SYBASE®

Working with Scripts

Sybase® PowerDesigner®

12.5

Windows

Part number: DC00425-01-1250-01
Last modified: April 2007

Copyright © 1991-2007 Sybase, Inc. and its subsidiaries. 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. and its subsidiaries.

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), 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, Advantage Database Server, Afaria, Answers Anywhere, AppModeler, APT Workbench, APT-Build, APT-Edit, APT-Execute, APT-Translator, APT-Library, ASEP, Avaki, Avaki (Arrow Design), Avaki Data Grid, AvantGo, 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, DataWindow .NET, DB-Library, dbQueue, Dejima, Dejima Direct, Developers Workbench, DirectConnect Anywhere, DirectConnect, Distribution Director, Dynamic Mobility Model, e-ADK, E-Anywhere, e-Biz Integrator, E-Whatever, EC Gateway, ECMAP, ECRTP, eFulfillment Accelerator, EII Plus, Electronic Case Management, Embedded SQL, EMS, Enterprise Application Studio, Enterprise Client/Server, Enterprise Connect, Enterprise Data Studio, Enterprise Manager, Enterprise Portal (logo), Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, eProcurement Accelerator, eremote, Everything Works Better When Everything Works Together, EWA, ExtendedAssist, Extended Systems, ExtendedView, Financial Fusion, Financial Fusion (and design), Financial Fusion Server, Formula One, Fusion Powered e-Finance, Fusion Powered Financial Destinations, Fusion Powered STP, Gateway Manager, GeoPoint, GlobalFIX, iAnywhere, iAnywhere Solutions, ImpactNow, Industry Warehouse Studio, InfoMaker, Information Anywhere, Information Everywhere, InformationConnect, InphoMatch, InstaHelp, Intelligent Self-Care, InternetBuilder, iremote, iScript, Jaguar CTS, jConnect for JDBC, KnowledgeBase, Logical Memory Manager, lrLite, M2M Anywhere, Mach Desktop, Mail Anywhere Studio, Mainframe Connect, Maintenance Express, Manage Anywhere Studio, MAP, M-Business Anywhere, M-Business Channel, M-Business Network, M-Business Suite, MDI Access Server, MDI Database Gateway, media.splash, Message Anywhere Server, MetaWorks, MethodSet, mFolio, Mirror Activator, ML Query, MobiCATS, Mobil 365, Mobileway, MySupport, Net-Gateway, Net-Library, New Era of Networks, Next Generation Learning, Next Generation Learning Studio, O DEVICE, OASiS, OASiS logo, ObjectConnect, ObjectCycle, OmniConnect, OmniQ, OmniSQL Access Module, OmniSQL Toolkit, OneBridge, 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, Pharma Anywhere, PhysicalArchitect, Pocket PowerBuilder, PocketBuilder, Power++, Power Through Knowledge, power.stop, PowerAMC, PowerBuilder, PowerBuilder Foundation Class Library, PowerDesigner, PowerDimensions, PowerDynamo, Powering the New Economy, PowerScript, PowerSite, PowerSocket, Powersoft, PowerStage, PowerStudio, PowerTips, Powersoft Portfolio, Powersoft Professional, PowerWare Desktop, PowerWare Enterprise, ProcessAnalyst, Pylon, Pylon Anywhere, Pylon Application Server, Pylon Conduit, Pylon PIM Server, Pylon Pro, QAnywhere, Rapport, Relational Beans, RepConnector, Report Workbench, Report-Execute, Replication Agent, Replication Driver, Replication Server, Replication Server Manager, Replication Toolkit, Resource Manager, RFID Anywhere, RW-DisplayLib, RW-Library, SAFE, SAFE/PRO, Sales Anywhere, Search Anywhere, SDF, Search Anywhere, Secure SQL Server, Secure SQL Toolset, Security Guardian, ShareSpool, Sharelink SKILS, smart.partners, smart.parts, smart.script, SOA Anywhere Trademark,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 365, Sybase Central, Sybase Client/Server Interfaces, Sybase Development Framework, Sybase Financial Server, Sybase Gateways, Sybase IQ, Sybase Learning Connection, Sybase MPP, SyberLearning LIVE, Sybase SQL Desktop, Sybase SQL Lifecycle, Sybase SQL Workgroup, Sybase Synergy Program, Sybase Virtual Server Architecture, Sybase User Workbench, SybaseWare, Syber Financial, SyberAssist, SybFlex, 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, TotalFix, TradeForce, Transact-SQL, Translation Toolkit, Turning Imagination Into Reality, UltraLite, UltraLite.NET, UNIBOM, Unilib, Uninull, Unisep, Unistring, URK Runtime Kit for UniCode, Unwired Accelerator, Unwired Orchestrator,Viafone, 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, XcelleNet, XP Server, XTNDConnect, and XTNDACCESS are trademarks of Sybase, Inc. or its subsidiaries.

All other trademarks are the property of their respective owners.

Contents

About This Manual

Subject

This book describes the PowerDesigner scripting environment. It shows you how to do the following:

- Access and manipulate PowerDesigner metamodel objects using scripts.
- Manipulate the repository using scripts.
- Manipulate reports using scripts.
- Manipulate metadata using scripts.
- Manipulate the workspace using scripts.
- Communicate with PowerDesigner using OLE Automation.
- Customize PowerDesigner menus using add-ins.
- Generate and reverse a database using scripts.

Audience

This book is for anyone who wants to use scripting in PowerDesigner. It requires an understanding of the PowerDesigner modeling environment together with an understanding of scripting languages. Some experience with programming will be helpful. For more information, see the Bibliography section at the end of this chapter.

Documentation primer

The PowerDesigner modeling environment supports several types of models:

- **Conceptual Data Model (CDM)** to model the overall logical structure of a data application, independent from any software or data storage structure considerations
- **Physical Data Model (PDM)** to model the overall physical structure of a database, taking into account DBMS software or data storage structure considerations
- **Object Oriented Model (OOM)** to model a software system using an object-oriented approach for Java or other object languages

- **Business Process Model (BPM)** to model the means by which one or more processes are accomplished in operating business practices
- **XML Model (XSM)** to model the structure of an XML file using a DTD or an XML schema
- **Requirements Model (RQM)** to list and document the customer needs that must be satisfied during a development process
- **Information Liquidity Model (ILM)** to model the replication of information from a source database to one or several remote databases using replication engines
- **Free Model (FEM)** to create any kind of chart diagram, in a context-free environment

This book only explains how to access and manipulate PowerDesigner metamodel objects using scripts. For information on other models or aspects of PowerDesigner, consult the following books:

General Features Guide To get familiar with the PowerDesigner interface before learning how to use any of the models.

Conceptual Data Model User's Guide To work with the CDM.

Physical Data Model User's Guide To work with the PDM.

Object Oriented Model User's Guide To work with the OOM.

XML Model User's Guide To work with the XSM.

Requirements Model User's Guide To work with the RQM.

Information Liquidity Model User's Guide To work with the ILM.

Reports User's Guide To create reports for any or all models.

Repository User's Guide To work in a multi-user environment using a central repository.

Typographic conventions

PowerDesigner documentation uses specific typefaces to help you readily identify specific items:

- `monospace text (normal and bold)`

  Used for: Code samples, commands, compiled functions and files, references to variables.

  Example: `declare user_defined...,` the `BeforeInsertTrigger` template.

- **bold text**

  Any new term.

  Example: A **shortcut** has a target object.

- SMALL CAPS

  Any key name.

  Example: Press the ENTER key.

Bibliography

Scripting Website: : http://msdn.microsoft.com/scripting/.

CHAPTER 1

Accessing Objects Using Scripts

About this chapter

This chapter explains the PowerDesigner metamodel objects and the structure of the Scripting Object Help file that details the PowerDesigner objects properties, collections and methods that you can use with scripting. It also explains how to use the editor to access the scripting environment.

Contents

Introducing Scripting in PowerDesigner

In critical size models or when dealing with several models at the same time, it can be sometimes very tedious to perform repetitive tasks on a series of objects or models, for example modifying objects using global rules, importing or generating new formats or even checking models.

Those operations can be eased using scripts. Indeed, Scripting is the only flexible way to let you access and edit the PowerDesigner wide range of models, diagrams and objects. It also allows you to reuse the same subroutines or functions many times in different situations.

Scripting is widely used in various PowerDesigner features. For example, when you want to:

- Create custom checks, event handlers, transformations, customs commands and custom popup menus, you use Profiles.
- Communicate with PowerDesigner from another application, you use OLE automation.
- Customize PowerDesigner menus by adding your own menu items, you use add-ins (Active X or XML file).
- Create VBscript macros and embed VBscript code inside a template, you use the Generation Template Language (GTL).

You can access PowerDesigner objects using any scripting language such as Java, VBScript or C# (C Sharp). However, the scripting language used to illustrate our examples in this manual is VBScript.

VBScript is a scripting language powered by Microsoft that you can use to write scripts for automating lots of diverse operations. PowerDesigner provides an integrated support for Microsoft VBScript so that you can write and run scripts to act over PowerDesigner metamodel objects in a development environment using **properties** and **methods**. Every PowerDesigner objects can be read and modified (creation, update or deletion).

☞ For more information on Microsoft VBScript, see the following Web address http://msdn.microsoft.com/scripting/default.htm?/scripting/vbscript/.

☞ For more information about the use of scripting to create custom checks, event handlers, transformations, custom commands and custom popup menus see the “Managing Profiles” chapter in the *Advanced User Documentation* .

☞ For more information about the use of scripting in OLE Automation, see the “Communicating With PowerDesigner Using OLE Automation” chapter.

☞ For more information about the use of scripting in the GTL, see “Using Macros” in the “Generation Reference Guide” chapter in the *Advanced User Documentation* .

☞ For more information about the use of scripting to create add-ins, see the “Customizing PowerDesigner Menus Using Add-Ins “ chapter.

Introducing PowerDesigner Metamodel Objects

PowerDesigner ships with a metamodel published in an Object Oriented Model (metamodel.oom) that illustrates how metadata interact in the software. All objects in the PowerDesigner metamodel have a name and a code. They correspond to the **public name** of the metadata. An HTML help file is also provided to allows you to find out which properties and methods can be used to drill down to a PowerDesigner object.

☞ For more information on metadata, see the “PowerDesigner Public Metamodel” chapter in the *Advanced User Documentation* .

PowerDesigner also provides a set of pre-written scripts that you can modify to meet your unique needs.

Scripting allows you to perform any kind of data manipulation but you can also insert and customize commands in the Tools menu that will allow you to automatically launch your own scripts.

Basic Concepts

When you use VBScript to access PowerDesigner objects, you can manipulate the following concepts :

- Objects
- Properties
- Collections
- Functions
- Constants
- Libraries

Objects

Objects correspond to any PowerDesigner objects. It can be:

- Design objects, such as tables, classes, processes or columns.
- Diagrams or symbols.
- Functional objects, such as the report or the repository.

An object belongs to a metaclass of the PowerDesigner metamodel.

Each object has properties, collections and methods that it inherits from its metaclass.

Root objects like models for example are created or retrieved using global methods. For more information, see "Global properties" on page 11.

Non root objects are created or retrieved using collections. For example, you create these objects using a Create method on collections and delete them using a Delete method on collections. For more information, see "Collections" on page 6.

You can browse the PowerDesigner metamodel to get information about the properties and collections available for each metaclass.

Example

```
'Variables are not typed in VBScript. You create them and the
'location where you use them determines what they are
' get the current active model
Dim mdl ' the current model
Set mdl = ActiveModel
```

Properties

A **property** is an elementary information available for the object. It can be the name, the code, the comment etc.

Example

```
'How to get a property value in a variable from table 'Customer
Dim Table_name
'Assuming MyTable is a variable that already contains a 'table
         object
Get the name of MyTable in Table_name variable
Table_name = MyTable.name
'Display MyTable name in output window
output MyTable.name
'How to change a property value : change value for name 'of
         MyTable
MyTable.name = 'new name'
```

Collections

A **collection** is a set of objects.

The model is the root object and the other objects can be reached by browsing the corresponding collection. The objects are grouped together within collections that can be compared to the category nodes appearing in the Browser tree view of the Workspace.

If an object CUSTOMER has a collection, it means the collection contains the list of objects with which the object CUSTOMER is in relation.

Some functions are available on collections. You can:

- Browse a collection
- Get the number of objects a collection contains
- Create a new object inside a collection, if it is a composition collection

Collections can be of the following types:

- Read-only collections are collections that can only be browsed
- Unordered collections are collections for which objects order in the list is not significant. For example the Relationships collection of a CDM Entity object is an unordered collection
- Ordered collections are collections for which object order is set by the user and must be respected. For example the Columns collection of the PDM Table object is an ordered collection

- Composition collections are collections for which objects belong to the collection owner. They are usually displayed in the Browser. Non composition collections can also be accessed using scripting and can be for example the list of business rules attached to a table or a class and displayed in the Rules tab of its property sheet or the list of objects displayed in the Dependencies tab of an object property sheet.

Read-only collections

Models (global collection for opened models) is an example of read-only collection.

The property and method available for read-only collections are the following:

Property or Method	Use
Count As Long	Retrieves the number of objects in collection
Item(idx As Long = 0) As BaseObject	Retrieves the item in collection for a given index. Item(0) is the first object
MetaCollection As BaseObject	Retrieves the MetaCollection object that defines this collection
Kind As Long	Retrieves the kind of objects the collection can contain. It returns a predefined constant such as cls_
Source As BaseObject	Retrieves the object that owns the collection

Example:

```
'How to get the number of open models and display it
'in the output window
output Models.count
```

Unordered collections

All methods and properties for read-only collections are also available for unordered collections.

Properties and methods available for unordered collections are the following:

Property or Method	Use
Add(obj As BaseObject)	Adds object as the last object of the collection
Remove(obj As BaseObject, delete As Boolean = False)	Removes the given object from collection and optionally delete the object
CreateNew(kind As Long = 0) As BaseObject	Creates an object of a given kind, and adds it at the end of collection. If no object kind is specified the value 0 is used which means that the Kind property of the collection will be used. See the Meta-model Objects Help file for restrictions on using this method
Clear(delete As Boolean = False)	Removes all objects from collection and optionally delete them

Example:

```
'remove table TEST from the active model
Set MyModel = ActiveModel
For each T in Mymodel.Tables
   If T.code = "TEST" then
       set MyTable = T
   End if
next
ActiveModel.Tables.Remove MyTable, true
```

Ordered collections

All methods and properties for read-only and unordered collections are also available for ordered collections.

Properties and methods available for ordered collections are the following:

Property or Method	Use
Insert(idx As Long = -1, obj As BaseObject)	Inserts objects in collection. If no index is provided, the index -1 is used, which means the object is simply added as the last object of the collection
RemoveAt(idx As Long = -1, delete As Boolean = False)	Removes object at given index from collection. If no index is provided the index -1 is used, which means the removed object is the last object in collection (if any). Optionally deletes the object
Move(source As Long, dest As Long)	Moves object from source index to destination index
CreateNewAt(idx As Long = -1, kind As Long = 0) As BaseObject	Creates an object of a given kind, and inserts it at given position. If no index is provided the index -1 is used which means the object is simply added as the last object of the collection. If no object kind is specified the value 0 is used which means that the Kind property will be used. See the Metamodel Objects Help file for restrictions on using this method

Example:

```
'Move first column in last position
'Assuming the variable MyTable contains a table
MyTable.Columns.move(0,-1)
```

Composition collections

Composition collections can be ordered or unordered.

All methods and properties for unordered collections are also available for unordered compositions.

Properties and methods available for unordered composition collections are the following:

Property or Method	Use
CreateNew(kind As Long = 0) As BaseObject	Creates an object of a given kind, and adds it at the end of collection. If no object kind is specified the value 0 is used, which means the Kind property of the collection will be used

All methods and properties for ordered collections are also available for ordered compositions.

All methods and properties for unordered compositions are also available for ordered compositions.

Properties and methods available for ordered composition collections are the following:

Property or Method	Use
CreateNewAt(idx As Long = -1, kind As Long = 0) As BaseObject	Creates an object of a given kind, and inserts it at a given position. If no index is provided the index -1 is used, which means the object is simply added as the last object of the collection. If no object kind is specified the value 0 is used which means that the Kind property of the collection will be used

These methods can be called with no object kind specified, but this is only possible when the collection is strongly typed. That is, the collection is designed to contain objects of a precise non-abstract object kind. In such cases, the Kind property of the collection corresponds to an instantiable class and the short description of the collection states the object kind name.

Example:

The Columns collection of a table is a composition collection as you can create columns from it. But the Columns collection of a key is not a composition collection as you cannot create objects (columns) from it, but only list them.

```
'Create a new table in a model
'Assuming the variable MyModel contains a PDM
'Declare a new variable object MyTable
Dim MyTable
'Create a new table in MyModel
Set MyTable = MyModel.Tables.Createnew

'Create a new column in a table
'Declare a new variable object MyColumn
Dim MyColumn
'Create a new column in MyTable in 3rd position
Set MyTable = MyTable.Columns.CreateNewAt(2)
' the column is created with a default name and code
```

Browsing the collections of a model
When you browse the collections of a model and want to retrieve its objects, be aware that you will also retrieve the shortcuts of objects of the same type.

