



Sybase[®]
PowerDesigner[®]

Advanced User Documentation

Version 9.5.1
38628-01-0951-01
Last modified: December 2002

Copyright © 2002 Sybase, Inc. All rights reserved.

Information in this manual may change without notice and does not represent a commitment on the part of Sybase, Inc. and its subsidiaries.

Sybase, Inc. provides the software described in this manual under a Sybase License Agreement. The software may be used only in accordance with the terms of the agreement.

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.

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, SYBASE (logo), AccelaTrade, ADA Workbench, Adaptable Windowing Environment, Adaptive Component Architecture, Adaptive Server, Adaptive Server Anywhere, Adaptive Server Enterprise, Adaptive Server Enterprise Monitor, Adaptive Server Enterprise Replication, Adaptive Server Everywhere, Adaptive Server IQ, Adaptive Warehouse, AnswerBase, Anywhere Studio, Application Manager, AppModeler, APT Workbench, APT-Build, APT-Edit, APT-Execute, APT-Translator, APT-Library, ASEP, Backup Server, BayCam, Bit-Wise, BizTracker, Certified PowerBuilder Developer, Certified SYBASE Professional, Certified SYBASE Professional Logo, ClearConnect, Client-Library, Client Services, CodeBank, Column Design, ComponentPack, Connection Manager, Convoy/DM, Copernicus, CSP, Data Pipeline, Data Workbench, DataArchitect, Database Analyzer, DataExpress, DataServer, DataWindow, DB-Library, dbQueue, Developers Workbench, Direct Connect Anywhere, DirectConnect, Distribution Director, e-ADK, E-Anywhere, e-Biz Integrator, E-Whatever, EC-GATEWAY, ECMAP, ECRTP, eFulfillment Accelerator, Electronic Case Management, Embedded SQL, EMS, Enterprise Application Studio, Enterprise Client/Server, Enterprise Connect, Enterprise Data Studio, Enterprise Manager, Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, eProcurement Accelerator, eremote, Everything Works Better When Everything Works Together, EWA, Financial Fusion, Financial Fusion Server, Formula One, Gateway Manager, GeoPoint, iAnywhere, iAnywhere Solutions, ImpactNow, Industry Warehouse Studio, InfoMaker, Information Anywhere, Information Everywhere, InformationConnect, InstaHelp, InternetBuilder, iremote, iScript, Jaguar CTS, jConnect for JDBC, KnowledgeBase, Logical Memory Manager, MainframeConnect, Maintenance Express, Manage Anywhere Studio, MAP, MDI Access Server, MDI Database Gateway, media.splash, MetaWorks, MethodSet, ML Query, MobiCATS, MySupport, Net-Gateway, Net-Library, New Era of Networks, Next Generation Learning, Next Generation Learning Studio, O DEVICE, OASIS, OASIS logo, ObjectConnect, ObjectCycle, OmniConnect, OmniSQL Access Module, OmniSQL Toolkit, Open Biz, Open Business Interchange, Open Client, Open ClientConnect, Open Client/Server, Open Client/Server Interfaces, Open Gateway, Open Server, Open ServerConnect, Open Solutions, Optima++, Partnerships that Work, PB-Gen, PC APT Execute, PC DB-Net, PC Net Library, PhysicalArchitect, Pocket PowerBuilder, PocketBuilder, Power++, Power Through Knowledge, power.stop, PowerAMC, PowerBuilder, PowerBuilder Foundation Class Library, PowerDesigner, PowerDimensions, PowerDynamo, Powering the New Economy, PowerJ, PowerScript, PowerSite, PowerSocket, Powersoft, PowerStage, PowerStudio, PowerTips, Powersoft Portfolio, Powersoft Professional, PowerWare Desktop, PowerWare Enterprise, ProcessAnalyst, Rapport, Relational Beans, Report Workbench, Report-Execute, Replication Agent, Replication Driver, Replication Server, Replication Server Manager, Replication Toolkit, Resource Manager, RW-DisplayLib, RW-Library, SAFE, SAFE/PRO, SDF, Secure SQL Server, Secure SQL Toolset, Security Guardian, SKILS, smart.partners, smart.parts, smart.script, SQL Advantage, SQL Anywhere, SQL Anywhere Studio, SQL Code Checker, SQL Debug, SQL Edit, SQL Edit/TPU, SQL Everywhere, SQL Modeler, SQL Remote, SQL Server, SQL Server Manager, SQL SMART, SQL Toolset, SQL Server/CFT, SQL Server/DBM, SQL Server SNMP SubAgent, SQL Station, SQLJ, Stage III Engineering, Startup.Com, STEP, SupportNow, S.W.I.F.T. Message Format Libraries, Sybase Central, Sybase Client/Server Interfaces, Sybase Development Framework, Sybase Financial Server, Sybase Gateways, Sybase Learning Connection, Sybase MPP, Sybase SQL Desktop, Sybase SQL Lifecycle, Sybase SQL Workgroup, Sybase Synergy Program, Sybase Virtual Server Architecture, Sybase User Workbench, SybaseWare, Syber Financial, SyberAssist, SybMD, SyBooks, System 10, System 11, System XI (logo), SystemTools, Tabular Data Stream, The Enterprise Client/Server Company, The Extensible Software Platform, The Future Is Wide Open, The Learning Connection, The Model For Client/Server Solutions, The Online Information Center, The Power of One, TradeForce, Transact-SQL, Translation Toolkit, Turning Imagination Into Reality, UltraLite, UNIBOM, Unilib, Uninull, Unisep, Unistring, URK Runtime Kit for UniCode, Versacore, Viewer, VisualWriter, VQL, WarehouseArchitect, Warehouse Control Center, Warehouse Studio, Warehouse WORKS, Watcom, Watcom SQL, Watcom SQL Server, Web Deployment Kit, Web.PB, Web.SQL, WebSights, WebViewer, WorkGroup SQL Server, XA-Library, XA-Server and XP Server are trademarks of Sybase, Inc. or its subsidiaries.

All other trademarks are property of their respective owners.

Contents

1	DBMS Reference Guide	1
	DBMS definition file overview	2
	DBMS editor.....	4
	Understanding DBMS categories.....	5
	DBMS property page	5
	General category.....	6
	Script and ODBC categories	6
	Profile category.....	10
	Managing generation and reverse engineering	16
	How does it work?	16
	Understanding statements and queries	16
	Script generation	18
	Script reverse engineering	21
	ODBC generation	22
	ODBC reverse engineering	22
	Syntax in SQL statements.....	33
	Defining physical options	42
	Defining physical options specified by a value	43
	Syntax for a physical option without a name	45
	How to define a default value for a physical option	46
	How to define a list of values for a physical option.....	46
	How to define a physical option corresponding to a tablespace or a storage.....	47
	Composite physical option syntax	47
	How to repeat options several times	49
	User interface changes	50
	General category	51
	EnableCheck	51
	EnableIntegrity.....	51
	EnableMultiCheck	51
	Enableconstname.....	52
	SqlSupport.....	52

UniqConstName	52
Script or ODBC categories	53
SQL category	54
Syntax	54
Format	56
File	59
Keywords	63
Objects category	67
Common object entries	67
Table	71
Column	77
Index	90
Pkey	96
Key	99
Reference	104
View	110
Tablespace	113
Storage	115
Database	116
Domain	118
Abstract Data Type	122
Abstract Data Type Attribute	124
User	125
Rule	126
Procedure	129
Trigger	133
Group	137
Role	138
Privilege	140
Permission	141
Join Index	142
Qualifier	144
Sequence	144
Synonym	146
DB Package	148
DB Package Procedure	150
DB Package Variable	151
DB Package Type	151
DB Package Cursor	151
DB Package Exception	152
DB Package Parameter	152
DB Package Pragma	153
Commands for all objects	154
MaxConstLen	154
EnableOption	154
Data type category	155

AmcdDataType.....	155
PhysDataType	155
PhysDttpSize	156
OdbcPhysDataType	156
PhysOdbcDataType	157
PhysLogADTType	157
LogPhysADTType	158
AllowedADT	158
HostDataType	158
PDM variables.....	160
Variables for database generation, and triggers and procedures generation	160
Variables for reverse engineering	160
Variables for database synchronization.....	161
Variables for database security.....	162
Variables for metadata	162
Common variables for all named objects	162
Common variables for objects.....	163
Variables for DBMS, database options	163
Variables for tables.....	163
Variables for domains and columns checks.....	164
Variables for columns.....	165
Variables for abstract data types.....	165
Variable for abstract data type attributes.....	166
Variable for domains	166
Variables for rules	166
Variables for ASE & SQL Server.....	167
Variables for sequences.....	167
Variables for indexes.....	167
Variables for join indexes (IQ).....	168
Variables for index columns	168
Variables for references	168
Variables for reference columns	169
Variables for keys	170
Variables for views	170
Variables for triggers	170
Variables for procedures	171

2	Managing Profiles	173
	Understanding the profile concept.....	174
	What is a profile?	174
	Available extensions.....	175
	Adding a metaclass to a profile	177
	Defining a stereotype	180
	Stereotype properties	180
	Creating a stereotype	181

Attach a tool to a stereotype.....	181
Defining a criterion	184
Defining a custom symbol in a profile	186
How to customize a symbol.....	186
Defining a custom symbol	186
Defining extended attributes in a profile	189
Creating an extended attribute type	189
Creating an extended attribute	190
Defining a custom check in a profile.....	192
Custom check properties	192
Defining the script of a custom check	193
Defining the script of an autofix.....	195
Using the global script.....	197
Troubleshooting VB script errors.....	198
Running a custom check.....	198
Defining templates and generated files in a profile.....	200
Creating a template.....	200
Creating a generated file	201
Using profiles: a case study.....	205
Scenario	206
Attaching a new extended model definition to the model.....	206
Create object stereotypes	208
Define custom symbols for stereotypes	211
Create instance links and messages between objects ..	215
Create custom checks on instance links.....	216
Generate a textual description of messages.....	222

3

Object Languages Reference Guide..... 231

Object language overview.....	232
Understanding the object language editor	233
Object language properties	234
Settings category.....	235
Generation category.....	238
Profile category	247
General category.....	256

4

Extended Model Definitions Reference Guide 259

Managing extended model definitions	260
Creating an extended model definition.....	261
Importing extended attributes from previous versions ...	264
Importing an extended model definition into a model.....	267
Exporting an extended model definition	268
Working with extended model definitions	270
Extended model definition properties	271

	Generation category	273
	Generating for an extended model definition	279
5	Generation Reference Guide	281
	Defining Generation Template Language	282
	Defining concepts used in the GTL	283
	Defining templates	283
	Defining variables	284
	Defining translation scope	290
	Defining inheritance	292
	Defining template overriding	292
	Defining polymorphism	293
	Defining template overloading	294
	Defining escape sequences	294
	Using macros	295
	Defining conditional blocks	313
	Defining error messages	314
	Generation tips and techniques	316
	Sharing templates	316
	Using environment variables	317
	Using new lines in head and tail string	317
	Using parameter passing	320
6	PowerDesigner Public Metamodel	323
	What is the PowerDesigner public metamodel?	324
	Metamodel concepts	325
	Public names	325
	Classes	325
	Associations and collections	326
	Generalizations	327
	Comments and notes on objects	327
	Understanding the metamodel structure	328
	Overall organization	328
	PdCommon content	329
	Other library packages content	329
	Navigating in the metamodel	329
	Using the metamodel with VBS	332
	Using the metamodel with the Generation Template Language	334
	Calculated attributes	334
	Calculated collections	336
	Using the metamodel to understand the PowerDesigner XML file format	337

PowerDesigner File Format Specification.....	339
XML file format.....	340
XML file structure and content	340
Understanding the XML format	344
Case study	344
Modifying an XML file.....	347
Modifying an XML file using a standard Editor	347
Modifying an XML file using an XML editor	349
Visualizing an XML file using an XML viewer.....	350

CHAPTER 1

DBMS Reference Guide

About this chapter This document describes the structure and contents of a PowerDesigner DBMS definition file. This guide is a reference for working with any PowerDesigner supported DBMS. It provides a global overview of the use of DBMS Definition files regardless of DBMS characteristics.

Contents

Topic	Page
DBMS definition file overview	2
DBMS editor	4
Understanding DBMS categories	5
Managing generation and reverse engineering	16
Defining physical options	42
General category	51
Script or ODBC categories	53
SQL category	54
Objects category	67
Commands for all objects	154
Data type category	155
PDM variables	160

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

DBMS definition file overview

PowerDesigner can be used with many different DBMS. For each of these DBMS, a standard definition file is included and provides an interface between PowerDesigner and the DBMS so as to establish the best relationships between them.

Caution

You should never modify the DBMS files shipped with PowerDesigner. For each original DBMS you want to modify, you should create a corresponding new DBMS. To do so you have to create a new DBMS from the List of DBMS, define a name and select the original file in the Copy From dropdown listbox. This allows you to create a new DBMS that is identical to the original file apart from the name.

✍ For more information on creating a new DBMS definition from an existing DBMS definition, see section Creating a new definition file in chapter The Target Definition Editor in the *General Features Guide*.

However, you may want to modify this interface to suit your particular applications. This is done with the DBMS definition file editor.

What is a DBMS definition file?

A **DBMS definition file** is a list of values and settings that represent specifications for a particular Database Management System (DBMS) in a format understandable by PowerDesigner. As an interface between an actual DBMS and PowerDesigner, it provides PowerDesigner with the syntax and guidelines for generating databases, triggers, and stored procedures appropriate for a target DBMS. The file itself is in .XML format.

The DBMS definition file is a required component of PowerDesigner when working with Physical Data Models (PDM). Each actual DBMS supported by PowerDesigner has its own DBMS definition.

What is contained in a DBMS definition?

All DBMS definition files have the same structure made up of a number of **categories**. A category can contain other categories, **entries**, and **values**. These entries are parameters recognizable by PowerDesigner.

The values for DBMS definition categories and entries vary for each DBMS. Some entries may not exist in the DBMS file if they are not applicable to the particular DBMS.

Certain entries contain mandatory parameters to generate correct syntax. Some entries contain SQL statements that will allow PowerDesigner to generate and reverse engineer correctly for the chosen database (**create**, **drop**, and so on).

What are
PowerDesigner
variables?

You can incorporate variables in the SQL queries of the selected DBMS. These variables are replaced with the actual values from your model when the scripts are generated. These variables are evaluated to create corresponding objects in PowerDesigner models during reverse engineering.

PowerDesigner variables are written between percent signs (%).

Example

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

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

↪ For the full list of PowerDesigner variables that you can use in a DBMS Definition, see section PDM variables.

DBMS editor

You can consult or modify a DBMS definition file using the DBMS editor.

❖ **To display the DBMS editor:**

- ◆ Select Database→Edit current DBMS
or
Select Tools→Resources→DBMS and select an existing DBMS or create a new DBMS definition file.

The DBMS Properties dialog box appears.

The DBMS editor lets you navigate through the DBMS categories and entries. When you select a **category** in the DBMS definition editor, the name, code, and related comment appear in the right side of the dialog box. When you select an **entry** in the DBMS definition window, the name, value, and related comment appear in the right side of the dialog box.

🔗 For more information on how to use the DBMS editor, see chapter The Resource Editor in the *General Features Guide*.

Triggers templates and template items

The DBMS contains the definition of trigger templates and trigger template items.

You access the Trigger templates or Template items pages by clicking on the corresponding tab.

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

🔗 For more information on using, creating, and editing trigger templates and trigger template items, see the chapter Triggers and Procedures in the *PDM User's Guide*.

Understanding DBMS categories

A DBMS definition can have the following categories:

Category	Description
General	Target DBMS identification
Script	DBMS characteristics, command definition, and data type translations for the Script generation and reverse engineering
ODBC	DBMS characteristics, command definition, and data type translations for the ODBC generation and reverse engineering
Profile	Extension of PowerDesigner metaclasses to customize models

Note that some DBMS do not display an ODBC category as it is not necessary to their definition.

DBMS property page

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

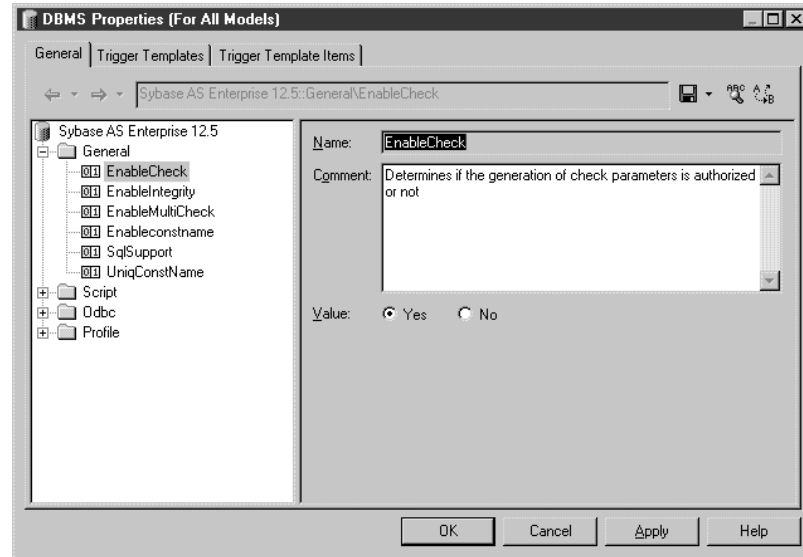
Property	Description
Name	Name of the DBMS. This name must be unique in a model
Code	Code of the DBMS. This code must be unique in a model
File Name	Path and name of the DBMS file. You cannot modify the content of this box
Family	Used to classify a DBMS. Family is designed to help establish a link between different database resource files. For example, Sybase AS Anywhere, and Sybase AS Enterprise belong to the SQL Server family. Triggers are not erased when changing the target database within the same family. Merge interface allows to merge models from the same family
Comment	Additional information about the DBMS

General category

The General category contains general information about the database, without any categories. All entries defined in the General category apply to all database objects.

Example

When the **EnableCheck** entry value is Yes, check parameters are generated.



Script and ODBC categories

The Script category is used for the Script generation and reverse engineering, and ODBC reverse engineering.

The ODBC category is used for the ODBC generation queries when the DBMS does not support standard statements for generation. Note that some DBMS will not display an ODBC category as it is not necessary to their definition.

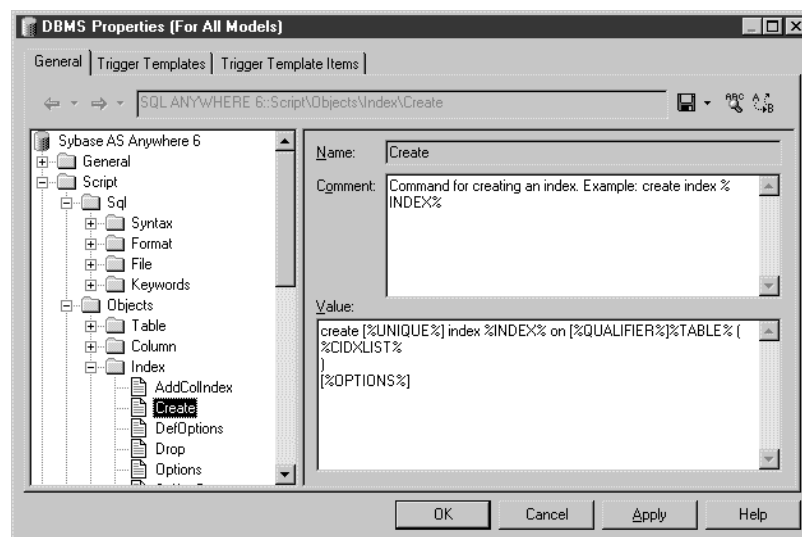
When a command value or its category is missing during the ODBC generation, PowerDesigner searches for it in the Script category. As a consequence, an entry is redefined in the ODBC category only if it is different from the Script value.

Both Script and ODBC categories contain the following categories:

Categories	Description
SQL	Contains categories Syntax, Format, File, and Keywords. Each category contains entries whose values define general syntax for the database
Objects	Contains categories for each type of object in the database. Each category contains entries whose values define database commands and object-related characteristics
Data Type	Contains data type translation entries. These entries list the correspondence between PowerDesigner internal data types and the target database data types
Customize	Retrieves information from PowerDesigner Version 6 DBMS definition files. It is not used in later versions

Example

In the definition file of Sybase Adaptive Server Anywhere 6, the Script category is expanded to show the category Objects where Index is also expanded. The **Create** entry is selected and the Value box associated with this command defines how the index is created.



SQL category

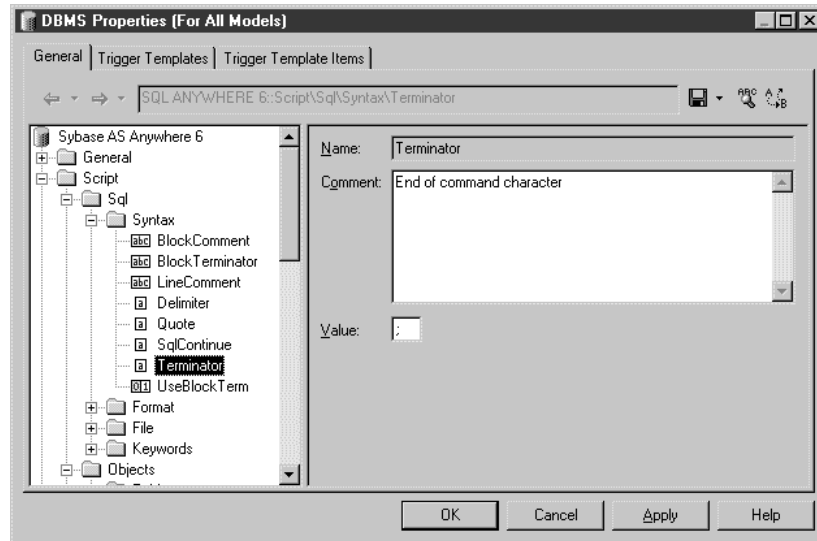
The SQL category contains the following categories:

- ◆ Syntax contains general parameters for SQL syntax

- ◆ Format contains parameters for allowed characters
- ◆ File contains header, footer and usage text entries used during generation
- ◆ Keywords contains the list of reserved words and functions available in SQL

Example

In the definition file of Sybase Adaptive Server Anywhere 6, the Script category is expanded to show the category SQL where Syntax is also expanded to show general syntax parameters such as **Terminator**, **BlockTerminator**.



The **Terminator** entry is selected and the Value box associated with this entry defines the current terminator symbol.

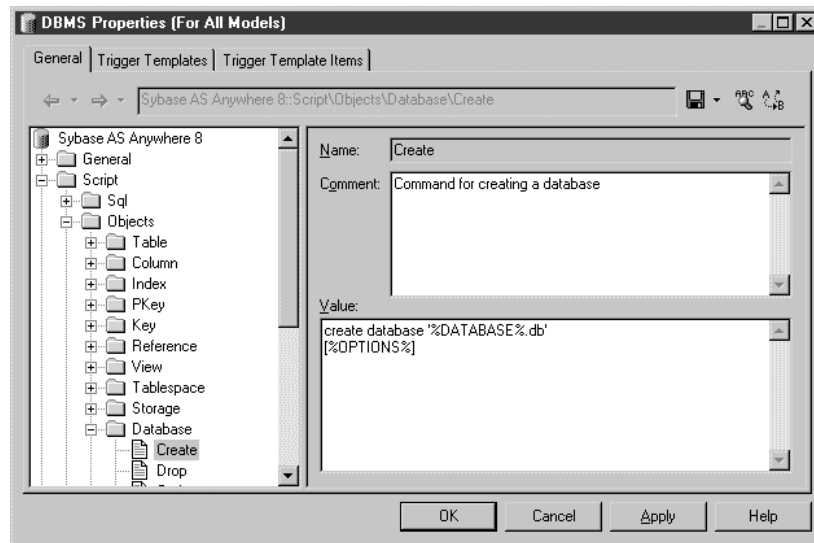
Object category

The Object category contains all the commands to create, delete or modify all the objects in the database. This also includes commands that define object behavior, defaults, necessary SQL queries, reverse engineering options, and so on.

Example

In the definition file of Sybase Adaptive Server Anywhere 8, the Objects category is expanded to show the category Database.

The **Create** entry is selected and the associated Value box contains the command for database creation.

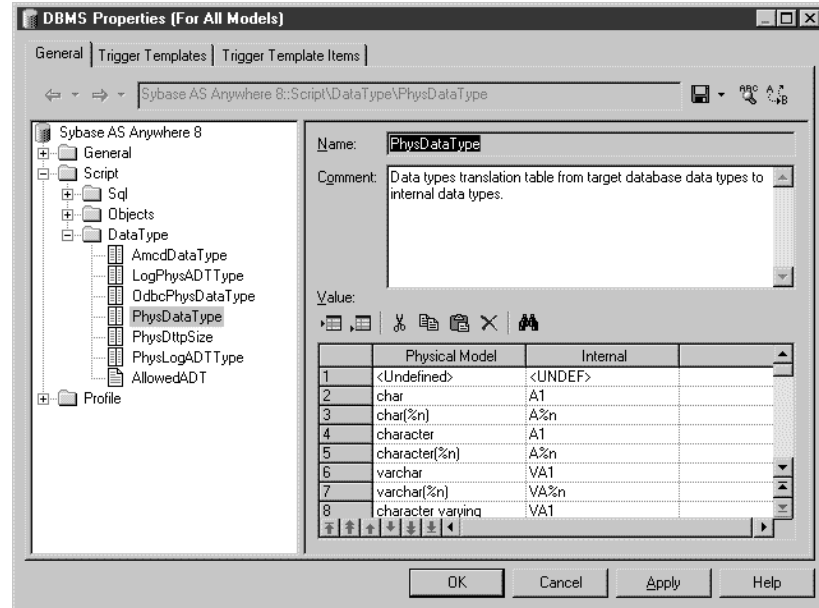


Data type category

The Data Type category contains the list of valid data types for the specified DBMS and the corresponding types in PowerDesigner.

Example

In the definition file of Sybase Adaptive Server Anywhere 8, the Data Type category is expanded to show all data types. The **PhysDataType** entry is selected and the associated value shows the mapping described in the Comment box.



Profile category

The profile category is used to define PowerDesigner profiles.

In a DBMS, you can define extended attribute types and extended attributes for database objects.

The Profile category contains the following categories:

Category	Description
Shared\Extended attribute types	For defining extended attribute types and shared templates. Extended attribute types are data types reused among extended attributes in an object language. Shared templates are pieces of code used in text generation

Category	Description
Metaclasses	For defining metaclass extensions like custom symbol, stereotypes, criteria, or generated files
	<p>↪ For more information on profiles, see chapter Managing Profiles.</p> <p>↪ For more information on templates and generated files, see chapter Generation Reference Guide.</p>
Extended attributes	<p>For each extended attribute defined in this category, an Extended Attributes tab appears in the corresponding object property sheet. You can use this tab to add an extended attribute value to the object definition.</p> <p>Some DBMS are delivered with extended attributes that are needed during generation, this is why we advise you not to modify these extended attributes, or at least to make a backup copy of each DBMS file before you start modifying them.</p> <p>If you wish to enhance model generation, you can copy the pattern of existing extended attributes and assign them to other object categories.</p>
Extended model definition	<p>If you want to complement the definition of modeling objects and expand the PowerDesigner metamodel, you should define extended attributes in an extended model definition. Such extended attributes are not used during the generation process.</p> <p>↪ For more information on extended model definitions, see chapter Extended Model Definitions Reference Guide.</p> <p>The Extended Attribute Category is divided into the following categories:</p> <ul style="list-style-type: none"> ◆ Types ◆ Objects

Defining an extended attribute in a DBMS

Each extended attribute has the following properties:

Entry property	Description
Name	Name of category or entry
Comment	Description of selected category or entry
Data type	Predefined or user-defined extended attributes types
Default value	Default value from the list of values. This depends on the selected data type

❖ **To add an extended attribute:**

- 1 Right-click a metaclass category in the Profile category and select New→Extended Attributes from the contextual menu.
A new extended attribute is created.
- 2 Type a name in the Name box.
- 3 Type a comment in the Comment box.
- 4 Select a data type from the Data Type dropdown list box.
- 5 <optional> Select a default value in the Default Value dropdown list box.
- 6 Click Apply.

Example

In DB2 UDB 7 OS/390, extended attribute **WhereNotNull** allows you to add a clause that specifies that index names must be unique provided they are not null.

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

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

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

In the **SqlListQuery** entry:

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

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

```
[ and tbcreator=%.q:OWNER%]
[ and dbname=%.q:CATALOG%]
order by
  1 , 2 , 3
```

Using extended attributes in the PDM

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

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

✍ For more information on extended model definitions, see chapter Extended Model Definitions Reference Guide.

By default, extended attributes appear on separate tabbed pages in the Extended Attributes page of an object property sheet or in object lists.

Each extended attribute has the following properties:

Property	Description
Name	Name of extended attribute
Data type	Extended attribute data type including boolean, color, date, file, float, font, etc or customized data types
Value	Value of the extended attribute. This field displays the default value defined for the extended attribute data type
R	Redefined value. This check box is selected if you modify the default value in the Value column, using either the arrow or the ellipsis button

The name and data type of the extended attributes cannot be modified from the property sheet of the object. These properties must be edited from the DBMS editor. However, the extended attribute value can be modified in the extended attribute page or in a list.

❖ To define the value of an extended attribute:

- 1 Open the property sheet of an object.
or
Select Model→*Object* to display a list of objects.

- 2 Click the Extended Attributes tab to display the corresponding page.
or
Click the Customize Columns and Filter tool, select extended attributes in the list of columns, and click OK.

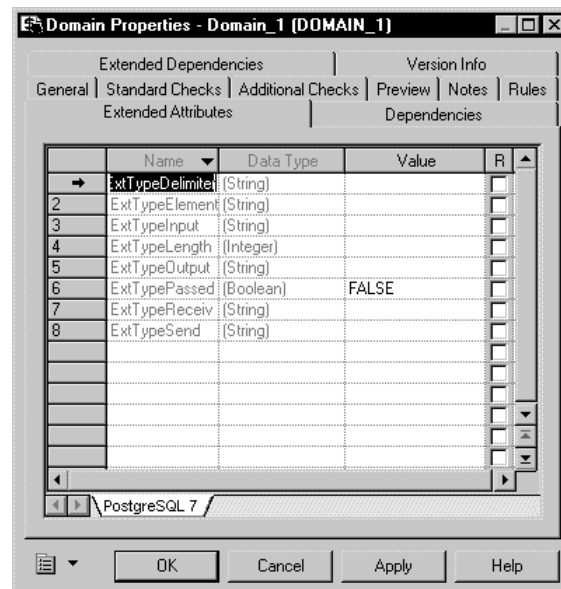
The extended attributes appear in different tabbed pages or as additional columns in the list.
- 3 Click the Value column of an extended attribute if you want to modify its value and select a value from the dropdown listbox.
or
Type or select a value in the value cell in the list.

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

Using extended attributes during generation

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

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



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

PowerDesigner is case-sensitive

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

Example

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

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

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

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

In the **SqllistQuery** entry:


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

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

Managing generation and reverse engineering

PowerDesigner uses the DBMS definition file to communicate with the database via reverse engineering and generation. PowerDesigner supports both **script** and **ODBC** for reverse engineering and generation. In this section, you will learn more about the process of reverse engineering and generation.

How does it work?

	This section explains the generation and reverse engineering mechanism.
Terminology	<p>The word statement is used to define a piece of SQL syntax; statements usually contain variables that will be evaluated during generation and script reverse engineering.</p> <p>The word query is reserved to ODBC reverse engineering.</p>
Generation	During generation, the statements are parsed and the variables contained in the statements are evaluated and replaced by their actual values taken from the current model. The same statements are used for script and ODBC generation.
Reverse engineering	<p>During script reverse engineering, PowerDesigner parses the script and identifies the different statements thanks to the terminator (defined in Script\Sql\Syntax). Each individual statement is "associated" with an existing statement in the DBMS definition file in order to commit the variables in the reversed statement as items in a PowerDesigner model.</p> <p>During ODBC reverse engineering, special queries are used to retrieve information from the database system tables. Each column of a query result set is associated with a variable. The query header specifies the association between the columns of the resultset and the variable. The values of the returned records are stored in these variables which are then committed as object attributes.</p> <p> For more information on variables, see section Optional strings and variables.</p>

Understanding statements and queries

Statements for script generation, script reverse engineering, and ODBC generation are identical, whereas ODBC reverse engineering requires specific queries.

Location	Script category contains	ODBC category contains
	Statements for script generation Statements for script reverse engineering ODBC reverse engineering queries	ODBC generation statements when DBMS does not support standard statements

Exploring the Script category

The Script category contains the following entries:

- ◆ Statements for generation and reverse engineering by script
- ◆ Modify statements
- ◆ Database definitions items
- ◆ ODBC reverse engineering queries

Statements for generation and reverse engineering

The Script category contains Database Description Language (ddl) statements used for script and ODBC generation and script reverse engineering.

For example, the standard statement for creating an index is:

```
create index %INDEX%
```

However, statement values vary from one definition file to another in order to respect the DBMS syntax and specific features. For example in Oracle 9i, the create statement for an index contains the definition of the index owner since Oracle 9i supports index owners:

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

Among other statements you have:

- ◆ Drop for deleting an object
- ◆ Options for defining the physical options of an object
- ◆ ConstName for defining the constraint name template for the checks of an object

and so on...

Modify statements	<p>These statements are used during a database modification to modify attributes of already existing objects. They can be easily identified because most of them start with the word "Modify". For example ModifyColumn is the statement used for modifying a column.</p> <p>However, not all statements start with Modify, for example Rename or AlterTableFooter.</p> <p>The statement for creating a key is also special depending on the where the key is defined: if the key is inside the table, then it will be created with a generation order, and if the key is created outside the table, it will be a modify order of the table.</p>
Database definition items	<p>The Script category also contains items related to the database definition. These are not statements, they are not used during generation or reverse engineering but rather for customizing the PowerDesigner interface and behavior according to database features.</p> <p>For example, item Maxlen in the table category, has to be set according to the maximum code length tolerated for a table in the current database.</p> <p>Permission, EnableOwner, AllowedADT are other examples of items defined to adapt PowerDesigner to the current DBMS.</p>
ODBC reverse engineering queries	<p>Most ODBC reverse engineering queries start with "Sql" which is an easy way to identify them. For example, SqlListQuery is the query used to retrieve a list of objects, or SqlOptsQuery is the query used to reverse engineer physical options.</p> <p>🔗 For more information on ODBC reverse engineering queries see section ODBC reverse engineering.</p>

Exploring the ODBC category

The ODBC category contains entries for ODBC 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.

Script generation

Script generation statements are available in the Script category, under the different object categories.

For example, in Sybase ASA 8, the Create statement in the Table category is the following:

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

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

Extension mechanism

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

☞ For more information on reverse engineering additional statements see section Script reverse engineering.

You use before or after statements to generate additional code; these are text items defined in a selected object category in the DBMS. In this section, we shall call them **extension statements**.

Generation Template Language

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

An extension statement can contain:

- ◆ Reference to other **statements** that will be evaluated during generation. These entries are text items that must be defined in the object category of the extension statements
- ◆ **Variables** used to evaluate object properties and extended attributes. Variables are enclosed between % characters
- ◆ **Macros** provide generic programming structures for testing variables (.if)

☞ For more information on the PowerDesigner Generation Template Language (GTL), see chapter Generation Reference Guide.

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

Example 1

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

AfterCreate is defined in GTL syntax as follows:

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

```
.endif
```

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

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

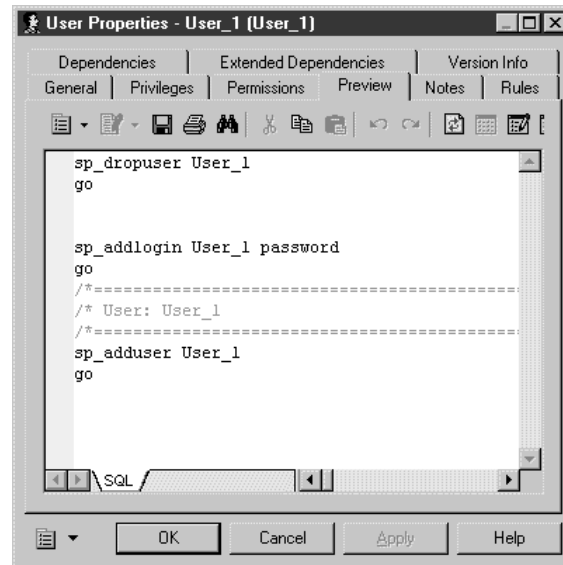
%CreatePartition% generates the statement for creating the number of table partitions specified in %ExtTablePartitions%.

Example 2

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

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

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



Modify statements

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

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

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


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

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

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

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

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

 For more information on the PowerDesigner Generation Template Language (GTL), see chapter Generation Reference Guide.

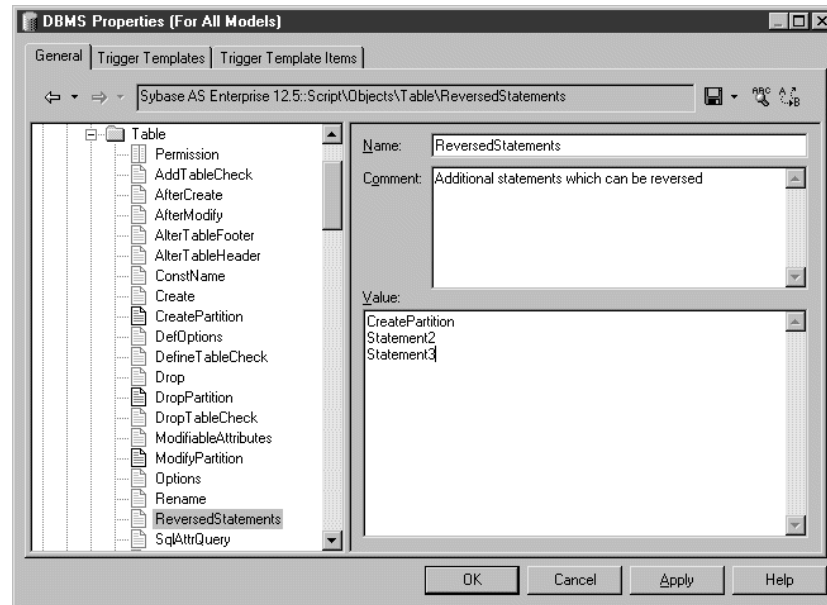
Script reverse engineering

The same statements are used for generation and reverse engineering.

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

For example, the extension statement AfterCreate uses statement CreatePartition. This text entry must be declared in ReversedStatements to be properly reverse engineered.

You could declare other statements in the following way:



ODBC generation

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

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

ODBC reverse engineering

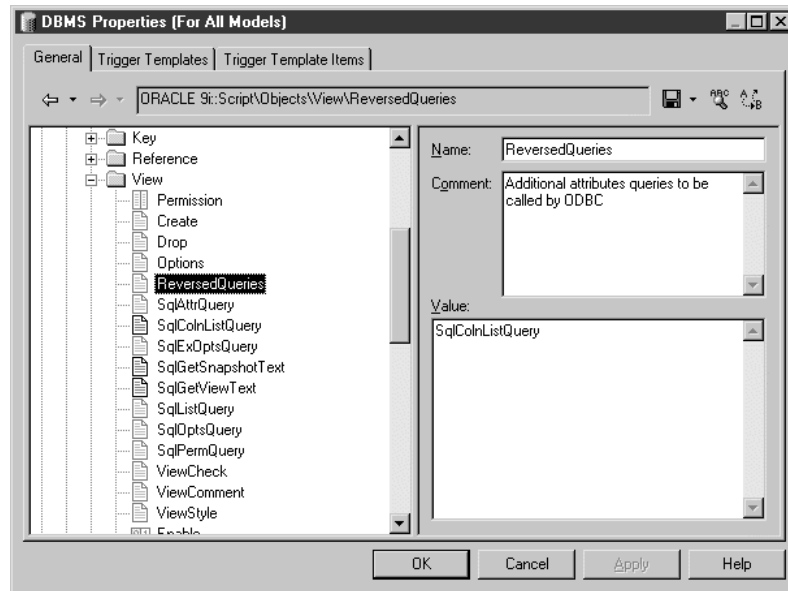
The DBMS contains ODBC reverse engineering queries for retrieving objects (like Table, Columns, and so on) from the database.

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

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

You can define additional ODBC queries to recover more attributes during reverse engineering. This is to avoid loading **SqlListQuery** with queries for retrieving attributes not supported by **SqlAttrQuery**, or objects not selected for reverse engineering. These queries must be listed in the **ReversedQueries** entry.

For example, **SqlColnListQuery** is used to exclusively retrieve view columns. This query has to be declared in the ReversedQueries entry in order to be taken into account during reverse engineering.



Query structure

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

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

For example:

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

The list of possible variables corresponds to the list of variables established in chapter PowerDesigner PDM variables.

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

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

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

Processing with
variable names:

```
{TABLE ID, ISPKEY ID, CONSTNAME ID, COLUMNS ...}
select
  t.table_name,
  1,
  null,
  c.column_name + ', ',
  c.column_id
from
  systable t,
  syscolumn c
where
  etc..
```

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

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

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

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

Processing with
conversion table:

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

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

where * means all other values.

For example:

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

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

Extension mechanism for ODBC reverse engineering queries

During reverse engineering, PowerDesigner executes queries to retrieve information from the columns of the system tables. The result of a query is mapped to PowerDesigner internal variables via the query header. When the system tables of a DBMS store information in columns with LONG, BLOB, TEXT and other incompatible data types, it is impossible to concatenate these information in a string.

You can bypass this limitation by creating user-defined queries and user-defined variables in the existing reverse engineering queries.

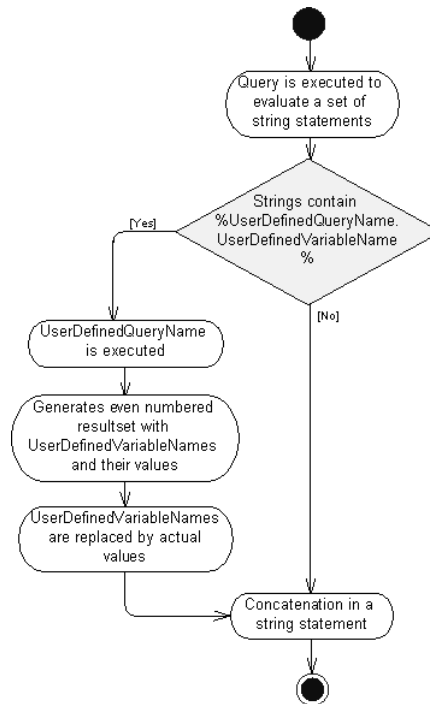
During reverse engineering, queries are executed to evaluate variables and create a string statement. When the query header contains the **EX** keyword, it means that the query return contains user-defined queries and variables.

For example:

```
select '%UserDefinedQueryName.UserDefinedVariableName%'  
|| i.indexname  
from...
```

These user-defined variables will be evaluated by separate queries defined by the user.

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



Step1

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

If variables contain user-defined queries and user-defined variables, the user-defined query is executed to evaluate the user-defined variables. These user-defined variables are created to be filled with data proceeding from columns with LONG/BLOB/TEXT... data type.

You can create user-defined queries in any ODBC reverse engineering query. Make sure you use the same variable name in these queries and in the user-defined queries, otherwise the variables will not be evaluated.

You should also check for variable name uniqueness if you want all the variables to be evaluated during the execution of the query.

The header of a user-defined query contains internal variables names that will not be evaluated. However, the translation rules for values expressed between brackets (like (0="", *=", ")) may be used during the string concatenation.

Step 2

The execution of the user-defined query must generate 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.

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

Variable 1	1	Variable 2	2
Variable 3	3	Variable 4	4
Variable 5	5	Variable 6	6

Step 3

The user-defined variable names are replaced by their values.

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

ODBC 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 %FragDbasp' || 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
FragDbbsp <i>n</i>	Fragment location for fragment number <i>n</i>
FragExpr <i>n</i>	Fragment expression for fragment number <i>n</i>
FragSprt <i>n</i>	Fragment separator for fragment number <i>n</i>

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.

ODBC reverse engineering function-based index

In Oracles 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.colu
mn_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.

Optimizing ODBC reverse engineering queries

ODBC reverse engineering has been optimized in order to improve performance. All ODBC queries run according to an optimization process rule. This process uses the following registry keys:

- ◆ **RevOdbcMinCount** defines a number of selected objects for reverse engineering
- ◆ **RevOdbcMinPerct** defines a percentage of selected objects for reverse engineering

Modify key value

You can modify the value of **RevOdbcMinCount** and **RevOdbcMinPerct** in the Registry Editor

During reverse engineering, PowerDesigner compares the total number of current objects for reverse engineering to the value of **RevOdbcMinCount**.

If the total number of listed items is lower than the value of RevOdbcMinCount A global reverse query is executed.

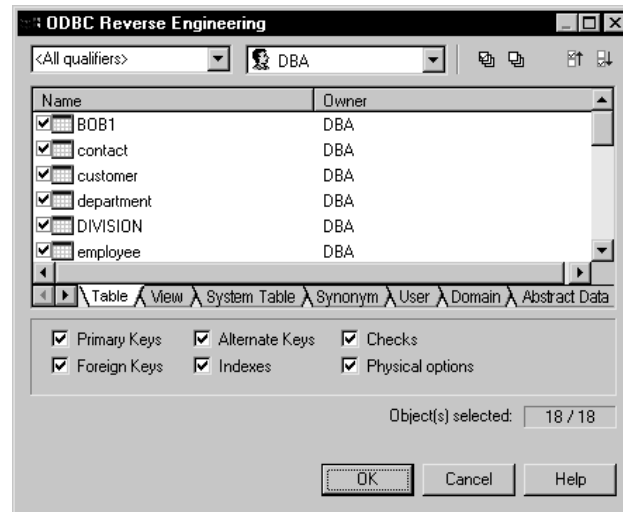
If the total number of listed items is higher than the value of RevOdbcMinCount The process uses key **RevOdbcMinPerct**.

- ◆ If the percentage of reversed items is lower than the percentage defined in RevOdbcMinPerct, then a global query is executed
- ◆ If the percentage of reversed items is higher than the percentage defined in RevOdbcMinPerct, then the same query is executed for each object

ODBC 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 ODBC 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 entries:

- ◆ enable: YES/NO
- ◆ SqlListQuery (script) : this entry 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 entries are shown in the table below:

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

For more information on qualifier filters, see section Filters and options for reverse engineering in chapter Reverse Engineering in the *PDM User's Guide*.

Example

In Adaptive Server Anywhere 7, a typical qualifier query is:
.`Qualifier.SqlListQuery :`
`select dbspace_name from sysfile`

Syntax in SQL statements

The syntax defined in the DBMS definition is used by PowerDesigner to parse SQL statements during the reverse engineering of a database script. It is used to set the appropriate variables during the creation and modification process of PowerDesigner objects.

You can incorporate variables in the SQL queries of the selected DBMS. PowerDesigner variables are written between percent signs (%).

Reserved keywords

For a domain or a column, standard check fields can indicate minimum, maximum, and default values, as well as a list of values.

In general, if the domain or column data type is a string data type, quotation marks surround these values in the generated script. However, quotation marks are not generated in the following cases:

- ◆ You define a data type that is not recognized as a string data type by PowerDesigner
- ◆ Value is surrounded by tilde characters
- ◆ Value is a reserved keyword defined in the DBMS (for example, NULL)

In addition, if the value is already surrounded by quotation marks additional quotation marks are not generated.

The generation of single- or double-quotation marks depends on the current DBMS.

The DBMS lists reserved keywords as values for the field **ReservedDefault**, under the category **Keywords**.

Example

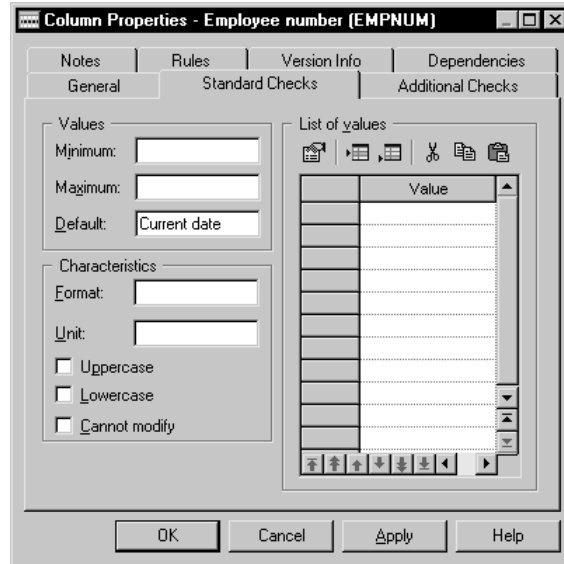
The following example shows the result of a reserved keyword for the DBMS Sybase AS Anywhere 6.

Sybase AS Anywhere 6 DBMS contains the following lines:

```
ReservedDefault =  
NULL  
AUTOINCREMENT  
CURRENT DATE  
CURRENT PUBLISHER  
CURRENT TIME
```

CURRENT_TIMESTAMP
CURRENT_USER
LAST_USER
USER

You define CURRENT DATE as the default value of a column:



The resulting script contains the string CURRENT DATE without quotation marks.

```
create table SALE
(
    SALE_ID          T_IDENTIFIER          not null,
    STOR_ID          T_AN_IDENTIFIER       not null,
    TITLE_ISBN       char(12)              not null,
    SALE_DATE        T_DATE                not null
        default CURRENT DATE,
    SALE_AMOUNT      T_AMOUNT              ,
    SALE_TERMS       T_LONG_TEXT           ,
    SALE_QTY         T_QUANTITY            ,
    primary key (SALE_ID)
);
```

When you run this script, Sybase Adaptive Server Anywhere will recognize CURRENT DATE as a reserved default value.

Optional strings and variables

You can use square brackets [] to:

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

Variable	Generation	Reverse
[%--%]	Generated if variable is defined. If the variable is empty or assigned NO or FALSE it is not generated	Valuated if the parser detects a piece of SQL order corresponding to the variable. If the variable is empty or assigned NO or FALSE it is not valuated
[%--%?: Is true : Is false] to test the value of the variable (conditional value)	If the variable is true, Is true is generated, if the variable is false, Is false is generated	If the parser detects Is true, Is true is reversed, if the parser detects Is false, Is false is reversed
[%--%=--?: Is true : Is false] to test the content of the variable (conditional value)	If the variable equals the constant value, Is true is generated, if the variable is different, Is false is generated	If the parser detects Is true , the constant is added to the variable, if it detects Is false the constant is not added to the variable

Examples

- ◆ [%--%]
[%OPTIONS%]

If %OPTIONS% is not FALSE, the variable is generated, this text is replaced by the value of %OPTIONS% (physical options for the objects visible in the object property sheet).
[default %DEFAULT%]

In reverse engineering, if a text **default 10** is found during reverse engineering, %DEFAULT% is filled with the value 10. However this specification is not mandatory and the SQL statement is reversed even if the specification is absent. In script generation, if default has a value (10 for example) during generation, the text is replaced by **default 10**.
- ◆ [%--%?: Is true : Is false]

You can use a conditional value for an optional string or variable. The two conditions are separated by a colon within the brackets used with the optional string or variable. For example, [%MAND%?Is true:Is false]. If %MAND% is evaluated as true during generation, this text is replaced by **Is true**. If not true, it is replaced by **Is false**.

- ◆ [%--%==--?: Is true : Is false]

You can also use keywords to test the content of a variable.

