

# **Object-Oriented Architecture**

## **SAP PowerDesigner Documentation Collection**



# Content

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Building OOMs . . . . .</b>                          | <b>7</b> |
| 1.1      | Getting Started with Object-Oriented Modeling . . . . . | 7        |
|          | Creating an OOM . . . . .                               | 9        |
|          | Previewing Object Code . . . . .                        | 11       |
|          | Customizing Object Creation Scripts . . . . .           | 14       |
|          | Customizing your Modeling Environment . . . . .         | 15       |
| 1.2      | Use Case Diagrams . . . . .                             | 20       |
|          | Use Case Diagram Objects . . . . .                      | 21       |
|          | Use Cases (OOM) . . . . .                               | 22       |
|          | Actors (OOM) . . . . .                                  | 24       |
|          | Use Case Associations (OOM) . . . . .                   | 28       |
| 1.3      | Structural Diagrams . . . . .                           | 30       |
|          | Class Diagrams . . . . .                                | 31       |
|          | Composite Structure Diagrams . . . . .                  | 33       |
|          | Package Diagrams . . . . .                              | 35       |
|          | Object Diagrams . . . . .                               | 36       |
|          | Classes (OOM) . . . . .                                 | 38       |
|          | Packages (OOM) . . . . .                                | 57       |
|          | Interfaces (OOM) . . . . .                              | 60       |
|          | Objects (OOM) . . . . .                                 | 62       |
|          | Attributes (OOM) . . . . .                              | 67       |
|          | Identifiers (OOM) . . . . .                             | 75       |
|          | Operations (OOM) . . . . .                              | 77       |
|          | Associations (OOM) . . . . .                            | 84       |
|          | Generalizations (OOM) . . . . .                         | 93       |
|          | Dependencies (OOM) . . . . .                            | 96       |
|          | Realizations (OOM) . . . . .                            | 99       |
|          | Require Links (OOM) . . . . .                           | 100      |
|          | Annotations (OOM) . . . . .                             | 102      |
|          | Instance Links (OOM) . . . . .                          | 105      |
|          | Domains (OOM) . . . . .                                 | 109      |
| 1.4      | Dynamic Diagrams . . . . .                              | 114      |
|          | Communication Diagrams . . . . .                        | 114      |
|          | Sequence Diagrams . . . . .                             | 116      |
|          | Activity Diagrams . . . . .                             | 120      |
|          | Statechart Diagrams . . . . .                           | 123      |

|  |     |
|--|-----|
| Interaction Overview Diagrams . . . . .                                | 126 |
| Interaction References and Interaction Activities (OOM) . . . . .      | 127 |
| Interaction Fragments (OOM) . . . . .                                  | 128 |
| Messages (OOM) . . . . .   | 131 |
| Activities (OOM) . . . . .   | 145 |
| Organization Units (OOM) . . . . .                                     | 160 |
| Starts and Ends (OOM) . . . . .  | 167 |
| Decisions (OOM) . . . . .  | 168 |
| Synchronizations (OOM) . . . . .                                       | 171 |
| Flows (OOM) . . . . .  | 173 |
| Object Nodes (OOM) . . . . .   | 175 |
| States (OOM) . . . . .   | 177 |
| Transitions (OOM) . . . . .  | 182 |
| Events (OOM) . . . . .   | 185 |
| Actions (OOM) . . . . .  | 188 |
| Junction Points (OOM) . . . . .  | 191 |
| 1.5 Implementation Diagrams . . . . .                                  | 192 |
| Component Diagrams . . . . .   | 192 |
| Deployment Diagrams . . . . .  | 194 |
| Components (OOM) . . . . .   | 196 |
| Nodes (OOM) . . . . .  | 203 |
| Component Instances (OOM) . . . . .                                    | 205 |
| Files (OOM) . . . . .  | 207 |
| Node Associations (OOM) . . . . .                                      | 209 |
| 1.6 Web Services . . . . .   | 211 |
| Web Service Components (OOM) . . . . .                                 | 212 |
| Web Service Methods (OOM) . . . . .                                    | 218 |
| Web Service Component Instances (OOM) . . . . .                        | 224 |
| Generating Web Services for Java . . . . .                             | 226 |
| Generating Web Services for .NET . . . . .                             | 227 |
| Importing WSDL Files . . . . .   | 229 |
| 1.7 Generating and Reverse Engineering OO Source Files . . . . .       | 233 |
| Generating OO Source Files from an OOM . . . . .                       | 233 |
| Reverse Engineering OO Source Files into an OOM . . . . .              | 237 |
| Synchronizing a Model with Generated Files . . . . .                   | 240 |
| 1.8 Generating Other Models from an OOM . . . . .                      | 243 |
| Managing Object Persistence During Generation of Data Models . . . . . | 245 |
| Managing Persistence for Generalizations . . . . .                     | 245 |
| Managing Persistence for Complex Data Types . . . . .                  | 246 |
| Customizing XSM Generation for Individual Objects . . . . .            | 249 |
| 1.9 Checking an OOM . . . . .  | 251 |

|   |     |
|---|-----|
| Domain Checks.                                    | 252 |
| Data Source Checks.                               | 252 |
| Package Checks.                                   | 253 |
| Actor/Use Case Checks.                            | 254 |
| Class Checks.                                     | 255 |
| Identifier Checks.                                | 259 |
| Interface Checks.                                 | 260 |
| Class/Interface Attribute Checks.                 | 262 |
| Class/Interface Operation Checks.                 | 263 |
| Realization Checks.                               | 265 |
| Generalization Checks.                            | 265 |
| Object Checks.                                    | 266 |
| Instance Link Checks.                             | 267 |
| Message Checks.                                   | 267 |
| State Checks.                                     | 267 |
| State Action Checks.                              | 268 |
| Event Checks.                                     | 269 |
| Junction Point Checks.                            | 270 |
| Activity Checks.                                  | 270 |
| Decision Checks.                                  | 271 |
| Object Node Checks.                               | 272 |
| Organization Unit Checks.                         | 273 |
| Start/End Checks.                                 | 273 |
| Synchronization Checks.                           | 274 |
| Transition and Flow Checks.                       | 275 |
| Component Checks.                                 | 275 |
| Node Checks.                                      | 276 |
| Data Format Checks.                               | 277 |
| Component Instance Checks.                        | 277 |
| Interaction Reference Checks.                     | 278 |
| Class Part Checks.                                | 279 |
| Class/Component Port Checks.                      | 280 |
| Class/component Assembly Connector Checks.        | 280 |
| Association Checks.                               | 281 |
| Activity Input and Output Parameter Checks.       | 281 |
| 1.10 Importing a Rational Rose Model into an OOM. | 282 |
| Importing Rational Rose Use Case Diagrams.        | 284 |
| Importing Rational Rose Class Diagrams.           | 284 |
| Importing Rational Rose Collaboration Diagrams.   | 286 |
| Importing Rational Rose Sequence Diagrams.        | 286 |
| Importing Rational Rose Statechart Diagrams.      | 287 |

|   |            |
|---|------------|
| Importing Rational Rose Activity Diagrams . . . . .         | 288        |
| Importing Rational Rose Component Diagrams . . . . .        | 289        |
| Importing Rational Rose Deployment Diagrams . . . . .       | 290        |
| 1.11 Importing and Exporting an OOM in XMI Format . . . . . | 290        |
| Importing XMI Files . . . . .                               | 290        |
| Exporting XMI Files . . . . .                               | 291        |
| <b>2 Object Language Definition Reference . . . . .</b>     | <b>292</b> |
| 2.1 Java . . . . .  | 292        |
| Java Public Classes . . . . .                               | 292        |
| Java Enumerated Types (Enums) . . . . .                     | 292        |
| JavaDoc Comments . . . . .                                  | 295        |
| Java 5.0 Annotations . . . . .                              | 301        |
| Java Strictfp Keyword . . . . .                             | 301        |
| Enterprise Java Beans (EJBs) V2 . . . . .                   | 302        |
| Enterprise Java Beans (EJBs) V3 . . . . .                   | 325        |
| Java Servlets . . . . .                                     | 333        |
| Java Server Pages (JSPs) . . . . .                          | 343        |
| Generating Java Files . . . . .                             | 352        |
| Reverse Engineering Java Code . . . . .                     | 355        |
| 2.2 Technical Architecture Modeling (TAM) . . . . .         | 359        |
| Block Diagrams (TAM) . . . . .                              | 360        |
| 2.3 Eclipse Modeling Framework (EMF) . . . . .              | 362        |
| Generating EMF Files . . . . .                              | 364        |
| Reverse Engineering EMF Files . . . . .                     | 365        |
| 2.4 PowerBuilder . . . . .                                  | 366        |
| PowerBuilder Objects . . . . .                              | 366        |
| Generating PowerBuilder Objects . . . . .                   | 369        |
| Reverse Engineering PowerBuilder . . . . .                  | 371        |
| 2.5 VB .NET . . . . .                                       | 376        |
| Inheritance & Implementation . . . . .                      | 376        |
| Namespace . . . . .   | 377        |
| Project . . . . .   | 377        |
| Accessibility . . . . .                                     | 378        |
| Classes, Interfaces, Structs, and Enumerations . . . . .    | 378        |
| Module . . . . .  | 380        |
| Custom Attributes . . . . .                                 | 380        |
| Shadows . . . . .   | 381        |
| Variables . . . . .   | 382        |
| Property . . . . .  | 382        |
| Method . . . . .  | 384        |
| Constructor & Destructor . . . . .                          | 385        |

|  |     |
|--|-----|
| Delegate .....   | 386 |
| Event .....  | 387 |
| Event Handler .....  | 388 |
| External Method .....  | 388 |
| Generating VB.NET Files .....  | 388 |
| Reverse Engineering VB .NET .....  | 391 |
| Working with ASP.NET .....   | 397 |
| 2.6 C# 2.0 .....   | 402 |
| C# 2.0 Assemblies .....  | 403 |
| C# 2.0 Compilation Units .....   | 405 |
| C# 2.0 Namespaces .....  | 407 |
| C# 2.0 Classes .....   | 407 |
| C# 2.0 Interfaces .....  | 409 |
| C# 2.0 Structs .....   | 409 |
| C# 2.0 Delegates .....   | 410 |
| C# 2.0 Enums .....   | 411 |
| C# 2.0 Fields .....  | 412 |
| C# 2.0 Methods .....   | 413 |
| C# 2.0 Events, Indexers, and Properties .....                              | 416 |
| C# 2.0 Inheritance and Implementation .....                                | 419 |
| C# 2.0 Custom Attributes .....   | 419 |
| Generating C# 2.0 Files .....  | 420 |
| Reverse Engineering C# 2.0 Code .....                                      | 422 |
| 2.7 C++ .....  | 427 |
| Designing for C++ .....  | 427 |
| Generating for C++ .....   | 429 |
| 2.8 Object/Relational (O/R) Mapping .....                                  | 430 |
| Top-Down: Mapping Classes to Tables .....                                  | 430 |
| Bottom-Up: Mapping Tables to Classes .....                                 | 440 |
| Meet in the Middle: Manually Mapping Classes to Tables .....               | 442 |
| 2.9 Generating Persistent Objects for Java and JSF Pages .....             | 463 |
| Generating Hibernate Persistent Objects .....                              | 463 |
| Generating EJB 3 Persistent Objects .....                                  | 482 |
| Generating JavaServer Faces (JSF) for Hibernate .....                      | 495 |
| 2.10 Generating .NET 2.0 Persistent Objects and Windows Applications ..... | 508 |
| Generating ADO.NET and ADO.NET CF Persistent Objects .....                 | 510 |
| Generating NHibernate Persistent Objects .....                             | 519 |
| Configuring Connection Strings .....                                       | 535 |
| Generating Code for Unit Testing .....                                     | 537 |
| Generating Windows or Smart Device Applications .....                      | 541 |

# 1 Building OOMs

The chapters in this part explain how to model your information systems in SAP® PowerDesigner®.

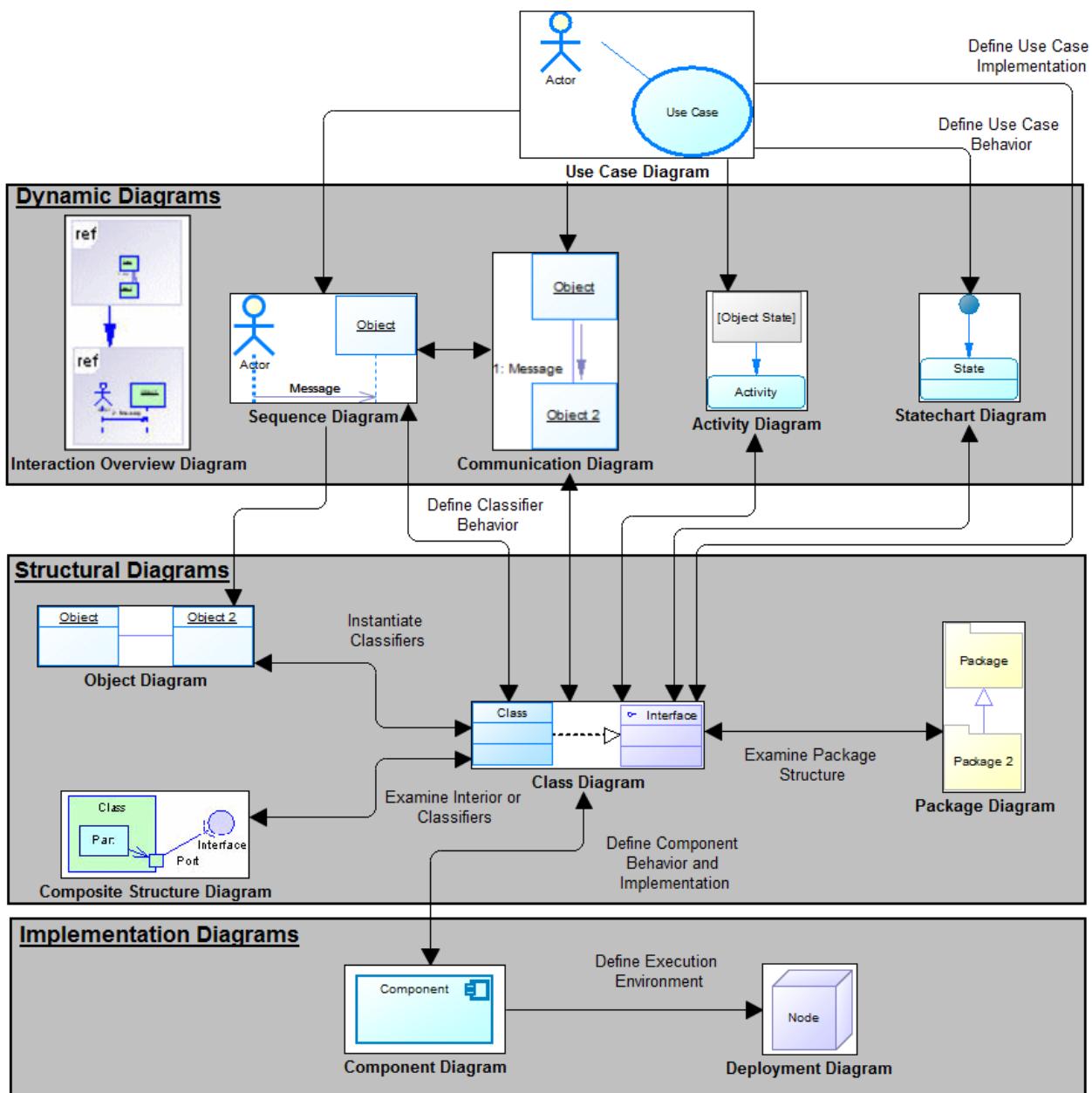
## 1.1 Getting Started with Object-Oriented Modeling

An object-oriented model (OOM) helps you analyze an information system through use cases, structural and behavioral analyses, and in terms of deployment, using the Unified Modeling Language (UML). You can model, reverse-engineer, and generate for Java, .NET and other languages.

SAP® PowerDesigner® supports the following UML diagrams:

- Use case diagram ( ) - see [Use Case Diagrams \[page 20\]](#)
- Structural Diagrams:
  - Class diagram ( ) - see [Class Diagrams \[page 31\]](#)
  - Composite structure diagram ( ) - see [Composite Structure Diagrams \[page 33\]](#)
  - Object diagram ( ) - see [Object Diagrams \[page 36\]](#)
  - Package diagram ( ) - see [Package Diagrams \[page 35\]](#)
- Dynamic Diagrams:
  - Communication diagram ( ) - see [Communication Diagrams \[page 114\]](#)
  - Sequence diagram ( ) - see [Sequence Diagrams \[page 116\]](#)
  - Activity diagram ( ) - see [Activity Diagrams \[page 120\]](#)
  - Statechart diagram ( ) - see [Statechart Diagrams \[page 123\]](#)
  - Interaction overview diagram ( ) - see [Interaction Overview Diagrams \[page 126\]](#)
- Implementation Diagrams:
  - Component diagram ( ) - see [Component Diagrams \[page 192\]](#)
  - Deployment diagram ( ) - see [Deployment Diagrams \[page 194\]](#)

In the picture below, you can see how the various UML diagrams can interact within your model:



## Suggested Bibliography

- James Rumbaugh, Ivar Jacobson, Grady Booch – The Unified Modeling Language Reference Manual – Addison Wesley, 1999
- Grady Booch, James Rumbaugh, Ivar Jacobson – The Unified Modeling Language User Guide – Addison Wesley, 1999
- Ivar Jacobson, Grady Booch, James Rumbaugh – The Unified Software Development Process – Addison Wesley, 1999

- Doug Rosenberg, Kendall Scott – Use Case Driven Object Modeling With UML A Practical Approach – Addison Wesley, 1999
- Michael Blaha, William Premerlani – Object-Oriented Modeling and Design for Database Applications – Prentice Hall, 1998
- Geri Schneider, Jason P. Winters, Ivar Jacobson – Applying Use Cases: A Practical Guide – Addison Wesley, 1998
- Pierre-Alain Muller – Instant UML – Wrox Press Inc, 1997
- Bertrand Meyer – Object-Oriented Software Construction – Prentice Hall, 2nd Edition, 1997
- Martin Fowler, Kendall Scott – UML Distilled Applying The Standard Object Modeling Language – Addison Wesley, 1997

### 1.1.1 Creating an OOM

You create a new object-oriented model by selecting  [File](#)  [New Model](#).