Global properties

The available global properties can be gathered as follows:

Type	Global property	Use
Global accessor	ActiveModel As BaseObject	Retrieves the model, package, or diagram that corresponds to the active view
	ActivePackage As BaseObject	
	ActiveDiagram As BaseObject	
	ActiveSelection As ObjectSet	Read-only collection that retrieves the list of selected objects in the active diagram
	ActiveWorkspace As BaseObject	Retrieves the Application active Workspace
	MetaModel As BaseObject	Retrieves the Application MetaModel
	Models As ObjectSet	Read-only collection that lists opened models
	RepositoryConnection As BaseObject	Retrieves the current repository connection, which is the object that manages the connection to the repository server and then provides access to documents and objects stored under the repository
Execution mode	ValidationMode As Boolean	Enables or disables the validation mode (True/False).
	InteractiveMode As long	Manages the user interaction by displaying dialog boxes or not using the following constants (im_+Batch, +Dialog or +Abort).

Type	Global property	Use
Application	UserName As String	Retrieves the user login name
	Viewer As Boolean	Returns True if the running application is a Viewer version that has limited features
	Version As String	Returns the PowerDesigner version
OLE specific	ShowMode As	Checks or changes the visibility status of the main application window in the following way: ♦ It returns True if the application main window is visible and not minimized ♦ False otherwise
	Locked As Boolean	Can be set to True to ensure that the application continues to run after an OLE client disconnects otherwise the application closes

Example:

```
'Create a new table in a model
'Get the active model in MyModel variable
Set MyModel = ActiveModel
```

You can use two types of execution mode when running a script in the editor. A default value can be specified for each mode:

- Validation mode
- Interactive mode

Validation mode

The validation mode is enabled by default (set to True), but you may choose to temporarily disable it by setting it to False.

State	Constant	Code	Use
Enabled (default value)	True	ValidationMode = True	Each time you act over a PowerDesigner object, all internal PowerDesigner methods are invoked to check the validity of your actions. In case of a forbidden action, an error occurs. This mode is very useful for debugging but is necessarily performance consuming
Disabled	False	ValidationMode = False	You use it for performance reasons or because your algorithm temporarily requires an invalid state. However, be aware, that no validation rules such as name uniqueness or link object with missing extremities are applied to your model in this case

Example:

```
ValidationMode = true
```

Interactive mode

The interactive mode is Batch by default.

The interactive mode supports the following constants:

Constant	Code	Description
im_Batch	InteractiveMode = im_Batch	Never displays dialog boxes and always uses default values. You use it for Automation scripts that require no user interaction
im_Dialog	InteractiveMode = im_Dialog	Displays information and confirmation dialog boxes that require user interaction for the script to keep running
im_Abort	InteractiveMode = im_Abort	Never displays dialog boxes and aborts the script instead of using default values each time a dialog is encountered

Option Explicit statement

We recommend to use the Option Explicit statement to declare your variables in order to avoid confusion in code as this option is disabled by default in VBScript. You have to declare a variable before using this option.

Example:

```
Option Explicit
ValidationMode = True
InteractiveMode = im_Batch
' get the current active model
Dim mdl ' the current model
Set mdl = ActiveModel
```

Global functions

The following global functions are available:

Global functions	Use
CreateModel (modelkind As Long, filename As String = “”, flags As Long =omf_Default) As BaseObject	Creates a new model
CreateModelFromTemplate (filename As String, flags As Long =omf_Default) As BaseObject	Creates a new model using given model file as template
OpenModel (filename As String, flags As Long =omf_Default) As BaseObject	Opens an existing model (including V6 models)

Global functions	Use
Output (message As String = "")	Writes a message in the Script tab of the Output window of PowerDesigner main window
NewPoint (X As Long = 0, Y As Long = 0) As APoint	Creates a point to position a symbol
NewRect (Left As Long = 0, Top As Long = 0, Right As Long = 0, Bottom As Long = 0) As Arect	Creates a rectangle to manipulate symbols position
NewPtList () As PtList	Creates a list of points to position a link
NewGUID() As String	Creates a new Global Unique IDentifier (GUID). This new GUID is returned as a string without the usual surrounding "{" "}"
IsKindOf(childkind As Long, parentkind As Long) As Boolean	Returns True if childkind corresponds to a metaclass derived from the metaclass of kind parentkind, False otherwise
ExecuteCommand (cmd As String, Optional arglist As String, Optional mode As Long) As String	Opens an external application
Rtf2Ascii (rtf As String) As String	Removes RTF (Rich-Text-File) tags from an RTF formatted text
ConvertToUTF8 (InputFileName As String, OutputFileName As String)	Converts <InputFileName> file into UTF8 (8-bit Unicode Transformation Format, where byte order is specified by an initial Byte-Order Mark) and writes the result to the file <OutputFileName>. The two filenames must be different

Global functions	Use
ConvertToUTF16 (InputFileName As String, OutputFileName As String)	Converts <InputFileName> file into UTF16 (16-bit Unicode Transformation Format Little Endian, where byte order is specified by an initial Byte-Order Mark) and writes the result to the file <OutputFileName>. The two filenames must be different
EvaluateNamedPath (FileName As String, QueryIfUnknown As Boolean = True, FailOnError As Boolean = False) As String	Replaces a variable in a path by the corresponding named path
MapToNamedPath (FileName As String) As String	Replaces the path of a file by the corresponding named path
Progress(Key As String, InStatusBar Boolean = False) As BaseObject	Create or retrieve a given progress indicator
BeginTransaction()	Starts a new transaction
CancelTransaction()	Cancels the ongoing transaction
EndTransaction()	Commits the ongoing transaction

OpenModel(), CreateModel() and CreateModelFromTemplate flags

OpenModel, CreateModel and CreateModelFromTemplate functions use the following global constants:

Constant	Use
Omf_Default	Default behavior for OpenModel/CreateModel
Omf_DontOpenView	Does not open default diagram view for OpenModel/CreateModel/ CreateModelFromTemplate
Omf_QueryType	For CreateModel ONLY: Forces querying initial diagram type
Omf_NewFileLock	For CreateModel ONLY: Creates and locks corresponding file
Omf_Hidden	Does not let the model appear in the workspace for OpenModel/CreateModel/CreateModelFromTemplate

Command execution modes

Command execution modes use the following global constants:

Constant	Use
cmd _ShellExec	Default behavior: lets MS-Windows shell execute the command
cmd _PipeOutput	Redirects the command output to the General tab of PowerDesigner Output window
cmd _PipeResult	Captures the whole command output to the returned string
cmd_InternalScript	Indicates that the first parameter of the Execute Command is a VBScript file to be executed as an internal script rather than letting the system run the application associated with the file type

Example:

```
'Create a new model and print its name in output window
CreateModel(PDOOm.cls_Model, "C:\Temp\
        Test.oom|Language=Java|Diagram=SequenceDiagram")
Output ActiveModel.name
```

Global constants

The following global constants are available:

Global constants	Use
Version As String	Returns the application version string
HomeDirectory As String	Returns the application home directory string
RegistryHome As String	Returns the application registry home path string
cls_... As Long	Identifies the class of an object. This value is used when you need to specify an object kind in creation method for example. This value is also used by IsKindOf method available on all PowerDesigner objects

Classes Ids constants

Constants are unique within a model and are used to identify object classes in each library. All classes Ids start with "cls_" followed by the public name of the object. For example cls_Process identifies the Process object class using the public name of the object.

However, when dealing with several models, some constants may be common, for example cls_Package.

To avoid confusion in code, you must prefix the constant name with the name of the module, for example PdOOM.cls_Package. Same, when you want to create a model, you need to prefix the cls_Model constant with the name of the module.

IsKindOf method

You can use the IsKindOf (ByVal Kind As Long) As Boolean method together with a class constant in order to check if an object inherits from a given class kind.

Example:

You can have a script with a loop that browses the Classifiers collection of an OOM and wants to check the type of encountered objects (in this case interfaces or classes) in order to perform different actions according to their type.

```
'Assuming the Activemodel is an OOM model
For each c in Activemodel.Classifiers
If c.IsKindOf(cls_Class) then
Output "Class " & c.name
ElsIf c.IsKindOf(cls_Interface) then
Output "Interface" & c.name
End If
Next
```

Example:

All the collections under a model can contain objects of a certain type but also shortcuts for objects of the same type. You can have a script with a loop that browses the Tables collection of a PDM and want to check the type of encountered objects (in this case tables or shortcuts) in order to perform different actions according to their type.

```
For each t in Activemodel.Tables
If t.IsKindOf(cls_Table) then
Output t.name
End If
Next
```

Libraries

The following libraries are available. Each of them (apart from PdCommon, PdRMG and PdWSP) is equivalent to a model type:

Library name	Corresponding model
PdCDM	Conceptual Data Model
PdPDM	Physical Data Model
PdOOM	Object Oriented Model
PdBPM	Business Process Model
PdXSM	XML Model
PdRQM	Requirements Model
PdILM	Information Liquidity Model
PdFRM	Free Model
PdRMG	Repository
PdWSP	Workspace
PdCommon	Objects common to all models

PdCommon does not correspond to a particular model, it gathers all objects shared among two or more models. For example, business rules are defined in this library.

It also defines the abstract classes of the model, for example, BaseObject is defined in diagram Common Abstract Objects in the Objects package of PdCommon.

Models are linked to PdCommon by generalization links indicating that each

model inherits common objects from the PdCommon library.

For each library, you can browse a list of:

- **Abstract classes** (located in the Abstract Classes expanded node). They are general classes that are used to factorize attributes and behaviors. They are not visible in PowerDesigner. Instantiable classes inherit from abstract classes
- **Instantiable classes** (located directly at the root of each library node). They are specific classes that correspond to interface objects, they have proper attributes like name or code, and they also inherit attributes and behaviors from abstract classes via generalization links. For example, NamedObject is the common class for most PowerDesigner design objects, it stores standard attributes like name, code, comment, annotation, and description

☞ For more information on PowerDesigner libraries, see the "PowerDesigner Public Metamodel" chapter in the *Advanced User Documentation.*

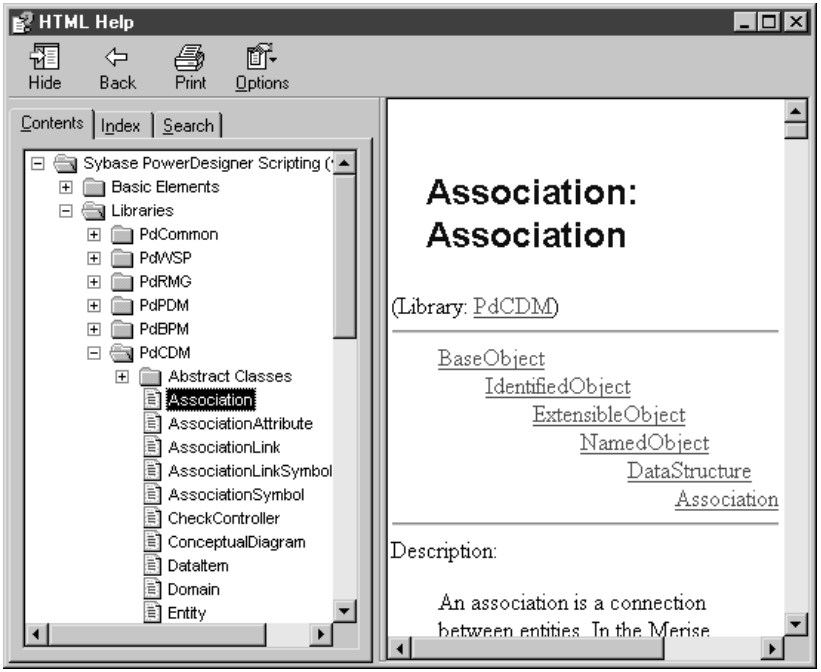
Understanding the Metamodel Objects Help File

PowerDesigner provides a compiled HTML help file that you can open from the Help ➤ Metamodel Objects Help command or from the Edit/Run Script editor dialog box. This reference guide is intended to help you get familiar with the PowerDesigner objects properties, collections and methods that you can use in scripting.

☞ For more information on the Edit/Run Script editor, see the ""Using the Edit/Run Script Editor" on page 23" section.

Metamodel Objects Help file structure

The Metamodel Objects Help file is composed of two distinct parts: the node tree view displayed on the left hand side to navigate through the objects hierarchy and their corresponding description displayed to the right of the tree view:



You can expand the following nodes from the tree view:

Nodes	What you can find...

Nodes	What you can find...
Basic Elements	General information on: Collections per type (read-only, ordered and unordered) Structured Types (points, rectangles, lists of points) Global properties, constants and functions
Libraries	PdCommon that contains: Basic object classes library used by all modules, for example File Object and Business Rules, or by at least two modules such as the Organization Unit used in the OOM and the BPM PdRMG that contains Repository object classes library PdWSP that contains Workspace object classes library Object classes libraries per module (in PdCDM, PdOOM, PdBPM, PdPDM, PdXSM, PdRQM, PdILM and PdFRM)
Appendix	Hierarchical representation of the PowerDesigner metamodel List of constants used to identify objects of each library

☞ For more information on collections, see the ""Collections" on page 6" section.

☞ For more information on global properties, constants and functions, see the ""Global properties" on page 11", ""Global constants" on page 17", ""Global functions" on page 14" section.

☞ For more information on libraries, see the ""Libraries" on page 19" section.

Metamodel Objects Help file content

The scripting objects provided by PowerDesigner correspond to the design objects (tables, entities, classes, processes etc.) that appear in the user interface.

For each PowerDesigner object you can browse a list of:

- Properties (Example: Name, Data Type, Transport)
- Collections (Example: Symbols, Columns of a table)
- Methods (Example: Delete (), UpadateNamingOpts())

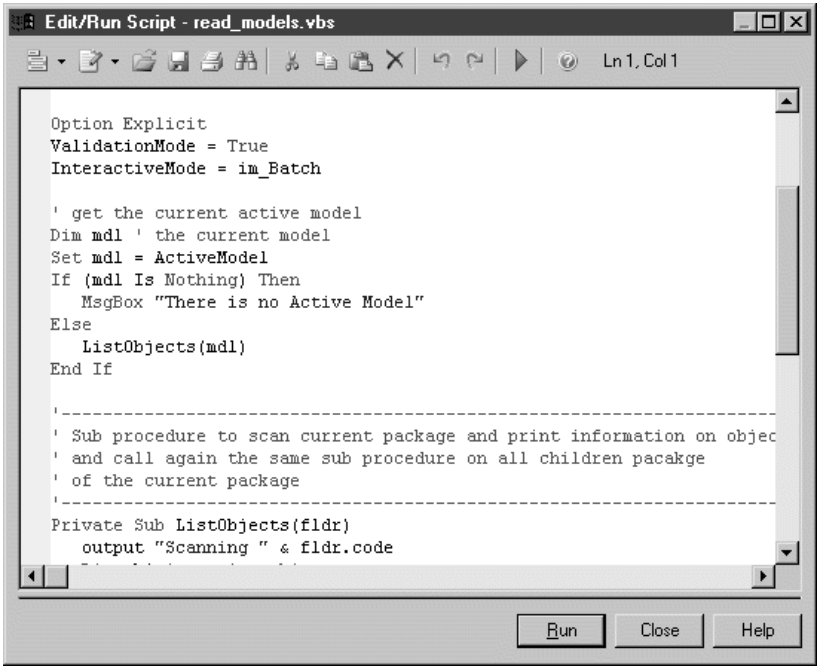
The nature of each collection is indicated: read-only, ordered, unordered, or composition.

Using the Edit/Run Script Editor

The Edit/Run Script editor runs in the PowerDesigner environment and provides access to the scripting environment. You open it from the Tools ➤ Execute Commands menu. It is available whatever the type of the active model and also when no model is active.

You can see the date and time when the script begins and ends in the Script tab of the Output window located in the lower part of the PowerDesigner main window, if you have used the Output global function.

The Edit/Run Script editor looks like the following:



The following tools and keyboard shortcuts are specific to the Edit/Run Script editor toolbar:

Tool	Description	Keyboard shortcut
	Editor Menu Note: When you use the Find feature, the parameter "Regular Expression" allows the use of wildcards in the search expression. For more information, see "Finding body of text using regular expressions" in the "Using the PowerDesigner interface" chapter	SHIFT + F11
	Edit With. Opens the previously defined default editor or allows you to select another editor if you click the down arrow beside this tool	CTRL + E
	Run. Executes the current script	F5
	Metamodel Objects Help provided to allow you to find out which properties and methods can be used to drill down to a PowerDesigner object	CTRL + F1

☞ For more information on defining a default editor, see "Defining a text editor" in the "Using the PowerDesigner Interface" chapter.

Script bookmarks

In the Edit/Run Script editor window, you can add and remove bookmarks at specific points in the code and then navigate forwards or backwards from bookmark to bookmark:

Keyboard shortcut	Description
CTRL + F2	Adds a new bookmark. A blue bookmark box is displayed. If you repeat this action from the same position, the bookmark is deleted and the blue marker disappears
F2	Jumps to bookmark
SHIFT + F2	Jumps to previous bookmark

Visual Basic

If you have Visual Basic (VB) installed on your machine, you can use the VB interface for your script writing in order to have access to the VB IntelliSense feature that checks all the standard methods and properties that you invoke and suggests the valid alternatives ones that you can choose in order to correct the code. However the PowerDesigner Edit/Run Script editor automatically recognizes VBScript keywords.

