigner by opening `install_dir\Examples\MetaModel.oom`, and find exhaustive documentation of all the metamodel objects, collections, and methods available via scripting, by selecting **Help > Metamodel Objects Help** (see *Using the Metamodel Objects Help File* on page 347).

This OOM and help file help you understand the structure of your models, especially when working with:

- Generation Template Language (GTL) templates (see *Chapter 5, Customizing Generation with GTL* on page 245).
- VB scripts (see *Chapter 7, Scripting PowerDesigner* on page 305).
- PowerDesigner XML model files (see *PowerDesigner Model File Format* on page 348).



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
- PdGLM - Glossary 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

Navigating in the Metamodel

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

Each metaclass has a name, contains zero or more attributes and assumes zero or more roles in associations with other classes, which allow you to identify collections. 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, which is the unique identifier of the object in a model library or

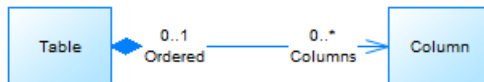
package. Public names are referenced in PowerDesigner XML model files and when using GTL and scripting. The public name often matches the object's name in the PowerDesigner interface, but where the two diverge, the public name must be used in scripts and GTL templates.

- *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/Concrete classes* - correspond to objects displayed in the interface. They have their own attributes and behaviors in addition to those they inherit from abstract classes through generalization links. For example, `NamedObject` is an abstract class, which contains standard attributes like `Name`, `Code`, `Comment`, `Annotation`, and `Description`, which are inherited by most PowerDesigner design objects.
- *Class attributes* - are object properties. 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* - express the semantic connections between classes. In the association property sheet, the roles carry information about the end object of the association. PowerDesigner objects are linked to other objects using collections, and the role at the other end of the association gives the name of the collection for an object. For example, `NamedObject` has a collection of business rules called `AttachedRules`, and `BusinessRule` has a collection of objects called `Objects`:



When associations have two roles, only the collection with the *navigable* role will be saved in the XML file. In the case, only the `AttachedRules` collection is saved.

- *Compositions* - express an association where the children live and die with the parent and, when the parent is copied, the child is also copied. For example, `Table` has a composition association with the `Column` class:

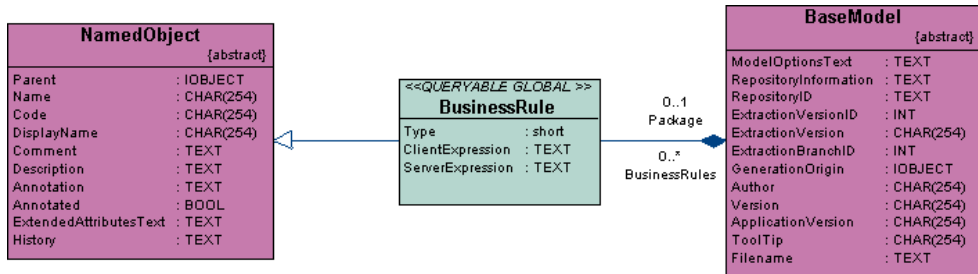


- *Generalizations* - show the *inheritance* links existing between a more general, usually abstract, class and a more specific, usually instantiable, class. The more specific class inherits from the attributes of the more generic class, these attributes are called derived attributes. For example, `Class` inherits from `Classifier`



Each diagram shows classes the connections between metaclasses via associations and generalizations. Classes in *green* are defined in the current diagram, while classes in *purple* are present only to provide context. To investigate a purple class, right-click it and select **Related Diagrams > diagram** to open the diagram where it is defined.

In the following example, `BusinessRule` is being defined, while `NamedObject` and `BaseModel` are present only to show inheritance and composition links:



Double-click any class to show its property sheet and review the following tabs:

- **General** - provides the public name in the **Name** and **Code** fields, a **Comment** providing a brief description of the class, and shows whether it is **Abstract**.

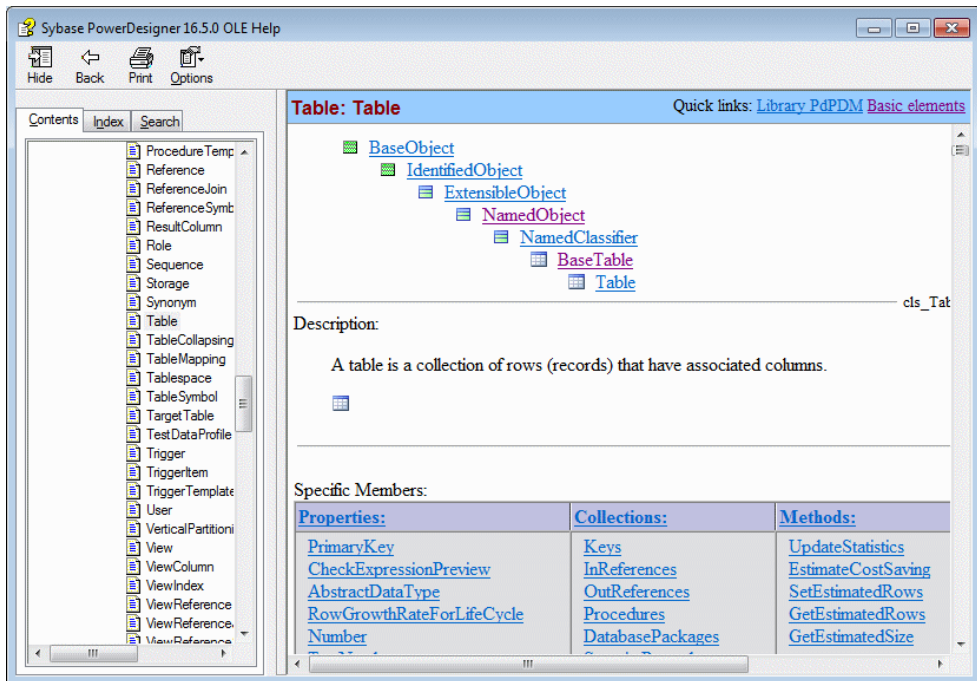
Note: Objects, such as `RepositoryGroup` that do not support scripting bear the `<<notScriptable>>` stereotype.

- **Attributes** - lists the properties defined directly on the class, but not those that it inherits via any parent classes.
- **Associations** - lists the migrated associations for the class, which represent collections. The **Role B** column lists the collections for the class, while the **Role A** column lists the collections in which the class figures.
- **Operations** - lists the methods available for scripting.
- **Dependencies** - contains the following sub-tabs (among others):
 - **Associations**
 - **Generalizations** - lists the generalization links where the current class is the child and inherits attributes from a parent class.
 - **Specializations** - lists the generalization links where the current class is the parent and its children inherit attributes from it.
 - **Shortcuts** - lists the shortcuts created for the current object.
- **Notes** - may include further information on the **Description** or **Annotation** sub-tabs.

Using the Metamodel Objects Help File

PowerDesigner provides documentation of the metamodel available from **Help > Metamodel Objects Help**.

The file can be opened from the **Edit/Run Script** dialog (see *Running Scripts in PowerDesigner* on page 307) or from a metaclass in a resource file (see *Metaclasses (Profile)* on page 33) by clicking the **Find in MetaModel Help** button or pressing Ctrl+F1. It can also be opened from any object property sheet by pressing Ctrl+F1 or clicking the **Property Sheet Menu** button and selecting **Find in MetaModel Help**.



The three top-level nodes contain the following documentation:

| Nodes | What you can find... |
|----------------|---|
| Basic Elements | Provides general information on: <ul style="list-style-type: none"> • Collections of objects - provide the principal way of navigating the metamodel (see <i>Browsing and Modifying Collections (Scripting)</i> on page 314). • Structured Types - used for positioning symbols in diagrams (see <i>Displaying, Formatting, and Positioning Symbols (Scripting)</i> on page 320). • Global properties, constants, and functions - provide entry points for scripting (see <i>Manipulating Models, Collections, and Objects (Scripting)</i> on page 312). |
| Libraries | Provides exhaustive documentation of all scriptable properties, collections, and methods for metamodel objects, organized by module. |
| Appendix | Includes an expandable hierarchy showing all the metaclasses in the PowerDesigner metamodel, a VBScript code sample, and a list of the class ID constants used to identify objects in certain contexts (see <i>Accessing and Modifying Objects and Properties (Scripting)</i> on page 316). |

To obtain information about the properties, collections and methods available for a particular metaclass, navigate to it under the Libraries category, or locate it in the index. All properties, collections, and methods are listed in the index.