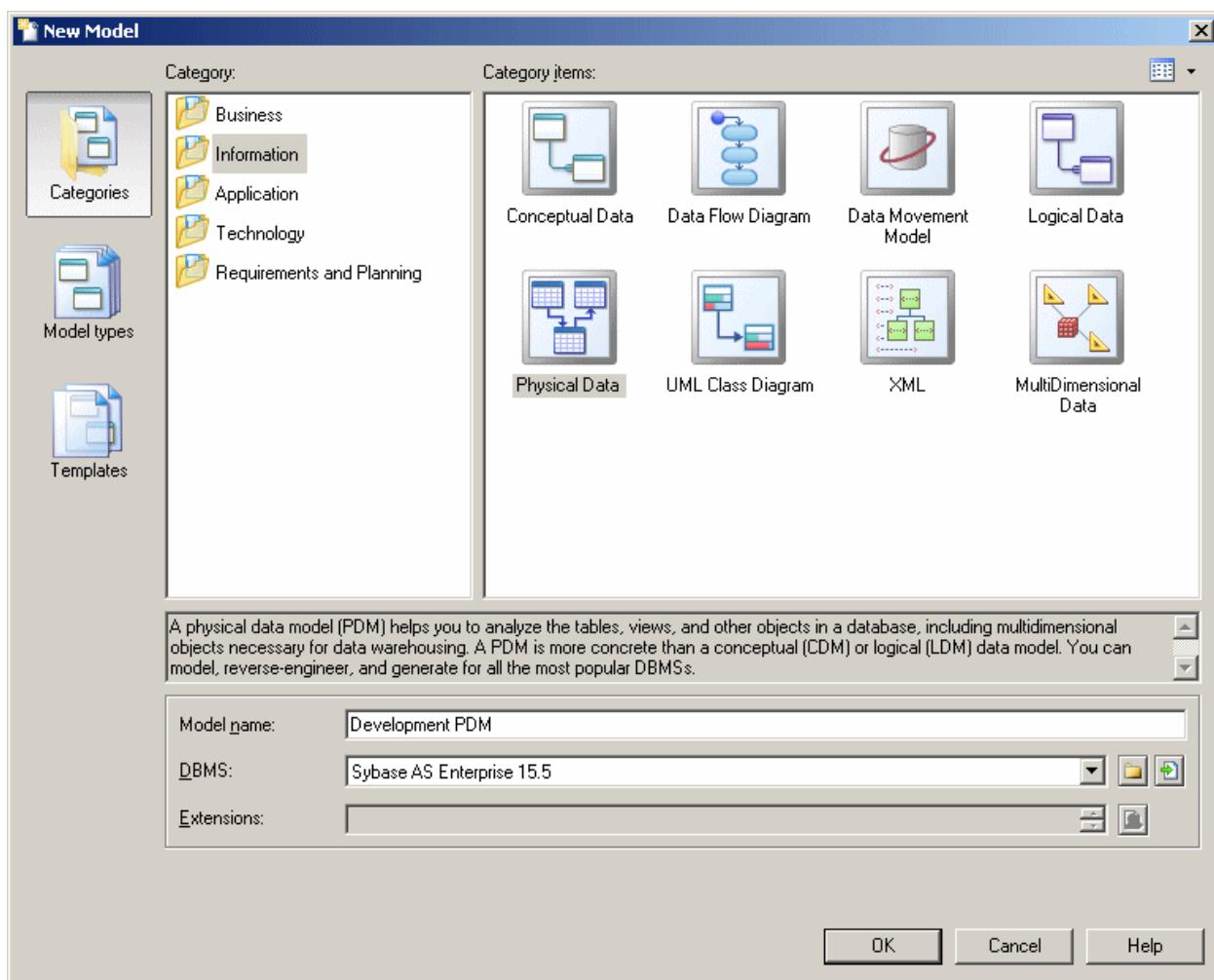
#### Context

##### Note

In addition to creating an OOM from scratch with the following procedure, you can also reverse-engineer a model from existing OO code (see [Reverse Engineering OO Source Files into an OOM \[page 237\]](#)).

The New Model dialog is highly configurable, and your administrator may hide options that are not relevant for your work or provide templates or predefined models to guide you through model creation. When you open the dialog, one or more of the following buttons will be available on the left hand side:

- [Categories](#) - which provides a set of predefined models and diagrams sorted in a configurable category structure.
- [Model types](#) - which provides the classic list of PowerDesigner model types and diagrams.
- [Template files](#) - which provides a set of model templates sorted by model type.



## Procedure

1. Select **File > New Model** to open the New Model dialog.
2. Click a button, and then select a category or model type (*Object-Oriented Model*) in the left-hand pane.
3. Select an item in the right-hand pane. Depending on how your New Model dialog is configured, these items may be first diagrams or templates on which to base the creation of your model.  
Use the *Views* tool on the upper right hand side of the dialog to control the display of the items.
4. Enter a model name. The code of the model, which is used for script or code generation, is derived from this name using the model naming conventions.
5. Select a target object language, which customizes PowerDesigner's default modifying environment with target-specific properties, objects, and generation templates.

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

6. [optional] Click the *Select Extensions* button and attach one or more extensions to your model.
7. Click *OK* to create and open the object-oriented model .

**i Note**

Sample OOMs are available in the Example Directory.

### 1.1.1.1 OOM Properties

You open the model property sheet by right-clicking the model in the Browser and selecting *Properties*.

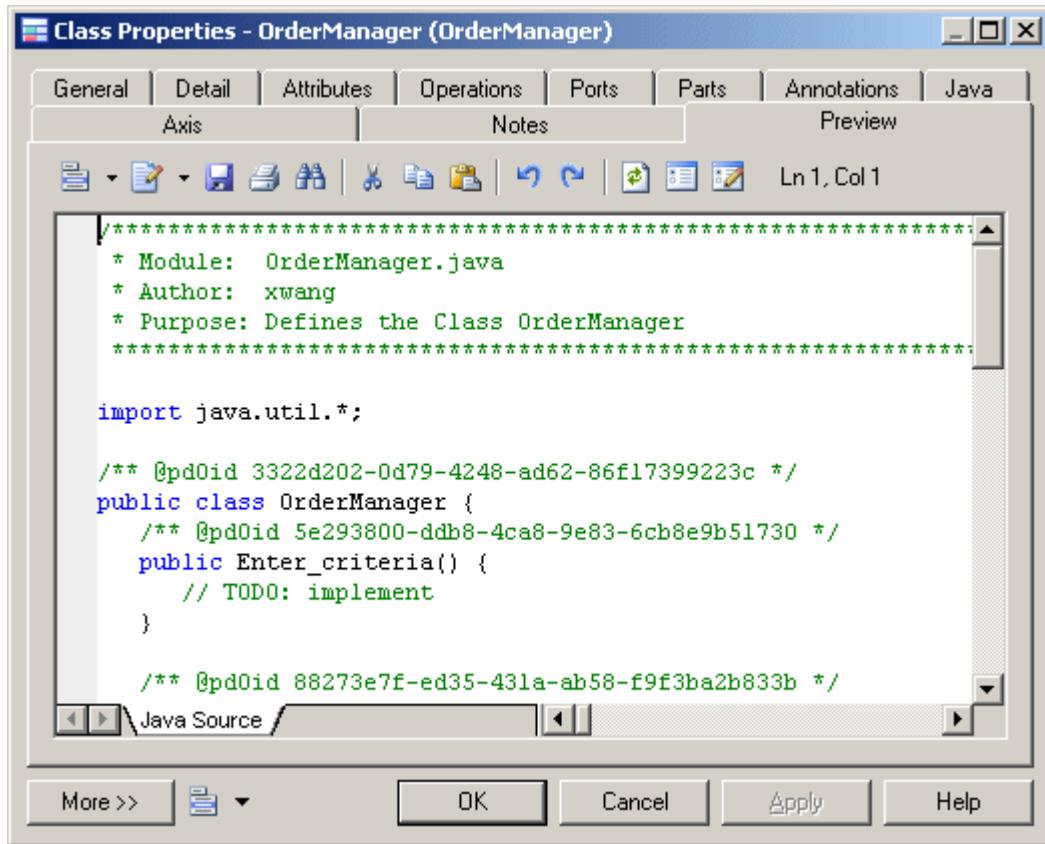
Each object-oriented model has the following model properties:

| Property             | Description  |
|----------------------|--|
| Name/Code/Comment    | Identify the model. The name should clearly convey the model's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the model. By default the code is auto-generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Filename             | Specifies the location of the model file. This box is empty if the model has never been saved.   |
| Author               | Specifies the author of the model. If you enter nothing, the Author field in diagram title boxes displays the user name from the model property sheet Version Info tab. If you enter a space, the Author field displays nothing.   |
| Version / Repository | Specify a user-defined version name and the read-only repository version number of the model. You can control which of these version values is displayed in a diagram Title symbol through the display preferences for the Title symbol.   |
| Object language      | Specifies the model target.  |
| Default diagram      | Specifies the diagram displayed by default when you open the model.  |
| Keywords             | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

### 1.1.2 Previewing Object Code

Click the *Preview* tab in the property sheet of the model, packages, classes, and various other model objects in order to view the code that will be generated for it.

For example, if you have created EJB or servlet components in Java, the Preview tab displays the EJB or Web deployment descriptor files. If you have selected an XML family language, the Preview tab displays the Schema file that corresponds to the XML file to be generated.



If you have selected the *Preview Editable* option (available from **Tools > Model Options**), you can modify the code of a classifier directly from its *Preview* tab. The modified code must be valid and apply only to the present classifier or your modifications will be ignored. You can create generalization and realization links if their classifiers already exist in the model, but you cannot rename the classifier or modify the package declaration to move it to another package. You should avoid renaming attributes and operations, as any other properties that are not generated (such as description, annotation or extended attributes) will be lost. Valid changes are applied when you leave the *Preview* tab or click the *Apply* button.

In a model targeting SAP® PowerBuilder®, this feature can be used to provide a global vision of the code of an object and its functions which is not available in PowerBuilder. You can use the *Preview* tab to check where instance variables are used in the code. You can also modify the body of a function or create a new function from an existing function using copy/paste.

The following tools are available on the *Preview* tab toolbar:

| Tools   | Description   |
|---|---|
|    | <p><i>Editor Menu</i> (<code>Shift</code> + <code>F11</code>) - Contains the following commands:</p> <ul style="list-style-type: none"> <li>• <i>New</i> (<code>Ctrl</code> + <code>N</code>) - Reinitializes the field by removing all the existing content.</li> <li>• <i>Open...</i> (<code>Ctrl</code> + <code>O</code>) - Replaces the content of the field with the content of the selected file.</li> <li>• <i>Insert...</i> (<code>Ctrl</code> + <code>I</code>) - Inserts the content of the selected file at the cursor.</li> <li>• <i>Save</i> (<code>Ctrl</code> + <code>S</code>) - Saves the content of the field to the specified file.</li> <li>• <i>Save As...</i> - Saves the content of the field to a new file.</li> <li>• <i>Select All</i> (<code>Ctrl</code> + <code>A</code>) - Selects all the content of the field.</li> <li>• <i>Find...</i> (<code>Ctrl</code> + <code>F</code>) - Opens a dialog to search for text in the field.</li> <li>• <i>Find Next...</i> (<code>F3</code>) - Finds the next occurrence of the searched for text.</li> <li>• <i>Find Previous...</i> (<code>Shift</code> + <code>F3</code>) - Finds the previous occurrence of the searched for text.</li> <li>• <i>Replace...</i> (<code>Ctrl</code> + <code>H</code>) - Opens a dialog to replace text in the field.</li> <li>• <i>Go To Line...</i> (<code>Ctrl</code> + <code>G</code>) - Opens a dialog to go to the specified line.</li> <li>• <i>Toggle Bookmark</i> (<code>Ctrl</code> + <code>F2</code>) - Inserts or removes a bookmark (a blue box) at the cursor position. Note that bookmarks are not printable and are lost if you refresh the tab, or use the <i>Show Generation Options</i> tool.</li> <li>• <i>Next Bookmark</i> (<code>F2</code>) - Jumps to the next bookmark.</li> <li>• <i>Previous Bookmark</i> (<code>Shift</code> + <code>F2</code>) - Jumps to the previous bookmark.</li> </ul> |
|  | <p><i>Edit With</i> (<code>Ctrl</code> + <code>E</code>) - Opens the previewed code in an external editor. Click the down arrow to select a particular editor or <i>Choose Program</i> to specify a new editor. Editors specified here are added to the list of editors available at ► <i>Tools</i> &gt; <i>General Options</i> &gt; <i>Editors</i> ▾.</p>  |
|  | <i>Save</i> ( <code>Ctrl</code> + <code>S</code> ) - Saves the content of the field to the specified file.  |
|  | <i>Print</i> ( <code>Ctrl</code> + <code>P</code> ) - Prints the content of the field.  |
|  | <i>Find</i> ( <code>Ctrl</code> + <code>F</code> ) - Opens a dialog to search for text.   |
|  | <i>Cut</i> ( <code>Ctrl</code> + <code>X</code> ), <i>Copy</i> ( <code>Ctrl</code> + <code>C</code> ), and <i>Paste</i> ( <code>Ctrl</code> + <code>V</code> ) - Perform the standard clipboard actions.  |
|  | <i>Undo</i> ( <code>Ctrl</code> + <code>Z</code> ) and <i>Redo</i> ( <code>Ctrl</code> + <code>Y</code> ) - Move backward or forward through edits.   |
|  | <p><i>Refresh</i> (<code>F5</code>) - Refreshes the Preview tab.</p> <p>You can debug the GTL templates that generate the code shown in the Preview tab. To do so, open the target or extension resource file, select the <i>Enable Trace Mode</i> option, and click <i>OK</i> to return to your model. You may need to click the <i>Refresh</i> tool to display the templates.</p>   |
|  | <i>Select Generation Targets</i> ( <code>Ctrl</code> + <code>F6</code> ) - Lets you select additional generation targets (defined in extensions), and adds a sub-tab for each selected target. For information about generation targets, see <i>Customizing and Extending PowerDesigner</i> > <i>Extension Files</i> > <i>Generated Files (Profile)</i> > <i>Generating Your Files in a Standard or Extended Generation</i> .   |

| Tools | Description   |
|-------|---|
|       | <i>Show Generation Options</i> ( <code>Ctrl</code> + <code>W</code> ) - Opens the Generation Options dialog, allowing you to modify the generation options and to see the impact on the code. This feature is especially useful when you are working with Java. For other object languages, generation options do not influence the code. |

### 1.1.3 Customizing Object Creation Scripts

The Script tab allows you to customize the object's creation script by, for example, adding descriptive information about the script.

#### Examples

For example, if a project archives all generated creation scripts, a header can be inserted before each creation script, indicating the date, time, and any other appropriate information or, if generated scripts must be filed using a naming system other than the script name, a header could direct a generated script to be filed under a different name.

You can insert scripts at the beginning (*Header* subtab) and the end (*Footer* subtab) of a script or insert scripts before and after a class or interface creation command (*Imports* subtab)

The following tools and shortcut keys are available on the *Script* tab:

| Tool | Description   |
|------|---|
|      | ( <code>Shift</code> + <code>F11</code> ) Opens the <i>Editor Contextual menu</i>   |
|      | <i>Edit With</i> ( <code>Ctrl</code> + <code>E</code> ) - Opens your default editor.  |
|      | <i>Import Folder</i> - [ <i>Imports</i> sub-tab] Opens a selection window to select packages to import to the cursor position, prefixed by the keyword 'import'.        |
|      | <i>Import Classifier</i> - [ <i>Imports</i> sub-tab] Opens a selection window to select classifiers to import to the cursor position, prefixed by the keyword 'import'. |

You can use the following formatting syntax with variables:

| Format code | Format of variable value in script |
|-------------|------------------------------------|
| .L          | Lowercase characters               |
| .T          | Removes blank spaces               |
| .U          | Uppercase characters               |

| Format code | Format of variable value in script                            |
|-------------|---|
| .C          | Upper-case first letter and lower-case next letters           |
| .<n>        | Maximum length where n is the number of characters            |
| .<n>J       | Justifies to fixed length where n is the number of characters |

You embed formatting options in variable syntax as follows:

```
%.<format>:<variable>%
```

## 1.1.4 Customizing your Modeling Environment

The PowerDesigner object-oriented model provides various means for customizing and controlling your modeling environment.

### 1.1.4.1 Setting OOM Model Options

You can set OOM model options by selecting **Tools** **Model Options** or right-clicking the diagram background and selecting **Model Options**. These options affect all the objects in the model, including those already created.

You can set the following options:

| Option                       | Definition  |
|------------------------------|---|
| Name/Code case sensitive     | Specifies that the names and codes for all objects are case sensitive, allowing you to have two objects with identical names or codes but different cases in the same model. If you change case sensitivity during the design process, we recommend that you check your model to verify that your model does not contain any duplicate objects. |
| Enable links to requirements | Displays a Requirements tab in the property sheet of every object in the model, which allows you to attach requirements to objects (see <i>Requirements Management</i> ).   |
| Show classes as data types   | Includes classes of the model in the list of data types defined for attributes or parameters, and return types defined for operations.  |
| Preview editable             | Applies to reverse engineering. You can edit your code from the Preview page of a class or an interface by selecting the Preview Editable check box. This allows you to reverse engineer changes applied to your code directly from the Preview page.   |

| Option  | Definition   |
|---|--|
| External Shortcut Properties                        | <p>Specifies the properties that are stored for external shortcuts to objects in other models for display in property sheets and on symbols. By default, <i>All</i> properties appear, but you can select to display only <i>Name/Code</i> to reduce the size of your model.</p> <p><b>i Note</b></p> <p>This option only controls properties of external shortcuts to models of the same type (PDM to PDM, EAM to EAM, etc). External shortcuts to objects in other types of model can show only the basic shortcut properties.</p> |
| Default Data Types                                  | <p>Specifies default data types for attributes, operations, and parameters.</p> <p>If you type a data type value that does not exist in the BasicDataTypes and AdditionalDataTypes lists of the object language, then the value of the DefaultDataType entry is used. For more information on data types in the object language, see <i>Customizing and Extending PowerDesigner &gt; Object, Process, and XML Language Definition Files &gt; Settings Category: Object Language</i>.</p>   |
| Domain/Attribute: Enforce non-divergence            | <p>Specifies that attributes attached to a domain must remain synchronized with the properties of that domain. You can specify any or all of:</p> <ul style="list-style-type: none"> <li>• Data type – data type, length, and precision</li> <li>• Check – check parameters, such as minimum and maximum values</li> <li>• Rules – business rules</li> </ul>   |
| Domain/Attribute: Use data type full name           | <p>Specifies that the full data type name is used for attribute data types instead of its abbreviated form. Provides a clear persistent data type list for attributes.</p>   |
| Default Association Container                       | <p>Specifies a default container for associations that have a role with a multiplicity greater than one.</p>   |
| Message: Support delay                              | <p>Specifies that messages may have duration (slanted arrow message). If this option is deselected, messages are treated as instantaneous, or fast (horizontal message).</p>   |
| Interface/Class: Auto-implement realized interfaces | <p>Adds to the realizing class any methods of a realized interface and its parents that are not already implemented by the class. The &lt;&lt;implement&gt;&gt; stereotype is applied to the methods.</p>  |
| Interface/Class: Class attribute default visibility | <p>Specifies the default visibility of class attributes.</p>   |

### **i Note**

For information about specifying naming conventions for your model objects, see *Core Features Guide > Modeling with PowerDesigner > Objects > Naming Conventions*.

## 1.1.4.2 Setting OOM Display Preferences

PowerDesigner display preferences allow you to customize the format of object symbols, and the information that is displayed on them. To set object-oriented model display preferences, select [Tools](#) [Display Preferences](#) or right-click the diagram background and select [Display Preferences](#).

In the [Display Preferences](#) dialog, select the type of object in the list in the left pane, and modify its appearance in the right pane.

You can control what properties it will display on the [Content](#) tab, and how it will look on the [Format](#) tab. If the properties that you want to display are not available for selection on the [Content](#) tab, click the [Advanced](#) button and add them using the [Customize Content](#) dialog.

For detailed information about controlling the appearance and content of object symbols, see [Core Features Guide > Modeling with PowerDesigner > Diagrams, Matrices, and Symbols > Display Preferences](#).

## 1.1.4.3 Viewing and Editing the Object Language Definition File

Each OOM is linked to a definition file that extends the standard PowerDesigner metamodel to provide objects, properties, data types, and generation parameters and templates specific to the language being modeled. Definition files and other resource files are XML files located in the `Resource Files` directory inside your installation directory, and can be opened and edited in the PowerDesigner Resource Editor.

### Caution

The resource files provided with PowerDesigner inside the `Program Files` folder cannot be modified directly. To create a copy for editing, use the [New](#) tool on the resource file list, and save it in another location. To include resource files from different locations for use in your models, use the [Path](#) tool on the resource file list.

To open your model's definition file and review its extensions, select [Language](#) [Edit Current Object Language](#).

For detailed information about the format of these files, see [Customizing and Extending PowerDesigner > Object, Process, and XML Language Definition Files](#).

### Note

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

### 1.1.4.3.1 Changing the Object Language

You can change the *object language* being modeled in your OOM at any time.