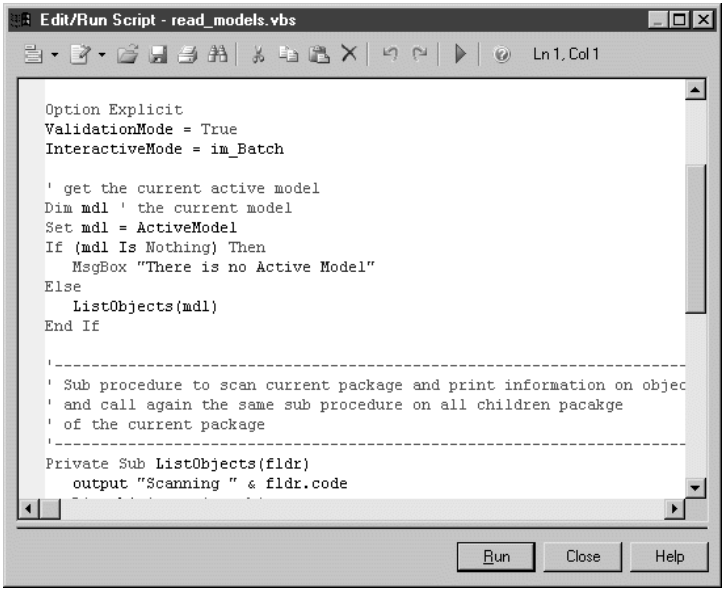
The Edit/Run Script editor lets you:

- Create a script
- Modify a script
- Save a script
- Run a script
- Use a sample script

Creating a VBScript file

❖ To create a VBScript file

1. Select Tools ➤ Execute Commands ➤ Edit/Run Script to display the Edit/Run Script dialog box.
2. Type the script instructions directly in the script editor window.



The script syntax is displayed as in Visual Basic.

☞ For more information on VB syntax, see the *Microsoft Visual Basic* documentation

Modifying a VBScript file

❖ To modify a VBScript file

1. Open the Edit/Run Script editor.
2. Click the Open tool.

   A standard dialog box opens.
3. Select a VBScript file (.VBS) and click Open.

   The VBScript file opens in the Edit/Run Script editor window. You can then modify it.

> **Inserting a script file in another script**
> You can insert a script file in a current script using the Insert command in the Editor Menu. The script will be inserted at the cursor position.

Saving a VBScript file

❖ To save a VBScript file

1. Open the Edit/Run Script editor.
2. Type the script instructions directly in the script editor window.
3. Click the Editor Menu tool and select Save from the list.

   or

   Click the Save tool.

   A standard dialog box opens if your VBScript file has never been saved before.
4. Browse to the directory where you want to save the script file.
5. Type a name for the script file and click Save.

It is strongly recommended to save your model and your script file before executing it.

Running a VBScript file

❖ To run a VBScript file

1. Open a script and click the Run tool or the Run button.

The script is executed and the Output window located in the lower part of the PowerDesigner main window shows the execution progress if you have used the Output global function that lets you display execution progress and errors in the Script tab.

If a compilation error occurs, a message box is displayed to inform you of the kind of error. A brief description error also is displayed in the Result pane of the Edit/Run Script dialog box and the cursor is set at the error position.

The Edit/Run Script editor supports multiple levels of Undo and Redo commands. However, if you run a script that modifies objects in several models, you must use the Undo or Redo commands in each of the models called by the script.

Catching errors
In order to avoid application abortions, you can catch errors using the On Error Resume Next statement. But you cannot catch errors with this statement when you use the im_Abort interactive mode.

You can also insert and customize commands in the Tools menu that will allow you to automatically launch your own scripts.

☞ For more information on customizing commands, see the “Using customized commands” section in the “Customizing PowerDesigner Menus Using Add-Ins” chapter.

Using VBScript file samples

PowerDesigner ships with a set of script samples, that you can use as a basis to create your own scripts. They are located in the VB Scripts folder of the PowerDesigner installation directory.

These scripts are intended to show you a range of the type of actions you can do over PowerDesigner objects using VBScript and also to help you in the code writing of your own scripts as you can easily copy/paste some code pieces from the sample into your script.

It is always recommended to make a backup copy of the sample file for it to remain intact.

Model scan sample

The following example illustrates a script with a loop that browses a model and its sub-packages to display objects information:

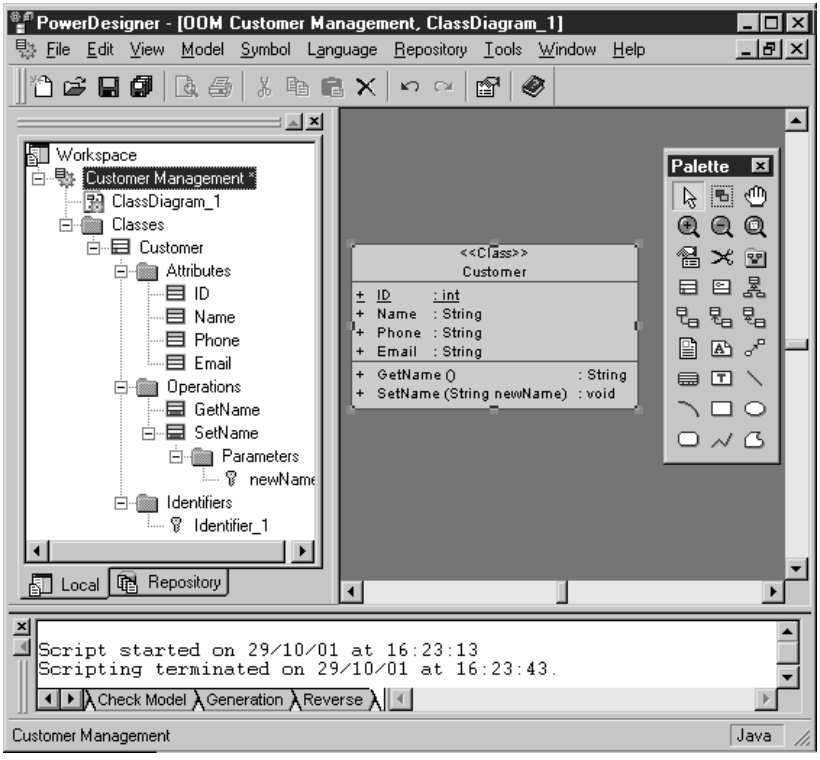
```
' Scan CDM Model and display objects information
' going down each package
Option Explicit
ValidationMode = True
InteractiveMode = im_Batch
' get the current active model
Dim mdl ' the current model
Set mdl = ActiveModel
If (mdl Is Nothing) Then
   MsgBox "There is no Active Model"
Else
   Dim fldr
   Set Fldr = ActiveDiagram.Parent
   ListObjects(fldr)
End If
' Sub procedure to scan current package and print information on
         objects from current package
' and call again the same sub procedure on all children package
 ' of the current package
Private Sub ListObjects(fldr)
   output "Scanning " & fldr.code
   Dim obj ' running object
   For Each obj In fldr.children
      ' Calling sub procedure to print out information on the
         object
      DescribeObject obj
   Next
   ' go into the sub-packages
   Dim f ' running folder
   For Each f In fldr.Packages
      'calling sub procedure to scan children package
      ListObjects f
   Next
End Sub
' Sub procedure to print information on current object in output
Private Sub DescribeObject(CurrentObject)
   if CurrentObject.ClassName ="Association-Class link" then
         exit sub
   'output "Found "+CurrentObject.ClassName
   output "Found "+CurrentObject.ClassName+"
         """+CurrentObject.Name+""", Created by
         "+CurrentObject.Creator+" On
         "+Cstr(CurrentObject.CreationDate)
   End Sub
```

Model creation sample

The following example illustrates a script that creates a new OOM model:

```
Option Explicit
' Initialization
' Set interactive mode to Batch
InteractiveMode = im_Batch
' Main function
' Create an OOM model with a class diagram
Dim Model
Set model = CreateModel(PdOOM.cls_Model,
         "|Diagram=ClassDiagram")
model.Name = "Customer Management"
model.Code = "CustomerManagement"
' Get the class diagram
Dim diagram
Set diagram = model.ClassDiagrams.Item(0)
' Create classes
CreateClasses model, diagram
' Create classes function
Function CreateClasses(model, diagram)
   ' Create a class
   Dim cls
   Set cls = model.CreateObject(PdOOM.cls_Class)
   cls.Name = "Customer"
   cls.Code = "Customer"
   cls.Comment = "Customer class"
   cls.Stereotype = "Class"
   cls.Description = "The customer class defines the attributes
         and behaviors of a customer."
   ' Create attributes
   CreateAttributes cls
   ' Create methods
   CreateOperations cls
   ' Create a symbol for the class
   Dim sym
   Set sym = diagram.AttachObject(cls)
   CreateClasses = True
End Function
' Create attributes function
Function CreateAttributes(cls)
   Dim attr
   Set attr = cls.CreateObject(PdOOM.cls_Attribute)
   attr.Name = "ID"
   attr.Code = "ID"
   attr.DataType = "int"
   attr.Persistent = True
   attr.PersistentCode = "ID"
   attr.PersistentDataType = "I"
   attr.PrimaryIdentifier = True
   Set attr = cls.CreateObject(PdOOM.cls_Attribute)
   attr.Name = "Name"
   attr.Code = "Name"
   attr.DataType = "String"
   attr.Persistent = True
   attr.PersistentCode = "NAME"
   attr.PersistentDataType = "A30"
   Set attr = cls.CreateObject(PdOOM.cls_Attribute)
   attr.Name = "Phone"
   attr.Code = "Phone"
   attr.DataType = "String"
   attr.Persistent = True
   attr.PersistentCode = "PHONE"
   attr.PersistentDataType = "A20"
```

The previous script gives the following result in the interface:



CHAPTER 2

Getting Started with Objects Manipulation Using Scripts

About this chapter

This chapter gives you some examples to help you get started with some generic manipulations on PowerDesigner objects that you can perform using scripting.

Contents

Creating a Model Using Scripts

You create a model using the CreateModel (modelkind As Long, filename As String = “”, flags As Long =omf_Default) As BaseObject global function together with the cls_Model constant prefixed with the Module name to identify the type of model you want to create.

Note that additional arguments may be specified in the filename parameter depending on the type of model (Language, DBMS, Copy, Diagram). The diagram argument uses the public name, but the localized name (the one in the Target selection dialog box) is also accepted. However, it is not recommended to use the localized name as your script will only work for the localized version of PowerDesigner.

Example

```
Option Explicit
' Call the CreateModel global function with the following
        parameters:
'  - The model kind is an Object Oriented Model (PdOOM.Cls_
        Model)
'  - The Language is enforced to be Analysis
'  - The first diagram will be a class diagram
'  - The language definition (for Analysis) is copied inside the
        model
'  - The first diagram will not be opened in a window
'  - The new created model will not appear in the workspace
Dim NewModel
set NewModel = CreateModel(PdOOM.Cls_Model,
         "Language=Analysis|Diagram=ClassDiagram|Copy", omf_
         DontOpenView Or omf_Hidden)
If NewModel is Nothing then
   msgbox "Fail to create UML Model", vbOkOnly, "Error"    '
         Display an error message box
Else
   output "The UML model has been successfully created"    '
         Display a message in the application output window
' Initialize model name and code
   NewModel.Name = "Sample Model"
   NewModel.Code = "Sample"
' Save the new model in a file
   NewModel.Save "c:\temp\MySampleModel.oom"
' Close the model
   NewModel.Close
' Release last reference to the model object to free memory
   Set NewModel = Nothing
End If
```

Opening a Model Using Scripts

You open a model using the OpenModel (filename As String, flags As Long =omf_Default) As BaseObject global function.

Example

```
Option Explicit
' Call the OpenModel global function with the following
        parameters:
'  - The model file name
'  - The default diagram will not be opened in a window
'  - The opened model will not appear in the workspace
Dim ExistingModel, FileName
FileName = "c:\temp\MySampleModel.oom"
On Error Resume Next          ' Avoid generic scripting error
        message like 'Invalid File Name
   Set ExistingModel = OpenModel(FileName, omf_DontOpenView Or
        omf_Hidden)
On Error Goto 0               ' Restore runtime error detection
If ExistingModel is nothing then
   msgbox "Fail to open UML Model:" + vbCrLf + FileName,
        vbOkOnly, "Error"    ' Display an error message box
Else
   output "The UML model has been successfully opened"    '
        Display a message in the application output window
End If
```

Creating an Object Using Scripts

It is recommended to create an object directly from the collection to which it belongs in order to directly obtain a valid state for the object. When you do so, you only create the object but not its graphical symbol.

You can also use the following method: CreateObject(ByVal Kind As Long, ByVal ParentCol As String = “”, ByVal Pos As Long = -1, ByVal Init As Boolean = -1) As BaseObject

Creating an object in a model

```
If not ExistingModel is Nothing Then
' Call the CreateNew() method on the collection that owns the
        object
   Dim MyClass
   Set MyClass = ExistingModel.Classes.CreateNew()
   If MyClass is nothing Then
      msgbox "Fail to create a class", vbOkOnly, "Error"    '
        Display an error message box
   Else
      output "The class objects has been successfully created"
        ' Display a message in the application output window
     ' Initialize its name and code using a specific method
      ' that ensures naming conventions (Uppercase or lowercase
        constraints,
      ' invalid characters...) are respected and that the name
        and code
       ' are unique inside the model
      MyClass.SetNameAndCode "Customer", "cust"
      ' Initialize other properties directly
      MyClass.Comment = "Created by script"
      MyClass.Stereotype = "MyStereotype"
      MyClass.Final = true
      ' Create an attribute inside the class
      Dim MyAttr
      Set MyAttr = MyClass.Attributes.CreateNew()
      If not MyAttr is nothing Then
         output "The class attribute has been successfully
         created"
         MyAttr.SetNameAndCode "Name", "custName"
         MyAttr.DataType = "String"
         Set MyAttr = Nothing
      End If
      ' Reset the variable in order to avoid memory leaks
      Set MyClass = Nothing
   End If
End If
```

Creating an object in a package

```
If not ExistingModel is Nothing Then
   ' Create a package first
   Dim MyPckg
   Set MyPckg = ExistingModel.Packages.CreateNew()
   If not MyPckg is Nothing then
      output "The package has been successfully created"
      MyPckg.SetNameAndCode "All interfaces", "intf"
      ' Create an interface object inside the package
      Dim MyIntf
      Set MyIntf = MyPckg.Interfaces.CreateNew()
      If not MyIntf is Nothing then
         output "The interface object has been successfully
         created inside the package"
         MyIntf.SetNameAndCode "Customer Interface", "custIntf"
         Set MyIntf = Nothing
      End If
      Set MyPckg = Nothing
   End If
End If
```

Creating a Symbol Using Scripts

You create the associated symbol of an object by attaching it to the active diagram using the following method: AttachObject(ByVal Obj As BaseObject) As BaseObject.

Example

```
set symbol1 = ActiveDiagram.AttachObject(entity1)
```

Other uses
The AttachObject method can also be used to create a graphical synonym or a shortcut. For more information, see sections on graphical synonym and shortcut creation.

Displaying Objects Symbol in a Diagram Using Scripts

You can display objects symbol in a diagram using the following methods:

- AttachObject(ByVal Obj As BaseObject) As BaseObject to create a symbol for a non-link object
- AttachLinkObject(ByVal Link As BaseObject, ByVal Sym1 As BaseObject = NULL, ByVal Sym2 As BaseObject = NULL) As BaseObject to create a symbol for a link object
- AttachAllObjects() As Boolean to create a symbol for each object in package which can be displayed in current diagram

Example

```
If not ExistingModel Is Nothing and not MyRealization Is Nothing
      Then
   ' Symbols are specific kind of objects that can be
      manipulated by script
   ' We are now going to display the class, interface and
      realization in the
   ' main diagram of the model and customize their presentation
   ' Retrieve main diagram
   Dim MyDiag
   Set MyDiag = ExistingModel.DefaultDiagram
   If not MyDiag is Nothing and MyDiag.IsKindOf(PdOOM.Cls_
      ClassDiagram) Then
      ' Display the class, interface shortcut and realization
         link in the diagram
      ' using default positions and display preferences
      Dim MyClassSym, MyIntfSym, MyRlzsSym
      Set MyClassSym = MyDiag.AttachObject(FoundClass)
      Set MyIntfSym  = MyDiag.AttachObject(IntfShct)
      Set MyRlzsSym  = MyDiag.AttachLinkObject(MyRealization,
         MyClassSym, MyIntfSym)
      If not MyRlzsSym is Nothing Then
         output "Objects have been successfully displayed in
         diagram"
      End If
      ' Another way to do the same is the use of
         AttachAllObjects() method:
      ' MyDiag.AttachAllObjects
      ' Changes class symbol format
      If not MyClassSym is nothing Then
         MyClassSym.BrushStyle = 1 ' Solid background (no
         gradient)
         MyClassSym.FillColor = RGB(255, 126, 126) ' Red
         background color
         MyClassSym.LineColor = RGB(0, 0, 0) ' Black line color
         MyClassSym.LineWidth = 2 ' Double line width
         Dim Fonts
         Fonts = "ClassStereotype " + CStr(RGB(50, 50, 126)) + "
         Arial,8,I"
         Fonts = Fonts + vbCrLf + "DISPNAME " + CStr(RGB(50, 50,
         50)) + " Arial,12,B"
         Fonts = Fonts + vbCrLf + "ClassAttribute " +
         CStr(RGB(150, 0, 0)) + " Arial,8,N"
         MyClassSym.FontList = Fonts ' Change font list
      End If
      ' Changes interface symbol position
      If not MyIntfSym is nothing Then
         Dim IntfPos
         Set IntfPos = MyIntfSym.Position
         If not IntfPos is Nothing Then
            IntfPos.x = IntfPos.x + 5000
            IntfPos.y = IntfPos.y + 5000
            MyIntfSym.Position = IntfPos
            Set IntfPos = Nothing
         End If
      End If
      ' Changes the link symbol corners
      If not MyRlzsSym is Nothing Then
         Dim CornerList, Point1, Point2
         Set CornerList = MyRlzsSym.ListOfPoints
         Set Point1 = CornerList.Item(0)
         Set Point2 = CornerList.Item(1)
```

Positioning a Symbol Next to Another Using Scripts

You position a symbol next to another using the X and Y (respectively Abscissa and Ordinate) points, together with a combination of method (Position As Apoint) and function (NewPoint(X As Long = 0, Y As Long = 0) As Apoint)).