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

Use of strings

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

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

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

Defining variable formatting options

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

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

You embed formatting options in variable syntax as follows:

```
%.format:variable%
```

The variable formatting options are the following:

Format option	Description
<i>n</i> (where <i>n</i> is an integer)	Blanks or zeros added to the left to fill the width and justify the output to the right
<i>-n</i>	Blanks or zeros added to the right to fill the width and justify the output to the left
.L	Lower-case characters
.U	Upper-case characters
.F	Combined with L and U, applies conversion to first character


Format option	Description
.T	Leading and trailing white space trimmed from the variable
.n	Truncates to <i>n</i> first characters
.-n	Truncates to <i>n</i> last characters
q	Enquotes the variable (single quotes)
Q	Enquotes the variable (double quotes)

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

Example

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

Format	Use	Example	Result
.L	Lower-case characters	PK_%.L:TABLE%	PK_customer_priority
.Un	Upper-case characters + left justify variable text to fixed length where <i>n</i> is the number of characters	PK_%.U12:TABLE%	PK_CUSTOMER_PRI
.T	Trim the leading and trailing white space from the variable	PK_%.T:TABLE%	PK_customer_priority
.n	Maximum length where <i>n</i> is the number of characters	PK_%.8:TABLE%	PK_Customer
.-n	Pad the output with blanks to the right to display a fixed length where <i>n</i> is the number of characters	PK_%.20:TABLE%	PK_ Customer_priority

 For a list of the variables used in PowerDesigner, see section PDM variables.

Constraint name template

DBMS uses variables and variable formatting to define constraint name templates. These are defined by the value of the DBMS field **ConstName**. There are ConstName fields for the following object categories:

Category	Description
TABLE	Constraint name template for table checks
COLUMN	Constraint name template for column checks
PKEY	Constraint name template for primary keys
KEY	Constraint name template for alternate keys
REFERENCE	Constraint name template for foreign keys

In PowerDesigner, you can define user-defined constraint names. Templates apply to all constraints for which you do not define user-defined constraint names.

Code and
Generated code
variables

Many objects have Code and Generated Code variables which are differentiated as follows:

Variable	Description
Code	Attribute code defined in the property sheet
Generated Code	Code computed according to generation options. The generated code can be different from the code in the following cases: Code is a reserved word or contains invalid characters. The generated code is quoted Code is longer than the length authorized by the DBMS. The generated code is truncated

For a list of all the variables used in PowerDesigner, see section PDM variables.

Common naming
variables for
ConstName

The ConstName field for all objects can accept the following common variables:

Variable	Description
%@OBJTNAME%	Object name
%@OBJTCODE%	Object code
%@OBJTLABL%	Comment for the object
%@OBJTDESC%	Description for the object

**TABLE
ConstName**

The ConstName field for the object TABLE can accept the following variables:

Variable	Value
%TABLE%	Generated code of the table
%TNAME%	Table name
%TCODE%	Table code
%TLABL%	Table comment

**COLUMN
ConstName**

The ConstName field for the object COLUMN can accept the following variables:

Variable	Value
%COLUMN%	Generated column code
%COLNAME%	Column name
%COLNCODE%	Column code

PKEY ConstName

The ConstName field for the object PKEY can accept the following variable:

Variable	Value
%CONSTNAME%	Constraint name

KEY ConstName

The ConstName field for the object KEY can accept the following variables:

Variable	Value
%AKEY%	Code of the alternate key
%TABLE%	Code of the table

**REFERENCE
ConstName**

The ConstName field for the object REFERENCE can accept the following variables:

Variable	Value
%REFR%	Generated code of the reference
%PARENT%	Generated code of the parent table
%PNAME%	Parent table name
%PCODE%	Parent table code

Variable	Value
%CHILD%	Generated code of the child table
%CNAME%	Child table name
%CCODE%	Child table code
%PQUALIFIER%	Parent table qualifier
%CQUALIFIER%	Child table qualifier
%REFRNAME%	Reference name
%REFRCODE%	Reference code
%PKCONSTRAINT%	Parent key constraint name used to reference the object
%POWNER%	Parent table owner
%COWNER%	Child table owner
%CHKONCMMT%	TRUE when check on commit is selected on the reference (ASA 6.0 specific)

Example

The following example shows the use of constraint name templates for Sybase Adaptive Server Anywhere 6.

The DBMS Adaptive Server Anywhere 6 contains the following values for the field ConstName:

Category	Comment	Value
PKEY	Constraint name template for Primary Keys	PK_%.U27:TABLE%
REFR	Constraint name template for Foreign Keys	FK_%.U8:CHILD%_%.U9:REFR%_%.U8:PARENT%
KEY	Constraint name template for Alternate Keys	AK_%.U18:AKEY%_%.U8:TABLE%
COLN	Constraint name template for Check of Column	CKC_%.U17:COLUMN%_%.U8:TABLE%
TABL	Constraint name template for Check of Table	CKT_%.U26:TABLE%

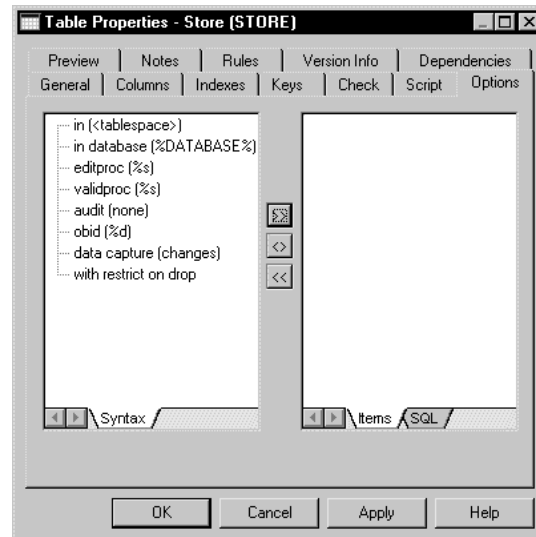
The resulting script that you could generate for a table using Adaptive Server Anywhere 6 declares constraint names as follows:

```
create table DISCOUNT
(
    DISCOUNT_ID      T_IDENTIFIER      not null,
    STOR_ID            T_AN_IDENTIFIER    not null,
    DISC_PERCENT        T_PERCENT         not null,
    DISC_TYPE          T_SHORT_TEXT       null
    constraint CKC_DISC_TYPE_DISCOUNT check
(DISC_TYPE in ('High','Medium','Low')),
    DISC_LOWQTY        T_QUANTITY         null    ,
    DISC_HIGHQTY       T_QUANTITY         null    ,
    constraint PK_DISCOUNT primary key (DISCOUNT_ID)
)
go
```

Defining physical options

In some DBMS definitions, Option entries are used to define a tree view of Physical options in the left pane of an object property sheet. A physical option is a parameter that defines how an object is optimized or stored in a database. Physical options can be included at the end of a Create statement and are DBMS specific.

A typical tree view is shown below:



During generation, the options selected in the model for each object are stored in a variable %OPTIONS%.

The %OPTIONS% variable contains the physical option string as if it was written in a SQL script. This string must appear at the end of the item it belongs to and must not be followed by anything else.

Example

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

is correct syntax.

Variables in physical options

You can use the PowerDesigner variables defined for a given object to set physical options for this object.

For example, in Oracle, you can set the following variable for a cluster if you want the cluster to take the same name as the table.

```
Cluster %TABLE%
```

Extended
Attributes

For a list of all the variables used in PowerDesigner, see section PDM variables.

You can use extended attributes in physical options.

For more information about extended attributes, see section Profile category.

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 ODBC 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.

Depending on the DBMS definition, you can define physical options for the following objects:

- ◆ Tablespace
- ◆ Storage
- ◆ Database
- ◆ Table
- ◆ Column
- ◆ Index
- ◆ Key (primary and alternate)

Typical Physical options are **pctfree**, **pctused**, **fillfactor**, and **partition**.

You define physical options from the property sheet of an object.

Defining physical options specified by a value

Option entries contain text that is analyzed and used to fill the left pane of the physical options page of an object. Each line in an Option entry creates a line in the left pane of the physical options page. The user can click the buttons between the panes of the physical options page to select these lines.

The lines may or may not contain %d or %s variables to let the user specify a value. Example:

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

When you select an option with a **%d** or **%s** value appearing in the right pane, an edit field appears at the bottom of the Physical Options box. It is used when the physical option requires a value. The possible variables are:

Variable	Description
%d	Numeric value follows the physical option. You should type a numeric value in the edit field
%s	String follows the physical option. You should type a string in the edit field

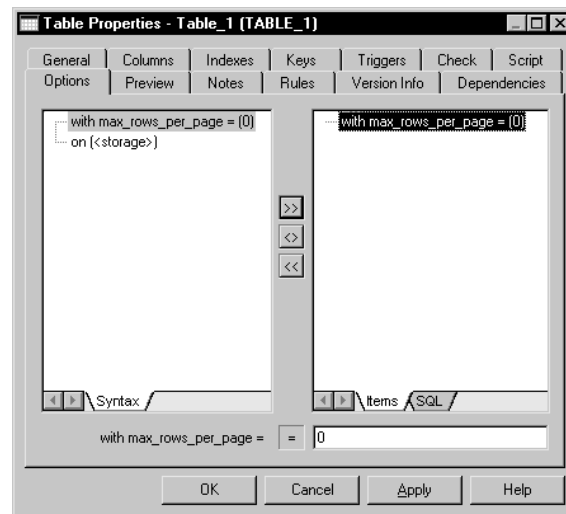
Each selected option creates a line in the right pane. If the selection contains a **%d** variable, a number value can be entered. If the selection contains a **%s** variable, a text is entered. If the selection contains neither a **%d** or **%s** variable, no value is entered.

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

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

Example

The Options page in a table property sheet is shown below.



When the option **with max_rows_per_page** is selected, the Value box appears at the bottom of the page, next to the Equals button. The default is zero (0).

With max_rows_per_page is an index physical option for Sybase AS Enterprise 11.x. This option limits the number of rows per data page. The syntax is the following:

```
with max_row_per_page = x
```

where **x** is the number of rows specified by the user.

In Sybase Adaptive Server Enterprise, this option can be defined for the index object as follows:

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

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

Syntax for a physical option without a name

A line in an option entry should not contain only a variable. It has to contain a text from the physical option or a name in order to be identified by PowerDesigner. A physical option without a text can not be represented in the PowerDesigner syntax. If a physical option does not have any name, you must add a name between angled brackets <> before the option. This indicator is a word describing the physical option to be entered.

Example

To define segment in Sybase AS Enterprise 11, the proper syntax is the following:

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

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

In SYSTEM11, this option is defined as follows:

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

A physical option without name must be followed by the **%d** or the **%s** variable.

How to define a default value for a physical option

A physical option can have a default value. The **Default=** keyword is used to specify this value. After the physical option name or after the **%d** or **%s** value, it is necessary to add a colon **default=x**, where **x** is the default value for the target database.

Example

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

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

The default value is displayed by default in the Options window.

How to define a list of values for a physical option

When you use the **%d** and **%s** variables, a physical option value can correspond to a list of possible options. The **List=** keyword is used to specify this list. After the physical option name, or after the **%d** or **%s** value, it is necessary to add a colon, then **list= x | y co** where **x**, **y** and **z** are the possible values. The values are separated by the **|** (pipe) character.

Example

The **dup_prow** option of a Sybase Adaptive Server Enterprise index corresponds to two mutually exclusive options for creating a non-unique, clustered index. This option can be either ignored or allowed.

In Sybase Adaptive Server Enterprise 11:

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

A dropdown list box with the values is displayed below the right pane of the physical options page.

Use a comma to separate default and list

If **Default=** and **List=** are used at the same time, they must be separated by a comma.

Example

If **ignore_dup_row** is the default value for an index, the syntax used in the DBMS definition file to define this value is the following:

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

How to define a physical option corresponding to a tablespace or a storage

A physical option can use the code of a tablespace or a storage: **category=tablespace** or **category=storage** builds a list with all the tablespace or storage codes defined in the List of Tablespaces or the List of Storages.

Example

The **on segmentname** index option specifies that the index is created on the named segment. A Sybase segment corresponds to a PowerDesigner storage. The syntax is:

```
on segmentname
```

In Sybase Adaptive Server Enterprise 11, this default value for the index object is defined in option items as follows:

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

You can select the correct storage from a dropdown list box in the physical options page.

Composite physical option syntax

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

The standard syntax for composite physical options is as follows:

```
with : composite=yes, separator=yes, parenthesis=no
{
  fillfactor=%d : default=0
  max_rows_per_page=%d : default=0
}
```

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

Keyword	Value and result
composite=	If composite=yes, brackets can be used to define a composite physical option
separator=	If separator=yes, the options are separated by a comma If separator=no, the options have no separator character. This is the default value

Keyword	Value and result
parenthesis=	If parenthesis=yes, the composite option is delimited by parenthesis, including all the other options, for example: with (max_row_per_page=0, ignore_dup_key) If parenthesis=no, nothing delimits the composite option. This is the default value
nextmand=	If nextmand=yes, the next line in the physical option is mandatory. If you do not use this keyword you will not be able to generate/reverse the entire composite physical option
prevmand=	If prevmand=yes, the previous line in the physical option is mandatory. If you do not use this keyword you will not be able to generate/reverse the entire composite physical option
childmand=	If childmand=yes, at least one child line is mandatory
category=	If category=tablespace, the item is linked to a tablespace If category=storage, then item is linked to a storage (*)
list=	List in which values are separated by a pipe ()
dquoted=	If dquoted=yes, the value is enclosed in double quotes ("" "")
squoted=	If squoted=yes, the value is enclosed in single quotes (' ')
enabledbprefix=	If enabledbprefix=yes, the database name is used as prefix (see tablespace options in DB2 OS/390)

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

Example

The IBM DB2 index options contain the following composite option:

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

(*) Special Case
with Oracle

In Oracle, the storage category is used as a template to define all the storage values in a storage entry. This is to avoid having to set values independently each time you need to use the same values in a storage clause. The Oracle physical option does not include the storage name (%s)


```
storage : category=storage, composite=yes, separator=no,
parenthesis=yes
{
```

How to repeat options several times

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

with: **composite=yes, multiple=yes**

For example, the Informix fragmentation options can be repeated n times as shown below:

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

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

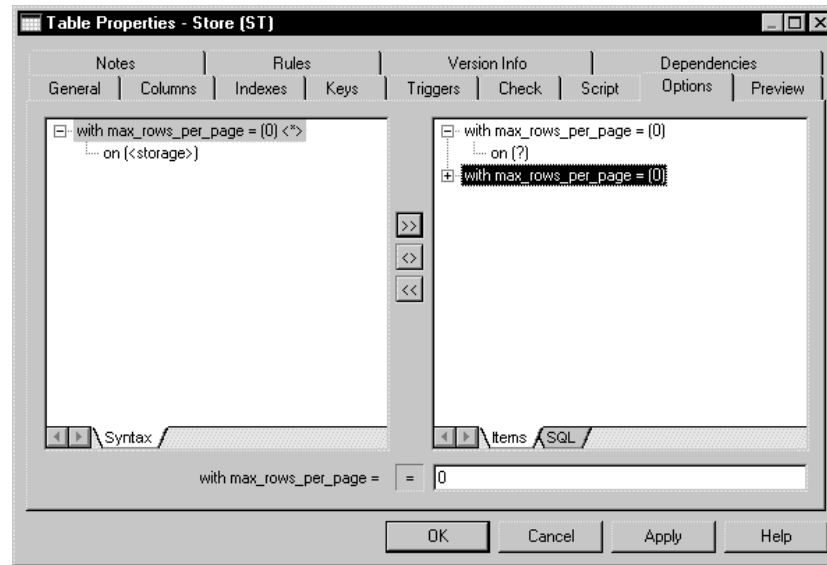
When you repeat a composite option, the option appears with **<*>** in the available physical options pane (left pane) of the physical options page.

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

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

If the selection is on the composite option in the right pane and you click the same composite option in the left pane to add it, a message box asks you if you want to reuse the selected option.

If you click No, the composite option is added to the right pane as a new line.



User interface changes

Your specific user interface may vary from what is shown in our illustrations. This is due to specific characteristics of the DBMS you are using or modifying and on the selected entry parameters.

If an option parameter is empty for an object, or if it is not enabled, the interface changes as follows:

- ◆ The corresponding Options tab does not appear in the object property sheet. This is applicable to Tablespace, Storage, Database, Primary key and Keys
- ◆ The Physical Options check boxes in the Generate Database window, Modify Database window, and Database page of the Generate Database box are grayed and inaccessible. This is applicable to Table, Index and Database

General category

The General category contains DBMS identification information.

EnableCheck

Determines if the generation of check parameters is authorized or not.

Value	Result
Yes	Check parameters generated
No	All variables linked to Check parameters will not be evaluated during generation and reverse

EnableIntegrity

Allows integrity constraints in the DBMS. This controls whether generation parameters for primary, foreign, and alternate keys are grayed or available.

Value	Result
Yes	Primary key, alternate key, and foreign key check boxes are available for database generation and modification
No	Primary key, alternate key, and foreign key check boxes are grayed and unavailable for database generation and modification

EnableMultiCheck

Determines if the generation of multiple check parameters for tables and columns is authorized or not.

Value	Result
Yes	Multiple check parameters are generated. The first constraint in the script corresponds to the concatenation of all validation business rules, the other constraints correspond to each constraint business rules attached to an object
No	All business rules (validation and constraint) are concatenated into a single constraint expression

Enableconstname

Determines if constraint names are used during generation.

Value	Result
Yes	Constraint names are used during generation
No	Constraint names are not used

SqlSupport

Determines if SQL syntax is allowed. This does not affect script generation but it impacts SQL Preview.

Value	Result
Yes	SQL syntax allowed and SQL Preview available
No	SQL syntax not allowed SQL Preview is not available

UniqConstName

Determines if unique constraint names for objects are authorized or not.

Value	Result
Yes	All constraint names must be unique in the database including index names
No	Constraint names must be unique for an object

Check model takes this entry into account in constraint name checking.

Script or ODBC categories

The Script category is used for the Script generation and the ODBC category is used for the ODBC generation. When an entry value or its category is missing from ODBC category during the ODBC generation, it is automatically searched for in the Script category.

The Script or ODBC categories are divided into the following categories:

- ◆ SQL
- ◆ Objects
- ◆ Data Type

All the entries in each category are described in the following sections.

SQL category

The SQL category contains the following categories:

- ◆ Syntax
- ◆ Format
- ◆ File
- ◆ Keywords

These contain entries that define the SQL syntax for the DBMS.

Syntax

The Syntax category contains entries that define DBMS-specific syntax.

Terminator

End of statement character. It is used to terminate statements such as the create table, view, index, or the open/close database statements.

Example

```
Terminator = ;
```

If empty, **BlockTerminator** is used instead.

BlockTerminator

End of block character. It is used to end expressions for triggers and stored procedures.

Example

```
BlockTerminator = /
```

UseBlockTerm

Syntax for use of **BlockTerminator**.

Value	Result
Yes	BlockTerminator is always used
No	BlockTerminator is used for triggers and stored procedures only

Example

In Oracle 8I:

```
UseBlockTerm = yes
```

Delimiter

Field separation character. For example: col1, col2, col3.

Example

Delimiter = ,

Columns are separated by commas in the create table statement.

```
create table CUSTOMER
(
    CUSNUM    numeric(5)           not null,
    CUSNAME   char(30)            not null,
    CUSADDR   char(80)            not null,
    CUSACT    char(80)            ,
    CUSTEL    char(12)            ,
    CUSFAX    char(12)            ,
    primary key (CUSNUM)
);
```

Quote

Character used to enclose string values. It is usually either a single or a double quotation mark.

Example

Quote = '

This quote is used to add quotes around default values in generated SQL.

Quotes must be the same

The same quote must be used in the check parameter tab to enclose reserved words used as default.

SqlContinue

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.

BlockComment

Character used to enclose a multi-line commentary.

Used in reverse engineering and generation.

Example

BlockComment = /* */

Add a space between the opening and closing characters.

LineComment

Character used to enclose a single line commentary.

Used in reverse engineering.

Example

```
LineComment = %%
```

Format

The Format category contains entries that define script formatting.

IllegalChar

Invalid characters for names. This is only used for generation. If there is an illegal character in a Code, the code is set between quotes during generation.

Example

```
IllegalChar = "+-*/!=<>'\" () "  
If the name of the table is "SALES+PROFITS", the  
generated create statement will be:  
CREATE TABLE "SALES+PROFITS"
```

Double quotes are placed around the table name to indicate that an invalid character is used.

During reverse engineering, any illegal character is considered as a separator unless it is located within a quoted name.

CaseSensitivityUsingQuote

Determines if the case sensitivity for identifiers is managed using double quotes. You should set this boolean to Yes if the DBMS you are using needs double quotes to preserve the case of object codes.

UpperCaseOnly

Uppercase only. When generating a script from a PDM, all objects (tables, columns, constraints, index, and so on) can be forced to be generated in uppercase independently from the model Naming Conventions (uppercase, lowercase, or mixed case) and the PDM codes.

Value	Result
Yes	Forces all generated script characters to uppercase
No	Generates all script unchanged from the way objects are written in the model (uppercase, lowercase or mixed case is kept)

Example

UpperCaseOnly = NO

Conflict between UpperCaseOnly and LowerCaseOnly

The **UpperCaseOnly** field is ignored if **UpperCaseOnly** and **LowerCaseOnly** entries are both set to **Yes**. In this case, the PowerDesigner script is generated in *lowercase*.

LowerCaseOnly

Lowercase only. When generating a script from a PDM, all objects (tables, columns, constraints, index, and so on) can be forced to be generated in lowercase independently from the model Naming Conventions (uppercase, lowercase, or mixed case) and the PDM codes.

Value	Result
Yes	Forces all generated script characters to lowercase
No	Generates all script unchanged from the way objects are written in the model (uppercase, lowercase or mixed case is kept)

Example

LowerCaseOnly = YES

EnableOwnerPrefix and EnableDtbsPrefix

Object codes can be prefixed by the object owner, the database name or both, to be uniquely identified. This is done using variable %QUALIFIER%.

If you want the prefix to be the object owner, you have to set the value of the EnableOwnerPrefix entry to Yes, and select the Owner Prefix check box in the generation dialog box. If you want the prefix to be the database name, you have to set the value of the EnableDtbsPrefix entry to Yes and select the Database Prefix check box in the generation dialog box.

If you select both Owner Prefix and Database Prefix check boxes, both names will be concatenated when %QUALIFIER% will be evaluated.

Value	Result
Yes	The Owner Prefix and Database Prefix check boxes are available in the Database Generation box. The variable %QUALIFIER% is filled with the object owner name, or the database name, or both during generation or reverse engineering
No	The Owner Prefix and Database Prefix check boxes are grayed and the prefix unavailable

MaxScriptLen

Indicates the maximum length of a script line.

Example 1024

AddQuote

Determines if object codes are systematically enquoted during the generation.

Value	Result
Yes	Quotes are systematically added to object codes during generation
No	Object codes are generated without quotes

Date and time format

You can customize the date and time format for test data generation by script or ODBC using DBMS entries in the Format category.

PowerDesigner uses the **PhysDataType** map entry in the script\data types category to convert the physical data types of columns to conceptual data types because the DBMS entries 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 ODBC
datetime	DT	DateTimeFormat	OdbcDateTimeFormat
timestamp	TS	DateTimeFormat	OdbcDateTimeFormat
date	D	DateFormat	OdbcDateFormat
time	T	TimeFormat	OdbcTimeFormat

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

The default date and time format is the following:

◆ SQL: 'yyyy-mm-dd HH:MM:SS'

◆ ODBC: {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 entry for SQL: **yy-mm-dd HH:MM**. For ODBC, this entry should have the following value: {ts 'yy-mm-dd HH:MM'}.

File

The File category contains entries that define script formatting.

Header

Header text for a database generation script.


Footer

Footer text for a database generation script.

EnableMultiFile

Multi-script allowed. This variable acts on the availability of the One File Only check box in the Generate database, Generate Triggers and Procedures, and Modify Database parameters windows.

Value	Result
Yes	<p>The One File Only check box is available. If you deselect this check box, one script is created for each table, and a general script will sum up all the single table script entries. Each single script has the table name and the extension defined in TableExt variable. If you select this check box, a unique script will include all the orders.</p> <p>The general script has the file extension described in the DBMS variable ScriptExt, and its name is customizable in the File Name entry of the generation or modification parameters boxes. The default names are CREBAS for database generation, CRETRG for triggers and stored procedures generation, and ALTER for database modification</p>
No	<p>The One File Only check box is grayed, and a unique script will include all the orders.</p> <p>The file extension is described in the variable ScriptExt, and its name is customizable in the File Name entry of the generation or modification parameters window. The default names are CREBAS for database generation, CRETRG for triggers and stored procedures generation, and ALTER for database modification</p>

 For information on variables used with PowerDesigner, see section PDM variables.

Example

```
EnableMultiFile = YES
```

ScriptExt

The variable **ScriptExt** defines the default script extension when you generate a database or modify a database for the first time.

Example

```
ScriptExt = sql
```

TableExt

If you do not select the “One File Only” check box when you generate a database, or when you modify a database, one script is generated for each table, with the file extension defined in the variable **TableExt**.

Example

```
TableExt = sql
```

See also *EnableMultiFile*.

StartCommand

Statement for executing a script. This parameter corresponds to the %STARTCMD% variable. The value of this variable depends on this parameter. The start statement corresponds to the statement used by the target database to execute a SQL script.

The contents of this variable is used inside the header file of a multi-file generation. It is used to call all the other generated files from the header file.

Example

In Sybase AS Enterprise 11.x:

```
StartCommand = isql %NAMESCRIPT%
```

Usage1

Applicable when using a single script.

It is displayed at the end of the generation in the Output window.

Example

- (1) Start the SQL interpreter: ISQL
- (2) Open the script %NAMESCRIPT%
- (3) Execute the script

Usage2

Applicable when using multiple scripts.

It is displayed at the end of the generation in the Output window.

Example

- (1) Start the SQL interpreter: ISQL
- (2) Open one script from the directory %PATHSCRIPT%
- (3) Execute this script

TriggerExt

Variable that defines the main script extension when you generate triggers and stored procedures for the first time.

Example

```
TriggerExt = trg
```

TrgUsage1

Applicable when using a single script for triggers and procedures generation.

It is displayed at the end of the generation in the Output window.

Example

- (1) Start the SQL interpreter: ISQL
- (2) Select the item "Options" of the menu "Command"
- (3) Change "Command Delimiter" to /
- (4) Open the script %NAMESCRIPT% from the directory %PATHSCRIPT%

- (5) Execute the script
- (6) Restore "Command Delimiter" to ;

TrgUsage2

Applicable when using multiple scripts for triggers and procedures generation.

It is displayed at the end of the generation in the Output window.

Example

- (1) Start the SQL interpreter: ISQL
- (2) Select the item "Options" of the menu "Command"
- (3) Change "Command Delimiter" to /
- (4) Open one script from the directory %PATHSCRIPT%
- (5) Execute the script
- (6) Restore "Command Delimiter" to ;

TrgHeader

Header script for triggers and procedures generation.

Example

In Oracle 8:

```
(1) Start the SQL interpreter: ISQL
(2) Select the item "Options" of the menu "Command"
Integrity package declaration
create or replace package IntegrityPackage AS
  procedure InitNestLevel;
  function GetNestLevel return number;
  procedure NextNestLevel;
  procedure PreviousNestLevel;
end IntegrityPackage;
/
-- Integrity package definition
create or replace package body IntegrityPackage AS
  NestLevel number;

-- Procedure to initialize the trigger nest level
procedure InitNestLevel is
begin
  NestLevel := 0;
end;

-- Function to return the trigger nest level
function GetNestLevel return number is
begin
  if NestLevel is null then
    NestLevel := 0;
  end if;
  return(NestLevel);
end;

-- Procedure to increase the trigger nest level
procedure NextNestLevel is
begin
  if NestLevel is null then
```

```
        NestLevel := 0;
    end if;
    NestLevel := NestLevel + 1;
end;

-- Procedure to decrease the trigger nest level
procedure PreviousNestLevel is
begin
    NestLevel := NestLevel - 1;
end;

end IntegrityPackage;
```

TrgFooter

Footer text for a script (triggers and procedures generation).

AlterHeader

Header text for a script (modify database).

AlterFooter

Footer text for a script (modify database).

Keywords

The Keywords category contains keywords and values that are reserved for special tasks and cannot be used as variable names or values.

ReservedWord

Reserved words.

If a reserved word is used as an object code, it is enquoted during generation (using quotes only in DBMS → Script → SQL → Syntax → Quote)

Example

In Adaptive Server Anywhere 6:

```
TABLE
CREATE
DELETE
WHenever
```

ReservedDefault

Reserved default values.

If a reserved word is entered in a default value, the value will not be enquoted, it is used as default value for columns.

Example

In Adaptive Server Anywhere 6, USER is a reserved default value:

```
Create table CUSTOMER (  
  Username varchar(30) default USER,  
)
```

GroupFunc

List of SQL functions to use with group keywords.

Used in SQL editor (views) to propose a list of available functions to help in entering SQL code. These functions are exactly the same as available in the DBMS.

Example

```
avg()  
count()  
max()  
min()
```

NumberFunc

List of SQL functions used on numbers.

Used in SQL editor (views) to propose a list of available functions to help in entering SQL code. These functions are exactly the same as available in the DBMS.

Example

```
abs()  
acos()  
asin()  
atan()
```

CharFunc

List of SQL functions for characters and strings.

Used in SQL editor (views) to propose a list of available functions to help in entering SQL code. These functions are exactly the same as available in the DBMS.

Example

```
ascii()  
char()  
charindex()  
char_length()  
difference()  
lower()
```


DateFunc

List of SQL functions for dates.

Used in SQL editor (views) to propose a list of available functions to help in entering SQL code. These functions are exactly the same as available in the DBMS.

Example

```
dateadd()  
datediff()  
datetime()
```

ConvertFunc

List of SQL functions used to convert values between hex and integer and handling strings.

Used in SQL editor (views) to propose a list of available functions to help in entering SQL code. These functions are exactly the same as available in the DBMS.

Examples

```
convert()  
hextoint()  
inttohex()
```

OtherFunc

List of SQL functions for estimating, concatenating and SQL checks.

Used in SQL editor (views) to propose a list of available functions to help in entering SQL code. These functions are exactly the same as available in the DBMS.

Example

```
db_id()  
db_name()  
host_id()
```

ListOperators

List of SQL operators for comparing values, boolean, and various semantic operators.

Used in SQL editor (views) to propose a list of available functions to help in entering SQL code. These functions are exactly the same as available in the DBMS.

Example

```
=  
!=  
not like  
not in
```

Commit

Statement for validating the transaction by ODBC.

Objects category

The Objects category defines each type of object that is supported by the DBMS definition.

At the root of the Objects category, the following two entries are defined for all objects of the DBMS:

- ◆ EnableOption: boolean that specifies if physical options are allowed for DBMS objects
- ◆ MaxConstLen: Sets the value for the constraint name length for tables, columns, primary and foreign keys

Common object entries

The following entries are common to many objects in the DBMS definition:

Enable

Statement for determining if an object is allowed.

Possible values: YES/NO.

Example

```
Enable = Yes
```

Maxlen

Statement for defining the maximum code length for an object. This value is implemented in the Check model and produces an error if the code exceeds the defined value. The object code is also truncated at generation time.

Example

```
MaxLen = 128
```

Create

Statement for creating an object (generation and reverse).

Example

```
create table %TABLE%
```

Add

Statement for adding an object inside the creation statement of another object (generation and reverse).

For example, **Table create** will call **Column add**.

Drop

Statement for dropping an object (generation only).


Example

In Sybase Adaptive Server Anywhere 6:

```
if exists(select 1 from sys.systable where
table_name='%TABLE%' and table_type='BASE' [%QUALIFIER?
and creator=user_id('%OWNER%')]) then
    drop table [%QUALIFIER%]%TABLE%
end if
```


BeforeCreate, BeforeDrop, BeforeModify

Extended statements executed before the main Create, Drop or Modify statements.

 For more information on extension statements, see section .

AfterCreate, AfterDrop, AfterModify

Extended statements executed after the main Create, Drop or Modify statements.

 For more information on extension statements, see section Script generation.

Options

Physical options for creating an object (generation and reverse).

Example

In Sybase Adaptive Server Anywhere 6:

```
in %s : category=tablespace
```

DefOptions

Default values for object physical options that will be applied to all objects. These values must respect SQL syntax.

For example:

```
in default_tablespace
```

SqlListQuery

SQL Query for listing objects in reverse engineering dialog.

The query is executed to fill header variables and create objects in memory.

Example

```
{OWNER, TABLE, COLUMN, DTTPCODE, LENGTH, SIZE, PREC,
NOTNULL (N='NOT NULL', *=NULL), DEFAULT, COMMENT}

[%ISODBCUSER% ?
SELECT '%SCHEMA%', C.TABLE_NAME, C.COLUMN_NAME,
C.DATA_TYPE, C.DATA_PRECISION, C.DATA_LENGTH,
C.DATA_SCALE, C.NULLABLE, C.DATA_DEFAULT, M.COMMENTS
FROM SYS.USER_COL_COMMENTS M, SYS.USER_TAB_COLUMNS C
WHERE M.TABLE_NAME = C.TABLE_NAME AND M.COLUMN_NAME =
C.COLUMN_NAME
      [AND C.TABLE_NAME='%TABLE%']
ORDER BY C.TABLE_NAME, C.COLUMN_ID
:
SELECT C.OWNER, C.TABLE_NAME, C.COLUMN_NAME,
C.DATA_TYPE, C.DATA_PRECISION, C.DATA_LENGTH,
C.DATA_SCALE, C.NULLABLE, C.DATA_DEFAULT, M.COMMENTS
FROM SYS.ALL_COL_COMMENTS M, SYS.ALL_TAB_COLUMNS C
WHERE M.OWNER = C.OWNER AND M.TABLE_NAME = C.TABLE_NAME
AND M.COLUMN_NAME = C.COLUMN_NAME
      [AND C.OWNER='%SCHEMA%'] [AND
C.TABLE_NAME='%TABLE%']
ORDER BY C.OWNER, C.TABLE_NAME, C.COLUMN_ID
]
```

SqlAttrQuery

SQL Query to retrieve additional information on object reversed by **SqlListQuery**.

Example

In Oracle 8:

```
{OWNER, TABLE, COMMENT}

[%ISODBCUSER% ?
SELECT '%SCHEMA%', TABLE_NAME, COMMENTS
FROM SYS.USER_TAB_COMMENTS
WHERE COMMENTS IS NOT NULL [AND TABLE_NAME='%TABLE%']
ORDER BY TABLE_NAME
:
SELECT OWNER, TABLE_NAME, COMMENTS
FROM SYS.ALL_TAB_COMMENTS
WHERE COMMENTS IS NOT NULL [AND OWNER='%SCHEMA%'] [AND
TABLE_NAME='%TABLE%']
ORDER BY OWNER, TABLE_NAME
]
```

See also **SqlListQuery**.

SqlOptsQuery

SQL Query to retrieve object physical options from object reversed by **SqlListQuery**.

The result of the query will fill the variable %OPTIONS% and must respect SQL syntax.

Example

In Sybase Adaptive Server Anywhere 6:

```
{OWNER, TABLE, OPTIONS}
select su.USER_NAME, st.TABLE_NAME, 'in '+ dbspace_name
from SYS.SYSUSERPERMS su, SYS.SYSTABLE st, SYS.SYSFILE
sf
where
st.file_id = sf.file_id and dbspace_name <> 'SYSTEM' and
[%TABLE% ? TABLE_NAME = '%TABLE%' and] [%SCHEMA% ?
su.USER_NAME = '%SCHEMA%' and]
st.CREATOR = su.USER_ID
```

See also **SqlListQuery**.

SqlFragQuery

See section Extension mechanism for ODBC reverse engineering queries.

ModifiableAttributes

List of extended attributes that will be taken into account in the merge dialog box during database synchronization.

🌀 For more information, see section Script generation.

Example

In Sybase ASE 12.5

```
ExtTablePartition
```

ReversedStatements

List of statements that will be reverse engineered.

🌀 For more information, see section Script reverse engineering.

Example

In Sybase ASE 12.5

```
CreatePartition
```

Table

The Table category contains entries defining table related parameters.

Common entries for Table

You can define values for the following common entries for the Table object in the DBMS definition:

Entry	Example
Enable	In Sybase Adaptive Server Anywhere 6: Enable = Yes
Maxlen	In Sybase Adaptive Server Anywhere 6: Maxlen = 128
Create	In Sybase Adaptive Server Anywhere 6: create table [%QUALIFIER%]%TABLE% (%TABLDEFN%) [%OPTIONS%]
Drop	In Sybase Adaptive Server Enterprise 11: if exists (select 1 from sysobjects where id = object_id(['%QUALIFIER%]%TABLE%') and type = 'U') drop table [%QUALIFIER%]%TABLE%
Options	In Sybase Adaptive Server Enterprise 11: with max_rows_per_page = %d : default=0 on %s : category=storage
DefOptions	This entry is often empty. It defines options that are applied to all tables such as with_max_row_per_page = 128
SqlListQuery	In Oracle 7: {OWNER, TABLE} select owner, table_name from sys.all_tables where 1=1 [and owner='%SCHEMA%'] [and table_name='%TABLE%'] order by owner, table_name

Entry	Example
SqlAttrQuery	<p>In Oracle 7:</p> <pre>{OWNER, TABLE, COMMENT} select owner, table_name, comments from sys.all_tab_comments where comments is not null [and owner='%SCHEMA%'] [and table_name='%TABLE%']</pre>
SqlOptsQuery	<p>In Sybase Adaptive Server Anywhere 6:</p> <pre>{OWNER, TABLE, OPTIONS} select su.USER_NAME, st.TABLE_NAME, 'in '+ dbspace_name from SYS.SYSUSERPERMS su, SYS.SYSTABLE st, SYS.SYSFILE sf where st.file_id = sf.file_id and dbspace_name <> 'SYSTEM' and [%TABLE% ? TABLE_NAME = '%TABLE%' and] [%SCHEMA% ? su.USER_NAME = '%SCHEMA%' and] st.CREATOR = su.USER_ID</pre>
SqlFragQuery	<p>In Oracle 9I:</p> <pre>{VAR1NAME, VAR1VALUE} select 'HighVal' tp.partition_position, tp.high_value from all_tab_partitions tp where 1=1 [and tp.table_owner=.q:OWNER%] [and tp.table_name=.q:TABLE%]</pre>
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

Permission

Available permissions for tables. The first column displays the SQL name of permission (SELECT for example). The second column is the shortname that appears in the title of grid columns.

Example

In Sybase ASE 12.5:
SELECT / Sel
INSER / Ins
DELETE / Del
UPDATE / Upd
REFERENCES / Ref

ConstName

Constraint name template for a table check parameter.

Used in the table property sheet to fill the constraint name.

Example

In Sybase Adaptive Server Anywhere 6:
CKT_%.U26:TABLE%

TableComment

Statement for adding a table comment. The table comment is a SQL statement which is not supported by all DBMS. If **TableComment** is empty, the Comment check box in the Tables and Views page of the Database Generation box is grayed and unavailable.

Used in generation and reverse engineering.

Example

In Sybase SQL Anywhere 5.5:
TableComment = comment on table %OWNERPREFIX%%TABLE% is
'%COMMENT%'

The %TABLE% variable is the name of the table defined in the List of Tables, or in the table property sheet. The %COMMENT% variable is the comment defined in the Comment textbox of the table property sheet.

Rename

Statement for renaming a table. If **Rename** is empty, 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.

Used in the database modification script when a table has been renamed.

Example

In Sybase Adaptive Server Enterprise 11:

```
sp_rename %OLDTABL%, %NEWTABL%
```

The %OLDTABL% variable is the code of the table before it was renamed.
The %NEWTABL% variable is the new code of the table.

AlterTableHeader

Alter table header. Anything added to this entry is added before the **alter table** statement. You can place an alter table header in your scripts to document or perform initialization logic.

Example

```
AlterTableHeader = /* Table name: %TABLE% */
```

AlterTableFooter

Alter table footer. Anything added to this entry is added after the **alter table** statement and before the terminator.

Example

```
AlterTableFooter = /* End of alter statement */
```

DefineTableCheck

Statement for customizing the script of table constraints (checks) within a **create table** statement.

Example

```
check (%CONSTRAINT%)
```

AddTableCheck

Statement for customizing the script to modify the table constraints within an **alter table** statement.

Example

```
alter table [%QUALIFIER%]%TABLE%  
add check (%CONSTRAINT%)
```

DropTableCheck

Statement for dropping a table check in an **alter table** statement.

Example

```
alter table [%QUALIFIER%]%TABLE%  
delete check
```

SqlChckQuery

SQL Query to reverse engineer table checks.

Example

In Sybase Adaptive Server Anywhere 6:

```
{OWNER, TABLE, CONSTRAINT}
SELECT U.USER_NAME, T.TABLE_NAME, T.VIEW_DEF
FROM SYSUSERPERMS U, SYSTABLE T
WHERE [U.USER_NAME = '%OWNER%' AND] [T.TABLE_NAME =
'%TABLE%' AND]
T.CREATOR = U.USER_ID AND T.TABLE_TYPE = 'BASE' AND
T.VIEW_DEF IS NOT NULL
```

SqlPermQuery

SQL Query to reverse engineer permissions granted on tables.

Example

In Sybase Adaptive Server Enterprise 12.5:

```
{ select ul.name grantee,
case
    when (s.action = 193) then 'SELECT'
    when (s.action = 195) then 'INSERT'
    when (s.action = 196) then 'DELETE'
    when (s.action = 197) then 'UPDATE'
end +
case
    when (s.protecttype = 0) then '+'
    when (s.protecttype = 1) then ''
    when (s.protecttype = 2) then '-'
end
|| ', '
from sysprotects s, sysusers u, sysusers ul, sysobjects
o
where
    o.name = %.q:TABLE% and
    o.uid = u.uid and
    s.id = o.id and
    ul.uid = s.uid
```

SqlListRefrTables

SQL query used to list the tables referenced by a table.

Example

In Oracle 9i:

```
{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%]
order by 1, 2, 3, 4
```

UniqConstraintName

Disallows the use of the same name for index and constraint name in the same table.

Value	Result
Yes	The table constraint name and index name must be different. This is taken into account during the check of the model
No	The table constraint name and index name can be identical

Header

Table header. Anything added to this entry is added before the **create table** statement.

Footer

Table footer. Anything added to this entry is added after the **create table** statement.

AllowedADT

List of abstract data types on which a table can be based. You can assign object abstract data types to tables. The table uses the properties of the abstract data type and the abstract data type attributes become table columns.

This list of ADT appears in the Based On dropdown list box in the table property sheet.

Example

In Oracle 8I2:
OBJECT

In DB2 UDB 5.x:
STRUCTURED

MaxConstLen

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

Maximum length for constraint names:

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.

Example

MaxConstLen = 128

Column

The Column category contains entries defining column related parameters.

Common entries for Column

You can define values for the following common entries for the Column object in the DBMS definition.

Entry	Example (Sybase Adaptive Server Enterprise 11)
Enable	Enable = Yes
Maxlen	Maxlen = 30
Create	alter table [%QUALIFIER%]%TABLE% add %20:COLUMN% %30:DATATYPE% [default %DEFAULT%] [%IDENTITY%?identity:%NULL%] [[constraint %CONSTNAME%] check (%CONSTRAINT%)]

Entry	Example (Sybase Adaptive Server Enterprise 11)
Add	<pre>%20: COLUMN% %30: DATATYPE% [default %DEFAULT%] [%IDENTITY%?identity:[%NULL%][%NOTNULL%]] [[constraint %CONSTNAME%] check (%CONSTRAINT%)]</pre>
Drop	<p>In Sybase Adaptive Server Anywhere 6:</p> <pre>alter table [%QUALIFIER%]%TABLE% delete %COLUMN%</pre>
DefOptions	<p>This entry is often empty. It defines options that are applied to columns</p>
SqlListQuery	<p>In Oracle 8I2:</p> <pre>{OWNER, TABLE, COLUMN, DTTPCODE, LENGTH, SIZE, PREC, NOTNULL (N='NOT NULL', *=NULL), DEFAULT, COMMENT} [%ISODBCUSER% ? SELECT '%SCHEMA%', C.TABLE_NAME, C.COLUMN_NAME, C.DATA_TYPE, C.DATA_PRECISION, C.DATA_LENGTH, C.DATA_SCALE, C.NULLABLE, C.DATA_DEFAULT, M.COMMENTS FROM SYS.USER_COL_COMMENTS M, SYS.USER_TAB_COLUMNS C WHERE M.TABLE_NAME = C.TABLE_NAME AND M.COLUMN_NAME = C.COLUMN_NAME [AND C.TABLE_NAME='%TABLE%'] ORDER BY C.TABLE_NAME, C.COLUMN_ID : SELECT C.OWNER, C.TABLE_NAME, C.COLUMN_NAME, C.DATA_TYPE, C.DATA_PRECISION, C.DATA_LENGTH, C.DATA_SCALE, C.NULLABLE, C.DATA_DEFAULT, M.COMMENTS FROM SYS.ALL_COL_COMMENTS M, SYS.ALL_TAB_COLUMNS C WHERE M.OWNER = C.OWNER AND M.TABLE_NAME = C.TABLE_NAME AND M.COLUMN_NAME = C.COLUMN_NAME [AND C.OWNER='%SCHEMA%'] [AND C.TABLE_NAME='%TABLE%'] ORDER BY C.OWNER, C.TABLE_NAME, C.COLUMN_ID</pre>

Entry	Example (Sybase Adaptive Server Enterprise 11)
SqlAttrQuery	<p>In Oracle 7:</p> <pre>{OWNER, TABLE, COLUMN, COMMENT} select c.owner, c.table_name, c.column_name, m.comments from sys.all_col_comments m, sys.all_tab_columns c where m.owner=c.owner and m.table_name=c.table_name and m.column_name=c.column_name [and c.owner='%SCHEMA%'] [and c.table_name='%TABLE%'] [and c.column_name='%COLUMN%']</pre>
SqlOptsQuery	-
SqlFragQuery	-
Options	<p>In DB2 UDB 6.x common server:</p> <pre><logged> %s : list=logged not logged, default=logged <compact> %s : list=compact not compact, default=not compact linktype url : composite=yes { no link control file link control : composite=yes { mode db2options integrity %s : default=all read permission %s : list=fs db, default=fs write permission %s : list=fs blocked, default=fs recovery %s : list=yes no, default=no on unlink %s : list=restore delete, default=restore } }</pre>
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

Permission

Available permissions for columns. The first column displays the SQL name of permission (SELECT for example). The second column is the shortname that appears in the title of grid columns.

Example

In Sybase ASA 8:
 REFERENCES / References
 SELECT / Select
 UPDATE / Update

AltEnableAddColnChk

Indicates 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.

Value	Result
Yes	AddColnChk can be used to modify the column check constraint in an alter table statement.
No	The modify database process does the following: Drops the foreign key constraints Creates a temporary table with the same structure and columns as the table before the column check constraint addition Inserts the rows from the old table to the new temporary table Drops the actual table Creates a new table with the new column check constraint Inserts the rows from the temporary table to the new table Creates the indexes and the constraints on the new table

Example

AltEnableAddColnChk = NO

See also **AddColnChk**.

CheckNull

Verifies if a column can be null.

Value	Result
Yes	Check model verifies if columns can be Null
No	No check on columns null value

EnableBindRule

Allows or disallows the binding of a business rule to a column for check parameters.

Value	Result
Yes	The Create and Bind entry of Rule are generated
No	The check is generated inside the column Add order

Example

EnableBindRule = No

EnableComputedColn

Statement to allow the use of computed columns.

Value	Result
Yes	Computed columns allowed
No	Computed columns not allowed

EnableDefault

Allows the use of predefined default values.

Value	Result
Yes	The default value is generated for columns when the default value is defined. The default value can be defined in the check parameters for each column. The %DEFAULT% variable will contain the default value Default Value check box for columns must be selected in the Tables & Views page of the Database Generation box
No	The Default Value check box for columns in the Tables and Views page of the Database Generation box is grayed. The default value can not be generated

Example

In SQL Anywhere 5.5:

```
EnableDefault = YES
```

where the default value for the column employee function **EMPFUNC** is Technical Engineer, the generated script is:

```
create table EMPLOYEE
(
    EMPNUM          numeric(5)          not null,
```

```
EMP_EMPNUM numeric(5)
DIVNUM      numeric(5)          not null,
EMPFNAM     char(30)
EMPLNAM     char(30)          not null,
EMPFUNC     char(30)
            default 'Technical Engineer',
EMPSAL      numeric(8,2)
primary key (EMPNUM)
);
```

EnableIdentity

Identity keyword entry support. Identity columns are serial counters maintained by the database (for example Sybase and Microsoft SQL Server).

Value	Result
Yes	Enables the Identity check box in the column property sheet
No	The Identity check box does not appear in the column property sheet

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 a null column: when checking the Identity check box, the Mandatory check box is automatically selected.

PowerDesigner ensures that:

- ◆ Only one identity column can be defined per table: when the identity is set for a column, other columns of the table have their identity check box grayed and unavailable
- ◆ A foreign key cannot be an identity column: the check box is grayed and unavailable
- ◆ Identity is only supported for certain data types: when the Identity check box is selected for a column whose data type does not match with identity, the data type is automatically changed to *numeric*. When the data type of an identity column is replaced by a data type that does not match the identity, PowerDesigner displays the error message *Identity cannot be used with the selected data type*.

Example

In Sybase AS Enterprise 11.x:

```
EnableIdentity=YES
```

The employee number column is an identity, the following script is generated:

```
create table EMPLOYEE
(
```

```

EMPNUM      numeric(5)          identity,
EMP_EMPNUM  numeric(5)          null      ,
DIVNUM      numeric(5)          not null,
EMPFNAM     char(30)            null      ,
EMPLNAM     char(30)            not null,
EMPFUNC     char(30)            null      ,
EMPSAL      numeric(8,2)        null      ,
constraint PK_EMPLOYEE primary key (EMPNUM)
)
go

```

Changing the contents of IDENTITY

During generation, the identity variable contains "identity" but you can easily change the Identity keyword using the following syntax when needed:

```
[%IDENTITY%?new identity keyword]
```

EnableNull

Allows the use of the NULL keyword.

Value	Target database	Result
Yes	Supports NULL keyword	Columns that are not checked as Mandatory in the PDM is generated with NULL keyword after the data type
No	Does not support NULL keyword	Columns that are not checked as Mandatory in the PDM is generated without NULL keyword after the data type

Example

In Sybase AS Enterprise 11.x:

```
EnableNull = YES
```

where **CUSACT**, **CUSTEL**, **CUSFAX** are *not* mandatory columns, the null keyword is set beside the column on the **create table** script.

```

create table CUSTOMER
(
  CUSNUM      numeric(5)          not null,
  CUSNAME     char(30)            not null,
  CUSADDR     char(80)            not null,
  CUSACT      char(80)            null    ,
  CUSTEL      char(12)            null    ,
  CUSFAX      char(12)            null    ,
  constraint PK_CUSTOMER primary key (CUSNUM)
)
go

```

If

```
EnableNull = NO
```

The SQL script is:

```
create table CUSTOMER
(
    CUSNUM      numeric(5)          not null,
    CUSNAME     char(30)            not null,
    CUSADDR     char(80)            not null,
    CUSACT      char(80)            ,
    CUSTEL      char(12)            ,
    CUSFAX      char(12)            ,
    constraint PK_CUSTOMER primary key (CUSNUM)
)
go
```

EnableNotNullWithDflt

Statement to enable not null with default.

Value	Result
Yes	When With Default is enabled in a column property sheet, a default value is assigned to a column when a Null value is inserted
No	The With Default option in a column property sheet is hidden

Example

In IBM DB2:

```
EnableNotNullWithDflt = YES
```

Bind

Statement for binding a rule to a column.

Example

In Sybase Adaptive Server Enterprise 12:

```
sp_bindrule %RULE%, %TABLE%.%COLUMN%
```

AddColnCheck

Statement for customizing the script for modifying column constraints within an alter table statement.

Example

In Oracle 9i:

```
alter table [%QUALIFIER%]%TABLE%
    modify (%COLUMN% %DATATYPE% constraint %CONSTNAME%
    check (%CONSTRAINT%));
```

ConstName

Constraint name template for a column check parameter.

Used in the Columns property sheet to fill Constraint column codes.

Example

```
CKC_%.U17: COLUMN%_%.U8: TABLE%
```

ColumnComment

Statement for adding a comment to a column.

Used in generation and reverse engineering.

Example

```
comment on column [%QUALIFIER%]%TABLE%.%COLUMN% is  
%.q: COMMENT%
```

DefineColnCheck

Statement for customizing the script of column constraints (checks) within a **create table** statement. This statement is called if the create, add, or alter statements contain %CONSTDEFN%.