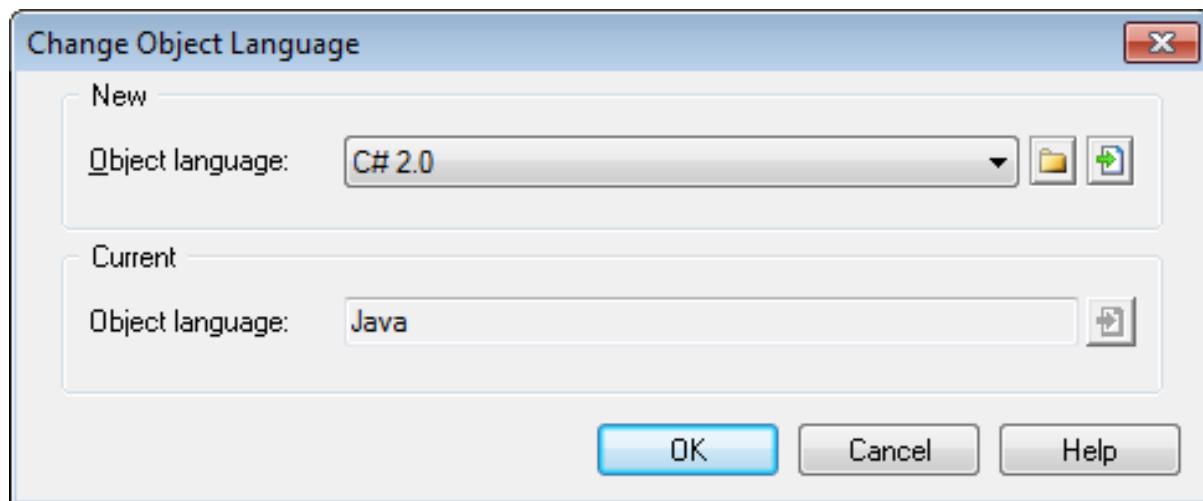
#### Context

##### i Note

You may be required to change the object language if you open a model and the associated definition file is unavailable. Language definition files are frequently updated in each version of PowerDesigner and it is highly recommended to accept this change, or otherwise you may be unable to generate for the selected language.

#### Procedure

1. Select **Language > Change Current Object Language**:



2. Select a *object language* from the list.

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

3. Click **OK**.

A message box opens to tell you that the object language has been changed.

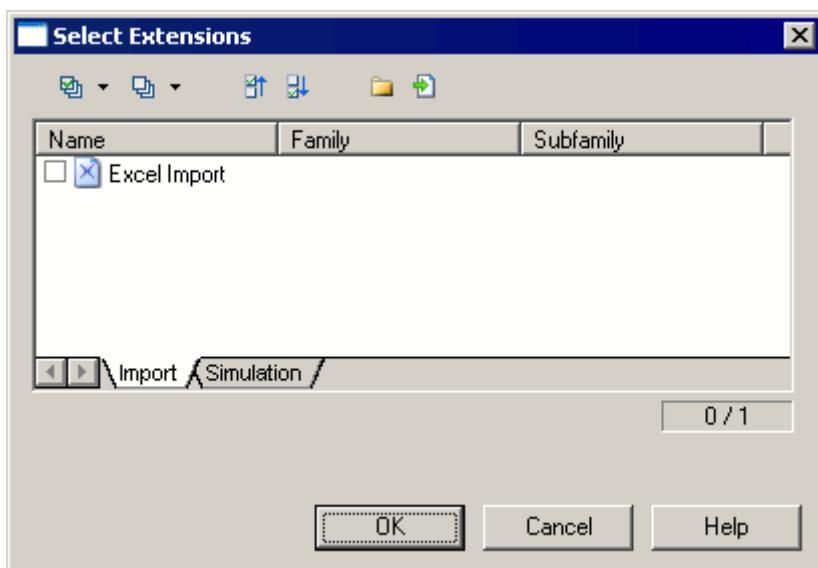
4. Click **OK** to return to the model.

## 1.1.4.4 Extending your Modeling Environment

You can customize and extend PowerDesigner metaclasses, parameters, and file generation with extensions, which can be stored as part of your model or in separate extension files (\*.xem) for reuse with other models.

To access extensions defined in a \*.xem file, simply attach the file to your model. You can do this when creating a new model by clicking the [Select Extensions](#) button at the bottom of the New Model dialog, or at any time by selecting **Model > Extensions** to open the List of Extensions and clicking the [Attach an Extension](#) tool.

In each case, you arrive at the Select Extensions dialog, which lists the extensions available, sorted on sub-tabs appropriate to the type of model you are working with:



To quickly add a property or collection to an object from its property sheet, click the menu button in the bottom-left corner (or press F11) and select [New Attribute](#) or [New List of Associated Objects](#). For more information, see [Core Features Guide > Modeling with PowerDesigner > Objects > Extending Objects](#).

To create a new extension file and define extensions in the Resource Editor, select **Model > Extensions**, click [Add a Row](#), and then click [Properties](#). For detailed information about working with extensions, see [Customizing and Extending PowerDesigner > Extension Files](#).

## 1.1.4.5 Traceability Links

Traceability links provide a flexible means for creating a connection between any object in any type of model and any other object in the same model or any other model open in your workspace. Traceability links have no formal semantic meaning, but can be followed when performing an impact analysis or otherwise navigating through the model structure.

To create a traceability link between objects in the same diagram, select the [Link/Traceability Link](#) tool in the Toolbox. Click inside the symbol of the object that is dependent and, while continuing to hold down the mouse

button, drag the cursor and release it on the symbol of the object on which it depends. In this example, the **Work** entity is shown as being dependent on **School** through a traceability link:



To create a traceability link to any object in any model that is open in the Workspace, open the property sheet of the dependent object, click its *Traceability Links* tab, and click the *Add Objects* tool. Use the *Model* list to select a different model, select the object to point to and click *OK* to create the link and return to the dependent object's *Traceability Links* tab. You can optionally specify a type for any traceability link in the *Link Type* column.

Click the *Types and Grouping* tool to perform various actions on this tab:

- To make a link type available for selection in the *Link Type* column, click the *Types and Grouping* tool and select *New Link Type*. Enter a *Name* for the link type and, optionally, a *Comment* to explain its purpose, and then click *OK*.

**i Note**

Traceability link types created in this way are stored as stereotypes in an extension file embedded in the model. To work directly with this file click the *Types and Grouping* tool and select *Manage Extensions*. For detailed information about working with these files, see *Customizing and Extending PowerDesigner > Extension Files* .

- To control the display and grouping of links, click the *Types and Grouping* tool and select:
  - *No Grouping* - to display all the links in a single list.
  - *Group by Object Type* - to display links to different types of objects on separate sub-tabs. To add a link to a new object type, click the plus sign on the leftmost sub-tab.
  - *Group by Link Type* - to display different link types on separate sub-tabs. To add a new link type, click the plus sign on the leftmost sub-tab.

**i Note**

To see all of the objects that point to an object via traceability links, open its property sheet, click its *Dependencies* tab, and click the *Incoming Traceability Links* sub-tab.

## 1.2 Use Case Diagrams

A use case diagram is a UML diagram that provides a graphical view of the requirements of your system, and helps you identify how users interact with it.

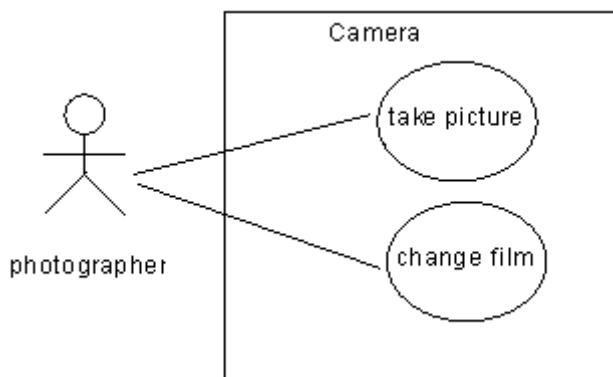
**i Note**

To create a use case diagram in an existing OOM, right-click the model in the Browser and select **► New ► Use Case Diagram**. To create a new model, select **► File ► New Model**, choose Object Oriented Model as the model type and **Use Case Diagram** as the first diagram, and then click **OK**.

With a use case diagram, you immediately see a snapshot of the system functionality. Further details can later be added to the diagram if you need to elucidate interesting points in the system behavior.

A use case diagram is well suited to the task of describing all of the things that can be done with a database system by all the people who might use it. However, it would be poorly suited to describing the TCP/IP network protocol because there are many exception cases, branching behaviors, and conditional functionality (what happens when the connection dies, what happens when a packet is lost?)

In the following example, the actor "photographer" does two things with the camera: take pictures and change the film. When he takes a picture, he has to switch the flash on, open the shutter, and then close the shutter but these activities are not of a high enough level to be represented in a use case.



### 1.2.1 Use Case Diagram Objects

PowerDesigner supports all the objects necessary to build use case diagrams.

| Object      | Tool | Symbol | Description   |
|-------------|------|--------|---|
| Actor       |      |        | Used to represent an external person, process or something interacting with a system, sub-system or class. See <a href="#">Actors (OOM) [page 24]</a> . |
| Use case    |      |        | Defines a piece of coherent behavior in a system, without revealing its internal structure. See <a href="#">Use Cases (OOM) [page 22]</a> .             |
| Association |      |        | Communication path between an actor and a use case that it participates in. See <a href="#">Use Case Associations (OOM) [page 28]</a> .                 |

| Object         | Tool | Symbol | Description   |
|----------------|------|--------|---|
| Generalization |      |        | A link between a general use case and a more specific use case that inherits from it and adds features to it. See <a href="#">Generalizations (OOM) [page 93]</a> . |
| Dependency     |      |        | Relationship between two modeling elements, in which a change to one element will affect the other element. See <a href="#">Dependencies (OOM) [page 96]</a> .      |

## 1.2.2 Use Cases (OOM)

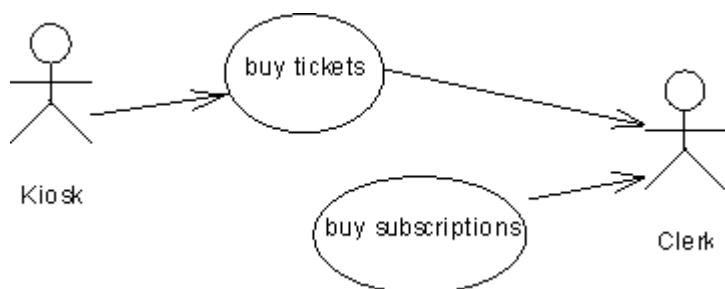
A use case is an interaction between a user and a system (or part of a system). It defines a discrete goal that a user wants to achieve with the system, without revealing the system's internal structure.

A use case can be created in the following diagrams:

- Use Case Diagram

### Example

In this example, "buy tickets" and "buy subscriptions" are use cases.



### 1.2.2.1 Creating a Use Case

You can create a use case from the Toolbox, Browser, or *Model* menu.

- Use the *Use Case* tool in the Toolbox.
- Select **Model > Use Cases** to access the List of Use Cases, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Use Case**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.2.2.2 Use Case Properties

To view or edit a use case's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## Specification Tab

The Specification tab contains the following properties, available on sub-tabs at the bottom of the dialog:

| Property         | Description   |
|------------------|---|
| Action Steps     | Specifies a textual description of the normal sequence of actions associated with a use case.<br><br>For example, the action steps for a use case called 'register patient' in a hospital might be as follows: "Open a file, give a new registration number, write down medical treatment".                         |
| Extension Points | Specifies a textual description of actions that extend the normal sequence of actions. Extensions are usually introduced with an "if ....then" statement.<br><br>For example, an extension to the action steps above might be: "If the patient already has a registration number, then retrieve his personal file". |
| Exceptions       | Specifies signals raised in response to errors during system execution.   |
| Pre-Conditions   | Specifies constraints that must be true for an operation to be invoked.   |
| Post-Conditions  | Specifies constraints that must be true for an operation to exit correctly.   |

## Implementation Classes Tab

A use case is generally a task or service, represented as a verb. When analyzing what a use case must do, you can identify the classes and interfaces that need to be created to fulfill the task, and attach them to the use case. The Implementation Classes tab lists the classes and interfaces used to implement a use case. The following tools are available:

| Tool | Action  |
|------|---|
|      | Add Objects – Opens a dialog box to select any class or interface in the model to implement the use case. |
|      | Create a New Class – Creates a new class to implement the use case.                                       |
|      | Create a New Interface - Creates a new interface to implement the use case.                               |

For example, a use case Ship product by express mail could be implemented by the classes Shipping, Product, and Billing.

## Related Diagrams Tab

The Related Diagrams tab lists diagrams that help you to further understand the use case. Click the [Add Objects](#) tool to add diagrams to the list from any model open in the workspace. For more information, [Core Features Guide > Modeling with PowerDesigner > Diagrams, Matrices, and Symbols > Diagrams > Specifying Diagrams as Related Diagrams](#).

### 1.2.3 Actors (OOM)

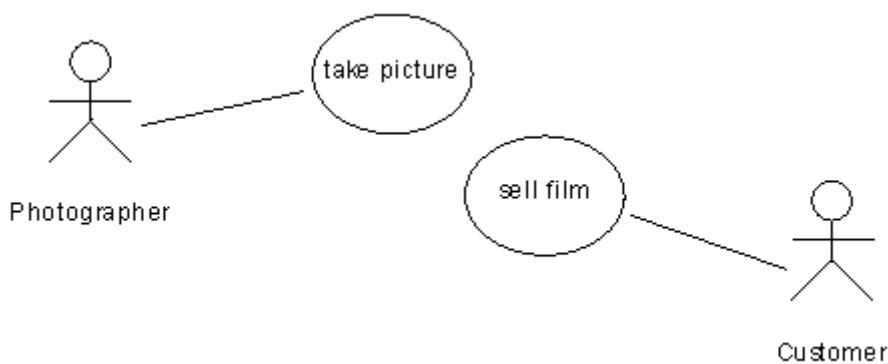
An actor is an outside user or set of users that interact with a system. Actors can be humans or other external systems. For example, actors in a computer network system may include a system administrator, a database administrator and users. Actors are typically those entities whose behavior you cannot control or change, because they are not part of the system that you are describing.

An actor can be created in the following diagrams:

- Communication Diagram
- Sequence Diagram
- Use Case Diagram

A single actor object may be used in a use case, a sequence, and a communication diagram if it plays the same role in each. Each actor object is available to all the diagrams in your OOM. They can either be created in the diagram type you need, or dragged from a diagram type and dropped into another diagram type.

## Actors in a Use Case Diagram



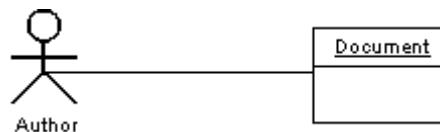
In the use case diagram, an actor is a primary actor for a use case if he asks for and/or triggers the actions performed by a use case. Primary actors are located to the left of the use case, and the association linking them should be drawn from the actor to the use case.

An actor is a secondary actor for a use case if it does not trigger the actions, but rather assists the use case to complete the actions. After performing an action, the use case may give results, documents, or information to the outside and, if so, the secondary actor may receive them. Secondary actors are located to the right of the use case, and the association linking them should be drawn from the use case to the actor.

On a global scale, a secondary actor for one use case may be a primary actor for another use case, either in the same or another diagram.

## Actors in a Communication Diagram

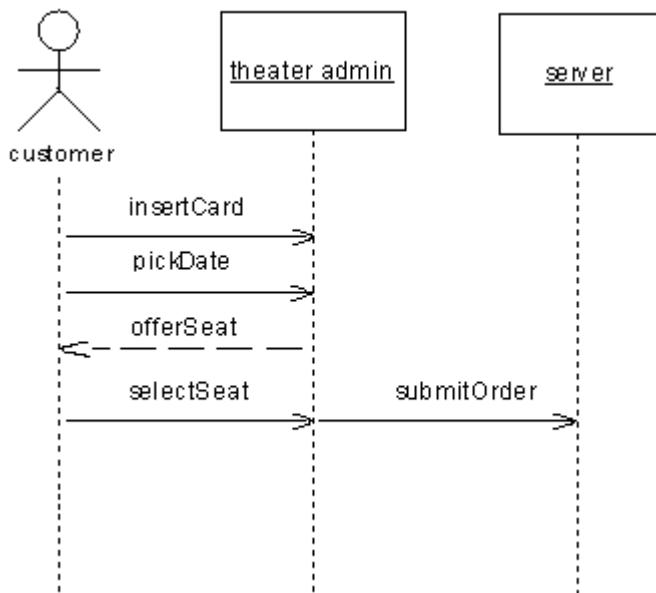
In a communication diagram, an actor may be connected to an object by an instance link, or may send or receive messages.



## Actors in a Sequence Diagram

In the sequence diagram, an actor has a lifeline representing the duration of its life. You cannot separate an actor and its lifeline.

If an actor is the invoker of the interaction, it is usually represented by the first (farthest left) lifeline in the sequence diagram. If you have several actors in the diagram, you should try to position them to the farthest left or to the farthest right lifelines because actors are, by definition, external to the system.



### 1.2.3.1 Creating an Actor

You can create an actor from the Toolbox, Browser, or *Model* menu.

- Use the *Actor* tool in the Toolbox.
- Select **Model > Actors** to access the List of Actors, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Actor**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

### 1.2.3.2 Actor Properties

To view or edit an actor's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <b>Code</b> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## Implementation Classes Tab

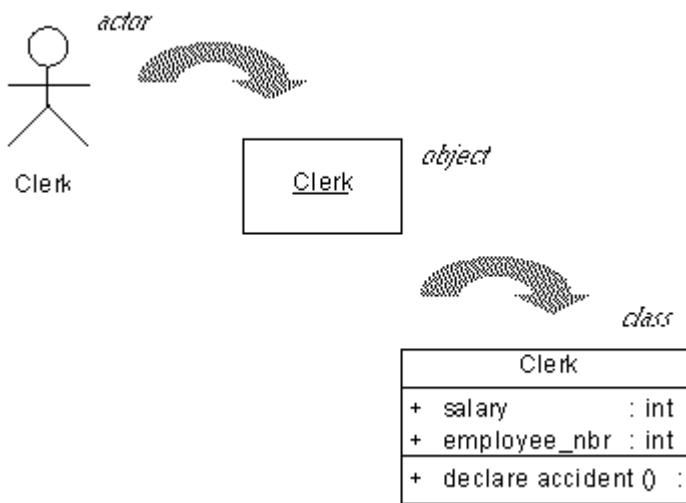
An actor can be a human being (person, partner) or a machine, or process (automated system). When analyzing what an actor must do, you can identify the classes and interfaces that need to be created for the actor to perform his task, and attach them to the actor. The *Implementation Classes* tab lists the classes and interfaces used to implement the actor. The following tools are available:

| Tool | Action  |
|------|---|
|      | <i>Add Objects</i> – Opens a dialog box to select any class or interface in the model to implement the actor. |
|      | <i>Create a New Class</i> – Creates a new class to implement the actor.                                       |
|      | <i>Create a New Interface</i> - Creates a new interface to implement the actor.                               |

For example, an actor **Car** could be implemented by the classes **Engine** and **Motorway**.

Conceptually, you may link elements even deeper. For example, a clerk working in an insurance company is represented as an actor in a use case diagram, dealing with customers who declare a car accident.

The clerk actor becomes an object in a communication or sequence diagram, receiving messages from customers and sending messages to his manager, which is an instance of the Clerk class in a class diagram with its associated attributes and operations:



## Related Diagrams Tab

The Related Diagrams tab lists diagrams that help you to further understand the actor. Click the [Add Objects](#) tool to add diagrams to the list from any model open in the workspace. For more information, see [Core Features Guide > Modeling with PowerDesigner > Diagrams, Matrices, and Symbols > Diagrams > Specifying Diagrams as Related Diagrams](#).

### 1.2.3.3 Reusing Actors

The same actor can be used in a Use Case Diagram, Communication Diagram, and Sequence Diagram. To reuse an actor created in one diagram in another diagram:

- Select the actor you need in the Browser, and drag it and drop it into the new diagram.
- Select [Show Symbols](#) in the new diagram to open the Show Symbols dialog box, select the actor to display, and click *OK*.

### 1.2.4 Use Case Associations (OOM)

An association is a unidirectional relationship that describes a link between objects.