Example

```
Dim diag
Set diag = ActiveDiagram
Dim sym1, sym2
Set sym1 = diag.Symbols.Item(0)
Set sym2 = diag.Symbols.Item(1)
X1 = sym1.Position.X
Y1 = sym1.Position.Y
' Move symbols next to each other using a fixed arbitrary space
sym2.Position = NewPoint(X1+5000, Y1)
' Move symbols for them to be adjacent
sym2.Position = NewPoint(X1 + (sym1.Size.X+sym2.Size.X)/2, Y1)
```

Deleting an Object from a Model Using Scripts

You delete an object from a model using the Delete As Boolean method.

Example

```
If not ExistingModel is Nothing Then
   ' Create another class first
   Dim MyClassToDelete
   Set MyClassToDelete = ExistingModel.Packages.CreateNew()
   If not MyClassToDelete is Nothing then
      output "The second class has been successfully created"
      ' Just call Delete method to delete the object
      ' This will remove the object from the collection of model
         classes
      MyClassToDelete.Delete
      ' The object is still alive but it has notified all other
      ' objects of its deletion. It is no more associated with
         other objects.
      ' Its status is deleted
      If MyClassToDelete.IsDeleted() Then
         output "The second class has been successfully deleted"
      End If
      ' The reset of the VbScript variable will release the last
      ' reference to this object abd provoke the physical
         destruction
      ' and free the memory
      Set MyClassToDelete = Nothing
   End If
End If
```

Retrieving an Object in a Model Using Scripts

The following example illustrates how you can retrieve an object by its code in the model

Example

```
‘ Call a function that is implemented just after in the script

Dim FoundIntf, FoundClass

Set FoundIntf = RetrieveByCode(ExistingModel, PDOOM.Cls_Interface, “custIntf”)

Set FoundClass = RetrieveByCode(ExistingModel, PDOOM.Cls_Class, “cust”)

If (not FoundIntf is nothing) and (not FoundClass is Nothing) Then

output “The class and interface objects have been successfully retrieved by their code”

End If

‘ Implement a method that retrieve an object by code

‘ The first parameter is the root folder on which the research begins

‘ The second parameter is the kind of object we are looking for

‘ The third parameter is the code of the object we are looking for

Function RetrieveByCode(RootObject, ObjectKind, CodeValue)

‘ Test root parameter

If RootObject is nothing Then

Exit Function ‘ Root object is not defined

End If

If RootObject.IsShortcut() Then

Exit Function ‘ Root object is a shortcut

End If

If not RootObject.IsKindOf(Cls_BaseFolder) Then

Exit Function ‘ Root object is not a folder

End If

‘ Loop on all objects in folder

Dim SubObject
```

```
For Each SubObject in RootObject.Children
If SubObject.IsKindOf(ObjectKind) and SubObject.Code = CodeValue Then
Set RetrieveByCode = SubObject ‘ Initialize return value
Set SubObject = Nothing
Exit Function
End If
Next
Set SubObject = Nothing
‘ Recursive call on sub-folders
Dim SubFolder
For Each SubFolder in RootObject.CompositeObjects
If not SubFolder.IsShortcut() Then
Dim Found
Set Found = RetrieveByCode(SubFolder, ObjectKind, CodeValue)
If not Found Is Nothing Then
Set RetrieveByCode = Found ‘ Initialize return parameter
Set Found = Nothing
Set SubFolder = Nothing
Exit Function
End If
End If
Next
Set SubFolder = Nothing
End Function
```

Creating a Shortcut in a Model Using Scripts

You create a shortcut in a model using the CreateShortcut(ByVal NewPackage As BaseObject, ByVal ParentCol As String = “”) As BaseObject method.

Example

```
' We want to reuse at the model level the interface defined in
        the package
' To do that, we need to create a shortcut of the interface at
        the model level
Dim IntfShct
If not FoundIntf is Nothing and not ExistingModel Is Nothing
        Then
   ' Call the CreateShortcut() method and specify the model
    ' for the package where we want to create the shortcut
   Set IntfShct = FoundIntf.CreateShortcut(ExistingModel)
   If not IntfShct is nothing then
      output "The interface shortcut has been successfully
        created"
   End If
   End If
```

Creating a Link Object Using Scripts

You create a link object using the CreateNew(kind As Long = 0) As BaseObject method, then you have to declare its ends.

Example

```
Dim MyRealization
If (not ExistingModel Is Nothing) and (not FoundClass Is
        Nothing) and (not IntfShct is Nothing) Then
   ' We are now going to create a realization link between the
        class and the interface
   ' The link is an object like others with two mandatory
        attributes: Object1 and Object2
   ' For oriented links, Object1 is the source and Object2 is
        the destination
   Set MyRealization = ExistingModel.Realizations.CreateNew()
   If not MyRealization is Nothing then
      output "The realization link has been successfully
        created"
      ' Initialize both extremities
      Set MyRealization.Object1 = FoundClass
      Set MyRealization.Object2 = IntfShct
      ' Initialize Name and Code
      MyRealization.SetNameAndCode "Realize Main interface",
        "Main"
   End If
End If
```

Browsing a Collection Using Scripts

All collections can be iterated through the usual “For Each variable In collection” construction.

This loop starts with “For each <variable> in <collection>” and ends with “Next”.

The loop is iterated on each object of the collection. The object is available in <variable>.

Example

```
'How to browse the collection of tables available on a model
Set MyModel = ActiveModel
'Assuming MyModel is a variable containing a PDM object.
For each T in MyModel.Tables
   'Variable T now contains a table from Tables collection of
         the model
   Output T.name
Next
```

Manipulating Objects in a Collection Using Scripts

In the following example, we are going to manipulate objects in collections by creating business rule objects and attaching them to a class object. To do so, we :

- Create the business rule objects
- Initialize their attributes
- Retrieve the first object in the class attributes collection
- Add the created rules at the beginning and at the end of the attached rules collection
- Move a rule at the end of the the attached rules collection
- Remove a rule from the attached rules collection

Example

```
If (not ExistingModel Is Nothing) and (not FoundClass Is
        Nothing) Then
   ' We are going to create business rule objects and attached
        them to the class
   ' Create first the business rule objects
   Dim Rule1, Rule2
   Set Rule1 = ExistingModel.BusinessRules.CreateNew()
   Set Rule2 = ExistingModel.BusinessRules.CreateNew()
   If (not Rule1 is Nothing) And (not Rule2 Is Nothing) Then
      output "Business Rule objects have been successfully
         created"
      ' Initialize rule attributes
      Rule1.SetNameAndCode "Mandatory Name", "mandatoryName"
      Rule1.ServerExpression = "The Name attribute cannot be
         empty"
      Rule2.SetNameAndCode "Unique Name", "uniqueName"
      Rule2.ServerExpression = "The Name attribute must be
         unique"
      ' Retrieve the first object in the class attributes
         collection
      Dim FirstAttr, AttrColl
      Set AttrColl = FoundClass.Attributes
      If not AttrColl is Nothing Then
         If not AttrColl.Count = 0 then
            Set FirstAttr = AttrColl.Item(0)
         End If
      End If
      Set AttrColl = Nothing
      If not FirstAttr is Nothing Then
         output "First class attribute successfully retrieved
         from collection"
         ' Add Rule1 at end of attached rules collection
         FirstAttr.AttachedRules.Add Rule1
         ' Add Rule2 at the beginning of attached rules
         collection
         FirstAttr.AttachedRules.Insert 0, Rule2
         ' Move Rule2 at end of collection
         FirstAttr.AttachedRules.Move 1, 0
         ' Remove Rule1 from collection
         FirstAttr.AttachedRules.RemoveAt 0
         Set FirstAttr = Nothing
      End If
   End If
  Set Rule1 = Nothing
   Set Rule2 = Nothing
End If
```

Extending the Metamodel Using Scripts

When you import a file using scripts, you can import as extended attributes or extended collections some properties that may not correspond to standard attributes. In the following example, we:

- Create a new extended model definition
- Initialize model extension attributes
- Define a new stereotype for the Class metaclass in the profile section
- Define an extended attribute for this stereotype

Example

```
If not ExistingModel Is Nothing Then
   ' Creating a new extended model definition
   Dim ModelExtension
   Set ModelExtension =
         ExistingModel.ExtendedModelDefinitions.CreateNew()
   If not ModelExtension is Nothing Then
      output "Model extension successfully created in model"
      ' Initialize model extension attributes
      ModelExtension.Name = "Extension for Import of XXX files"
      ModelExtension.Code = "importXXX"
      ModelExtension.Family = "Import"
      ' Defines a new Stereotype for the Class metaclass in the
         profile section
      Dim MySttp
      Set MySttp = ModelExtension.AddMetaExtension(PdOOM.Cls_
         Class, Cls_StereotypeTargetItem)
      If not MySttp Is Nothing Then
         output "Stereotype extension successfully created in
         extended model definition"
         MySttp.Name = "MyStereotype"
         MySttp.UseAsMetaClass = true   ' The stereotype will
         behave as a new metaclass (specific list and category
         in browser)
         ' Defines an extended attribute for this stereotype
         Dim MyExa
         Set MyExa = MySttp.AddMetaExtension(Cls_
         ExtendedAttributeTargetItem)
         If not MyExa is Nothing Then
            output "Extended Attribute successfully created in
         extended model definition"
            MyExa.Name = "MyAttribute"
            MyExa.Comment = "custom attribute coming from
         import"
            MyExa.DataType = 10  ' This corresponds to integer
            MyExa.Value = "-1"   ' This is the default value
            Set MyExa = Nothing
         End If
         ' Defines an extended collection for this stereotype
         Dim MyExCol
         Set MyExCol = MySttp.AddMetaExtension(Cls_
         ExtendedCollectionTargetItem)
         If not MyExCol is Nothing Then
            output "Extended collection successfully created in
         extended model definition"
                     MyExCol.Name = "MyCollection"
            MyExCol.Comment = "custom collection coming from
         import"
            MyExCol.DestinationClassKind = PdOOM.Cls_class  '
         The collection can store only classes
            MyExCol.Destinationstereotype = "MyStereotype"  '
         The collection can store only classes with stereotype
         "MyStereotype"
            Set MyExCol = Nothing
         End If
         Set MySttp = Nothing
      End If
      Set ModelExtension = Nothing
   End If
End If
```

Manipulating Objects Extended Properties Using Scripts

You can dynamically get and set objects extended properties like attributes and collections using scripts.

The syntax for identifying any object property is:

```
"<TargetCode>.<PropertyName>"
```

For example, to get the extended attribute MyAttribute from the importXXX object, use:

```
GetExtendedAttribute("importXXX.MyAttribute")
```

Note that if the script is inside a profile (for example, in a custom check script), you can use the %CurrentTargetCode% variable instead of a hard-coded TargetCode, in order to improve the portability of your script.

For example:

```
GetExtendedAttribute("%CurrentTargetCode%.MyAttribute")
```

In the following example we:

- Modify extended attribute on the class
- Modify extended collection on the class
- Add the class in its own extended collection to be used as a standard collection

Example

```
If (not ExistingModel Is Nothing) and (not FoundClass Is
      Nothing) Then
   ' Modify extended attribute on the class
   Dim ExaName
    ExaName = "importXXX.MyAttribute"  ' attribute name prefixed
      by extended model definition code
   If FoundClass.HasExtendedAttribute(ExaName) Then
      output "Extended attribute can be accessed"
       FoundClass.SetExtendedAttributeText ExaName, "1024"
      FoundClass.SetExtendedAttribute ExaName, 2048
      Dim valAsText, valAsInt
      valAsText = FoundClass.GetExtendedAttributeText(ExaName)
      valAsInt = FoundClass.GetExtendedAttribute(ExaName)
   End If
   ' Modify extended collection on the class
   Dim ExColName, ExCol
   ExColName = "importXXX.MyCollection"  ' collection name
      prefixed by extended model definition code
   Set ExCol = FoundClass.GetExtendedCollection(ExColName)
   If not ExCol is Nothing Then
      output "Extended collection can be accessed"
       ' The extended collection can be used as a standard
         collection
      ' for example, we add the class in its own extended
         collection
      ExCol.Add FoundClass
      Set ExCol = Nothing
   End If
End If
```

Creating a Graphical Synonym Using Scripts

You create a graphical synonym by attaching the same object twice to the same package.

Example

```
set diag = ActiveDiagram
set pack = ActivePackage
set class = pack.classes.createnew
set symbol1 = diag.AttachObject (class)
set symbol2 = diag.AttachObject (class)
```

Creating an Object Selection Using Scripts

Object Selection is a model object that is very useful to select other model objects in order to apply to them a specific treatment. You can for example add some objects to the Object Selection to move them to another package in a unique operation instead of repeating the same operation for each and every objects individually.

When dealing with a set of objects in the user interface, you use the Object Selection in scripting.

- Create Object Selection

You create the Object Selection from a model using the CreateSelection method: CreateSelection() As BaseObject.

Example

```
Set MySel = ActiveModel.CreateSelection
```

- Add objects individually

You can add objects individually by adding the required object to the Objects collection.

You use the Object Selection following method: Add(obj As BaseObject)

Example

Adding of an object named Publisher:

```
MySel.Objects.Add(Publisher)
```

- Add objects of a given type

You can add all objects of a given type by using the Object Selection following method: AddObjects(ByVal RootPackage As BaseObject, ByVal ClassType As Long, ByVal IncludeShortcuts As Boolean = 0, ByVal Recursive As Boolean = 0).

RootPackage is the package from which to add objects.

ClassType is the type of object to add.

IncludeShortcuts is the parameter to include shortcuts.

Recursive is the parameter to search in all the sub-packages.

Example

An adding of classes with no inclusion of shortcuts and no recursiveness into the sub-packages:

```
MySel.AddObjects(folder,cls_class)
```

- Remove objects from the current selection

You can remove objects from the current selection using the Object Selection following method: RemoveObjects(ByVal ClassType As Long, ByVal IncludeShortcuts As Boolean = -1)

Example

Withdrawal of all classes and shortcuts from the Object Selection:

```
MySel.RemoveObjects(cls_class, -1)
```

- Move objects of the current selection to a destination package

You can move objects of the current selection to a destination package using the Object Selection following method: MoveToPackage(ByVal TargetPackage As BaseObject)

Example

Move of objects of the selection to a destination package named Pack:

```
MySel.MoveToPackage Pack
```

- Copy objects of the current selection to a destination package

You can copy objects of the current selection to a destination package using the Object Selection following method: CopyToPackage(ByVal TargetPackage As BaseObject)

Example

Copy of objects of the selection in a destination package named Pack:

```
MySel.CopyToPackage Pack
```

- Filter a selection list by stereotype

You can create an object selection and filter this selection using a stereotype. You have to use the following method: ShowObjectPicker(ByVal classNames As String, ByVal stereotypeFilter As String = “”, ByVal dialogCaption As String = “”) As BaseObject

Example

Opens a selection dialog box for selecting a business transaction:

```
If Not Fldr is Nothing then
         ' Create a selection object
         Set Sel = Mdl.CreateSelection
         If Not Sel is Nothing then
           'Show the object picker dialog for selecting a BT
            Set Impl = Sel.ShowObjectPicker ("Process",
         "BinaryCollaboration", "Select a Binary Collaboration
         Process")
            ' Retrieve the selection
            If not Impl is Nothing Then
               If Impl.IsKindOf(PDBPM.Cls_Process) and
         Impl.Stereotype = "BinaryCollaboration" then
                  Set Shct = Impl.CreateShortcut (Fldr)
                  If not Shct is Nothing Then
                     obj.Implementer = Shct
                        %Initialize% = True
                   End If
               End If
             End If
         End If
```

Creating an Extended Model Definition Using Scripts

As any PowerDesigner object, extended model definitions can be read, created and modified using scripting.

☞ For more information on extended model definitions, see the “Extended Model Definitions Reference Guide” chapter in the *Advanced User Documentation* .

☞ The following script illustrates how you can **access** an existing extended model definition, **browse** it , **create** an extended attribute within the definition and at last **modify** the extended attribute values. A function is created to drill down the categories tree view that is displayed in the Extended Model Definition Properties dialog box.

Example

```
Dim M
Set M = ActiveModel
'Retrieve first extended model definition in the active model
Dim X
Set X = M.ExtendedModelDefinitions.Item(0)
'Drill down the categories tree view using the searchObject
         function (see below for details)
Dim C
Set C = SearchObject (X.Categories, "Settings")
Set C = SearchObject (C.Categories, "Extended Attributes")
Set C = SearchObject (C.Categories, "Objects")
Set C = SearchObject (C.Categories, "Entity")
'Create extended attribute in the Entity category
Dim A
Set A = C.Categories.CreateNew (cls_ExtendedAttributeTargetItem)
'Define properties of the extended attribute
A.DataType = 10 'integer
A.Value = 10
A.Name = "Z"
A.Code = "Z"
'Retrieve first entity in the active model
Dim E
Set E = M.entities.Item(0)
'Retrieve the values of the created extended attribute in a
         message box
msgbox E.GetExtendedAttribute("X.Z")
'Changes the values of the extended attribute
E.SetExtendedAttribute "X.Z", 5
'Retrieve the modified values of the extended attribute in a
         message box
msgbox E.GetExtendedAttribute("X.Z")
********************
'Detail SearchObject function that allows you to browse a
         collection from its name and the searched object
'* SUB   SearchObject
Function SearchObject (Coll, Name)
'For example Coll = Categories and Name = Settings
   Dim Found, Object
   For Each Object in Coll
      If Object.Name = Name Then
         Set Found = Object
      End If
   Next
   Set SearchObject = Found
End Function
```

Mapping Objects Using Scripts

You can use scripting to map objects from heterogeneous models.