Example

```
[constraint %CONSTNAME%] check (%CONSTRAINT%)
```

DropColnChck

Statement for dropping a column check in an **alter table** statement. This statement is used in the database modification script when the check parameters have been removed on a column.

If **DropColnChck** is empty, the Modify database process does the following:

- ◆ Drops the foreign key constraints
- ◆ Creates a temporary table with the same structure and columns as the table before the check parameter deletion
- ◆ Inserts the rows from the old table to the new temporary table
- ◆ Drops the actual table
- ◆ Creates a new table without any check parameters on the modified column
- ◆ Inserts the rows from the temporary table to the new table
- ◆ Creates the indexes and the constraints on the new table

Example

In Sybase Adaptive Server Anywhere 6:

```
DropColnChk = alter table [%QUALIFIER%]%TABLE%  
                modify %COLUMN% check null
```

The %COLUMN% variable is the name of the column defined in the column list window of a table.

DropColnComp

Statement for dropping a column computed expression in an alter table statement.

Example

In Sybase Adaptive Server Anywhere 6:

```
alter table [%QUALIFIER%]%TABLE%  
    alter %COLUMN% drop compute
```

ModifyColumn

Statement for modifying a column. This parameter is used in the database modification script when the column definition has been modified. The **ModifyColumn** statement defined in this parameter is a specific SQL statement different from the **alter table** statement. It is not supported by all DBMS.

Example

In Sybase Adaptive Server Anywhere 6:

```
alter table [%QUALIFIER%]%TABLE%  
    modify %COLUMN% %DATATYPE% %NOTNULL%
```

ModifyColnDflt

Statement for modifying a column default value in an **alter table** statement. This parameter is used in the database modification script when the default value of a column has been modified in the table.

If **ModifyColnDflt** is empty, the modify database process does the following:

- ◆ Drops the foreign key constraints
- ◆ Creates a temporary table with the same structure and columns as the table before the default value modification
- ◆ Inserts the rows from the old table to the new temporary table
- ◆ Drops the actual table
- ◆ Creates a new table with the new default value for the modified column

- ◆ Inserts the rows from the temporary table to the new table
- ◆ Creates the indexes and the constraints on the new table

Example

In Sybase Adaptive Server Anywhere 6:

```
ModifyColnDflt = alter table [%QUALIFIER%]%TABLE%
modify %COLUMN% default %DEFAULT%
```

The %COLUMN% variable is the name of the column defined in the table property sheet. The %DEFAULT% variable is the new default value of the modified column.

AddColnChck

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.

AltEnableAddColnChk must be set to YES to allow use of this statement.

If **AddColnChck** is empty, the modify database process does the following:

- ◆ Drops the foreign key constraints
- ◆ Creates a temporary table with the same structure and columns as the table before the check parameter modification
- ◆ Inserts the rows from the old table to the new temporary table
- ◆ Drops the actual table
- ◆ Creates a new table with the check parameters for the column
- ◆ Inserts the rows from the temporary table to the new table
- ◆ Creates the indexes and the constraints on the new table

Example

In Sybase Adaptive Server Anywhere 6:

```
AddColnChck = alter table [%QUALIFIER%]%TABLE%
modify %COLUMN% [default %DEFAULT%]
[check (%CONSTRAINT%)]
```

The %COLUMN% variable is the name of the column defined in the table property sheet. The %CONSTRAINT% variable is the check constraint built from the new check parameters.

See also **AltEnableAddColnChk**.

ModifyColInNull

Statement for modifying the null/not null status of a column in an **alter table** statement.

Example

In Oracle 7:

```
alter table [%QUALIFIER%]%TABLE%
  modify %COLUMN% %NOTNULL%
```

ModifyColInComp

Statement for modifying a computed expression for a column in an alter table.

Example

In Sybase Adaptive Server Anywhere 6:

```
alter table [%QUALIFIER%]%TABLE%
  alter %COLUMN% set compute (%COMPUTE%)
```

MaxConstLen

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

Maximum length for constraint names:

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.

Example

```
MaxConstLen = 128
```

Rename

Statement for renaming a column within an **alter table** statement.

Example

```
alter table [%QUALIFIER%]%TABLE%
  rename %OLDCOLUMN% to %NEWCOLUMN%
```

SqlChckQuery

SQL Query to reverse engineer column check parameters.

The result must conform to proper SQL syntax.

Example

In Sybase Adaptive Server Anywhere 6:


```

{OWNER, TABLE, COLUMN, CONSTRAINT}
SELECT SU.USER_NAME, ST.TABLE_NAME, SC.COLUMN_NAME,
SC."CHECK"
FROM SYSUSERPERMS SU, SYSTABLE ST, SYSCOLUMN SC
WHERE [SU.USER_NAME='%OWNER%' AND]
[ST.TABLE_NAME='%TABLE%' AND]
ST.CREATOR=SU.USER_ID AND SC.TABLE_ID=ST.TABLE_ID AND
SC."CHECK" IS NOT NULL
ORDER BY SU.USER_NAME, ST.TABLE_ID

```

SqlPermQuery

SQL Query to reverse engineer object permissions.

The result must conform to proper SQL syntax.

Example

In Sybase Adaptive Server Anywhere 8:

```

{ GRANTEE, GRANTOR, PERMISSION}

select
u1.user_name grantee, u.user_name grantor,
case
when (p.privilege_type = 1 and p.is_grantable = 'Y' )
then 'SELECT+'
when (p.privilege_type = 1) then 'SELECT'
when (p.privilege_type = 8 and p.is_grantable = 'Y' )
then 'UPDATE+'
when (p.privilege_type = 8) then 'UPDATE'
when (p.privilege_type = 16 and p.is_grantable = 'Y' )
then 'REFERENCES+'
when (p.privilege_type = 16) then 'REFERENCES'
end
permission
from sysuserperm u, sysuserperm u1, syscolumn s,
syscolperm p, systable t
where
(t.table_name = %.q:TABLE%) and
(t.table_id = p.table_id) and
(s.column_name = %.q:COLUMN% ) and
(s.table_id = t.table_id) and
(u1.user_id = p.grantee) and
(u.user_id = p.grantor)

```

Unbind

Statement for unbinding a rule to a column.

Example

In Sybase Adaptive Server Enterprise 12:

```
sp_unbindrule %TABLE%.%COLUMN%, %RULE%
```

Index

The Index category contains entries defining index related parameters.

Common entries for Index

You can define values for the following common entries for the Index object in the DBMS definition.

Entry	Example
Enable	Enable = Yes
Maxlen	Maxlen = 128
Create	In Sybase ASA 8: <pre>create [%UNIQUE%] index %INDEX% on [%QUALIFIER%]%TABLE% (%CIDXLIST%) [%OPTIONS%]</pre>
Drop	<pre>if exists(select 1 from sys.sysindex I, sys.systable T where I.table_id=T.table_id and I.index_name='%INDEX%' and T.table_name='%TABLE%') then drop index [%QUALIFIER%]%TABLE%.%INDEX% end if</pre>
Options	in %s : category=tablespace
DefOptions	in %s : category=tablespace
SqlListQuery	—
SqlAttrQuery	—

Entry	Example
SqlOptsQuery	<p>In Sybase ASA 8:</p> <pre>{OWNER, TABLE, INDEX, OPTIONS} select u.user_name, t.table_name, i.index_name, 'in ' + f.dbspace_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) join sys.sysfile f on (f.file_id=i.file_id) where i."unique" in ('Y', 'N') [and t.table_name=%q:TABLE%] [and u.user_name=%q:OWNER%]</pre>
SqlFragQuery	<p>In Oracle 9I:</p> <pre>{VAR1NAME, VAR1VALUE} select 'Highval' ip.partition_position, ip.high_value from all_ind_partitions ip where 1=1 [and ip.index_owner=%q:OWNER%] [and ip.index_name=%q:INDEX%]</pre>
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

EnableAscDesc

ASC, DESC keywords allowed on index definition. You can indicate ascending or descending order by selecting the Ascending or Descending statement from the Sort dropdown list box in the Columns page of an Index property sheet. Ascending is selected by default.

Value	Target database	Result
Yes	Supports index columns sorted in ascending or descending order	The variable %ASC% is calculated. The keyword ASC or DESC is generated when creating or modifying the database

Value	Target database	Result
No	Does not support index column sort	The Sort dropdown listbox does not appear in the Columns page of an Index property sheet. The variable %ASC% is not calculated. The keyword ASC or DESC is not generated when creating or modifying the database

Example

In Sybase Adaptive Server Anywhere 6:

```
EnableAscDesc = YES
```

In the following example, a primary key index is created on the TASK table, the column project number (PRONUM) is given an ascending order and the column task name (TSKNAME) is given a descending order.

```
create index IX_TASK on TASK (PRONUM asc, TSKNAME desc);
```

EnableCluster

Allows to define an index as cluster index.

Value	Result
Yes	The Cluster check box appears in the index property sheet
No	Index does not cluster index

EnableFunction

Allows to use function-based indexes.

Value	Result
Yes	You can define expressions for indexes
No	Index do not expressions

EnableOwner

Allows to define index owners. The index owner can be identical or different from the table owner.

Value	Result
Yes	The Owner dropdown listbox appears in the index property sheet and the user can select an owner for the current index
No	Index does not support owner

Example

EnableOwner = YES

If you enable owner on indexes, you have to make sure the Create statement reflects this change in order to take into account the table and index owner. For example, in Oracle 9i, the Create statement of an index is the following:

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

Where %QUALIFIER% refers to the current object (index) and %TABLQUALIFIER% refers to the parent table of the index.

MandIndexType

Determines if the index type is mandatory for an index.

Value	Result
Yes	The type is mandatory
No	The type is not mandatory

MaxColIndex

Maximum number of columns allowed in an index in the DBMS. The Check Model verifies and indicates an error if there are more columns included in the index definition than the allowed number.

Example

MaxColIndex = 99

Cluster

Cluster keyword. If this parameter is empty, the default value of the %CLUSTER% variable is CLUSTER.

Example

In Sybase AS Enterprise 11.x:

```
Cluster = clustered
```

The value **clustered** is assigned to the %CLUSTER% variable.

AddColIndex

Statement for adding a column in the **Create Index** statement. This parameter defines each column in the column list of the **Create Index** statement.

Example

In Sybase Adaptive Server Anywhere 6:

```
AddColIndex = %COLUMN% [%ASC%]
```

%COLUMN% is the code of the column defined in the column list of the table. %ASC% is ASC (ascending order) or DESC (descending order) depending on the Sort radio button state for the index column.

DefIndexType

Defines the default type of an index.

Example

In DB2:

```
Type2
```

DefineIndexColumn

Defines the column of an index.

Footer

Index footer. Anything added to this entry is added after the **create index** statement.

Header

Index header. Anything added to this entry is added before the **create index** statement.

IndexType

List of types available for an index.

Example

In Sybase Adaptive Server Anywhere 6:

```
bitmap
```

SqlSysIndexQuery

SQL query used to list system indexes created by the database. These indexes are excluded during reverse engineering.

Example

In Sybase Adaptive Server Anywhere 6:

```
{OWNER, TABLE, INDEX}
select u.user_name, t.table_name, i.index_name
from sys.sysindex i, sys.systable t, SYSUSERPERMS u
where "unique" = 'U' and t.table_id = i.table_id and
u.user_id = t.creator [and u.user_name='%OWNER%'] [and
t.table_name='%TABLE%']
union
select distinct u.user_name, t.table_name, t.table_name
|| ' (primary key)'
from sys.systable t, SYSUSERPERMS u
where u.user_id = t.creator and t.primary_root <> 0 [and
u.user_name='%OWNER%'] [and t.table_name='%TABLE%']
union
select distinct u.user_name, t.table_name, f.role ||
' (foreign key)'
from sys.systable t, SYSUSERPERMS u, sys.sysforeignkey f
where u.user_id = t.creator and t.table_id =
f.foreign_key_id [and u.user_name='%OWNER%'] [and
t.table_name='%TABLE%']
union
select distinct u.user_name, t1.table_name, f.role ||
' (foreign key)'
from sys.systable t1, sys.systable t2, SYSUSERPERMS u,
sys.sysforeignkey f
where u.user_id = t1.creator and t2.table_id =
f.primary_table_id and t1.table_id=f.foreign_table_id
[and u.user_name='%OWNER%'] [and
t1.table_name='%TABLE%']
```

IndexComment

Statement for adding a comment to an index.

Example

```
comment on index [%QUALIFIER%]%TABLE%.%INDEX% is
%.q:COMMENT%
```

UniqName

Unique index name scope assignment. This is used in check model.

Value	Result
Yes	A unique index name is used for the entire database
No	An index name may be used several times if associated with different objects

Example

UniqName = YES

CreateBeforeKey

Allows to invert generation order of indexes and keys.

Value	Result
Yes	Indexes are generated before keys
No	Indexes are generated after keys

Example

CreateBeforeKey = YES

Pkey

The Pkey category contains entries defining primary key related parameters.

Common entries for Pkey

You can define values for the following common entries for the Pkey object in the DBMS definition.

Entry	Example (Sybase Adaptive Server Enterprise 11)
Enable	Enable = Yes
Create	<pre>[%USE_SP_PKEY%][execute] sp_primarykey %TABLE%, %PKEYCOLUMNS% :alter table [%QUALIFIER%]%TABLE% add [constraint %CONSTNAME%] primary key [%CLUSTER%](%PKEYCOLUMNS%) [%OPTIONS%]</pre>
Add	<pre>[constraint %CONSTNAME%] primary key [%CLUSTER%](%PKEYCOLUMNS%) [%OPTIONS%]</pre>

Entry	Example (Sybase Adaptive Server Enterprise 11)
Drop	<code>[%USE_SP_PKEY%?execute sp_dropkey primary, %TABLE% :alter table [%QUALIFIER%]%TABLE% drop constraint %CONSTNAME%]</code>
Options	<code><clustered> %s : list=clustered nonclustered with : composite=yes, separator=yes, parenthesis=no { fillfactor=%d : default=0 max_rows_per_page=%d : default=0 } on %s : category=storage</code>
DefOptions	—

For a description of each of the common object entries, see the section Common object entries.

AllowNullableColn

Statement to allow the inclusion of nullable columns.

Value	Result
Yes	Nullable column allowed
No	Nullable column not allowed

PkAutoIndex

Primary key is auto-indexed. This option primarily determines whether to generate a **Create Index** statement for every Primary key statement.

Value	Target database	Result
Yes	Automatically generates a primary key index when generating the primary key statement	Primary key check boxes and create index cannot be selected simultaneously in the database generation or modification parameters. If you select the primary key check box under create index, the primary key check box of the create table will automatically be cleared, and vice versa

Value	Target database	Result
No	Does not generate the primary key indexes automatically when generating the primary keys	Primary key indexes have to be generated explicitly. Primary key check boxes and create index can be selected simultaneously

Example

In Sybase Adaptive Server Anywhere 6:

```
PkAutoIndex = YES
```

UseSpPrimKey

Use the **Sp_primarykey** statement to generate a primary key.

For a database that supports the procedure to implement key definition, you can test the value of the corresponding variable %USE_SP_PKEY% and choose between the creation key in the table or launching a procedure.

Value	Result
Yes	The Sp_primarykey statement is used to generate a primary key
No	Primary keys are generated separately in an alter table statement

Example

In Microsoft SQL Server:

```
UseSpPrimKey = YES
Add entry of Pkey contains

[%USE_SP_PKEY%?[execute] sp_primarykey %TABLE%,
%PKEYCOLUMNS%
:alter table [%QUALIFIER%]%TABLE%
    add [constraint %CONSTNAME%] primary key
[%IsClustered%] (%PKEYCOLUMNS%)
[%OPTIONS%]]
```

ConstName

Allows to declare a customizable template for all the primary key constraint names in a PDM.

For each table in a model, the primary key constraint name is defined in the Key properties entry *Key Constraint Name* for the Key defined as the Primary key. This constraint name is generated in the database if the database supports primary keys and constraint names.

Some PowerDesigner variables can be used to define the primary key constraint name:

- ◆ %TABLE%: table code
- ◆ %KEY%: key code
- ◆ %COLUMN%: column code
- ◆ %COLNNO%: column number

There is an automatic renaming of duplicate constraint names according to the length of the constraint.

%OID%

The %OID% variable used in some Def. files in Version 6 is no longer available. Object identifiers are coded on more than 20 characters, making them unusable for constraint names (often limited to 8 or 18 characters)

Example

```
PKConstraintName = PK_%.U27:TABLE%
```

PKeyComment

Statement for adding a primary key comment.

Example

```
comment on primary key [%QUALIFIER%]%TABLE%.%PKEY% is
%.q:COMMENT%
```

Key

The Key category contains entries defining Key related parameters.

Common entries for Key

You can define values for the following common entries for the Key object in the DBMS definition.

Entry	Example (Sybase Adaptive Server Anywhere 6)
Enable	Enable = Yes
Create	alter table [%QUALIFIER%]%TABLE% add unique (%COLUMNS%)
Add	unique (%COLUMNS%)
Drop	alter table [%QUALIFIER%]%TABLE% delete unique (%COLNLIST%)

Entry	Example (Sybase Adaptive Server Anywhere 6)
DefOptions	—
Options	<p>Oracle 7:</p> <pre> using index : composite=yes, separator=no, parenthesis=no { pctfree %d initrans %d : default=1 maxtrans %d tablespace %s : category=tablespace storage : category=storage, composite=yes, separator=no, parenthesis=yes { initial %s : default=10K next %s : default=10K minextents %d : default=1 maxextents %s maxextents unlimited pctincrease %d : default=50 freelists %d : default=1 freelist groups %d : default=1 optimal %d optimal NULL } nosort <recoverable> %s : list=recoverable unrecoverable } disable </pre>
SqlAttrQuery	—
SqlOptsQuery	—

Entry	Example (Sybase Adaptive Server Anywhere 6)
SqlListQuery	<pre> {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 t.table_name='%TABLE%' and c.pkey='Y' and c.table_id=t.table_id union select t.table_name, 0, i.index_name, c.column_name + ' ', x.sequence from sysindex i, systable t, syscolumn c, sysixcol x where t.table_name='%TABLE%' and i."unique"='U' and i.table_id=t.table_id and x.table_id=t.table_id and c.table_id=t.table_id and x.index_id=i.index_id and x.column_id=c.column_id order by 1, 3, 5 </pre>
SqlFragQuery	-
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

AllowNullableColn

Statement to allow the inclusion of nullable columns.

Value	Result
Yes	Nullable column allowed
No	Nullable column not allowed

UniqConstAutoIndex

Alternate key unique Constraint is auto-indexed. This option primarily determines if a **Create Index** statement is required for every key statement.

Value	Target database	Result
Yes	Automatically generates an alternate key index when generating the alternate key statement	Alternate key check boxes and create alternate index cannot be selected simultaneously in the database generation or modification parameters. If you select the alternate key check box under create index, the alternate key check box of the create table will automatically be cleared, and vice versa
No	Does not generate the alternate key indexes automatically when generating the alternate keys	Alternate key indexes have to be generated explicitly. Alternate key check boxes and create alternate index can be selected simultaneously

Example

In Sybase Adaptive Server Anywhere 6:

```
UniqConstAutoIndex = YES
```

UniqInTable

Alternate Key created in the table. This entry determines where the alternate key is defined in the SQL script.

Value	Result
Yes	Alternate keys are included in the create table statement. The Add entry for Key is concatenated inside the %TABLEDEFN% variable

Value	Result
No	Alternate keys generated separately in an alter table statement, the content of the Create entry for Alternate key is used during generation

Example

`UniqInTable = Yes`

MaxConstLen

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

Maximum length for constraint names:

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.

Example

`MaxConstLen = 128`

ConstName

Allows the declaration of a customizable template for all the alternate key constraint names in a PDM. It is used in the Key property sheet for all the keys that are alternate keys and not primary keys.

For each table in a model, an alternate key constraint name is defined in the Key properties entry *Key Constraint Name* for a key defined as alternate key. This constraint name is generated in the database if the database supports alternate keys and constraint names.

Some PowerDesigner variables can be used to define alternate key constraint name:

- ◆ %TABLE%: table code
- ◆ %KEY%: key code

There is an automatic renaming of duplicate constraint names according to the length of the constraint.

Example

In Sybase Adaptive Server Anywhere 6:

`AK_% .U18:AKEY%_%.U8:TABLE%`

SqlAkeyIndex

Reverse engineering query for obtaining the alternate key indexes of a table by ODBC.

Example

In Sybase Adaptive Server Anywhere 6:

```
select distinct I.INDEX_NAME
from SYSINDEX I, SYSUSERPERMS U, SYSTABLE T
where I."UNIQUE" = 'U' and I.TABLE_ID = T.TABLE_ID and
      T.TABLE_NAME = '%TABLE%' and T.CREATOR = U.USER_ID
and U.USER_NAME = '%USER%'
```

AKeyComment

Statement for adding an alternate key comment.

Example

```
comment on alternate key [%QUALIFIER%]%TABLE%.%AKEY% is
%.q:COMMENT%
```

Reference

The Reference category contains entries defining reference related parameters.

Common entries for Reference

You can define values for the following common entries for the Reference object in the DBMS definition.

Entry	Example (Sybase Adaptive Server Anywhere 6)
Enable	Enable = Yes
Create	<p>Note that the check on commit (CHCKONCMMT) at the end of the example is useful during reverse engineering to parse the text.</p> <pre>Alter table [%QUALIFIER%]%TABLE% Add foreign key %CONSTNAME% (%FKEYCOLUMNS%) references [%PQUALIFIER%]%PARENT% (%CKEYCOLUMNS%) [on update %UPDCONST%] [on delete %DELCONST%] [%CHCKONCMMT% ? check on commit]</pre>

Entry	Example (Sybase Adaptive Server Anywhere 6)
Add	foreign key %CONSTNAME% (%FKEYCOLUMNS%) references [%PQUALIFIER%]%PARENT% (%CKEYCOLUMNS%) [on update %UPDCONST%] [on delete %DELCONST%] [%CHCKONCMMT% ? check on commit]
Drop	if exists(select 1 from sys.sysforeignkey where role='%CONSTNAME%') then alter table [%QUALIFIER%]%TABLE% delete foreign key %CONSTNAME% end if
SqlAttrQuery	—
SqlListQuery	—
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

EnableChangeJoinOrder

When a reference is linked to a key as shown in the Joins page of reference properties, this entry determines if the auto arrange join order check box and features are available.

Value	Result
Yes	The join order can be established automatically or manually by using the Auto arrange join order check box. Selecting this check box sorts the list according to the key column order (the move buttons are not available). Clearing this check box allows manual sort of the join order with the move buttons (the move buttons are available)
No	The auto arrange join order is grayed and the option unavailable

Example

EnableChangeJoinOrder = YES

EnablefKeyName

Foreign key role allowed during database generation.

Value	Result
Yes	The code of the reference is used as role for the foreign key
No	The foreign key role is not allowed

FKAutoIndex

Foreign key is auto-indexed. This option determines whether or not to generate a **Create Index** statement for every Foreign key statement.

Value	DBMS	Result
Yes	Automatically generates a foreign key index when generating the primary, alternate, or foreign key statements	The Foreign key check box in the Index Filter group of check boxes, and the Create Foreign Keys check box, cannot be selected simultaneously in the Keys and Indexes page of the Database Generation box
No	Does not generate the foreign key indexes automatically when generating foreign keys	Foreign key indexes have to be generated explicitly. Foreign key check boxes of both create table and create index can be selected simultaneously

Example

FKAutoIndex = YES

MaxConstLen

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

Maximum length for constraint names:

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.

Example

MaxConstLen = 128

ConstName

Allows to declare a customizable template for all the foreign key constraint names in a PDM. It is used in the Reference property sheet, in the Constraint name entry on the Integrity tab.

For each table in a model, a foreign key constraint name is defined in the Reference properties entry *Key Constraint Name* for a key defined as foreign key. This constraint name is generated in the database if the database supports foreign keys and constraint names.

The following PowerDesigner variables can be used to define foreign key constraint name:

%TABLE%: table code (parent and child table, Version 6 compliant)

There is an automatic renaming of duplicate constraint names according to the length of the constraint.

Example

```
FK_%.U8:CHILD%.%.U9:REFR%.%.U8:PARENT%
```

CheckOnCommit

Referential integrity test is only performed after the COMMIT.

Contains the keyword used to specify a reference with the CheckOnCommit option.

Example

```
CHECK ON COMMIT
```

DclUpdIntegrity

Declarative referential integrity constraints allowed for update.

Must contain a list of words from the following selection:

RESTRICT

CASCADE

SET NULL

SET DEFAULT

Depending on the selected words, the radio button in the Integrity tab of the Reference property sheet is available or not.

Example

In Sybase Adaptive Server Anywhere 6:

```
RESTRICT  
CASCADE  
SET NULL  
SET DEFAULT
```

DclDelIntegrity

Declarative referential integrity constraints allowed for delete.

Must contain a list of words from the following selection.

RESTRICT

CASCADE

SET NULL

SET DEFAULT

Depending on the selected list of words, the radio button in the Integrity tab of the Reference property sheet is available or not.

Example

In Sybase Adaptive Server Anywhere 6:

```
RESTRICT
CASCADE
SET NULL
SET DEFAULT
```

DefineJoin

Defines a join for a reference, this corresponds to the %JOINS% variable. This is another way of defining the contents of the **create reference** statement.

Usually the **create** script for a reference uses the variable %CKEYCOLUMNS% and %PKEYCOLUMNS% which contains the list of Child and parent columns separated by comma.

If you use %JOINS%, you can refer to each paired parent and child columns separately.

When using %JOINS%, a loop is executed on Join for each paired parent and child columns, allowing to have a syntax mix of PK and FK.

Example

In Microsoft Access:

```
DefineJoin = P=%PK% F=FK%
Create = CreateJoin C=%CONSTANME% T=%TABLE% P=%PARENT%
(
  %JOINS%
)
```

SqlListChildrenQuery

SQL query used to list the joins in a reference.

Example

In Oracle 8:

```
SELECT COL2.COLUMN_NAME, COL1.COLUMN_NAME
FROM SYS.ALL_CONS_COLUMNS COL1, SYS.ALL_CONS_COLUMNS
COL2
WHERE
  COL1.POSITION = COL2.POSITION
  AND COL1.OWNER=&#39;%SCHEMA%&#39; AND
  COL1.TABLE_NAME=&#39;%TABLE%&#39;
  AND COL2.OWNER=&#39;%POWNER%&#39; AND
  COL2.TABLE_NAME=&#39;%PARENT%&#39;
  AND COL1.CONSTRAINT_NAME=&#39;%FKCONSTRAINT%&#39; AND
  COL2.CONSTRAINT_NAME=&#39;%PKCONSTRAINT%&#39;
ORDER BY COL1.POSITION
```

UseSpFornKey

Use the **Sp_foreignkey** statement to generate a foreign key.

Value	Result
Yes	The Sp_foreignkey statement is used to create a reference
No	Foreign keys are generated separately in an alter table statement using the Create order of reference

Example

In Sybase Adaptive Server Enterprise 11:

```
UseSpFornKey = Yes
```

See also **UseSpPrimKey**.

FKeyComment

Command for adding a foreign key comment.

Example

```
comment on foreign key [%QUALIFIER%]%TABLE%.%FK% is
%.q:COMMENT%
```

View

The View category contains entries defining view related parameters.

Common entries for View

You can define values for the following common entries for the view object in the DBMS definition.

Entry	Example (Oracle 8)
Enable	Enable = Yes
Create	create [or replace][%R% [no] force]view [%QUALIFIER%]%VIEW% [(%VIEWCOLN%)]as %SQL% [%VIEWCHECK%] [%R%with read only]
Drop	drop view [%QUALIFIER%]%VIEW%
SqlListQuery	—
SqlAttrQuery	{OWNER, VIEW, SCRIPT} [%ISODBCUSER% ? SELECT '%SCHEMA%', VIEW_NAME, TEXT FROM SYS.USER_VIEWS ORDER BY VIEW_NAME : SELECT OWNER, VIEW_NAME, TEXT FROM SYS.ALL_VIEWS [WHERE OWNER='%SCHEMA%'] ORDER BY OWNER, VIEW_NAME]
SqlFragQuery	{VAR1NAME, VAR1VALUE} select 'HighVal' tp.partition_position, tp.high_value from all_tab_partitions tp where 1=1 [and tp.table_owner=%q:OWNER%] [and tp.table_name=%q:VIEW%]
SqlOptsQuery	—
Options	—
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

Permission

Available permissions for views. The first column displays the SQL name of permission (SELECT for example). The second column is the shortname that appears in the title of grid columns.

Example

In Sybase ASE 12.5:

```
SELECT / Sel
INSERT / Ins
DELETE / Del
UPDATE / Upd
```

Footer

View footer. Anything added to this entry is added after the **create view** statement.

Header

View header. Anything added to this entry is added before the **create index** statement.

SqlPermQuery

SQL Query to reverse engineer permissions granted on views.

Example

In Sybase Adaptive Server Enterprise 12.5:

```
select ul.name grantee,
case
  when (s.action = 193) then 'SELECT'
  when (s.action = 195) then 'INSERT'
  when (s.action = 196) then 'DELETE'
  when (s.action = 197) then 'UPDATE'
end +
case
  when (s.protecttype = 0) then '+'
  when (s.protecttype = 1) then ''
  when (s.protecttype = 2) then '-'
end
|| ', '
from sysprotects s, sysusers u, sysusers ul, sysobjects
o
where
  o.name = %.q:VIEW% and
  o.uid = u.uid and
```

```
s.id = o.id and  
ul.uid = s.uid
```

ViewComment

Statement for adding a view comment. The view comment is a SQL statement. It is not supported by all DBMS. If this parameter is empty, the Comment check box in the Views groupbox in the Tables & Views page of the Generate Database box is grayed and unavailable.

Example

In Oracle 8:

```
comment on table [%QUALIFIER%.]%VIEW% is '%COMMENT%'
```

The %VIEW% variable is the name of the view defined in the List of Views, or in the view property sheet. The %COMMENT% variable is the comment defined in the Comment textbox in the View property sheet.

ViewCheck

Option for checking a view. This parameter determines if the With Check Option check box in the view property sheet is available or grayed.

If the check box is selected and the **ViewCheck** parameter is not empty, the value of **ViewCheck** is generated at the end of the view select statement and before the terminator.

Example

In Sybase SQL Anywhere 5.5:

```
ViewCheck = with check option
```

The generated script for this example is:

```
create view TEST as  
select CUSTOMER.CUSNUM, CUSTOMER.CUSNAME,  
CUSTOMER.CUSTEL  
from CUSTOMER  
with check option;
```

ViewStyle

Option for defining a view usage. The value defined for this entry will be displayed in the Usage dropdown listbox of the view property sheet.

Example

In Oracle 9i:

```
ViewStyle = materialized view
```


Tablespace

The Tablespace category contains entries defining tablespace related parameters.

Common entries for Tablespace

You can define values for the following common entries for the Tablespace object in the DBMS definition.

Entry	Example (Oracle 8)
Enable	Enable = Yes
Create	create tablespace %TABLESPACE% [%OPTIONS%]
Drop	drop tablespace %TABLESPACE% [including contents [cascade constraints]]

Entry	Example (Oracle 8)
Options	<pre> datafile : composite=yes, chldmand=yes { <datafile_clause> : composite=yes, separator=yes, multiple=yes { <file_spec> : composite=yes { <filename> %s : squoted=yes size %d reuse } <autoextend_clause> : composite=yes { autoextend off autoextend on : composite=yes { next %s maxsize %s : default=UNLIMITED } } } minimum extent %s <log> %s : list=logging nologging default storage : category=storage, composite=yes, parenthesis=yes { initial %d : default=10K next %d : default=10K minextents %d : default=1 maxextents %d maxextents unlimited pctincrease %d : default=50 freelists %d : default=1 optimal %d optimal NULL buffer_pool %s : list=keep recycle default } <online> %s : default=online, list=online offline <permanent> %s : default=permanent, list=permanent temporary </pre>
DefOptions	—
SqlListQuery	<pre> {TABLESPACE} SELECT TABLESPACE_NAME FROM USER_TABLESPACES ORDER BY 1 </pre>

Entry	Example (Oracle 8)
SqlAttrQuery	{TABLESPACE, OPTIONS} SELECT TABLESPACE_NAME, 'DEFAULT STORAGE (INITIAL ' INITIAL_EXTENT ' NEXT ' NEXT_EXTENT ' MIN_EXTENTS ' MIN_EXTENTS ' MAX_EXTENTS ' MAX_EXTENTS ' PCTINCREASE ' PCT_INCREASE ') ' STATUS OPTS FROM USER_TABLESPACES
SqlOptsQuery	—
SqlFragQuery	—
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

TablespaceComment

Statement for adding a tablespace comment.

Example comment on tablespace [%QUALIFIER%]%TABLESPACE% is
 %.q:COMMENT%

Storage

The Storage category contains entries defining storage related parameters.

Common entries for Storage

You can define values for the following common entries for the Storage object in the DBMS definition.

Entry	Example (Oracle 8)
Enable	Enable = Yes
Create	-
Drop	-

Entry	Example (Oracle 8)
Options	<pre> initial %d : default=10K next %d : default=10K minextents %d : default=1 maxextents %d pctincrease %d : default=50 freelists %d : default=1 freelist groups %d : default=1 optimal %d buffer_pool %s : list=keep recycle default </pre>
DefOptions	—
SqlList Query	—
SqlAttrQuery	<pre> {STORAGE, OPTIONS} SELECT SEGMENT_NAME, 'INITIAL ' INITIAL_EXTENT ' NEXT ' NEXT_EXTENT ' MIN_EXTENTS ' MIN_EXTENTS ' MAX_EXTENTS ' MAX_EXTENTS PCTINCREASE ' PCT_INCREASE ' FREELISTS ' FREELISTS FREELIST_GROUPS ' FREELIST_GROUPS OPTS FROM USER_SEGMENTS </pre>
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

☞ For a description of each of the common object entries, see the section Common object entries.

StorageComment

Statement for adding a storage comment.

Example

```
comment on storage [%QUALIFIER%]%STORAGE% is
%.q:COMMENT%
```

Database

The Database category contains entries defining database related parameters.

Common entries for Database

You can define values for the following common entries for the Database object in the DBMS definition.

Entry	Example (Adaptive Server Anywhere 6)
Enable	Enable = No
Create	create database '%DATABASE%.db' [%OPTIONS%]
Drop	drop database %DATABASE%
Options	transaction log %s : list= on off <logonfile> %s mirror %s case %s : default=respect, list=respect ignore page size %d : default=1024, list=1024 2048 4096 collation %s encrypted %s : list=on off blank padding %s : list=on off ASE compatible java %s : list=on off jconnect %s : list=on off
DefOptions	—
SqlListQuery	—
SqlAttrQuery	—
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

BeforeCreateDatabase

Determines if the **Create Tablespace** and **Create Storage** statements are generated before or after the database is created.

Default value: YES.

Value	Result
Yes	Create Tablespace statements and the Create Storage statements are generated before the Create Database statement

Value	Result
No	Create Tablespace statements and the Create Storage statements are generated after the Create Database statement

Example

In IBM DB2 MVS version 4:
`BeforeCreateDatabase = NO`

CloseDatabase

Statement for closing a database. If this parameter is empty, the Close Database check box in the Database page of the Generate Database box is grayed and unavailable.

Example

In Sybase AS Enterprise 11.x:
`CloseDatabase =`

OpenDatabase

Statement for opening a database. If this parameter is empty, the Open Database check box in the Database page of the Generate Database box is grayed and unavailable.

Example

In Sybase AS Enterprise 11.x:
`OpenDatabase = use %DATABASE%`
 The %DATABASE% variable is the name of the model defined in the Model property sheet.

Domain

The Domain category contains entries defining domain related parameters.

Common entries for Domain

You can define values for the following common entries for the Domain object in the DBMS definition.

Entry	Example (Sybase Adaptive Server Anywhere 6)
Enable	Enable = Yes
Maxlen	MaxLen = 30

Entry	Example (Sybase Adaptive Server Anywhere 6)
Create	create datatype %DOMAIN% %DATATYPE% [%NOTNULL%] [default %DEFAULT%] [check (%CONSTRAINT%)]
Drop	if exists(select 1 from systypes where name=%DOMAIN%)then drop datatype %DOMAIN% end if
SqlListQuery	{@OBJTCODE id, DTTPCODE, LENGTH, PREC} Select d.type_name, t.domain_name, d.width, t."precision" from sysusertype d, sysdomain t where d.domain_id=t.domain_id and d.domain_id <> 25 and d.domain_id <> 26 and creator <> 0
SqlAttrQuery	{@OBJTCODE id, DTTPCODE, LENGTH, PREC, DEFAULT, CONSTRAINT} Select d.type_name, t.domain_name, d.width, d.scale, d."default", d."check" from sysusertype d, sysdomain t where d.domain_id=t.domain_id and d.domain_id <> 25 and d.domain_id <> 26 and creator <> 0
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

EnableBindRule

Allows or disallows the binding of a business rule to a domain for check parameters.

Value	Result
Yes	The Create and Bind entry of Rule are generated
No	The check inside the domain Add order is generated

Example

In Microsoft SQL Server and Sybase 11:

```
EnableBindRule = Yes
```

EnableDefault

Allows the use of predefined default values.

Value	Result
Yes	The default value is generated for domain when the default value is defined. The default value can be defined in the check parameters. The %DEFAULT% variable will contain the default value
No	The default value can not be generated

EnableCheck

Determines if the generation of check parameters is authorized or not.

This entry is tested during column generation: if User-defined Type is selected for columns in the Generation dialog box, and if EnableCheck is set to Yes for domains, then the check parameters are not generated for column, 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.

Value	Result
Yes	Check parameters generated
No	All variables linked to Check parameters will not be evaluated during generation and reverse

EnableOwner

Allows to define domain owners.

Value	Result
Yes	The Owner dropdown listbox appears in the domain property sheet and the user can select an owner for the current domain
No	Domain does not support owner

Example

```
EnableOwner = YES
```

UserTypeName

You can define data types with user-defined names. This improves the readability of the contents of objects in your specific applications.

Example

```
UserTypeName = T_ %DOMAIN%
```


Bind

Binds a business rule to a domain.

Example

In Sybase Adaptive Server Enterprise 11:

```
[execute] sp_bindrule %RULE%, %DOMAIN%
```

BindDefault

Binds a default value to a domain.

Example

In Microsoft SQL Server and Sybase 11:

```
sp_bindefault %DEFAULTNAME%, %DOMAIN%
```

CreateDefault

Creates a default value for a domain.

Example

In Microsoft SQL Server and Sybase 11:

```
create default %DEFAULTNAME%  
as %DEFAULT%
```

SqlListDefaultQuery

SQL Query to list domain default values.

For certain DBMS, the ODBC reverse engineering process retrieves default domain values in the system tables. This query lists these default values.

UddtComment

Statement for adding a user-defined data type comment.

Example

```
comment on user-defined data type [%QUALIFIER%]%DOMAIN%  
is %.q:COMMENT%
```

Unbind

Unbinds a business rule to a domain.

Example

In Sybase Adaptive Server Enterprise 11:

```
[execute] sp_unbindrule %DOMAIN%, %RULE%
```

Abstract Data Type

The Abstract Data Type category contains entries defining abstract data type related parameters.

Common entries for Abstract Data Type

You can define values for the following common entries for the Abstract Data Types object in the DBMS definition.

Entry	Example
Enable	Sybase Adaptive Server Anywhere 6: Enable = Yes
Create	In Oracle 8: [<i>%ISARRAY%</i> ? create type <i>%ADT%</i> as <i>%TYPE%</i> (<i>%SIZE%</i>) of <i>%DATATYPE%</i>] [<i>%ISLIST%</i> ? create type <i>%ADT%</i> as <i>%TYPE%</i> of <i>%DATATYPE%</i>] [<i>%ISOBJECT%</i> ? create type <i>%ADT%</i> as <i>%TYPE%</i> (<i>%ADTDEF%</i>)]
Drop	In Oracle 8: drop type <i>%ADT%</i>
SqlListQuery	In Sybase Adaptive Server Anywhere 6: {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 ORDER BY type_name
SqlAttrQuery	—
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

EnableAdtOnDomn

Statement for enabling Abstract Data Types on domains.

Value	Result
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

Example

```
EnableAdtOnDomn = Yes
```

EnableAdtOnColn

Statement for enabling Abstract Data Types on columns.

Value	Result
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

Example

```
EnableAdtOnColn = Yes
```

Install

Statement for installing a Java class. This entry is equivalent to a **create** statement.

In Sybase Adaptive Server Anywhere, the abstract data types are not really created but installed.

Example

In Sybase Adaptive Server Anywhere 6:

```
install JAVA UPDATE from file '%FILE%'
```

Remove

Statement for uninstalling a Java class. This entry is equivalent to a drop statement.

In Sybase Adaptive Server Anywhere, the abstract data types are not really deleted but removed.

Example

In Sybase Adaptive Server Anywhere 6:

```
remove JAVA class %ADT%
```

AllowedADT

List of abstract data types which can be used as data types for an abstract data type.

Example

In Oracle 8i:
OBJECT

ADTComment

Statement for adding an abstract data type comment.

Example

comment on abstract datatype [%QUALIFIER%]%ADT% is
%.q:COMMENT%

Abstract Data Type Attribute

The Abstract Data Types Attribute category contains entries defining abstract data type attribute related parameters.

Common entries for Abstract Data Type Attribute

You can define values for the following common entries for the Abstract Data Types Attribute object in the DBMS definition.

Entry	Example
Create	—
Add	In Oracle 8: %ADTATTR% %DATATYPE%
Drop	—
Modify	—
SqlListQuery	In Oracle 8: { ADT, ADTATTR , DTTPCODE, LENGTH, PREC } SELECT type_name, attr_name, attr_type_name, length, precision FROM all_type_attrs ORDER BY type_name
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

AllowedADT

List of abstract data types which can be used as data type for Abstract Data Type Attributes.

In PowerDesigner, when you select the type OBJECT for an abstract data type, an Attributes tab is added to the abstract data type property sheet. This allows you to specify the attributes of the object data type.

Example

In Oracle 8i:
 OBJECT
 TABLE
 VARRAY

User

The User category contains entries defining user related parameters.

Common entries for User

You can define values for the following common entries for the User object in the DBMS definition.

Entry	Example (Sybase Adaptive Server Anywhere 6)
Enable	Enable = Yes
Maxlen	Maxlen = 128
SqlListQuery	{@OBJTCODE id} Select USER_NAME from SYS.SYSUSERPERMS order by USER_NAME
SqlAttrQuery	—
Create	In Sybase ASE 12.5: sp_adduser %USERID% [%PRIVILEGE%]
Drop	In Sybase ASE 12.5: sp_dropuser %USERID%
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

☞ For a description of each of the common object entries, see the section Common object entries.

SqlPermQuery

SQL Query to reverse engineer permissions granted on users.

Example

In Sybase Adaptive Server Enterprise 12.5:
{USER ID, PRIVILEGE ...}

```
select u1.name grantee,
case
  when (s.action = 198) then 'CREATE TABLE'
  when (s.action = 203) then 'CREATE DATABASE'
  when (s.action = 207) then 'CREATE VIEW'
  when (s.action = 221) then 'CREATE TRIGGER'
  when (s.action = 222) then 'CREATE PROCEDURE'
  when (s.action = 233) then 'CREATE DEFAULT'
  when (s.action = 236) then 'CREATE RULE'
end +
case
  when (s.protecttype = 0) then '+'
  when (s.protecttype = 1) then ''
  when (s.protecttype = 2) then '-'
end
|| ','
from sysprotects s, sysusers u1
where u1.uid = s.uid
order by 1
```

Rule

The Rule category contains entries defining rule related parameters.

Common entries for Rule

You can define values for the following common entries for the Rule object in the DBMS definition.

Entry	Example (Sybase Adaptive Server Enterprise 11)
Enable	Enable = Yes
Maxlen	MaxLen = 30
Create	create rule %RULE% as %RULESEXP%

Entry	Example (Sybase Adaptive Server Enterprise 11)
Drop	if exists (select 1 from sysobjects where name='%RULE%' and type='R') drop rule %RULE%
SqlListQuery	{@OBJTNAME ID, @OBJTCODE ID, SCRIPT ...} select o.name,convert(char(20), o.id),t.text from dbo.sysobjects o, dbo.syscomments t where [%SCHEMA%? o.uid=user_id('%SCHEMA%') and] o.type='R' and t.id=o.id and t.texttype=0 order by o.id,t.number,t.colid
SqlAttrQuery	—
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

MaxDefaultLen

Maximum length that the DBMS supports for the name of the column Default name. This name is only defined in the Sybase databases family syntax.

Example MaxDefaultLen = 254

ColnDefaultName

Name of Default for Column. When a column has a specific default value defined in its check parameters, a name is created for this column default value. This entry is used with DBMS that do not support check parameters on columns such as SQL Server 4.

The corresponding variable is %DEFAULTNAME%

Example In Sybase Adaptive Server Enterprise 11:

ColnDefaultName = D_%.19: COLUMN%_%.8: TABLE%

The column Employee function **EMPFUNC** of the table EMPLOYEE has the default value, **Technical Engineer**. The column default name, **D_EMPFUNC_EMPLOYEE**, is created:

```
create default D_EMPFUNC_EMPLOYEE
as 'Technical Engineer'
go

execute sp_bindefault D_EMPFUNC_EMPLOYEE,
"EMPLOYEE.EMPFUNC"
go
```

ColnRuleName

The column check parameters are defined in separate rules in the **create table** for Sybase SQL Server 4.x and Microsoft SQL Server. When a column has a specific rule defined in its check parameters, these DBMS create a name for this column rule.

The corresponding variable is %RULE%

Example

In Sybase Adaptive Server Enterprise 11:

```
ColnRuleName = R_%.19:COLUMN%_%.8:TABLE%
```

The column Specialty (TEASPE) of the table Team has a list of values defined in its check parameters: Industry, Military, Nuclear, Bank, Marketing.

The following rule name, R_TEASPE_TEAM, is created and associated with the TEASPE column:

```
create rule R_TEASPE_TEAM
as
    @TEASPE in
    ('Industry', 'Military', 'Nuclear', 'Bank', 'Marketing')
go

execute sp_bindrule R_TEASPE_TEAM, "TEAM.TEASPE"
go
```

UddtDefaultName

Default name for a user-defined data type. When a domain has a specific default value defined in its check parameters, Sybase databases create a name as a default value for this user-defined data type.

The corresponding variable is %DEFAULTNAME%

Example

In Sybase Adaptive Server Enterprise 11:

```
UddtDefaultName = D_%.28:DOMAIN%
```


The **FunctionList** domain has a default value defined in its check parameters: **Technical Engineer**. The following SQL script will generate a default name for that default value:

```
create default D_Functionlist
as 'Technical Engineer'
go
```

UddtRuleName

Name of a rule defined for a user-defined data type. When a domain has a specific rule defined in its check parameters, the Sybase database creates a name for this user-defined data type value for a rule.

The corresponding variable is %RULE%

Example

In Sybase Adaptive Server Enterprise 11:

```
UddtRuleName = R_%.28:DOMAIN%
```

The **Domain_speciality** domain has to belong to a set of values. This domain check has been defined in a validation rule. The SQL script will generate the rule name following the template defined in the entry

UddtRuleName:

```
create rule R_Domain_speciality
as
    (@Domain_speciality in
    ('Industry', 'Military', 'Nuclear', 'Bank', 'Marketing'))
go

execute sp_bindrule R_Domain_speciality,
T_Domain_speciality
go
```

RuleComment

Statement for adding a rule comment.

Example

```
comment on rule [%QUALIFIER%]%RULE% is %.q:COMMENT%
```

Procedure

The Procedure category contains entries defining stored procedure related parameters.

Common entries for Procedure

You can define values for the following common entries for the Procedure object in the DBMS definition.

Entry	Example (Sybase Adaptive Server Anywhere 6)
Enable	Enable = Yes
Maxlen	Maxlen = 128
Create	create procedure %PROC%[(%PROCPRMS%)] %TRGDEFN%
Drop	if exists(select 1 from sys.sysprocedure where proc_name = '%PROC%') then drop procedure %PROC% end if
SqlListQuery	{OWNER, @OBJTCODE} SELECT U.USER_NAME, P.PROC_NAME FROM SYSUSERPERMS U,SYSPROCEDURE P WHERE [%SCHEMA% ? U.USER_NAME='%SCHEMA%' AND] P.CREATOR=U.USER_ID
SqlAttrQuery	{OWNER, @OBJTCODE, SCRIPT, @OBJTLABL} SELECT U.USER_NAME, P.PROC_NAME, P.PROC_DEFN, P.REMARKS FROM SYSUSERPERMS U,SYSPROCEDURE P WHERE [%SCHEMA% ? U.USER_NAME='%SCHEMA%' AND] P.CREATOR=U.USER_ID ORDER BY U.USER_NAME
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

Permission

Available permissions for procedures. The first column displays the SQL name of permission (SELECT for example). The second column is the shortname that appears in the title of grid columns.

Example

In Sybase ASE 12.5:
EXECUTE / Exe

EnableFunc

Determines if functions are allowed or not.

Functions are forms of procedure that return a value to the calling environment for use in queries and other SQL statements.

Value	Result
Yes	The function is allowed
No	The function is not allowed

Example

```
EnableFunc = Yes
```

EnableOwner

Allows to define procedure owners.

Value	Result
Yes	The Owner dropdown listbox appears in the procedure property sheet and the user can select an owner for the current procedure
No	Procedure does not support owner

Example

```
EnableOwner = YES
```

MaxFuncLen

Determines the maximum length of the name of a function.

Example

```
MaxFuncLen = 128
```

CreateFunc

Statement for creating a function.

Example

In Sybase Adaptive Server Anywhere 6:

```
create function %FUNC%[(%PROCPRMS%)]
%TRGDEFN%
```

CustomProc

Statement for creating a stored procedure.

Stored procedures are scripts using pre-defined operators; variables, functions and macros to handle data and otherwise modify it.

Example

In Sybase Adaptive Server Anywhere 6:

```
create procedure %PROC% (IN <arg> <type>)
begin
.
.
.
end
/
```

CustomFunc

Statement for creating a function.

User-defined functions are a form of procedure that returns a value to the calling environment for use in queries and other SQL statements.

Example

In Sybase Adaptive Server Anywhere 6:

```
create function %FUNC% (<arg> <type>)
RETURNS <type>
begin
.
.
.
end
/
```

DropFunc

Statement for dropping a function.

Example

In Sybase Adaptive Server Anywhere 6:

```
if exists(select 1 from sys.sysprocedure where proc_name
= '%FUNC%') then
    drop function %FUNC%
end if
```

SqlPermQuery

SQL Query to reverse engineer permissions granted on procedures.

Example

In Sybase Adaptive Server Enterprise 12.5:

```
select ul.name grantee,
case
    when (s.action = 193) then 'SELECT'
    when (s.action = 195) then 'INSERT'
    when (s.action = 196) then 'DELETE'
    when (s.action = 197) then 'UPDATE'
end +
case
    when (s.protecttype = 0) then '+'
```

```

        when (s.protecttype = 1) then ''
        when (s.protecttype = 2) then '-'
    end
    || ','
from sysprotects s, sysusers u, sysusers ul, sysobjects
o
where
    o.name = %.q:PROCEDURE% and
    o.uid = u.uid and
    s.id = o.id and
    ul.uid = s.uid

```

ProcedureComment

Statement for adding a procedure comment.

Example `comment on procedure [%QUALIFIER%]%PROC% is %.q:COMMENT%`

FunctionComment

Statement for adding a function comment.

Example `comment on procedure [%QUALIFIER%]%PROC% is %.q:COMMENT%`

Trigger

The Trigger category contains entries defining trigger related parameters.

Common entries for Trigger

You can define values for the following common entries for the Trigger object in the DBMS definition.