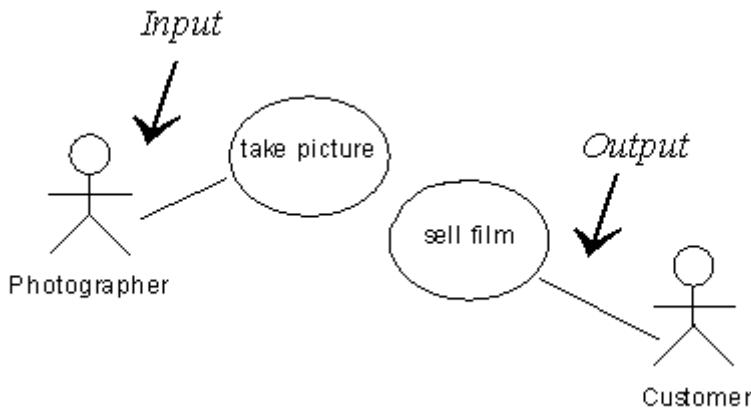
Use case associations can only be created in use case diagrams. You can create them by drawing from:

- An actor to a use case - an input association
- A use case to an actor - an output association

The UML standard does not explicitly display the direction of the association, and instead has the position of actors imply it. When an actor is positioned to the left of the use case, the association is an input, and when he is to

the right, it is an output. To explicitly display the orientation of the association click **Tools > Display Preferences**, select **Use Case Association** in the Category tree, and select the **Orientation** option. For detailed information about working with display preferences, see *Core Features Guide > Modeling with PowerDesigner > Diagrams, Matrices, and Symbols > Display Preferences*.

## Example



### 1.2.4.1 Creating a Use Case Association

You can create use case association from the Toolbox, Browser, or **Model** menu.

- Use the **Use Case Association** tool in the Toolbox.
- Select **Model > Associations** to access the List of Associations, and click the **Add a Row** tool.
- Right-click the model (or a package) in the Browser, and select **New > Association**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.2.4.2 Use Case Association Properties

To view or edit an association's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Orientation       | Defines the direction of the association. You can choose between: <ul style="list-style-type: none"><li>• Primary Actor – the association leads from the actor to the use case</li><li>• Secondary Actor – the association leads from the use case to the actor</li></ul>  |
| Source            | Specifies the object that the association leads from. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.   |
| Destination       | Specifies the object that the association leads to. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.   |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## 1.3 Structural Diagrams

The diagrams in this chapter allow you to model the static structure of your system. PowerDesigner provides three types of diagrams for modeling your system in this way, each of which offers a different view of your objects and their relationships:

- A class diagram shows the static structure of the classes that make up the system. You use a class diagram to identify the kinds of objects that will compose your system, and to define the ways in which they will be associated. For more information, see [Class Diagrams \[page 31\]](#).
- A composite structure diagram allows you to define in greater detail the internal structure of your classes and the ways that they are associated with one another. You use a composite structure diagram in particular to model complex forms of composition that would be very cumbersome to model in a class diagram. For more information, see [Composite Structure Diagrams \[page 33\]](#).

- An object diagram is like a class diagram, except that it shows specific object instances of the classes. You use an object diagram to represent a snapshot of the relationships between actual instances of classes. For more information, see [Object Diagrams \[page 36\]](#).
- A package diagram shows the structure of the packages that make up your application, and the relationships between them. For more information, see [Package Diagrams \[page 35\]](#).

### 1.3.1 Class Diagrams

A class diagram is a UML diagram that provides a graphical view of the classes, interfaces, and packages that compose a system, and the relationships between them.

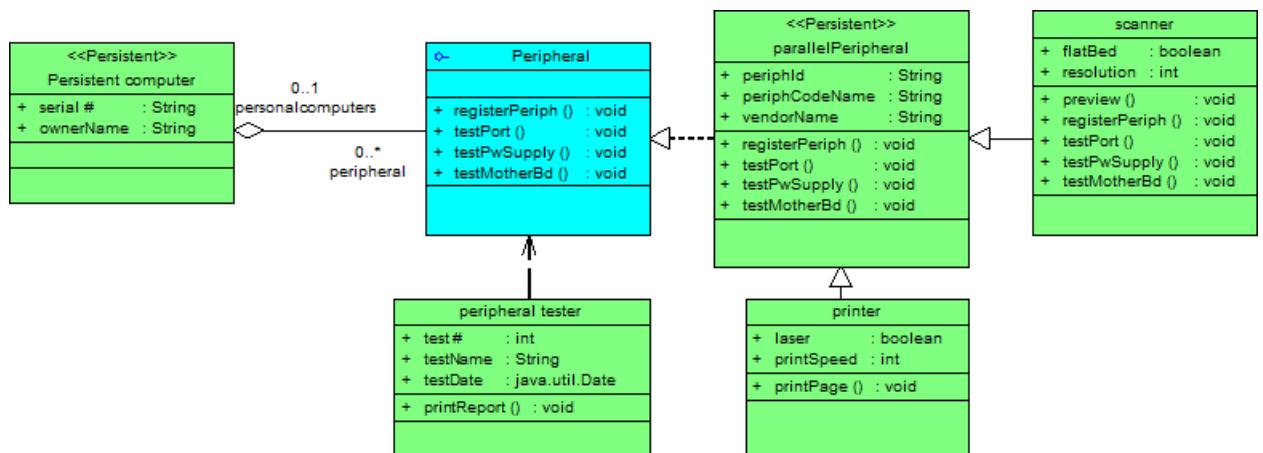
#### i Note

To create a class diagram in an existing OOM, right-click the model in the Browser and select **New > Class Diagram**. To create a new model, select **File > New Model**, choose Object Oriented Model as the model type and *Class Diagram* as the first diagram, and then click **OK**.

You build a class diagram to simplify the interaction of objects in the system you are modeling. Class diagrams express the static structure of a system in terms of classes and relationships between those classes. A class describes a set of objects, and an association describes a set of links; objects are class instances, and links are association instances.

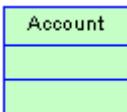
A class diagram does not express anything specific about the links of a given object, but it describes, in an abstract way, the potential link from an object to other objects.

The following example shows an analysis of the structure of peripherals in a class diagram:



### 1.3.1.1 Class Diagram Objects

PowerDesigner supports all the objects necessary to build class diagrams.

| Object         | Tool  | Symbol  | Description   |
|----------------|---|---|---|
| Class          |    |    | Set of objects sharing the same attributes, operations, methods, and relationships. See <a href="#">Classes (OOM) [page 38]</a> .   |
| Interface      |    |    | Descriptor for the externally visible operations of a class, object, or other entity without specification of internal structure. See <a href="#">Interfaces (OOM) [page 60]</a> .  |
| Port           |    |    | Interaction point between a classifier and its environment. See <a href="#">Ports (OOM) [page 44]</a> .   |
| Generalization |  |  | Link between classes showing that the sub-class shares the structure or behavior defined in one or more superclasses. See <a href="#">Generalizations (OOM) [page 93]</a> .   |
| Require Link   |  |  | Connects a class, component, or port to an interface. See <a href="#">Require Links (OOM) [page 100]</a> .  |
| Association    |  |  | Structural relationship between objects of different classes. See <a href="#">Associations (OOM) [page 84]</a> .  |
| Aggregation    |  |  | A form of association that specifies a part-whole relationship between a class and an aggregate class (example: a car has an engine and wheels). See <a href="#">Associations (OOM) [page 84]</a> .                                 |
| Composition    |  |  | A form of aggregation but with strong ownership and coincident lifetime of parts by the whole; the parts live and die with the whole (example: an invoice and its invoice line). See <a href="#">Associations (OOM) [page 84]</a> . |
| Dependency     |  |  | Relationship between two modeling elements, in which a change to one element will affect the other element. See <a href="#">Dependencies (OOM) [page 96]</a> .  |
| Realization    |  |  | Semantic relationship between classifiers, in which one classifier specifies a contract that another classifier guarantees to carry out. See <a href="#">Realizations (OOM) [page 99]</a> .   |

| Object     | Tool | Symbol | Description   |
|------------|------|--------|---|
| Inner link |      |        | Exists when a class is declared within another class or interface. See <a href="#">Creating Composite and Inner Classifiers [page 54]</a> . |
| Attribute  | N/A  | N/A    | Named property of a class. See <a href="#">Associations (OOM) [page 84]</a> .   |
| Operation  | N/A  | N/A    | Service that can be requested from a class. See <a href="#">Operations (OOM) [page 77]</a> .  |

### 1.3.2 Composite Structure Diagrams

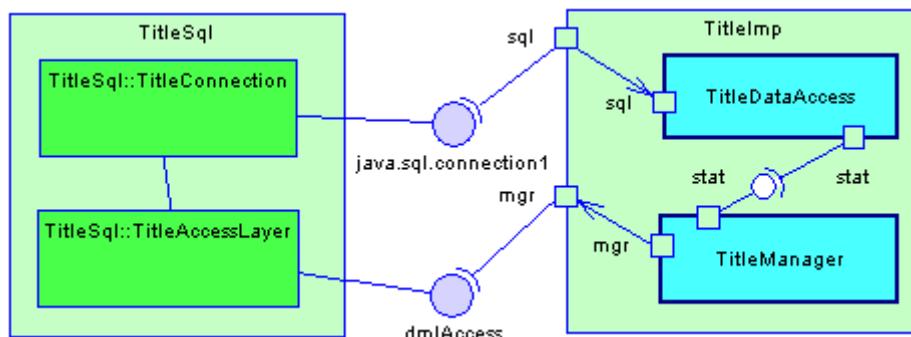
A composite structure diagram is a UML diagram that provides a graphical view of the classes, interfaces, and packages that compose a system, including the ports and parts that describe their internal structures.

#### Note

To create a composite structure diagram in an existing OOM, right-click the model in the Browser and select **New > Composite Structure Diagram**. To create a new model, select **File > New Model**, choose Object Oriented Model as the model type and *Composite Structure Diagram* as the first diagram, and then click **OK**.

A composite structure diagram performs a similar role to a class diagram, but allows you to go into further detail in describing the internal structure of multiple classes and showing the interactions between them. You can graphically represent inner classes and parts and show associations both between and within classes.

In the following example, the internal structures of the classes TitleSql (which contains two inner classes) and TitleImp (which contains two parts) are connected via the interfaces dmlAccess and java.sql.connection1:



## 1.3.2.1 Composite Structure Diagram Objects

PowerDesigner supports all the objects necessary to build composite structure diagrams.

| Object               | Tool | Symbol | Description   |
|----------------------|------|--------|---|
| Class                |      |        | Set of objects sharing the same attributes, operations, methods, and relationships. See <a href="#">Classes (OOM) [page 38]</a> .   |
| Interface            |      |        | Descriptor for the externally visible operations of a class, object, or other entity without specification of internal structure. See <a href="#">Interfaces (OOM) [page 60]</a> .                  |
| Port                 |      |        | Interaction point between a classifier and its environment. See <a href="#">Ports (OOM) [page 44]</a> .   |
| Part                 |      |        | Classifier instance playing a particular role within the context of another classifier. See <a href="#">Parts (OOM) [page 42]</a> .   |
| Generalization       |      |        | Link between classes showing that the sub-class shares the structure or behavior defined in one or more superclasses. See <a href="#">Generalizations (OOM) [page 93]</a> .                         |
| Require Link         |      |        | Connects classifiers to interfaces. See <a href="#">Require Links (OOM) [page 100]</a> .  |
| Assembly Connector   |      |        | Connects parts to each other. See <a href="#">Assembly and Delegation Connectors (OOM) [page 47]</a> .  |
| Delegation Connector |      |        | Connects parts to ports on the outside of classifiers. See <a href="#">Assembly and Delegation Connectors (OOM) [page 47]</a> .   |
| Association          |      |        | Structural relationship between objects of different classes. See <a href="#">Associations (OOM) [page 84]</a> .  |
| Aggregation          |      |        | A form of association that specifies a part-whole relationship between a class and an aggregate class (example: a car has an engine and wheels). See <a href="#">Associations (OOM) [page 84]</a> . |

| Object      | Tool | Symbol | Description   |
|-------------|------|--------|---|
| Composition |      |        | A form of aggregation but with strong ownership and coincident lifetime of parts by the whole; the parts live and die with the whole (example: an invoice and its invoice line). See <a href="#">Associations (OOM) [page 84]</a> . |
| Dependency  |      |        | Relationship between two modeling elements, in which a change to one element will affect the other element. See <a href="#">Dependencies (OOM) [page 96]</a> .  |
| Realization |      |        | Semantic relationship between classifiers, in which one classifier specifies a contract that another classifier guarantees to carry out. See <a href="#">Realizations (OOM) [page 99]</a> .   |
| Attribute   | N/A  | N/A    | Named property of a class. See <a href="#">Associations (OOM) [page 84]</a> .   |
| Operation   | N/A  | N/A    | Service that can be requested from a class. See <a href="#">Operations (OOM) [page 77]</a> .  |

### 1.3.3 Package Diagrams

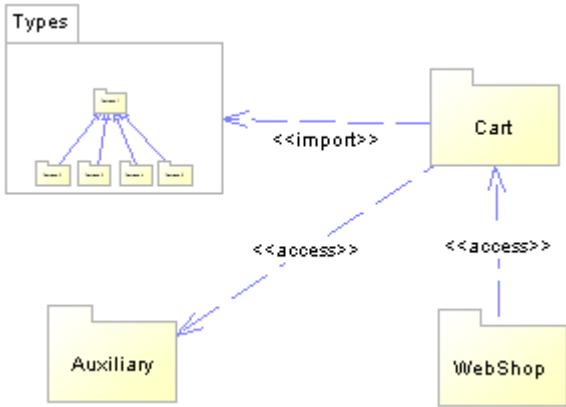
A package diagram is a UML diagram that provides a high-level graphical view of the organization of your application, and helps you identify generalization and dependency links between the packages.

#### i Note

To create a package diagram in an existing OOM, right-click the model in the Browser and select [New > Package Diagram](#). To create a new model, select [File > New Model](#), choose Object Oriented Model as the model type and [Package Diagram](#) as the first diagram, and then click [OK](#).

You can control the level of detail shown for each package, by toggling between the standard and composite package views via the Edit or contextual menus.

In the following example, the WebShop package imports the Cart package, which, in turn, imports the Types package, and has access to the Auxiliary package. The Types package is shown in composite (sub-diagram) view:



### 1.3.3.1 Package Diagram Objects

PowerDesigner supports all the objects necessary to build package diagrams.

| Object         | Tool | Symbol | Description  |
|----------------|------|--------|--|
| Package        |      |        | A container for organizing your model objects. See <a href="#">Packages (OOM) [page 57]</a> .  |
| Generalization |      |        | Link between packages showing that the sub-package shares the structure or behavior defined in one or more super-packages. See <a href="#">Generalizations (OOM) [page 93]</a> . |
| Dependency     |      |        | Relationship between two packages, in which a change to one package will affect the other. See <a href="#">Dependencies (OOM) [page 96]</a> .                                    |

### 1.3.4 Object Diagrams

An object diagram is a UML diagram that provides a graphical view of the structure of a system through concrete instances of classes (objects), associations (instance links), and dependencies.

#### i Note

To create an object diagram in an existing OOM, right-click the model in the Browser and select **New > Object Diagram**. To create a new model, select **File > New Model**, choose Object Oriented Model as the model type and *Object Diagram* as the first diagram, and then click **OK**.

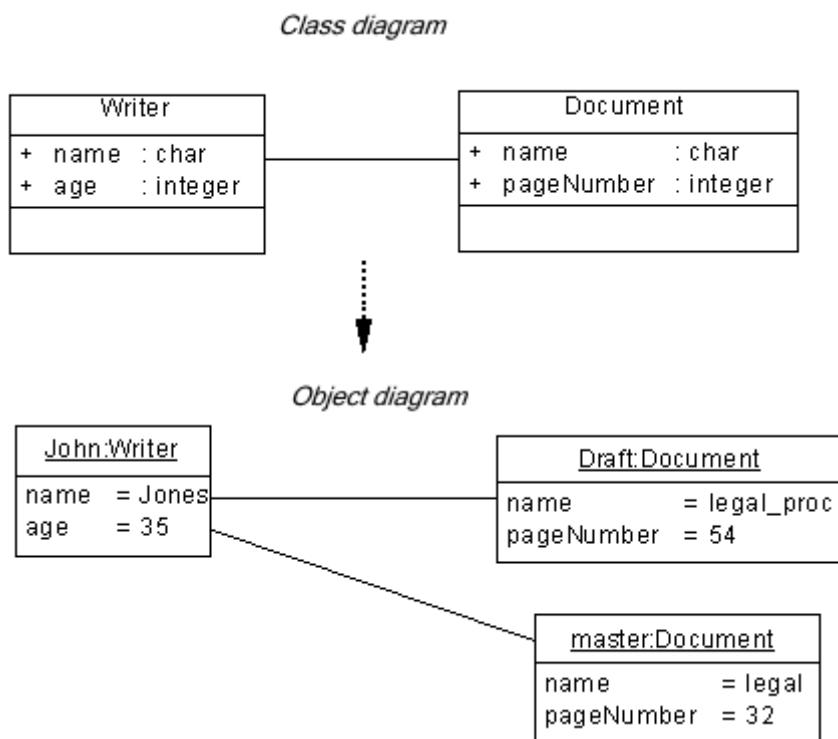
As a diagram of instances, the object diagram shows an example of data structures with data values that corresponds to a detailed situation of the system at a particular point in time.

The object diagram can be used for analysis purposes: constraints between classes that are not classically represented in a class diagram can typically be represented in an object diagram.

If you are a novice in object modeling, instances usually have more meaning than classifiers do, because classifiers represent a level of abstraction. Gathering several instances under the same classifier helps you to understand what classifiers are. Moreover, even for analysts used to abstraction, the object diagram can help understand some structural constraints that cannot be easily graphically specified in a class diagram.

In this respect, the object diagram is a limited use of a class diagram. In the following example, the class diagram specifies that a class Writer is linked to a class Document.

The object diagram, deduced from this class diagram, highlights some of the following details: the object named John, instance of the class Writer is linked to two different objects Draft and Master that are both instances of the class Document.



### i Note

You can drag classes and associations from the Browser and drop them into an object diagram. If you drag classes, new objects as instances of classes are created. If you drag an association, a new instance link as instance of the association, and two objects are created.

### 1.3.4.1 Object Diagram Objects

PowerDesigner supports all the objects necessary to build object diagrams.

| Object           | Tool | Symbol | Description  |
|------------------|------|--------|--|
| Object           |      |        | Instance of a class. See <a href="#">Objects (OOM) [page 62]</a> .   |
| Attribute values | N/A  | N/A    | An attribute value represents an instance of a class attribute, this attribute being in the class related to the object. See <a href="#">Object Properties [page 65]</a> . |
| Instance link    |      |        | Communication link between two objects. See <a href="#">Instance Links (OOM) [page 105]</a> .  |
| Dependency       |      |        | Relationship between two modeling elements, in which a change to one element will affect the other element. See <a href="#">Dependencies (OOM) [page 96]</a> .             |

### 1.3.5 Classes (OOM)

A class is a description of a set of objects that have a similar structure and behavior, and share the same attributes, operations, relationships, and semantics.

A class can be created in the following diagrams:

- Class Diagram
- Composite Structure Diagram

The structure of a class is described by its attributes and associations, and its behavior is described by its operations.

Classes, and the relationships that you create between them, form the basic structure of an OOM. A class defines a concept within the application being modeled, such as:

- a physical thing (like a car),
- a business thing (like an order)
- a logical thing (like a broadcasting schedule),
- an application thing (like an OK button),
- a behavioral thing (like a task)

The following example shows the class Aircraft with its attributes (range and length) and operation (startengines).

| aircraft                |
|-------------------------|
| + range : int           |
| + length : int          |
| + startengines() : void |

### 1.3.5.1 Creating a Class

You can create a class from an interface, or from the Toolbox, Browser, or *Model* menu.

- Use the *Class* tool in the Toolbox.
- Select **Model > Classes** to access the List of Classes, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Class**.
- Right-click an interface, and select *Create Class* from the contextual menu (this method allows you to inherit all the operations of the interface, including the getter and setter operations, creates a realization link between the class and the interface, and shows this link in the *Realizes* sub-tab of the *Dependencies* tab of the class property sheet).