Each metaclass shows the hierarchy of ancestors from which it is descended and inherits. After a brief description and symbol, it then lists:

- Specific Members - a table which lists the properties, collections, and methods defined directly on this metaclass
- Full definition - which lists, in separate tables, the properties, collections, and methods inherited from each of its ancestors. For example, the Table metaclass (located at `Libraries\PdPDM\Table`) inherits members from:
 - PdCommon.BaseObject
 - PdCommon.IdentifiedObject
 - PdCommon.ExtensibleObject
 - PdCommon.NamedObject
 - PdCommon.NamedClassifier
 - PdPDM.BaseTable
 - PdPDM.View

PowerDesigner Model File Format

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.

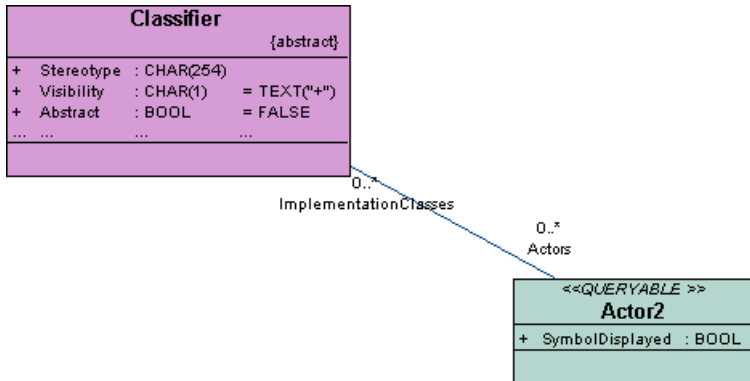
PowerDesigner checks each object and drills down the collections of this object to define each new object and collection in these collections, and so on, until the process finds terminal objects that do not need further analysis.

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:

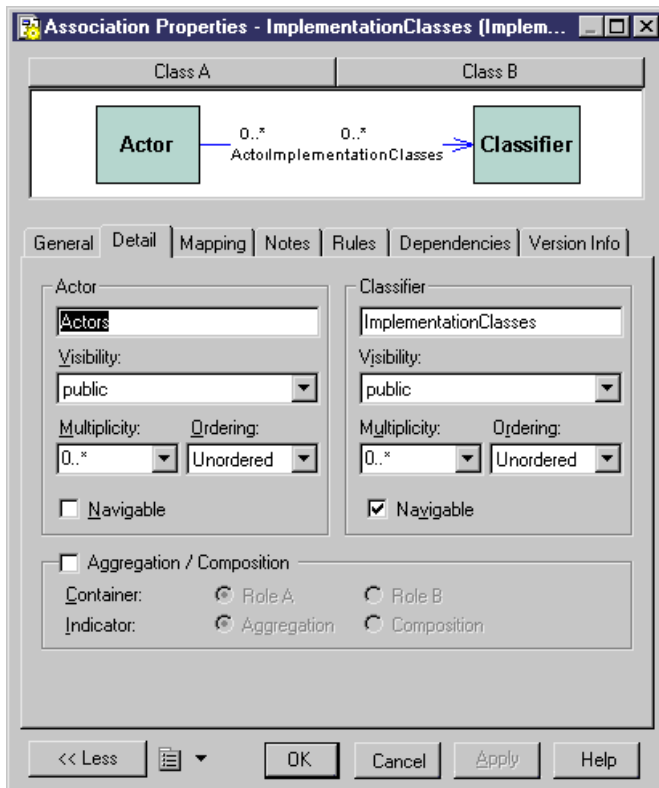
- Each PowerDesigner object can have several collections corresponding to other objects to interact with, these collections are represented by the associations existing between objects. The *roles* of the associations (aggregations and compositions included) correspond to the collections of an object.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



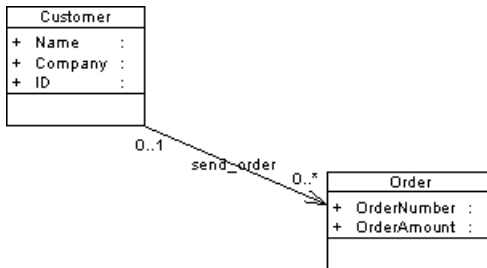
If you display the association property sheet, you can see that the Navigable check box is selected for role ImplementationClass, which means that only collection ImplementationClass will be saved in file.



- Attributes with the *OBJECT* 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

In this example, we will explore the structure of a simple OOM model file containing two classes and one association.