You create or reuse a mapping for an object using the following method on the DataSource object and on the ClassifierMap object: CreateMapping(ByVal Object As BaseObject) As BaseObject.

Example

Given the following example where an OOM (oom1) contains a class (class_1) with two attributes (att1 and att2) and a PDM (pdm1) contains a table (table_1) with two columns (col1 and col2). To map the OOM class and attributes to the PDM table and columns, you have to do the following:

- Create a data source in the OOM

```
set ds = oom1.datasources.createnew
```

- Add the PDM as source for the data source

```
ds.AddSource pdm1
```

- Create a mapping for class_1 and set this mapping as the default for class_1 (current data source being the default)

```
set map1 = ds.CreateMapping(class_1)
```

- Add table_1 as source for class_1

```
map1.AddSource table_1
```

- Add a mapping for att1

```
set attmap1 = map1.CreateMapping(att1)
```

- Set col1 as source for att1

```
attmap1.AddSource col1
```

- Add a mapping for att2

```
set attmap2 = map1.CreateMapping(att2)
```

- Set col2 as source for att2

```
attmap.AddSource col2
```

You can also get the mapping of an object using the following method on the DataSource object and on the ClassifierMap object: GetMapping(ByVal Object As BaseObject) As BaseObject.

- Get the mapping of class_1

```
Set mymap = ds.GetMapping (class_1)
```

- Get the mapping of att1

```
Set mymap = map1.GetMapping (att1)
```

☞ For more information about objects mapping, see the “Mapping objects”section in the “Working with” chapter in the OOM, the PDM and the XSM user’s guides.

CHAPTER 3

Generating and Reversing a DataBase Using Scripts

About this chapter

This chapter gives you some examples to help you get started with some generic manipulations on PDM databases that you can perform using scripting.

Contents

Generating a Database Using Scripts

When you need to generate a database using script, you may use the following methods:

- GenerateDatabase(ByVal ObjectSelection As BaseObject = Nothing)
- GenerateTestData(ByVal ObjectSelection As BaseObject = Nothing)

In the following example, you:

- Open an existing model.
- Generate a script for the model.
- Modify the model.
- Generate a modified database script.
- Generate a set of test data.

Opening an existing model

In the following example, we begin with opening an existing model (ASA 9) using the following method: OpenModel (filename As String, flags As Long =omf_Default) As BaseObject.

Be sure to add a final backslash (\) to the generation directory.

Then we are going to generate a database script for the model, modify the model, generate a modified data script, and generate a set of test data using respectively the following methods:

- GenerateDatabaseScripts pModel
- ModifyModel pModel
- GenerateAlterScripts pModel
- GenerateTestDataScript pModel

Example

```
Option Explicit
Const GenDir = "D:\temp\test\"
Const Modelfile = "D:\temp\phys.pdm"
Dim fso : Set fso = CreateObject("Scripting.FileSystemObject")
Start
Sub Start()
   dim pModel : Set pModel = OpenModel(Modelfile)
   If (pModel is Nothing) then
      Output "Unable to open the model"
      Exit Sub
   End if
End Sub
```

Generating a script for the model

Then you generate a script for this model in the folder defined in the "GenDir" constant using the following method: GenerateDatabase(ByVal ObjectSelection As BaseObject = Nothing).

As you would do in the generation database dialog box, you have to define the generation directory and the sql file name before starting the generation, see the following example.

Example

```
Sub GenerateDatabaseScripts(pModel)
   Dim pOpts : Set pOpts = pModel.GetPackageOptions()
   InteractiveMode = im_Batch ' Avoid displaying generate window
   ' set generation options using model package options
   pOpts.GenerateODBC = False ' Force sql script generation
         rather than
   ' ODBC
   pOpts.GenerationPathName = GenDir ' Define generation
         directory
   pOpts.GenerationScriptName = "script.sql" ' Define sql file
         name
   pModel.GenerateDatabase ' Launch the Generate Database
         feature
End Sub
```

Modifying the model

After, you modify the model by adding a column to each table:

Example

```
Sub ModifyModel(pModel)
   dim pTable, pCol
   ' Add a new column in each table
   For each pTable in pModel.Tables
      Set pCol = pTable.Columns.CreateNew()
      pCol.SetNameAndCode "az" & pTable.Name, "AZ" & pTable.Code
      pCol.Mandatory = False
   Next
End Sub
```

Generating a modified database script

Before generating the modified database script, you have to get package option and change generation parameters, then you generate the modified database script accordingly.

For more information about the generation options, see section BasePhysicalPackageOptions in the Metamodel Object Help file.

Example

```
Sub GenerateAlterScripts(pModel)
   Dim pOpts : Set pOpts = pModel.GetPackageOptions()
   InteractiveMode = im_Batch ' Avoid displaying generate window
' set generation options using model package options
   pOpts.GenerateODBC = False ' Force sql script generation
         rather than ODBC
pOpts.GenerationPathName = GenDir
   pOpts.DatabaseSynchronizationChoice = 0 'force already saved
         apm as source
   pOpts.DatabaseSynchronizationArchive = GenDir & "model.apm"
   pOpts.GenerationScriptName = "alter.sql"
pModel.ModifyDatabase ' Launch the Modify Database feature
End Sub
```

Generating a set of test data

Finally, you generate a set of test data:

Example

```
Sub GenerateTestDataScript(pModel)
   Dim pOpts : Set pOpts = pModel.GetPackageOptions()
   InteractiveMode = im_Batch ' Avoid displaying generate window
' set generation options using model package options
   pOpts.TestDataGenerationByODBC = False ' Force sql script
         generation rather than ODBC
   pOpts.TestDataGenerationDeleteOldData = False
pOpts.TestDataGenerationPathName = GenDir
   pOpts.TestDataGenerationScriptName = "Test.sql"
pModel.GenerateTestData ' Launch the Generate Test Data feature
End Sub
```

Generating a Database via ODBC using script

You can generate a database via ODBC using script.

To do so, you first begin with connecting to the database using the ConnectToDatabase(ByVal Dsn As String, ByVal User As String, ByVal Password As String) As Boolean method from the model, then you set up the generation options and launch the generation feature.

For more information about the generation options, see section BasePhysicalPackageOptions in the Metamodel Object Help file.

Example:

```
Const cnxDSN = "ODBC:ASA 9.0 sample"
Const cnxUSR = "dba"
Const cnxPWD = "sql"

Const GenDir = "C:\temp\"
Const GenFile = "test.sql"
Const ModelFile = "C:\temp\phys.pdm"

set pModel = openModel(ModelFile)

   set pOpts=pModel.GetPackageOptions()

      pModel.ConnectToDatabase cnxDSN, cnxUSR, cnxPWD
      pOpts.GenerateODBC = true

   pOpts.GenerationPathName = GenDir
   pOpts.GenerationScriptName = 'script.sql"
   pModel.GenerateDatabase
```

Generating a Database Using Setting and Selection

You can use settings and selections with scripting before starting the database generation using respectively the following methods from the model: UseSettings(ByVal Function As Long, ByVal Name As String = “”) As Boolean and UseSelection(ByVal Function As Long, ByVal Name As String = “”) As Boolean.

Given the PDM sample (Project.PDM) in the PowerDesigner installation folder, which contains two selections:

- ”Organization” selection includes tables DIVISION, EMPLOYEE, MEMBER & TEAM.
- ”Materials” selection includes tables COMPOSE, MATERIAL, PROJECT & USED.

The following example shows you how to

- Generate a first script of this model for the “Organization” selection using first setting (setting1)
- Generate a test data creation script for the tables contained in this selection.
- Generate a second script of this model for the “Materials” selection and a test data creation script for the tables it contains using second setting (setting2).

Example:

```
' Generated sql scripts will be created in 'GenDir' directory
' there names is the name of the used selection with extension
        ".sql" for DDL scripts
' and extension "_td.sql" for DML scripts (for test data
        generations).
Option Explicit

Const GenDir = "D:\temp\test\"

Const setting1 = "Tables & Views (with permissions)"
Const setting2 = "Triggers & Procedures (with permissions)"
Start EvaluateNamedPath("%_EXAMPLES%\project.pdm")

Sub Start(sModelPath)
   on error resume next
   dim pModel : Set pModel = OpenModel(sModelPath)
   If (pModel is Nothing) then
      Output "Unable to open model " & sModelPath
      Exit Sub
   End if

   GenerateDatabaseScripts pModel, "Organization" setting1
   GenerateTestDataScript pModel, "Organization" setting1

   GenerateDatabaseScripts pModel, "Materials" setting2
   GenerateTestDataScript pModel, "Materials" setting2
   pModel.Close
   on error goto 0
End Sub

Sub GenerateDatabaseScripts(pModel, sSelectionName,
        sSettingName)
   Dim pOpts : Set pOpts = pModel.GetPackageOptions()
   InteractiveMode = im_Batch ' Avoid displaying generate window
' set generation options using model package options
   pOpts.GenerateODBC = False ' Force sql script generation
        rather than ODBC
   pOpts.GenerationPathName = GenDir
   pOpts.GenerationScriptName = sSelectionName & ".sql"
 ' Launch the Generate Database feature with selected objects
   pModel.UseSelection fct_DatabaseGeneration, sSelectionName
   pModel.UseSetting fct_DatabaseGeneration, sSettingName
   pModel.GenerateDatabase
End Sub

Sub GenerateTestDataScript(pModel, sSelectionName)
   Dim pOpts : Set pOpts = pModel.GetPackageOptions()
   InteractiveMode = im_Batch ' Avoid displaying generate window
   ' set generation options using model package options
   pOpts.TestDataGenerationByODBC = False ' Force sql script
        generation rather than ODBC
   pOpts.TestDataGenerationDeleteOldData = False
   pOpts.TestDataGenerationPathName = GenDir
   pOpts.TestDataGenerationScriptName = sSelectionName & "_
        td.sql"
' Launch the Generate Test Data feature for selected objects
   pModel.UseSelection fct_TestDataGeneration, sSelectionName
   pModel.GenerateTestData
 End Sub
```

Selection and setting creation

You can create a persistent selection that can be used in database generation by transforming a selection into a persistent selection..

Example:

```
Option Explicit
   Dim pActiveModel
   Set pActiveModel = ActiveModel

   Dim Selection, PrstSel
   Set Selection = pActiveModel.createselection
   Selection.AddActiveSelectionObjects

   Set PrstSel =
         Selection.CreatePersistentSelectionManager("test")
```

Reverse Engineering a Database Using Scripts

You reverse engineer a database using scripts using the ReverseDatabase(ByVal Diagram As BaseObject = Nothing) method.

In the following example, the ODBC database is reversed into a new PDM.

The first lines of the script define the constants used:

- cnxDSN is either the ODBC dsn string or the path to an ODBC file dsn.
- cnxUSR is the ODBC connection user name.
- cnxPWD is the ODBC connection password.

Example

```
option explicit

' To use a user or system datasource, define constant with
        "ODBC:<datasourcename>"
' -> Const cnxDSN = "ODBC:ASA 9.0 sample"
' To use a datasource file, define constant with the full path
        to the DSN file
' -> Const cnxDSN = "\\romeo\public\DATABASES\_filedsn\sybase_
        asa9_sample.dsn"

' use ODBC datasource
Const cnxDSN = "ODBC:ASA 9.0 sample"
Const cnxUSR = "dba"
Const cnxPWD = "sql"
Const GenDir = "C:\temp\"
Const filename = "D:\temp\phys.pdm"

' Call to main function with the newly created PDM
' This sample use an ASA9 database
Start CreateModel(PdPDM.cls_Model, "|DBMS=Sybase AS Anywhere 9")

Sub Start(pModel)

   If (pModel is Nothing) then
      output "Unable to create a physical model for selected
         DBMS"
      Exit Sub
   End If

   InteractiveMode = im_Batch

' Reverse database phase
' First connect to the database with connection parameters
   pModel.ConnectToDatabase cnxDSN, cnxUSR, cnxPWD
' Get the reverse option of the model
   Dim pOpt
   Set pOpt = pModel.GetPackageOptions()

' Force ODBC Reverse of all listed objects
   pOpt.ReversedScript = False
   pOpt.ReverseAllTables = true
   pOpt.ReverseAllViews = true
   pOpt.ReverseAllStorage = true
   pOpt.ReverseAllTablespace = true
   pOpt.ReverseAllDomain = true
   pOpt.ReverseAllUser = true
   pOpt.ReverseAllProcedures = true
   pOpt.ReverseAllTriggers = true
   pOpt.ReverseAllSystemTables = true
   pOpt.ReverseAllSynonyms = true
' Go !
   pModel.ReverseDatabase
   pModel.save(filename)
' close model at the end
   pModel.Close false
End Sub
```

CHAPTER 4

Manipulating The Repository Using Scripts

About this chapter

This chapter explains how to access the PowerDesigner repository using VBScript and gives you some examples to help you get started with the main Repository actions you can perform using this scripting language.

Contents

Introducing the Repository Using Scripts

PowerDesigner lets you access the Repository feature via scripting using the RepositoryConnection as BaseObject global property.

It allows you to retrieve the current repository connection, which is the object that manages the connection to the repository server and provides access to documents and objects stored under the repository.

The **RepositoryConnection** is equivalent to the root node in the Repository browser.

You can access the repository documents, but you cannot access the repository administration objects, like users, groups, configurations, branches, and list of locks.

In addition, only the last version of a repository document is accessible using scripting.

The main actions you can perform are the following:

- Connect to a repository database
- Access repository documents
- Extract a document
- Consolidate a document
- Manage document versions
- Manage the repository browser

Connecting the repository database

To retrieve the current repository connection:

Use the following	Description
RepositoryConnection As BaseObject	Global property which manages the connection to the repository database

To connect to a repository database:

Use the following	Description
Open (ByVal RepDef As String = “”, ByVal User As String = “”, ByVal Pass As String = “”, ByVal DBUser As String = “”, ByVal DBPass As String = “”) As Boolean	Method on RepositoryConnection that allows you to perform a repository connection

To disconnect from the repository:

Use the following	Description
Close()	Method on RepositoryConnection that allows you to disconnect from the repository database

Accessing the repository document

To browse for a document:

Use the following	Description
ChildObjects As ObjectCol	Collection on StoredObject which manages the access to the repository documents

To update a document version:

Use the following	Description
Refresh()	Method on RepositoryConnection which lets you visualize new documents, update versions of existing documents, or hide deleted ones

To find a document:

Use the following	Description
FindInRepository() As BaseObject	Method on BaseModel that allows you to check if a model has already been consolidated

Extracting repository document

To extract any document:

Use the following	Description
ExtractToFile(ByVal FileName As String, ByVal MergeMode As Long = 2, ByVal OpenMode As Boolean = -1, ByRef Actions As String = NULL, ByRef Conflicts As String = NULL) As BaseObject	Method on RepositoryModel that allows you to extract any kind of document

To extract a PowerDesigner document:

Use the following	Description
UpdateFromRepository(ByVal MergeMode As Integer = 2, ByRef actions As String = NULL, ByRef conflicts As String = NULL) As Boolean	Method on BaseModel that allows you to extract PowerDesigner documents

Consolidating repository document

To consolidate any document:

Use the following	Description
ConsolidateDocument(ByVal FileName As String, ByVal MergeMode As Long = 2, ByRef Actions As String = NULL, ByRef Conflicts As String = NULL) As BaseObject	Method on RepositoryFolder that allows you to consolidate any kind of document

To consolidate a PowerDesigner document:

Use the following	Description
ConsolidateNew(ByVal RepositoryFolder As BaseObject, ByRef actions As String = NULL, ByRef conflicts As String = NULL) As BaseObject	Method on BaseModel that allows you to consolidate PowerDesigner documents
Consolidate(ByVal MergeMode As Integer = 2, ByRef actions As String = NULL, ByRef conflicts As String = NULL) As BaseObject	Method on BaseModel that allows you to consolidate additional repository versions of a PowerDesigner document

Managing document version

To freeze and unfreeze a document version:

Use the following	Description
Freeze(ByVal Comment As String = "") As Boolean	Method on RepositoryDocumentBase that allows you to create an archived version of a document
Unfreeze() As Boolean	Method on RepositoryDocumentBase that allows you to modify the current version in the repository to reflect changes performed on your local machine

To lock and unlock a document version:

Use the following	Description
Lock(ByVal Comment As String = "") As Boolean	Method on RepositoryDocumentBase that allows you to prevent other users from updating the consolidated version
Unlock() As Boolean	Method on RepositoryDocumentBase that allows other users to update the consolidated version

To delete a document version:

Use the following	Description
DeleteVersion() As Boolean	Method on RepositoryDocumentBase that allows you to delete a document version

Managing the repository browser

To create a folder:

Use the following	Description
CreateFolder(ByVal FolderName As String) As BaseObject	Method on RepositoryFolder that allows you to create a new folder in the repository browser

To delete an empty folder:

Use the following	Description
DeleteEmptyFolder() As Boolean	Method on RepositoryFolder that allows you to delete an empty folder in the repository browser

☞ For more information on documents, see the ""Accessing a Repository Document" on page 77" section.

Connecting to a Repository Database

Before you connect to the repository database using scripting, definitions of repositories must exist on your workstation, as you cannot define a new repository definition via the scripting feature.