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

### 1.3.5.2 Class Properties

To view or edit a class's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property              | Description  |
|-----------------------|--|
| Name/Code/<br>Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Extends               | Specifies the parent class (to which the present class is linked by a generalization). Click the <i>Select Classifier</i> tool to the right to specify a parent class and click the <i>Properties</i> tool to access its property sheet.   |

| Property    | Description  |
|-------------|--|
| Stereotype  | <p>Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.</p> <p>The following common stereotypes are available by default:</p> <ul style="list-style-type: none"> <li>• <b>&lt;&lt;actor&gt;&gt;</b> - Coherent set of roles that users play</li> <li>• <b>&lt;&lt;enumeration&gt;&gt;</b> - List of named values used as the range of an attribute type</li> <li>• <b>&lt;&lt;exception&gt;&gt;</b> - Exception class, mainly used in relation to error messages</li> <li>• <b>&lt;&lt;implementationClass&gt;&gt;</b> - Class whose instances are statically typed. Defines the physical data structure and methods of a class as implemented in traditional programming languages</li> <li>• <b>&lt;&lt;process&gt;&gt;</b> - Heavyweight flow that executes concurrently with other processes</li> <li>• <b>&lt;&lt;signal&gt;&gt;</b> - Specification of asynchronous stimulus between instances</li> <li>• <b>&lt;&lt;metaclass&gt;&gt;</b> - a metaclass of some other class</li> <li>• <b>&lt;&lt;powertype&gt;&gt;</b> - a metaclass whose instances are sub-classes of another class</li> <li>• <b>&lt;&lt;thread&gt;&gt;</b> - Lightweight flow that executes concurrently with other threads within the same process. Usually executes inside the address space of an enclosing process</li> <li>• <b>&lt;&lt;type&gt;&gt;</b> - Abstract class used to specify the structure and behavior of a set of objects but not the implementation</li> <li>• <b>&lt;&lt;utility&gt;&gt;</b> - Class that has no instances</li> </ul> <p>Other language-specific stereotypes may be available if they are specified in the object language file (see <i>Customizing and Extending PowerDesigner &gt; Extension Files &gt; Stereotypes (Profile)</i> ).</p> |
| Visibility  | <p>Specifies the visibility of the object, how it is seen outside its enclosing namespace. When a class is visible to another object, it may influence the structure or behavior of the object, and/or be affected by it. You can choose between:</p> <ul style="list-style-type: none"> <li>• <b>Private</b> – only to the object itself</li> <li>• <b>Protected</b> – only to the object and its inherited objects</li> <li>• <b>Package</b> – to all objects contained within the same package</li> <li>• <b>Public</b> – to all objects (option by default)</li> </ul>   |
| Cardinality | <p>Specifies the number of instances a class can have. You can choose between:</p> <ul style="list-style-type: none"> <li>• <b>0..1</b> – None to one</li> <li>• <b>0..*</b> – None to an unlimited number</li> <li>• <b>1..1</b> – One to one</li> <li>• <b>1..*</b> – One to an unlimited number</li> <li>• <b>*</b> – Unlimited number</li> </ul>   |

| Property      | Description   |
|---------------|---|
| Type          | <p>Allows you to specify that a class is a generic type, or that it is bound to one. You can choose between:</p> <ul style="list-style-type: none"> <li>• <b>Class</b></li> <li>• <b>Generic</b></li> <li>• <b>Bound</b> – an additional list is displayed, which lets you specify the generic type to which the class is bound. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected type.</li> </ul> <p>If you specify <b>either</b> or <b>Bound</b>, then the <b>Generic</b> tab is displayed, allowing you to control the associated type variables. For more information on generic types and binding classes to them, see <a href="#">Creating Generic Types [page 50]</a>.</p> |
| Abstract      | Specifies that the class cannot be instantiated and therefore has no direct instances.  |
| Final         | Specifies that the class cannot have any inherited objects.   |
| Generate code | Specifies that the class is included when you generate code from the model, it does not affect inter-model generation.  |
| Keywords      | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.   |

## Detail Tab

The **Detail** tab contains a **Persistent** groupbox whose purpose is to define the persistent generated code of a class during OOM to CDM or PDM generation, and which contains the following properties:

| Property   | Description  |
|------------|--|
| Persistent | <p>Specifies that the class must be persisted in a generated CDM or PDM. You have to select one of the following options:</p> <ul style="list-style-type: none"> <li>• Generate table - the class is generated as an entity or table.</li> <li>• Migrate columns – [PDM only] the class is not generated, and its attributes and associations are migrated to the generated parent or child table.</li> <li>• Generate ADT – [PDM only] the class is generated as an abstract data type (see <i>Data Architecture &gt; Building Data Models &gt; Physical Diagrams &gt; Abstract Data Types</i> ).</li> <li>• Value Type – the class is not generated, and its attributes are generated in their referencing types.</li> </ul> <p>For more information, see <a href="#">Managing Object Persistence During Generation of Data Models [page 245]</a>.</p> |
| Code       | <p>Specifies the code of the table or entity that will be generated from the current class in a CDM or PDM model. Persistent codes are used for round-trip engineering: the same class always generates the same entity or table with a code compliant with the target DBMS.</p> <p>Example: to generate a class Purchaser into a table PURCH, type PURCH in the <b>Code</b> box.</p>  |

| Property          | Description   |
|-------------------|---|
| Inner to          | Specifies the name of the class or interface to which the current class belongs as an inner classifier  |
| Association class | Specifies the name of the association related to the class to form an association class. The attributes and operations of the current class are used to complement the definition of the association. |

The following tabs are also available:

- *Attributes* - lists and lets you add or create attributes (including accessors) associated with the class (see [Attributes \(OOM\) \[page 67\]](#)). Click the Inherited button to review the public and protected attributes inherited from a parent class.
- *Identifiers* - lists and lets you create identifiers associated with the class (see [Identifiers \(OOM\) \[page 75\]](#)).
- *Operations* - lists and lets you add or create operations associated with the class (see [Operations \(OOM\) \[page 77\]](#)).
- *Generic* - lets you specify the type parameters of a generic class or values for the required type parameters for a class that is bound to a generic type (see [Creating Generic Types \[page 50\]](#))
- *Ports* - lists and lets you create ports associated with the class (see [Ports \(OOM\) \[page 44\]](#)).
- *Parts* - lists and lets you create parts associated with the class (see [Parts \(OOM\) \[page 42\]](#)).
- *Associations* - lists and lets you create associations associated with the class (see [Associations \(OOM\) \[page 84\]](#)).
- *Inner Classifiers* - lists and lets you create inner classes and interfaces associated with the class (see [Creating Composite and Inner Classifiers \[page 54\]](#)).
- *Related Diagrams* - lists and lets you add model diagrams that are related to the class (see [Core Features Guide > Modeling with PowerDesigner > Diagrams, Matrices, and Symbols > Diagrams > Specifying Diagrams as Related Diagrams](#)).
- *Script* - lets you customize the class creation script (see [Customizing Object Creation Scripts \[page 14\]](#))
- *Preview* - lets you view the code to be generated for the class (see [Previewing Object Code \[page 11\]](#))

#### i Note

If the class is a Web service implementation class, see also [Web Service Implementation Class Properties \[page 217\]](#).

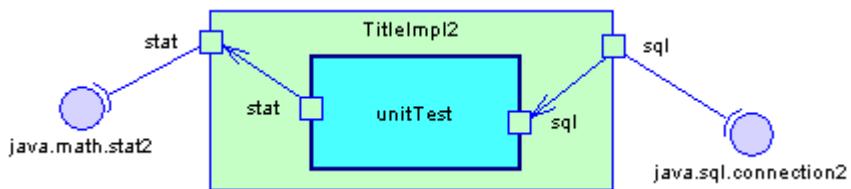
### 1.3.5.3 Parts (OOM)

A part allows you to define a discrete area inside a class or a component. Parts can be connected to other parts or to ports, either directly or via a port on the outside of the part.

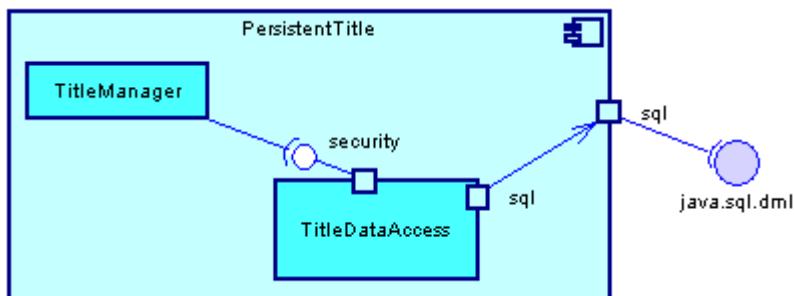
You connect a part to another part by way of an assembly connector. You connect a part to a port on the outside of a class or component by way of a delegation connector.

A part can be created in the following diagrams:

- Composite Structure Diagram (inside a class) - In this example, the class TitleImpl2 contains a part called `unitTest`:



- Component Diagram (inside a component) - In this example, the component PersistentTitle contains two parts, TitleManager and TitleDataAccess:



You can only create a part within a class or a component. If you attempt to drag a part outside of its enclosing classifier, the classifier will grow to continue to enclose it.

## Creating a Part

You can create a part from the Toolbox or from the *Parts* tab of a class or component property sheet:

- Use the *Part* tool in the Toolbox.
- Open the *Parts* tab in the property sheet of a class or component, and click the *Add a Row* tool.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## Part Properties

To view or edit a part's properties, double-click its diagram symbol or Browser or list entry. The *General* tab contains the following properties:

| Property | Description                  |
|----------|------------------------------|
| Parent   | Specifies the parent object. |

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Visibility        | Specifies the visibility of the object, how it is seen outside its enclosing namespace. You can choose between: <ul style="list-style-type: none"> <li>• Private – only to the object itself</li> <li>• Protected – only to the object and its inherited objects</li> <li>• Package – to all objects contained within the same package</li> <li>• Public – to all objects (option by default)</li> </ul>   |
| Data type         | Specifies a classifier as a data type.   |
| Multiplicity      | Specifies the number of instances of the part. If the multiplicity is a range of values, it means that the number of parts can vary at run time.<br><br>You can choose between: <ul style="list-style-type: none"> <li>• * – none to unlimited</li> <li>• 0..* – zero to unlimited</li> <li>• 0..1 – zero or one</li> <li>• 1..* – one to unlimited</li> <li>• 1..1 – exactly one</li> </ul>   |
| Composition       | Specifies the nature of the association with the parent object. If this option is selected, it is a composition and if not, an aggregation.  |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

The following tabs are also available:

- *Ports* - lists the ports associated with the part (see [Ports \(OOM\) \[page 44\]](#)). You can create ports directly in this tab.

### 1.3.5.4 Ports (OOM)

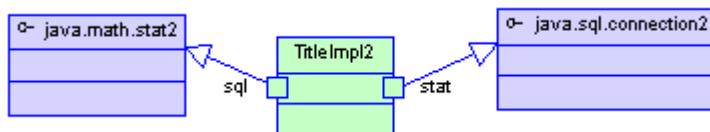
A port is created on the outside of a classifier and specifies a distinct interaction point between the classifier and its environment or between the (behavior of the) classifier and its internal parts.

Ports can be connected to:

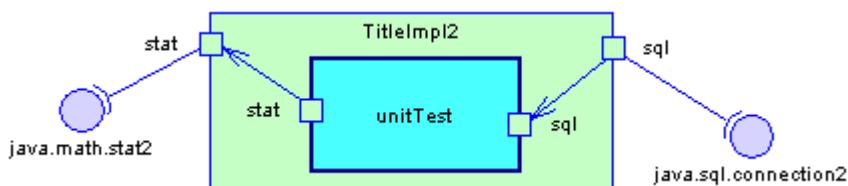
- a part via a delegation connector, through which requests can be made to invoke the behavioral features of a classifier
- an interface via a require link, through which the port may specify the services a classifier provides (offers) to its environment as well as the services that a classifier expects (requires) of its environment.

A port can be created in the following diagrams:

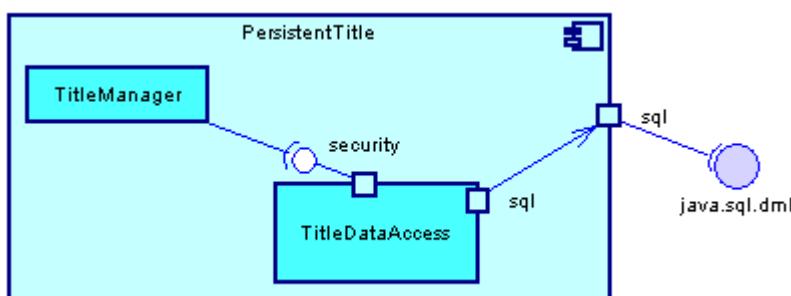
- Class Diagram (on a class) - In this example, the class TitleImpl2 contains the ports sql and stat, which are connected by require links to the interfaces java.math.stat2 and java.sql.connection2:



- Composite Structure Diagram (on a class, a part, or an interface) - In this example, the internal structure of the class TitleImpl2 is shown in more detail, and demonstrates how ports can be used to specify interaction points between a part and its enclosing classifier:



- Component Diagram (on a component or a part) - In this example, ports are used to connect parts with an enclosing component:



## Creating a Port

You can create a port from the Toolbox or from the *Ports* tab of a class, part or component property sheet:

- Use the *Port* tool in the Toolbox.
- Open the *Ports* tab in the property sheet of a class, part, or component, and click the *Add a Row* tool.

### Note

A classifier that is connected to a parent by way of a generalization can redefine the ports of the parent. Click the *Redefine* button at the bottom of the tab to open the *Parent Ports* dialog, select a port and click *Redefine* and then *Close* to redefine the port and add it to the child's list of ports.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## Port Properties

To view or edit a port's properties, double-click its diagram symbol or Browser or list entry. The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Parent            | Specifies the parent classifier.   |
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Visibility        | Specifies the visibility of the object, how it is seen outside its enclosing namespace. You can choose between: <ul style="list-style-type: none"><li>• Private – only to the object itself</li><li>• Protected – only to the object and its inherited objects</li><li>• Package – to all objects contained within the same package</li><li>• Public – (default) to all objects</li></ul>  |
| Data type         | Specifies a classifier as a data type.   |
| Multiplicity      | Specifies the number of instances of the port. If the multiplicity is a range of values, it means that the number of ports can vary at run time.<br>You can choose between: <ul style="list-style-type: none"><li>• * – none to unlimited</li><li>• 0..* – zero to unlimited</li><li>• 0..1 – zero or one</li><li>• 1..* – one to unlimited</li><li>• 1..1 – exactly one</li></ul>   |

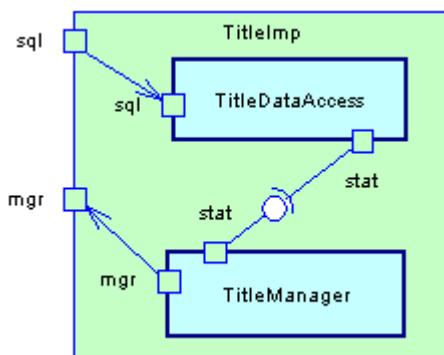
| Property    | Description  |
|-------------|--|
| Redefines   | A port may be redefined when its containing classifier is specialized. The redefining port may have additional interfaces to those that are associated with the redefined port or it may replace an interface by one of its subtypes.  |
| Is Service  | Specifies that this port is used to provide the published functionality of a classifier (default). If this property is cleared, the port is used to implement the classifier but is not part of the essential externally-visible functionality of the classifier. It can, therefore, be altered or deleted along with the internal implementation of the classifier and other properties that are considered part of its implementation. |
| Is Behavior | Specifies that the port is a "behavior port", and that requests arriving at this port are sent to the classifier behavior of the classifier. Any invocation of a behavioral feature targeted at a behavior port will be handled by the instance of the owning classifier itself, rather than by any instances that this classifier may contain.  |
| Keywords    | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

### 1.3.5.5 Assembly and Delegation Connectors (OOM)

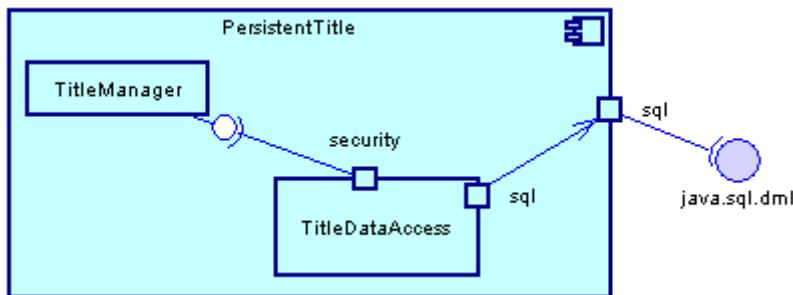
Assembly connectors represent the paths of communication by which parts in your classifiers request and provide services to each other. Delegation show how parts inside a classifier connect to ports on the outside of that classifier and request and provide services to each other.

Assembly and delegation connectors can be created in the following diagrams:

- Composite Structure Diagram - In this example, an assembly connector connects the supplier part `TitleDataAccess` to the client part `TitleManager` and a delegation connector connects the supplier port `sql` on the outside of the class `TitleImp` to the client part `TitleDataAccess` via a port `sql`. A second delegation connector connects the supplier part `TitleManager` via the port `mgr` to the port `mgr` on the outside of the class `TitleImp`:



- Component Diagram - In this example, an assembly connector connects the supplier part `TitleDataAccess` to the client part `TitleManager` and a delegation connector connects the supplier part `TitleDataAccess` via the port `sql` to the client port `sql` on the outside of the component `PersistentTitle`:



## Creating an Assembly or Delegation Connector

You can create assembly and delegation connectors using the *Require Link/Connector* tool in the Toolbox.

## Assembly and Delegation Connector Properties

To view or edit a delegation connector's properties, double-click its diagram symbol or Browser or list entry. The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Supplier / Client | Specify the part or port providing the service and the part or port requesting the service. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.   |
| Interface         | [assembly connectors only] Specifies the interface that the supplier part uses to provide the service. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.  |

| Property | Description   |
|----------|---|
| Keywords | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas. |

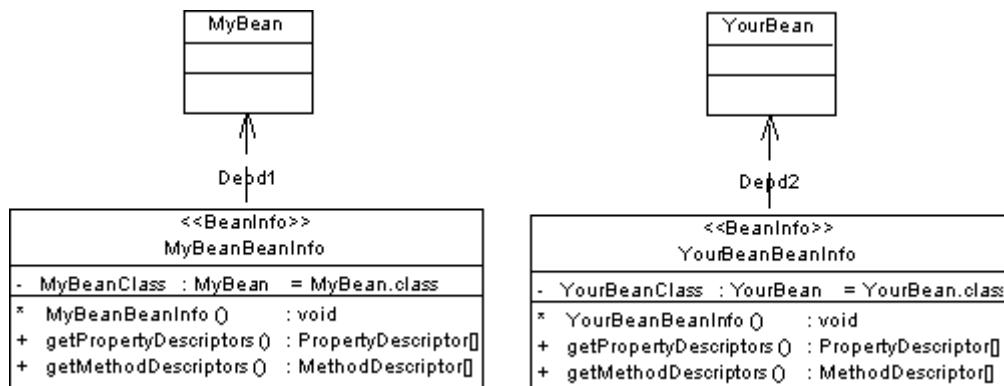
### 1.3.5.6 Creating Java BeanInfo Classes

A JavaBean is a reusable software component written in Java that can be manipulated visually in a builder tool. Bean implementors may want to provide explicit information about the methods, properties, and events of a Bean by providing a Java BeanInfo class, which is used as a standard view of a Bean. You can create Java BeanInfo classes from any class with a type of "JavaBean".

To create a BeanInfo class in a Java OOM:

- Right-click a class bearing the JavaBean stereotype, and select *Create BeanInfo Class*, or
- Select **Language > Create BeanInfo Classes** to open a list of all the JavaBean classes in the model, select one or more classes, and click *OK*.