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>
```


Index

- %(x,y)% subtraction operator 252
 - ! power evaluation operator 252
 - != not equal to operator 252
 - ? existence operator 252
 - .foreach_item
 - example 27
 - .xems
 - See extension files
 - [] operators 213
 - [] conditional block 250
 - * dereferencing operator 252
 - %(x,y)% multiplication operator 252
 - %(x,y)% division operator 252
 - ./ macro 267
 - \\ escape sequence 256
 - \\n escape sequence 256
 - \\t escape sequence 256
 - %&(x,y)% logical bitfield and operator 252
 - && logical AND operator 252
 - %% escape sequence 256
 - + visibility operator 252
 - %(x,y)% addition operator 252
 - < less than operator 252
 - <= less than or equal to operator 252
 - = assignment operator 252
 - == equal to operator 252
 - > greater than operator 252
 - >= greater than or equal to operator 252
 - || logical OR operator 252
- A**
- A formatting option (align left) 248
 - .abort_command macro 265
 - abstract classes 344
 - abstract data type attributes
 - AllowedADT 176
 - abstract data types
 - ADTComment 174
 - AllowedADT 174
 - EnableAdtOnColn 174
 - EnableAdtOnDomn 174
 - Install 174
 - PDM variables 224
 - Remove 174
 - ActiveDiagram global property 312, 320
 - %ActiveModel% global variable 251
 - ActiveModel global property 312, 313
 - ActiveSelection global collection 312
 - ActiveWorkspace global property 312, 322
 - ActiveX
 - add-in 334
 - DoCommand 334
 - Initialize 334
 - IsCommandSupported 334
 - method 334
 - ProvideMenuItems 334
 - Uninitialize 334
 - Add 144
 - add-ins 305
 - ActiveX 334
 - launching 338
 - XML file 335
 - Add() method 314
 - AddColIndex 160
 - AddColnChck 152
 - AddColnCheck 152
 - adding items in resource files 6
 - AdditionalDataTypes 110
 - AddJoin 185
 - AddMetaExtension() method 329
 - AddObjects() method 321
 - AddQuote 136
 - AddSource() method 324
 - AddTableCheck 148
 - ADTComment 174
 - AfterCreate 144, 189
 - AfterDatabaseGenerate event handler 75, 133
 - AfterDatabaseReverseEngineer event handler 75, 133
 - AfterDrop 144
 - AfterModify 144
 - .AKCOLN PDM macro 235
 - AKeyComment 164
 - All Attributes and Collections tab 300
 - All Classes tab 299
 - All Report Titles tab 297
 - .ALLCOL PDM macro 236
 - AllowedADT 148, 174, 176
 - AllowNullableColn 164

Index

- AltEnableAddColnChk 152
- Alter 144
- AlterDBIgnored 144
- AlterFooter 138
- AlterHeader 138
- AlterStatementList 144
- AlterTableFooter 148
- AlterTableHeader 148
- ASE
 - PDM variables 227
- associations 344
- AttachLinkObject() method 320
- AttachObject() method 320
- attribute icon sets 46
- attribute value icons 46
- attributes 344
 - creating from a property sheet 15
- Attributes collection 330
- auto-attach 14
- autofixes 74

- B**
- BasicDataTypes 110
- BeforeCreate 144
- BeforeCreateDatabase 172
- BeforeDatabaseGenerate event handler 75, 133
- BeforeDatabaseReverseEngineer event handler 75, 133
- BeforeDrop 144
- BeforeModify 144
- BeginTransaction() global function 312
- Bind 152, 173, 187, 188
- BinDefault 173
- .block macro 265
- BlockComment 135
- BlockTerminator 135
- .bool macro 266
- booleans
 - .bool macro 266
 - testing in GTL 266
- .break macro 266

- C**
- calculated collections 50
- CancelTransaction() global function 312
- CanLinkKind event handler 75
- cases
 - changing in GTL 277
 - .lowercase macro 277
 - .uppercase macro 277
- CaseSensitivityUsingQuote 136
- .change_dir macro 266
- CharFunc 140
- CheckNull 152
- CheckOnCommit 166
- CheckOut() method 325
- Choreography category
 - process language 109
- class attributes 344
- classes 344
- Clear() method 314
- .CLIENTEXPRESSION macro 242
- CloseDatabase 172
- Cluster 160
- code property
 - .convert_code macro 267
 - converting in GTL 267
- .collection macro 278
- collections
 - accessing first item 247
 - Add() method 314
 - calculated collections 50
 - Clear() method 314
 - .collection macro 278
 - Count keyword 247
 - Count property 314
 - counting members 247
 - CreateNew() method 314, 318
 - CreateNewAt() method 314
 - extended collections 48
 - extended compositions 48
 - First keyword 247
 - .foreach_item macro 247, 271
 - GetCalculatedCollection() method 329
 - GetCollectionByStereotype() method 329
 - GetExtendedCollection() method 329
 - Insert() method 314
 - IsEmpty keyword 247
 - Item property 314
 - iterating over in GTL 271
 - Kind property 314
 - MetaCollection property 314
 - modifying by script 314
 - Move() method 314
 - Outer scope 255
 - Parent scope 255
 - Remove() method 314

- returning by GTL 278
 - scope 255
 - Source property 314
 - testing for members 247
- ColnDefaultName 177
- ColnRuleName 177
- ColumnComment 152
- columns
 - AddColnChck 152
 - AddColnCheck 152
 - AltEnableAddColnChk 152
 - Bind 152
 - CheckNull 152
 - ColumnComment 152
 - ConstName 152
 - DefineColnCheck 152
 - DropColnChck 152
 - DropColnComp 152
 - EnableBindRule 152
 - EnableComputedColn 152
 - EnableDefault 152
 - EnableIdentity 152
 - EnableNotNullWithDflt 152
 - MaxConstLen 152
 - ModifyColnComp 152
 - ModifyColnDflt 152
 - ModifyColnNull 152
 - ModifyColumn 152
 - null values 159
 - NullRequired 152, 159
 - PDM variables 217
 - Permission 152
 - Rename 152
 - SqlChckQuery 152
 - SqlPermQuery 152
 - SqlStatistics 152
 - Unbind 152
 - variables 235
- commands
 - creating custom commands 338, 339
- .comment macro 267
- comments
 - . // 267
 - .comment 267
 - in GTL 267
- Commit 140
- comparing resource file 7
- Complement language generation 14, 89
- composite physical options 209
- ConceptualDataTypes 110
- concrete classes 344
- conditional blocks 250
 - .block macro 265
 - in GTL 265
- conditional processing
 - GTL 275
 - .if macro 275
- connecting to databases by script 326, 328
- ConnectToDatabase() method 328
- Consolidate() method 325
- ConsolidateNew() method 325
- Constants category
 - object language 110
- ConstName 148, 152, 163, 164, 166
- constraints
 - PDM variables 217
- conversion tables 1
 - .convert_code macro 267
 - .convert_name macro 267
- ConvertFunc 140
- copying resource files 7
- Count keyword 247
- Count property 314
- Create 144
 - .create_path macro 266
- CreateBeforeKey 160
- CreateBody 189
- CreateDefault 173
- CreateFunc 180
- CreateModel() global function 312
- CreateModel() method 313
- CreateNew() method 314, 318
- CreateNewAt() method 314
- CreateObject() method 318
- CreateReport() method 325
- CreateSelection() method 321
- CreateShortcut() method 323
- creating data sources by script 324
- creating mappings by script 324
- creating metaclasses from stereotypes 39
- creating resource files 7
- criteria 40
- csv (conversion tables) 1
- %CurrentDate% global variable 251
- %CurrentUser% global variable 251
- custom checks 71
 - autofixes 74
 - example 21, 73, 74

Index

- script 73
- custom commands
 - adding to menus 338, 339
- custom properties
 - See extended attributes
- custom symbols 70
 - example 19
- CustomFunc 180
- CustomProc 180

D

- D formatting option (interface values) 248
- DashStyle property 320
- data sources
 - AddSource() method 324
 - creating by script 324
- Data Type category (DBMS) 197
- data types 45, 112, 197
- database package templates 120
- database security
 - PDM variables 230
- database synchronization
 - PDM variables 227
- databases
 - AfterCreate 122
 - BeforeCreate 122
 - BeforeCreateDatabase 172
 - CloseDatabase 172
 - connecting by script 326, 328
 - ConnectToDatabase() method 328
 - database package templates 120
 - EnableManyDatabases 172
 - estimating size 202, 205
 - EX keyword 128
 - GenerateDatabase() method 326
 - GenerateTestData() method 326
 - generating by script 326
 - generating test data by script 326
 - generation 120–122, 125, 133, 200
 - GetPackageOptions() method 326, 328
 - live connection 125
 - ModifyDatabase() method 326
 - object generation order 142
 - OpenDatabase 172
 - PDM variables 234
 - physical options 129, 207
 - procedure templates 120
 - reverse engineering 120, 124, 125, 128, 129, 131–133

- reverse-engineering by script 328
- ReverseDatabase() method 328
- ReversedQueries 128
- ReversedStatements 124
- scripts 121, 124