You can connect to the Repository database using the following method on RepositoryConnection: Open(ByVal RepDef As String = “”, ByVal User As String = “”, ByVal Pass As String = “”, ByVal DBUser As String = “”, ByVal DBPass As String = “”) As Boolean.

Example

```
Dim C
Set C = RepositoryConnection
C.Open
```

You disconnect from the repository database using the following method: Close().

Example

```
C.Close
```

Accessing a Repository Document

You can drill down to the repository documents located in the Repository browser using the ChildObjects collection (containing both documents and folders) that also allows you to drill down to documents located in folders of the Repository if any.

The repository documents are the following:

Repository document	Description
RepositoryModel	Contains any type of PowerDesigner model (CDM, PDM, OOM, BPM, XSM, ILM, RQM or FEM)
RepositoryReport	Contains consolidated multi-model reports
RepositoryDocument	Contains non-PowerDesigner files (text, Word, or Excel)
OtherRepositoryDocument	Contains non-PowerDesigner models defined using the Java Repository interface, which allows you to define your metamodels

You can access a RepositoryModel document and the sub-objects of a RepositoryModel document using the following collection: ChildObjects As ObjectCol.

Example

```
' Retrieve the deepest folder under the connection
   Dim CurrentObject, LastFolder
   set LastFolder = Nothing
   for each CurrentObject in C.ChildObjects
   if CurrentObject.IsKindOf(cls_RepositoryFolder) then
         set LastFolder = CurrentObject
      end if
   next
```

The ChildObjects collection is not automatically updated when the Repository is modified during a script execution. To refresh all the collections, you can use the following method: Refresh().

Example

```
C.Refresh
```

You can test if a model has already been consolidated using the following method: FindInRepository() As BaseObject.

Example

```
Set repmodel = model.FindInRepository()
If repmodel Is Nothing Then
  ' Model was not consolidated yet...
   model.ConsolidateNew
Else
   ' Model was already consolidated...
   repmodel.Freeze
   model.Consolidate
End If
```

Extracting a Repository Document

There are two ways to extract a repository document using scripting:

- A generic way that is applicable to any repository document
- A specific way that is only applicable to RepositoryModel and RepositoryReport documents

Generic way

To extract a repository document you must:

- Browse for a repository document using the ChildObjects collection
- Extract the document using the method ExtractToFile (ByVal FileName As String, ByVal MergeMode As Long = 2, ByVal OpenMode As Boolean = -1, ByRef Actions As String = NULL, ByRef Conflicts As String = NULL) As BaseObject

Example

```
set C = RepositoryConnection
C.Open
Dim D, P
set P = Nothing
for each D in C.ChildObjects
if D.IsKindOf (cls_RepositoryModel) then
D.ExtractToFile ("C:\temp\OO.OOM")
end if
next
```

Specific way

To extract a RepositoryModel document or a RepositoryReport document you must:

- Retrieve the document from the local model or multi-model report, (provided it has already been consolidated) using the method UpdateFromRepository (ByVal MergeMode As Integer = 2, ByRef actions As String = NULL, ByRef conflicts As String = NULL) As Boolean

Example

```
set MyModel = OpenModel ("C:\temp\OO3.OOM")
MyModel.UpdateFromRepository
```

Consolidating a Repository Document

There are two ways to consolidate a repository document using scripting:

- A generic way that is applicable to any repository document
- A specific way that is only applicable to RepositoryModel and RepositoryReport documents

Generic way

To consolidate any repository document you must:

- Specify a filename when using the following method ConsolidateDocument (ByVal FileName As String, ByVal MergeMode As Long = 2, ByRef Actions As String = NULL, ByRef Conflicts As String = NULL) As BaseObject)

Example

```
set C = RepositoryConnection
C.open
C.ConsolidateDocument ("c:\temp\test.txt")
```

Specific way

To consolidate a RepositoryModel document or a RepositoryReport document you can use one of the following methods:

- ConsolidateNew (ByVal RepositoryFolder As BaseObject, ByRef actions As String = NULL, ByRef conflicts As String = NULL) As BaseObject, to consolidate the first repository version of a document
- Consolidate (ByVal MergeMode As Integer = 2, ByRef actions As String = NULL, ByRef conflicts As String = NULL) As BaseObject, to consolidate additional repository versions of a document

Example

```
Set model = CreateModel(PdOOM.cls_Model,
         "|Diagram=ClassDiagram")
set C = RepositoryConnection
C.Open
model.ConsolidateNew c
```

Example

```
set C = RepositoryConnection
C.Open
model.Consolidate
```

Understanding the Conflict Resolution Mode

If you update a document that has already been modified since last extraction or consolidation, a conflict can occur.

Consolidation conflicts

You can resolve conflicts that arise when consolidating a repository document by specifying a merge mode in the second parameter of the following method: ConsolidateDocument(ByVal FileName As String, ByVal MergeMode As Long = 2, ByRef Actions As String = NULL, ByRef Conflicts As String = NULL) As BaseObject.

This parameter (ByVal MergeMode As Long = 2)can contain the following values:

Value	Description
1	Replaces the document in the repository with the local document without preserving any repository changes
2 (default value)	Tries to automatically select the default merge actions by taking into account the modification dates of objects and cancels the consolidation if a conflict has been found (objects modified both locally and in the repository)
3	Selects the default merge actions but always favors local changes in case of conflict instead of canceling the consolidation
4	Selects the default merge actions and favors the repository changes in case of conflict

Merge actions performed during consolidation and conflicts that may have occurred can be retrieved in the strings specified in the third and fourth parameters: ByRef Actions As String = NULL and ByRef Conflicts As String = NULL.

Extraction conflicts

You can resolve conflicts that arise when extracting a repository document by specifying a merge mode in the second parameter of the following method: ExtractToFile(ByVal FileName As String, ByVal MergeMode As Long = 2, ByVal OpenMode As Boolean = -1, ByRef Actions As String = NULL, ByRef Conflicts As String = NULL) As BaseObject.

This parameter (ByVal MergeMode As Long = 2)can contain the following values:

Value	Description
0	Extracts the document without merge, thus erases the existing local document if any, and sets the extracted file as read-only
1	Extracts the document without merge, thus erases the existing local document if any
2 (default value)	Tries to automatically select the default merge actions by taking into account the modification dates of objects and cancels the extraction if a conflict has been found (objects modified both locally and the repository)
3	Selects the default merge actions but always favors local changes in case of conflict instead of canceling the extraction
4	Selects default merge actions and favors the repository changes in case of conflict

Merge actions performed during extraction and conflicts that may have occurred can be retrieved in the strings specified in the fourth and fifth parameters: ByRef Actions As String = NULL and ByRef Conflicts As String = NULL. The third parameter (ByVal OpenMode As Boolean = -1) allows you to keep open the extracted model.

Managing Document Versions

You can manage the document versions using scripting as follows:

- Freeze and unfreeze a document version

You can freeze and unfreeze a document version using the following methods: Freeze(ByVal Comment As String = “”) As Boolean and Unfreeze() As Boolean.

Example

```
MyDocument.Freeze "Update required"
```

Example

```
MyDocument.Unfreeze
```

- Lock and unlock a document version

You can lock and unlock a document version using the following methods: Lock(ByVal Comment As String = “”) As Boolean and Unlock() As Boolean.

Example

```
MyDocument.Lock "Protection required"
```

Example

```
MyDocument.Unlock
```

- Delete a document version

You can delete a document version using the following method: DeleteVersion() As Boolean.

Example

```
MyDocument.Delete
```

Managing the Repository Browser

You can manage the repository browser using scripting as follows:

- Create folder

You can create a folder using the following method: CreateFolder(ByVal FolderName As String) As BaseObject

Example

```
RepositoryConnection.CreateFolder("VBTest")
```

- Delete empty folder

You can delete an empty folder using the following method: DeleteEmptyFolder() As Boolean

Example

```
Dim C
Set C = RepositoryConnection
C.Open "MyRepDef"
' Retrieve the deepest folder under the connection
Dim D, P
   set P = Nothing
   for each D in C.ChildObjects
      if D.IsKindOf (cls_RepositoryFolder) then
            D.DeleteEmptyFolder
            c.refresh
      end if
   next
```

CHAPTER 5

Manipulating Reports Using Scripts

About this chapter

This chapter explains how to access the PowerDesigner reports using VBScript and gives you some examples to help you get started with the main report actions you can perform using this scripting language.

Contents

Managing Reports Using Scripts

You can generate HTML and RTF reports using scripting, but you cannot create reports.

The main actions you can perform for model reports and multi-model reports are the following:

- Browse a report for a model
- Retrieve a multimodel report
- Generate an HTML report
- Generate a RTF report

Browsing a Report for a Model Using Scripts

You can browse a report for a model using the following collection on BaseModelReport: Reports As ObjectCol.

Example

```
set m = ActiveModel
For each Report in m.Reports
Output Report.name
```

Retrieving a Multimodel Report Using Scripts

You can retrieve a multimodel report using the following function: OpenModel(filename As String, flags As Long =omf_Default) As BaseObject

Example

```
OpenModel ("c:\temp\mmr1.mmr")
```

Generating an HTML Report Using Scripts

You can generate a model report or a multimodel report as HTML using the following method on BaseModelReport: GenerateHTML(ByVal FileName As String) As Boolean.

Example

```
set m = ActiveModel
For each Report in m.Reports
   Filename = Report.name & ".htm"
   Report.GenerateHTML (filename)
Next
```

Generating a RTF Report Using Scripts

You can generate a model report or a multimodel report as RTF using the following method on BaseModelReport: GenerateRTF(ByVal FileName As String) As Boolean

Example

```
set m = ActiveModel
For each Report in m.Reports
   Filename = Report.name & ".rtf"
   Report.GenerateRTF (filename)
Next
```

CHAPTER 6

Manipulating MetaData Using Scripts

About this chapter

This chapter explains how to access the PowerDesigner metadata using VBScript and gives you some examples to help you get started with the main actions on metadata you can perform using this scripting language.

Contents

Introducing MetaData Using Scripts

You can access and manipulate PowerDesigner internal objects using Visual Basic Scripting. The scripting lets you access and modify object properties, collections, and methods using the public names of objects.

The PowerDesigner metamodel provides useful information about objects:

Information	Description	Example
Public name	The name and code of the metamodel objects are the public names of PowerDesigner internal objects	AssociationLinkSymbol ClassMapping CubeDimensionAssociation
Object collections	You can identify the collections of a class by observing the associations linked to this class in the diagram. The role of each association is the name of the collection	In PdBPM, an association exists between classes MessageFormat and MessageFlow. The public name of this association is Format. The role of this association is Usedby which corresponds to the collection of message flows of class MessageFormat
Object attributes	You can view the attributes of a class together with the attributes this class inherits from other classes via generalization links	In PdCommon, in the Common Instantiable Objects diagram, you can view objects BusinessRule, ExtendedDependency and FileObject with their proper attributes, and the abstract classes from which they inherit attributes via generalization links
Object operations	Operations in metamodel classes correspond to object methods used in VBS	BaseModel contains operation Compare that can be used in VB scripting
<<notScriptable>> stereotype	Objects that do not support VB scripting have the <<notScriptable>> stereotype	CheckModelInternalMessage FileReportItem

PowerDesigner lets you access the MetaData via scripting using the MetaModel As BaseObject global property. There is only one instance of the MetaModel and it can be reached from anywhere through the global property Application.MetaModel.

This generic feature allows you to acccess the MetaModel on a generic way and implies a neutral code that you can use for any type of model. For example, you can use it to search for the last object modified in a given model.

Properties and collections are read-only for all MetaData objects.

The main actions you can perform on MetaData are the following:

- Retrieve the MetaModel version
- Retrieve the available types of MetaClass libraries
- Access the MetaClass of an object using its public name
- Access the MetaAttribute and MetaCollection of an object using its public name
- Retrieve the children of a MetaClass
- List all MetaAttribute and MetaCollection objects available for a MetaClass

Accessing MetaData objects using scripts

You can access the MetaData objects using scripts:

Use the following	Description
MetaModel As BaseObject	Global property. Entry point to access MetaData objects

Retrieving the MetaModel version using scripts

You can retrieve the MetaModel version using scripts:

Use the following	Description
Version As String	Property. Allows you to retrieve the MetaModel version

Retrieving the available types of MetaClass libraries using scripts

You can retrieve the available types of MetaClass libraries using scripts:

Use the following	Description
MetaLibrary	Collection. Allows you to retrieve the available MetaClass of libraries of a given module

Accessing the MetaClass of an object using scripts

You can access the MetaClass of an object using scripts:

Use the following	Description
MetaClass As BaseObject	Property. Provides access to the Metaclass of each object

Accessing the MetaClass of an object using its public name using scripts

You can access the MetaClass of an object using its public name from the MetaModel using scripts:

Use the following	Description
GetMetaClassByPublic-Name (ByVal name As String) As BaseObject	Method. Provides access to the MetaClass of an object using its public name

Accessing the MetaAttribute and MetaCollection of a Metaclass using scripts

You can access the MetaAttribute and MetaCollection of a MetaClass using its public name (from the MetaClass):

Use the following	Description
GetMetaMemberByPublic-Name (ByVal name As String) As BaseObject	Method. Provides access to a MetaAttribute or a MetaCollection using its public name

Retrieving the Children of a MetaClass using scripts

You can retrieve the children of a MetaClass using scripts:

Use the following	Description
Children As ObjectSet	Collection. Lists the MetaClasses that inherit from the parent MetaClass

CHAPTER 7

Manipulating The Workspace Using Scripts

About this chapter

This chapter explains how to access the PowerDesigner workspace using VBScript and gives you some examples to help you get started with the main workspace actions you can perform using this scripting language.

Contents

Managing the Workspace Using Scripts

PowerDesigner lets you access the workspace feature via VBScript using the ActiveWorkspace As BaseObject global property.

It allows you to retrieve the application current workspace.

The **Workspace** object corresponds to the workspace root in the Browser.

The main actions you can perform with the Workspace are the following:

- Load a workspace
- Save a workspace
- Close a workspace
- Manipulate the content of a workspace

Loading, saving and closing a workspace using scripts

The following methods are available to load, save and close a workspace using scripts:

To load a workspace

Use the following	Description
Load (ByVal filename As String = "") As Boolean	Loads the workspace from the given location

To save a workspace

Use the following	Description
Save (ByVal filename As String = "") As Boolean	Saves the workspace at the given location

To close a workspace

Use the following	Description
Close ()	Closes the active workspace

Manipulating the content of a workspace using scripts

You can also manipulate the content of a workspace using the following items:

- The **WorkspaceDocument** that corresponds to the documents you can add to a workspace. It contains the **WorkspaceModel** (models attached

to a workspace) and the **WorkspaceFile** (external files attached to the workspace)

- The **WorkspaceFolder** that corresponds to the folders of the workspace. You can create, delete and rename them. You can also add documents to folders.

You can use the AddDocument(ByVal filename As String, ByVal position As Long = -1) As BaseObject method on the WorkspaceFolder to add documents to the workspace.

Example of a workspace manipulation:

```
Option Explicit
' Close existing workspace and save it to Temp
Dim workspace, curentFolder
Set workspace = ActiveWorkspace
workspace.Load "%_EXAMPLES%\mywsp.sws"
Output "Saving current workspace to ""Example directory :
        "+EvaluateNamedPath("%_EXAMPLES%\temp.sws")
workspace.Save "%_EXAMPLES%\Temp.SWS"
workspace.Close
workspace.Name = "VBS WSP"
workspace.FileName = "VBSWSP.SWS"
workspace.Load "%_EXAMPLES%\Temp.SWS"
dim Item, subitem
for each Item in workspace.children
   If item.IsKindOf(PdWsp.cls_WorkspaceFolder) Then
       ShowFolder (item)
      renameFolder item,"FolderToRename", "RenamedFolder"
         deleteFolder item,"FolderToDelete"
      curentFolder = item
   ElsIf item.IsKindOf(PdWsp.cls_WorkspaceModel) Then
       ElsIf item.IsKindOf(PdWsp.cls_WorkspaceFile) Then
       End if
    next
 Dim subfolder
'insert folder in root
 Set subfolder = workspace.Children.CreateNew(PdWsp.cls_
        WorkspaceFolder)
 subfolder.name = "Newfolder(VBS)"
 'insert folder in root at pos 6
 Set subfolder = workspace.Children.CreateNewAt(5, PdWsp.cls_
        WorkspaceFolder)
 subfolder.name = "Newfolder(VBS)insertedAtPos5"'
   ' add a new folder in this folder
 Set subfolder = subfolder.Children.CreateNew(PdWsp.cls_
        WorkspaceFolder)
 subfolder.name = "NewSubFolder(VBS)"
 subfolder.AddDocument EvaluateNamedPath("%_EXAMPLES%\
        pdmrep.rtf")
 subfolder.AddDocument EvaluateNamedPath("%_EXAMPLES%\
        cdmrep.rtf")
 subfolder.AddDocument EvaluateNamedPath("%_EXAMPLES%\
        project.pdm")
 subfolder.AddDocument EvaluateNamedPath("%_EXAMPLES%\demo.oom")
 dim lastmodel
 set lastmodel = subfolder.AddDocument (EvaluateNamedPath("%_
        EXAMPLES%\Ordinateurs.fem"))
 lastmodel.open
 lastmodel.name = "Computers"
 lastmodel.close
 'detaching model from workspace
 lastmodel.delete
workspace.Save "%_EXAMPLES%\Final.SWS"
```

CHAPTER 8

Communicating With PowerDesigner Using OLE Automation

About this chapter

This chapter explains how to access the PowerDesigner environment using OLE Automation.