Entry	Example (Sybase Adaptive Server Anywhere 6)
Enable	Enable = Yes
Maxlen	MaxLen = 30
Drop	if exists(select 1 from sys.systrigger where trigger_name = '%TRIGGER%') then drop trigger %TRIGGER% end if
Create	create trigger %TRIGGER%[%TRGTIME%=before? no cascade][%TRGTIME%][%TRGEVENT%[of %COLUMNS%]] on [%QUALIFIER%]%TABLE% %TRGDEFN%

Entry	Example (Sybase Adaptive Server Anywhere 6)
SqlListQuery	<pre>{OWNER, TABLE, TRIGGER, TRGEVENT (C=Update, D=Delete, *=Insert), TRGTIME (A=After, *=Before)} SELECT U.USER_NAME, T.TABLE_NAME, R.TRIGGER_NAME, R.EVENT, R.TRIGGER_TIME FROM SYSUSERPERMS U,SYSTABLE T,SYSTRIGGER R WHERE [%SCHEMA% ? U.USER_NAME='%SCHEMA%' AND] R.TRIGGER_NAME IS NOT NULL AND T.CREATOR=U.USER_ID AND R.TABLE_ID = T.TABLE_ID ORDER BY U.USER_NAME, T.TABLE_NAME</pre>
SqlAttrQuery	<pre>{OWNER, TABLE, TRIGGER, SCRIPT} SELECT U.USER_NAME, T.TABLE_NAME, R.TRIGGER_NAME, R.TRIGGER_DEFN FROM SYSUSERPERMS U,SYSTABLE T,SYSTRIGGER R WHERE [%OWNER% ? U.USER_NAME='%OWNER%' AND] [%TABLE% ? T.TABLE_NAME='%TABLE%' AND] T.CREATOR=U.USER_ID AND R.TABLE_ID = T.TABLE_ID ORDER BY U.USER_NAME, T.TABLE_NAME</pre>
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

EnableMultiTrigger

Determines if multiple triggers by type are allowed or not.

Possible values: YES/NO.

Example EnableMultiTrigger = Yes

EnableOwner

Allows to define trigger owners. The trigger owner can be identical or different from the table owner.

Value	Result
Yes	The Owner dropdown listbox appears in the trigger property sheet and the user can select an owner for the current trigger

Value	Result
No	Trigger does not support owner

Example `EnableOwner = YES`

DefaultTriggerName

Default trigger name.

Example In Sybase Adaptive Server Anywhere 6:

`%TEMPLATE%_%.L:TABLE%`

UseErrorMsgTable

Handles errors during trigger generation using user-defined messages.

When you select User-defined in the Error Messages page of the Rebuild Triggers dialog box, the .ERROR macro will be replaced by the content of the **UseErrorMsgTable** entry during the rebuild process.

The .ERROR macro, is called by a template item in the trigger template, it has two parameters: %ERRNO% is the error number and %ERRMSG% is the standard error message.

User-Defined messages are stored in a message table which you have to create in your database:

Table variable	Description
%MSGTAB%	Table name
%MSGNO%	Message number column that stores the error message number that is referenced in the trigger script
%MSGTXT%	Message text column that stores the text of the message

With the User-defined error messages option selected, if an error number in the trigger script corresponds to an error number in the message table, the error message default parameter of the .ERROR macro is replaced by the user-defined message from the message table.

Example In Sybase Adaptive Server Enterprise 12:

```
begin
  select @errno = %ERRNO%,
         @errmsg = %MSGTXT%
  from   %MSGTAB%
  where  %MSGNO% = %ERRNO%
  goto error
```

end

%MSGTAB%, **%MSGNO%** and **%MSGTXT%** will be replaced by the values defined in the message number column, message text column, and message table name of the message table.

UseErrorMsgText

Handles errors during trigger generation using standard error messages.

When you select Standard in the Error Messages page of the Rebuild Triggers dialog box, the .ERROR macro will be replaced by the content of the **UseErrorMsgText** entry during the rebuild process.

The .ERROR macro, is called by a template item in the trigger template, it has two parameters: %ERRNO% is the error number and %ERRMSG% is the text message.

With the Standard error messages option selected, if an error number in the trigger script corresponds to the %ERRNO% parameter of the .ERROR macro, the corresponding message of the macro is used.

Example

In Sybase Adaptive Server Enterprise 12:

```
begin
  select @errno = %ERRNO%,
         @errmsg = '%ERRMSG%'
  goto error
end
```

In this case, the error number and the text message are directly used.

TriggerComment

Statement for adding a trigger comment.

Example

```
comment on trigger [%QUALIFIER%]%TRIGGER% is
%.q:COMMENT%
```

Time

List of trigger time attributes.

Example

```
Before
After
```

Event

List of trigger event attributes.

Example Select

EventDelimiter

Character used to separate events if you want to assign multiple events to a trigger.

Example , (comma)

Group

The Group category contains entries defining group related parameters.

Common entries for Group

You can define values for the following common entries for the Group object in the DBMS definition.

Entry	Example (Sybase ASE 12.5)
Enable	Enable = Yes
SqlListQuery	{GROUP} select u.name from [%CATALOG%.]dbo.sysusers u where u.uid = u.gid and u.gid not in (select r.lrid from [%CATALOG%.]dbo.sysroles r) order by 1
SqlAttrQuery	—
Drop	sp_dropgroup %GROUP%
Create	sp_addgroup %GROUP% [%BIND%] [%PRIVILEGE%]
Maxlen	—
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

🔗 For a description of each of the common object entries, see the section Common object entries.

Bind

Statement for adding a member to a group.

Example

In Sybase Adaptive Server Enterprise 12.5:
`sp_adduser %USERID%, null, %GROUP%`

Unbind

Statement for removing a member from a group.

Example

In Sybase Adaptive Server Anywhere 8.0:
`revoke membership in group %GROUP% from %USER%`

SqlListChildrenQuery

SQL query for listing the members or a group.

Example

In Sybase Adaptive Server Enterprise 12.5:
{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`

SqlPermQuery

SQL Query to reverse engineer permissions granted to groups.

Role

The Role category contains entries defining role related parameters.

Common entries for Role

You can define values for the following common entries for the Group object in the DBMS definition.

Entry	Example (Sybase ASE 12.5)
Enable	Enable = Yes
SqlListQuery	<pre>{ROLE} select u.name from master.dbo.syssrvroles u order by 1</pre>
SqlAttrQuery	—
Drop	drop role %ROLE% [with override]
Create	<pre>create role %ROLE% [with passwd "%PASSWORD%"] [%BIND%] [%PRIVILEGE%]</pre>
Maxlen	—
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

Bind

Statement for assigning a role to a user or to another role.

Example

In Sybase Adaptive Server Enterprise 12.5:

```
grant role %ROLE% to %USERID%
```

Unbind

Statement for unassigning a role from a user or another role.

Example

In Sybase Adaptive Server Enterprise 12.5:

```
revoke role %ROLE% from %USER%
```

SqlListChildrenQuery

SQL query for listing the users to which the role has been assigned.

Example

In Sybase Adaptive Server Enterprise 12.5:

```
{ 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
```

SqlPermQuery

SQL Query to reverse engineer permissions granted to roles.


Privilege

The Privilege category contains entries defining privilege related parameters.

Common entries for Privilege

You can define values for the following common entries for the Privilege object in the DBMS definition.

Entry	Example (Sybase ASE 12.5)
Enable	Enable = Yes
Drop	revoke %PRIVLIST% from %USERID%
Create	grant %PRIVLIST% to %USERID%
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

 For a description of each of the common object entries, see the section Common object entries.

GrantOption

Option for the grant order.

Example

In Oracle 9i:
with admin option

RevokeOption

Option for the revoke order.

System

List of predefined system privileges allowed in the current DBMS.

Example

In Sybase ASE 12.5:
CREATE DATABASE
CREATE DEFAULT
CREATE PROCEDURE
CREATE TRIGGER
CREATE RULE
CREATE TABLE
CREATE VIEW

Permission

The Permission category contains entries defining permission related parameters.

Common entries for Permission

You can define values for the following common entries for the Permission object in the DBMS definition.

Entry	Example (Sybase ASE 12.5)
Enable	Enable = Yes
Drop	revoke %PERMLIST% on %OBJECTLIST% from %IDLIST% [%REVOKEOPTION%]
Create	grant %PERMLIST% on %OBJECTLIST% to %IDLIST% [%GRANTOPTION%]
SqlListQuery	Query for listing objects

For a description of each of the common object entries, see the section Common object entries.

GrantOption

Option for the grant order.

Example

In Sybase ASE 12.5:
with grant option

RevokeOption

Option for the revoke order.

Example

In Sybase ASE 12.5:
cascade

Join Index

The Join Index category contains entries defining join index related parameters.

Common entries for Join Index

You can define values for the following common entries for the Join Index object in the DBMS definition.

Entry	Example (Sybase Adaptive Server Anywhere 6)
Enable	Enable = Yes
Maxlen	MaxLen = 128
SqlListQuery	In Adaptive Server IQ: {JIDX ID, OWNER ID, REFRLIST ...} SELECT j.joinindex_name, u.user_name, lt.table_name ',' rt.table_name ',' FROM sysiqjoinindex j, sysuserperm u, sysiqjoinixcolumn jc, systable lt, systable rt WHERE j.joinindex_id = jc.joinindex_id AND j.creator = u.user_id AND jc.left_table_id = lt.table_id AND jc.right_table_id = rt.table_id ORDER BY joinindex_name

Entry	Example (Sybase Adaptive Server Anywhere 6)
SqlAttrQuery	<p>In Adaptive Server IQ:</p> <pre>{JIDX ID, OWNER ID, RFJNLIST ...} SELECT j.joinindex_name, u.user_name, lt.table_name '.' lc.column_name '=' rt.table_name '.' rc.column_name ',' FROM sysiqjoinindex j, sysuserperm u, sysiqjoinixcolumn jc, systable lt, systable rt, syscolumn lc, syscolumn rc WHERE j.joinindex_id = jc.joinindex_id AND j.creator = u.user_id AND jc.left_table_id = lt.table_id AND jc.right_table_id = rt.table_id AND jc.left_column_id = lc.column_id AND jc.right_column_id = rc.column_id AND lc.table_id = lt.table_id AND rc.table_id = rt.table_id ORDER BY joinindex_name</pre>
Options	—
Drop	<pre>if exists(select 1 from sys.sysiqjoinindex where joinindex_name='%JIDX%') then drop join index %JIDX% end if</pre>
Add	—
DefOptions	—
Create	<pre>create join index %JIDX% for %JIDXDEFN%</pre>
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

Header

Join index header. Anything added to this entry is added before the **create join index** statement.

Footer

Join index footer. Anything added to this entry is added after the **create join index** statement.

AddJoin

SQL statement used to define joins for join indexes.

Example Table1.coln1 = Table2.coln2

JoinIndexComment

Statement for adding a join index comment.

Example comment on join [%QUALIFIER%]%JIDX% is %.q:COMMENT%

Qualifier

You can define values for the following common entries for the Sequence object in the DBMS definition.

Entry	Example (Oracle 8)
Enable	Enable = Yes
SqlListQuery	-
Label	Label=All Qualifiers

🔗 For a description of each of the common object entries, see the section Common object entries.

Sequence

The Sequence category contains entries defining sequence related parameters.

Common entries for Sequence

You can define values for the following common entries for the Sequence object in the DBMS definition.

Entry	Example (Oracle 8)
Enable	Enable = Yes
Create	create sequence %SQNC% [%OPTIONS%]
Drop	drop sequence %SQNC%

[illegible]

Entry	Example (Oracle 8)
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

EnableOwner

Allows to define sequence owners.

Value	Result
Yes	The Owner dropdown listbox appears in the sequence property sheet and the user can select an owner for the current sequence
No	Sequence does not support owner

Example

EnableOwner = YES

SequenceComment

Statement for adding a sequence comment.

Example

comment on sequence [%QUALIFIER%]%SQNC% is %.q:COMMENT%

Synonym

The Synonym category contains entries defining synonym related parameters.

Common entries for Synonym

You can define values for the following common entries for the Synonym object in the DBMS definition.

Entry	Example (Oracle 9i)
Enable	Enable = Yes
Create	create [%VISIBILITY%]synonym [%QUALIFIER%]%SYNONYM% for [%BASEOWNER%]%BASEOBJECT%
Drop	drop [%VISIBILITY%]synonym [%QUALIFIER%]%SYNONYM%

Entry	Example (Oracle 9i)
SqlListQuery	<pre> {OWNER, SYNONYM, BASEOWNER, BASEOBJECT} [%ISODBCUSER% ? select %.q:SCHEMA%, t.synonym_name, t.table_owner, t.table_name from sys.user_synonyms t order by t.table_name : select t.owner, t.synonym_name, t.table_owner, t.table_name from sys.all_synonyms t where (1=1) [and t.synonym_name=%.q:TABLE%] [and t.owner=%.q:SCHEMA%] order by t.owner, t.synonym_name] </pre>
SqlAttrQuery	—

For a description of each of the common object entries, see the section Common object entries.

Maxlen

Statement for defining the maximum code length for an object. This value is implemented in the Check model and produces an error if the name or code exceeds the defined value.

Example MaxLen = 30

EnableAlias

Statement for defining if the alias type of synonym is allowed.

Example In DB2
 EnableAlias = Yes

DB Package

The DB Package category contains entries defining database package related parameters.

Common entries for DB Package

You can define values for the following common entries for the DB Package object in the DBMS definition.

Entry	Example (Oracle 9i)
Enable	Enable = Yes
Maxlen	MaxLen = 30
Create	Statement for creating the specification of the database package <pre>create [or replace]package %DBPACKAGE% [authid %DBPACKAGEPRIV%][%R%?[is][as]:as] %DBPACKAGESPEC% end [%DBPACKAGE%]</pre>
Drop	drop package %DBPACKAGE%
SqlListQuery	<pre>{{OWNER, DBPACKAGE} [%!SODBCUSER% ? select distinct %.q:SCHEMA%, decode (type, 'PACKAGE', name, "") from sys.user_source where type in ('PACKAGE') order by 2 : select distinct owner, decode (type, 'PACKAGE', name, "") from sys.all_source where type in ('PACKAGE') [and owner = %.q:SCHEMA%] order by 1, 2]</pre>

Entry	Example (Oracle 9i)
SqlAttrQuery	<pre> {OWNER ID, DBPACKAGE ID, TYPE ID, DBPACKAGESPEC ..., DBPACKAGEBODY ...} [%ISODBCUSER% ? SELECT %.q:SCHEMA%, NAME, TYPE, TEXT, NULL, LINE FROM SYS.USER_SOURCE S WHERE TYPE = 'PACKAGE' AND LINE > 1 AND LINE <> (SELECT MAX(S2.LINE) FROM SYS.USER_SOURCE S2 WHERE S2.TYPE = S.TYPE AND S2.NAME = S.NAME) UNION SELECT %.q:SCHEMA%, NAME, TYPE, NULL, TEXT, LINE FROM SYS.USER_SOURCE S WHERE TYPE = 'PACKAGE BODY' AND LINE > 1 AND LINE <> (SELECT MAX(S2.LINE) FROM SYS.USER_SOURCE S2 WHERE S2.TYPE = S.TYPE AND S2.NAME = S.NAME) ORDER BY NAME, TYPE, LINE : SELECT OWNER, NAME, TYPE, TEXT, NULL, LINE FROM SYS.ALL_SOURCE S WHERE TYPE = 'PACKAGE' AND LINE > 1 AND LINE <> (SELECT MAX(S2.LINE) FROM SYS.USER_SOURCE S2 WHERE S2.TYPE = S.TYPE AND S2.NAME = S.NAME) UNION SELECT OWNER, NAME, TYPE, NULL, TEXT, LINE FROM SYS.ALL_SOURCE S WHERE TYPE = 'PACKAGE BODY' AND LINE > 1 AND LINE <> (SELECT MAX(S2.LINE) FROM SYS.ALL_SOURCE S2 WHERE S2.TYPE = S.TYPE AND S2.NAME = S.NAME) ORDER BY OWNER, NAME, TYPE, LINE] </pre>
ModifiableAttributes	List of extended attributes that will be taken into account in the merge dialog box during database synchronization
ReversedStatements	List of statements that will be reverse engineered

For a description of each of the common object entries, see the section Common object entries.

CreateBody

Template for defining the body of the database package. This statement is used in the extension statement AfterCreate.

Example

In Oracle 9i:

```

create [or replace ]package body %DBPACKAGE% as
%DBPACKAGEBODY%

```

```
[begin
  %DBPACKAGEINIT%
]end [%DBPACKAGE%]
```

AfterCreate

Extension statement evaluated after the create DB package statement.

☞ For more information on extension statements, see section Script generation.

DB Package Procedure

The DB Package Procedure category contains entries defining package procedure related parameters.

Common entries for DB Package Procedure

You can define values for the following common entries for the DB Package Procedure object in the DBMS definition.

Entry	Example (Oracle 9i)
Add	%DBPKPROCTYPE% %DBPKPROC%[(%DBPKPROCPARAM%)] [return %DBPKPROCRETURN%] [%R%?[is][as]:as] %DBPKPROCCODE%]

DBProcedureBody

Template for defining the body of the package procedure in the Definition page of the package procedure property sheet.

Example

In Oracle 9i:

```
begin
end
```

ParameterTypes

Available types for package procedure.

Example

In Oracle 9i:

```
in
in out
```

out
out no copy

DB Package Variable

The DB Package Variable category contains entries defining package variable related parameters.

Common entries for DB Package Variable

You can define values for the following common entries for the DB Package Variable object in the DBMS definition.

Entry	Example (Oracle 9i)
Add	%DBPKVAR% [%DBPKVARCONST%]%DBPKVARTYPE%[:= %DBPKVARVALUE%]

DB Package Type

The DB Package Type category contains entries defining package type related parameters.

Common entries for DB Package Type

You can define values for the following common entries for the DB Package Type object in the DBMS definition.

Entry	Example (Oracle 9i)
Add	type %DBPKTYPE%[is %DBPKTYPEVAR%]

DB Package Cursor

The DB Package Cursor category contains entries defining package cursor related parameters.

Common entries for DB Package Cursor

You can define values for the following common entries for the DB Package Cursor object in the DBMS definition.

Entry	Example (Oracle 9i)
Add	cursor %DBPKCURSOR% [(%DBPKPROCPARAM%)] return %DBPKCURSORRETURN% is %DBPKCURSORQUERY%

ParameterTypes

Available types for package cursor.
in

Example

DB Package Exception

The DB Package Exception category contains entries defining package exception related parameters.

Common entries for DB Package Exception

You can define values for the following common entries for the DB Package Exception object in the DBMS definition.

Entry	Example (Oracle 9i)
Add	%DBPKEXC% exception

DB Package Parameter

The DB Package Parameter category contains entries defining package parameter related parameters.

Common entries for DB Package Parameter

You can define values for the following common entries for the DB Package Parameter object in the DBMS definition.

Entry	Example (Oracle 9i)
Add	%DBPKPARAM% [%DBPKPARMTYPE%]%DBPKPARMDTTP%

DB Package Pragma

The DB Package Pragma category contains entries defining package pragma related parameters.

Common entries for DB Package Pragma

You can define values for the following common entries for the DB Package Pragma object in the DBMS definition.

Entry	Example (Oracle 9i)
Add	pragma %DBPKPRAGMA% (%DBPKPRAGMAOBJ%, %DBPKPRAGMAPARAM%)

Commands for all objects

The following commands are defined in the Objects category and apply to all objects.

MaxConstLen

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

Maximum length for constraint names:

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.

Example

```
MaxConstLen = 128
```

EnableOption

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

Value	Target database	Result
Yes	Supports physical options, for example, Tablespace, Storages, or physical options such as pctfree, and fillfactor	The Options page is available from the object property sheet
No	Does not support physical options	The Options page is not available from the object property sheet. In the Database page of the database generation box, all the generation parameters are grayed and can not be selected

Example

```
EnableOption = YES
```

Data type category

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

AmcdDataType

Data type translation table showing the mapping of PowerDesigner internal data types to the DBMS definition file data types. This is used during CDM to PDM generation and with the Change Current DBMS command. The following variables are used to qualify the data type:

Variable	Indicates
%n	Length of the data type
%s	Size of the data type
%p	Precision of the data type

Using the variables defined to qualify the mapping, PowerDesigner creates an effective interface between its native data types and those of the chosen DBMS.

Example

In Sybase Adaptive Server Enterprise 12:

ASE 12 data type	PowerDesigner data type
A%n	char(%n)
VA%n	varchar(%n)
LA%n	varchar(%n)
BT	tinyint

PhysDataType

Data type translation table showing the mapping from the DBMS definition file data types to the PowerDesigner internal data types. This is used during PDM to CDM generation and with the Change Current DBMS command (use of PowerDesigner internal data types to find a match between two DBMS definition file data types).

The following variables are used to qualify the data type:

Variable	Indicates
%n	Length of the data type
%s	Size of the data type
%p	Precision of the data type

Using the variables defined to qualify the mapping, PowerDesigner creates an effective interface between its native data types and those of the chosen DBMS.

Example

In Sybase Adaptive Server Enterprise 12:

PowerDesigner data type	ASE 12 data type
sysname	VA30
integer	I

PhysDttpSize

Table of storage sizes of DBMS data types.

The proper storage size is assigned to each data type for the selected DBMS.

Example

In Sybase Adaptive Server Enterprise 12.5:

ASA 6 data type	Storage size
smallmoney	8
smalldatetime	4
datetime	8
timestamp	8

OdbcPhysDataType

Data type translation table showing the mapping from ODBC data types to the DBMS data types.

This table is used during reverse engineering through ODBC to map the data types extracted from the database to DBMS data types. Although data types are identical concepts between the database and the DBMS, the way data types are stored in the database differs from the DBMS notation. For example, you create a column with a data type decimal in the database. The decimal data type is stored as decimal(30,6) in Sybase Adaptive Server Anywhere 6. This precision (30,6) did not appear in your create order, and it is not required when using DBMS data types, "decimal" is enough. When you reverse engineer the database, the process retrieves data types as they are stored in the database, i.e. decimal(30,6). At this stage PowerDesigner will use this translation table to map the ODBC data types (decimal(30,6)) with the preferred notation of DBMS data type (decimal).

Example

In Sybase Adaptive Server Anywhere 6:

ODBC data type	ASA 6 data type
numeric (30,6)	numeric
char (1)	char
binary (1)	binary
decimal (30,6)	decimal

PhysOdbcDataType

Data types translation table from target database data types to ODBC data types.

Example

In MS Access 95/97:

Physical	ODBC data type
Integer	Short
LongInteger	Long
OLE	LongBinary

PhysLogADTType

Abstract Data types translation table from target database ADT to internal ADT. You should not modify these values.

Example In Sybase Adaptive Server Anywhere 6:

Database physical data type	PowerDesigner internal data type
Java	Java

LogPhysADTType

Abstract Data types translation table from internal abstract data types to target database abstract data types. You should not modify these values.

Example In Sybase Adaptive Server Anywhere 6:

PowerDesigner internal data type	Database physical data type
Undefined	Array
Undefined	List
Java	Java
Undefined	Object
Undefined	Structured

AllowedADT

List of ADT which can be used as data type for domains and columns.

Example In Sybase Adaptive Server Anywhere 6:

JAVA

HostDataType

Data types translation from database data type to procedure data type.

Example In Oracle8:

PowerDesigner internal data type	Database physical data type
Number	DEC
Number	REAL
Number	DOUBLE PRECISION

PowerDesigner internal data type	Database physical data type
Float	FLOAT
Integer	INT
Integer	INTEGER
Varchar	VARCHAR (%n)
Varchar	VARCHAR2 (%n)

PDM variables

You can incorporate variables in the SQL queries of the selected DBMS. These variables are replaced with the actual values from your model when the scripts are generated.

Variables for database generation, and triggers and procedures generation

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

Variables for reverse engineering

Variable name	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

Variable name	Comment
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 ODBC 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 ODBC 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 reverse ODBC, when the length is not defined in the system tables
VALUE	One value from the list of values in a column or domain
PERMISSION	Allow to reverse engineer permissions set on a database object
PRIVILEGE	Allow to reverse engineer privileges set on a user, a group, or a role

Variables for database synchronization

Variable name	Comment
OLDOWNER	Old owner name of Object. See also OWNER
NEWOWNER	New owner name of Object. See also OWNER
OLDQUALIFIER	Old qualifier of Object. See also QUALIFIER
NEWQUALIFIER	New qualifier of Object. See also QUALIFIER
OLDTABL	Old code of Table
NEWTABL	New code of Table
OLDCOLN	Old code of Column
NEWCOLN	New code of Column
OLDNAME	Old code of Sequence
NEWNAME	New code of Sequence

Variables for database security

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

Variables for metadata

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

Common variables for all named objects

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

Common variables for objects

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

Variable name	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 ODBC reverse dialog boxes, because their owner is not defined yet
DBPREFIX	Database prefix of objects (name of Database + '.' if database defined)
QUALIFIER	Whole object qualifier (database prefix + owner prefix)
OPTIONS	SQL text defining physical options for Object
CONSTNAME	Constraint name of Object
CONSTRAINT	Constraint SQL body of Object. Ex: (A <= 0) AND (A >= 10)
CONSTDEFN	Column constraint definition. Ex: constraint C1 checks (A>=0) AND (A<=10)
RULES	Concatenation of Server expression of business rules associated with Object
NAMEISCODE	True if the object (table, column, index) name and code are identical (AS 400 specific)

Variables for DBMS, database options

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

Variables for tables

Variable name	Comment
TABLE	Generated code of Table
TNAME	Name of Table

Variable name	Comment
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
CLUSTERCOLUMNS	List of columns used for a cluster

Variables for domains and columns checks

Variable name	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
ISRONLY	TRUE if Read-only attribute of standard check has been selected
DEFAULT	Default value
MINVAL	Minimum value
MAXVAL	Maximum value
VALUES	List of values. Ex: (0, 1, 2, 3, 4, 5)
LISTVAL	SQL constraint associated with List of values. Ex: C1 in (0, 1, 2, 3, 4, 5)
MINMAX	SQL constraint associated with Min and max values. Ex: (C1 <= 0) AND (C1 >= 5)
ISMAND	TRUE if Domain or column is mandatory
MAND	Contains Keywords "null" or "not null" depending on Mandatory attribute
NULL	Contains Keyword "null" if Domain or column is not mandatory

Variable name	Comment
NOTNULL	Contains Keyword "not null" if Domain or column is mandatory
IDENTITY	Keyword "identity" if Domain or Column is identity (Sybase specific)
WITHDEFAULT	Keyword "with default" if Domain or Column is with default
ISUPPERVAL	TRUE if the upper-case attribute of standard check has been selected
ISLOWerval	TRUE if the lower-case attribute of standard check has been selected

Variables for columns

Parent Table variables are also available.

Variable name	Comment
COLUMN	Generated code of Column
COLNNO	Position of Column in List of columns of Table
COLNNAME	Name of Column
COLNCODE	Code of Column
PRIMARY	Contains Keyword "primary" if Column is primary key column
ISPKEY	TRUE if Column is part of Primary key
FOREIGN	TRUE if Column is part of one foreign key
COMPUTE	Compute constraint text

Variables for abstract data types

Variable name	Comment
ADT	Generated code of Abstract data type
TYPE	Abstract data type's type. It contains keywords like "array", "list", ...
SIZE	Abstract data type size

Variable name	Comment
FILE	Abstract data type Java file
ISARRAY	TRUE if Abstract data type is of type array
ISLIST	TRUE if Abstract data type is of type list
ISSTRUCT	TRUE if Abstract data type is of type structure
ISOBJECT	TRUE if Abstract data type is of type object
ISJAVA	TRUE if Abstract data type is of type JAVA class
ADTDEF	Contains Definition of Abstract data type

Variable for abstract data type attributes

Variable name	Comment
ADTATTR	Generated code of Abstract data type attribute

Variable for domains

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

Variables for rules

Variable name	Comment
RULE	Generated code of Rule
RULENAME	Rule name
RULECODE	Rule code
RULECEXP	Rule client expression
RULESEXP	Rule server expression

Variables for ASE & SQL Server

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

Variables for sequences

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

Variables for indexes

Variable name	Comment
INDEX	Generated code of Index
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
INDXDEF	List of index columns without separator, on different lines. Example: ColnA ColnB ColnC

Variables for join indexes (IQ)

Variable name	Comment
JIDX	Generated code for join index
JIDXDEFN	Complete body of join index definition
REFRLIST	List of references (for ODBC)
RFJNLIST	List of reference joins (for ODBC)

Variables for index columns

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

Variables for references

Variable name	Comment
REFR	Generated code of Reference
PARENT	Generated code of Parent table
PNAME	Name of Parent table
PCODE	Code of Parent table
PQUALIFIER	Qualifier of Parent table. See also QUALIFIER.
CHILD	Generated code of Child table
CNAME	Name of Child table
CCODE	Code of Child table
CQUALIFIER	Qualifier of Child table. See also QUALIFIER.
REFRNAME	Reference name
REFRCODE	Reference code
FKCONSTRAINT	Foreign key (reference) constraint name

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

Variables for reference columns

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

Variables for keys

Variable name	Comment
COLUMNS 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 views

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

Variables for triggers

Parent Table variables are also available.

Variable name	Comment
ORDER	Order number of Trigger (in case DBMS support more than one trigger of one type)
TRIGGER	Generated code of trigger

Variable name	Comment
TRGTYPE	Trigger type. It contains Keywords "beforeinsert", "afterupdate", ...etc.
TRGEVENT	Trigger event. It contains Keywords "insert", "update", "delete"
TRGTIME	Trigger time. It contains Keywords NULL, "before", "after"
REFNO	Reference order number in List of references of Table
ERRNO	Error number for standard error
ERRMSG	Error message for standard error
MSGTAB	Name of Table containing user-defined error messages
MSGNO	Name of Column containing Error numbers in User-defined error table
MSGTXT	Name of Column containing Error messages in User-defined error table
SCRIPT	SQL script of trigger or procedure.
TRGBODY	Trigger body (only for Oracle ODBC reverse engineering)
TRGDESC	Trigger description (only for Oracle ODBC reverse engineering)
TRGDEFN	Trigger definition

Variables for procedures

Variable name	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)

CHAPTER 2

Managing Profiles

About this chapter

This chapter explains how to manage PowerDesigner profiles.

Contents

Topic	Page
Understanding the profile concept	174
Defining a stereotype	180
Defining a criterion	184
Defining a custom symbol in a profile	186
Defining extended attributes in a profile	189
Defining a custom check in a profile	192
Defining templates and generated files in a profile	200
Using profiles: a case study	205

Understanding the profile concept

PowerDesigner uses the UML profile concept to extend the definition of its metamodel in order to address the need of customized models and methods for building different applications.

Prerequisite

It is strongly recommended to have a good understanding of the metamodel structure and philosophy to better use PowerDesigner profiles.

What is a profile?

A profile is an extension mechanism used for customizing a metamodel with additional semantics. Profiles are used for creating categories of objects (stereotypes and criteria), customizing the graphics of objects, adding additional metadata to objects (extended attributes), and defining new or modified generation capabilities (templates).

You create a profile when you need to design a user-defined methodology, a model with predefined meaning or for a specific generation target.

In PowerDesigner, a profile lets you define different extensions for the metaclasses available in the metamodel. Such extensions, like stereotypes, symbols, or checks, are used to complement the standard definition of a metaclass.

Profiles appear in all DBMS, object languages and extended model definitions delivered with PowerDesigner. By default, each new resource file has a profile upon creation.

Example

The PowerDesigner Business Process Model lets you design the various internal tasks and processes of a business and the way partners interact with these tasks and processes.

If you import the ebXML extended model definition into the model, you can slightly modify the semantics of the model by providing stereotypes, extended attributes, templates and other extensions adapted to the ebXML standard. You can use the stereotypes defined in the ebXML extended model definition to further define the processes in your model and turn them into <<BinaryCollaboration>>, <<BusinessTransaction>> and so on.

🔗 For more information on ebXML, see the technical document in \ebXML directory.

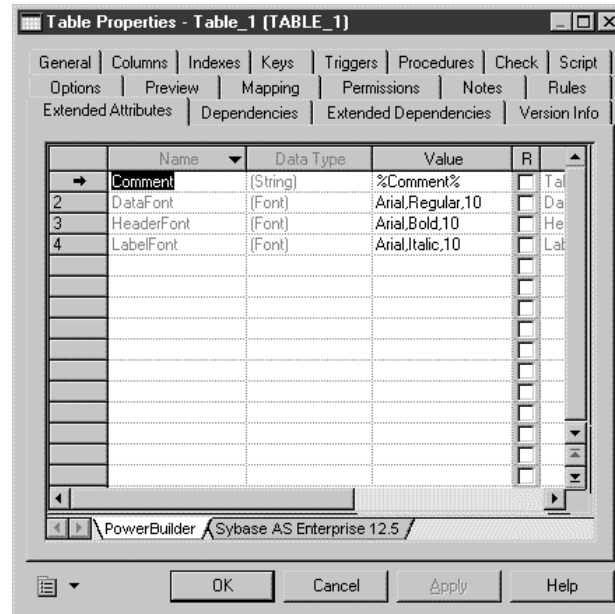
Extension levels	<p>The extension mechanism can be implemented at different levels:</p> <p>At the metaclass level When you define an extension in a metaclass, it applies to all instances of the selected metaclass. The extension is global</p> <p>At the instance level If you define an extension in a stereotype or in a criterion, it applies to the instances with the correct stereotype or that verify the criterion condition. The extension is on a per-instance basis</p>
Inheritance	<p>Profile extensions support inheritance in order to reuse extensions among the metaclasses of a common parent. Common extensions should be defined on abstract metaclasses in order for the concrete metaclasses to inherit from them.</p>

Available extensions

Depending on the level where you define the extensions, different items are available.

You can attach several resource files to a model, that are several extended model definitions together with the object language or the DBMS of the model. The extensions defined in each resource file appear in separate pages to indicate their origin.

For example in a PDM, extended attributes may be defined for a table in the DBMS and in the different extended model definitions attached to the model, they appear in different tabbed pages in the Extended Attributes page in the property sheet of the table.



Extension conflict

A conflict occurs when extensions with identical names are defined on the same metaclasses in different resource files attached to the same model.

For example, stereotype <<document>> is defined for components in two extended model definitions attached to the same OOM. In each extended model definition, stereotype <<document>> has a different custom symbol. A conflict occurs when you create a component and assign the <<document>> stereotype: PowerDesigner will randomly select one of the symbols.

In case of conflict between DBMS or object language and extended model definition, the extended model definition extension usually prevails.

Metaclass extensions

Extensions defined on a metaclass apply to all instances of the metaclass. You can extend a metaclass semantics using the following elements:


- ◆ **Stereotypes** are used to sub-classify instances of a metaclass
- ◆ **Extended attributes** are used to further define a metaclass and also control generation
- ◆ **Custom symbol** and **tool** can help users better identify the metaclass

	<ul style="list-style-type: none"> ◆ Custom checks are used to fine-tune the verification of the metaclass in your model ◆ Generated files and templates are used to customize generation for the metaclass ◆ Criteria are used to evaluate conditions on a metaclass
Stereotype and criterion	<p>Extensions defined for a stereotype or a criterion, apply to the metaclass instances with the stereotype or to the metaclass instances that verify the criterion condition.</p> <p>You can define the following extensions in a stereotype or a criterion:</p> <ul style="list-style-type: none"> ◆ Extended attributes are used to further define the metaclass instance and also control generation ◆ Custom symbol and tool (custom tool is only available for stereotypes) can help users better identify the metaclass instance ◆ Custom checks are used to fine-tune the verification of the metaclass instance in your model ◆ Generated files and templates are used to customize generation for the metaclass instance

Adding a metaclass to a profile

Depending on the type of resource file you are working on, the list of pre-existing metaclasses can change in the Profile category.

The procedure for adding a metaclass requires selecting items among a wide range of existing PowerDesigner metaclasses. In the Selection dialog box, you can use the Modify Metaclass Filter tool to display all, concrete or abstract conceptual metaclasses in the selection list.

 For more information on the PowerDesigner metamodel, see chapter PowerDesigner Public Metamodel.

Enable selection in file generation	<p>When you add a metaclass in a profile, you can select the Enable Selection in File Generation check box in order to have the corresponding metaclass instances appear in the Selection page of the extended generation dialog box. If a parent metaclass is selected for file generation, children metaclasses also appear in the Selection page.</p>
-------------------------------------	--

❖ To add a metaclass to a profile:

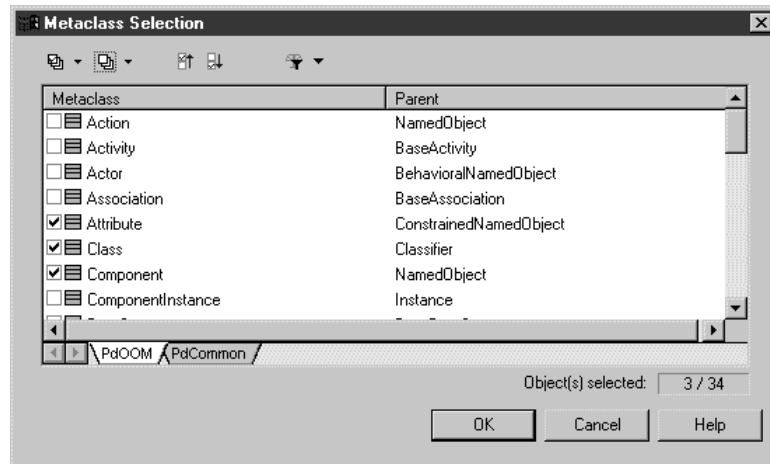
- 1 Right-click the Profile category and select Add Metaclasses in the contextual menu.

The Metaclass Selection dialog box appears.

- 2 Click the appropriate tab to display the page containing the metaclass you want to select.

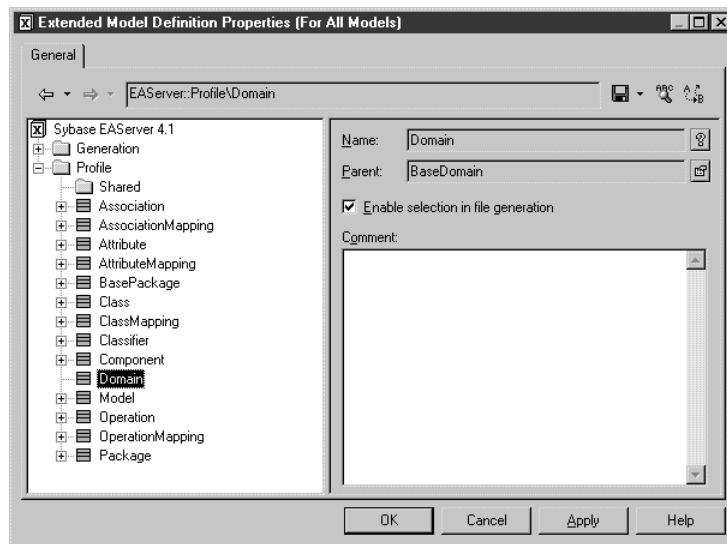
You can use the filter tool to display the metaclass you want to add to the profile.

- 3 Select one or several metaclasses.



- 4 Click OK.

The categories corresponding to the selected metaclasses appear under the Profile category. The name and parent of the metaclass cannot be modified.



- 5 <optional> Select or clear the Enable Selection in File Generation check box.
- 6 <optional> Type a comment in the Comment box.

Add parent metaclass

If the parent metaclass does not appear under Profile, when you click the Properties tool beside the Parent metaclass, a message appears to let you automatically insert the metaclass node under Profile

Defining a stereotype

Stereotypes are used to sub-classify instances and gather extensions for a metaclass supporting the concept of stereotype.

You can define several stereotypes for a given metaclass. The stereotypes you create can be applied to any instance of the metaclass. However, the extensions defined in a stereotype only apply to the instances on which the stereotype is applied. This is the reason why using stereotypes is considered as a per-instance extension mechanism.


Inheritance

Stereotypes support **inheritance**: characteristics of the parent of a stereotype are inherited by the children stereotypes.

Stereotype properties

When you define a stereotype, you have to define the following properties:

Property	Description
Name	Name of the stereotype that will appear in the Stereotype dropdown listbox in the object property sheet
Parent	Name of the parent of the current stereotype. You can use the dropdown listbox to select a stereotype defined in the same metaclass or in a parent metaclass. You can use the Properties button to select the parent stereotype in the tree view and display its properties
Abstract	The stereotype cannot be applied to metaclass instances, this stereotype does not appear in the stereotype dropdown listbox in the object property sheet, and can only be used as a parent of other child stereotypes
Palette Custom Tool	If selected, allows you to associate a tool in a palette to the current stereotype. This option is available for objects supporting symbols, it cannot be used for the stereotype of an attribute for example
Comment	Additional information about the stereotype

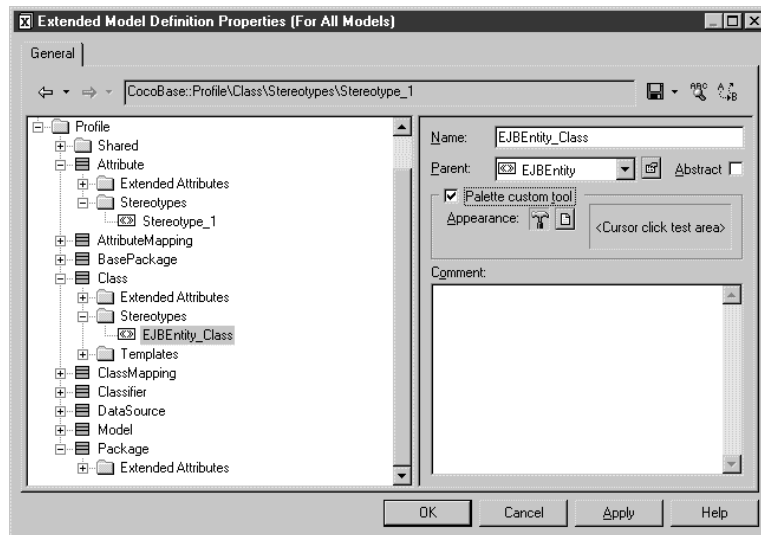
 For more information on how to attach a custom tool to a stereotype, see section [Attach a tool to a stereotype](#).

Creating a stereotype

You create one or several stereotypes for a given metaclass. You cannot create a stereotype within another stereotype or within a criterion.

❖ To create a stereotype:

- 1 Right-click a metaclass and select New→Stereotype in the contextual menu.
A new stereotype is created with a default name.
- 2 Type a stereotype name in the Name box.
- 3 <optional> Select a parent stereotype in the Parent dropdown listbox.
- 4 <optional> Select the Palette Custom Tool check box to associate a tool to the stereotype, and select a tool appearance.



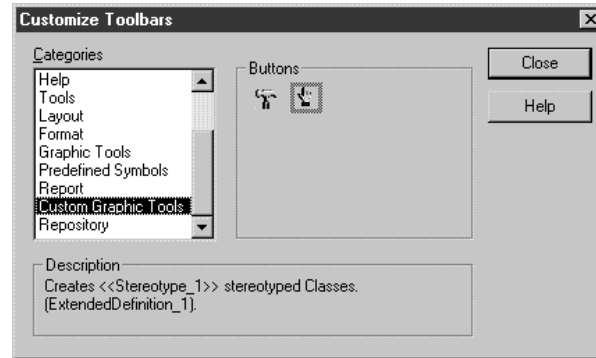
Once you have created the stereotype, you can define extensions like a custom tool, or custom checks for the stereotype. These extensions will apply to the metaclass instances with the stereotype.

Attach a tool to a stereotype

You can attach a tool to the stereotype you have defined in order to ease the creation of stereotyped instances of the metaclass. All custom tools appear in a tool palette with the same name as the resource file to which they belong.

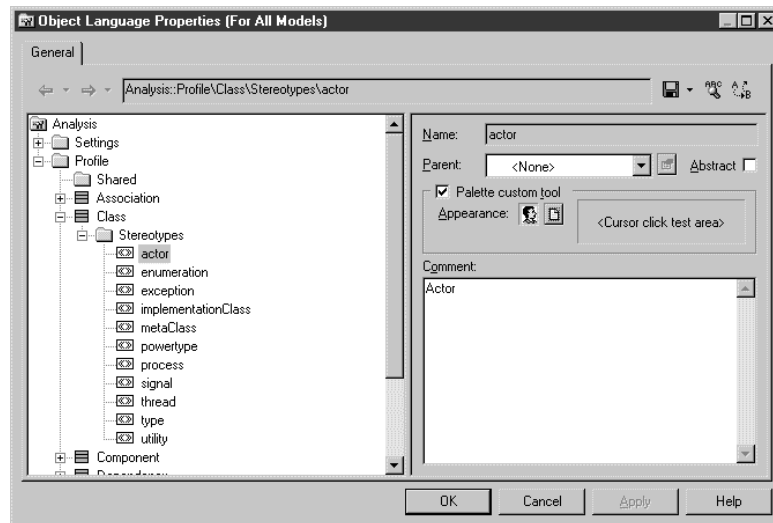
You can modify the default icon of the tool using the browse button and selecting a file with the .cur or .ico extension. You can create your own icons or cursors using 3rd party editors or you can purchase them from graphic designers.

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



❖ **To attach a tool to a stereotype:**

- 1 In the Stereotype property page, select the Palette Custom Tool check box to enable the fields in the lower part of the dialog box.
- 2 Click the Browse tool to display a standard Open dialog box in which you can select a file with the .cur or .ico extension.



- 3 <optional> You can click inside the <Cursor Click Text Area> to verify how the cursor looks.
- 4 Click Apply.

Defining a criterion

Criteria are another and more generic extension mechanism for the PowerDesigner metaclasses. Criteria can also be used for objects that do not support stereotypes, like CDM or PDM objects.

A criterion defines a condition with the expressions used in the .if macro of the PowerDesigner generation template language (GTL). You can also use the extended attributes defined at the metaclass level in the criterion condition, but you cannot use those defined in the criterion itself.

When a metaclass instance verifies the criterion condition, the extensions defined on the criterion are applied to this instance.

🔗 For more information on the PowerDesigner generation template language and the .if macro, see chapter Generation Reference Guide.

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

Example

In an OOM for Java, you want to define special attributes for bean classes. You can create a criterion called "Bean Class" using the following template:

```
(%isBeanClass%)
```

And then you define a custom symbol for this criterion.

🔗 For more information on custom symbols, see section Defining a custom symbol in a profile.

❖ To define a criterion:

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

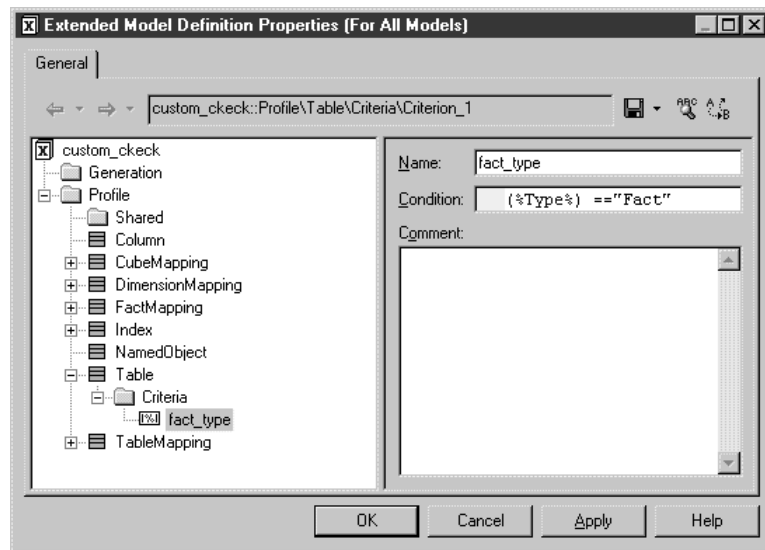
A new criterion is created with a default name.

- 2 Modify the default name in the Name box.
- 3 Type a condition in Condition box.

In the criterion syntax, you can use any valid expression used by the .if macro but you should not type the macro itself.

🔗 For more information on the .if macro, see section Using macros in chapter Generation Reference Guide.

- 4 <optional> Type a comment in the Comment box.



- 5 Click OK.

You can define extensions for the new criterion.

Defining a custom symbol in a profile

PowerDesigner lets you customize the format of symbols in order to make the diagram reflect metaclass semantics. You can customize symbols and the modified format of the symbol can become a default display preference.

You can define one custom symbol per metaclass, stereotype or criterion.

If you define a custom symbol in a metaclass category All instances of the metaclass and the children of this metaclass will use the custom symbol


If you define a custom symbol in a selected stereotype or criterion Only the instances with the selected stereotype or criterion display the custom symbol

In any other situation The default symbol format as shown in the display preferences dialog box is used

How to customize a symbol

You customize the format of a symbol using the Symbol Format dialog box. Format parameters are available in the different tabbed pages of the Symbol Format dialog box:

- ◆ Size: to define a size, auto adjustment and aspect ratio
- ◆ Line Style: to define the color, width, style, and corners of lines, and the display of arrows on lines
- ◆ Fill: to define the symbol fill color and style
- ◆ Shadow: to add a shadow to the symbol and define its color
- ◆ Font: to define the font style and size of the symbol
- ◆ Custom Shape: to modify the default shape of a symbol by selecting among bitmaps, metafiles, or icons

 For more information on the Symbol Format dialog box, see section Modifying symbol appearance in chapter Model Graphics in the *General Features Guide*.

Defining a custom symbol

You define a single custom symbol per metaclass, stereotype or criterion. Metaclass children inherit any custom symbol defined on a parent metaclass.

When you create a custom symbol, the name of this symbol is automatically assigned, you may type a comment in the corresponding box if you need to further define the custom symbol. You can also modify the default size of the symbol. Most customization actions are performed from the Symbol Format dialog box that appears when you click the Modify button in the Custom Symbol page.

For more information on the Symbol Format dialog box, see section Modifying symbol appearance in chapter Model Graphics in the *General Features Guide*.

❖ **To define a custom symbol:**

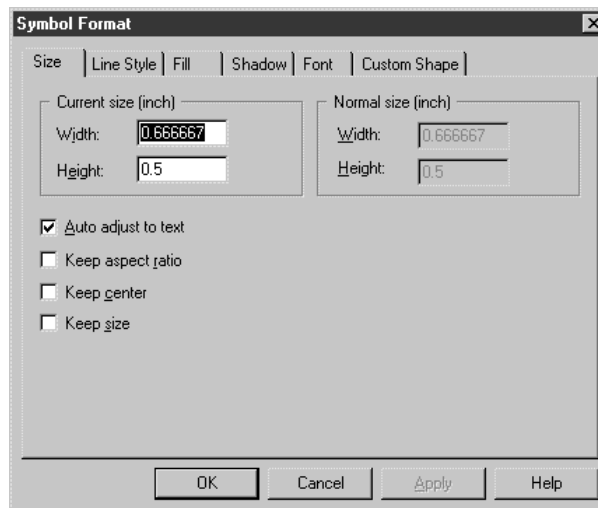
- 1 Right-click a metaclass in the Profile category and select New→Custom Symbol.

or

Right-click a stereotype or a criterion in a metaclass and select New→Custom Symbol.

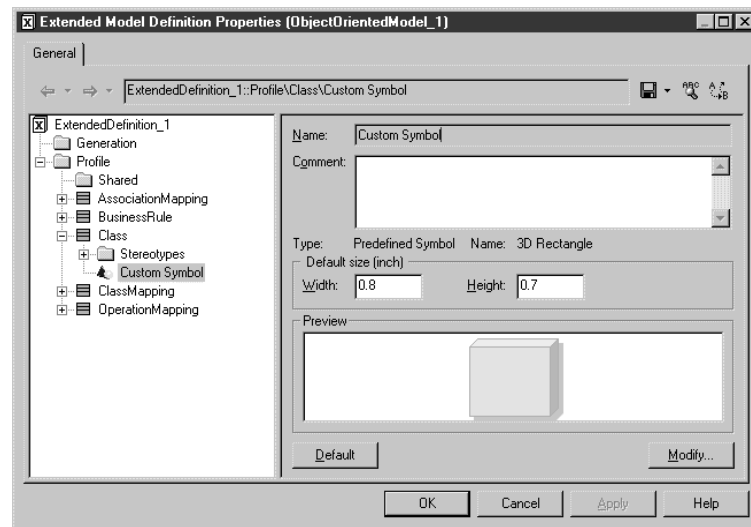
A new custom symbol is created under the selected category.