- trigger template items 120
- trigger templates 120
 - See also DBMS definition files
- DataHandling category
 - process language 109
- DataTypes category
 - XML language 112
- date formats 137
- DateFormat 136
- DateFunc 140
- dates
 - %CurrentDate% global variable 251
- DateTimeFormat 136
- DB package cursors 190
- DB package exceptions 190
- DB package pragmas 190
- DB package types 190
- DB package variables 190
- DB packages
 - AfterCreate 189
 - CreateBody 189
 - PDM variables 228
- DBMS definition files 1, 119
 - [] operators 213
 - abstract data type attributes 176
 - abstract data types 174, 224
 - AfterCreate 122
 - ASE 227
 - BeforeCreate 122
 - columns 152, 159, 217
 - constraints 217
 - Data Type category 197
 - database package templates 120
 - database security 230
 - database synchronization 227
 - databases 172, 234
 - date formats 137
 - DB package cursors 190
 - DB package exceptions 190
 - DB package pragmas 190
 - DB package types 190
 - DB package variables 190
 - DB packages 189, 228
 - DBMS triggers 184

- defaults 193, 231
- dimensions 196, 232
- domains 173, 217
- EnableOption 142
- estimating database size 202, 205
- EX keyword 128
- extended attributes 200, 207
- extended objects 197, 233
- extensions 207, 210
- File category 138
- Format category 136, 137
- forms 210
- General category 134
- generation 120–122, 125, 133, 200
- GenerationOrder 142
- GetEstimatedSize 202, 205
- groups 187
- index columns 220
- indexes 160, 220
- introduction 119
- join indexes 185, 227
- keys 163, 164, 219
- Keywords category 140
- live database connection 125
- live database generation 206
- MaxConstLen 142
- Objects category 142, 144, 148, 152, 159, 160, 163, 164, 166, 169, 171–174, 176, 177, 180, 181, 184–197
- ODBC category 206
- parameters 191
- PDM macros 212, 235–243
- PDM variables 212, 213, 215–217, 219, 220, 222–224, 227, 228, 230–234
- permissions 192
- physical options 129, 207, 209, 210
- Physical Options (Common) tab 207
- Physical Options tab 207
- primary keys 163
- privileges 191
- procedure templates 120
- procedures 180, 222, 234
- Profile category 200
- properties 119
- qualifiers 185
- reference columns 220
- references 166, 220
- result columns 195
- reverse engineering 120, 124, 125, 128, 129, 131–133, 234
- ReversedQueries 128
- ReversedStatements 124
- roles 188
- rules 177, 223
- scripts 121, 124
- sequences 186, 224
- SQL category 135
- SQL Server 227
- storages 171, 224
- synonyms 187, 224
- Syntax category 135
- tables 144, 148, 216
- tablespaces 171, 224
- testing values 213
- time formats 137
- trigger template items 120
- trigger templates 120
- triggers 181, 222, 234
- users 177
- views 169, 216
- Web operations 194
- Web parameters 195
- Web services 194, 231
- DBMS triggers 184
- DclDelIntegrity 166
- DclUpdIntegrity 166
- Default association container 117
- default variable 144
- DefaultDataType 110
- defaults 193
 - PDM variables 231
- DefaultTriggerName 181
- DefIndexColumn 160
- DefIndexType 160
- .DEFINE PDM macro 236
- DefineColnCheck 152
- .DEFINEIF PDM macro 237
- DefineJoin 166
- DefineTableCheck 148
- DefOptions 144
- .delete macro 268
- Delete() method 321
- deleting items in resource files 6
- Delimiter 135
- dependencies 344
- dependency matrices 52
 - dependency paths 54

Index

- dependency paths 54
 - diagrams
 - ActiveDiagram global property 312, 320
 - AttachLinkObject() method 320
 - AttachObject() method 320
 - displaying symbols by script 320
 - dialog boxes
 - creating from forms 55
 - example 67
 - dimensions 196
 - PDM variables 232
 - directories
 - .change_dir 266
 - changing in GTL 266
 - domains
 - Bind 173
 - BinDefault 173
 - CreateDefault 173
 - EnableBindRule 173
 - EnableCheck 173
 - EnableDefault 173
 - EnableOwner 173
 - PDM variables 217
 - SqlListDefaultQuery 173
 - UddtComment 173
 - Unbind 173
 - UserTypeName 173
 - Drop 144
 - DropColnChck 152
 - DropColnComp 152
 - DropFunc 180
 - DropTableCheck 148
- ## E
- Edit/Run Script editor 307
 - editing resource files 6
 - embedding resource files 6
 - Enable 144
 - EnableAdtOnColn 174
 - EnableAdtOnDomn 174
 - EnableAlias 187
 - EnableAscDesc 160
 - EnableBindRule 152, 173
 - EnableChangeJoinOrder 166
 - EnableCheck 134, 173
 - EnableCluster 160, 163, 164, 166
 - EnableComputedColn 152
 - EnableConstName 134
 - EnableDefault 152, 173
 - EnableDtbsPrefix 136
 - EnablefKeyName 166
 - EnableFunc 180
 - EnableFunction 160
 - EnableIdentity 152
 - EnableIntegrity 134
 - EnableJidxColn 185
 - EnableManyDatabases 172
 - EnableMultiCheck 134
 - EnableMultiFile 138
 - EnableMultiTrigger 181
 - EnableNotNullWithDflt 152
 - EnableOption 142
 - EnableOwner 160, 173, 180, 181, 186
 - EnableOwnerPrefix 136
 - EndTransaction() global function 312
 - .error macro 269
 - .ERROR PDM macro 237
 - error messages 237
 - errors
 - .error macro 269
 - in GTL 269
 - escape sequences
 - \\ backslash 256
 - \n new line 256
 - \t tab 256
 - %% percent sign 256
 - EvaluateNamedPath() global function 312
 - Event 181
 - event handlers
 - AfterDatabaseGenerate 75, 133
 - AfterDatabaseReverseEngineer 75, 133
 - BeforeDatabaseGenerate 75, 133
 - BeforeDatabaseReverseEngineer 75, 133
 - CanCreate 75
 - CanLinkKind 75
 - GetEstimatedSize 75, 202, 205
 - Initialize 75
 - OnLanguageChanged 75
 - OnLanguageChangeRequest 75
 - OnLanguageChanging 75
 - OnModelClose 75
 - OnModelOpen 75
 - OnModelSave 75
 - OnNewFromTemplate 75
 - Validate 75
 - EventDelimiter 181
 - Events category
 - object language 110

- EX keyword 128
- examples
 - .foreach_item 27
 - creating a property sheet tab 61
 - creating custom check autofix 74
 - creating custom check script 73
 - creating custom checks 21
 - creating custom symbols 19
 - creating generated files 29
 - creating stereotypes 18
 - creating templates 27
 - extended attribute 15
 - extension files 15
 - extensions 15, 16, 18, 19, 21, 27, 29, 30
 - generated files 87
 - generation commands 114
 - generation options 113
 - generation tasks 114
 - including a form in a form 64
 - opening a dialog box from a form 67
 - opening a dialog from a menu 82
 - templates 87
 - XML model file format 351
- excluding metaclasses from models 33
- executables
 - .execute_command macro 269
 - launching with GTL 269
 - .execute_command macro 269
 - .execute_vbscript macro 270
- ExecuteCommand() global function 312
- exporting
 - extension files 14
 - extensions 14
- extended attribute types
 - type 45
- extended attributes 41, 207
 - accessing in other extension files 246
 - adding to forms 41
 - attribute value icons 46
 - creating from a property sheet 15
 - creating types for 45
 - data types 45
 - displaying in forms 57
 - example 15
 - generation 200
 - specifying objects as data types 48
- extended collections 48
 - displaying in forms 57
- extended compositions 48
- extended generation 89
- extended links 36
- extended model definitions
 - See extension files
- extended objects 36, 197
 - generation 133
 - PDM variables 233
 - reverse engineering 133
- extended sub-objects 36
- ExtendedLink metaclass 36
- ExtendedModelDefinitions collection 329
- ExtendedObject metaclass 36
- ExtendedSubObject metaclass 36, 48
- extension category 14
- extension files 1, 11
 - attaching to a model 12
 - auto-attach 14
 - category 14
 - Complement language generation 14
 - conflict resolution 11
 - creating 12, 16
 - embedded 12
 - example 15
 - exporting from a model 14
 - Generation category 14
 - properties 14
 - shared 12
 - trace mode 14
 - Transformation Profile category 14
 - See also extensions
- extensions 11, 207, 210
 - accessing by script 329
 - AddMetaExtension() method 329
 - attaching to a model 12
 - calculated collections 50
 - collections 329
 - criteria 40
 - custom checks 21, 71
 - custom symbols 19, 70
 - dependency matrices 52, 54
 - event handlers 75, 133, 202, 205
 - example 15, 16, 18, 19, 21, 27, 29, 30, 82
 - exporting from a model 14
 - extended attribute types 45
 - extended attributes 15, 41, 46, 48, 57, 200
 - extended collections 48, 57
 - extended compositions 48
 - extended links 36
 - extended objects 36

Index

- extended sub-objects 36