A BeanInfo class is created in the model for each selected class connected via a dependency link to the JavaBean class:



The BeanInfo class is generated with the following content:

- an attribute:

```
private static final Class <ClassCode>Class = <ClassCode>.class;
```

- a constructor:

```
<ClassCode>BeanInfo()
{
    super();
}
```

- getPropertiesDescriptors();

```
public PropertyDescriptor[] getPropertiesDescriptors ()
{
    // Declare the property array
```

```

PropertyDescriptor properties[] = null;
// Set properties
try
{
    // Create the array
    properties = new PropertyDescriptor[<nbProperties>];
    // Set property 1
    properties[0] = new PropertyDescriptor("<propertyCode1>" ,<ClassCode>Class;
    properties[0].setConstrained(false);
    properties[0].setDisplayName("propertyName1");
    properties[0].setShortDescription("propertyComment1");
    // Set property 2
    properties[1] = new PropertyDescriptor("<propertyCode2>" ,<ClassCode>Class;
    properties[1].setConstrained(false);
    properties[1].setDisplayName("propertyName2");
    properties[1].setShortDescription("propertyComment2");
}
catch
{
    // Handle errors
}
return properties;
}

```

- `getMethodDescriptors()`:

```

public MethodDescriptor[] getMethodDescriptors ()
{
    // Declare the method array
    MethodDescriptor methods[] = null;
    ParameterDescriptor parameters[] = null;

    // Set methods
    try
    {
        // Create the array
        methods = new MethodDescriptor[<nbMethods>];
        // Set method 1
        parameters = new ParameterDescriptor[<nbParameters1>];
        parameters[0] = new ParameterDescriptor();
        parameters[0].setName("parameterCode1");
        parameters[0].setDisplayName("parameterName1");
        parameters[0].setShortDescription("parameterComment1");
        methods[0] = new MethodDescriptor("<methodCode1>" , parameters);
        methods[0].setDisplayName("methodName1");
        methods[0].setShortDescription("methodComment1");
        // Set method 2
        methods[1] = new MethodDescriptor("<methodCode2>" );
        methods[1].setDisplayName("methodName2");
        methods[1].setShortDescription("methodComment2");
    }
    catch
    {
        // Handle errors
    }
    return methods;
}

```

You can view the complete code by clicking the [Preview](#) tab in the BeanInfo class property sheet.

### 1.3.5.7 Creating Generic Types

Java 5.0 and higher supports generic types, which are classifiers that have one or more type variables and one or more methods that use a type variable as a placeholder for an argument or return type. Generic types allow you to

take advantage of stronger compile-time type checking, as when a generic type is used, an actual type is specified for each type variable and this information is used to automatically cast the associated return values.

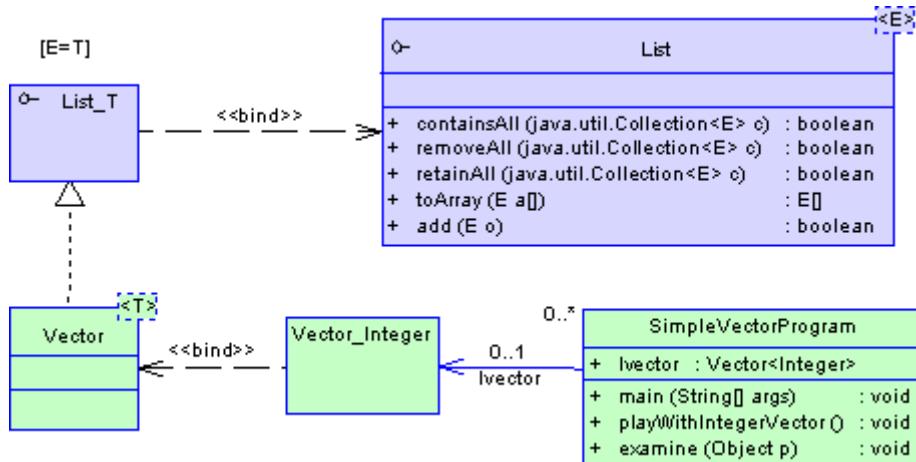
## Context

You must define a list of type variables to be used as datatypes for attributes, method parameters, or return types, along with a bound class to create a generalization, realization, or association. You then bind a classifier to the generic type via this intermediate bound class, and specify the actual types to be used in place of the required type variables.

In this example, the bound interface, **List\_T**, specifies a type 'T' for the type parameter **<E>** of **List**. The generic class **Vector<T>** realizes the generic interface **List<E>** (via the bound interface **List\_T**) with a type **<T>** (that is defined in its own generic definition):

```
public class vector <T> implements List <E>
```

The bound class **Vector\_Integer** specifies a type 'Integer' for the type parameter **<T>** of **Vector<T>**. The **SimpleVectorProgram** class is associated to **Vector\_Integer**, allowing it to use the attribute data type of the **Vector** class set to Integer.



You must create a bound class for a generalization or a realization. However, we could have specified a parameter value for the generic type **<T>** directly (without creating a bound class) as an attribute data type, parameter data type, or return data type, by simply typing the following expression in the type field of **SimpleVectorProgram**:

```
Vector<integer>
```

## Procedure

1. Open the property sheet of the classifier, and select **Generic** from the **General** tab **Type** list. The **Generic** tab is displayed, and a type variable is created in the list on the tab.

2. Click the *Generic* tab, and use the *Add a Row* tool to add any additional type variables. You can also specify a derivation constraint in the form of a list of types.
3. Click *OK* to return to the diagram. The classifier symbol now displays the type variables on its top-left corner.



In order for the classifier to become a true generic type, it must contain at least one generic method.

### 1.3.5.7.1 Creating Generic Methods

PowerDesigner allows you to designate operations as generic methods. Generic methods are methods that have their own list of type variables.

#### Procedure

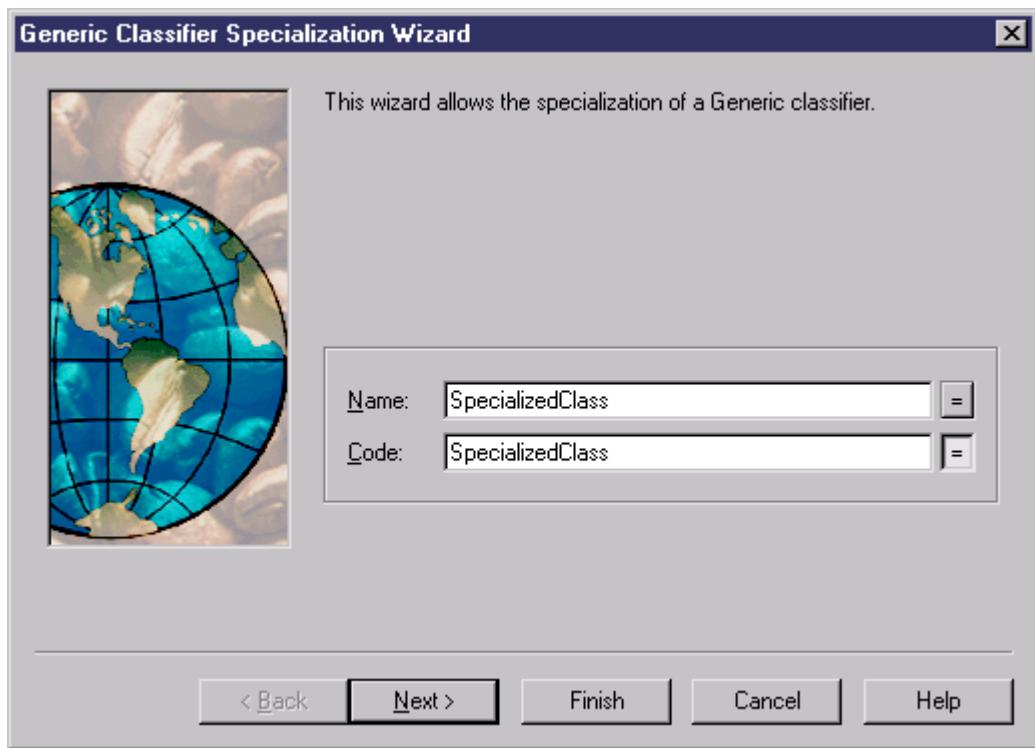
1. Open the property sheet of the class or interface and click on its *Operations* tab.
2. Click the *Add a Row* tool to create a new operation, and then click the *Properties* tool to open its property sheet.
3. Click *Yes* to confirm the creation of the operation, and then select the *Generic* checkbox on the *General* tab of the new operation property sheet to designate the operation as a generic method. The *Generic* tab will be automatically displayed, and a type variable created in the list in the tab.
4. Add any additional type variables that you require with the *Add a Row* tool, and then click *OK*.

### 1.3.5.7.2 Creating a Specialized Classifier

If you need to create a classifier that will inherit from a generic type, you must create an intermediary bound classifier. The Generic Classifier Specialization Wizard can perform these steps for you.

#### Procedure

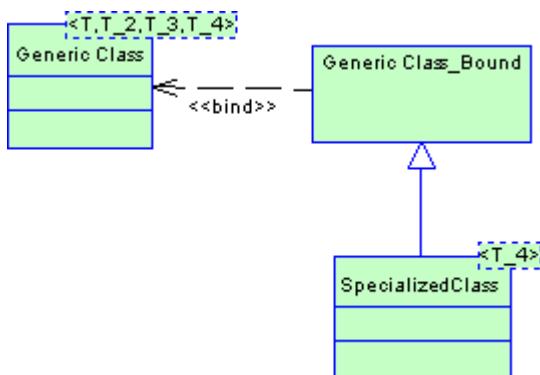
1. Right-click a generic class or interface, and select Create Specialized Class (or Interface) from the contextual menu to open the Generic Classifier Specialization Wizard:



2. Enter a Name and Code for the specialized classifier, and then click Next to go to the type parameters page.
3. Specify values for each of the type parameters in the list. If you do not specify a value for a type parameter, it will be added as a type parameter to the new specialized classifier.
4. Click Finish to return to the diagram. The wizard will have created the specialized classifier and also a bound classifier which acts as an intermediary between the generic and the specialized classifiers, in order to specify values for the type parameters.

The bound classifier is attached to the generic classifier via a dependency with a stereotype of `<<bind>>`, and acts as the parent of the specialized classifier, which is connected to it by a generalization.

In the example below, `SpecializedClass` inherits from `GenericClass` via `GenericClass_Bound`, which specifies type parameters for the generic types `T`, `T_2`, and `T_3`.



At compile time, the specialized classifier can inherit the methods and properties of the generic classifier, and the generic type variables will be replaced by actual types. As a result, the compiler will be able to provide stronger type checking and automatic casting of the associated return values.

### 1.3.5.7.3 Creating a Bound Classifier

You may need to bind a classifier to a generic classifier without creating a specialized classifier. The Bound Classifier Wizard can do this for you.

#### Procedure

1. Right-click a generic class or interface, and select Create Bound Class (or Interface) from the contextual menu to launch the Bound Classifier Wizard.
2. The wizard will create the bound classifier, which is attached to the generic classifier via a dependency with a stereotype of <>bind>>.

### 1.3.5.8 Creating Composite and Inner Classifiers

A composite classifier is a class or an interface that contains other classes or interfaces (called inner classifiers). PowerDesigner supports the creation of composite and inner classifiers, and can display them attached to the composite classifier or in a composite classifier diagram contained in the classifier.

You can create inner classifiers in a class or interface:

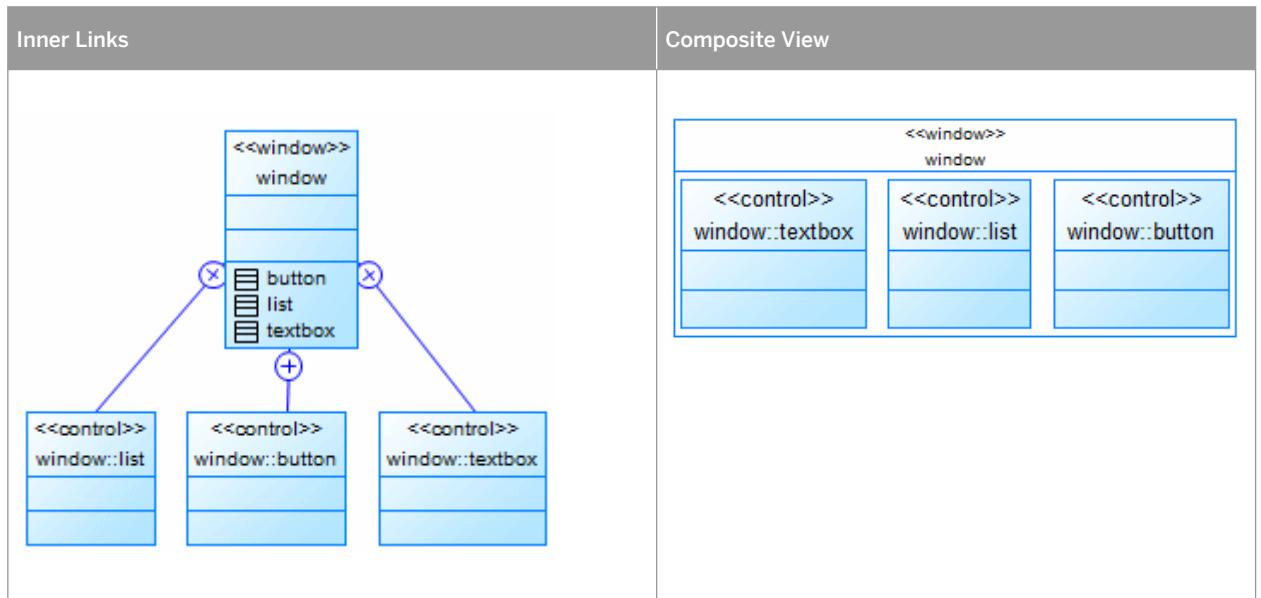
- Open the *Inner Classifiers* tab in the property sheet of a class or interface, and click the *Add Inner Class* or *Add Inner Interface* tool.
- Right-click a class or interface in the Browser, and select or .
- Select the *Inner Link* tool in the Toolbox and click and use it to connect two classes in the diagram. The second class is transformed into an inner classifier to the first class.
- Create a composite classifier diagram inside the class to show its internal structure, and create classes or interfaces there:
  - Right-click a class or interface in the Browser, and select
  - Press the `ctrl` key and double-click a class or interface.

#### i Note

The composite classifier diagram is empty by default, even if the classifier already includes some inner classifiers. To display symbols for existing internal classifiers in the diagram, select , or drag and drop them from the Browser.

Inner classifiers are listed in the Browser as children of their composite classifier and in the bottom of the class symbol. They can be shown in the main diagram connected to the composite classifier via inner links or within the

composite classifier symbol (right-click the symbol of a classifier containing a diagram and select  [Composite View](#)  [Read-only \(Sub-Diagram\)](#)):



 **Note**

You can display multiple composite classifiers in a composite structure diagram (see [Composite Structure Diagrams \[page 33\]](#)).

### 1.3.5.9 Specifying a Classifier as a Data Type or Return Type

You can specify a class or an interface in the current model or in another model (including a JDK library) as an attribute or parameter data type or as an operation return type. If the classifier belongs to the current model or package, it is displayed together with the other classifiers. If it belongs to another model or package, a shortcut of the classifier is created in the current package.

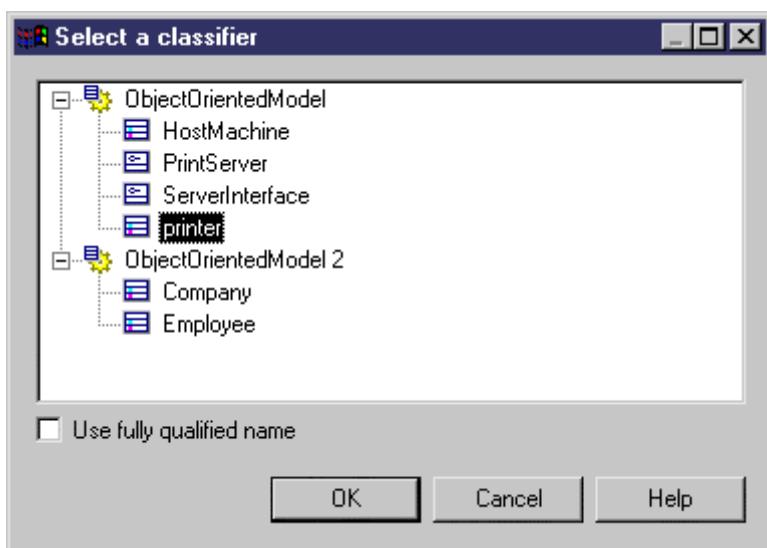
#### Context

 **Note**

For information about generating classifiers linked in this way, see [Managing Persistence for Complex Data Types \[page 246\]](#).

## Procedure

1. Open the appropriate object property sheet:
  - To specify a classifier as an attribute data type, open the attribute property sheet (see [Attribute Properties \[page 69\]](#)).
  - To specify a classifier as an operation return type, open the operation property sheet (see [Operation Properties \[page 79\]](#)).
  - To specify a classifier as an operation parameter data type, open the operation property sheet (see [Parameters \(OOM\) \[page 83\]](#)).
2. On the *General* tab, click the *Select Classifier* tool to the right of the *Data type* field, and select a classifier from the list, which lists all available classifiers in all models open in the workspace.



### i Note

Select the *Use fully qualified name* option to include the package hierarchy leading to the classifier.

3. Click *OK* to specify the classifier as the Data type.

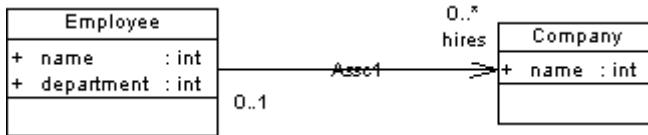
### i Note

You can alternatively enter the code of the classifier directly in the *Data type* field. To enter a qualified name use a dot separator. For example `Manufacturing.Core.Person`.

## 1.3.5.10 Viewing the Migrated Attributes of a Class

Navigable associations migrate attributes to classes during code generation. You can display these migrated attributes in the *Associations* tab of a class property sheet.

In the following example, the class *Employee* is associated with the class *Company*.



If you preview the generated code of the class **Employee**, you can see the following three attributes (in Java language):

```

public class EMPLOYEE
{
    public COMPANY hires[];
    public int NAME;
    public int DEPARTMENT;
}

```

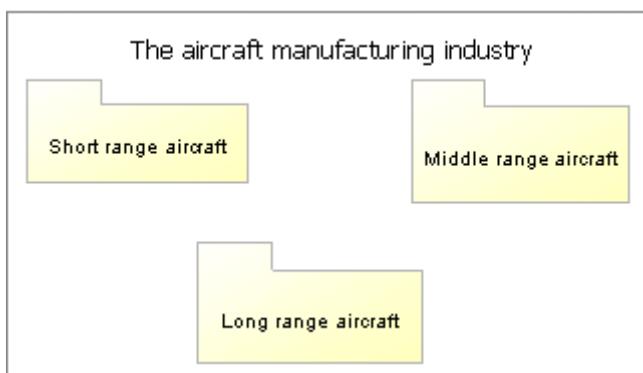
The association between **Employee** and **Company** is migrated as the attribute `public COMPANY hires []`.

You can use the [Associations](#) tab of a class property sheet to display the list of all migrated attributes proceeding from navigable associations.

## 1.3.6 Packages (OOM)

A package is a general purpose mechanism for organizing elements into groups. It contains model objects and is available for creation in all diagrams.

When you work with large models, you can split them into smaller subdivisions to avoid manipulating the entire set of data of the model. Packages can be useful to assign portions of a model, representing different tasks and subject areas to different development teams.

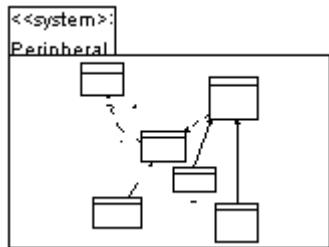


You can create several packages at the same hierarchical level within a model, or decompose a package into other packages and continue this process without limitation in decomposition depth. Each package at each level of decomposition can contain one or more diagrams.