- 2 <optional> Type a comment in the Comment box.
- 3 Define a default size in the Default Size groupbox.
- 4 Click the Modify button to display the Symbol Format dialog box.



- 5 Modify the symbol format in the different pages of the Symbol Format dialog box.

6 Click OK



7 Click Apply in the resource editor.

Defining extended attributes in a profile

Extended attributes are defined for a metaclass, a stereotype or a criterion. They are used to complement the definition of the metaclass or its instances in order to:

- ◆ Control generation for a given generation target. For example, the extended model definition for BEA Weblogic contains extended attributes like Weblogic-generator-type used to define a type of generator
- ◆ Further define model objects

Extended attributes have a type used to define their data type and authorized values.

Creating an extended attribute type

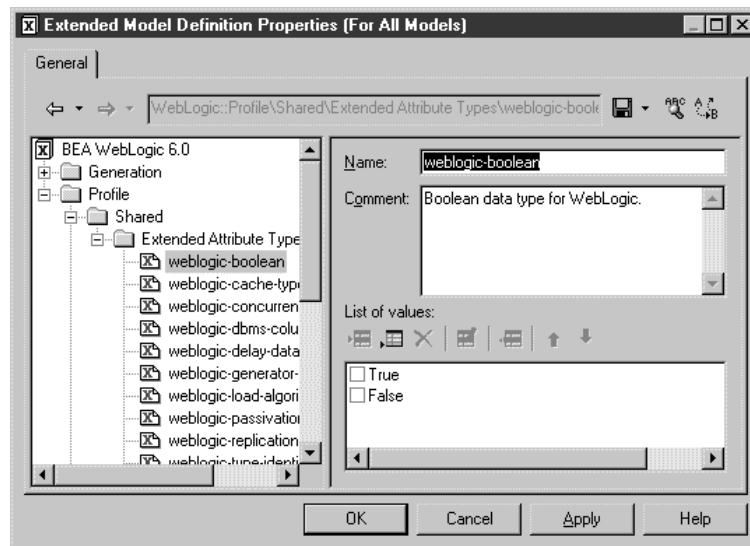
Extended attribute types are used to define the data type and authorized values of extended attributes. Extended attribute types are created in the Shared folder. Once defined, these types are available from the Data Type dropdown listbox in the extended attribute property page.

❖ To create an extended attribute type:

- 1 Right-click the Profile\Shared category and select New→Extended Attribute Type in the contextual menu.

A new extended attribute type is created under the Profile\Shared\Extended Attribute Types category.

- 2 Select the new extended attribute type and type a name, comment, list of values, and a default value in the corresponding boxes in the right pane of the editor.



- 3 Click Apply.

Creating an extended attribute

An extended attribute is another extension to the definition of a metaclass. You create an extended attribute:

- ◆ In the metaclass if you want to apply it to all the instances of the metaclass
- ◆ In a stereotype or a criterion, if you want to apply the extended attribute to selected instances of a metaclass

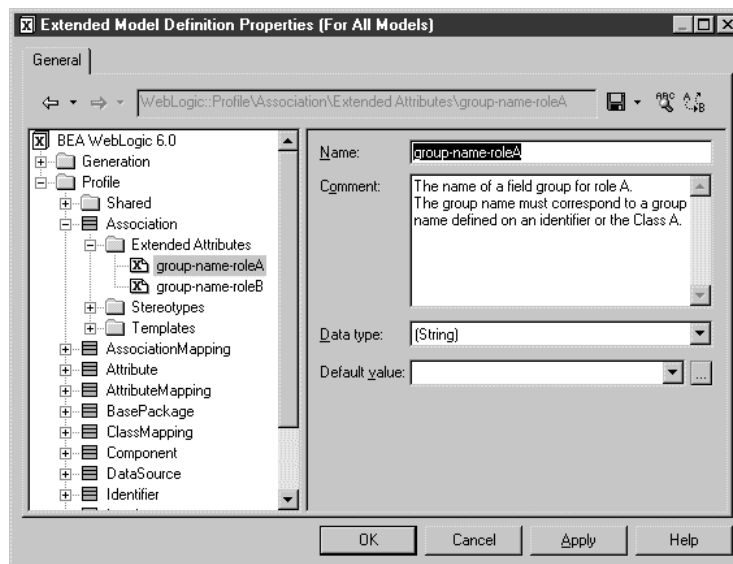
Extended attributes defined in a parent metaclass are inherited by children metaclasses. They appear in the Extended Attributes tab of an object property sheet.

❖ To create an extended attribute:

- 1 Right-click a metaclass in the Profile category and select New→Extended Attribute.
or
Right-click a stereotype or a criterion in a metaclass and select New→Extended Attribute.

A new extended attribute is created under the selected category.

- 2 Select the new extended attribute and type a name, comment, data type and default value in the corresponding boxes in the right pane of the editor.



- 3 Click Apply.

Defining a custom check in a profile

You can add custom checks on concrete and abstract metaclasses, however, custom checks are only relevant on modeling metaclasses (for example business rules); they should not be defined on technical objects like symbols or report items.

✍ For more information on concrete and abstract metaclasses, see chapter PowerDesigner Public Metamodel.

You create custom checks using the **Visual Basic scripting language**.

✍ For more information on the Visual Basic scripting language, see section Accessing objects using VBScript in chapter Managing Objects in the *General Features Guide*.

Custom checks can be defined on a metaclass and also for a stereotype or a criterion.

When you define a check on a metaclass It applies to all instances of the selected metaclass. The custom checks defined on an abstract metaclass appear in all its children categories in the Check Model dialog box. For example, you define a check on the metaclass Classifier, this check will appear in the Class and Interface categories.

When you define a check on a stereotype or a criterion It allows you to bind the custom check to a certain stereotype or criterion condition: if the object has the correct stereotype or meets a condition, then PowerDesigner will invoke the custom check during the check model process.

Several custom checks can be added to a given metaclass, they will all appear in the Check Model Parameters dialog box where you can modify their severity or auto-fix parameters.

Custom check properties

When you create custom checks you have to define the following general properties:

Parameters	Description
Name	Name of the custom check. This name appears under the selected object category in the Check Model Parameters dialog box. This name is also used (concatenated) in the check function name to uniquely identify it
Comment	Additional information about the custom check

Parameters	Description
Help Message	Text displayed in the message box that appears when the user selects Help in the custom check context menu in the Check Model Parameters dialog box
Output message	Text displayed in the Output window during check execution
Default severity	Allows you to define if the custom check is an error (major problem in the model syntax) or a warning (minor problem or just recommendation)
Execute the check by default	Allows you to make sure that this custom check is selected by default in the Check Model Parameters dialog box
Enable automatic correction	Allows you to authorize automatic correction for the custom check
Execute the automatic correction by default	Allows you to make sure that automatic correction for this custom check is executed by default

The **Check Script** page is used to define the body of the custom check function.

The **Autofix Script** page is used to define the body of the autofix function.

The **Global Script** page is used for sharing library functions and static attributes in the resource file.

Defining the script of a custom check

You type the script of a custom check in the Check Script page of the custom check properties. By default, the Check Script page displays the following script items:

- ◆ %Check% is the function name, it is passed on parameter obj. It is displayed as a variable, this variable is a concatenation of the name of the resource file, the name of the current metaclass, the name of the stereotype or criterion, and the name of the check itself defined in the General page. If any of these names contains an empty space, it is replaced by an underscore
- ◆ A comment explaining the expected script behavior
- ◆ The return value line

Example

In Sybase AS IQ, you need to create additional checks on indexes in order to verify their columns. The custom check you are going to create verifies if indexes of type HG, HNG, CMP, or LF are linked with columns which data type VARCHAR length is higher than 255.

❖ To define the script of a custom check:

- 1 Right-click a metaclass in the Profile category and select New→Custom Check.
or
Right-click a stereotype or a criterion in a metaclass and select New→Custom Check.

A new custom check is created under the selected category.

- 2 Click the Check Script tab in the custom check properties to display the script editor.

By default, the function is declared at the beginning of the script. You should not modify this line.

- 3 Type a comment after the function declaration in order to document the custom check.

- 4 Declare the different variables used in the script.

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

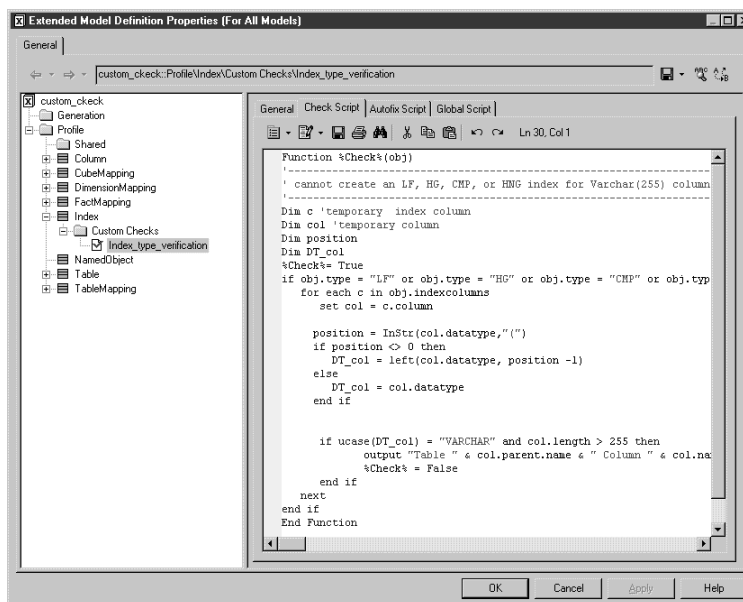
- 5 Declare the function body.

```
%Check%= True

if obj.type = "LF" or obj.type = "HG" or obj.type =
"CMP" or obj.type = "HNG" then
    for each c in obj.indexcolumns
        set col = c.column

        position = InStr(col.datatype, "(")
        if position <> 0 then
            DT_col = left(col.datatype, position -1)
        else
            DT_col = col.datatype
        end if
    if ucase(DT_col) = "VARCHAR" and col.length > 255
    then
        output "Table " & col.parent.name & "
Column " & col.name & " : Data type is not compatible
with Index " & obj.name & " type " & obj.type
        %Check% = False
    end if
```

6 Declare the end of the function.



7 Click Apply.

Defining the script of an autofix

If the custom check you have defined supports an automatic correction, you can type the body of this function in the Autofix Script page of the custom check properties.

The autofix is visible in the Check Model Parameters dialog box, it is selected by default if you select the Execute the Automatic Correction by Default check box in the General page of the custom check properties.

By default, the Autofix Script page displays the following script items:

- ◆ **%Fix%** is the function name, it is passed on parameter **obj**. It is displayed as a variable, this variable is a concatenation of the name of the resource file, the name of the current metaclass, the name of the stereotype or criterion, and the name of the fix. If any of these names contains an empty space, it is replaced by an underscore
- ◆ The variable **outmsg** is a parameter of the fix function. You need to specify the fix message that will appear when the fix script will be executed

- ◆ The return value line

We will use the same example as in section Defining the script of a custom check, to define an autofix script that removes the columns with incorrect data type from index.

❖ **To define the script of an autofix:**

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

By default, the function is declared at the beginning of the script. You should not modify this line.

- 2 Type a comment after the function declaration in order to document the custom check.

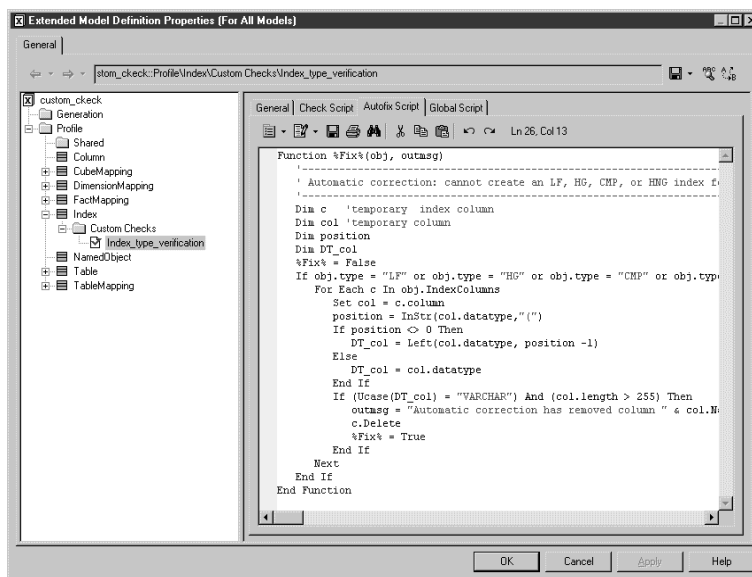
- 3 Declare the different variables used in the script.

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

- 4 Declare the function body.

```
%Fix% = False
If obj.type = "LF" or obj.type = "HG" or obj.type
= "CMP" or obj.type = "HNG" Then
  For Each c In obj.IndexColumns
    Set col = c.column
    position = InStr(col.datatype, "(")
    If position <> 0 Then
      DT_col = Left(col.datatype, position -1)
    Else
      DT_col = col.datatype
    End If
    If (Ucase(DT_col) = "VARCHAR") And
(col.length > 255) Then
      outmsg = "Automatic correction has
removed column " & col.Name & " from index."
      c.Delete
      %Fix% = True
    End If
  Next
End If
```

5 Declare the end of the function.



6 Click Apply.

Using the global script

The Global Script page is used to store functions and static attributes that may be reused among different functions. This page displays a library of available sub-functions.

Example

In the Sybase AS IQ example, you could have a function called `DataTypeBase` that retrieves the data type of an item in order to further analyze it.

This function is defined as follows:

```

Function DataTypeBase(datatype)
Dim position
position = InStr(datatype, "(")
If position <> 0 Then
DataTypeBase = Ucase(Left(datatype, position - 1))
Else
DataTypeBase = Ucase(datatype)
End If
End Function
  
```

In this case, this function only needs to be referenced in the check and autofix scripts:

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

```

Dim c 'temporary index column
Dim col 'temporary column
Dim position
%Check%= True
If obj.type = "LF" or obj.type = "HG" or obj.type =
"CMP" or obj.type ="HNG" then
    For Each c In obj.IndexColumns
        Set col = c.column
        If (DataTypeBase(col.datatype) = "VARCHAR") And
(col.length > 255) Then
            Output "Table " & col.parent.name & " Column
" & col.name & " : Data type is not compatible with
Index " & obj.name & " type " & obj.type
            %Check% = False
        End If
    Next
End If
End Function

```

Troubleshooting VB script errors

The scripts you have defined are executed during check model.

If errors are found in the custom check, the autofix or the global script, a message box appears to let you perform the following actions:

Button	Action
Ignore	Allows you to skip the problematic script and resume check
Ignore All	Allows you to skip all problematic scripts and resume process with standard checks
Abort	Stops check model
Debug	Stops check model, opens the resource editor and indicate on which line the problem is. You can correct error and restart check model

Running a custom check

If custom checks are defined in different resource files attached to the current model, all global sections are merged and all the functions for all custom checks are appended to build one single script. Potential errors in the Global Script sections proceeding from the different resource files will raise error messages. The user will be prompted to ignore and continue check model without custom checks, or to abort check process in order to fix scripting problems.

The Check Model Parameters dialog box displays all custom checks defined on metaclasses, stereotypes and criteria under the corresponding categories. A custom check defined on an abstract metaclass appears in all the children of this metaclass.

Defining templates and generated files in a profile

What are templates and generated files?

You now define templates and generated files in a profile, for a selected metaclass, stereotype or criterion.

The PowerDesigner Generation Template Language allows you to generate pieces of text for metaclasses. This generation can be helpful to output code (like JAVA or C++ code), reports (extract textual information from the model) or to produce external exchange format like XMI.

GTL is used to generate a file (called **generated file**) for a given metaclass and also to generate a piece of text for one metaclass that will be included in a more global file generated for another metaclass. This piece of text is called a **template**. It is made of plain text, mixed with variables that consist of information coming from the model itself.

A file is generated for each instance of the metaclass where it was defined. If you define a generated file for a stereotype or a criterion, a file will be generated for each instance with the correct stereotype or verifying the criterion.

For example, if you define a generated file in the model metaclass, then only one file will be generated per model. However, if you define a generated file in the class metaclass, a file will be generated for each class in the model.

🔗 For more information on the GTL syntax, see chapter Generation Reference Guide.

Creating a template

The Template category contains template items. You can access any information in the model using variables. This information can be simple attributes (for example the name of a class or the data type of an attribute) or a collection of objects (for example list of attributes of a class) depending on the metaclass where you have defined the template.

You can use the PowerDesigner metamodel, together with the help file pdvbs9.chm to visualize metaclass interactions. This should also help you select the appropriate metaclass where to define a template.

You use the GTL to define a template. Templates are used in the generated files: during generation, each template is evaluated and replaced by its actual value in the generated file.

 For more information on the GTL syntax, see chapter Generation Reference Guide.

Templates can be created in the Shared category when they apply to all metaclasses. They can also be created at the metaclass level or for a given stereotype or criterion.

New syntax

In the previous versions of PowerDesigner, you could bind the use of a particular template to the existence of a stereotype using the following syntax:

```
template name <<stereotype>>
```

In the current version of PowerDesigner, you create a template in a given stereotype to make sure this template is used only on metaclass instances with the stereotype.

Browse tool (F12)

You can use the Browse tool to find all templates of the same name. To do so, open a template, position the cursor on a template name in-between % characters, and click Browse (or F12). This opens a Result window that displays all templates prefixed by their metaclass name. You can double-click a template in the result window to locate its definition in the resource editor.

❖ To create a template:

- 1 Right-click a metaclass in the Profile category and select New→Template.
or
Right-click a stereotype or a criterion in a metaclass and select New→Template.
A new template is created under the corresponding category.
- 2 Type a clear name in the Name box. It is recommended not to use spaces in the template name.
- 3 <optional but very useful> Type a comment in the Comment box in order to explain the use of the template.
- 4 Type the template body using GTL in the central box.

Creating a generated file

The Generated Files category contains file entries. These entries define the files that will be generated for a given metaclass or for the instances of a metaclass with a selected stereotype or criterion.

Generated file properties

You use the GTL to define a generated file. You can define generated file entries for any metaclass, however, only the files defined for objects belonging to a model or a package collection will be generated. Sub-objects, like attributes, columns, or parameters do not support file generation, however it can be interesting to see the piece of code generated for these sub-objects in their Preview page.

You can identify the model or package collections in the PowerDesigner metamodel.

For more information on the PowerDesigner metamodel, see chapter PowerDesigner Public Metamodel.

For more information on the GTL syntax, see chapter Generation Reference Guide.

If an extended model definition complementing the generation of an object language contains a generated file name identical to a generated file name defined in the object language, then the generated file defined in the extended model definition will replace the generated file in the object language.

In the previous version of PowerDesigner, you had to define a template corresponding to the file type to generate. Now, you can directly edit the template of the file to generate in the generated file property page.

Each generated file has the following properties:

Property	Description
Name	Name of the generated file entry in the resource editor
File Name	Name of the file that will be generated, it can contain variables
Encoding	Format of the generated file, PowerDesigner supports several formats. You can use the Ellipsis button to display the Text Output Encoding Format dialog box in which you can select a format in a dropdown listbox. In this dialog box, the Abort on Character Loss check box allows you to stop generation if it causes the loss of characters
Comment	Additional information about the generated file
Use Package hierarchy as file path	Indicates that the package hierarchy should be used to generate the hierarchy of file directories
Generated file template (text zone)	Template of the file to generate. You can open an editor using the Edit With tool, and if you are using templates, you can use the Browser tool to display the definition of this template. You can use the Browse tool to find all templates of the same name (see section Creating a template)

Syntactic coloring

If the File Name box is empty, there is no file generated. However, it can be very useful because it allows you to preview the content of the file before generation. You can use the Preview page of the corresponding object at any time for this purpose.

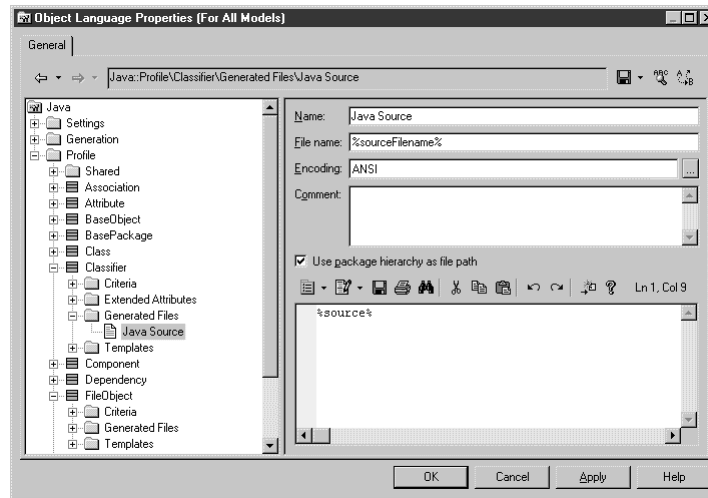
If the File Name box contains a '.' (dot) character, there is no file generated either but it allows you to use the Preview page, as explained above, with the addition of syntax color highlighting. You can add the extensions after the dot character to use the language editor of your choice (example: .dtd for XML).

❖ To define a generated file:

- 1 Right-click a metaclass in the Profile category and select New→Generated File.
or
Right-click a stereotype or a criterion in a metaclass and select New→Generated File.

A new file entry is created under the selected category.
- 2 Type a name in the Name box.
- 3 Type the template of the name for the file to be generated in the File Name box.
- 4 Select an encoding format.
- 5 <optional but very useful> Type a comment in the Comment box in order to explain what the generated file will contain.
- 6 Select the Use Package Hierarchy as File Path checkbox if you want to preserve the package hierarchy.

- 7 Type the definition of the generated file in the text zone.



- 8 Click Apply.

Using profiles: a case study

Robustness diagram

To illustrate the concept of profile, you are going to build an extended model definition for an OOM. This model extension will let you design a **robustness diagram**.

The robustness diagram is used to get across the gap between what the system has to do, and how it is actually going to accomplish this task. In the UML analysis, the robustness diagram is between use case and sequence diagram analysis. It allows you to verify that the use case is correct and that you haven't specified system behavior that is unreasonable given the sets of objects you have. The robustness diagram also enables to verify if no object is missing in the model.

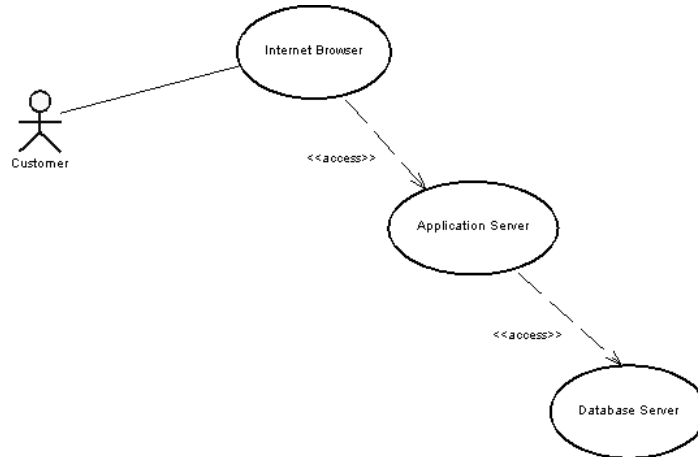
You are going to build a robustness diagram using the collaboration diagram in the OOM, together with an extended model definition containing a user-defined profile. This profile contains the following extensions:

- ◆ Stereotypes for objects
- ◆ Custom tool and symbol for each object stereotype
- ◆ Custom checks for instance links
- ◆ Generated file to output a description of messages exchanged between objects

You will follow this case study to create a new extended model definition. This extended model definition corresponds to the resource file Robustness.XEM, delivered by default and located in the \Resource Files\Extended Model Definitions folder of the PowerDesigner installation directory.

Scenario

You are going to build the robustness extended model definition starting from a concrete example. The following use case represents a very basic Web transaction: a customer wants to know the value of his stocks in order to decide to sell or not. He sends a stock value query from his Internet Browser. The query is transferred from the browser to the database server via the application server.



You are going to use the robustness diagram to verify this use case diagram.

To do so, you will use a standard collaboration diagram and extend it with a profile.

Attaching a new extended model definition to the model

In your workspace, you already have created an OOM with a use case diagram containing an actor and 3 use cases.

You have to create a new extended model definition and import it into the current model before starting the profile definition.

❖ To attach an extended model definition to the model:

- 1 Select Tools→Resources→Extended Model Definitions→Object-Oriented Model.

The List of Extended Model Definitions for an OOM appears.

- 2 Click the New tool in the list toolbar.

The New Extended Model Definition dialog box appears.

- 3 Type Robustness_Extension in the Name box.
- 4 Keep <Default Template> in the Copy From box.
- 5 Click OK.

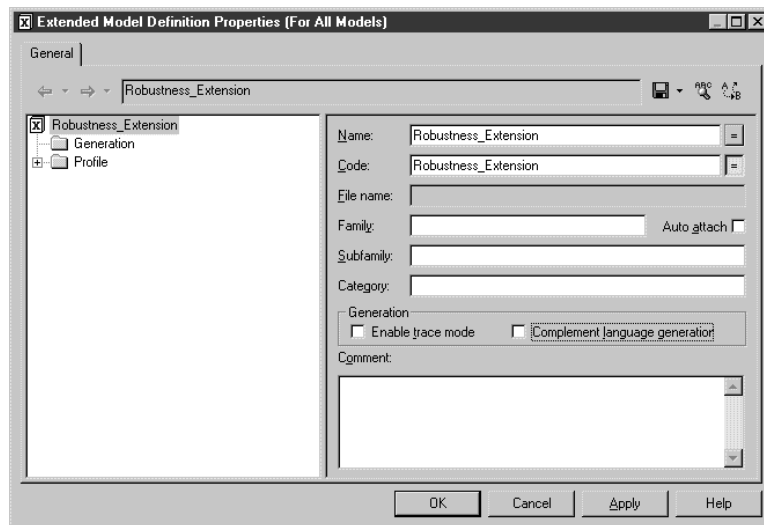
A standard Save As dialog box appears. By default the file name is identical to the name of the extended model definition.

- 6 Click Save.

The Extended Model Definition editor appears.

- 7 Clear the Complement Language Generation check box.

This extended model definition does not belong to any object language family and will not be used to complement any object language generation.



- 8 Click OK to close the Extended Model Definition editor.
- 9 Click Close in the List of Extended Model Definitions.

A Confirmation box asks you to save the extended model definition file.

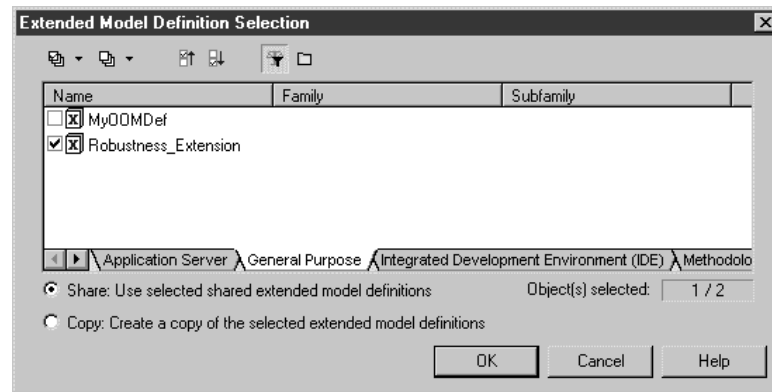
- 10 Click Yes.
- 11 Select Model→Extended Model Definitions.

The List of Extended Model Definitions appears.

- 12 Click the Import an Extended Model Definition tool in the List toolbar.

The Extended Model Definition Selection dialog box appears.

- 13 Click the General Purpose tab and select the Robustness_Extension check box.



- 14 Click OK, the extended model definition appears in the List of Extended Model Definitions.
- 15 Click OK in the list.
- 16 Right-click the model node in the Browser, and select New→Collaboration Diagram in the contextual menu.

The diagram property sheet appears.

- 17 Type Robustness Diagram in the name box and click OK in the property sheet.

Create object stereotypes

The robustness analysis classifies objects in three categories:

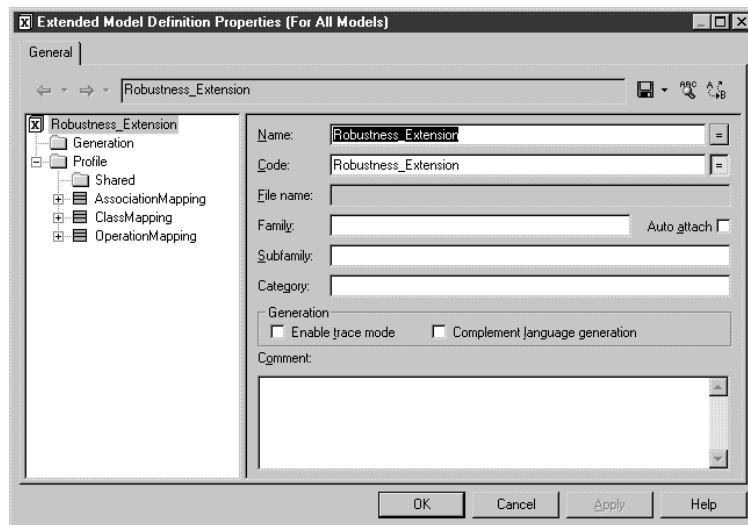
- ◆ **Boundary** objects are used by actors when communicating with the system, they can be windows, screens, dialog boxes or menus
- ◆ **Entity** objects represent stored data like a database, database tables, or any kind of transient object such as a search result
- ◆ **Control** objects are used to control boundary and entity objects, and represent transfer of information

To implement the robustness analysis in PowerDesigner, you are going to create stereotypes for objects in the collaboration diagram. These stereotypes correspond to the three object categories defined above. You will also attach custom tools in order to create a palette specially designed for creating objects with the <<Entity>>, <<Control>>, or <<Boundary>> stereotype.

You create these stereotypes in the Profile category of the extended model definition attached to your model.

❖ **To create object stereotypes:**

- 1 Select Model→Extended Model Definition and double-click the arrow beside Robustness Extension in the list to display the resource editor.

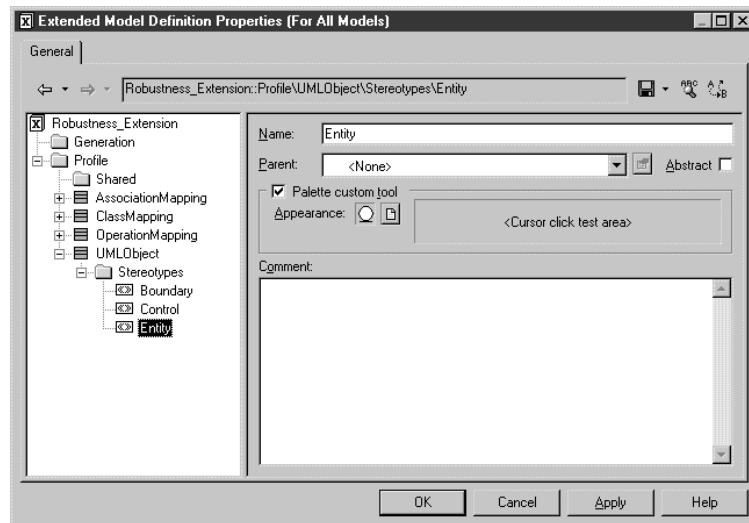


- 2 Right-click the Profile category and select Add Metaclasses.
The Metaclass Selection dialog box appears.
- 3 Select UMLObject in the list of metaclasses displayed in the PdOOM page and click OK.
The UMLObject category appears under Profile.
- 4 Right-click the UMLObject category and select New→Stereotype.
A new stereotype is created under the UMLObject category.
- 5 Type Boundary in the Name box.
- 6 Select the Palette Custom Tool check box.

- 7 Click the Browse tool and select file boundary.cur in folder \Examples\Tutorial.
- 8 Repeat steps 4 to 7 to create the following stereotypes and tools:

Stereotype	Cursor file
Entity	entity.cur
Control	control.cur

You should see the 3 stereotypes under the UMLObject metaclass category.



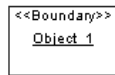
- 9 Click OK.
- 10 Click Yes to save the extended model definition.

The tool palette with the stereotype tools appears in the diagram.



- 11 Move the List of Extended Model Definitions to the bottom of the screen.
- 12 Click one of the tools and click in the collaboration diagram.

An object with the predefined stereotype is created:



- 13 Click the right mouse button to release the tool.

Define custom symbols for stereotypes

You are going to define a custom symbol for UMLObjects related to the Boundary, Control, or Entity stereotypes. The Custom Symbol feature lets you apply the standard robustness graphics into your collaboration diagram.

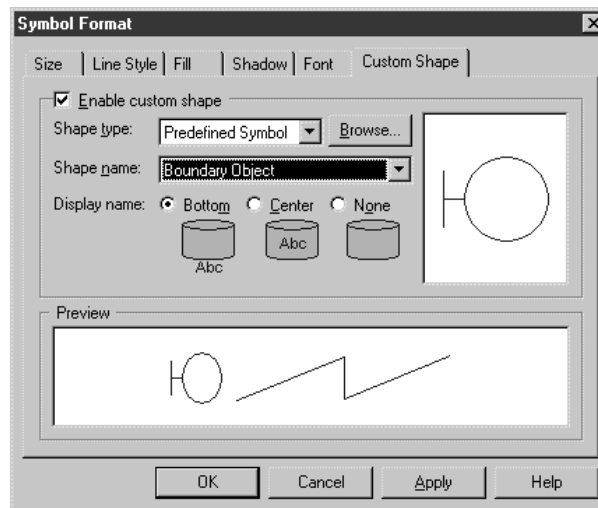
❖ To define custom stereotypes for stereotypes:

- 1 Double-click the arrow beside Robustness Extension in the list of extended model definitions to display the resource editor.
- 2 Right-click stereotype Boundary in the UMLObject category and select New→Custom Symbol.

A custom symbol is created.
- 3 Click the Modify button.

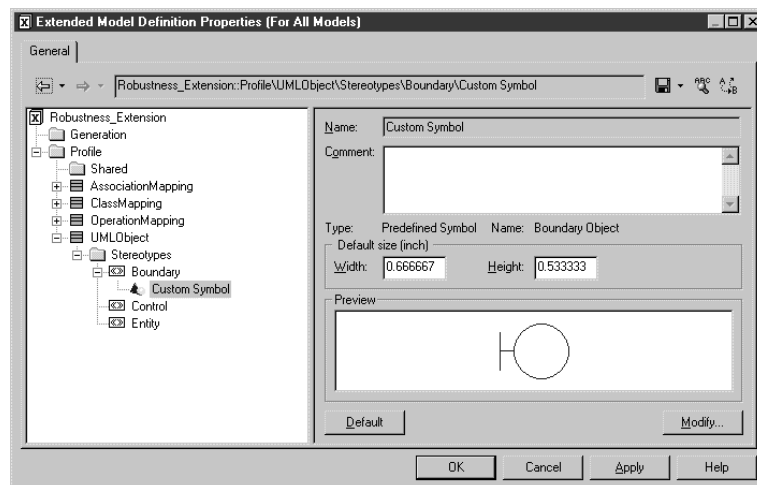
The Symbol Format dialog box appears.
- 4 Click the Custom Shape tab.
- 5 Select the Enable Custom Shape check box.
- 6 Select Predefined Symbol in the Shape Type dropdown listbox.

- 7 Select Boundary Object in the Shape Name dropdown listbox.



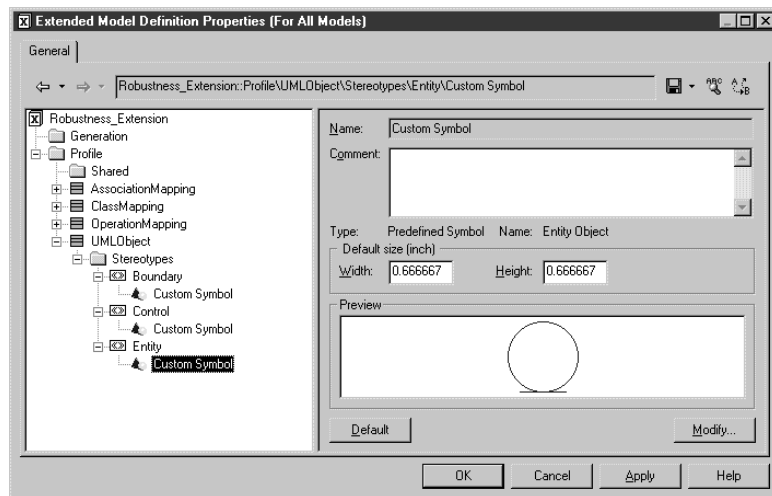
- 8 Click OK.

The custom symbol appears in the Preview box.



- 9 Repeat steps 2 to 8 for the following stereotypes:

Stereotype	Shape Name
Entity	Entity Object
Control	Control Object



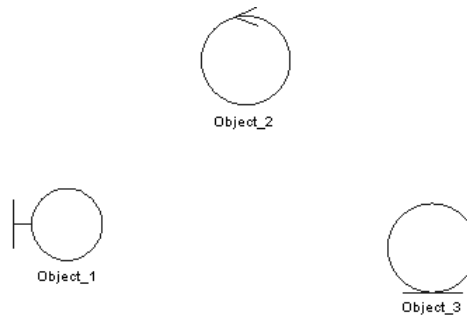
- 10 Click OK in each of the dialog boxes.

The symbol of the object you had previously created changes according to its stereotype:



- 11 In the collaboration diagram, create an object corresponding to each stereotype.

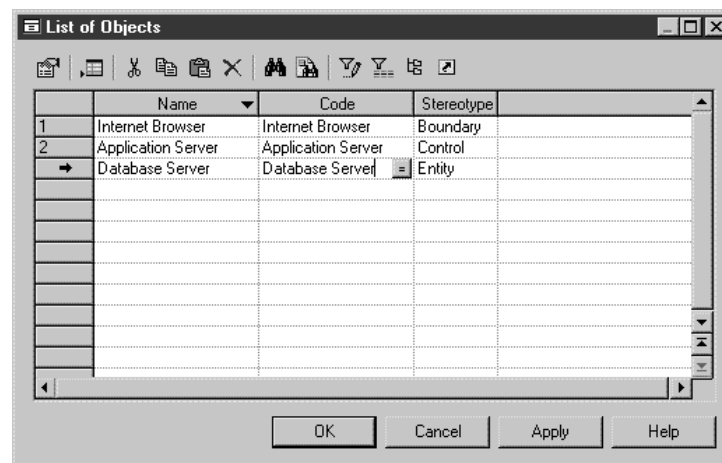
Your diagram now contains 3 objects with different symbols corresponding to different stereotypes.



- 12 Select Model→Objects to display the list of objects.
- 13 Click the Customize Columns and Filter tool in the list toolbar and select Stereotype in the list of columns.

The object stereotypes appear in the list. You are going to define the name and code of each object based on their stereotype.

Object	Stereotype	Name & Code
Object_1	<<Boundary>>	Internet Browser
Object_2	<<Control>>	Application Server
Object_3	<<Entity>>	Database Server



- 14 Click OK in the List of Objects.

- 15 Drag the actor Customer from the Browser to the collaboration diagram in order to create a symbol for Customer.

Create instance links and messages between objects

You are going to create instance links between objects to express the collaboration between objects:

Source	Destination
Customer	Internet Browser
Internet Browser	Application Server
Application Server	Database Server

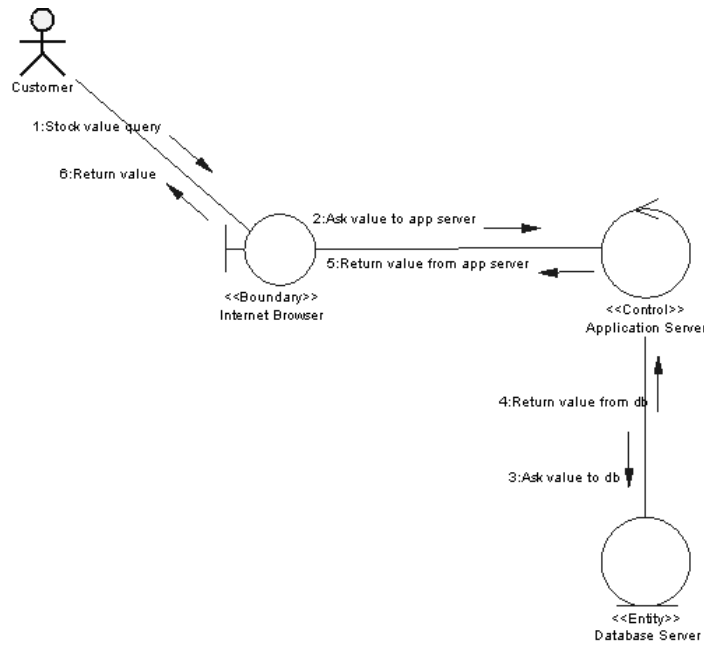
For more information on how to create instance links in the collaboration diagram, see section Creating an instance link in a collaboration diagram in chapter Building a Collaboration Diagram in the *OOM User's Guide*.

You then define messages on the different instance links to express the data carried by the links.

For more information on how to create messages on instance links in the collaboration diagram, see section Creating a message in a collaboration diagram in chapter Building a Collaboration Diagram in the *OOM User's Guide*.

You are going to create the following messages:

Direction	Message name	Sequence number
Customer - Internet Browser	Stock value query	1
Internet Browser - Application Server	Ask value to app server	2
Application Server - Database Server	Ask value to db	3
Database Server - Application Server	Return value from db	4
Application Server - Internet Browser	Return value from app server	5
Internet Browser - Customer	Return value	6



Create custom checks on instance links

The robustness analysis implies some behavioral rules between objects. For example, an actor should always communicate with a boundary object (an interface), entity objects should always communicate with control objects, and so on. To represent these constraints, you are going to define custom checks on the instance links.

Custom checks do not prevent users from performing a syntactically erroneous action, but they allow you to define rules that will be verified by the Check Model function.

You define custom checks with VB scripting.

For more information on VBS syntax, see section Accessing objects using VBScript in chapter Managing Objects in the *General Features Guide*.

❖ To create custom checks on instance links:

- 1 Double-click the arrow beside Robustness Extension in the List of Extended Model Definitions to display the resource editor.
- 2 Right-click the Profile category and select Add Metaclass.

The Metaclass Selection dialog box appears.

- 3 Select InstanceLink in the list of metaclasses displayed in the PdOOM page and click OK.

The InstanceLink category appears under Profile.

- 4 Right-click the InstanceLink category and select New→Custom Check.

A new check is created.

- 5 Type Incorrect Actor Collaboration in the Name box.

This check verifies if actors are linked to boundary objects. Linking actors to control or entity objects is not allowed in the robustness analysis.

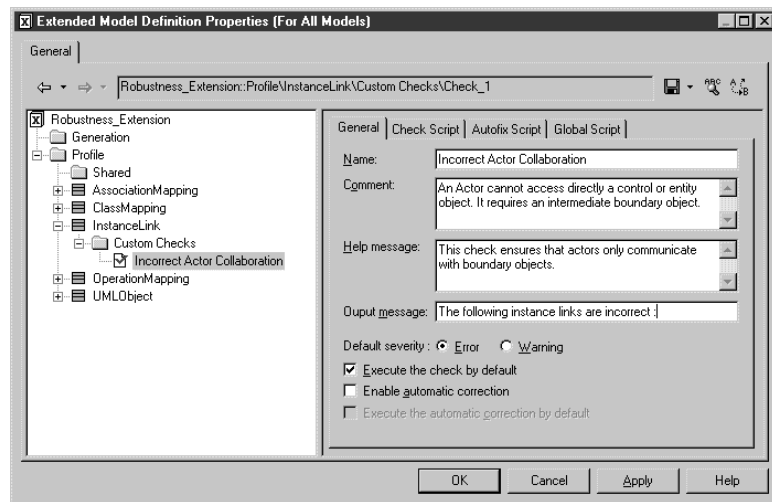
- 6 Type "This check ensures that actors only communicate with boundary objects." in the Help Message box.

This text is displayed in the message box that appears when the user selects Help in the custom check context menu in the Check Model Parameters dialog box.

- 7 Type " The following instance links are incorrect:" in the Output Message box.

- 8 Select Error as default severity.

- 9 Select the Execute the Check by Default check box.



- 10 Click the Check Script tab.

The Check Script page is the page where you type the script for the additional check.

- 11 Type the following script in the text zone.

```
Function %Check%(link)
  ' Default return is True
  %Check% = True
  ' The object must be an instance link
  If link is Nothing or not
  link.IsKindOf(PdOOM.cls_InstanceLink) then
    Exit Function
  End If
  ' Retrieve the link extremities
  Dim src, dst
  Set src = link.ObjectA
  Set dst = link.ObjectB
  ' Source is an Actor
  ' Call CompareObjectKind() global function defined
in Global Script pane
  If CompareObjectKind(src, PdOOM.Cls_Actor) Then
    ' Check if destination is an UML Object with
    "Boundary" Stereotype
    If not CompareStereotype(dst,
PdOOM.Cls_UMLObject, "Boundary") Then
      %Check% = False
    End If
  ElseIf CompareObjectKind(dst, PdOOM.Cls_Actor) Then
    ' Check if source is an UML Object with
    "Boundary" Stereotype
    If not CompareStereotype(src,
PdOOM.Cls_UMLObject, "Boundary") Then
      %Check% = False
    End If
  End If
End Function
```

- 12 Click the Global Script tab.

The Global Script page is the page where you store functions and static attributes that may be reused among different functions.

- 13 Type the following script in the text zone.

```
' This global function check if an object is of given
kind
' or is a shortcut of an object of given kind
Function CompareObjectKind(Obj, Kind)
  ' Default return is false
  CompareObjectKind = False
  ' Check object
  If Obj is Nothing Then
    Exit Function
  End If
  ' Shortcut specific case, ask to it's target object
  If Obj.IsShortcut() Then
    CompareObjectKind =
CompareObjectKind(Obj.TargetObject)
  Exit Function
```

```

    End If
    If Obj.IsKindOf(Kind) Then
        ' Correct object kind
        CompareObjectKind = True
    End If
End Function
' This global function check if an object is of given
kind
' and compare it's stereotype value
Function CompareStereotype(Obj, Kind, Value)
    ' Default return is false
    CompareStereotype = False
    ' Check object
    If Obj is Nothing or not
    Obj.HasAttribute("Stereotype") Then
        Exit Function
    End If
    ' Shortcut specific case, ask to it's target object
    If Obj.IsShortcut() Then
        CompareStereotype =
        CompareStereotype(Obj.TargetObject)
        Exit Function
    End If
    If Obj.IsKindOf(Kind) Then
        ' Correct object kind
        If Obj.Stereotype = Value Then
            ' Correct Stereotype value
            CompareStereotype = True
        End If
    End If
End Function

```

14 Click Apply.

You are going to repeat steps 4 to 14 and create one check to verify that an instance link is not defined between two boundary objects:

Check definition	Content
Name	Incorrect Boundary to Boundary Link
Help Message	This check ensures that an instance link is not defined between two boundary objects
Output Message	The following links between boundary objects are incorrect:
Default Severity	Error
Execute the check by default	Selected

15 Type the following check in the Check Script page:

```
Function %Check%(link)
```

```
' Default return is True
%Check% = True

' The object must be an instance link
If link is Nothing or not
link.IsKindOf(PdOOM.cls_InstanceLink) then
    Exit Function
End If

' Retrieve the link extremities
Dim src, dst
Set src = link.ObjectA
Set dst = link.ObjectB

' Error if both extremities are 'Boundary' objects
If CompareStereotype(src, PdOOM.cls_UMLObject,
"Boundary") Then
    If CompareStereotype(dst, PdOOM.cls_UMLObject,
"Boundary") Then
        %Check% = False
    End If
End If

End Function
```

- 16 Repeat steps 4 to 14 and create one check to verify that entity objects are accessed only from control objects:

Check definition	Content
Name	Incorrect Entity Access
Help Message	This check ensures that entity objects are accessed only from control objects
Output Message	The following links are incorrect:
Default Severity	Error
Execute the check by default	Selected

- 17 Type the following check in the Check Script page:

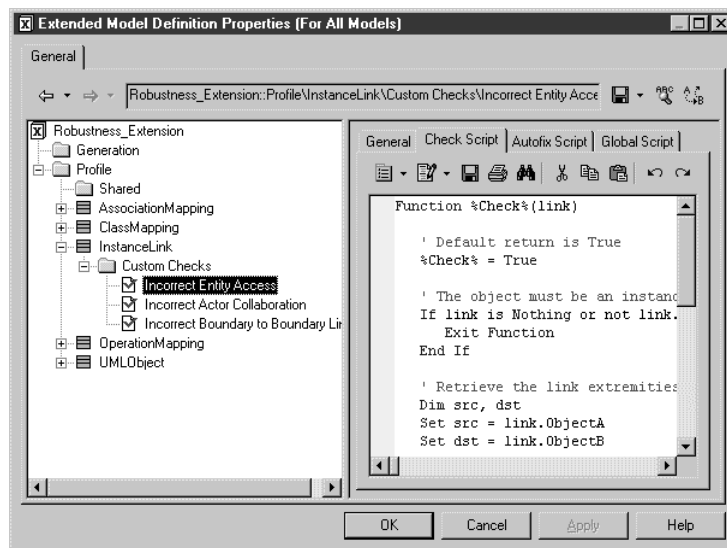
```
Function %Check%(link)
' Default return is True
%Check% = True
' The object must be an instance link
If link is Nothing or not
link.IsKindOf(PdOOM.cls_InstanceLink) then
    Exit Function
End If
' Retrieve the link extremities
Dim src, dst
Set src = link.ObjectA
```

```

        Set dst = link.ObjectB
        ' Source is and UML Object with "Entity"
        stereotype?
        ' Call CompareStereotype() global function defined
        in Global Script pane
        If CompareStereotype(src, PdOOM.Cls_UMLObject,
        "Entity") Then
            ' Check if destination is an UML Object with
            "Control" Stereotype
            If not CompareStereotype(dst,
            PdOOM.Cls_UMLObject, "Control") Then
                %Check% = False
            End If
            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

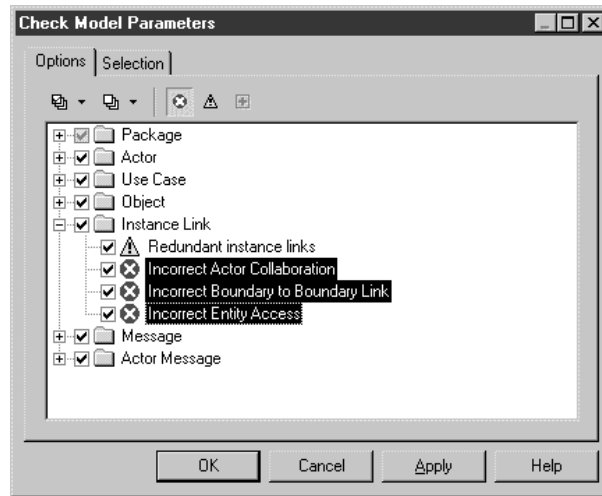
```

- 18 Click Apply.



- 19 Click OK in the resource editor.
- 20 Select Tools→Check Model to display the Check Model Parameters dialog box.

The custom checks appear in the Instance Link category.



You can test the checks by creating instance links between Customer and Application server for example, and then press F4 to start the check model feature.

For more information on the Check Model feature, see sections on checking a model in the different user's guides.

Generate a textual description of messages

You are going to generate a textual description of the messages existing in the collaboration diagram. The description should provide for each diagram in the model, the name of the message sender, the name of the message and the name of the receiver.

You generate this description using templates and generated files, because by default PowerDesigner does not provide such a feature. Templates and generated files use the PowerDesigner Generation Template Language (GTL). Generated files evaluate templates defined on metaclasses and output the evaluation result in files.

For more information on GTL, see chapter Generation Reference Guide.

What template to define and where?