- ExtendedModelDefinitions collection 329
- forms 55, 57, 61, 64, 67
- generated files 29, 85, 87, 89, 245
- Generation category 11
- GetCalculatedCollection() method 329
- GetCollectionByStereotype() method 329
- GetExtendedAttribute() method 329
- GetExtendedAttributeText() method 329
- GetExtendedCollection() method 329
- global script 105
- in DBMS definition files 200
- menus 81, 338
- metaclasses 33
- methods 57, 79, 81, 338
- object generations 100, 102, 103
- object language definition files 117
- process language definition files 117
- Profile category 11
- SetExtendedAttribute() method 329
- SetExtendedAttributeText() method 329
- stereotypes 18, 37, 39, 329
- templates 27, 84, 87, 245
- transformation profiles 94
- transformations 92, 95
- UseAsMetaclass property 329
- XML imports 96, 97, 100
- XML language definition files 117

external applications

- ExecuteCommand() global function 312

F

File category (DBMS)

- AlterFooter 138
- AlterHeader 138
- EnableMultiFile 138
- Footer 138
- Header 138
- ScriptExt 138
- StartCommand 138
- TableExt 138
- TrgFooter 138
- TrgHeader 138
- TrgUsage1 138
- TrgUsage2 138
- TriggerExt 138
- Usage1 138
- Usage2 138

FindChildByCode() method 316

FindChildByName() method 316

FindChildByPath() method 316

First keyword 247

FKAutoIndex 166

- .FKCOLN PDM macro 235

FKKeyComment 166

folders

- creating by script 322

Footer 138, 160

- .FOREACH_CHILD PDM macro 238
- .FOREACH_COLUMN PDM macro 239
- .foreach_item macro 271
- .foreach_line macro 273
- .FOREACH_PARENT PDM macro 240
- .foreach_part macro 274

foreign key

- variable 241

Format category (DBMS)

- AddQuote 136
- CaseSensitivityUsingQuote 136
- date formats 137
- DateFormat 136, 137
- DateTimeFormat 136, 137
- EnableDtbsPrefix 136
- EnableOwnerPrefix 136
- IllegalChar 136
- LowerCaseOnly 136
- MaxScriptLen 136
- time formats 137
- TimeFormat 136, 137
- UpperCaseOnly 136

formatting options

- A (align left) 248
- D (interface values) 248
- H (hexadecimal) 215, 248
- L (lowercase) 215, 248
- LF (first character lowercase) 215, 248
- M (delete substring) 215, 248
- Q (double quotes) 215, 248
- q (single quotes) 215, 248
- T (trim whitespace) 215, 248
- U (uppercase) 215, 248
- UF (first character uppercase) 215, 248
- X (escape XML characters) 248

forms 210

- adding buttons 57
- adding controls 57
- creating dialog boxes 55
- creating property tabs 55

- dialog box example 67
- displaying extended attributes 57
- displaying extended collections 57
- form-in-form example 64
- property sheet example 61
- replacing property tabs 55

function based indexes 131

FunctionComment 180

G

General category (DBMS)

- EnableCheck 134
- EnableConstName 134
- EnableIntegrity 134
- EnableMultiCheck 134
- SqlSupport 134
- UniqConstName 134

generalizations 344

generated files 85, 87, 89, 245

- example 29

GenerateDatabase() method 326

GenerateHTML() method 325

GenerateRTF() method 325

GenerateTestData() method 326

generating database objects 142

generating databases by script 326

generating models 102

generating test data by script 326

generation 120

- extended generation 89
- %GenOptions% global variable 251
- live connection 125
- PDM extended objects 133
- script after 133
- script before 133
- scripts 121, 122

generation category 112

Generation category 11, 14

generation commands 112, 114

- .abort_command macro 265
- aborting 265
- GTL 265

generation options 112, 113

generation tasks 112, 114

Generation Template Language

- See GTL

GenerationOrder 142

%GenOptions% global variable 251

GetAttribute() method 316

GetAttributeText() method 316

GetCalculatedCollection() method 329

GetCollectionByStereotype() method 329

GetEstimatedSize 202, 205

GetEstimatedSize event handler 75

GetExtendedAttribute() method 329

GetExtendedAttributeText() method 329

GetExtendedCollection() method 329

GetMapping() method 324

GetMetaClassByPublicName() method 330

GetMetaMemberByPublicName() method 330

GetPackageOptions() method 326, 328

global script 71, 75, 79, 105

global variables 105

- %ActiveModel% 251
- %CurrentDate% 251
- %CurrentUser% 251
- %GenOptions% 251
- %NewUUID% 251
- %PreviewMode% 251

go to super-definition 5

GrantOption 191, 192

GroupFunc 140

groups

- Bind 187
- SqlListChildrenQuery 187
- SqlPermQuery 187
- Unbind 187

GTL 84, 85, 245

- aborting generation commands 265
- accessing extended attributes in other extension files 246
- breaking loops 266
- calling templates 257
- changing directory 266
- changing text case 277
- collections 247
- comments 267
- conditional blocks 250, 265
- conditional generation 275
- controlling user interaction 278
- converting names and codes 267
- creating generated files 245
- creating paths 266
- creating templates 245
- defining local variable and value types 279
- deleting substrings 268
- embedding VBScript 281
- errors 283

Index

- escape sequences 256
 - executing VBScript 270
 - extended attributes 246
 - formatting text 248
 - generated files 245
 - global variables 251
 - GTL operators 252
 - head string 250
 - inheritance 257
 - introduction 245
 - %IsShortcut% 256
 - iterating over a collection 271
 - iterating over lines in a text block 273
 - iterating over parts of a string 274
 - launching executables 269
 - line breaks 250
 - macros 264
 - metamodel extensions 262
 - new line 250
 - object properties 246
 - Outer scope 255
 - outputting unique lines 281
 - overloading templates 257
 - overriding templates 257
 - parameters 260
 - Parent scope 255
 - polymorphism 257
 - printing error messages 269
 - printing warning messages 269
 - properties 246
 - recursive templates 262
 - replacing substrings 268
 - returning collections by OID 278
 - returning objects by OID 278
 - scope 255
 - %Shortcut% 256
 - shortcuts 256
 - syntax errors 283
 - tail string 250
 - templates 245
 - testing boolean conditions 266
 - text blocks 250
 - translation errors 283
 - writing log messages 277
- GTL macros
- .// 267
 - .abort_command 265
 - .block 265
 - .bool 266
 - .break 266
 - .change_dir 266
 - .collection 278
 - .comment 267
 - .convert_code 267
 - .convert_name 267
 - .create_path 266
 - .delete 268
 - .error 269
 - .execute_command 269
 - .execute_vbscript 270
 - .foreach_item 247, 271
 - .foreach_line 273
 - .foreach_part 274
 - .if 275
 - .log 277
 - .lowercase 277
 - .object 278
 - .replace 268
 - .set_interactive_mode 278
 - .set_object 279
 - .set_value 279
 - .unique 281
 - .unset 279
 - .uppercase 277
 - .vbscript 281
 - .warning 269
- GTL operators
- ! power evaluation 252
 - != not equal to 252
 - ? existence 252
 - %(x,y)% subtraction 252
 - * dereferencing 252
 - %(x,y)% multiplication 252
 - %(x,y)% division 252
 - %&(x,y)% logical bitfield and 252
 - && logical AND 252
 - + visibility 252
 - %(x,y)% addition 252
 - < less than 252
 - <= less than or equal to 252
 - = assignment 252
 - == equal to 252
 - > greater than 252
 - >= greater than or equal to 252
 - || logical OR 252
- ## H
- H formatting option (hexadecimal) 215, 248

head string 250
 Header 138, 160
 HomeDirectory global constant 312

I

IdentifierDelimiter 135
 .if macro 275
 IllegalChar 136
 impact and analysis rule sets 1
 Implementation category
 process language 109
 .INCOLN PDM macro 240
 index columns
 PDM variables 220
 IndexComment 160
 indexes
 AddColIndex 160
 Cluster 160
 CreateBeforeKey 160
 DefIndexColumn 160
 DefIndexType 160
 EnableAscDesc 160
 EnableCluster 160
 EnableFunction 160
 EnableOwner 160
 Footer 160
 Header 160
 IndexComment 160
 IndexType 160
 MandIndexType 160
 MaxColIndex 160
 PDM variables 220
 SqlSysIndexQuery 160
 UniqName 160
 IndexType 160
 inheritance 257
 Insert() method 314
 Install 174
 instantiable classes 344
 InteractiveMode global property 312
 inverse collections 48
 IsEmpty keyword 247
 IsKindOf() global function 312
 %IsShortcut% 256
 Item property 314

J

.JOIN PDM macro 241

join indexes
 AddJoin 185
 EnableJidxColn 185
 JoinIndexComment 185
 PDM variables 227
 JoinIndexComment 185

K

keys
 AKeyComment 164
 AllowNullableColn 164
 ConstName 164
 EnableCluster 164
 MaxConstLen 164