Contents

Introducing OLE Automation

OLE Automation is a way to communicate with PowerDesigner from another application using scripting. OLE Automation can also be called Visual Basic for Application.

OLE Automation lets other applications communicate with PowerDesigner using the COM architecture in the same application or in other applications. It allows you to write a program using any languages that support COM. Word and Excel macros, VB, C++, or even PowerBuilder are examples of languages that can be used to communicate with PowerDesigner objects and functions.

OLE Automation samples for different languages are located in the PowerDesigner OLE Automation directory.

Differences Between VBScript and OLE Automation

VBScript programs and OLE Automation programs are very similar. You can easily create VB or VBA programs, if you know how to use VBScript.

However, some differences remain. The following example of program highlights what differentiate OLE Automation from VBScript.

VBScript program

The following VBScript program allows you to count the number of classes defined in an OOM and display that number in PowerDesigner Output window, then create a new OOM and display its name in the same Output window.

To do so, the following steps are necessary:

- Get the current active model using the ActiveModel global function
- Check the existence of an active model and if the active model is an OOM
- Count the number of classes in the active OOM and display a message in the Output window
- Create a new OOM and display its name in the Output window

```
'* Purpose:  This script displays the number of classes defined
         in an OOM in the output window.
Option Explicit
' Main function
' Get the current active model
Dim model
Set model = ActiveModel
If model Is Nothing Then
   MsgBox "There is no current model."
ElsIf Not Model.IsKindOf(PdOOM.cls_Model) Then
   MsgBox "The current model is not an OOM model."
Else
   ' Display the number of classes
   Dim nbClass
   nbClass = model.Classes.Count
   Output "The model '" + model.Name + "' contains " +
         CStr(nbClass) + " classes."
' Create a new OOM
   Dim model2
   set model2 = CreateModel(PdOOM.cls_Model)
   If Not model2 Is Nothing Then
      ' Copy the author name
      model2.author = model.author
      ' Display a message in the output window
      Output "Successfully created the model '" + model2.Name +
         "'."
   Else
      MsgBox "Cannot create an OOM."
   End If
End If
```

OLE Automation program

To do the same with OLE Automation program, you should modify it as follows:

- Add the definition of the PowerDesigner application
- Call the CreateObject function to create an instance of the PowerDesigner Application object
- Prefix all the global functions (ActiveModel, Output, CreateModel) by the PowerDesigner Application object
- Release the PowerDesigner Application object
- Use specific types for the variables “model” and “model2”

```
'* Purpose:  This script displays the number of classes defined
        in an OOM in the output window.
Option Explicit
' Main function
Sub VBTest()
   ' Defined the PowerDesigner Application object
   Dim PD As PdCommon.Application
   ' Get the PowerDesigner Application object
   Set PD = CreateObject("PowerDesigner.Application")
' Get the current active model
   Dim model As PdCommon.BaseModel
   Set model = PD.ActiveModel
   If model Is Nothing Then
      MsgBox "There is no current model."
   ElsIf Not model.IsKindOf(PdOOM.cls_Model) Then
      MsgBox "The current model is not an OOM model."
   Else
      ' Display the number of classes
      Dim nbClass
      nbClass = Model.Classes.Count
      PD.Output "The model '" + model.Name + "' contains " +
         CStr(nbClass) + " classes."
' Create a new OOM
      Dim model2 As PdOOM.Class
      Set model2 = PD.CreateModel(PdOOM.cls_Model)
      If Not model2 Is Nothing Then
         ' Copy the author name
         model2.Author = Model.Author
         ' Display a message in the output window
         PD.Output "Successfully created the model '" +
         model2.Name + "'."
      Else
         MsgBox "Cannot create an OOM."
      End If
   End If
' Release the PowerDesigner Application object
   Set PD = Nothing
End Sub
```

OLE Automation requirements to communicate with PowerDesigner

To use OLE Automation to communicate with PowerDesigner, you need to:

- Create an instance of the PowerDesigner Application object
- Prefix all global functions with the PowerDesigner Application object
- Release the PowerDesigner Application object before exiting the program
- Specify objects type whenever possible (Dim obj As <ObjectType>)
- Adapt the object class ID syntax to the language when you create object
- Add references to the object type libraries you need to use

Creating the PowerDesigner Application Object

PowerDesigner setup registers the PowerDesigner Application object by default.

You should check if the returned variable is empty.

When you create the PowerDesigner Application object, the current instance of PowerDesigner will be used, otherwise PowerDesigner will be launched.

If PowerDesigner is launched when you create the PowerDesigner Application object, it will be closed when you release the PowerDesigner Application object.

You create the PowerDesigner application object, using the following method in Visual Basic: CreateObject(ByVal Kind As Long, ByVal ParentCol As String = “”, ByVal Pos As Long = -1, ByVal Init As Boolean = -1) As BaseObject

Example

```
' Defined the PowerDesigner Application object
   Dim PD As PdCommon.Application
   ' Get the PowerDesigner Application object
   Set PD = CreateObject("PowerDesigner.Application")
```

PowerDesigner version number

If you want to make sure that the application works with a selected version of PowerDesigner, you should type the version number in the PowerDesigner application object creation orders:

```
' Defined the PowerDesigner Application object
   Dim PD As PdCommon.Application
   ' Get the PowerDesigner Application object
   Set PD = CreateObject("PowerDesigner.Application.11.0")
```

If you do not use a particular feature of PowerDesigner, your application can work with any version of PowerDesigner and you do not need to specify a version number. In this case, the last version installed is used.

Release the PowerDesigner Application object
You must release the PowerDesigner Application object before you exit the application in which you use it. To do so, you use the following syntax: Set Pd = Nothing.

Specifying Object Type

When you create VB or VBA programs, it is strongly recommended to specify object type.

For example, you should use:

```
Dim cls As PdOOM.Class
```

Instead of:

```
Dim cls
```

If you do not specify object type, you may encounter problems when you execute your program and debugging can be really difficult.

Shortcuts

If the model contains shortcuts, we recommend to use the following syntax: Dim obj as PdCommon.IdentifiedObject.

If the target model is closed, you will get a runtime error.

Adapting Object Class ID Syntax to the Language

When you create an object using VBScript, you indicate the class ID of the object to create in the following way:

```
Dim cls
Set cls = model.CreateObject(PdOOM.cls_Class)
```

This syntax works properly for VBScript, VBA and VB, but it does not work for other languages, as class Ids constants are defined as an enumeration. Only languages that support enumeration defined outside a class can support this syntax.

For C# and VB.NET, you can use the following syntax:

```
Dim cls As PdOOM.Class
Set cls = model.CreateObject(PdOOM.PdOOM_Classes.cls_Class)
'Where PdOOM_Classes is the name of the enumeration.
```

For other languages such as JavaScript or PowerBuilder, you have to define constants that represent the objects you want to create.

For a complete list of class ID constants, see file VBScriptConstants.vbs in the PowerDesigner OLE Automation directory.

Adding References to Object Type Libraries

You must add references to the PowerDesigner type libraries you want to use, for example Sybase PdCommon, Sybase PdOOM, Sybase PdPDM, etc. for programs like VB, VBA, VB .NET and C#.

❖ **To add references to object type libraries in a VBA editor**

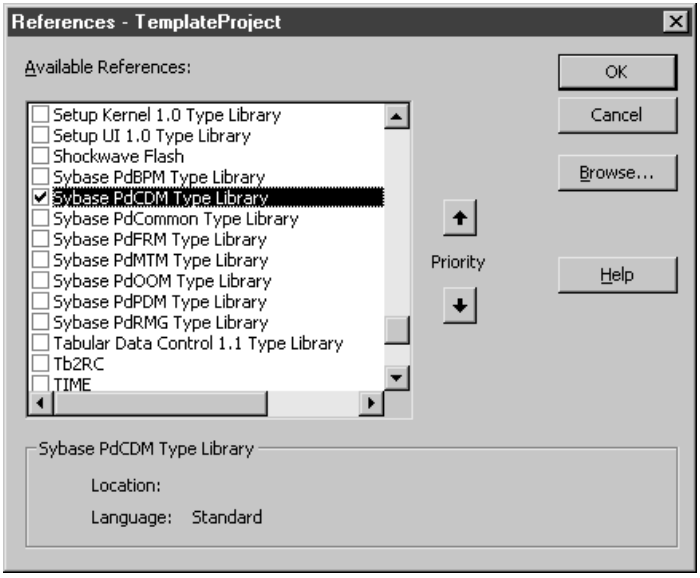
1. Select Tools ➤ References.

❖ **To add references to object type libraries in a Visual Basic editor**

1. Select Project ➤ References.

❖ **To add references to object type libraries in a C# and VB.NET editor**

1. Right-click the project in the project explorer.
2. Select Add References from the contextual menu.



Example of a References window for a VBA program in Word

CHAPTER 9

Customizing PowerDesigner Menus Using Add-Ins

About this chapter

This chapter explains how to customize PowerDesigner menus using add-ins.

Contents

Introducing Customizable Menus and Add-In Types

PowerDesigner add-ins allow you to customize PowerDesigner menus by adding your own menu items.

Customizable menus

You can customize the following menus:

- All contextual menus of objects that are accessible from the Browser or from a symbol in the diagram
- Main menus of each module from each diagram type (i.e. Import, Export, Reverse, Tools, Help)

Types of menu items

You can add the following menu items:

- Commands that call a method script defined using scripting
- Submenus that are cascading menus that appear under a menu item
- Separators that are lines used to organize commands in menus

Available add-ins

An add-in is a module that adds a specific feature or service to PowerDesigner standard behavior. You can use the following add-ins to create menu items in PowerDesigner:

- Customized commands mechanism
- Resource file (XEM, XOL, XDB, XSL or XPL)
- ActiveX
- XML file

☞ For more information on the use of resource files to create custom commands and custom popup menus, see “Defining menus in a profile” in the “Managing Profiles” chapter in the *Advanced User Documentation* .

Add-Ins Overview

Customized commands add-in

This add-in helps you define commands to call executable programs or VB scripts using the Customize Commands dialog box from the Tools application menu.

Commands you define can appear as submenus only in the Execute Commands menu items and in the Import and Export menu items of the File application menu, but not in objects contextual menu. You can hide their display in the menu while keeping their definition.

Usage You should use it when you want to define a unique or very few commands.

☞ For more information on customized commands, see the ""Using customized commands" on page 112" section.

Resource file add-in

This add-in uses a resource file (XEM, XDB, XOL, XPL or XSL) to customize menus by filling them with commands that call method scripts.

Methods and menus are created under the node of their corresponding metaclass, for example the contextual menu of a process is defined under the \Profile\Process\Menu node. You can filter methods and menus using a stereotype or a criterion.

However, the resource file must always be attached to the model in order for the commands to be displayed.

Usage You should use it when you want to define commands for a specific target, such as Sybase Integration Orchestrator or PowerBuilder for example.

☞ For more information on how to customize menus using a resource file, see "Defining methods in a profile" and "Defining menus in a profile" in the *Advanced User Documentation* .

Ease the XML syntax writing

The XML syntax of a menu defined in the Menu page of the resource editor is the same for XML file and ActiveX add-ins. You can use the interface of the resource editor to visualize in the XML page the syntax of a menu you created in the Menu page to help you construct the same XML syntax in your ActiveX or XML file. For more information on XML files, see the ""Using XML file add-in" on page 122" section.

ActiveX add-in

This add-in uses an ActiveX which implements a specific interface that defines methods.

These methods are invoked by PowerDesigner in order to dialog with menus and execute commands that are defined by the ActiveX.

You must enable the ActiveX in PowerDesigner using the Add-ins general option. For more information on this general option, see “Managing add-ins” in the “Using the PowerDesigner Interface” chapter.

Usage You should use it when your plug-in requires more complex interaction between itself and PowerDesigner. Such as enabling and disabling menu items based on object selection, interaction with the windows display environment or for plug-ins written in other languages, such as Visual Basic.NET or C++.

☞ For more information on how to customize menus using an ActiveX, see the ““Using ActiveX add-in” on page 120” section.

XML file add-in

This add-in uses an XML file, which is a simple declarative program with a language linked to an .EXE file or a VB script.

Commands linked to the same applications (for example, ASE, IQ etc.) should be gathered into the same XML file.

You must enable the XML file in PowerDesigner using the Add-ins general option. For more information on this general option, see “Managing add-ins” in the “Using the PowerDesigner Interface” chapter.

Usage You should use it when you want to define several commands that will always be available independently from the target you selected.

☞ For more information on how to customize menus using an XML file, see the ““Using XML file add-in” on page 122” section.

Using customized commands

You can create your own menu items in the PowerDesigner Tools menu to access PowerDesigner objects using your own scripts.

From the Tools application menu, you can add your own submenu entries that will allow you to execute the following commands:

- Executable programs
- VB scripts

You can also gather commands into submenus, modify existing commands, and apply to them keyboard shortcut.

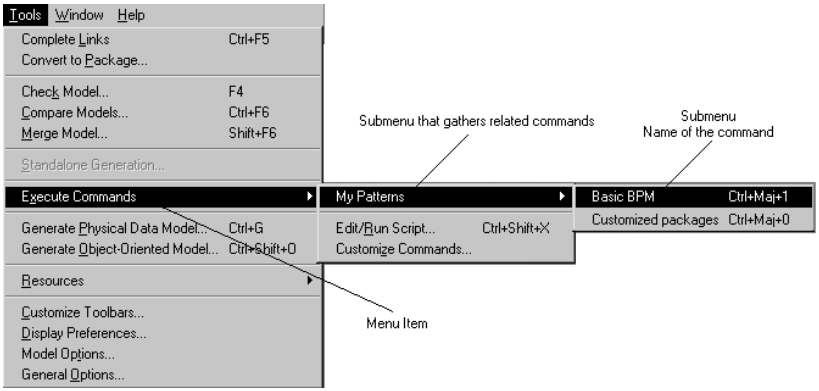
Defining a customized command

You can define commands in the Customize Commands dialog box. The number of commands you can define is limited to 256.

When you define a command, the name you typed for the command is displayed as a submenu entry of the Execute Commands menu item. Command names appear alphabetically sorted.

You can define a context for that command, so it becomes diagram dependent and displays as a submenu only when it is relevant.

The following picture illustrates the result of commands definition performed in the Customize Commands dialog box.



To define a command, you have to specify the following in the Customize commands dialog box:

Command definition	Description
Name	Name of the command that is displayed as a submenu in the Execute Commands menu item. Names are unique and can contain a pick letter (&Generate Java will appear as Generate Java)
Submenu	Name of the submenu that groups commands. It is displayed in the Execute Commands menu item. You can select a default submenu from the list (<None>, Check Model, Export, Generation, Import, Reverse) or create your own submenu that will be added to the listbox. If you select <None> or leave the box empty, the command you defined will be directly added in the submenu of the Execute Commands menu item

Command definition	Description
Context	Optional information that allows the display of the command according to the opened diagram. If you do not define a context for the command, it will appear in the Execute Commands menu item whatever the opened diagram, and even when no diagram is active
Type	Type of the command that you select from the list. It can be an executable or a VB script
Command Line	Path of the command file. The Ellipsis button allows you to browse for a file or any argument. If the command file is a VB script, you can click the button in the toolbar to directly open the scripting editor and preview or edit the script
Comment	Descriptive label for the command. It is displayed in the status bar when you select a command name in the Execute Commands menu item
Show in Menu	Indicates whether the command name should be displayed in the Execute Commands menu item or not. It allows you to disable a command in the menu without deleting the command definition
Keyboard shortcut	Allows you to apply a keyboard shortcut to the command. You can select one from the list. The use of a keyboard shortcut must be unique

Context option

The Context option allows you to define a diagram dependent command that will appear only when the parameters you declared in its definition match the current diagram.

When no matches are found, the command is unavailable.

When you click the Ellipsis button in the Context column of the Customize Commands dialog box, you open the Context Definition dialog box in which you are asked to select the following optional parameters:

Parameter	Description
Model	Allows you to select a model type from the Model list
Diagram	Allows you to select a diagram type for the selected model from the Diagram list
Target resource	Allows you to select or type a XEM file from the Target Resource list, which contains all the XEM files defined for the selected model type. The path button allows you to browse for another particular target resource (XOL, XPL, XSL or XDB) in another folder

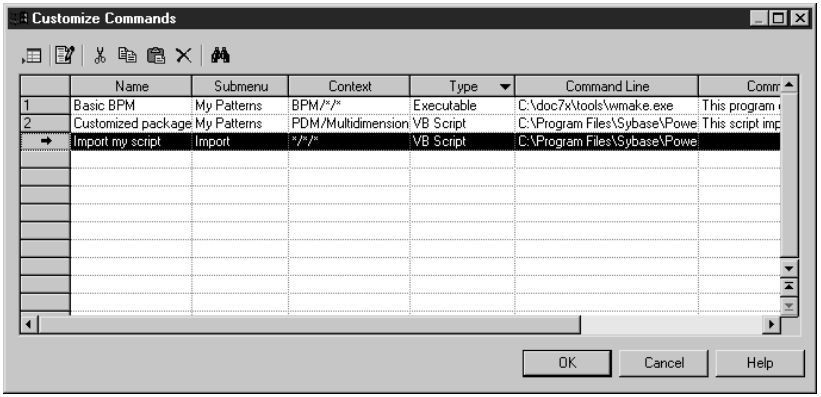
Here are some examples of context definitions as they display in the Context column of the Customize Commands dialog box:

Context definition	Description
//*	Default value. The command is displayed in the Execute Commands menu item whatever the opened diagram, and even when no diagram is active
OOM/*/*	The command is displayed in the Execute Commands menu item whenever an OOM is opened, whatever the opened diagram and the selected target resource
OOM/Class diagram/*	The command is displayed in the Execute Commands menu item whenever an OOM is opened with a class diagram, whatever the selected target resource
OOM/Class diagram/Java	The command is displayed in the Execute Commands menu item whenever an OOM is opened with a class diagram which target resource is Java

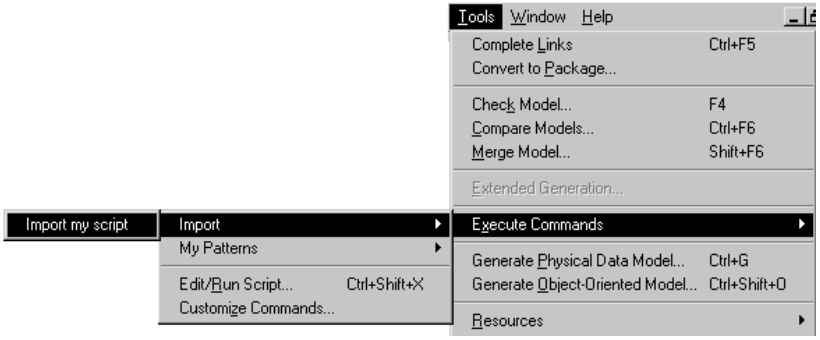
Import/Export submenus

When you select Import or Export in the Submenu list of the Customize Commands dialog box, the command you defined is displayed not only as a submenu entry of the Execute Commands menu item of the Tools menu but also as a submenu entry of the Import or Export menu items of the File menu.