To generate a textual description of the collaboration diagram message, you have to define templates on the following metaclasses:

- ◆ **Message:** this metaclass contains details about the message sequence number and the message name, sender and receiver, this is why you define a template to evaluate the message sequence number and description in this metaclass
- ◆ **CollaborationDiagram:** you define in this metaclass the templates that will sort the messages in the diagram and gather all message descriptions from the current diagram

☞ For more information on the PowerDesigner metaclasses, see the PowerDesigner metamodel and its documentation in chapter PowerDesigner Public Metamodel.

The generated file is defined on metaclass **BasePackage** in order to scan the entire model, that is to say the model itself and the packages it may contain. This is to make sure that all messages in the model and its packages are described in the generated file. There will be one generated file since metaclass BasePackage has only one instance.

Defining a template for generation

A generated file uses templates defined in metaclasses. The first template you have to define is on messages. The role of this template is to evaluate the message sequence number and to provide the name of the sender, the name of the message, and the name of the receiver for each message in the diagram.

The syntax for defining this template is the following:

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

The first part of the template aims at creating indentation according to the sequence number of the message. The macro `foreach_parts` loops on the sequence numbers:

```
.foreach_part(%SequenceNumber%, '.')
.set_value(_tabs, "  %_tabs%")
```

It browses each sequence number, and whenever a dot is found, it adds 3 empty spaces automatically for indentation. This calculates the `_tab` variable, which is later used to create the correct indentation based on the sequence numbering.

The last line contains the actual generated text of the template: for each sequence number, the appropriate tab value is created, followed by the name of the sender (short description), the text "sends message", then the name of the message, the text "to", and the name of the receiver.

❖ **To define a template used for generation:**

- 1 Select Model→Extended Model Definition and double-click the arrow beside Robustness Extension in the List of Extended Model Definitions to display the resource editor.

- 2 Right-click the Profile category and select Add Metaclasses.

The Metaclass Selection dialog box appears.

- 3 Select Message in the list of metaclasses displayed in the PdOOM page and click OK.

The Message category appears under Profile.

- 4 Right-click the Message category and select New→Template.

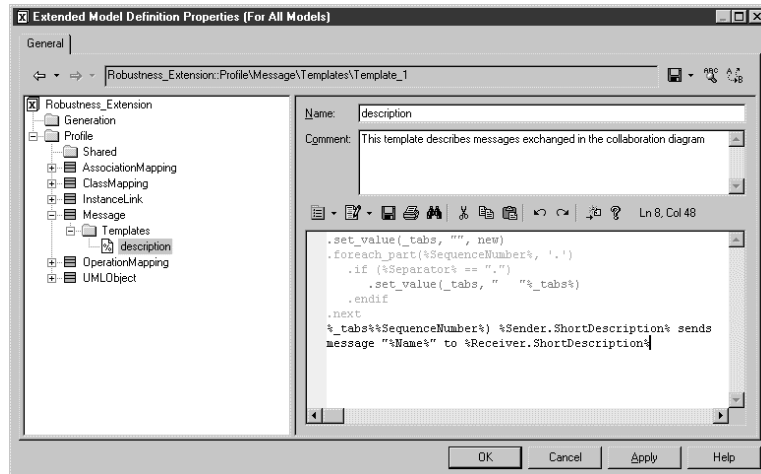
A new template is created.

- 5 Type description in the Name box of the template

- 6 <optional> Type a comment in the Comment box of the template.

- 7 Type the following code in the text area:

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

8 Click Apply.

Defining the templates for the collaboration diagram metaclass

Once you have defined the template used to evaluate each message sequence number, you have to create a template to sort these sequence numbers (**compareCbMsgSymbols**), and another template to retrieve all messages from the collaboration diagram (**description**).

The template **compareCbMsgSymbols** is a boolean that allows verifying if a message number is greater than another message number. The syntax of this template is the following:

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

The returned value for this template is used with parameter **compare** in template **description** which code is the following:

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

In this template, the first line is used to generate the title of the scenario using the name of the collaboration diagram **%Name%**. Then it creates a new line.

Then the macro for each item loops on each message symbol, verifies and sorts the message number, and outputs the message description using the syntax defined in previous section.

❖ **To define templates for the collaboration diagram metaclass:**

- 1 Right-click the Profile category and select Add Metaclasses.
The Metaclass Selection dialog box appears.
- 2 Click the Modify Metaclass Filter tool and select Show All Metaclasses.
- 3 Select CollaborationDiagram in the list of metaclasses displayed in the PdOOM page and click OK.
The CollaborationDiagram category appears under Profile.
- 4 Right-click the CollaborationDiagram category and select New→Template.
A new template is created.
- 5 Type compareCbMsgSymbols in the Name box of the template
- 6 <optional> Type a comment in the Comment box of the template.
- 7 Type the following code in the text area:

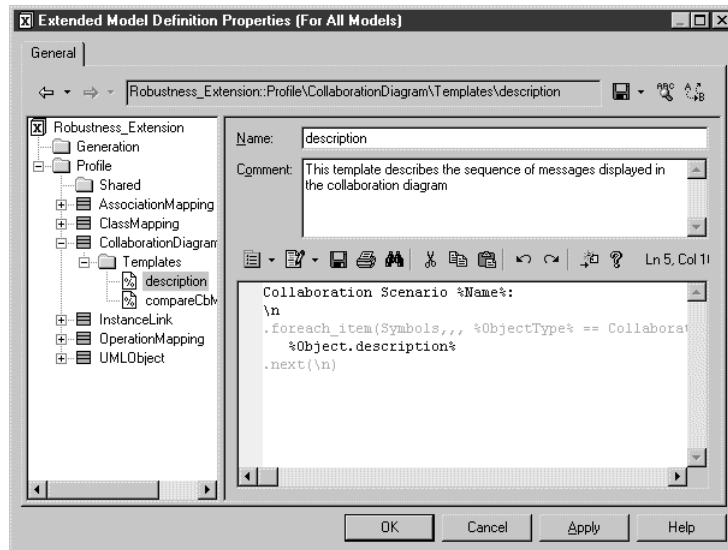
```
.bool (%Item1.Object.SequenceNumber% >=
      %Item2.Object.SequenceNumber%)
```
- 8 Click Apply.
- 9 Right-click the CollaborationDiagram category and select New→Template.
A new template is created.
- 10 Type description in the Name box of the template
- 11 <optional> Type a comment in the Comment box of the template.

- 12 Type the following code in the text area:

```

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

```



- 13 Click Apply.

Defining a generated file

You are going to define the generated file in order to list the messages of each collaboration diagram existing in your model. To do so, you have to define the generated file in the BasePackage metaclass. This metaclass is the common class for all packages and models, it owns objects, diagrams and other packages.

The generated file will contain the result of the evaluation of the template **description** defined on the CollaborationDiagram metaclass. The code of the generated file also contains a `foreach_item` macro in order to loop on the different collaboration diagrams of the model.

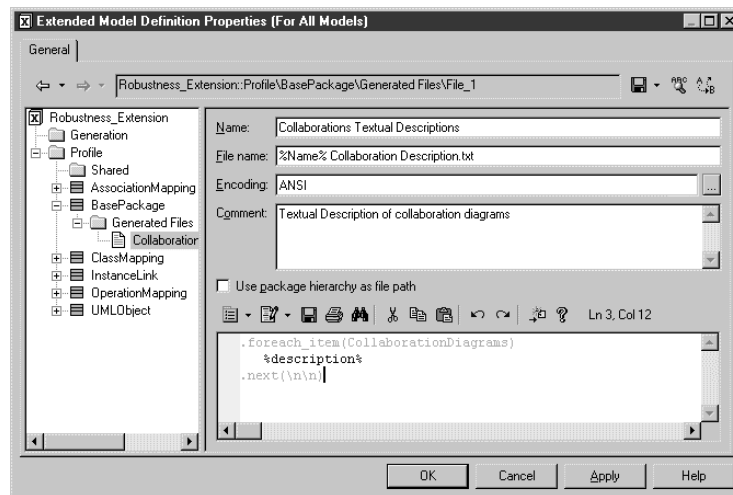
❖ To define a generated file:

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

The Metaclass Selection dialog box appears.

- 2 Click the Modify Metaclass Filter tool, select Show Abstract Modeling Metaclasses, and click the PdCommon tab.
- 3 Select BasePackage in the list of metaclasses and click OK.
The BasePackage category appears under Profile.
- 4 Right-click the BasePackage category and select New→Generated File.
A new generated file is created.
- 5 Type Collaborations Textual Descriptions in the Name box.
This name is used in the resource editor.
- 6 Type %Name% Collaboration Description.txt in the File Name box.
This is the name of the file that will be generated. It will be a .txt file, and it will contain the name of the current model thanks to variable %Name%.
- 7 Keep the Encoding value to ANSI.
- 8 <optional> Type a comment in the Comment box.
- 9 Deselect the check box Use Package Hierarchy as file path because you don't need the hierarchy of files to be generated.
- 10 Type the following code in the text box:


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



- 11 Click OK and accept to save the extended model definition.

- 12 Click OK to close the List of Extended Model Definitions.

Preview the textual description of the collaboration diagram

Since the generated file is defined in the metaclass BasePackage, you can preview the generated text in the Preview page of the model property sheet. This also allows you to verify that the syntax of the templates and the output correspond to what you want to generate.

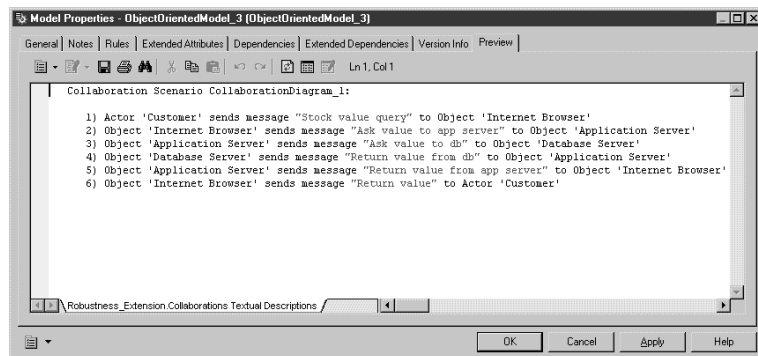
❖ To preview the textual description of the collaboration diagram:

- 1 Right-click the collaboration diagram background and select Properties.

The model property sheet appears.

- 2 Click the Preview tab to display the Preview page.

The Preview page displays the content of the file to generate.



- 3 Click OK.

Object Languages Reference Guide

About this document

This document describes the structure and contents of object languages in an OOM. This guide is a reference for working with any PowerDesigner supported object language. It provides a global overview of the use of an object language regardless of its characteristics.


Contents

Topic	Page
Object language overview	232
Understanding the object language editor	233

Before you begin

You should never modify an object language shipped with PowerDesigner. Always create new object languages from the original files. The object language is saved in a file with the XOL extension.

Modifications to an object language can change the way PowerDesigner functions, especially when generating objects. You should ensure that you thoroughly test generated objects.

 For more information on creating a new object language that is a copy of an existing object language, see section Creating a new resource file, in chapter The Resource Editor in the *General Features Guide*.

Object language overview

You can use many different object languages in an OOM. For each object language, a standard definition file is included and provides an interface between PowerDesigner and the object language so as to establish the best relationships between them.

Caution

You should never modify the object languages shipped with PowerDesigner. For each original object language you want to modify, you should create a corresponding new object language. To do so, you have to create a new object language from the List of Object Languages, available from the Resources command in the Tools menu, then define a name and select the original file in the Copy From dropdown listbox. This allows you to create a new object language that is identical to the original file apart from the name.

What is an object language?

The object language is a required component when working with Object-Oriented Models (OOM). Each object language has its own object language definition in XML format that contains specifications for a particular object language in a format understandable by an OOM. It provides with the syntax and guidelines for generating objects and implementing stereotypes, data types, scripts and constants for an object language.

Each OOM is by default attached to one object language. When you create a new OOM, you must choose an object language: you can create a new object language or use the object languages delivered with the OOM.

The definition of an object language is available from its property sheet in Tools→Resources→Object Languages. You can select and configure parameters used when defining objects or generating from an OOM.

What does an object language contain?

All object languages have the same structure made up of a number of categories. A category can contain other categories, entries, and values. These entries are parameters recognizable by an OOM.

The values for object language entries vary for each object language. Some entries may not be available if the feature does not apply to the object language.

Object language examples

Even if one particular object language is used to illustrate an example, the mechanism explained in this chapter applies to all object languages supported in an OOM.

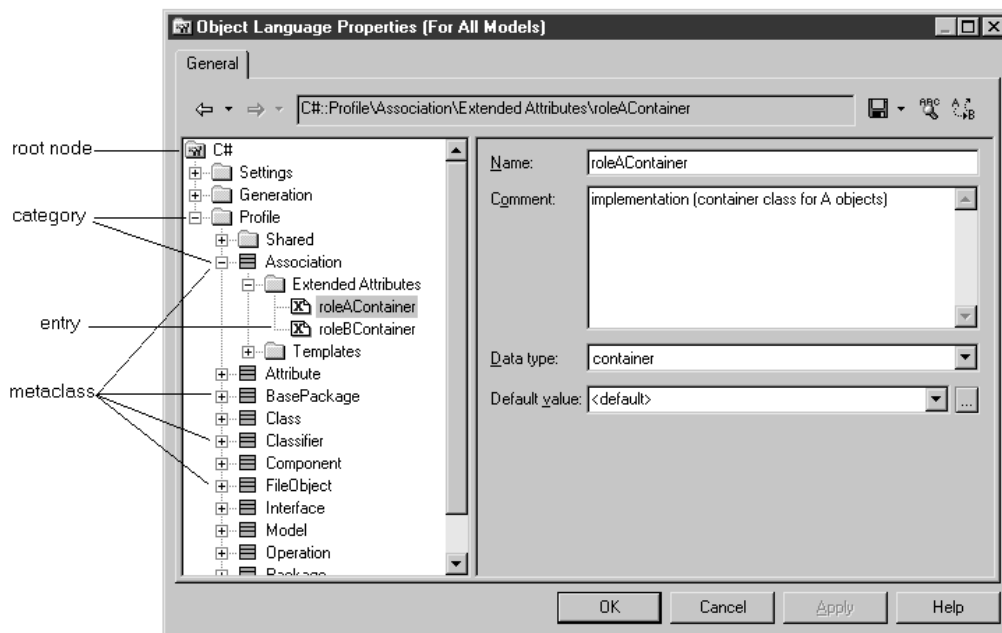
Understanding the object language editor

You use the object language editor to consult or modify an object language (.XOL file).

❖ To display the object language editor:

- ◆ Select Language→Edit Current Object Language.
or
Select Tools→Resources→Object Languages and double-click an existing object language.

The Object Language Properties page appears.



The object language editor lets you navigate through the object language categories and entries. Depending on the selected item, the right side of the page changes. The name **Metaclass** is used in the PowerDesigner documentation to identify an object as a metaclass. Metaclasses in the tree view of the resource editor are categories.

For more information on how to manipulate categories and entries in the Object Language Editor, see chapter The Resource Editor in the *General Features Guide*.

Object language properties

The object language editor contains the following categories that include properties specific to each supported object language:

Category	Description
Settings	Contains data types, constants, namings, and events categories used in the OOM
Generation	Contains generation objects with options and command scripts defined in the .XOL file
Profile	Contains templates, stereotypes, extended attributes for generation
General	Used to validate generic generation

An object language has a property page available when you click the root node in the tree view of the object language editor. It includes the following properties:

Property	Description
Name	Name of the object language
Code	Code of the object language
File Name	Path and name of the object language file. This box is filled when the object language is shared. If the object language is a copy, the box is empty. You cannot modify the content of this box
Family	Classification used to imply a certain behavior in the object model. In PowerDesigner, the family is used to enable certain features in the model, such features do not exist by default in the standard OOM. See example below
Subfamily	Sub-classification that may be combined with the family classification to imply a certain behavior in the object language
Enable Trace Mode	Displays the generation templates used for the generated text in the Preview page of object property sheets
Comment	Additional information about the object language

Family and subfamily

The Family and Subfamily properties imply certain behaviors in the object language. For example, object languages of the Java, XML, IDL and PowerBuilder families support reverse engineering.

If you select File→Reverse Engineer→Object Language, you can verify that only object languages from these families are available in the Object Language dropdown listbox in the Choose Targets dialog box.

The subfamily is used to fine tune the features defined for a given family: for example, in the Java family, the J2EE subfamily allows you to handle Enterprise Java beans or makes it possible to create servlets and JSP.

Java EJB subfamily


If you work on a Java OOM created in version 9.0, the subfamily of Java is EJB 2.0 and it supports EJB. In version 9.5, the subfamily of Java has become J2EE, it supports J2EE components (servlets, JSPs **and** EJB).

You should use the new Java object language provided with the current version to use J2EE components. If you had customized the Java object language in the previous version, you have to apply these changes manually in the new Java object language.

Enable trace mode

The Enable Trace mode check box allows you to preview which templates are used during generation. Before starting the generation, click the Preview page of the object involved in the generation to see these templates, and hit the Refresh tool to display these templates.

When you double-click on a trace line from the Preview page, the corresponding template definition is displayed in the object language editor in the Profile\Object\Templates category where it is located. The code of the template may appear with distinct colors.

 For more information on these colors, see paragraph Syntactic coloring in section Generated Files category.

Settings category

The Settings category contains items to customize and manage OOM generation features. The items that are specific to an object language are not detailed in this section.

Category	Description
Data Types	Tables for data type mappings
Constants	List of constant values
Namings	Contains getter and setter operation default values, illegal character, or J2EE component name values for example
Events	Standard event values

Data types category

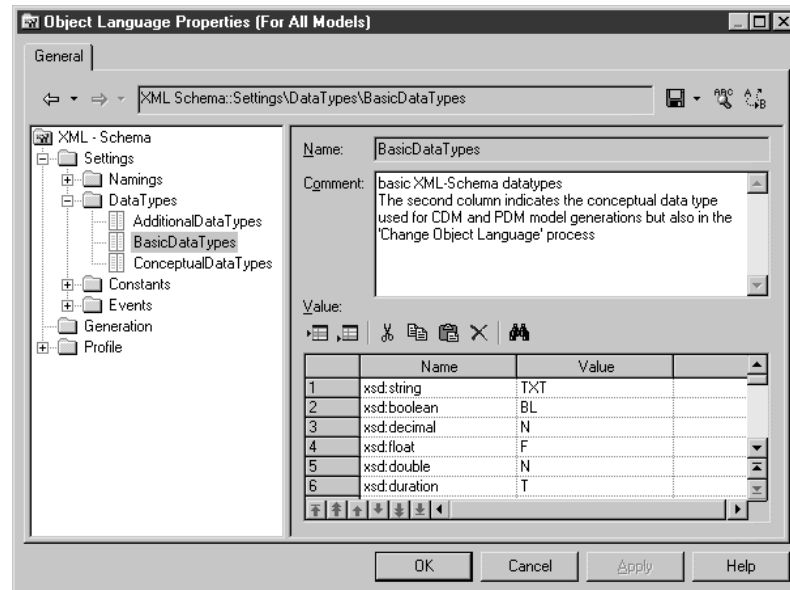
The data types category shows a mapping of internal data types with object language data types.

The following data types values are defined by default:

Constant	Default value
BasicDataTypes	Basic object language data types. The Value column indicates the conceptual data type used for CDM and PDM model generations. BasicDataTypes are the most common data types used
ConceptualDataTypes	The Value column indicates the object language data type used for CDM and PDM model generations. Conceptual data types are the internal data types of PowerDesigner, and cannot be modified
AdditionalDataTypes	Additional data types added to data type lists. The Value column indicates the conceptual data type used for CDM and PDM model generations. Can be used to add or change data types of your own
DefaultDataType	Data type value selected by default

Example

In the XML-Schema object language, the Settings category is expanded to show the list of values for Basic data types.



Constants category

The Constants category contains the following constant values defined by default:

Null
True
False
Void
Bool

This category contains a mapping between the constants and their default values. The default values vary depending on the object language.

Namings category

The Namings category contains parameters that influence what will be included in the files that you generate from an OOM. You can modify default values from this category to customize the generation process in PowerDesigner.

You can modify the following default name values:

Constant	Description
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	List of illegal characters in current object language. See example below

Illegal characters

The list of illegal characters is used when creating an OOM, or when changing the object language to initialize the lists of illegal characters for all codes of all objects of the model. The list of illegal characters is visible after creation of the model from Model Options→Naming Convention.

For example, the following characters are not accepted in XML-Schema if names or codes include them:

`"/!<>"' ' () "`

Events category

You can use this category to define standard events on operations. You may find default existing events such as constructor and destructor, depending on the object language.

An event is linked to an operation. The contents of the Events category is displayed in the Event dropdown listbox in operation property sheets. It describes the events that can be used by an operation.

In PowerBuilder for example, the Events category is used to associate operations with PowerBuilder events.

Generation category

The Generation category contains categories and entries to define and activate a generation process. You can use the following categories:

- ◆ Generation commands
- ◆ Generation tasks
- ◆ Generation options

Generation Reference Guide

The PowerDesigner generation process is defined in this manual in chapter Generation Reference Guide. It is strongly advised that you first read this chapter in order to get familiar with the concepts and features of the generation process.

Commands category

The Commands category allows you to write generation **commands**. Generation commands can be executed at the end of the generation process, after the generation of all files. For example, compile, or create an archive file are commands for generation that you can use in an OOM.

All commands written in the Commands category are available after creation in the Tasks page of the Generation dialog box where they can be executed at the end of the generation. You may find default commands in the current object language. All default commands are not listed here as they are fully specific to the object language you are using.

Commands must be included within **tasks** in order to be started.

✍ For more information on the role of tasks, see section Tasks category.

Commands are specifically tied to an object language. The text defining the code of the command is written using the Generation Template Language (GTL). It uses environment variables that may correspond to executables like `javac.exe` for example.

For more information on the GTL, see chapter Generation Reference Guide.

Specific macros are available within the frame of commands execution.

For more information on macros, see section Using macros in chapter Generation Reference Guide.

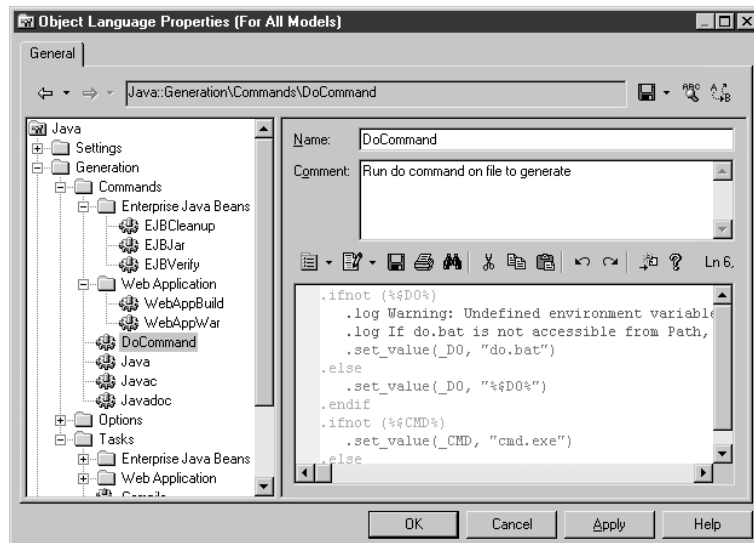
❖ To define a command for generation:

- 1 Right-click the Generation\Commands category in the object language.
- 2 Select New.

The new entry appears with a default name.

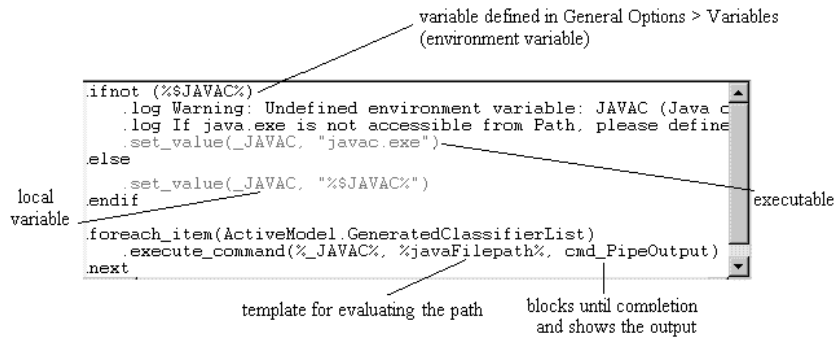
 Command_1

- 3 Type a name in the Name box.
- 4 Type a comment in the Comment box.
- 5 Type the command code in the empty text area using the different tools at your disposal.



- 6 Click Apply.

Command code explanation



Removing a generation command

You can remove a generation command in the object language by right clicking the entry and selecting Remove. A confirmation dialog box asks you to confirm the entry deletion. It may indicate that the default value set for this entry will be used, this default value is stored in the hidden template of the object language.

For more information on these entries, see section Adding a category or an entry to a resource file, in chapter The Resource Editor in the *General Features Guide*.

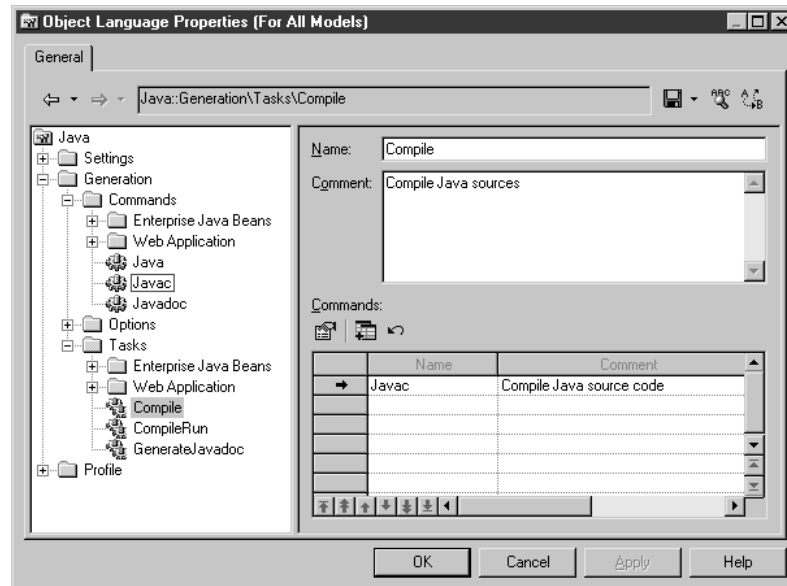
Tasks category

A task is a list of commands to be executed after generation. Commands must be referenced within the different tasks defined for generation, a command entry that is not referenced in a task entry cannot be executed. An example at the end of this section shows a command, and a task containing the command.

The mechanism of command execution is the following: when a task is selected in the Tasks page of the Generation dialog box, the commands included in the task are retrieved and the template associated with the command is evaluated and executed.

Tasks appear in the Tasks page of the Generation dialog box and are executed in a well-defined order. This order can easily be modified using the Move a value up or Move a value down arrows from the Tasks page.


You must **first** create the command before creating the task that contains the command: in the following example, the task named Compile refers to a command named Javac (written in the Name column). The code of the Javac command is located in the Commands category.



❖ **To define a task for generation:**

- 1 Right-click the Generation\Tasks category.
- 2 Select New.

The new entry appears with a default name.

 Task_1

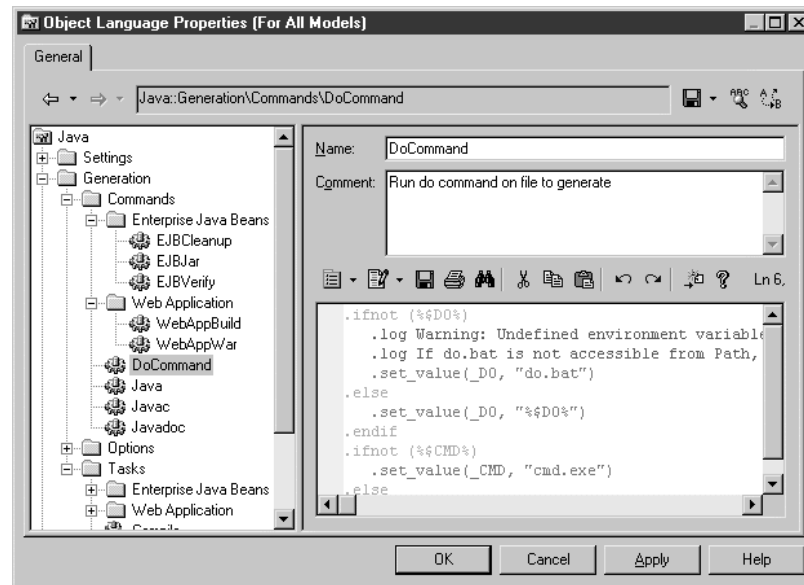
- 3 Type a name in the Name box.
- 4 Type a comment in the Comment box.
- 5 From the Commands groupbox, click the Add commands tool and select the command(s) you want to include into the current task.

The name of the command(s) appear in the Name column.

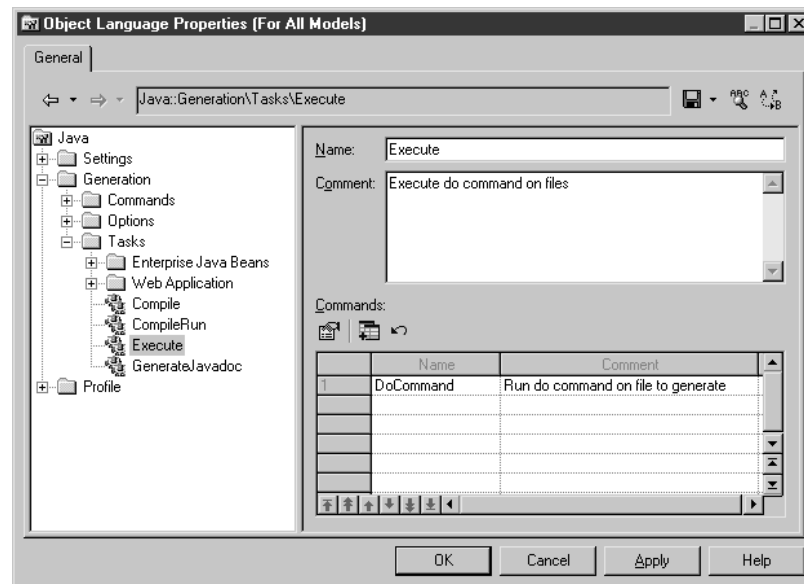
- 6 <optional> Double-click the arrow at the beginning of the line to display the code of the command.
- 7 Click Apply.

Example

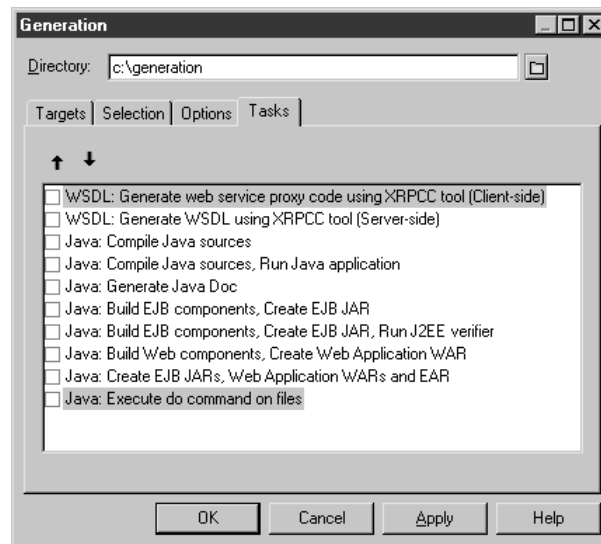
Open the Java object language from Language→Edit Current Object Language, create and define a command entry named DoCommand in the Generation\Commands category:



Create a task entry named Execute in the Generation\Tasks category, click the Add Commands tool, and select DoCommand:



The new task is available in the Commands page of the Generation dialog box. The comment of the task is displayed by default. If no comment has been provided, then the name of the task is displayed.



Options category

The Options category allows you to customize the generation process. It contains default entries that represent options for generation.

You can define generation options from the Generation\Options category in the object language. Those user-defined options are available after creation in the Options page of the Generation dialog box.

Generation options can be used in all templates.

You can create the following types of options:

Type	Description
Boolean	When selected, a Value box allows you to choose between Yes and No
String	When selected, a Value box allows you to enter the string
List	When selected, a Value area appears. You can add a value using the Add a Value tool, then check the value check box

The format of values may appear as follows in the Options page of the Generation dialog box: when a value comment is specified, it is displayed in the list instead of the value name.

For example, if a **Value:Attribute** check box is selected in a list option named myListOption from the Generation\Options category, only the **Attribute** description is displayed in the Options page of the Generation dialog box. However, the GTL will evaluate %GenOptions.MyListOption% to **Value**.


You may find default options created in the object language. They are not all listed here, as they are fully specific to the object language you are using.

☞ For more information on the list of options of your object language, see chapter Generating for a language in the *Object-Oriented Model User's Guide*.

❖ **To define an option for generation:**

- 1 Right-click the Generation\Options category in the object language.
- 2 Select New.

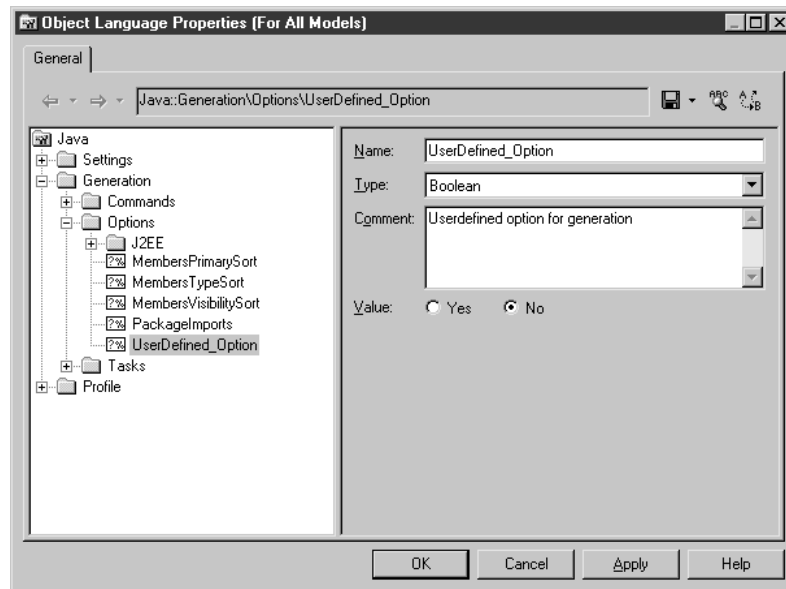
The new entry appears with a default name.

 Option_1

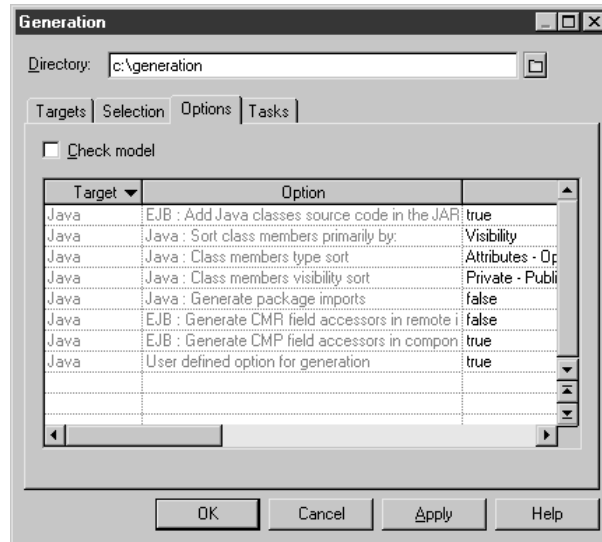
- 3 Type a name in the Name box.
- 4 Select a type.
- 5 Type a comment in the Comment box.
- 6 Depending on the option type, select or type the value in the Value area.
- 7 Click Apply.

Example

Open the object language from Language→Edit Current Object Language, and create a boolean entry named UserDefined_Option in the Generation\Options category:



The new option is available in the Options page of the Generation dialog box from Language→Generate Java Code. The comment of the option is displayed by default. If no comment has been provided, then the name of the option is displayed.



Example

The value of an option may be accessed in a template using the following syntax:

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

For example, if you define a boolean option named GenerateComment, %GenOptions.GenerateComment% will evaluate to either true or false in a template, depending on the value specified in the Options page of the Generation dialog box.

Removing a generation option


You can remove a generation option in the object language by right clicking the entry and selecting Remove. A confirmation dialog box asks you to confirm the entry deletion. It may indicate that the default value set for this entry will be used, this default value is stored in the hidden template of the object language.

For more information on these entries, see section Adding a category or an entry to a resource file, in chapter The Resource Editor in the *General Features Guide*.


Profile category

Each object has a category identified by its name that may contain the following categories:

- ◆ Criteria
- ◆ Stereotypes
- ◆ Extended attributes
- ◆ Generated Files and Templates
- ◆ Custom Check
- ◆ Custom Symbol and tool


 For more information on custom checks and custom symbols, see sections [Defining a custom check in a profile](#) and [Defining a custom symbol in a profile](#), in chapter [Managing profiles](#).

Those objects are metaclasses (of the metamodel), since each object is represented as a metaclass in the metamodel.

 For more information on the metamodel, see chapter [PowerDesigner Public Metamodel](#).

Shared category


The Profile category also includes a Shared category used to store reusable entries like extended attribute types and shared templates.

 For more information on items in the Profile category, see chapter [Managing Profiles](#).

Criteria category

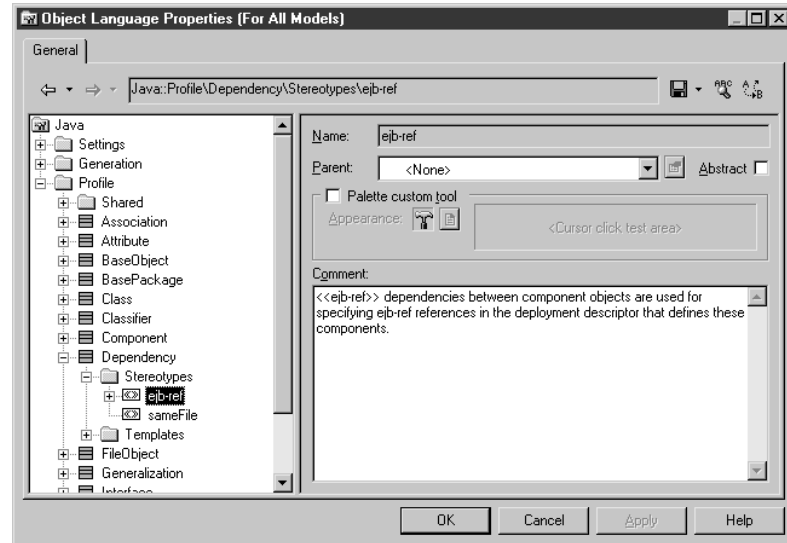
Criteria are a generic extension mechanism for the PowerDesigner metaclasses. They can be used for objects that do not support stereotypes in a CDM and a PDM.

A criterion defines a condition with the expressions used in the .if macro of the PowerDesigner Generation Template Language (GTL). When a metaclass instance verifies the criterion condition, the extensions defined on the criterion are applied to this instance.

 For more information on criteria, see section [Defining a criterion](#), in chapter [Managing Profiles](#).

Stereotypes category

In the Stereotypes category, you can define stereotypes for objects that support stereotypes. You can modify existing default stereotypes, or define new stereotypes for any object in the model.



When you modify the values of a stereotype for an object, the changes apply to all existing objects and all new objects of the same type that you create in the model.

For more information on stereotypes, see section Defining a stereotype, in chapter Managing Profiles.

Extended attributes category

The extended attributes defined for an object are created in addition to standard attributes. Extended attribute names can be used as variables in the scripts defined in the object language.

Each extended attribute may be needed during generation, this is why we advise you not to modify these extended attributes, but rather make a backup copy of each object language before you start to modify them.

By default the extended attributes values that you defined in the object language appear in the Extended Attributes page of the object property sheet.

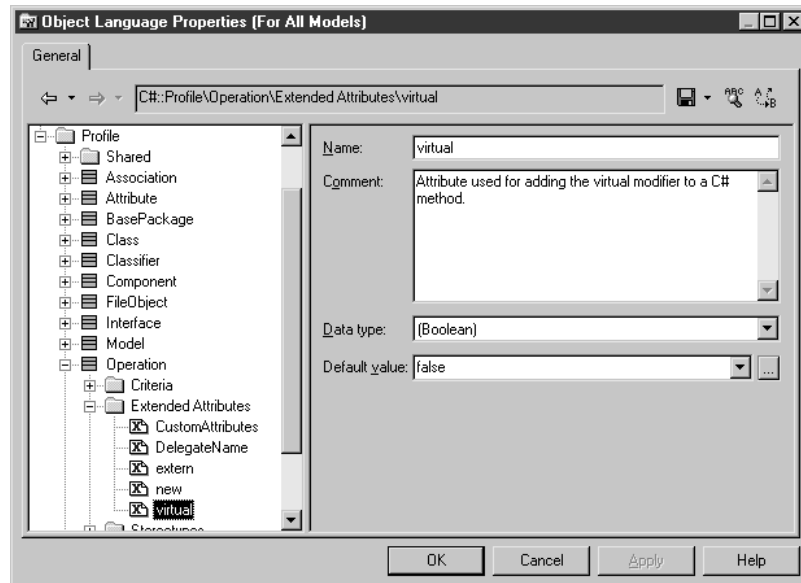
For more information on extended attributes, see section Defining extended attributes in a profile, in chapter Managing Profiles.

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

For more information on extended model definitions, see section Working with extended model definitions, in chapter Managing objects, in the *General Features Guide*.

Example

In the C# Object Language, the **virtual** extended attribute is used to identify the operation:



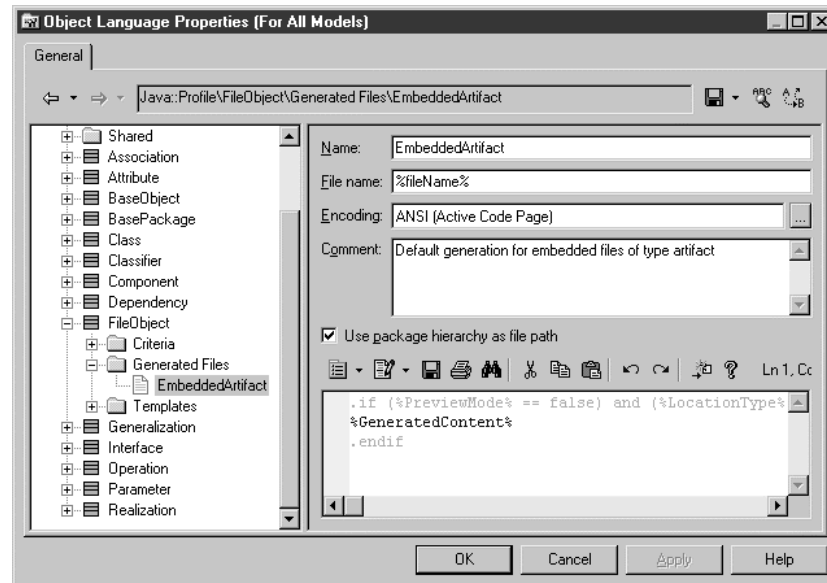
Generated Files category

Some objects in the Profile category include a Generated Files entry that contains file entries. These entries define the files that will be generated for a given metaclass or for the instances of a metaclass with a selected stereotype or criterion.

Example

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

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



For more information on the Generated Files entry, see section Defining templates and generated files in a profile, in chapter Managing profiles.

Encoding

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

This dialog box includes the following properties:

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

Syntactic coloring

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

If the File Name box contains a '.' (dot) character, there is no file generated either but it allows you to use the Preview page, as explained above, with the addition of syntax color highlighting. You can add the following extension after the dot character to use the language editor of your choice (example: .cs for C++):

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

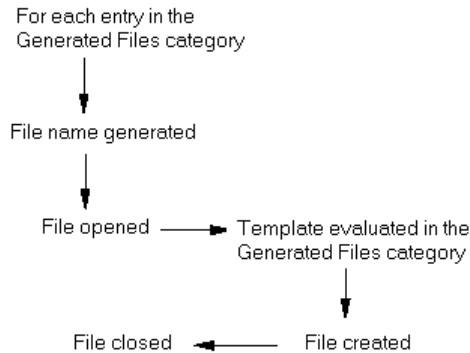
There are two possible scenario during generation:

- ◆ A file is generated

- ◆ No file is generated

File generated

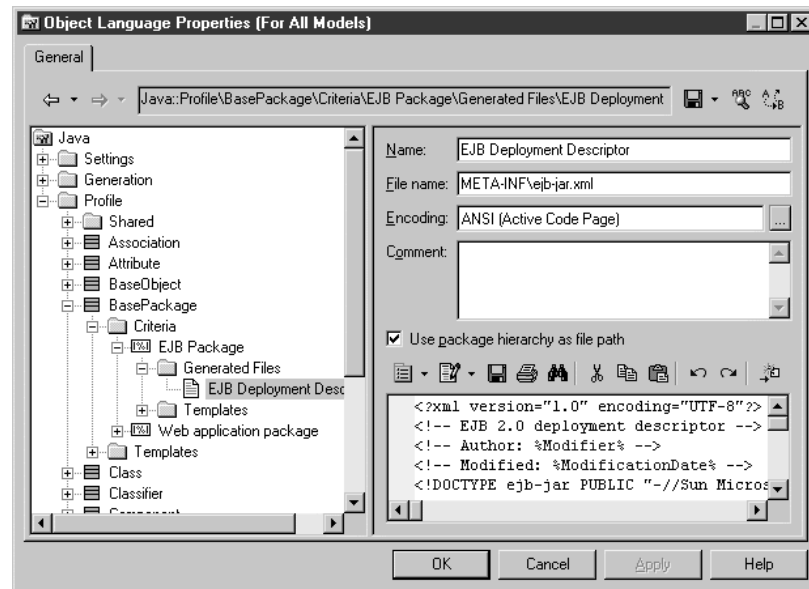
The mechanism of file generation is the following for each object having a Generated Files entry that is not empty:



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

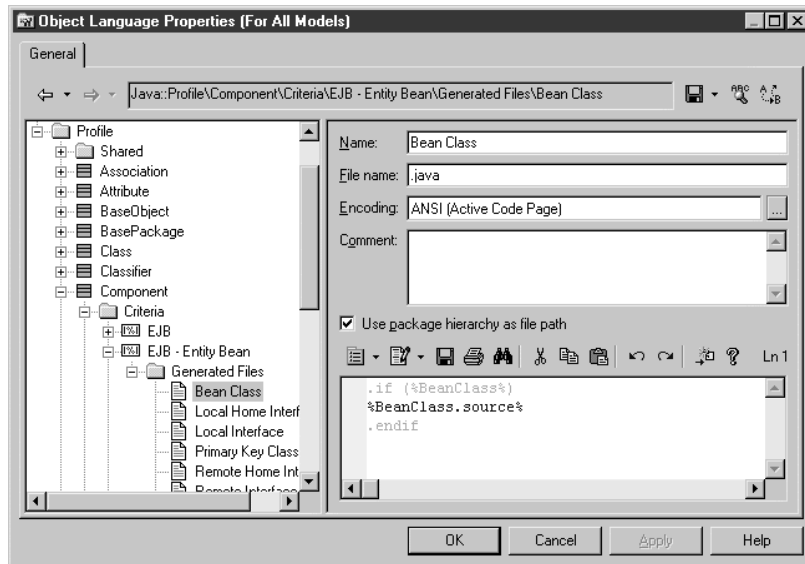
- ◆ file_name.extension (for example, ejb-jar.xml)
- ◆ %extensionfile_name% (for example, %asmxFileName%)

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



No file generated

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

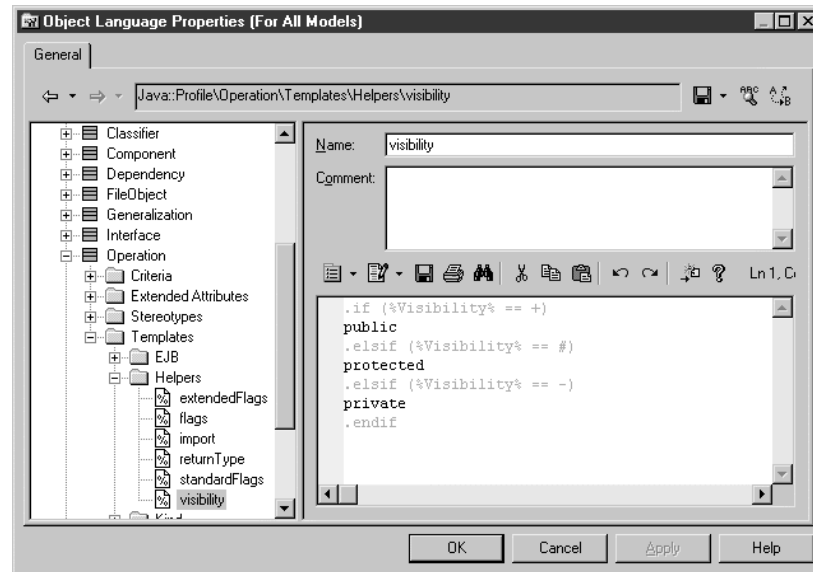
**Templates category**

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

For more information on the use of templates, see section Defining templates, in chapter Generation Reference Guide.

Example

The following template is defined for operations in Java. It sets the rules on visibility display. It is also available in the Templates category for attributes and classes.



For more information on the Templates category, see section Defining templates and generated files in a profile, in chapter Managing profiles.

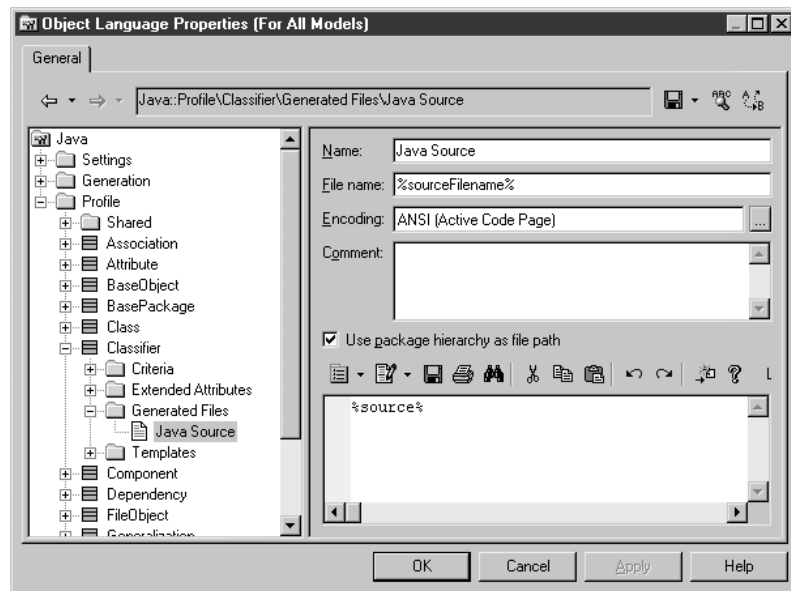
Use the F12 key to find templates

You can find all templates of the same name using the F12 key. To do so, open a template, position the cursor on a template name in-between % characters, and hit the F12 key. This opens a Browse window that displays all templates prefixed by their metaclass name.

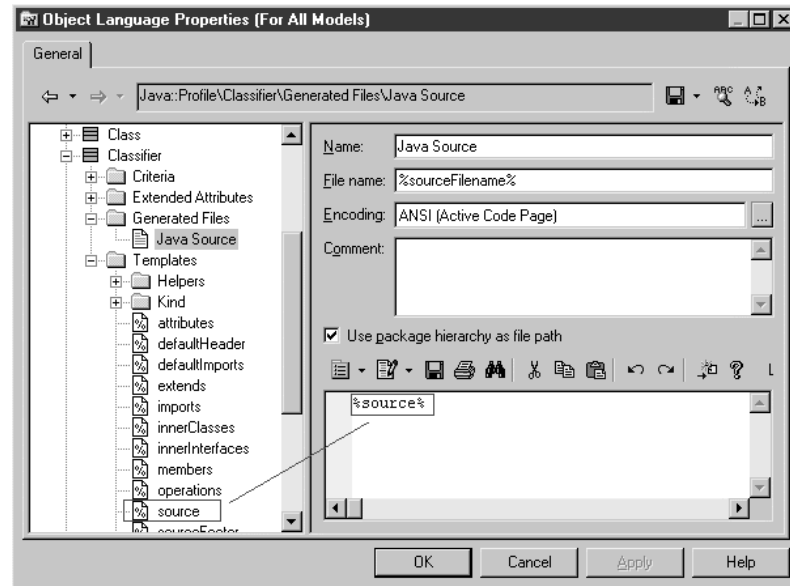
Example: position the cursor on %definition% inside a template, hit F12. The Browse window displays all <metaclass_name>::definition. You can then double-click the template of your choice in the Browse window to directly position the cursor on the selected template

How do generated files and templates work together?