 PDM variables 219
 primary keys 163
 SqlAkeyIndex 164
 UniqConstAutoIndex 164
 UniqInTable 164
 Keywords category (DBMS)
 CharFunc 140
 Commit 140
 ConvertFunc 140
 DateFunc 140
 GroupFunc 140
 ListOperators 140
 NumberFunc 140
 OtherFunc 140
 ReservedDefault 140
 ReservedWord 140
 Kind property 314

L

L formatting option (lowercase) 215, 248
 LF formatting option (first character lowercase)
 215, 248
 Libraries collection 330
 Library property 330
 line breaks
 controlling in GTL 250
 LineComment 135
 LineWidth property 320
 Linguistic Variables category 300
 link objects
 creating by script 318
 link symbols
 setting extremities by script 320

Index

- ListOperators 140
- live database generation 206
- local variables
 - defining in GTL 279
 - .set_object 279
 - .set_value 279
 - .unset 279
- Locked global property 312
- .log macro 277
- logs
 - .log macro 277
 - writing to in GTL 277
- loops
 - .break macro 266
 - breaking in GTL 266
- .lowercase macro 277
- lowercase 277
- LowerCaseOnly 136

- M**
- M formatting option (delete substring) 215, 248
- MandIndexType 160
- Mapping Editor 100, 103
 - XML imports 97
- mappings
 - creating by script 324
 - GetMapping() method 324
 - metamodel 97, 103
 - metamodel objects 101
 - properties 100
 - retrieving by script 324
 - SourceClassifiers collection 324
- MapToNamedPath() global function 312
- MaxColIndex 160
- MaxConstLen 142, 148, 152, 164, 166
- MaxDefaultLen 177
- MaxFuncLen 180
- Maxlen 144
- MaxScriptLen 136
- mcc (model category sets) 1
- MDA 92
- menus
 - creating custom commands in 338, 339
 - customizing via extensions 81, 338
 - customizing via XML add-ins 338
 - example 82
 - launching add-ins from 338
 - launching scripts from 338
- merging resource files 8
- metaclass 33
- Metaclass property 330
- metaclasses 33
 - adding to extension file 33
 - creating from stereotypes 39
 - creating new 36
 - excluding from model 33
 - ExtendedLink 36
 - ExtendedObject 36
 - ExtendedSubObject 36, 48
 - extending 33
 - subclassifying with criteria 40
 - subclassifying with stereotypes 37
- MetaCollection property 314
- metamodel
 - abstract classes 344
 - associations 344
 - attributes 344
 - Attributes collection 330
 - calculated attributes 262
 - calculated collections 262
 - class attributes 344
 - classes 344
 - concrete classes 344
 - dependencies 344
 - extending by script 329
 - generalizations 344
 - GetMetaClassByPublicName() method 330
 - GetMetaMemberByPublicName() method 330
 - GTL-specific extensions 262
 - instantiable classes 344
 - Libraries collection 330
 - Library property 330
 - Metaclass property 330
 - MetaModel global property 330
 - Metamodel Objects Help 343, 347
 - metamodel.oom 343
 - navigating 344
 - notScriptable stereotype 344
 - operations 344
 - Parent property 330
 - PdBPM 344
 - PdCDM 344
 - PdCommon 344
 - PdEAM 344
 - PdFRM 344
 - PdGLM 344
 - PdILM 344

- PdLDM 344
- PdMTM 344
- PdOOM 344
- PdPDM 344
- PdPRJ 344
- PdRMG 344
- PdRQM 344
- PdWSP 344
- PdXSM 344
- PowerDesigner 343
- public names 344
- PublicName property 330
- shortcuts 344
- specializations 344
- XML model file format 348, 351
- MetaModel global property 330
- metamodel objects
 - properties 101
- Metamodel Objects Help 343, 347
- metamodel.oom 343
- methods 79
 - adding to menus 81, 338
 - attaching to form buttons 57
- model category sets 1
- model checks
 - creating custom checks 71
- model file format
 - bin 348
 - DTD 348
 - XML 348, 351
- model generation 102
- models
 - %ActiveModel% global variable 251
 - ActiveModel global property 312
 - CreateModel() global function 312
 - CreateObject() method 318
 - creating by script 313
 - Models global collection 312
 - opening by script 313
 - OpenModel() global function 312
- Models global collection 312, 313
- ModifiableAttributes 144
- ModifyColnComp 152
- ModifyColnDflt 152
- ModifyColnNull 152
- ModifyColumn 152
- ModifyDatabase() method 326
- Move() method 314
- MoveToPackage() method 321

N

- name property
 - .convert_name macro 267
 - converting in GTL 267
- named paths
 - EvaluateNamedPath() global function 312
 - MapToNamedPath() global function 312
- Namings category
 - object language 110
- navigating in resource files 5
- new line 250
- NewPoint() global function 320
- %NewUUID% global variable 251
- .NMFCOL PDM macro 242
- not certified resource files 3
- notScriptable stereotype 344
- NullRequired 152, 159
- NumberFunc 140

O

- .object macro 278
- Object Attributes category 298
- Object container 117
- object generations 102
 - initialization scripts 100
 - mapping properties 100
 - mappings 103
 - post-processing scripts 100
- object language definition files 1
 - AdditionalDataTypes 110
 - BasicDataTypes 110
 - ConceptualDataTypes 110
 - Constants category 110
 - data types 110
 - Default association container 117
 - DefaultDataType 110
 - Events category 110
 - extensions 117
 - generation category 112
 - generation commands 112, 114
 - generation options 112, 113
 - generation tasks 112, 114
 - Namings category 110
 - Object container 117
 - profile category 117
 - properties 107
 - Settings category 110
- object permission profiles 1

Index

- object properties 246
 - accessing by script 316
 - modifying by script 316
- object selections
 - ActiveSelection global collection 312
 - AddObjects() method 321
 - CreateSelection() method 321
 - creating by script 321
 - MoveToPackage() method 321
 - Remove() method 321
- objects
 - accessing by script 316
 - creating by script 318
 - creating shortcuts by scripts 323
 - Delete() method 321
 - deleting by script 321
 - displaying in diagrams by script 320
 - FindChildByCode() method 316
 - FindChildByName() method 316
 - FindChildByPath() method 316
 - GetAttribute() method 316
 - GetAttributeText() method 316
 - IsKindOf() global function 312
 - .object macro 278
 - Outer scope 255
 - Parent scope 255
 - returning by GTL 278
 - scope 255
 - SetAttribute() method 316
 - SetAttributeText() method 316
 - Symbols collection 320
- Objects category (DBMS)
 - abstract data type attributes 176
 - abstract data types 174
 - Add 144
 - AfterCreate 144
 - AfterDrop 144
 - AfterModify 144
 - Alter 144
 - AlterDBIgnored 144
 - AlterStatementList 144
 - BeforeCreate 144
 - BeforeDrop 144
 - BeforeModify 144
 - columns 152, 159
 - Create 144
 - databases 172
 - DB package cursors 190
 - DB package exceptions 190
 - DB package pragmas 190
 - DB package types 190
 - DB package variables 190
 - DB packages 189
 - DBMS triggers 184
 - default variable 144
 - defaults 193
 - DefOptions 144
 - dimensions 196
 - domains 173
 - Drop 144
 - Enable 144
 - EnableOption 142
 - extended objects 197
 - GenerationOrder 142
 - groups 187
 - indexes 160
 - join indexes 185
 - keys 163, 164
 - MaxConstLen 142
 - Maxlen 144
 - ModifiableAttributes 144
 - Options 144
 - parameters 191
 - permissions 192
 - primary keys 163
 - privileges 191
 - procedures 180
 - qualifiers 185
 - references 166
 - result columns 195
 - ReversedStatements 144
 - roles 188
 - rules 177
 - sequences 186
 - SqlAttrQuery 144
 - SqlListQuery 144
 - SqlOptsQuery 144
 - storages 171
 - synonyms 187
 - tables 144, 148
 - tablespaces 171
 - triggers 181
 - users 177
 - views 169
 - Web operations 194
 - Web parameters 195
 - Web services 194
- ODBC category 206

- OLE
 - Locked global property 312
 - ShowMode global property 312
- OLE Automation 305, 332
- OnLanguageChanged event handler 75
- OnLanguageChangeRequest event handler 75
- OnLanguageChanging event handler 75
- OnModelClose event handler 75
- OnModelOpen event handler 75
- OnModelSave event handler 75
- OnNewFromTemplate event handler 75
- OpenDatabase 172
- opening resource files 3
- OpenModel() global function 312
- OpenModel() method 313
- operations 344
- Options 144
- OtherFunc 140
- Outer 255
- Output window
 - Output() global function 312
- Output() global function 312
- overloading
 - templates 257
- overriding
 - templates 257
- P**
- parameters 191, 260
- Parent 255
- Parent property 330
- paths
 - .create_path 266
 - creating in GTL 266
 - specifying for resource files 1
- PdBPM 344
- PdCDM 344