### Note

In activity and statechart diagrams, you do not create packages but instead decompose activities and states, which act like packages in this context.

You can expand a package to view its contents by right-clicking its symbol and selecting  **Composite View**  **Read-only (Sub-Diagram)**. You may need to resize the symbol to see all its content. Double-click the composite symbol to go to the package diagram.



To return to the standard symbol, right-click the symbol and select  **Composite View**  **None**.

### 1.3.6.1 OOM Package Properties

Packages have properties displayed on property sheets. All packages share the following common properties:

| Property              | Description  |
|-----------------------|--|
| Name/Code/<br>Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <b>Code</b> field. |

| Property             | Description  |
|----------------------|--|
| Stereotype           | <p>Sub-classification derived from an existing package. The following stereotypes are available by default:</p> <ul style="list-style-type: none"> <li>• &lt;&lt;archive&gt;&gt; – Jar or War archive (Java only)</li> <li>• &lt;&lt;assembly&gt;&gt; – Specifies that a package produces a portable executable (PE), (C# and VB.NET only)</li> <li>• &lt;&lt;CORBAModule&gt;&gt; – UML Package identified as IDL module (IDL-CORBA only)</li> <li>• &lt;&lt;facade&gt;&gt; – Package is a view of another package</li> <li>• &lt;&lt;framework&gt;&gt; – Package consists mostly of patterns</li> <li>• &lt;&lt;metamodel&gt;&gt; – Package is an abstraction of another package</li> <li>• &lt;&lt;model&gt;&gt; – Specifies a semantically closed abstraction of a system</li> <li>• &lt;&lt;stub&gt;&gt; – Package serves as a proxy for the public contents of another package</li> <li>• &lt;&lt;subsystem&gt;&gt; – Grouping of elements, some of which constitute a specification of the behavior offered by the other contained elements</li> <li>• &lt;&lt;system&gt;&gt; – Package represents the entire system being modeled</li> <li>• &lt;&lt;systemModel&gt;&gt; – Package that contains other packages with the same physical system. It also contains all relationships and constraints between model elements contained in different models</li> <li>• &lt;&lt;topLevel&gt;&gt; – Indicates the top-most package in a containment hierarchy</li> </ul> |
| Default diagram      | Diagram displayed by default when opening the package  |
| Use parent namespace | [package only] Specifies that the package does not represent a separate namespace from its parent and thus that objects created within it must have names that are unique within the parent container. If this property is not selected, then the package and its parent package or model can both contain classes that are called Class A.  |
| Keywords             | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

### 1.3.6.2 Defining the Diagram Type of a New Package

When you create a new package, the default diagram of the package is defined according to the various parameters.

- If you create a package using the [Package](#) tool from the Toolbox, the diagram is of the same type as the parent package or model.
- If you create a package from the Browser, the diagram is of the same type as existing diagrams in the parent package or model, if these diagrams share the same type. If diagrams in the parent package or model are of different types, you are asked to select the type of diagram for the new sub-package.
- If you create a package from the List of Packages, the diagram is of the same type as the active diagram.

## 1.3.7 Interfaces (OOM)

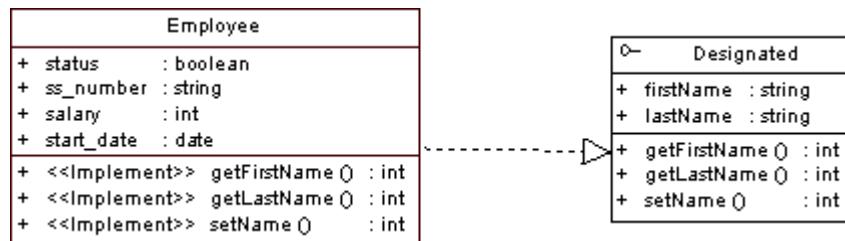
An interface is similar to a class but it is used to define the specification of a behavior. It is a collection of operations specifying the externally visible behavior of a class. It has no implementation of its own.

An interface can be created in the following diagrams:

- Class Diagram
- Composite Structure Diagram
- Component Diagram

An interface includes the signatures of the operations. It specifies only a limited part of the behavior of a class. A class can implement one or more interfaces.

A class must implement all the operations in an interface to realize the interface. The following example shows an interface (Designated) realized by a class (Employee).



### 1.3.7.1 Creating an Interface

You can create an interface from a class, or from the Toolbox, Browser, or *Model* menu.

- Select the *Interface* tool in the Toolbox.
- Select **Model > Interfaces** to access the List of Interfaces, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Interface**.
- Right-click a class, and select *Create Interface* from the contextual menu (this method allows you to inherit all the operations of the class, including the getter and setter operations, creates a realization link between the interface and the class, and shows this link in the *Realizes* sub-tab of the *Dependencies* tab of the interface property sheet).

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.3.7.2 Interface Properties

To view or edit an interface's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field.   |
| Extends           | Indicates the name of the class or interface that the current interface extends.   |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.<br><br>The following common stereotypes are available by default: <ul style="list-style-type: none"><li>• &lt;&lt;metaclass&gt;&gt; - interface that will interact with a model that contains classes with metaclass stereotypes</li><li>• &lt;&lt;powertype&gt;&gt; - a metaclass whose instances are sub-classes of another class</li><li>• &lt;&lt;process&gt;&gt; - heavyweight flow that can execute concurrently with other processes</li><li>• &lt;&lt;thread&gt;&gt; - lightweight flow that can execute concurrently with other threads within the same process. Usually executes inside the address space of an enclosing process</li><li>• &lt;&lt;utility&gt;&gt; - a class that has no instances</li></ul> |
| Visibility        | Specifies the visibility of the object, how it is seen outside its enclosing namespace. When an interface is visible to another object, it may influence the structure or behavior of the object, or similarly, the other object can affect the properties of the interface. You can choose between: <ul style="list-style-type: none"><li>• Private – only to the object itself</li><li>• Protected – only to the object and its inherited objects</li><li>• Package – to all objects contained within the same package</li><li>• Public – to all objects (option by default)</li></ul>   |
| Inner to          | Indicates the name of the class or interface to which the current interface belongs as an inner classifier.  |

| Property      | Description  |
|---------------|--|
| Type          | <p>Allows you to specify that an interface is a generic type, or that it is bound to one. You can choose between:</p> <ul style="list-style-type: none"> <li>• <b>Interface</b></li> <li>• <b>Generic</b></li> <li>• <b>Bound</b> – If you select this option, then an additional list becomes available to the right, where you can specify the generic type to which the interface is bound.</li> </ul> <p>If you specify either <b>Generic</b> or <b>Bound</b>, then the <b>Generic</b> tab is displayed, allowing you to control the associated type variables (see <a href="#">Creating Generic Types [page 50]</a>).</p> |
| Generate code | The interface is automatically included among the objects generated from the model when you launch the generation process.   |
| Keywords      | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

The following tabs list objects associated with the interface:

- [Attributes](#) - lists the attributes associated with the interface. You can create attributes directly in this page, or add already existing attributes. For more information, see [Attributes \(OOM\) \[page 67\]](#).
- [Operations](#) - lists the operations associated with the interface. You can create operations directly in this page, or add already existing operations. For more information, see [Operations \(OOM\) \[page 77\]](#).
- [Generic Parameters](#) - lets you specify the type parameters of a generic interface or values for the required type parameters for an interface that is bound to a generic type (see [Creating Generic Types \[page 50\]](#))
- [Inner Classifiers](#) - lists the inner classes and interfaces associated with the interface. You can create inner classifiers directly in this page. For more information, see [Creating Composite and Inner Classifiers \[page 54\]](#).
- [Related Diagrams](#) - lists and lets you add model diagrams that are related to the interface (see [Core Features Guide > Modeling with PowerDesigner > Diagrams, Matrices, and Symbols > Diagrams > Specifying Diagrams as Related Diagrams](#)).

## 1.3.8 Objects (OOM)

At the conceptual level, an object is an element defined as being part of the system described. It represents an object that has not yet been instantiated because the classes are not yet clearly defined at this stage.

If you need to go further with the implementation of your model, the object that has emerged during analysis will probably turn into an instance of a defined class. In this case, an object is considered an instance of a class.

Three possible situations can be represented:

- When an object is not an instance of a class - it has only a name
- When an object is an instance of a class - it has a name and a class
- When an object is an instance of a class but is actually representing any or all instances of a class - it has a class but no name

An object can be created in the following diagrams:

- Communication Diagram
- Object Diagram
- Sequence Diagram

The object shares the same concept in the object, sequence and communication diagrams. It can either be created in the diagram type you need, or dragged from a diagram type and dropped into another diagram type.

## Defining Multiples

A multiple defines a set of instances. It is a graphical representation of an object that represents several instances, however a multiple can only hold one set of attributes even if it represents several instances. An object can communicate with another object that is a multiple. This feature is mainly used in the communication diagram but can also be used in the object diagram.

A clerk handles a list of documents: it is the list of documents that represents a multiple object.

When the Multiple check box is selected in the object property sheet, a specific symbol (two superposed rectangles) is displayed.



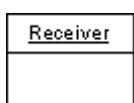
## Objects in an Object Diagram

In the object diagram, an object instance of a class can display the values of attributes defined on the class. When the class is deleted, the associated objects are not deleted.

## Objects in a Communication Diagram

In a communication diagram, an object is an instance of a class. It can be persistent or transient: persistent is the situation of an object that continues to exist after the process that created it has finished, and transient is the situation of an object that stops to exist when the process that created it finishes.

The name of the object is displayed underlined. The Underline character traditionally indicates that an element is an instance of another element.



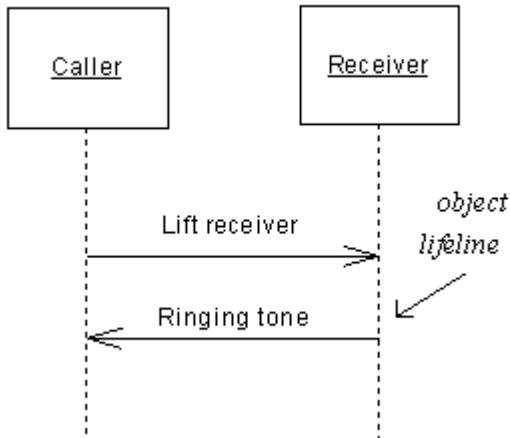
## Objects in a Sequence Diagram

In the sequence diagram, an object has a lifeline: it is the dashed vertical line under the object symbol. Time always proceeds down the page. The object lifeline indicates the period during which an object exists. You cannot separate an object and its lifeline.

If the object is created or destroyed during the period of time shown on the diagram, then its lifeline starts or stops at the corresponding point.

Objects appear at the top of the diagram. They exchange messages between them.

An object that exists when a transaction, or message starts, is shown at the top of the diagram, above the first message arrow. The lifeline of an object that still exists when the transaction is over, continues beyond the final message arrow.



### 1.3.8.1 Creating an Object

You can create an object from the Browser or *Model* menu.

- Select the *Object* tool in the Toolbox.
- Select **Model > Objects** to access the List of Objects, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Object**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.3.8.2 Object Properties

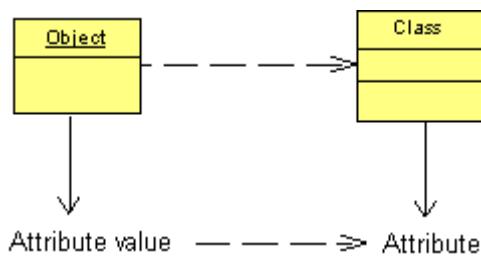
To view or edit an object's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description   |
|-------------------|---|
| Name/Code/Comment | <p>Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field.</p> <p>You need not specify a name (as you can have an object representing an unnamed instance of a class or interface), but in this case, you must specify a <i>Classifier</i>. Names must be unique per classifier.</p> |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.  |
| Classifier        | Specifies the class or interface of which an object is an instance. You can link an object to an existing class or interface, or create a new one using the Create Class button beside this box (see <a href="#">Linking a Classifier to an Object [page 66]</a> ).   |
| Multiple          | Specifies that the object represents multiple instances.  |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.   |

## Attribute Values Tab

An attribute value is an instance of a class attribute from the class of which the object is an instance, or of an attribute inherited from a parent of the class.



You can add class attributes to the object and assign values to them on the Attribute Values tab using the [Add Attribute Values](#) tool, which opens a dialog listing all attributes of the class of the object, including inherited

attributes of classes from which the class inherits. Once the attribute is added to the object, you can specify its value in the Value column. All other columns are read-only.

You can control the display of attribute values on object symbols using display preferences (▶ *Tools* ▶ *Display Preferences* ▶):

|                |
|----------------|
| PC:Computer    |
| OS = Win2000   |
| name = Dev_lab |

### 1.3.8.3 Linking a Classifier to an Object

The object diagram represents instances of class or interface, the sequence diagram represents the dynamic behavior of a class or interface, and the communication diagram represents those instances in a communication mode. For all these reasons, you can link a class or an interface to an object in an OOM.

#### Context

From the object property sheet, you can:

- Link the object to an existing class or interface
- Create a class

#### Procedure

1. Select a class or interface from the *Classifier* list in the object property sheet.

or

Click the *Create Class* tool beside the *Classifier* list to create a class and display its property sheet.

Define the properties of the new class and click *OK*.

The class or interface name is displayed in the *Classifier* list.

2. Click *OK*.

The object name is displayed in the sequence diagram, followed by a colon, and the name of the class or interface selected.

You can similarly view the object name in the class or interface property sheet: click the *Dependencies* tab and select the *Objects* sub-tab. The object name is automatically added in this sub-tab.

## Results

### Note

You can drag a class or interface node from the Browser and drop it into the sequence, communication or object diagrams. You can also copy a class or interface and paste it, or paste it as shortcut, into these diagrams. This automatically creates an object, instance of the class or of the interface.

## 1.3.9 Attributes (OOM)

An attribute is a named property of a class (or an interface) describing its characteristics.

An attribute can be created for a class or interface in the following diagrams:

- Class Diagram
- Composite Structure Diagram
- Component Diagram

A class or an interface may have none or several attributes. Each object in a class has the same attributes, but the values of the attributes may be different.

Attribute names within a class must be unique. You can give identical names to two or more attributes only if they exist in different classes.

In the following example, the class Printer contains two attributes: printspeed and laser:

| Printer      |           |
|--------------|-----------|
| + printspeed | : int     |
| + laser      | : boolean |

## Interface Attributes

An attribute of an interface is slightly different from an attribute of a class because an interface can only have constant attributes (static and frozen). For example, consider an interface named Color with three attributes RED, GREEN, and BLUE. They are all static, final and frozen.

| Color   |                  |
|---------|------------------|
| + RED   | : int = 0xFF0000 |
| + GREEN | : int = 0x00FF00 |
| + BLUE  | : int = 0x0000FF |

If you generate in Java, you see:

```
public interface Color
{
    public static final int RED = 0xFF0000;
    public static final int GREEN = 0x00FF00;
    public static final int BLUE = 0x0000FF;
}
```

All these attributes are constants because they are static (independent from the instances), final (they can not be overloaded), and frozen (their value cannot be changed).

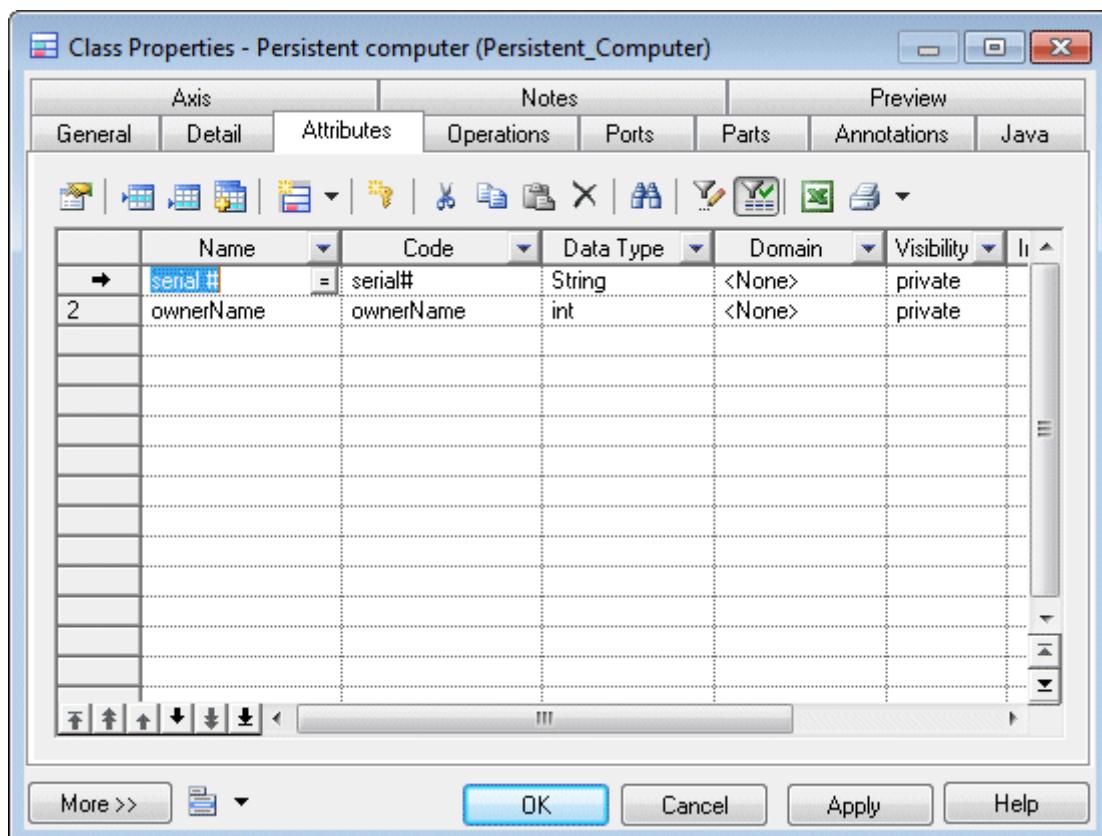
You can use the attributes of other interfaces or classes and add them to the current interface.

### 1.3.9.1 Creating an Attribute

You can create an attribute from the property sheet of a class, identifier, or interface.