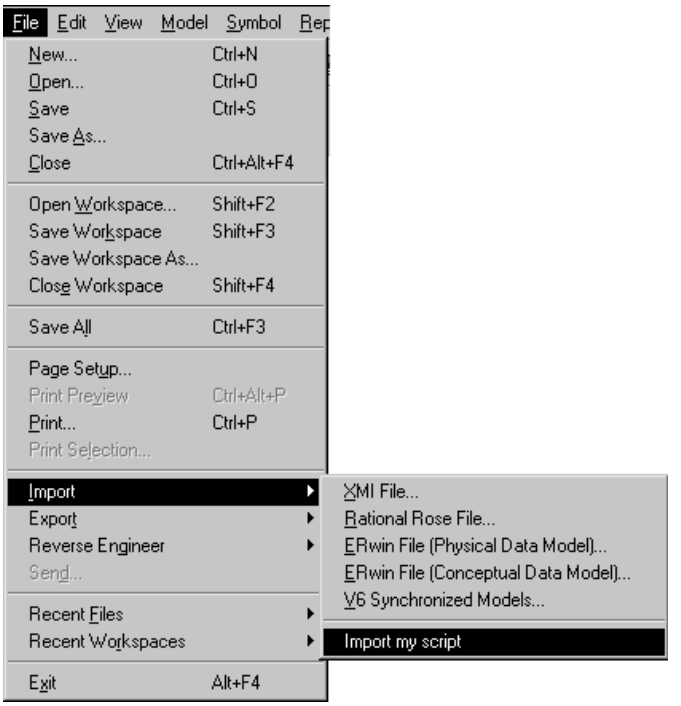
For example you defined the following command in the Customize Commands dialog box:



The command is displayed as follows in the Tools menu:



The command is displayed as follows in the File menu:

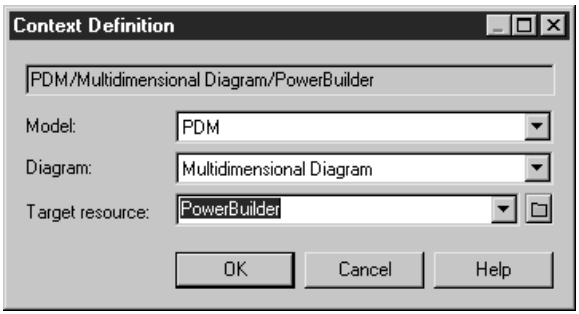


❖ To define a customized command

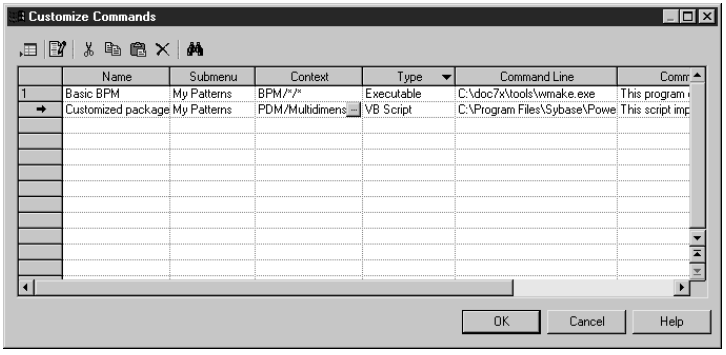
1. Select Tools ➤ Execute Commands ➤ Customize Commands to display the Customize Commands dialog box.
2. Click a blank line in the list.

   or

   Click the Add a row tool.
3. Type a name for the command in the Name column.
4. (Optional) Select a submenu from the list in the Submenu column.
5. (Optional) Define a context by clicking the Ellipsis button in the Context column.



6. Select a type from the list in the Type column.
7. Browse to the directory that contains the command file or argument in the Command Line column.
8. (Optional) Type a comment in the Comment column.
9. Select the Show in Menu check box to display the command name in the menu.
10. (Optional) Select a keyboard shortcut from the list in the Shortcut key column.
11. Click OK.



You can vizualise or modify the command you have just defined by selecting Tools ➤ Execute Commands.

Managing customized commands

Understanding how customized commands are stored in PowerDesigner will allow you to easily plug your programs in PowerDesigner while installing them.

Storage

Customized Commands are saved in the Registry. You can define values for customized commands in the CURRENT USER Registry or in the LOCAL MACHINE Registry.

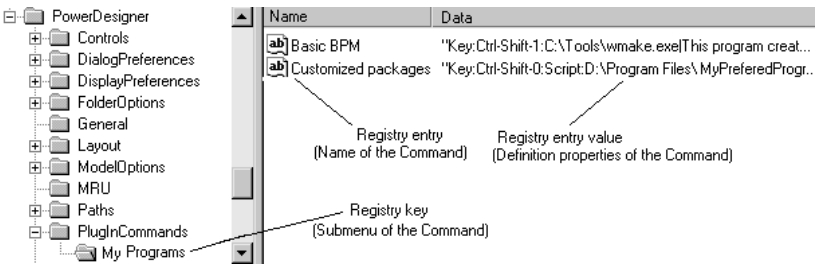
If you define values in the LOCAL MACHINE Registry, customized commands are available for any user of the machine. Thus, when you remove a customized command defined in that Registry from the Customize Commands dialog box, you only remove the line from the list but not the corresponding Registry entry. When you do so, the default value (the one defined in the LOCAL MACHINE Registry) is restored when you open the dialog box again.

The location of customized commands definition can be:

- HKEY_CURRENT_USER\Software\Sybase\PowerDesigner <version>\PlugInCommands
- HKEY_LOCAL_MACHINE\Software\Sybase\PowerDesigner <version>\PlugInCommands

Each customized command is stored in a single Registry string value:

- The name of the customized command is a Registry entry, which has the same name as the command
- The submenu of the customized command is a Registry key, which has the same name as the submenu
- Other command properties are stored in the Value Data field of the Registry entry (Registry entry value)



Definition format

The syntax of the Registry entry is the following:

[Hide:][Key:<key specification>:][Script:]<command>[|comment]

Note that none of the above quoted prefix is localized.

Syntax Keyword	Description
Hide:	Defines the command as hidden

Syntax Keyword	Description
Key:<key specification>:	Allows the association of a keyboard shortcut to the command. This is an optional field. The <key specification> element can include the following optional prefixes in this order: ♦ CTRL- for CONTROL flag ♦ SHIFT- for SHIFT flag Immediately followed by a single character, included between "0-9" (for example:Ctrl-Shift-0)
Script:	Defines the command to be interpreted as an internal script
<Command>	Defines the filename with optional arguments for the command. The command is mandatory and is terminated by a 'l' character. If you want to insert a 'l' character within a command, you must double it
Comment	Describes the command. This is an optional field

Note: The Customize Commands dialog box only supports "Ctrl-Shift-0" to "Ctrl-Shift-9" keyboard shortcuts. If you define a keyboard shortcut outside that range, conflicts with some other built-in keyboard shortcuts may occur and lead to unpredictable results. The reuse of the same keyboard shortcut for two distinct commands may also lead to unpredictable results.

Using ActiveX add-in

You can create your own menu items in PowerDesigner menus by using an ActiveX.

The ActiveX must implement a specific interface called IPDAddIn to become a PowerDesigner add-in.

This interface defines the following methods:

- HRESULT Initialize([in] IDispatch * pApplication)
- HRESULT Uninitialize()
- BSTR ProvideMenuItems([in] BSTR sMenu, [in] IDispatch *pObj)
- BOOL IsCommandSupported([in] BSTR sMenu, [in] IDispatch * pObject, [in] BSTR sCommandName)

- HRESULT DoCommand(in BSTR sMenu, in IDispatch *pObj, in BSTR sCommandName)

Those methods are invoked by PowerDesigner in order to dialog with menus and execute the commands defined by the ActiveX.

Initialize / Uninitialize method

The Initialize method initializes the communication between PowerDesigner and the ActiveX. PowerDesigner starts the communication by providing the ActiveX with a pointer to its application object. The application object allows you to handle the PowerDesigner environment (output window, active model etc.) and must be saved for later reference. The application object type is defined into the PdCommon type library.

The Uninitialize method is used to clean references to PowerDesigner objects. It is called when PowerDesigner is closed and must be used to release all global variables.

ProvideMenuItems method

The ProvideMenuItems method returns an XML text that describes the menu items to add into PowerDesigner menus. The method is invoked each time PowerDesigner needs to display a menu.

When you right-click a symbol in a diagram, this method is called twice: once for the object and once for the symbol. Thus, you can create a method that is only called on graphical contextual menus.

The ProvideMenuItems is called once at the initialization of PowerDesigner to fill the Import and Reverse menus. No object is put in parameter in the method at this moment.

The XML text that describes a menu can use the following elements (DTD):

```
<!ELEMENT Menu (Command | Separator | Popup)*>
<!ELEMENT Command>
<!ATTLIST Command
   Name   CDATA   #REQUIRED
   Caption   CDATA   #REQUIRED
>
<!ELEMENT Separator>
<!ELEMENT PopUp (Command | Separator | Popup)*>
<!ATTLIST PopUp
   Caption   CDATA   #REQUIRED
>
```

Example:

```
ProvideMenuItems ("Object", pModel)
```

The following text results:

```
<MENU>
<POPUP Caption="&Perforce">
   <COMMAND Name="CheckIn" Caption="Check &In"/>
   <SEPARATOR/>
   <COMMAND Name="CheckOut" Caption="Check &Out"/>
</POPUP>
</MENU>
```

Note: This syntax is the same used in the creation of a menu using a resource file.

Ease the XML syntax writing
You can use the interface of the resource editor to visualize in the XML page the syntax of a menu you created in the Menu page that will help you construct the same XML syntax.

☞ For more information on how to customize menus using a resource file, see "Defining methods in a profile" and "Defining menus in a profile" in the *Advanced User Documentation* .

IsCommandSupported method

The IsCommandSupported method allows you to dynamically disable commands defined in a menu. The method must return true to enable a command and false to disable it.

DoCommand method

The DoCommand method implements the execution of a command designated by its name.

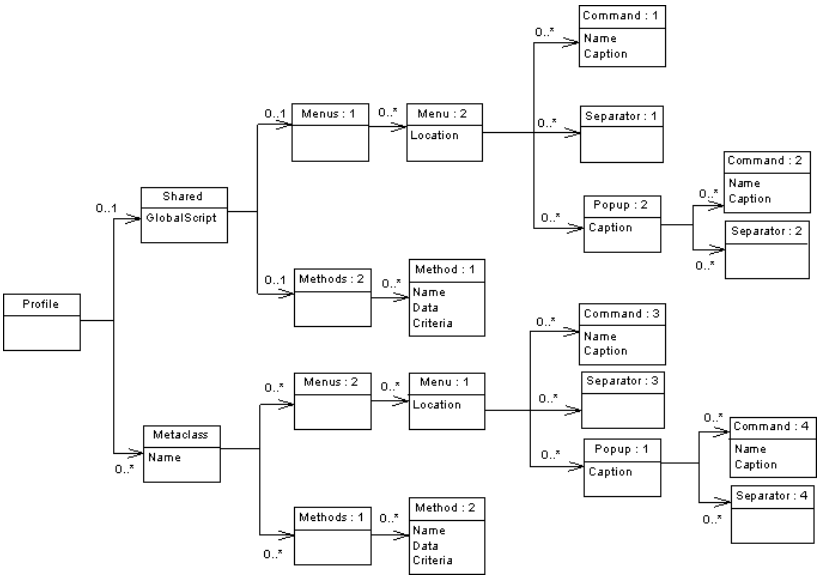
Example:

```
DoCommand ("Object", pModel, "CheckIn")
```

Using XML file add-in

You can create your own menu items in PowerDesigner menus by using an XML file.

The following illustration helps you understand the XML file structure:



The Profile is the root element of the XML file add-in descriptor. It contains the following parts:

- Shared for which menus and commands are defined
- Metaclass which defines commands and menus for a specific metaclass

<!ELEMENT Profile ((Shared)?, (Metaclass)*)>.

Shared

The Shared element defines the menus that are always available and their associated methods (Menus, and Methods elements) and the shared methods (GlobalScript attribute).

The GlobalScript attribute is used to specify an optional global script (VBS) that can contain shared functions.

The Menus element contains menus that are always available for the application. A Location can be specified to define the menu location. It can take the following values:

- FileImport
- File reverse
- Tools
- Help

You can only define one menu per location.

The Methods defines the methods used in the menus described in the Menus element and that are available for the application.

Metaclass

The Metaclass element is used to specify menus that are available for a specific PowerDesigner metaclass. A metaclass is identified by a name. You must use the public name.

The Menus element contains menus available for a metaclass.

The Menu element describes a menu available for a metaclass. It contains a series of commands, separators or popups. A location can be specified to define the menu location. It can take the following values:

- FileExport
- Tools
- Help
- Object

Object is the default value for the Location attribute.

The Methods element contains a series of method available for a metaclass.

The Method element defines a method. A method is identified by a name and a VB script.

The Command element defines a command menu item. Its name must be equal to the name of a Method in order to be implemented.

The Popup element defines a sub-menu item that may contain commands, separators or popups.

The Caption is the displayed value in the menu.

A separator indicates that you want to insert a line in the menu.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Profile>
   <Metaclass Name="PdOOM.Model">
   <Menus>
   <Menu Location="Tools">
      <Popup Caption="Perforce">
         <Command Name="CheckIn" Caption="Check In"/>
         <Separator/>
         <Command Name="CheckOut" Caption="Check Out"/>
      </Popup>
   </Menu>
   </Menus>
   <Methods>
      <Method Name="CheckIn">
Sub %Method%(obj)
execute_command( p4, submit %Filename%, cmd_PipeOutput)
End Sub
      </Method>
      <Method Name="CheckOut">
Sub %Method%(obj)
execute_command( p4, edit %Filename%, cmd_PipeOutput)
End Sub
      </Method>
   </Methods>
</Metaclass>
</Profile>
```

A method defined under a metaclass is supposed to have the current object as parameter; its name is calculated from the attribute name of the method tag.

Example:

```
<Method Name="ToInt" >
Sub %Method%(obj)
   Print obj
   ExecuteCommand(&quot;%MORPHEUS%\ToInt.vbs&quot;,
         &quot;&quot;, cmd_InternalScript)
End Sub
```

Each metaclass name must be prefixed by its Type Library public name like PdOOM.Class.

Inheritance is taken into account: a menu defined on the metaclass PdCommon.NamedObject will be available for a PdOOM.Class.

You can only define one menu for a given location. If you define several locations only the last one will be preserved.

Menus defined in the Shared section can refer to “FileImport” “Reverse” and “Help” locations.

These menus can only refer to method defined under Shared and no object is put in parameter in the methods defined under Shared.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Profile>
   <Shared>
   <GlobalScript>
Option Explicit
Function Print (obj)
Output obj.classname &amp; &quot; &quot; &amp; obj.name
End Function
   /GlobalScript>
   </Shared>
   <Metaclass Name="PdOOM.Class">
   <Menus>
   <Menu>
      <Popup Caption="Transformation">
         <Command Name="ToInt" Caption="Convert to interface"/>
         <Separator/>
      </Popup>
   </Menu>
   </Menus>
   <Methods>
      <Method Name="ToInt" >
Sub %Method%(obj)
   Print obj
   ExecuteCommand(&quot;%MORPHEUS%\ToInt.vbs&quot;,
         &quot;&quot;, cmd_InternalScript)
End Sub
      </Method>
   </Methods>
   </Metaclass>
</Profile>
```

You can find the DTD in the Add-ins folder of the PowerDesigner directory.

Note: You can retrieve in this example the same syntax used in the creation of a menu using a resource file.

Ease the XML syntax writing
You can use the interface of the resource editor to visualize in the XML page the syntax of a menu you created in the Menu page that will help you construct the same XML syntax.

☞ For more information on how to customize menus using a resource file, see “Defining methods in a profile” and “Defining menus in a profile” in the *Advanced User Documentation* .

☞

Index

O

P

R

S