- PdCommon 344
- PdEAM 344
- PdFRM 344
- PdGLM 344
- PdILM 344
- PdLDM 344
- PDM macros 212
 - .AKCOLN 235
 - .ALLCOL 236
 - .CLIENTEXPRESSION 242
 - .DEFINE 236
 - .DEFINEIF 237
 - .ERROR 237
 - .FKCOLN 235
 - .FOREACH_CHILD 238
 - .FOREACH_COLUMN 239
 - .FOREACH_PARENT 240
 - .INCOLN 240
 - .JOIN 241
 - .NMFCOL 242
 - .PKCOLN 235
 - .SERVEREXPRESSION 242
 - .SQLXML 243
- PDM variables 212
 - [] operators 213
 - abstract data types 224
 - ASE 227
 - columns 217
 - constraints 217
 - database security 230
 - database synchronization 227
 - databases 234
 - DB packages 228
 - defaults 231
 - dimensions 232
 - domains 217
 - extended objects 233
 - formatting 215
 - index columns 220
 - indexes 220
 - join indexes 227
 - keys 219
 - procedures 222, 234
 - reference columns 220
 - references 220
 - reverse engineering 234
 - rules 223
 - sequences 224
 - SQL Server 227
 - storages 224
 - synonyms 224
 - tables 216
 - tablespaces 224
 - testing values 213
 - triggers 222, 234
 - views 216
 - Web services 231
- PdMTM 344
- PdOOM 344
- PdPDM 344
- PdPRJ 344

Index

- PdRMG 344
 - PdRQM 344
 - PdWSP 344
 - PdXSM 344
 - Permission 148, 152, 180
 - permissions
 - GrantOption 192
 - RevokeOption 192
 - physical options 210
 - composite options 209
 - default value 207
 - defining defaults in a DBMS file 144
 - defining in a DBMS file 144, 207
 - DefOptions DBMS item 144
 - extended attributes 207
 - list of values 207
 - Options DBMS item 144
 - Physical Options (Common) tab 207
 - Physical Options tab 207
 - reverse engineering 129
 - simple options 207
 - Physical Options (Common) tab 207
 - Physical Options tab 207
 - PkAutoIndex 163
 - .PKCOLN PDM macro 235
 - PKeyComment 163
 - platform-independent models 92
 - platform-specific models 92
 - polymorphism 257
 - Position property 320
 - PowerDesigner
 - metamodel 343
 - XML model file format 351
 - ppf (object permission profiles) 1
 - Preview tab
 - %PreviewMode% global variable 251
 - %PreviewMode% global variable 251
 - primary key
 - variable 241
 - primary keys
 - ConstName 163
 - EnableCluster 163
 - PkAutoIndex 163
 - PKeyComment 163
 - UseSpPrimKey 163
 - privileges
 - GrantOption 191
 - RevokeOption 191
 - System 191
 - procedure templates 120
 - ProcedureComment 180
 - procedures
 - CreateFunc 180
 - CustomFunc 180
 - CustomProc 180
 - DropFunc 180
 - EnableFunc 180
 - EnableOwner 180
 - FunctionComment 180
 - MaxFuncLen 180
 - PDM variables 222, 234
 - Permission 180
 - ProcedureComment 180
 - SqlPermQuery 180
 - process language definition files 1
 - Choreography category 109
 - DataHandling category 109
 - extensions 117
 - generation category 112
 - generation commands 112, 114
 - generation options 112, 113
 - generation tasks 112, 114
 - Implementation category 109
 - profile category 117
 - properties 107
 - Settings category 109
 - profile
 - See extension files
 - profile category
 - object language definition files 117
 - process language definition files 117
 - XML language definition files 117
 - Profile category 11
 - DBMS definition files 200
 - promoting a stereotype to metaclass 37
 - promoting a sterotype to metaclass 39
 - properties 246
 - property sheets
 - example 61
 - form-in-form example 64
 - property tabs
 - creating from forms 55
 - replacing by forms 55
 - public names 344
 - PublicName property 330
- ## Q
- Q formatting option (double quotes) 215, 248

q formatting option (single quotes) 215, 248
 qualifiers 185
 Quote 135

R

recursive templates 262
 reference columns
 PDM variables 220
 references
 CheckOnCommit 166
 ConstName 166
 DclDelIntegrity 166
 DclUpdIntegrity 166
 DefineJoin 166
 EnableChangeJoinOrder 166
 EnableCluster 166
 EnableKeyName 166
 FKAutoIndex 166
 FKeyComment 166
 MaxConstLen 166
 PDM variables 220
 SqlListChildrenQuery 166
 UseSpFornKey 166
 RegistryHome global constant 312
 Remove 174
 Remove() method 314, 321
 Rename 148, 152
 .replace macro 268
 Report Item Templates category 303
 report language files 1, 287
 All Attributes and Collections tab 300
 All Classes tab 299
 All Report Titles tab 297
 creating 289
 Linguistic Variables category 300
 Object Attributes category 298
 opening 288
 properties 290
 Report Item Templates category 303
 Report Titles category 294
 translation example 295
 Values Mapping category 291
 report templates 1
 Report Titles category 294
 reports
 CreateReport() method 325
 creating by script 325
 GenerateHTML() method 325
 GenerateRTF() method 325

 generating by script 325
 Reports collection 325
 translating 287
 Reports collection 325
 repository
 checking documents in by script 325
 checking documents out by script 325
 checking resource files into 1
 Checkout() method 325
 comparing resource files 1
 connecting by script 325
 Consolidate() method 325
 ConsolidateNew() method 325
 RepositoryConnection global property 312
 updating resource files from 1
 RepositoryConnection global property 312
 ReservedDefault 140
 ReservedWord 140
 resource editor
 See also resource files
 resource files
 adding items 6
 checking into the repository 1
 comparing 7
 comparing with the repository 1
 conversion tables 1
 copying 7
 creating 7
 csv (conversion tables) 1
 DBMS definition files 1
 deleting items 6
 editing 6
 embedding 6
 extension files 1
 impact and analysis rule sets 1
 mcc (model category sets) 1
 merging 8
 model category sets 1
 navigating in 5
 not certified 3
 object language definition files 1
 object permission profiles 1
 opening 3
 paths 1
 ppf (object permission profiles) 1
 process language definition files 1
 report language files 1
 report templates 1
 repository 1

Index

- restoring defaults 6
 - rtp (report templates) 1
 - rul (impact and analysis rule sets) 1
 - saving 6
 - searching 5
 - sharing 6
 - updating from the repository 1
 - upf (user profiles) 1
 - user profiles 1
 - xdb (DBMS definition files) 1
 - xem (extension files) 1
 - XML language definition files 1
 - xol (object language definition files) 1
 - xpl (process language definition files) 1
 - xrl (report language files) 1
 - xsl (XML language definition files) 1
 - restoring defaults in resource files 6
 - result columns 195
 - retrieving mappings by script 324
 - reverse engineering 120
 - attributes 128
 - EX keyword 128
 - extending 128
 - function based indexes 131
 - live connection 125
 - live databases 129, 131, 132
 - PDM extended objects 133
 - PDM variables 234
 - physical options 129
 - qualifiers 132
 - ReversedQueries 128
 - ReversedStatements 124
 - script after 133
 - script before 133
 - scripts 124
 - reverse-engineering databases by script 328
 - ReverseDatabase() method 328
 - ReversedQueries 128
 - ReversedStatements 124, 144
 - RevokeOption 191, 192
 - robustness diagrams
 - creating custom checks 21
 - creating custom symbols 19
 - creating extension for 16
 - creating generated files 29
 - creating stereotypes 18
 - creating templates 27
 - testing extension 30
 - roles
 - Bind 188
 - SqlListChildrenQuery 188
 - SqlPermQuery 188
 - Unbind 188
 - rtp (report templates) 1
 - rul (impact and analysis rule sets) 1
 - RuleComment 177
 - rules
 - ColnDefaultName 177
 - ColnRuleName 177
 - MaxDefaultLen 177
 - PDM variables 223
 - RuleComment 177
 - UddtDefaultName 177
 - UddtRuleName 177
- ## S
- saving resource files 6
 - scope
 - Outer 255
 - Parent 255
 - ScriptExt 138
 - scripting
 - accessing extensions 329
 - accessing object properties 316
 - accessing objects 316
 - ActiveModel global property 313
 - browsing collections 314
 - changing symbol format 320
 - checking documents into the repository 325
 - checking out of the repository 325
 - connecting to databases 326, 328
 - connecting to the repository 325
 - create shortcut 323
 - CreateModel() method 313
 - creating data sources 324
 - creating extensions 329
 - creating link objects 318
 - creating mappings 324
 - creating models 313
 - creating object selections 321
 - creating objects 318
 - creating reports 325
 - creating shortcuts 323
 - creating symbols 320
 - databases 326, 328
 - deleting objects 321
 - Edit/Run Script editor 307

- extending the metamodel 329