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



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



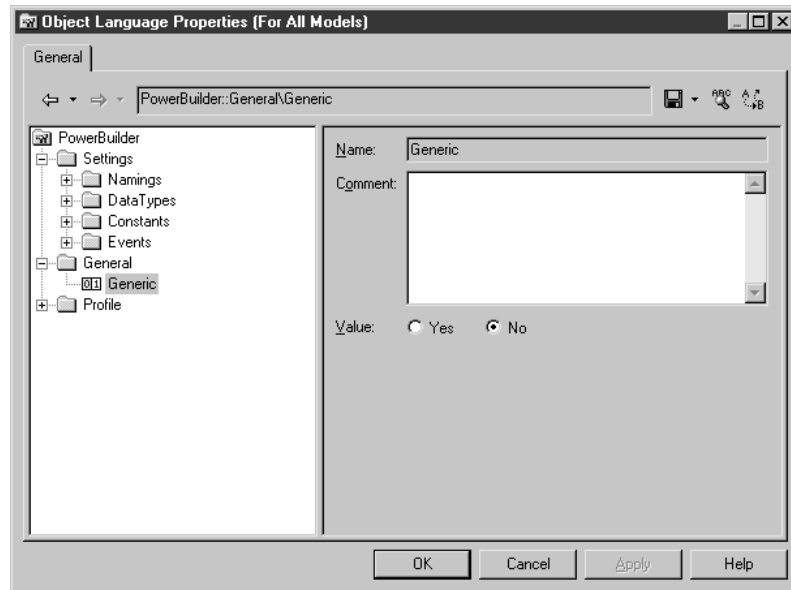
General category

The object language editor includes a General category for some object languages. The General category contains information used when you generate from an Object-Oriented Model.

The following parameters are defined by default in an OOM:

Parameter	Description
Generic	Allows you to indicate whether the object language supports generic generation. Generic generation is the principle of extending generation with the use of the GTL (Generation Template Language)

For example, the PowerBuilder object language does not support generic generation:



For more information on the GTL, see chapter Generation Reference Guide.

CHAPTER 4

Extended Model Definitions Reference Guide

About this chapter This document describes the structure and contents of a PowerDesigner extended model definition. This guide is a reference for working with any extended model definition.

Contents

Topic	Page
Managing extended model definitions	260
Working with extended model definitions	270
Generating for an extended model definition	279

Managing extended model definitions

Extended model definitions provide means for customizing and extending PowerDesigner metaclasses, parameters and generation.

Caution

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

What is an extended model definition?

✍ For more information on creating a new extended model definition from an existing extended model definition, see section Creating a new resource file in chapter The Resource Editor in the *General Features Guide*.

An extended model contains a **profile** definition and **generation** parameters. The profile is a set of metamodel extensions defined on metaclasses.

✍ For more information on profiles, see chapter Managing Profiles.

The generation parameters are used to develop or complement the default PowerDesigner object generation or for separate generation.

Extended model definitions are typed like models in PowerDesigner. You create an extended model definition for a specific type of model and you cannot share these files between heterogeneous models.

You can attach one or several extended model definitions to a model. Some extended model definitions are delivered with PowerDesigner, and you can create your own extensions. Extended model definitions are global to a model, they cannot be attached to a particular package.

✍ For more information on the generation of extended model definitions, see section Generating for an extended model definition.

What is contained in an extended model definition?

All extended model definitions have the same structure made up of a number of **categories**. A category can contain other categories, **entries**, and **values**. These entries are parameters recognizable by PowerDesigner.

The values for extended model definition categories and entries vary for each extended model definition. Some entries may not exist in the extended model definition file if they are not applicable to the particular extended model definition.

How to use
extended model
definitions?

You can create generic and specific extended model definitions.

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

Creating an extended model definition

When you create a new extended model definition you can choose to:

- ◆ Create a generic extended model definition file to reuse between models
- ◆ Create a specific extended model definition for the needs of a given model

The creation procedure differs according to the type of extended model definition you want to create.

Creating a generic extended model definition

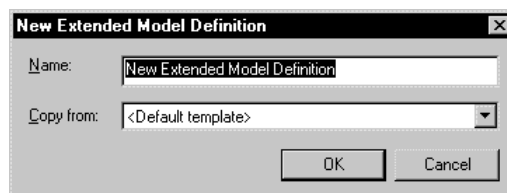
You can create generic extended model definitions to share between models of the same type.

❖ To create a generic extended model definition:

- 1 Open a model.
- 2 Select Tools→Resources→Extended Model Definitions→*Model type*.

The List of Extended Model Definitions appears.

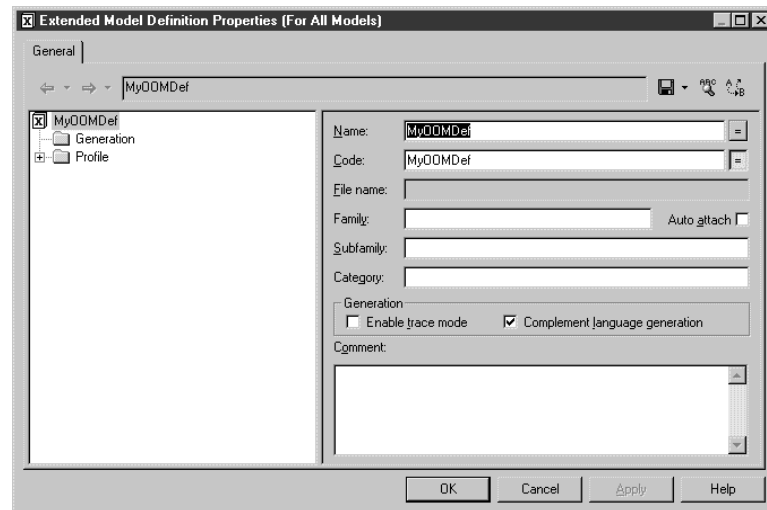
- 3 Click the New tool to display the New Extended Model Definition window.



- 4 Type a name for the new extended model definition in the Name box.

- 5 <optional> Select a template from the Copy From dropdown listbox. This dropdown listbox displays the existing extended model definitions.
- 6 Click OK.
A standard Save As dialog box appears.
- 7 Type a name and select a directory for the new extended model definition. If you do not save the extended model definition in the PowerDesigner default directory, it does not appear in the List of Extended Model Definitions. You have to click the Path tool to browse to the directory where you saved the file.
- 8 Click Save.

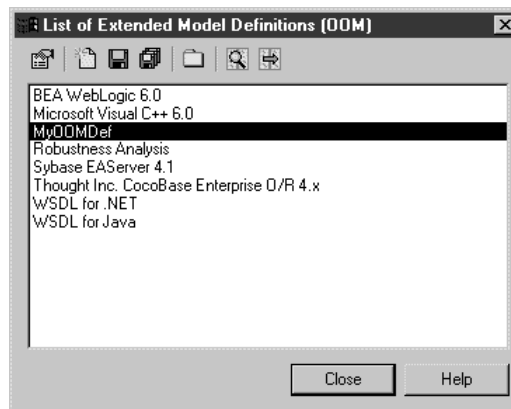
The Extended Model Definition Properties dialog box appears. The General page displays categories in a tree view together with an editor on the right side.



- 9 Define the extended model definition.
- 10 Click OK.

The extended model definition is saved in a file with the .XEM extension.


The List of Extended Model Definitions displays the new extended model definition if you saved the file in the PowerDesigner default directory.



- 11 Click Close.

Importing an extended model definition file

If you want to integrate a generic extended model definition into a model, you can import an extended model definition file.

 For more information on importing extended model definitions, see section Importing an extended model definition into a model.

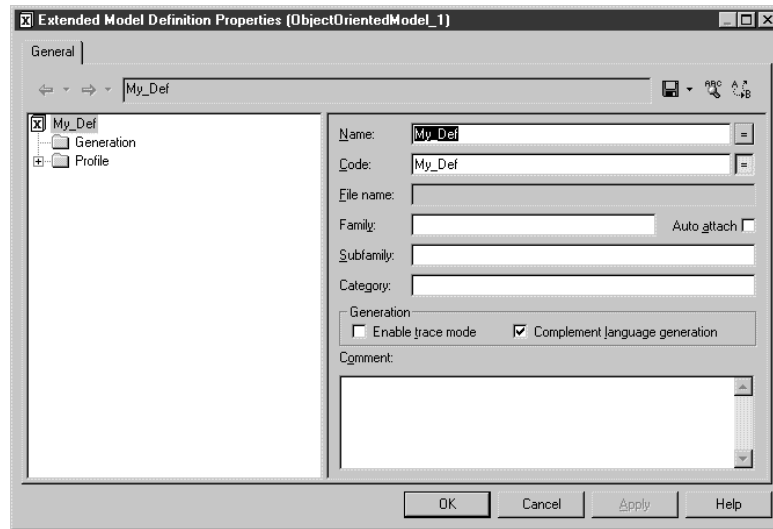
Creating a specific extended model definition for a model

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

❖ To create a specific extended model definition for a model:

- 1 Open a model.
- 2 Select Model→Extended Model Definitions.
The List of Extended Model Definitions appears.
- 3 Click the Add a Row tool to create a new extended model definition.
- 4 Click Apply.
- 5 Click the Properties tool to display the property sheet of the extended model definition.

The Extended Model Definition Properties dialog box appears.



6 Define the extended model definition.

7 Click OK.

You return to the List of Extended Model Definitions.

8 Click OK.

Exporting an extended model definition

If you want to share an extended model definition created for a model with other models, you can export this extended model definition in order to reuse it with other models.

For more information on exporting extended model definitions, see section Exporting an extended model definition.

Importing extended attributes from previous versions

You can import the extended attributes defined in a previous version into the current version of PowerDesigner. The import process transfers most extended attributes in a model or in an EXA file into an extended model definition.

You can import extended attributes using one of the following methods:

- ◆ Open a model with extended attributes that was saved with a previous version of PowerDesigner
- ◆ Open a file with the .EXA extension

Importing model extended attributes from a previous version

When you open a model saved with a PowerDesigner version 5 or 6, and this model contains extended attributes, these are automatically imported into a specific extended model definition embedded into the imported model.

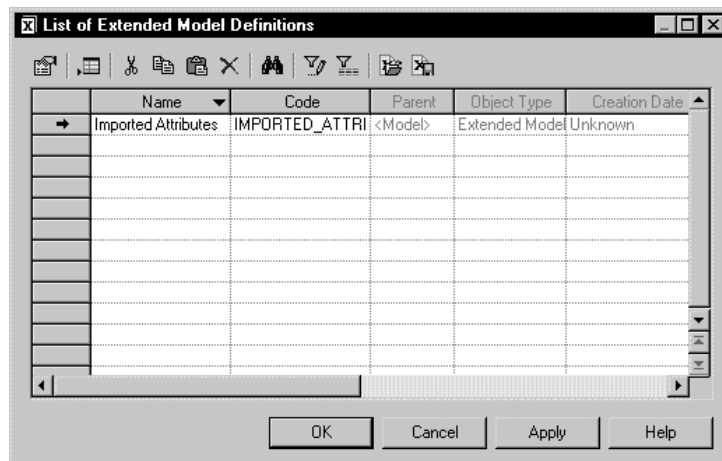
If you open a model with PowerBuilder extended attributes, the PowerBuilder extended model definition is automatically attached to the imported model.

For more information on how to import PowerBuilder extended attributes, see section Importing models with PowerBuilder extended attributes from a previous version in chapter Working with Physical Data Models in the *PDM User's Guide*.

- ❖ To import model extended attributes from a previous version:

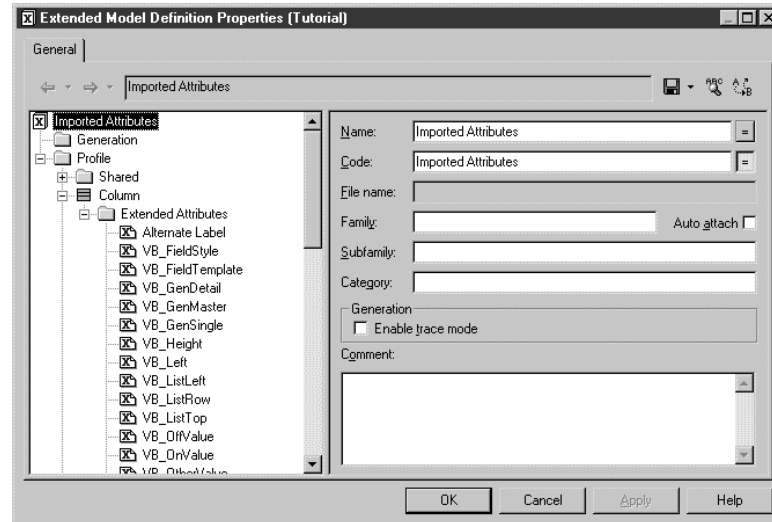
- 1 Open a model saved with a previous version of PowerDesigner.
- 2 Select Model→Extended Model Definitions.

A new extended model definition appears in the list, with the default name Imported Attributes.



- 3 Double-click the new extended model definition to display its property sheet.

You can expand the Extended Attributes category in the Profile\metaclass category and see the imported extended attributes.



Importing the extended attributes from a .EXA file

You can import an extended attribute file (with the .EXA extension) into a generic extended model definition, available for all models of the same type. This feature allows you to recover the extended attributes defined in a previous version of PowerDesigner.

❖ To import the extended attributes from a .EXA file:

- 1 Select Tools→Resources→Extended Model Definitions→Physical Data Model.

The List of Extended Model Definitions appears.

- 2 Click the Extended Attributes Import tool.

A standard Open dialog box appears.

- 3 Select an extended attribute file with the .EXA extension.

- 4 Click OK.

A standard Save As dialog box appears.

- 5 Select the name and path of the new extended model definition (with the .XEM extension) that will be created from the .EXA file.
- 6 Click OK.
The new extended model definition appears in the List of Extended Model Definitions.
- 7 <optional> Double-click the new extended model definition to display its property sheet.

Importing an extended model definition into a model

You can import a generic extended model definition file into your current model. When you import an existing extended model definition, you reuse the profile and generation parameters defined in a library available on your machine.

When you import an extended model definition, you attach it to the model. You can choose one of the following import options:

Import option	Description
Share: Use selected shared extended model definition	You use a shared version of the extended model definition. This definition is shared by all models or even all users in a design team. Any change in the extended model definition properties applies to all models that share the extended model definition
Copy: Create a copy of the selected extended model definition	You create a copy of the Extended model definition in the model. You can modify this definition in your model. Any change in the extended model definition properties applies only to the current model

Naming conventions

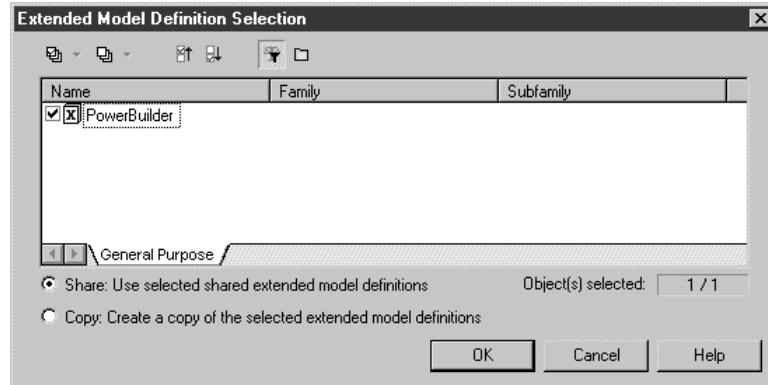
When you import an extended model definition and copy it into a model, the name and code of the extended model definition may be modified in order to respect the naming conventions of the Other Objects category in the Model Options dialog box.

❖ To import an extended model definition into a model:

- 1 Select Model→Extended Model Definitions.
The List of Extended Model Definitions appears.
- 2 Click the Import an Extended Model Definition tool.

The Extended Model Definitions Selection dialog box appears.

- 3 Use the Path tool to select the directory where the extended model definition files are stored.
- 4 Select the Share or Copy radio button.



- 5 Click OK

The imported extended model definition file appears in the List of Extended Model Definitions of the model.

Exporting an extended model definition

You can export an extended model definition created in a model if you wish to share this definition with other models. Export allows you to create a file with the .XEM extension that will be stored into your extended model definition library directory. When you export an extended model definition, the specific extended model definition remains embedded in the model.

An extended model definition created in a model does not appear in the List of Extended Model Definitions. Whereas an exported extended model definition appears in the List of Extended Model Definitions.

❖ To export an extended model definition into a model:

- 1 Select Model→Extended Model Definitions.
The List of Extended Model Definitions appears.
- 2 Select an extended model definition in the list.
- 3 Click the Export an Extended Model Definition tool.
A standard Save As dialog box appears.

- 4 Type a name and select a directory for the extended model definition.
- 5 Click Save.

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

Working with extended model definitions


Extended model definition editor

You use extended model definitions to:

- ◆ Extend the PowerDesigner metamodel and develop the definition of metaclasses using **profiles**
- ◆ Complement the generation **targets** and **commands** of an object language
- ◆ **Generate** for an extended model definition

You can consult or modify an extended model definition using the extended model definition editor.

The extended model definition editor lets you navigate through categories and entries. When you select a **category** in the extended model definition editor, the name, code, and related comment appear in the right side of the dialog box. When you select an **entry** in the extended model definition editor, the name, value, and related comment appear in the right side of the dialog box.

 For more information on how to use the extended model definition editor, see chapter The Resource Editor in the *General Features Guide*.

An extended model definition contains the following main categories:

Category	Description
Generation	Is used to define and/or activate a generation
Profile	Is used to define extensions for the PowerDesigner metaclasses in order to define stereotypes, criteria, extended attributes, generated files, generation templates, custom symbols, and custom checks

These categories are further analyzed in the following sections of this chapter.

Extended model definition properties


An extended model definition has a property page available when you click the root node in the tree view. The following properties are defined:

Property	Description
Name	Name of the extended model definition. This name must be unique in a model for generic or specific extended model definitions
Code	Code of the extended model definition. This code must be unique in a model for generic or specific extended model definitions
File Name	Path and name of the extended model definition file. This box is filled when the extended model definition is shared. If the extended model definition is a copy, the box is empty. You cannot modify the content of this box
Family	Used to classify an extended model definition. Family is designed to help establish a link between the object language of an OOM and an extended model definition. When the object language family corresponds to the extended model definition family, it suggests that the extended model definition may be used to complement the object language. For example, when an extended model definition has the family JAVA, it implies that it is designed to work with the JAVA object language. This feature is available for the OOM only
Subfamily	Used to refine the family. For example, EJB 2.0 is a sub-family of Java
Auto attach	Used to link the extended model definition to the current object language. If the Auto attach check box is selected, the corresponding extended model definition will be selected in the Extended Model Definition page of the New model dialog box
Category	Used to group concurrent extended model definitions for generation. For example, a category called "Application servers" can be created to group extended model definitions dedicated to different servers. Extended model definitions having the same category cannot be generated simultaneously. The category also corresponds to the different tabbed pages available in the Choose Targets dialog box. See section below.
Enable Trace Mode	When selected, displays the generation templates used for the generated text in the Preview page of an object property sheet

Property	Description
Complement language generation	Indicates that the extended model definition is used to complement the generation of an object language. See section below
Comment	Comments of the extended model definition

Enable trace mode

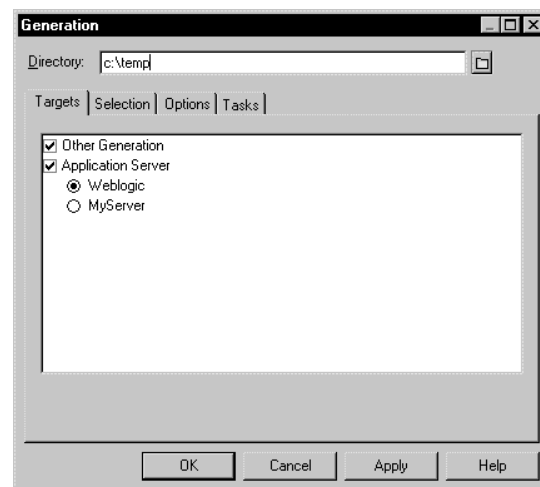
The Enable Trace mode check box allows you to preview which templates are used during generation. Before starting the generation, click the Preview page of the object involved in the generation to see these templates. When you double-click on a trace line in the Preview page, the corresponding template definition is displayed in the resource editor in the category where it is located. The code of the template may appear with distinct colors.

 For more information on syntactic coloring, see section Creating a generated file in chapter Managing Profiles.

Generation targets

The Category entry influences the layout of the Targets page of the Generation dialog box. Extended model definitions are grouped by category in this page: each extended model definition appears beside a radio button and within the category it belongs to. If you do not define a category, the extended model definition is considered as a generation target.

In the following example, the category Application Server gathers two exclusive extended model definitions; you have to select one of them. "Other generation" corresponds to an extended model definition where the Category entry is not defined.



Language generation complement

Extended model definitions can be used to extend generation or to complement the generation of an object language. In the later case, the **Complement Language Generation** check box should be selected in order to authorize the generation of the extended model definition with the object language.

PowerBuilder object language

PowerBuilder does not support extended model definitions for complementary generation.

The generation items of the object language are merged with those of the extended model definition before generation.

Generated files

All generated files defined in extended model definitions and object languages are generated. In case of generated files with identical names, the file in the extended model definition overrides the file defined in the object language.

Generation category

The Generation category contains categories and entries to define and activate a generation process. You can use the following categories:

- ◆ Generation commands, to start commands during generation
- ◆ Generation options, to define options for generation
- ◆ Generation tasks, to store commands, and to define templates for generation and O/R and R/R queries

An extended model definition can be used either to complement the generation of an object model, or for extended generation. You define this with the Complement Language Generation check box located in the extended model definition property page.

Commands and tasks

The Commands category allows you to define generation commands. The generation commands can be executed at the end of the generation, after the generation of all files.

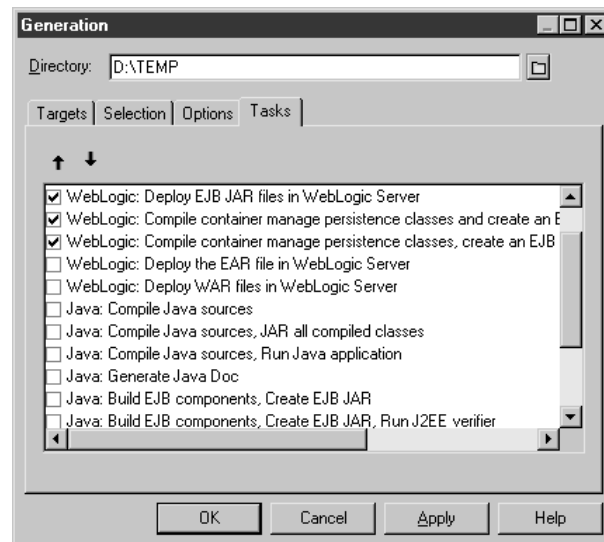
The text defining the code of the command is written using the Generation Template Language. (GTL). It uses environment variables that may correspond to executables. You can also use macros within your commands.

For more information on the GTL, see chapter Generation Reference Guide.

All commands written in the Commands category are available after creation in the **Tasks** page of the Generation dialog box where they can be executed at the end of the generation. A task is a list of commands to be executed after generation. Each task appears in the Tasks page of the Generation dialog box beside the name of the extended model definition. You can select them and use the arrows to set a generation order among these lists of commands.

A command entry that is not referenced in a task entry cannot be executed during generation. When a task is selected in the Tasks page of the Generation dialog box, the commands included in the task are retrieved and the template associated with the command is evaluated and executed.

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



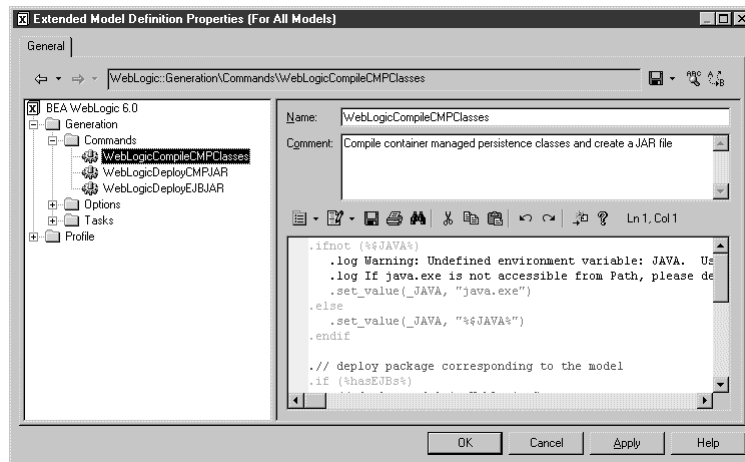
Creating a
command

You must create commands before tasks.

❖ To create a command:

- 1 Right-click the Commands category and select New in the contextual menu.
A new command is created.
- 2 Type a name in the Name box.

- 3 <optional> Type a comment in the Comment box.
- 4 Type the command code in the empty text area using the different tools available in the toolbar.



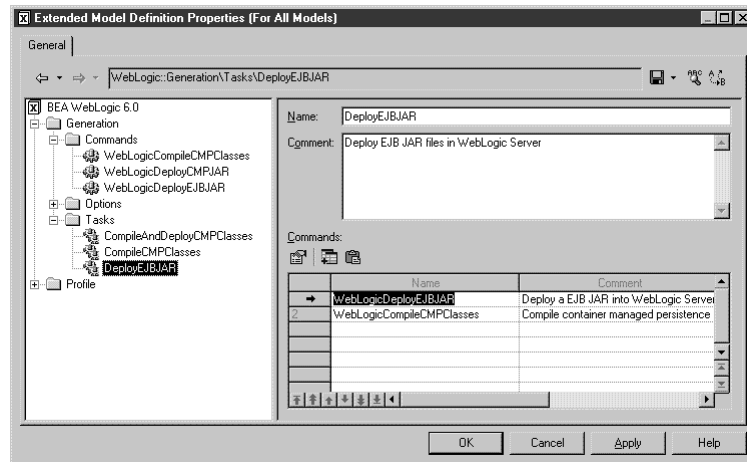
- 5 Click Apply.

Creating a task

❖ To create a task:

- 1 Right-click the Tasks category and select New in the contextual menu.
A new task is created.
- 2 Type a name in the Name box.
- 3 <optional> Type a comment in the Comment box.
The task comment appears in the Generation dialog box.
- 4 Click the Add Commands tool.
The Add Command dialog box appears.
- 5 Select one or several commands and click OK.
The commands appear in the list of commands.

- 6 Use the arrows below the list to define an order in the list of commands.



- 7 Click Apply.

Options

The Options category allows you to customize the generation process.

You can define the following types of entries in the Options category:

Entry	Definition	Example
Boolean	Option to evaluate with two possible values: Yes or No	EnableComment: Yes / No
String	Single line character value. All characters are valid in the value	WebLogicPassword value
List	List of strings which must respect the format <value>[:<label>]	WebLogicDeploymentType lets you specify if you want to deploy or re-deploy: you can choose "deploy" or "update" in the list of values

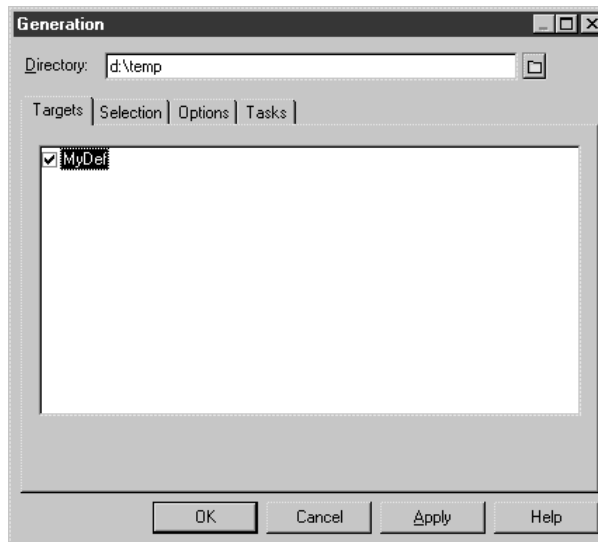
Each entry has a name and a comment, it corresponds to a generation option. The name is used if no comment is specified, otherwise the comment appears in the Option column in the Options page of the Generation dialog box.

These appear in the Options page of the Generation dialog box if the following conditions are met:

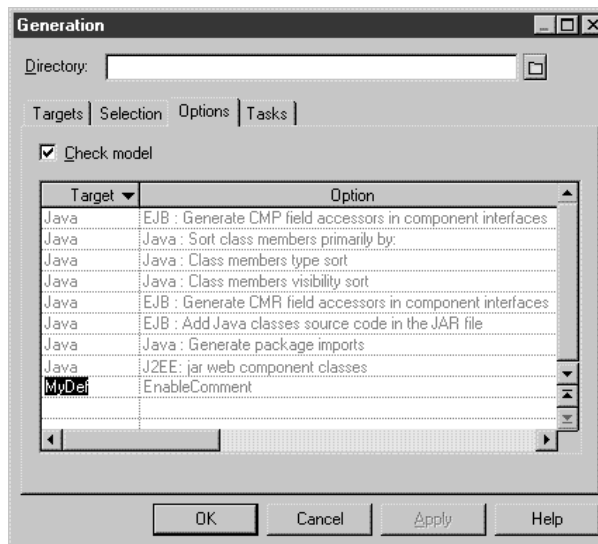
- ◆ The extended model definition contains at least one generated file

- ◆ The extended model definition contains at least one task

Both conditions allow you to display and select the extended model definition in the Targets page of the Generation dialog box:




When the extended model definition is selected, the options appear in the Options page:



You can use generation options in all generation templates and commands. To evaluate the generation option value, you have to use the following syntax: **%GenOptions.<Options name>%**.

For example, to evaluate EnableComment, you can use the following syntax:

```
.if (%GenOptions.EnableComment%)  
  %Comment%  
.endif ( )
```

 For more information on GTL syntax, see chapter Generation Reference Guide.

Generating for an extended model definition

You attach an extended model definition to a model to:

- ◆ Complement the main generation of the model
- ◆ Generate for a separate target

Complement main generation

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

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

Generate for a separate target

Extended model definitions can be used to create new generation targets. In this case, the Complement Language Generation check box should be deselected and extended model definition should contain GeneratedFiles entries.

This generation is available for all PowerDesigner modules, at any time you can generate for an extended generation using the Tools→Extended Generation command.

CHAPTER 5

Generation Reference Guide

About this chapter This chapter provides a complete reference to the concepts used in the GTL, the Generation Template Language available in PowerDesigner. It also provides some hints and tips for using the GTL.

Contents

Topic	Page
Defining Generation Template Language	282
Defining concepts used in the GTL	283
Generation tips and techniques	316

Defining Generation Template Language

It is possible to define your own generation preferences in an object language (.XOL file) or an extended model definition (.XEM file) and to customize the generation process by using the Generation Template Language (GTL).

GTL stands for Generation Template Language, it is a PowerDesigner template-based language for text generation. This language gives you flexibility and complete control over the output as generation logic is fully contained in the editable object language or extended model definition. Moreover, it allows you to easily plug-in additional code generation for your own purposes.

Process overview

GTL helps you generate pieces of text for any object (called metaclass) in a model. This generation can be helpful to generate code, generate reports (extract textual information from the model) or produce external exchange format like XMI.

You can generate one file per metaclass (for example a class in an OOM, or a table in a PDM), but you can also generate a piece of text for one metaclass that will be included in a more global file generated for another object.

This piece of text is called a template. It is made of plain text, mixed with variables that consist of information coming from the model itself.

You can access any information in the model using variables. This information can be simple attributes (for example name of a class, data type of an attribute, etc...) or collection of objects (for example list of attributes of a class). The GTL lets you also test variables, and use some macros for specific layout purpose.

The generation process evaluates which metaclass has some files to be generated and based on this, creates one file for each object in the model that belongs to this metaclass. During generation, each template and variable is evaluated and replaced by its actual value in the generated file.

Supported concepts

The Generation Template Language supports object-oriented concepts such as inheritance and polymorphism thus enabling reusability and improved maintainability. It also offers conditional, iterative and text formatting macros for specifying template logic. Macros provide generic programming structures for testing variables (example: .if) and for iterating through the dependent items of an object (example: .foreach.).

Java for examples

In this chapter, most examples proceed from the Java language. However, the mechanism explained in this section also applies to all object languages and all extended model definitions.

Defining concepts used in the GTL

The following concepts are used in the Generation Template Language.

Defining templates

A template consists of plain text, variables, and may contain macros. It is a piece of text generated for an object. It behaves like a special type property whose value is recalculated each time you try to obtain it. The content of a template is code that is generated in a text format.

A template is associated to a given metaclass, you may define as many templates as you want for any given metaclass, it is available to all objects (instances) of this metaclass. A metaclass is a class of the metamodel (or abstract class). It can be an entity attribute from the Conceptual Data Model (CDM), a table from the Physical Data Model (PDM), an operation from the Object-Oriented Model (OOM), etc ... Templates give full access to the PowerDesigner metamodel attributes, collections, and all elements of the model.

Templates are used to generate files, their role is to produce text for generation purposes. During generation, the evaluation of a template always generates text which can be stored in a file and used to gather information about a given object.

A template may also refer to any template (including itself) to promote recursion and sharing of template code, the second template is evaluated and its value is located in the result of the first template.

Example

Here is an example of a template:

```
.foreach_item (Parameters)
%definition%
.next(", ")
```

Examples

You will find specific examples on the use of templates in the different sections of this chapter.

A template is built with the following syntax:

```
template = <complex-template>
complex-template = (<block-macro> | <simple-macro> |
<simple-template>)*
```

```
simple-template = (<text> | <variable-block> |
<conditional-block>)*
```

☞ For more information on some pieces of the above syntax, see their definition later in this chapter.

You can use simple or complex templates. A simple template does not contain any macros, whereas a complex template does.

Simple template example

Here is an example of a simple template syntax:

```
%Visibility% %DataType% %Code% [= %InitialValue]
```

When evaluated, the above template which contains the four variables Visibility, DataType, Code and InitialValue will be replaced with the values for the current object from the model. Because the = sign and the value for the last variable InitialValue are in brackets, this section of the template will only be generated if InitialValue is not void.

Complex template example

Here is an example of a complex template syntax:

```

.if (%isInner% == false) and ((%Visibility% == +) or (%Visibility% == *))
[%sourceHeader%\n\n]\
[%Parent.package%\n\n]
.unique
%imports%
.endunique(\n)
[%definition%\n\n]
.foreach_item(ChildDependencies)
[%isSameFile%?%InfluentObject.definition%\n\n]
.next
[%sourceFooter%\n]
.endif

```

Variable block

Macro

Each variable enclosed in % characters is either the name of an attribute, or a property of the current object, or the name of a template.

🔗 For more information on variables, see [Defining variables](#).

Java templates example

In Java, open the Profile\Operation\Templates category in the object language to display all templates used for generation for existing operations in the OOM.

Defining variables

Variables are qualified values enclosed in % characters and optionally preceded by formatting options. At evaluation-time, they are substituted by their corresponding value in the active translation scope.

A variable can be of the following types:

- ◆ An attribute of an object
- ◆ A member of a collection
- ◆ A template
- ◆ An environment variable

For example, the variable %Name% of an interface can be directly evaluated by a macro and replaced by the name of the interface in the generated file.

Case sensitivity

Be careful when using variable names as they are case sensitive. The variable name must have the first letter with an upper case, as in %Code%.

Variables syntax

The following variables are shown with their possible syntaxes:

Variable	Syntax
variable-block	'%' ['<formatting-options> '] <variable> '%'
variable	[<outer-scope> '.'] [<variable-object> '.'] [<object-scope> '.']<object-member>
	[<outer-scope> '.'] [<variable-object> '.'] [<collection-scope> '.']<collection-member>
	[<outer-scope> '.']<local-variable>
	[<outer-scope> '.'] <global-variable>
object-member	<property>
	[<target-code> '::'] [<metaclass-name> '::'] <template-name>['('<parameter-list>')']
	[<target-code> '::']<extended-attribute>
	[' * ']+ <local-value> ['('<parameter-list>')']
collection-member	'First'
	'IsEmpty'
	'Count'
local-variable	<local-object>
	[' * '] <local-value>
global-variable	<global-object>
	<global-value>

Variable	Syntax
	'\$' <environment variable>
variable-object	<global-object> <local-object>
outer-scope	[<outer-scope> '.' 'Outer'
object-scope	[<object-scope> '.'] <object-member-object> <collection-scope> '.' <collection-member-object>
object-member-object	<objecttype-property>
collection-member-object	'First'
collection-scope	[<object-scope> '.'] <collection>

Object members

An object member can be a standard property, a template or an extended attribute. There can be three types of standard property: boolean, string or object. The value of a standard property can be:

- ◆ 'true' or 'false' if it is of boolean type
- ◆ 'null' or 'nonnull' if it is of object type

The value of a template is the result of its translation (note that a template may be defined in terms of itself, that is to say recursively).

The value of an extended attribute may itself be a template, in which case it is translated. This allows for the definition of templates on a per object (instance) basis instead of a per metaclass basis.

To avoid name collisions when a template evaluation spans multiple targets, one may prefix both extended attributes and templates by their parent target code. For example: **%Java::strictfp%** or **%C++::definition%**

Template names may also be prefixed by their parent metaclass name. This allows for the invocation of an overridden template, actually bypassing the standard dynamic template resolution mechanism. For example :

%Classifier::definition%

A parameter list can optionally be specified. Parameter values should not contain any % characters and should be separated by commas. Parameters are passed as local variables @1, @2, @3... defined in the template's translation scope.

If the template MyTemplate is defined as:

```
Parameter1 = %@1%
```

```
Parameter2 = %@2%
```

Then the evaluation of %MyTemplate(MyParam1, MyParam2)% will yield:

```
Parameter1 = MyParam1
```

```
Parameter2 = MyParam2
```

Dereferencing operator

The syntax ['*']+ <local-value> ['('<parameter-list>')'] returns the object member defined by the evaluation of ['*']+ <local-value>. If the given object member happens to be a template, a parameter list may be specified. Applying the star operator corresponds to a double evaluation (the * operator acts as a dereferencing operator).

Suppose a local variable is defined as: **.set_value(C, Code)**

Then %C% will return "Code" and %*C% will return the result of the evaluation of %Code%. In other words, %*C% can be thought of as %(%C)% (the latter syntax being invalid).

Collection members

The available collection members are:


Name	Type	Description
First	Object	Returns the first element of the collection
IsEmpty	Boolean	Used to test whether a collection is empty or not. True if the collection is empty, false otherwise
Count	Integer	Number of elements in the collection

Note: Count is particularly useful for defining criteria based on collection size, for example (Attributes.Count>=10).

Local variables

Local variables are only visible in the scope where they are defined and inside its inner scopes.

Local variables may be defined through the use of the set_object and set_value macros.

 For more information on the set_object and set_value macros, see sections set_object macro, and set_value macro.

Dereferencing operator

Variables defined through the `set_object` macro are referred to as local objects, whereas those defined with the `set_value` macro are called local values. The `*` dereferencing operator may be applied to local values.

The `*` operator allows for the evaluation of the variable whose name is the value of the specified local variable.

```
'%[' '.' <formatting-options> ':' ] '*' <local-
variable> '%'
```

For example, the following code:

```
.set_value(i, Code)

%*i%
```

Is equivalent to:

```
%Code%
```

Global variables

Global variables are available regardless of the current scope. A number of GTL-specific variables are defined as global as listed in the following table:

Name	Type	Description
ActiveModel	Object	Active model
GenOptions	struct	Gives access to the user-defined generation options
PreviewMode	boolean	True if in Preview mode, false if in File Generation mode
CurrentDate	String	Current system date and time formatted using local settings
CurrentUser	String	Current user login
NewUUID	String	Returns a new UUID (format: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX)

Outer scope

An outer scope may be accessed using the `Outer` keyword. Specifying a scope effectively changes the current translation scope used for evaluating the variable.

🔗 For more information and examples on the outer scope, see the figures in section Defining translation scope.

Object scope

To gain access to the members of an object that is not active in the current translation scope, one may specify an object scope. Specifying an object scope changes the current translation scope used to evaluate the variable.

The '.' (dot) character behaves as an indirection operator as in the Java programming language, the right-hand side corresponds to a member of the object referred to by the left-hand side.

Collection scope

To gain access to the members of a collection, one should specify a collection scope. The '.' (dot) character behaves as an indirection operator as in the Java programming language, the right-hand side corresponds to a member of the collection referred to by the left-hand side.

For example:

```
%Table.Columns.First.DataType%
```

The diagram illustrates the scope resolution for the expression `%Table.Columns.First.DataType%`. Brackets are used to group parts of the expression: a bracket under 'Table' is labeled 'object scope'; a bracket under 'Columns' is labeled 'collection scope'; a bracket under 'First' is labeled 'collection member'; and a bracket under 'DataType' is labeled 'object member'.

Defining variable formatting options

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

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

You embed formatting options in variable syntax as follows:

```
%.format:variable%
```

The variable formatting options are the following:

Format option	Description
<i>n</i> (where <i>n</i> is an integer)	Extracts the first <i>n</i> characters. Blanks or zeros added to the left to fill the width and justify the output to the right
<i>-n</i>	Extracts the last <i>n</i> characters. Blanks or zeros added to the right to fill the width and justify the output to the left
L	Converts to lowercase characters

Format option	Description
U	Converts to uppercase characters
D	Displays the interface value of an object property when this property is stored with a different name in the code line. In the following example, the value for visibility will be 'public', whereas this value is stored as '+' in the OOM. % Visibility% = + %.D:Visibility% = public
F	Combined with L and U, applies conversion on the first character
T	Leading and trailing white space trimmed from the variable
q	Enquotes the variable with single quotes
Q	Enquotes the variable with double quotes
X	Escapes XML forbidden characters

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

Defining translation scope

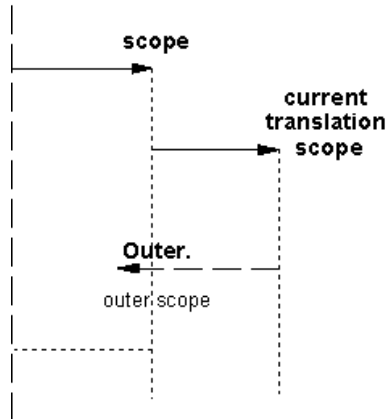
Templates are instantiated through a process called translation. Translation is done in a well-defined scope and consists in substituting the variables by their corresponding values in the scope. All templates are defined with respect to metamodel classes, they are considered special type properties that take on the result of their translation as value.

A scope is a context for evaluating a template, it comprises the active object and local variables. At any given point during translation, only one object is active; it is the object which the template is applied onto. In a translation scope, all metamodel attributes and collections defined on the active object's metaclass and its parents are visible, as well as the corresponding extended attributes and templates. When the translation engine starts evaluating a template, a new translation scope is created.

✍ For more information on object collections, see section Associations and collections, in chapter PowerDesigner Public Metamodel.

Whenever a scope is created it is the current translation scope, the old translation scope being the outer scope for the current translation scope

Whenever a scope is exited its outer scope is restored as the current translation scope



For more information on the outer scope, see section Defining variables.

New scopes may be created during evaluation of a template that forces the active object to change. For example, the **foreach_item** macro that allows for iteration on collections defines a new scope, as well as the **foreach_line** macro. The outer scope is restored when leaving the block.

For more information on both macros, see section **foreach_item** macro, and section **foreach_line** macro.

Nested scopes form a hierarchy that can be viewed as a tree, the top level scope being the root.

Example

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

```

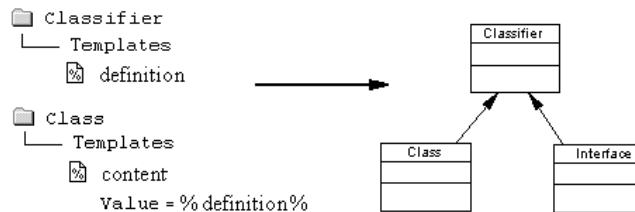
%Code%                ← class code
    .foreach_item(Operations)
%Code%                ← i-th operation code
%Outer.Code%          ← class code
    .foreach_item(Parameters)
%Code%                ← i-th parameter code
%Outer.Code%          ← i-th operation code
%Outer.Outer.Code%    ← class code
    .next
    .next
    
```

Defining inheritance

Templates are defined with respect to a given metamodel class and are inherited by its children. They are inherited by all the children of the metaclass they are defined for. This mechanism is useful for sharing template code between metaclasses having a common parent.

Example

Example of inheritance: Classifier/definition is available through inheritance of Class and Interface. In the following example, the definition template defined on the parent metaclass is used in the evaluation of the content template on the child metaclass.



Defining template overriding

Overriding a template means that a template defined with respect to a given metaclass may be redefined on a child class, in this case the template of the parent metaclass is said to be overridden. The new definition effectively hides the previous one for objects of the child metaclass.

In order to use the definition of a specific parent metaclass, GTL provides the "::" qualifying operator:

Example

```

Profile
  Classifier
    Templates
      isAbstract
        Value = false
  Class
    Templates
      isAbstract
        Value = true

```

The same template name "isAbstract" is used in two different categories: Classifier and Class. "false" is the original value that has just been overridden by the new "true" value. You retrieve the original value back by using the following syntax: <metaclassName::template>, in this case:

```

%isAbstract%
%Classifier::isAbstract%

```

Defining polymorphism

Templates are dynamically bound. In other words, the choice of the template to be evaluated is made at translation-time based on the type of object it is applied to. This mechanism is comparable to the virtual method invocation mechanism found in object-oriented languages.

Polymorphism is achieved by allowing template redefinition in derived classes. For instance, it is useful in Java to define Generated Files on the classifier. Polymorphism allows template code defined on a classifier to use templates defined on its children (class, interface), the template being used does not have to be defined on the parent metaclass. Coupled with inheritance, this feature helps you share template code.

Example

```

Classifier
  source
    Value = %definition%
Class
  definition
Interface
  definition

```

The content of %definition% is the one defined in Classifier for an interface, and the one defined in Class for a class.

Defining template overloading

You may have different definitions of the same template that apply in different conditions. If it is the case, the template is only defined for objects that satisfy the actual condition (templates are always dynamically bound). The overloading feature can be used as a powerful switching mechanism: you can sort items in alphabetic order to provide more readability when gathering different versions of the same template. It also provides more readability because it avoids long .if conditions.

At translation-time, the translation engine evaluates each condition successively until it finds one that is verified. When it does, it takes the corresponding template definition. If no condition is satisfied, the unconditioned template, if defined, is used as default (see syntax1 in the following figure).

Conditions must be mutually exclusive to guarantee deterministic behavior.

Templates may be defined under criteria or stereotypes as well, in which case the corresponding conditions are combined.

☞ For more information on criteria and stereotypes, see sections Defining a criterion and Defining a stereotype, in chapter Managing profiles.

Example

```
full-template-name = {syntax1} <template-name> |
                    {syntax2} <template-name> '<<' stereotype '>>' |
                    {syntax3} <template-name> '<' <simple-condition> '>'
template-name      = <text>
```

☞ For more information on <simple-condition>, see section if macro.

Defining escape sequences

Escape sequences are specific characters sequences used for layout of the generated file output.

The following escape sequences can be used inside templates:

Escape sequence	Description
\n	New line character, creates a new line
\t	Tab character, creates a tab
\\	Creates a backslash
\ at the end of a line	Creates a continuation character (ignores the new line)

Escape sequence	Description
. at the beginning of a line	Ignores the line
.. at the beginning of a line	Creates a dot character (to generate a macro)
%%	Creates a percent character

For more information on escape sequences, see section Using new lines in head and tail string.

Using macros

Macros can be used to express template logic, and to loop on object collections. Each macro keyword must be preceded by a . (dot) character and has to be the first non blank character of a line. Make sure you also respect the macro syntax in terms of line breaks.

You define a macro inside a template, or a command entry.

You can use three types of macros, as shown in the following table:

Macro	Syntax
Block-macro	<if> <vbscript> <unique> <lowercase> <uppercase> <replace> <delete> <block>
Loop-macro	<foreach_item> <foreach_line> <foreach_part>
Simple-macro	<bool> <set_object> <set_value> <execute_vbscript> <execute_command> <abort_command> <change_dir> <create_path> <log> <warning> <error>

Block macro

Block macros consist of a begin and an end keyword delimiting an input block over which the macro is applied.

They have the following structure:

```

'.'<macro-name> ['(' <parameters> ')']
<macro-block-input>
'.'end<macro-name> > ['(' <tail> ')']

```

<tail> is an optional string constant appended to the generated text, if there is one.

Loop-macro

Loop macros are used for iteration. At each iteration, a new scope is created. The template specified inside the block is translated successively with respect to the iteration scope.

```

'.foreach_<macro-name> ['(' <parameters> [',' <head>
[',' <tail>]] ')' ]
<complex-template>
'.next [ '(' <separator> ')' ]

```

<head> and <tail> are both optional string constants. <head> is generated before, and <tail> is appended to the generated text, if there is one.

A <separator> may optionally be specified as an argument to the .next keyword, it is placed between non-empty evaluations of <complex-template>.

Simple-macro

Simple macros are macros that hold on a single line.

Macro parameters delimiters

Macro parameters may be delimited by double quotes. The delimiters are required whenever the parameter's value includes commas, braces, leading or trailing blanks. The escape sequence for double quotes inside a parameter value is \".

if macro

The if macro is used for conditional generation, it has the following syntax:

```

if=      .if[not] <condition>
          <complex-template>
          [(.elsif[not] <condition>
          <complex-template>)*]
          [.else
          <complex-template>]
          .endif ['(' <tail> ')']

condition=  '(' <condition> ')' or '(' <condition> ')' |
          '(' <condition> ')' and '(' <condition> ')' |
          '(' <condition> ')' |
          <simple-condition>

simple-condition=  <variable> [ <comparison-operator>
<condition-rhs> ]

comparison-operator=  '==' |
                     '!=' |
                     '<=' |
                     '>=' |

```



```

'<' |
'>' |
condition-rhs= <simple-template> |
                <string-constant> |
                true |
                false |
                null |
                notnull |
string constant= ''' <text> '''

```

Parameters

Parameter	Type	Description
<tail>	Text	Appended to the output, if there is one

When a simple condition consists of a single variable without any comparison operator, the condition is false if the variable's value is "false", "null" or the null string, otherwise it is considered to be true.

The <, >, >=, and <= comparison operators perform integer comparisons if both operands are integers, otherwise they induce an intelligent string comparison that takes into account embedded numbers (example: Class_10 is greater than Class_2).

vbscript macro

The vbscript macro is used to embed VB script code inside a template. It is a block macro.

A vbscript macro has the following syntax:

```

.vbscript ['(' <script-param-list> ')]
<vbscript-block-input>
.endvbscript ['(' <tail> ')]

```

Parameters

Parameter	Type	Description
<script-param-list> (optional)	List of arguments of type simple-template separated by commas	Parameters that are passed onto the script through the ScriptInputArray table
<vbscript-block-input>	Text	VB script text
<tail>	Text	Appended to the output, if there is one

Output

The output is the ScriptResultArray value.

Example:

```
.vbscript(hello, world)

ScriptResult = ScriptInputArray(0) + " " +
ScriptInputArray(1)

.endvbscript
```

The output is:

```
hello world
```

Note: the active object of the current translation scope can be accessed through the ActiveSelection collection as ActiveSelection.Item(0).

🔗 For more information on ActiveSelection, see section Global properties, in section Accessing objects using VBScript, in the *General Features Guide*.

unique macro

The purpose of the unique macro is to define a block in which each line of the text generated is guaranteed to be unique. It can be useful for calculating imports, includes, typedefs, or forward declarations in languages such as Java, C++ or C#.

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

Parameters

Parameter	Type	Description
<unique-block-input>	Complex template	Parameter used to input text
<tail>	Text	Appended to the output, if there is one

Output

The output is the unique block input where every redundant line has been removed.

Example:

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

lowercase macro

The .lowercase macro transforms a text block in lowercase characters.

```
.lowercase
<lowercase-block-input>
.endlowercase
```

This macro is particularly useful when you work with naming conventions.

✍ For more information about naming conventions, see section Defining naming conventions, in chapter Managing Models, in the *General Features Guide*.

Parameters

Parameter	Type	Description
<lowercase-block-input>	Complex template	Parameter used to input text

Output

The output is the lowercase block input that is changed to lowercase characters.

In the following example, the variable %Comment% is 'HELLO WORLD' and it is converted to 'hello world'.

```
.lowercase
%Comment%
.endlowercase
```

uppercase macro

The .uppercase macro transforms a text block in uppercase characters.

```
.uppercase
<uppercase-block-input>
.enduppercase
```

This macro is particularly useful when you work with naming conventions.

✍ For more information about naming conventions, see section Defining naming conventions, in chapter Managing Models, in the *General Features Guide*.

Parameters

Parameter	Type	Description
<uppercase-block-input>	Complex template	Parameter used to input text

Output

The output is the uppercase block input that is changed to uppercase characters.

In the following example, the variable %Comment% is 'hello world' and it is converted to 'HELLO WORLD'.

```
.uppercase
%Comment%
.enduppercase
```

replace macro

The .replace macro replaces all occurrences of a string with another string in a text block.

This macro is particularly useful when you work with naming conventions.

For more information about naming conventions, see section Defining naming conventions, in chapter Managing Models, in the *General Features Guide*.

The .replace macro replaces the old string <OldString> with the <NewString> string in the text block <Block>.

```
.replace '(' <old-string> ',' <new-string> ')'
<replace-block-input>
.endreplace
```

Parameters

Parameter	Type	Description
<old-string>	Text	String containing the character to be replaced by <new-string>
<new-string>	Text	String containing the character replacing <old-string>
<replace-block-input>	Complex template	Parameter used to input text

Output

The output is that all instances of the string <old-string> are replaced by instances of the string <new-string> in the replace block input.

In the following example, 'GetCustomerName' is converted to 'SetCustomerName'.

```
.replace( get , set )
GetCustomerName
.endreplace
```

In the following example, the variable %Name% is 'Customer Factory' and it is converted to 'Customer_Factory'.

```
.replace(" ", "_")
%Name%
.endreplace
```

delete macro

The .delete macro deletes all occurrences of a string in a text block. In the following script, it deletes the string <del-string> in its block content.

```
.delete '('<del-string> ' '
<delete-block-input>
.enddelete
```

This macro is particularly useful when you work with naming conventions.

For more information about naming conventions, see section Defining naming conventions, in chapter Managing Models, in the *General Features Guide*.

Parameters

Parameter	Type	Description
<del-string>	Text	String to be deleted in the input block
<delete-block-input>	Complex template	Parameter used to input text

Output

The output is that all instances of the string <del-string> are removed in the delete block input.

In the following example, 'GetCustomerName' is converted to 'CustomerName'.

```
.delete( get )
GetCustomerName
.enddelete
```

In the following example, the variable %Code% is 'm_myMember' and it is converted to 'myMember'.

```
.delete(m_)
%Code%
.enddelete
```

block macro

The .block macro is used to add a header and/or a footer to its content when it is not empty.

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

Parameters

Parameter	Type	Description
<head> (optional)	Simple template	Generated before output, if there is one
<tail> (optional)	Text	Appended to the output, if there is one
<block-input>	Complex template	Parameter used to input text

Output

The output is the concatenation of <head>, the evaluation of the <block-input> and the <tail>.

Example:

```
.block (<b>)
The current text is in bold
.endblock (</b>)
```

convert_name macro

The .convert_name macro uses the conversion table of a name into a code. When no occurrence is found in the table, the name is returned.

```
.convert_name (<Expression>)
```

The expression <Expression> is the name to be converted in the corresponding conversion table.

You can use the .convert_name macro together with a user-defined conversion table that you select in the Conversion Table dropdown listbox. To do so, open the Model Options dialog box, select the required object in the Naming Convention category and click the Name To Code tab.

Conversion tables are not case sensitive. You can indifferently use lower or uppercases in tables. This macro is particularly useful when you work with naming conventions.

For more information about naming conventions, see section Defining naming conventions, in chapter Managing Models, in the *General Features Guide*.

Note

You can also use this macro outside the naming conventions context provided the conversion table is the table of the current object of the script. Here is an example of a macro that can be added from the Profile\Column category in a new Generated Files entry:

```
.foreach_item(Columns)
  %Name%,
  .foreach_part(%Name%)
    .convert_name(%CurrentPart%)
  .next("_")
.next("\n")
```

For more information on the .convert_name macro, see section .convert_name & .convert_code macros, in chapter Defining name/code conversions, in the *General Features Guide*.

convert_code macro

The .convert_code macro uses the conversion table of a code into a name. When no occurrence is found in the table, the code is returned.

```
.convert_code (<Expression>)
```

The expression <Expression> is the code to be converted in the corresponding conversion table.

You can use this macro together with a user-defined conversion table that you select in the Conversion Table dropdown listbox. To do so, open the Model Options dialog box, select the required object in the Naming Convention category and click the Code To Name tab.

Conversion tables are not case sensitive. You can indifferently use lower-or-uppercases in tables. This macro is particularly useful when you work with naming conventions.

For more information about naming conventions, see section Defining naming conventions, in chapter Managing Models, in the *General Features Guide*.

Note

You can also use this macro outside the naming conventions context provided the conversion table is the table of the current object of the script.

For more information on the `.convert_code` macro, see section `.convert_name` & `.convert_code` macros, in chapter Defining name/code conversions, in chapter Managing Models, in the *General Features Guide*.

foreach_item macro

The `foreach_item` macro is used for iterating on object collections. The template specified inside the block is translated over all objects contained in the specified collection. The block defines a new scope wherein the active object at iteration `i` is the `i`-th collection member.

If a comparison is specified, items in the collection are pre-sorted according to the corresponding rule before being iterated upon:

```
.foreach_item '(' <collection-scope> [',' <head> [','  
<tail> [',' <simple-condition> [',' <comparison> ]]]')'  
  
<complex-template>  
  
.next ['(' <separator> ')']
```

Defining collections

All collections are accessible for a given object. Each PowerDesigner object can have one or several collections corresponding to the objects it interacts with. Collections express the link between objects, for example a table has collections of columns, indexes, business rules and so on.

Collections are represented in the PowerDesigner public metamodel by associations between objects. The roles of the associations correspond to the collections of an object.

For more information about object collections, see section Associations and collections, in chapter PowerDesigner Public Metamodel.

Parameters

Parameter	Type	Description
<collection-scope>	Simple template	Collection over which iteration is performed
<head> (optional)	Text	Generated before output, if there is one
<tail> (optional)	Text	Appended to the output, if there is one
<simple-condition> (optional)	Simple condition	If specified, only objects satisfying the given condition are considered during the iteration
<comparison>	Simple condition	<comparison> is evaluated in a scope where two local objects respectively named 'Item1' and 'Item2' are defined. These correspond to items in the collection. <comparison> should evaluate to true if Item1 is to be placed after Item2 in the iteration

Parameter	Type	Description
<separator> (optional)	Text	Generated between non empty evaluations of <complex-template>

Macro parameters delimiters

Macro parameters may be delimited by double quotes. The delimiters are required whenever the parameter's value includes commas, braces, leading or trailing blanks. The escape sequence for double quotes inside a parameter value is `\`.

Output

The output is the concatenated evaluations of <complex-template> over all the objects in the collection.

Example:

Attribute	Data type	Initial value
cust_name	String	—
cust_foreign	Boolean	false

```
.foreach_item(Attributes,,,%Item1.Code% >=
%Item2.Code%)
Attribute %Code%[ = %InitialValue%];
.next (\n)
```

The result is:

Attribute cust_foreign = false

Attribute cust_name;

Note

The four commas after (Attributes,,, means that all parameters (head, tail, condition and comparison) are skipped.

foreach_line macro

The foreach_line macro is a simple macro that iterates on the lines of the input template specified as the first argument to the macro. The template specified inside the block is translated for each line of the input. This macro creates a new scope with the local variable CurrentLine. This one is defined inside the block to be the i-th line of the input template at iteration i.

```
.foreach_line '(' <input> [',' <head> [',' <tail>]] ')'
<complex-template>
.next ['(( <separator> ')']
```

Parameters

Parameter	Type	Description
<input>	Simple template	Input text over which iteration is performed
<head> (optional)	Text	Generated before output, if there is one
<tail> (optional)	Text	Appended to the output, if there is one
<separator> (optional)	Text	Generated between non empty evaluations of <complex-template>

Output

The output is the concatenated evaluations of <complex-template> for each line of the translated template <input>.

Example:

```
.foreach_line(%Comment%)
// %CurrentLine%
.next(\n)
```

foreach_part macro

The .foreach_part macro iterates on the part of the input template specified as the first argument to the macro. The template specified inside the block is translated for each part of the input.

A part is delimited with a separator pattern. There are two kinds of separator:

- ◆ Char separator: for each char separator, the separator specified in the next statement of the macro is returned (even for consecutive separators)
- ◆ Word separator: they are specified as interval, for example [A-Z] specifies that all capital letters are separator. For a word separator, no separator (specified in next statement) is returned

```
.foreach_part '(' <input> ['<separator-pattern> ['<head> [, <tail>]]] ')'
<simple-template>
.next['(' <separator> ')']
```

This macro creates a new scope wherein the local variable CurrentPart is defined to be the i-th part of the input template at iteration i. The Separator local variable contains the following separator.

This macro is particularly useful when you work with naming conventions.

For more information about naming conventions, see section Defining naming conventions, in chapter Managing Models, in the *General Features Guide*.

Separator-pattern

The `<separator-pattern>` is defined in a double quote string " " and behaves as follows:

- ◆ Any character specified in the pattern can be used as separator
- ◆ [`<c1>` - `<c2>`] specifies a character within the range defined between both characters `<c1>` and `<c2>`

For example, the following pattern “ -_, [A-Z] ” specifies that each part can be separated by a space, a dash, an underscore, a comma or a character between A and Z (in capital letter).

By default, the `<separator-pattern>` is initialized with the pattern (“ -_, \t”). If the specified pattern is empty, the pattern is initialized using the default value.

A separator `<separator>` can be concatenated between each part. `<head>` and `<tail>` expressions can be added respectively at the bottom or at the end of the generated expression.

Parameters

Parameter	Type	Description
<code><input></code>	Simple template	Input text over which iteration is performed
<code><separator-pattern></code>	Text	Char and word separators
<code><head></code> (optional)	Text	Generated before output, if there is one
<code><tail></code> (optional)	Text	Appended to the output, if there is one
<code><separator></code> (optional)	Text	Generated between non empty evaluations of <code><complex-template></code>

Output

The output is the concatenated evaluations of `<simple-template>` for each part of the translated template `<input>`.

Examples:

Convert a name into a class code (Java naming convention). In the following example, the variable `%Name%` is equal to 'Employee shareholder', and it is converted to 'EmployeeShareholder':

```
.foreach_part (%Name%, " _- ' ")
%.FU:CurrentPart%
.next
```

Convert a name into a class attribute code (Java naming convention). In the following example, the variable %Name% is equal to 'Employee shareholder', and it is converted to 'employeeShareholder':

```
.set_value(_First, true, new)
.foreach_part(%Name%, "' _-'" )
.if (%_First%)
%.L:CurrentPart%
.set_value(_First, false, update)
.else
%.FU:CurrentPart%
.endif
.next
```

bool macro

This macro returns 'true' or 'false' depending on the value of the condition specified.

```
.bool '(' <condition> ')'
```

Parameters

Parameter	Type	Description
<condition>	Condition	Condition to be evaluated

Example:

```
.bool (%.3:Code%= =ejb)
```

set_object macro

This macro is used to define a local variable of object type (local object).

```
.set_object '(' <local-var-name> [' ,' <object-ref> [' ,' <new> ]] ')'
```

Parameters

Parameter	Type	Description
<local-var-name>	Simple-template	Variable name
<object-ref> (optional)	[<scope>.]<object-scope>]	Describes an object reference. If it is not specified or is an empty string, the variable is a reference to the active object in the current translation scope

Parameter	Type	Description
<new> (optional)	new or update (the default value is update)	If the parameter is 'new', it creates a new variable even if there is already one in the (enclosing) scope, whereas the 'update' parameter modifies the value of the variable in the current scope, or it creates a new variable if there is none

The variable is a reference to the object specified using the second argument.

Example:

```
.set_object (Attribute1, Attributes.First)
```

Note

When specifying a new variable, it is recommended to specify 'new' as third argument to ensure that a new variable is created in the current scope.

set_value macro

This macro is used to define a local variable of value type.

```
.set_value '(' <local-var-name> ')' <value>[',' <new> '']
```

Parameters

Parameter	Type	Description
<local-var-name>	Simple template	Variable name
<value>	Simple template (escape sequences ignored)	Value
<new> (optional)	New or update (the default value is update)	If the value is new, it (re)defines the variable in the current scope. If the value is update, it updates the existing variable, otherwise it defines a new one

The variable's value is set to be the translated template value specified as the second argument.

Example:

```
.set_value (FirstAttributeCode, %Attributes.First.Code%)
```

Note

When specifying a new variable, it is recommended to specify 'new' as third argument to ensure that a new variable is created in the current scope.

execute_vbscript macro

This macro is used to execute a VB script specified in a separate file.

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

Parameters

Parameter	Type	Description
<vbs-file>	Simple template (escape sequences ignored)	VB script file path
<script- parameter> (optional)	Simple template	Parameter passed on to the script through the ScriptInputParameters global property

Output

The output is the ScriptResult global property value.

Example:

```
.execute_vbscript (C:\samples\vbs\login.vbs, %username%)
```

Note: the active object of the current translation scope can be accessed through the ActiveSelection collection as ActiveSelection.Item(0).

For more information on ActiveSelection, see section Global properties, in section Accessing objects using VBScript, in the *General Features Guide*.

execute_command macro

This macro is used to launch executables as separate processes. It is available to execute generation commands only, and may be used in addition to standard GTL macros when defining commands.

```
.execute_command '(' <cmd> [',' <args> [',' <mode>]] ')' '
```

Parameters

Parameter	Type	Description
<cmd>	Simple template (escape sequences ignored)	Executable path

Parameter	Type	Description
<args> (optional)	Simple template (escape sequences ignored)	Arguments for the executable
<mode> (optional)	cmd_ShellExecute or cmd_PipeOutput	cmd_ShellExecute runs as independent process cmd_PipeOutput blocks until completion, and shows the executable's output in the output window

Note that if an `.execute_command` fails for any given reason (executables not found, or output sent to stderr), the command execution is stopped.

Example:

```
.execute_command(notepad, file1.txt, cmd_ShellExecute)
```

abort_command macro

This macro stops command execution altogether. It is available to execute generation commands only, and may be used in addition to standard GTL macros when defining commands.

Example:

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

change_dir macro

This macro changes the current directory. It is available to execute generation commands only, and may be used in addition to standard GTL macros when defining commands.

```
.change_dir '(' <path> ')'
```

Parameters

Parameter	Type	Description
<path>	Simple template (escape sequences ignored)	New current directory

Example:

```
.change_dir (C:\temp)
```

create_path macro

This macro creates a specified path if it does not exist.

```
.create_path '(' <path> ')'
```

Parameters

Parameter	Type	Description
<path>	Simple template (escape sequences ignored)	Path to be created

Example:

```
.create_path(C:\temp)
```

log macro

This macro logs a message to the Generation page of the Output window, located in the lower part of the main window. It is available to execute generation commands only, and may be used in addition to standard GTL macros when defining commands.

```
.log <message>
```

Parameters

Parameter	Type	Description
<message>	Simple template	Message to be logged

Example:

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

warning macro

The warning macro is used to output a warning during translation. It may be useful to display a message when an inconsistency is detected while applying the template on a particular object. Warnings do not stop generation, they do not appear in the generated file either. The content of the warning message is the evaluated simple-template, passed by parameter.

It appears in the Output window (located in the lower part of the main window) during generation, and is shown in the code from the Preview page of the object, if at least one error has been encountered.

```
.warning <message>
```

Parameters

Parameter	Type	Description
<message>	Simple template	Warning message

Example:

```
.warning bidirectional associations between inner
classes are not supported
```

error macro

The error macro is similar to the warning macro as it does not appear in the generated file, however it stops generation. The error message is the specified template translated over the active object. It is displayed in both Preview page of the object, and the Output window.

```
.error <message>
```

Parameters

Parameter	Type	Description
<message>	Simple template	Error message

Example:

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

comment and // macro

The comment, and // macro is helpful for inserting comments in a template. Lines starting with `./` or `.comment` are ignored during generation.

Example:

```
./ This is a user defined comment
```

Defining conditional blocks

Conditional blocks can be used to specify different templates based on the value of a variable.

```
{syntax 1}: '[' <variable> '?' <simple-template> [ ':'
<simple-template> ] ' ] '
```

```
{syntax 2}: '[' <text> <variable> <text> ' ] '
```

Syntax 1 is similar to C and Java ternary expressions. If the value of the variable is false, null, or the null string, the second template, if specified, is evaluated, otherwise the first one is evaluated.

Syntax 2 has a slightly different meaning. The simple template `<text><variable><text>` is translated if, and only if the value of the variable is not the null string.

Example: an attribute declaration in Java:

```
%Visibility% %DataType% %Code% [= %InitialValue%]
```

Defining error messages

Error messages stop the generation of the file in which errors have been found, these errors are displayed in the Preview page 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>
```

You can find the following types of errors:

- ◆ Syntax errors
- ◆ Translation errors

Syntax errors

You may encounter the following syntax errors:

Syntax error message	Description and correction
condition parsing error	Syntax error in a boolean expression
expecting .endif	Add a .endif
.else with no matching .if	Add a .if to the .else
.endif with no matching .if	Add a .if to the .endif
expecting .next	Add a .next
expecting .end%s	Add a .end%s (for example, .endunique, .endreplace, ...)
.end%s with no matching .%s	Add a .<macro> to the .end<macro>
.next with no matching .foreach	Add a .foreach to the .next
missing or mismatched parentheses	Correct any mismatched braces
unexpected parameters: <extra-params>	Remove unnecessary parameters
unknown macro	The macro is not valid

Syntax error message	Description and correction
.execute_command incorrect syntax	The correct syntax appears in the Preview page, or in the Output window. It should be: <code>.execute_command(<executable>[,<arguments>[, {cmd_ShellExec cmd_PipeOutput}]]])</code>
Change_dir incorrect syntax	The syntax should be: <code>.change_dir(<path>)</code>
convert_name incorrect syntax	The syntax should be: <code>.convert_name(<name>)</code>
convert_code incorrect syntax	The syntax should be: <code>.convert_code(<code>)</code>
set_object incorrect syntax	The syntax should be: <code>.set_object(<local-var-name>[,<scope>.]<object-scope>[, {new update}]]])</code>
set_value incorrect syntax	The syntax should be: <code>.set_value(<local-var-name>,<simple-template>[, {new update}]]])</code>
execute_vbscript incorrect syntax	The syntax should be: <code>.execute_vbscript(<script-file>[,<script-input_params>])</code>

Translation errors

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

You may encounter the following translation errors:

Translation error message	Description and correction
unresolved collection: <collection>	Unknown collection
unresolved member: <member>	Unknown member
no outer scope	Invalid use of Outer keyword
null object	Occurs when trying to access a null object's member
expecting object variable: <object>	Occurs when using string instead of object
VBScript execution error	VB script error
Deadlock detected	Deadlock due to an infinite loop

Generation tips and techniques

This section provides additional information about the GTL mechanism.

Sharing templates

In the GTL mechanism you can share conditions, templates and sub-templates to ease object language maintenance and readability.

Sharing conditions

A template can contain a condition expression. You can also create templates to share long and fastidious condition expressions:

Template name	Template value
%ConditionVariable%	.bool (condition)

Instead of repeating the condition in other templates, you simply use %ConditionVariable% in the conditional macro:

```
.if (%ConditionVariable%)
```

Example

The template %isInner% contains a condition that returns true if the classifier is inner to another classifier.

```
.bool (%ContainerClassifier!=null)
```

This template is used in the %QualifiedCode% template used to define the qualified code of the classifier:

```
.if (%isInner%)
%ContainerClassifier.QualifiedCode%:%Code%
.else
%Code%
.endif
```

Using recursive templates

A recursive template is a template that is defined in terms of itself.

Example

Consider three classes X, Y, and Z. X is inner to Y, and Y is inner to Z.

The variable %topContainerCode% is defined to retrieve the value of the parent container of a class.

The value of the template is the following:

```
.if (%isInner%)
%ContainerClassifier.topContainerCode%
.else
%Code%
.endif
```

If the class is inner to another class, %topContainerCode% is applied to the container class of the current class (%ContainerClassifier.topContainerCode%).

If the class is not an inner class, the code of the class is generated.

Using environment variables

In GTL, you can access variables located in the General Options feature of the Tools menu. If the variable has not been set, a null string is returned.

Using new lines in head and tail string

The head and tail string are useful because they are only generated when necessary, it is especially useful when using new lines '\n'. They are added respectively at the beginning, and at the end of the generated code from a macro. If no code is generated, the head and tail string do not appear in the generated code.

Example

You want to generate the name of a class and its attributes under the following format (one empty line between attributes and class):

```
Attribute 1 attr1
Attribute 2 attr2

Class
```

You can insert the separator "\n" after the .foreach statement to make sure each attribute appears in a separate line. You can also add "\n\n" after the .endfor statement to insert an empty line after the attribute list and before the word "Class".

```
.foreach (Attribute) ("\n")
Attribute %Code%
.endfor ("\n\n")
Class
```

Additional example

Consider a class named **Nurse**, with a class code Nurse, and two attributes:

Attribute name	Data type	Initial value
NurseName	String	—
NurseGender	Char	'F'

The following templates are given as examples, together with the text generated for each of them, and a description of each output:

Template 1

```
class "%Code%" {
  // Attributes
  .foreach_item(Attributes)
  %DataType% %Code%
  .if (%InitialValue%)
  = %InitialValue%
  .endif
  .next
  // Operations
  .foreach_item(Operations)
  %ReturnType% %Code%(...)
  .next
}
```

Text generated 1

```
class "Nurse" {
  // Attributes String nurseName char nurseGender = 'F' // Operations}
```

Description 1

Below the class code, the code is generated on one line. It is an example of a block macro (.if, .endif macro).

Template 2 (new line)

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

Text generated 2

```
class "Nurse" {
  // Attributes String nurseName
  char nurseGender = 'F' // Operations}
```

Description 2	<p>String nurseName and char nurseGender are on two lines</p> <p>In Template 1, String nurseName and char nurseGender were on the same line, whereas in Template 2, the addition of the \n at .next(\n) puts String nurseName and char nurseGender on two different lines.</p> <p>In addition, // Operations is displayed in the output even if there is no operation (see Description 3).</p>
Template 3 (blank space)	<pre>class "%Code%" { .foreach_item(Attributes, // Attributes\n,\n) %DataType% %Code% .if (%InitialValue%) = %InitialValue% .endif .next(\n) .foreach_item(Operations, // Operations\n,\n) %ReturnType% %Code%(...) .next(\n) }</pre>
Text generated 3	<pre>class "Nurse" { // Attributes String nurseName char nurseGender = 'F' }</pre>
Description 3	<p>The blank space between .foreach_item(Attributes, and // Attributes\n,\n) is not generated, as shown in the output: class "Nurse" { // Attributes instead of { // Attributes</p> <p>// Operations is not displayed in the output because it is positioned in the .foreach_item macro. It is positioned in the head of the macro for this purpose.</p>
Template 4 (blank space)	<pre>class "%Code%" { \n .foreach_item(Attributes, " // Attributes\n", \n) %DataType% %Code% [= %InitialValue%] .next(\n) .foreach_item(Operations, " // Operations\n", \n) %ReturnType% %Code%(...) .next(\n) }</pre>
Text generated 4	<pre>class "Nurse" { // Attributes String nurseName char nurseGender = 'F' }</pre>

Description 4

The double quote characters (") in " // Attributes\n" allows you to insert a blank space as shown in the output: // Attributes

Newline preceding the macro

The newline immediately preceding a macro is ignored as well as the one immediately following it, as in the following example:

```
Jack
.set_value(v, John)
Paul
```

yields: JackPaul

instead of:

Jack

Paul

Using parameter passing

You can pass in, out or in/out parameters to a template through local variables by taking advantage of nested translation scopes. You can access parameters with the %@<number>% variable.

Example

Template 1

Class templates:

```
<show> template
<<<
Class "%Code%" attributes :
// Public
%publicAttributes%

// Protected
%protectedAttributes%

// Private
%privateAttributes%
>>>
```

Template 2

```
<publicAttributes> template
<<<
.foreach_item(Attributes)
.if (%Visibility% == +)
%DataType %Code%
.endif
.next(\n)
>>>
```


Template 3

```
<protectedAttributes> template
<<<
.foreach_item(Attributes)
  .if (%Visibility% == #)
    %DataType %Code%
  .endif
.next(\n)
>>>
```

Template 4

```
<privateAttributes> template
<<<
.foreach_item(Attributes)
  .if (%Visibility% == -)
    %DataType %Code%
  .endif
.next(\n)
>>>
```

To give you more readability and to enhance code reusability, these four templates can be written in just two templates by using parameters:

First template

```
<show> template
<<<
Class "%Code%" attributes :
// Public
%attributes(+) %

// Protected
%attributes(#) %

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

Second template

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

Description

The first parameter in this example %attributes(+, or #, or -)% can be accessed using the variable %@1%, the second parameter when it exists, is accessed using the %@2% variable, etc ...

CHAPTER 6

PowerDesigner Public Metamodel

About this chapter This chapter describes the content and use of the PowerDesigner public metamodel.

Contents

Topic	Page
What is the PowerDesigner public metamodel?	324
Metamodel concepts	325
Understanding the metamodel structure	328
Using the metamodel with VBS	332
Using the metamodel with the Generation Template Language	334
Using the metamodel to understand the PowerDesigner XML file format	337

What is the PowerDesigner public metamodel?

Metamodel definition

Metamodeling is a logical extension of the abstraction process that is used to create object models. An object model is an abstraction of data, and can be described using metadata. A metamodel is an abstraction of metadata.

Metamodels describe formally the model elements, and the syntax and semantics of the notation that allow their manipulation. They are used to increase the power, flexibility, and versatility of a software because they isolate the application software from changes in the application model. For example, you can use a metamodel to represent the object model of an object model, the object model of a dynamic model, or the grammar of a grammar.

PowerDesigner metamodel

The PowerDesigner public metamodel is an abstraction of the PowerDesigner metadata represented in an Object Oriented Model.

The PowerDesigner public metamodel is divided in different packages containing classes which relate to each other via associations and generalizations. Each class has a name (the public name) and is described by zero or more attributes; it may assume various roles in associations with other classes.

Why publishing a metamodel?

The PowerDesigner public metamodel is intended to:

- ◆ Make sure that all coding and design changes are represented fully in the metamodel architecture and avoid inconsistent coding
- ◆ Help users have a big picture and in-depth understanding of PowerDesigner metadata
- ◆ Complement the Visual Basic help file in order to let users create their own VB scripts
- ◆ Help users use the Generation Template Language (GTL)
- ◆ Help users understand and modify PowerDesigner files saved in XML format

Metamodel concepts

The PowerDesigner public metamodel fully supports the UML method and formalism. The following sections briefly summarize the metamodel terms and concepts.

Public names

Each object in the PowerDesigner metamodel has a name and a code corresponding to the **public name** of the object. The public name of an object is the unique identifier of this object in a model library. A model library corresponds to a package visible in the Modules diagram in the metamodel, for example PdCommon. The public name does not always match the object name in the PowerDesigner interface.

Public names are also used in the XML files and in the PowerDesigner Generation Template Language.

🔗 For more information on XML files, see chapter PowerDesigner File Format Specification.

🔗 For more information on text generation, see chapter Generation Reference Guide.

Classes

Classes are used to represent metadata in the following way:

- ◆ **Abstract classes** are used to share attributes and behaviors. They are not visible in the PowerDesigner interface. Instantiable classes inherit from abstract classes via generalization links. For example, NamedObject is an abstract class, it stores standard attributes like name, code, comment, annotation, and description inherited by most PowerDesigner design objects
- ◆ **Instantiable/Concrete classes** correspond to objects displayed in the interface, they have their own attributes like type or persistence, and they inherit attributes and behaviors from abstract classes through generalization links

The PowerDesigner metamodel highlights the inheritance links among metadata.

Class attributes

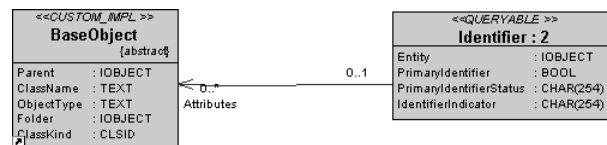
Attributes are class properties that can be **derived** or not. Classes linked to other classes with generalization links usually contain derived attributes that are calculated from the attributes or collections of the parent class. Non-derived attributes are proper attributes of the class. These attributes are stored in the model and saved in the model file.

Neither derived attributes, nor attributes migrated from navigable associations, are stored in the model file.

Associations and collections

Associations are used to express the semantic connections between classes called **collections**. In the association property sheet, the roles carry information about the end object of the association. In the PowerDesigner metamodel, this role has the same name as a collection for the current object. PowerDesigner objects are linked to other objects using collections.

Usually associations have only one role, the role is at the opposite of the class for which it represents a collection. In the following example, Identifier has a collection called Attributes:



When associations have two roles, both collections cannot be saved in the XML file, only the collection with the **navigable** role will be saved.

For more information on navigable roles, see section XML and the PowerDesigner metamodel in chapter PowerDesigner File Format Specification.

Composition

Among associations, the **compositions** express a strong ownership of child classes by the parent class; the children live and die with the parent. When the parent is copied, the child is also copied.

For example, in package PdCommon, diagram Option Lists, class NamingConvention is associated with class BaseModelOptions with 3 composition associations: NameNamingConventions, CodeNamingConventions, and NamingConventionsTemplate. These composition associations express the fact that class NamingConvention would not exist without class BaseModelOptions.

Generalizations

The PowerDesigner metamodel uses generalizations to show the **inheritance** links existing between a more general class (usually an abstract class) and a more specific (usually an instantiable class). The more specific class inherits from the attributes of the more generic class, these attributes are called derived attributes.

By sharing attributes and behaviors, inheritance increases the readability of the metamodel.

Comments and notes on objects

Most classes, diagrams, and packages have a comment that explains their role in the metamodel.

Some internal implementation details are also available in the Notes→Annotation page of the classes property sheets.

Understanding the metamodel structure

The PowerDesigner metamodel is published in an Object Oriented Model. It uses the UML class diagram formalism to represent PowerDesigner metadata.

Overall organization

When you open the PowerDesigner metamodel, the Modules diagram appears, and you can observe a series of packages linked to PdCommon. These packages represent the different libraries of PowerDesigner, each library (apart from PdCommon) is equivalent to a model type:

Package name	Corresponding model
PdCDM	Conceptual Data Model
PdPDM	Physical Data Model
PdOOM	Object Oriented Model
PdBPM	Business Process Model
PdFRM	Free Model
PdRMG	Repository

Package **PdCommon** does not correspond to a particular model, it gathers all objects shared among two or more models, for example, business rules are defined in this package.

It also defines the abstract classes of the model, for example, BaseObject is defined in diagram Common Abstract Objects in the Objects package of PdCommon.

🌀 For more information on abstract classes, see section Classes.

Library packages are linked to PdCommon by generalization links indicating that each model inherits common objects from the PdCommon library.

PdCommon content

The library PdCommon is organized in sub-packages containing different diagrams that illustrate a certain aspect of the library.

Sub-package	Description
Features	All the features implemented by classes in PdCommon. For example, Report belongs to PdCommon because this feature is shared by all models
Objects	Design objects shared by two or more models
Symbols	Graphical representation of shared design objects

Other library packages content

All other library packages display diagrams corresponding to different aspects of the model.

Diagram	Description
Features	All the features implemented by classes in the current library. For example, AbstractDataType is a feature in PdPDM
Objects	Design objects of the current library
Symbols	Graphical representation of design objects

In package PdOOM and PdPDM, the different types of diagrams supported in PowerDesigner appear as different diagrams in the package.

Navigating in the metamodel

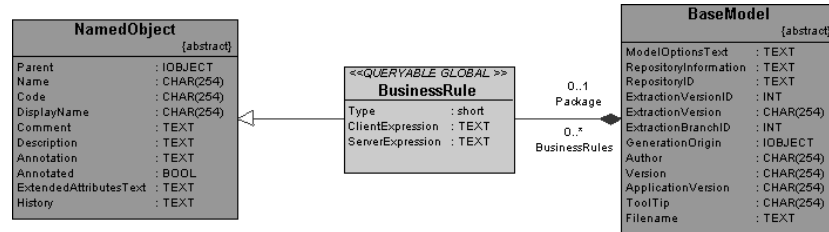
You can use the graphical interface of the OOM to navigate through the metamodel structure.

Color code

Classes in **green** are classes which behavior is explained in the current diagram.

Classes in **purple** are visible to help understanding the context but they are not explained in the current diagram.

In the following example taken from the Common Instantiable Objects in the Objects package in PdCommon, BusinessRule (green color) is developed whereas NamedObject and BaseModel are used to express inheritance and composition links with abstract classes.



Navigating using the interface

You can use the Browser to develop or collapse the categories to understand the global structure of the metamodel from a tree view. You can also display diagrams and observe the graphical representation of metadata.

Classes in purple are usually shortcuts of a class existing in another package. The shortcut makes it easier to read the diagram and understand the generalization links between classes. If you want to have an explanation about a purple class in a diagram, you can right-click the class and select **Open Related Diagram** to display the diagram where the class is actually defined.

Association roles

The diagram also displays most associations with their roles which makes it possible to identify object collections.

For more information on association roles, see section Associations and collections.

Navigating using object property sheets

You can use the **Dependency** tab in the property sheets of metamodel classes to observe:

- ◆ Associations tab: you can customize the filter in order to display association roles, this provides a list of the collections of the current object
- ◆ Generalizes tab: Displays a list of generalization links where the current object is the parent. You can use this list to display all the children of the current class. Child classes inherit attributes from the parent class and do not display derived attributes

- ◆ Specializes tab: Displays the parent of the current object. The current class inherits attributes from this parent
- ◆ Shortcuts tab displays the list of shortcuts created for the current object

You can use the **Associations** tab in the property sheets of metamodel classes to have a list of migrated associations for the current class.

Using the metamodel with VBS

You can access and manipulate PowerDesigner internal objects using Visual Basic Scripting. The scripting lets you access and modify object properties, collections, and methods using the public names of objects.

The PowerDesigner metamodel provides useful information about objects:

Information	Description	Example
Public name	The name and code of the metamodel objects are the public names of PowerDesigner internal objects	AssociationLinkSymbol ClassMapping CubeDimensionAssociation
Object collections	You can identify the collections of a class by observing the associations linked to this class in the diagram. The role of each association is the name of the collection	In PdBPM, an association exists between classes MessageFormat and MessageFlow. The public name of this association is Format. The role of this association is Usedby which corresponds to the collection of message flows of class MessageFormat
Object attributes	You can view the attributes of a class together with the attributes this class inherits from other classes via generalization links	In PdCommon, in the Common Instantiable Objects diagram, you can view objects BusinessRule, ExtendedDependency and FileObject with their proper attributes, and the abstract classes from which they inherit attributes via generalization links
Object operations	Operations in metamodel classes correspond to object methods used in VBS	BaseModel contains operation Compare that can be used in VB scripting
<<notScriptable>> stereotype	Objects that do not support VB scripting have the <<notScriptable>> stereotype	RepositoryModel ReportLanguage

 For more information on public names, see section Public names.

VBS help file

↪ For more information on generalizations, see section Generalizations.

If you need more information on object collections and attributes during VB scripting, you can use the help file available from the Edit/Run Script dialog box. This HTML help file is another view of the metamodel, designed to help VBS users identify object collections and attributes.

↪ For more information on VB scripting, see section Accessing Objects using VBScript in the *General Features Guide*.

Using the metamodel with the Generation Template Language

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

These templates can be considered as metamodel extensions as they are defined on metamodel classes, they are special kinds of metamodel attributes. The user may define as many templates as he wants for any given metaclass using the following syntax:

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

Inheritance

Templates are inherited by all the descendants of the metaclass they are defined for. This mechanism is useful for sharing template code between metaclasses having a common ancestor. For example, if you define a template for an abstract class like BaseObjects, all the classes linked via generalization links to this class inherit from this template.

Collections

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

Calculated attributes

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

Metaclass name	Attribute name	Type	Description
PdCommon. BaseObject	isSelected	boolean	True if the corresponding object is part of the selection in the generation dialog, false otherwise
	isShortcut	boolean	True if the object was accessed by dereferencing a shortcut, false otherwise
PdCommon. BaseModel	GenOptions	struct	Gives access to the user-defined generation options

Metaclass name	Attribute name	Type	Description
PdOOM. Class	MinCardinality	string	
	MaxCardinality	string	
	SimpleTypeAttribute (XML-specific)		
	@<tag> (Java-specific)	string	Javadoc@<tag> extended attribute with additional formatting
PdOOM. Interface	@<tag> (Java-specific)	string	Javadoc@<tag> extended attribute with additional formatting
PdOOM. Attribute	MinMultiplicity	string	
	MaxMultiplicity	string	
	Overriden	boolean	
	DataTypeModifier Prefix	string	
	DataTypeModifier Suffix	string	
	@<tag> (Java-specific)	string	Javadoc@<tag> extended attribute with additional formatting
PdOOM. Operation	DeclaringInterface	object	
	GetSetAttribute	object	
	Overriden	boolean	
	ReturnTypeModifier Prefix	string	
	ReturnTypeModifier Suffix	string	
	@<tag> (Java-specific)	string	Javadoc@<tag> extended attribute with additional formatting (esp. for @throws, @exception, @params)

Metaclass name	Attribute name	Type	Description
PdOOM. Parameter	DataTypeModifier Prefix	string	
	DataTypeModifier Suffix	string	
PdOOM. Association	RoleAMinMultiplicity	string	
	RoleAMaxMultiplicity	string	
	RoleBMinMultiplicity	string	
	RoleBMaxMultiplicity	string	
PdOOM.*	ActualComment	string	Cleaned-up comment (with /**, /*, */ and // removed)

Calculated collections

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

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

Using the metamodel to understand the PowerDesigner XML file format

PowerDesigner models are made up of objects which properties and interactions are explained in the public metamodel. You can use the public metamodel to better understand:

- ◆ Object names in the XML files, these names correspond to the public names available in the metamodel
- ◆ Object collections appear as association roles in the metamodel

You can also use the metamodel to view items that do not appear in the XML files:

- ◆ Derived attributes can be deducted from the metamodel when there is a generalization link between a parent and a child class. These attributes do not appear in the XML code
- ◆ Objects with the <<internal>> stereotype correspond to temporary features like CheckModelControler, that do not appear in the XML file

🔗 For more information on using the public metamodel to understand the XML file format, see section XML and the PowerDesigner metamodel in chapter PowerDesigner File Format Specification.

CHAPTER 7

PowerDesigner File Format Specification

About this chapter

This chapter describes the file format used for saving PowerDesigner models.

Contents

Topic	Page
XML file format	340
Modifying an XML file	347

XML file format

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

Model file format	Enables you to ...
BIN (Binary)	Have files that are small in size which take up less space on your disk, as well as being significantly quicker to open and save in PowerDesigner
XML (Extensible Markup Language)	Visualize and modify the structure and content of model objects in a text or XML editor

XML stands for Extensible Markup Language. It is intended to make it easy and straightforward to use SGML-defined documents, and easy to transmit and share them across the Web.

DTD and XML editors

Furthermore PowerDesigner XML files support a DTD (Document Type Definition) for each type of model. The DTD enables you to use an XML editor to browse through the hierarchical structure of the objects in the model. The different DTDs are available in the \DTD folder in the PowerDesigner installation directory.

You can find more information on XML at the following site:

<http://www.xml.com>

XML file structure and content

The following sections explain the XML syntax and structure in PowerDesigner files.

XML markups

XML files contain objects, attributes and collections usually declared with a begin and an end markup; referenced objects have single markup for begin and end as shown in table below.

The difference between a begin and an end markup is that the end markup has a slash (/) after the character <.

The following markups are used in PowerDesigner XML files:

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

XML and the PowerDesigner metamodel

PowerDesigner models are made up of objects which properties and interactions are explained in the public metamodel. The metamodel uses the OOM formalism to document PowerDesigner metadata (under the form of classes) and their links (under the form of associations and generalizations).

You can use the PowerDesigner public metamodel to better understand the format of PowerDesigner XML files.

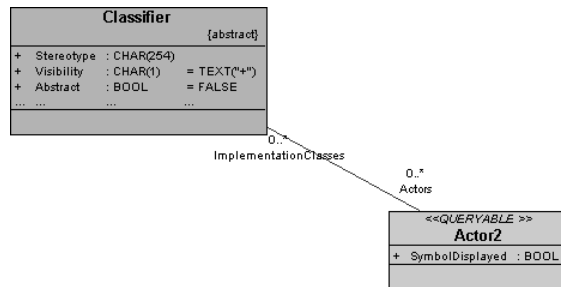
First of all, object names, as declared in markup `<o:name of object>`, correspond to public names in the metamodel. You can search for an object in the metamodel using the object name found in the XML file.

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

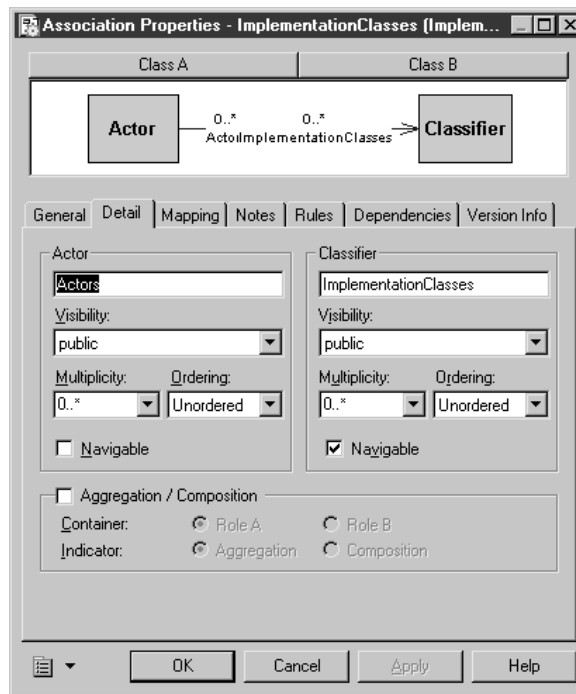
- ◆ Each PowerDesigner object can have several collections corresponding to other objects to interact with, these collections are represented by the associations existing between objects. The **roles** of the associations (aggregations and compositions included) correspond to the collections of an object.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

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

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



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



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

The PowerDesigner public metamodel is available in the \Examples directory.

For more information on the PowerDesigner Public metamodel, see chapter PowerDesigner Public Metamodel.

You can also use the help file with the .chm extension delivered with PowerDesigner. This file is intended to help users who want to create Visual Basic scripts since it provides useful information about object properties, collections, and methods.

For more information on how to create Visual Basic scripts for PowerDesigner objects, see section Accessing objects using VBScript in chapter Managing Objects in the *General Features Guide*.

Understanding the XML format

Overlapping collections

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

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

Because of overlapping collections, the format of PowerDesigner model files can be compared to a tree view: it starts from a root node (the root object containing any model collection) and cascades through collections. This analogy is used by most XML editors to display the structure of XML files.

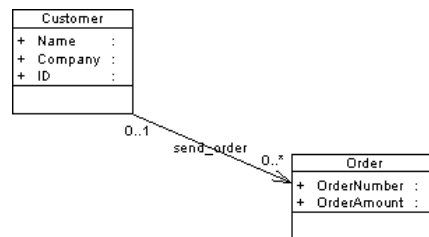
Object definition or reference

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

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

Case study

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



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

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

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

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

```

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

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

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

```

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

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

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

```

```
<a:FontList>DISPNAME 0 Arial,8,N
```

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

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

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

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

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

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

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

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

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

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

Modifying an XML file

When you need to modify a model using its XML file, you should be very careful when adding or removing objects. Because of the overlapping structure of the file, even a minor syntax error could result in the file being unusable.

OID

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

You can modify an XML file using:

- ◆ A standard text editor
- ◆ An XML editor

You can visualize an XML file using an XML viewer.

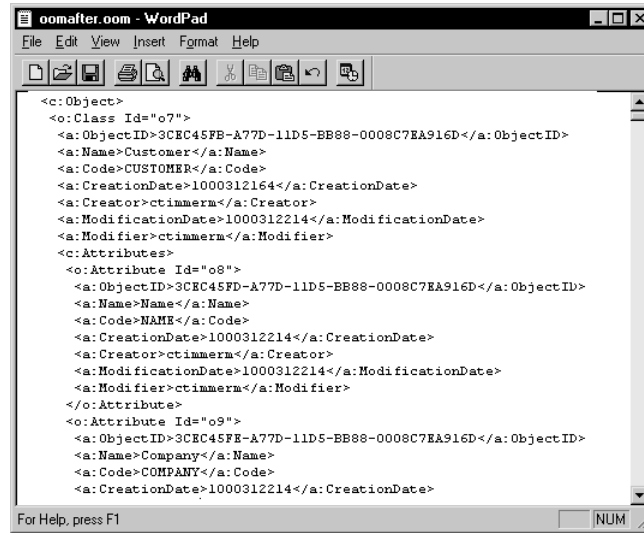
Modifying an XML file using a standard Editor

If you are using a standard editor, you can customize the XML formatting as defined in section Modifying an XML file.

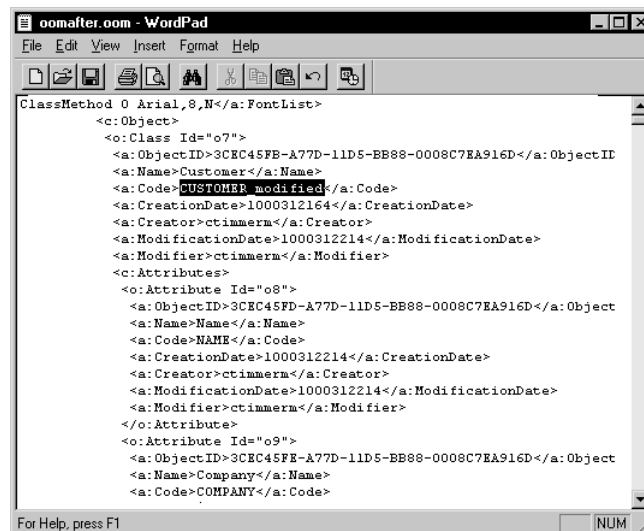
❖ To modify the content of an XML file:

- 1 Open the file in a text editor.

The content of the XML file is displayed. The objects are ordered according to the collections they belong to.



- 2 Browse through the file or use the search tool to find an object name in the model.
- 3 You can modify directly the properties of an object being careful not to modify the begin and end markups of the object's definition, i.e. between the < and > symbols.



- 4 Save and close the file.

If you open the file in PowerDesigner, the object will appear with the new name.

Modifying an XML file using an XML editor

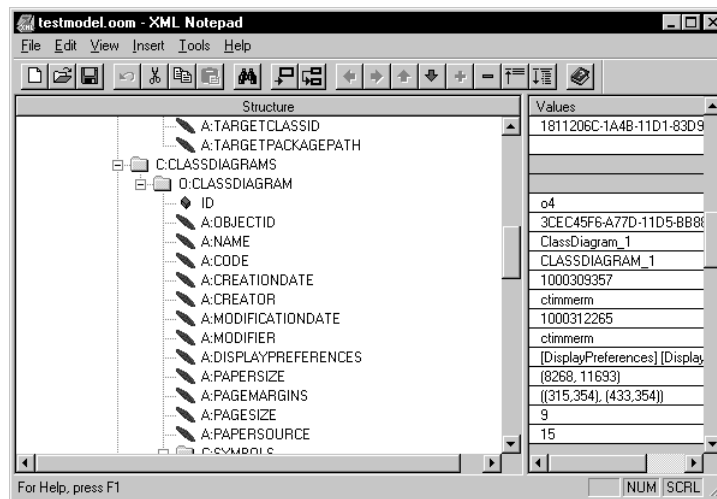
You can modify the file in an XML editor. In this section, the XML editor used is Microsoft XML Notepad.

❖ To visualize the content of an XML file in an XML editor:

- 1 Open the file in an XML editor.

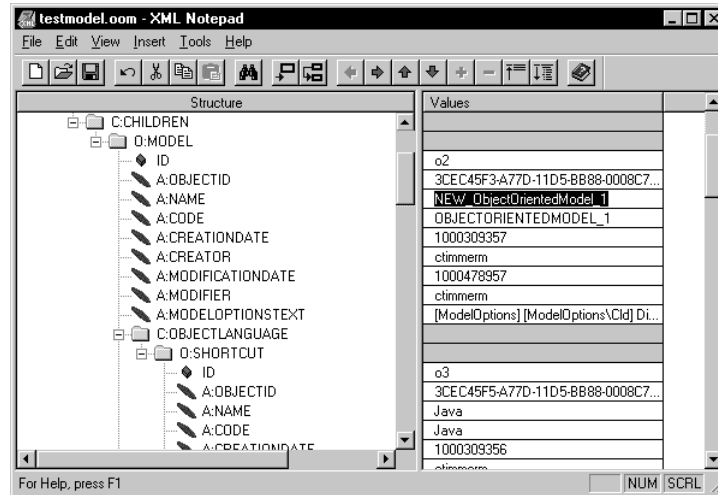
The XML file is displayed. The objects are displayed according to the hierarchy that reflects the way in which they are organized in the model.

- 2 Expand or collapse the nodes located to the left of the objects.



- 3 You can modify a value by typing directly in the corresponding field in the Values pane.

In the example below, the model properties were changed by adding NEW to the model name:



- 4 Once you have completed your modifications, save and close the file.

Visualizing an XML file using an XML viewer

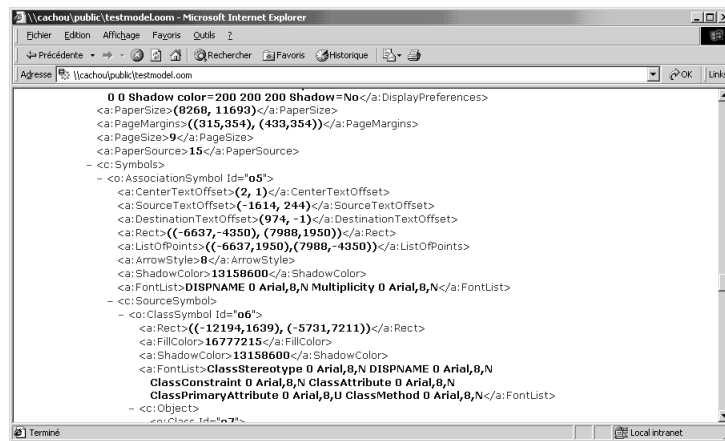
You can visualize an XML file using an XML viewer, such as some XML capable browsers used to explore the Internet.

You cannot modify the file in an XML viewer, only visualize its content.

❖ To visualize the content of an XML file in a browser:

- 1 Open the file in the browser.

The content of the XML file is displayed. The objects are displayed according to the hierarchy that reflects the way in which they are organized in the model.



- 2 Expand or collapse the nodes situated to the left of the objects to browse through the hierarchy of objects.
- 3 Close the browser when you have finished.