- folders 322
- generating databases 326
- generating reports 325
- generating test data 326
- global constants 312
- global functions 312
- global properties 312
- introduction 305
- launching scripts via custom commands 338, 339
- mappings 324
- metamodel 330
- Metamodel Objects Help 347
- Models global collection 313
- modifying collections 314
- modifying object properties 316
- navigating in the metamodel 330
- notScriptable stereotype 344
- OLE Automation 332
- opening models 313
- OpenModel() method 313
- positioning symbols 320
- reports 325
- repository 325
- reverse-engineering databases 328
- running scripts 307
- sample scripts 309
- transactions 312
- VBScript example 305
- workspace 322
- searching in resource files 5
- SequenceComment 186
- sequences
 - EnableOwner 186
 - PDM variables 224
 - SequenceComment 186
- .SERVEREXPRESSION macro 242
- .set_interactive_mode macro 278
- .set_object macro 279
- .set_value macro 279
- SetAttribute() method 316
- SetAttributeText() method 316
- SetExtendedAttribute() method 329
- SetExtendedAttributeText() method 329
- Settings category
 - object language 110
 - process language 109
 - XML language 112
- sharing resource files 6
- %Shortcut% 256
- shortcuts 344
 - CreateShortcut() method 323
 - creating by script 323
 - in GTL 256
- ShowMode global property 312
- Source property 314
- SourceClassifiers collection 324
- specializations 344
- SQL category (DBMS) 135
- SQL Server
 - PDM variables 227
- SqlAkeyIndex 164
- SqlAttrQuery 144
- SqlChckQuery 148, 152
- SqlContinue 135
- SqlListChildrenQuery 166, 187, 188
- SqlListDefaultQuery 173
- SqlListQuery 144
- SqlListRefrTables 148
- SqlListSchema 148, 169
- SqlOptsQuery 144
- SqlPermQuery 148, 152, 169, 177, 180, 187, 188
- SqlStatistics 152
- SqlSupport 134
- SqlSysIndexQuery 160
- .SQLXML PDM macro 243
- SqlXMLTable 148
- SqlXMLView 169
- StartCommand 138
- stereotypes 37, 39
 - example 18
 - promoting to metaclass 37, 39
 - Use as metaclass 18, 37, 39
 - UseAsMetaclass property 329
- StorageComment 171
- storages
 - PDM variables 224
 - StorageComment 171
- strings
 - A (align left) 248
 - aligning left 248
 - converting to first character lowercase 215, 248
 - converting to first character uppercase 215, 248
 - converting to lowercase 215, 248
 - converting to uppercase 215, 248

Index

- deleting substrings 215, 248
- `.foreach_part` macro 274
- iterating over in GTL 274
- L (lowercase) 215, 248
- LF (first character lowercase) 215, 248
- M (delete substring) 215, 248
- Q (double quotes) 215, 248
- q (single quotes) 215, 248
- surrounding in double quotes 215, 248
- surrounding in single quotes 215, 248
- T (trim whitespace) 215, 248
- trimming whitespace 215, 248
- U (uppercase) 215, 248
- UF (first character uppercase) 215, 248
- subclassifying metaclasses with criteria 40
- subclassifying metaclasses with stereotypes 37
- submenus
 - creating 339
- substrings
 - deleting in GTL 248, 268
 - deleting in PDM variables 215
 - M (delete substring) 215, 248
 - replacing in GTL 268
- symbols
 - creating by script 320
 - DashStyle property 320
 - formatting by script 320
 - LineWidth property 320
 - NewPoint() global function 320
 - Position property 320
 - positioning by script 320
- Symbols collection 320
- synonyms
 - EnableAlias 187
 - PDM variables 224
- Syntax category (DBMS)
 - BlockComment 135
 - BlockTerminator 135
 - Delimiter 135
 - IdentifierDelimiter 135
 - LineComment 135
 - Quote 135
 - SqlContinue 135
 - Terminator 135
 - UseBlockTerm 135
- syntax errors 283
- System 191
- T**
 - T formatting option (trim whitespace) 215, 248
 - TableComment 148
 - TableExt 138
 - tables
 - AddTableCheck 148
 - AllowedADT 148
 - AlterTableFooter 148
 - AlterTableHeader 148
 - ConstName 148
 - DefineTableCheck 148
 - DropTableCheck 148
 - MaxConstLen 148
 - PDM variables 216
 - Permission 148
 - Rename 148
 - SqlChckQuery 148
 - SqlListRefrTables 148
 - SqlListSchema 148
 - SqlPermQuery 148
 - SqlXMLTable 148
 - TableComment 148
 - TypeList 148
 - UniqConstraintName 148
 - TablespaceComment 171
 - tablespaces
 - PDM variables 224
 - TablespaceComment 171
 - tail string 250
 - templates 84, 87, 245
 - calling 257
 - example 27
 - F12 5
 - jumping to referenced template 5
 - Outer scope 255
 - overloading 257
 - overriding 257
 - Parent scope 255
 - passing parameters 260
 - recursive 262
 - referencing shortcuts 256
 - scope 255
 - Terminator 135
 - testing PDM variable values 213
 - text
 - formatting in GTL 248
 - text blocks
 - changing case in GTL 277
 - `.foreach_line` macro 273
 - iterating over in GTL 273
 - `.lowercase` macro 277

- outputting unique lines in GTL 281
 - .unique macro 281
 - .uppercase macro 277
 - Time 181
 - time formats 137
 - TimeFormat 136
 - Tools menu
 - creating custom commands in 338, 339
 - trace mode 14
 - transactions
 - BeginTransaction() global function 312
 - CancelTransaction() global function 312
 - EndTransaction() global function 312
 - Transformation Profile category 14
 - transformation profiles 94
 - transformations 92
 - transformation profiles 94
 - transformation scripts 95
 - translation errors 283
 - TrgFooter 138
 - TrgHeader 138
 - TrgUsage1 138
 - TrgUsage2 138
 - trigger template items 120
 - trigger templates 120
 - TriggerComment 181
 - TriggerExt 138
 - triggers
 - DefaultTriggerName 181
 - EnableMultiTrigger 181
 - EnableOwner 181
 - Event 181
 - EventDelimiter 181
 - PDM variables 222, 234
 - Time 181
 - TriggerComment 181
 - UseErrorMsgTable 181
 - UseErrorMsgText 181
 - TypeList 148, 169
- U**
- U formatting option (uppercase) 215, 248
 - UddtComment 173
 - UddtDefaultName 177
 - UddtRuleName 177
 - UF formatting option (first character uppercase)
 - 215, 248
 - Unbind 152, 173, 187, 188
 - UniqConstAutoIndex 164
 - UniqConstName 134
 - UniqConstraintName 148
 - UniqInTable 164
 - UniqName 160
 - .unique macro 281
 - unique identifiers
 - %NewUUID% global variable 251
 - .unset macro 279
 - upf (user profiles) 1
 - uppercase 277
 - .uppercase macro 277
 - UpperCaseOnly 136
 - Usage1 138
 - Usage2 138
 - Use as metaclass 18, 37, 39
 - UseAsMetaclass property 329
 - UseBlockTerm 135
 - UseErrorMsgTable 181
 - UseErrorMsgText 181
 - user interaction
 - controlling in GTL 278
 - .set_interactive_mode macro 278
 - user profiles 1
 - UserName global constant 312
 - users
 - %CurrentUser% global variable 251
 - SqlPermQuery 177
 - UserName 173
 - UseSpFornKey 166
 - UseSpPrimKey 163
- V**
- ValidationMode global property 312
 - Values Mapping category 291
 - variable
 - foreign key 241
 - primary key 241
 - variables
 - columns 235
 - .vbscript macro 281
 - VBScript 79, 95, 309
 - embedding in GTL 281
 - example 305
 - .execute_vbscript macro 270
 - executing with GTL 270
 - .vbscript macro 281
 - Version global constant 312
 - ViewCheck 169
 - ViewComment 169

Index

Viewer global constant 312

views

 PDM variables 216

 SqlListSchema 169

 SqlPermQuery 169

 SqlXMLView 169

 TypeList 169

 ViewCheck 169

 ViewComment 169

 ViewStyle 169

ViewStyle 169

W

.warning macro 269

warnings

 in GTL 269

 .warning macro 269

Web operations 194

Web parameters 195

Web services 194

 PDM variables 231

workspace

 accessing by script 322

 ActiveWorkspace global property 312, 322

 Children collection 322

 modifying by script 322

 saving by script 322

X

X formatting option (escape XML characters) 248

xdb (DBMS definition files) 1

xem (extension files) 1

xems

 See extension files

XML

 extensions for importing 96

 importing objects from 97

XML file

 add-in 335

 structure 335

XML imports 96

 initialization scripts 100

 mapping properties 100

 mappings 97

 post-processing scripts 100

XML language definition files 1

 DataTypes category 112

 extensions 117

 generation category 112

 generation commands 112, 114

 generation options 112, 113

 generation tasks 112, 114

 profile category 117

 properties 107

 Settings category 112

XML model file format 351

xol (object language definition files) 1

xpl (process language definition files) 1

xrl (report language files) 1

xsl (XML language definition files) 1