## Procedure

1. Open the property sheet of your classifier and click the *Attributes* tab.



2. Use one of the following tools to add attributes:

| Tool | Description   |
|------|---|
|      | <i>Add a Row / Insert a Row</i> - Enter a name and any other appropriate properties. Alternatively, right-click a class or interface in the Browser, and select ► <b>New &gt; Attribute</b> ▶.  |
|      | <i>Add Attributes</i> - Select from a list of existing attributes to add to the classifier. If the classifier already contains an attribute with the same name or code, the copied attribute is renamed by appending a number.  |
|      | [PowerBuilder] <i>Override Inherited Attributes</i> - Select from a list of the attributes that belong to all the parents of the classifier to add a copy to the child. Inherited attributes are grayed to indicate that their properties cannot be modified (except for their initial value), and their stereotype is set to <<Override>>. |

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

### 1.3.9.2 Attribute Properties

To view or edit an attribute's properties, double-click its Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description   |
|-------------------|---|
| Parent            | Specifies the classifier to which the attribute belongs.  |
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field.  |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.  |
| Visibility        | Specifies the visibility of the object, how it is seen outside its enclosing namespace. When a class is visible to another object, it may influence the structure or behavior of the object, or similarly, the other object can affect the properties of the class. You can choose between: <ul style="list-style-type: none"> <li>• <b>Private</b> – only to the class to which it belongs</li> <li>• <b>Protected</b> – only to the class and its derived objects</li> <li>• <b>Package</b> – to all objects contained within the same package</li> <li>• <b>Public</b> – to all objects (option by default)</li> </ul> |

| Property     | Description  |
|--------------|--|
| Data type    | Set of instances sharing the same operations, abstract attributes, relationships, and semantics.   |
| Multiplicity | <p>Specifies the range of allowable number of values the attribute may hold. You can choose between:</p> <ul style="list-style-type: none"> <li>• 0..1 – zero or one</li> <li>• 0..* – zero to unlimited</li> <li>• 1..1 – exactly one</li> <li>• 1..* – one to unlimited</li> <li>• * – none to unlimited</li> </ul> <p>You can change the default format of multiplicity from the registry.</p> <pre>HKEY_CURRENT_USER\Software\Sybase\&lt;PowerDesigner version&gt; \ModelOptions\Cld\ MultiplicityNotation = 1 (0..1) or 2 (0,1)</pre> |
| Array size   | <p>Specifies multiplicity in the syntax of a given language, when attribute multiplicity cannot express it. For example, you can set array size to [4,6,8] to get the PowerBuilder syntax int n[4,6,8] or set array size to [..] to get the c# syntax int[..] n;</p> <p>Depending on the model language, the following will be generated:</p> <ul style="list-style-type: none"> <li>• Java, C# and C++ – [2][4][6]</li> <li>• PowerBuilder – [2,4,6]</li> <li>• VB .NET – (2,4,6)</li> </ul>  |
| Enum class   | [Java 5.0 and higher] Specifies an anonymous class for an EnumConstant. Use the tools to the right of the field to create, browse for, or view the properties of the currently selected class.   |
| Static       | The attribute is associated with the class, as a consequence, static attributes are shared by all instances of the class and have always the same value among instances.   |
| Derived      | Indicates that the attribute can be computed from another attribute. The derivation formula can be defined in the attribute description tab, it does not influence code generation.  |
| Mandatory    | Boolean calculated attribute selected if the minimum multiplicity is greater than 0.   |
| Volatile     | Indicates that the attribute is not a member of the class. It is only defined by getter and setter operations, in C# it replaces the former extended attribute volatile. For more information on adding operations to a class, see <a href="#">Adding Getter and Setter Operations to a Classifier [page 72]</a> .   |
| Keywords     | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## Detail Tab

The *Detail* tab contains the following properties:

| Property           | Description  |
|--------------------|--|
| Initial value      | Specifies the initial value assigned to the attribute on creation.   |
| Changeability      | Specifies if the value of the attribute can be modified once the object has been initialized. You can choose between: <ul style="list-style-type: none"><li>• <i>Changeable</i> – The value can be changed</li><li>• <i>Read-only</i> – Prevents the creation of a setter operation (a setter is created in the method inside the class)</li><li>• <i>Frozen</i> – Constant</li><li>• <i>Add-only</i> – Allows you to add a new value only</li></ul> |
| Domain             | Specifies a domain (see <a href="#">Domains (OOM) [page 109]</a> ), which will define the data type and related data characteristics for the attribute and may also indicate check parameters, and business rules.<br>Select a domain from the list, or click the Ellipsis button to the right to create a new domain in the List of Domains.  |
| Primary Identifier | [class attributes] Specifies that the attribute is part of a primary identifier. Primary identifiers are converted to primary keys after generation of an OOM to a PDM. Exists only in classes   |
| Migrated from      | Specifies the attribute being overridden (PowerDesigner only) or association migrated (see <a href="#">Migrating Association Roles [page 91]</a> ). Click the <i>Properties</i> tool to the right of the field to open the referenced object's property sheet.   |
| Persistent         | [class attributes] Specifies that the attribute will be persisted and stored in a database (see <a href="#">Managing Object Persistence During Generation of Data Models [page 245]</a> ).   |
| Code               | Specifies the code of the table or entity that will be generated in a persistent CDM or PDM model.   |
| Data type          | Specifies a persistent data type used in the generation of a persistent model, either CDM or PDM. The persistent data type is defined from default PowerDesigner conceptual data types.  |
| Length             | Specifies the maximum number of characters of the persistent data type.  |
| Precision          | Specifies the number of places after the decimal point, for persistent data type values that can take a decimal point  |

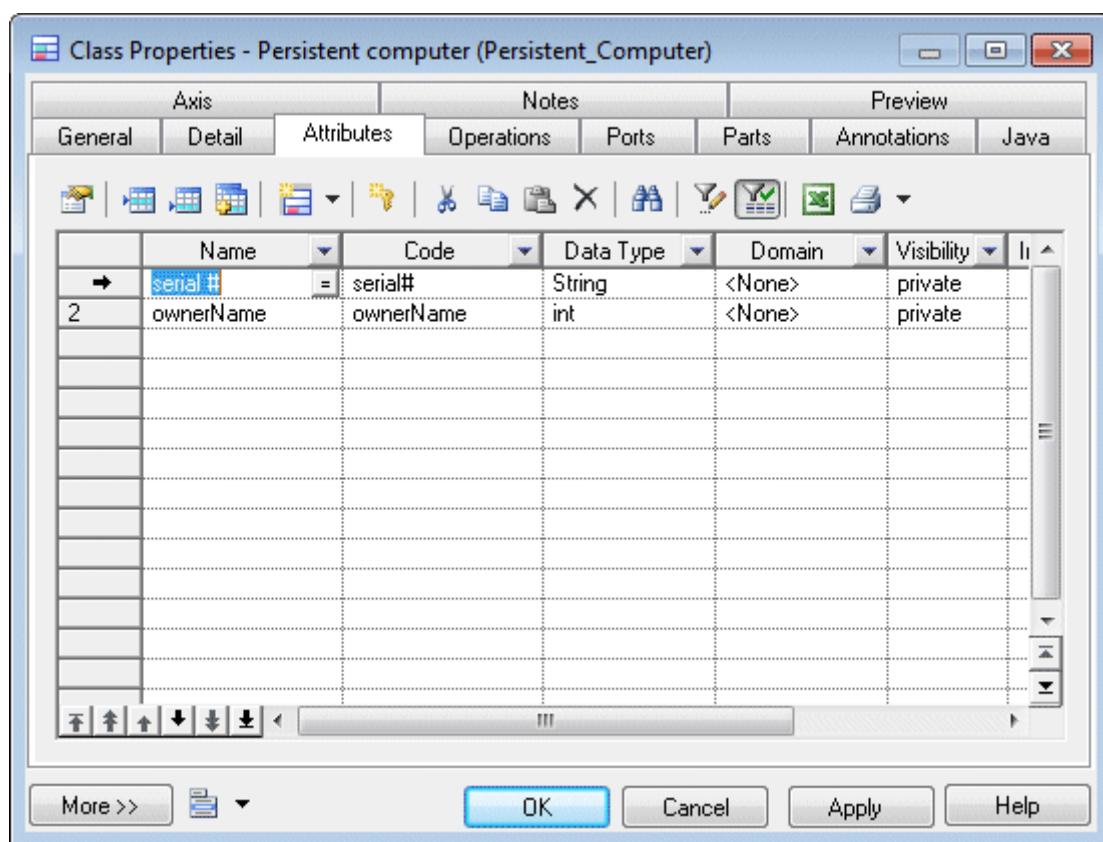
### 1.3.9.3 Adding Getter and Setter Operations to a Classifier

PowerDesigner helps you to quickly create Getter and Setter operations for your attributes from the [Attributes](#) tab of your classifier.

## Context

## Procedure

1. Open the property sheet of your classifier and click the *Attributes* tab.



2. Select one or more attributes and then click the *Add...* button at the bottom of the attributes tab and select the action you want to perform. Depending on the target language, some of the following actions will be available:
    - Get/Set Operations - Creates get and set operations on the *Operations tab* for the selected attributes
    - Property - [C#/VB.NET only] Creates a property on the *Attributes tab* and get and set operations on the *Operations tab* to access the original attribute via the property.

- Indexer - [C# only] Creates an indexer on the *Attributes* tab and get and set operations on the *Operations tab* to access the original attribute via the indexer.
  - Event Operations - [C#/VB.NET only, for attributes with the Event stereotype] Creates add and remove operations on the *Operations tab* for the event.
3. [optional] Click the *Operations* tab to view the newly created operations. Certain values, including the names cannot be modified.
  4. Click **OK** to close the property sheet and return to your model.

### 1.3.9.4 Setting Data Profiling Constraints

PowerDesigner supports data profiling in the physical data model (PDM) and the *Standard Checks* and *Additional Checks* tabs are provided in OOM attribute and domain property sheets solely to retain this information when linking and synching between an OOM and a PDM.

The following constraints are available on the *Standard Checks* tab of OOM attributes and domains:

| Property        | Description  |
|-----------------|--|
| Values          | <p>Specifies the range of acceptable values. You can set a:</p> <ul style="list-style-type: none"> <li>● Minimum - The lowest acceptable numeric value</li> <li>● Maximum - The highest acceptable numeric value</li> <li>● Default - The value assigned in the absence of an expressly entered value.</li> </ul>  |
| Characteristics | <p>These properties are for documentation purposes only, and will not be generated. You can choose a:</p> <ul style="list-style-type: none"> <li>● Format - A number of standard formats are available in the list. You can enter a new format directly in the field or use the tools to the right of the field to create a data format for reuse elsewhere.</li> <li>● Unit - A standard measure.</li> <li>● No space - Space characters are not allowed.</li> <li>● Cannot modify - The value cannot be updated after initialization.</li> </ul> |
| Character case  | <p>Specifies the acceptable case for the data. You can choose between:</p> <ul style="list-style-type: none"> <li>● Mixed case [default]</li> <li>● Uppercase</li> <li>● Lowercase</li> <li>● Sentence case</li> <li>● Title case</li> </ul>   |
| List of values  | <p>Specifies the various values that are acceptable.</p> <p>Select the <i>Complete</i> check box beneath the list to exclude all other values not appearing in the list.</p>   |

## 1.3.9.4.1 Creating Data Formats For Reuse

You can create data formats to reuse in constraints for multiple objects by clicking the [New](#) button to the right of the [Format](#) field on the [Standard Checks](#) tab. Data formats are informational only, and are not generated as constraints.

### i Note

To create multiple data formats, use the List of Data Formats, available by selecting [Model](#) [Data Formats](#).

## Data Format Properties

To view or edit a data format's properties, double-click its Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The [General](#) tab contains the following properties:

| Property              | Description   |
|-----------------------|---|
| Name/Code/<br>Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <a href="#">Code</a> field. |
| Stereotype            | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.  |
| Type                  | Specifies the type of the format. You can choose between: <ul style="list-style-type: none"><li>• Date/Time</li><li>• String</li><li>• Regular Expression</li></ul>   |
| Expression            | Specifies the form of the data to be stored in the column; For example, 9999 . 99 would represent a four digit number with two decimal places.  |
| Keywords              | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.   |

## 1.3.9.4.2 Specifying Advanced Constraints

The *Additional Checks* tab is used in the physical data model (PDM) to specify complex column constraints, and is provided in OOM attribute and domain property sheets solely to retain this information when linking and synching between an OOM and a PDM.

## 1.3.10 Identifiers (OOM)

An identifier is a class attribute, or a combination of class attributes, whose values uniquely identify each occurrence of the class. It is used during intermodel generation when you generate a CDM or a PDM into an OOM, the CDM identifier and the PDM primary or alternate keys become identifiers in the OOM.

An identifier can be created for a class in the following diagrams:

- Class Diagram
- Composite Structure Diagram

Each class can have at least one identifier. Among identifiers, the primary identifier is the main identifier of the class. This identifier corresponds to a primary key in the PDM.

When you create an identifier, you can attach attributes or business rules to it. You can also define one or several attributes as being primary identifier of the class.

For example, the social security number for a class employee is the primary identifier of this class.

### 1.3.10.1 Creating an Identifier

You can create an identifier from the property sheet of a class or interface.

Open the *Identifiers* tab in the property sheet of a class or interface, and click the *Add a Row* tool. To specify a primary identifier, select the appropriate identifier in the list, click the *Properties* tool to open its property sheet, and then select the *Primary Identifier* property.

#### i Note

You can, alternatively, create a primary identifier from an attribute, by selecting the *Primary Identifier* property on the *Detail* tab of the attribute property sheet (see [Attribute Properties \[page 69\]](#)).

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.3.10.2 Identifier Properties

To view or edit an identifier's properties, double-click its Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

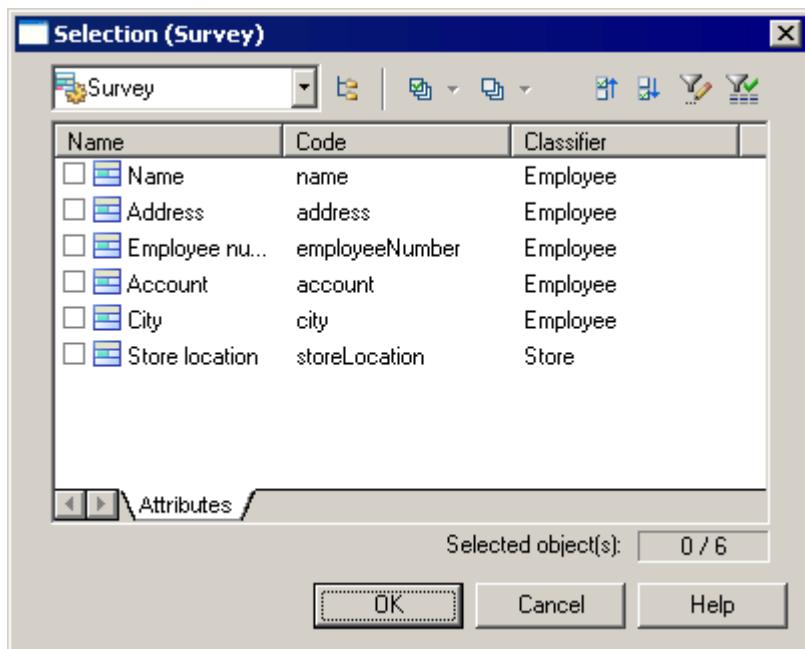
| Property           | Description  |
|--------------------|--|
| Parent             | Specifies the class to which the identifier belongs.   |
| Name/Code/Comment  | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Primary Identifier | Specifies that the identifier is a primary identifier.   |
| Keywords           | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## 1.3.10.3 Adding Attributes to an Identifier

An identifier can contain one or several attributes. You can add these attributes to an identifier to further characterize the identifier.

### Procedure

1. Select an identifier from the List of Identifiers or the *Identifiers* tab in the property sheet of a class, and click the *Properties* tool to display its property sheet.
2. Click the *Attributes* tab and click the *Add Attributes* tool to display the list of attributes for the class.



3. Select the check boxes for the attributes you want to add to the identifier.
4. Click **OK** in each of the dialog boxes.

### 1.3.11 Operations (OOM)

An operation is a named specification of a service that can be requested from any object of a class to affect behavior. It is a specification of a query that an object may be called to execute.

An operation can be created for a class or interface in the following diagrams:

- Class Diagram
- Composite Structure Diagram
- Component Diagram

A class may have any number of operations or no operations at all.

In the following example, the class Car, has 3 operations: start engine, brake, and accelerate.

| Car                    |
|------------------------|
| trademark              |
| model                  |
| engine                 |
| + startEngine() : void |
| + brake() : void       |
| + accelerate() : void  |

Operations have a name and a list of parameters. Several operations can have the same name within the same class if their parameters are different.

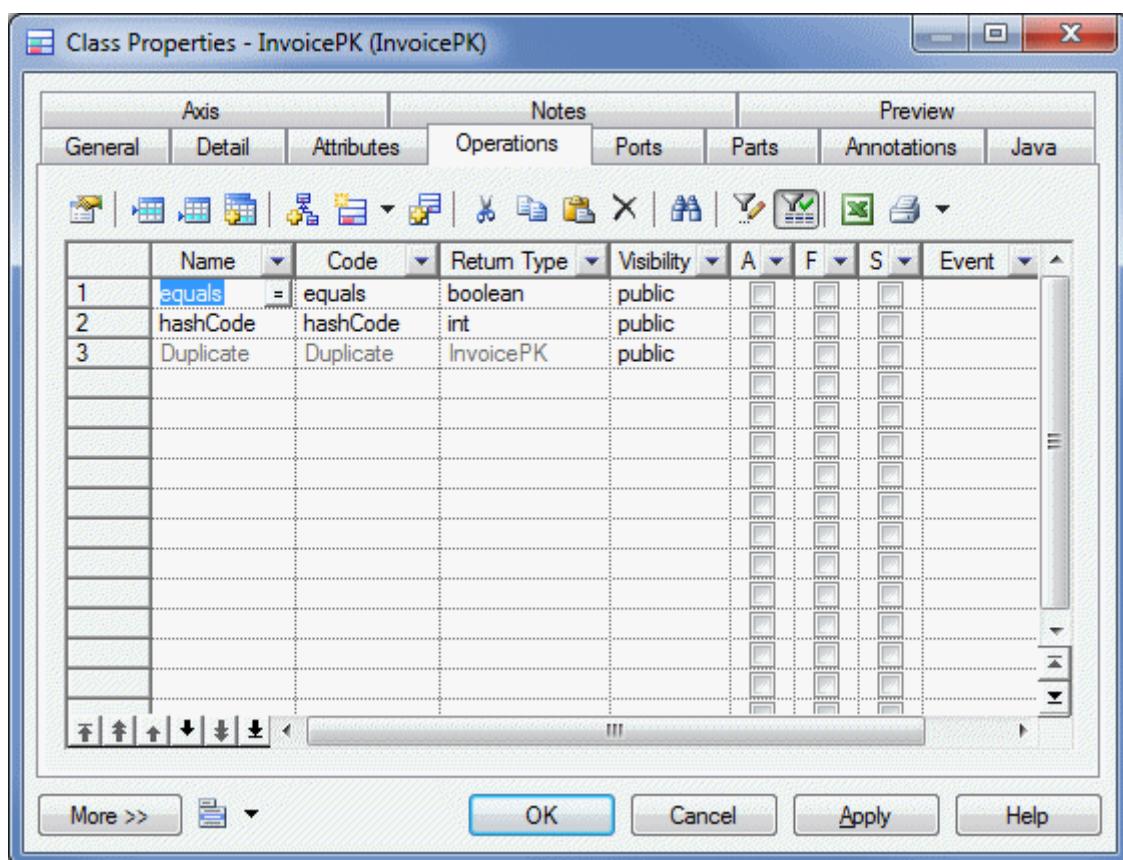
For more information on EJB operations, see [Defining Operations for EJBs \[page 309\]](#).

### 1.3.11.1 Creating an Operation

You can create an operation from the property sheet of, or in the Browser under, a class or interface.

#### Procedure

1. Open the property sheet of your classifier and click the *Operations* tab:



2. Use one of the following tools to add operations:

| Tool | Description  |
|------|--|
|      | <i>Add a Row / Insert a Row</i> - Enter a name and any other appropriate properties. Alternatively, right-click a class or interface in the Browser, and select <b>New &gt; Operation</b> .                                    |
|      | <i>Add Operations</i> - Select from a list of existing operations to add to the classifier. If the classifier already contains an operation with the same name or code, the copied operation is renamed by appending a number. |

| Tool  | Description  |
|---|--|
|  | <p><i>Override Inherited Operations</i> - Select from a list of the operations that belong to all the parents of the classifier to add a copy to the child bearing the stereotype &lt;&lt;Override&gt;&gt;. Each operation has the same signature (name and parameters) as the original operation, and you can only modify the code on its <i>Implementation</i> tab.</p>  |
|  | <p><i>Add...</i> - Click the arrow to the right of the tool and select the appropriate type of standard operation from the list:</p> <ul style="list-style-type: none"> <li>○ Default Constructor/Destructor - to perform initialization/cleanup for classifiers. You can add parameters afterwards.</li> <li>○ Copy Constructor - to copy the attributes of a class instance to initialize another instance.</li> <li>○ Initializer/Static Initializer- [Java only] to initialize a class before any constructor.</li> <li>○ Duplicate Operation - to create and initialize an instance of a class within the class.</li> <li>○ Activate/Deactivate Operations - [PowerBuilder only]</li> </ul> <p>Some or all of the operation's properties will be dimmed to indicate that they are uneditable.</p>   |
|  | <p><i>Unimplemented Operations</i> - Select from a list of the operations defined in interfaces to which the classifier is connected by realization links, and waiting to be implemented by the classifier to add a copy to the child bearing the stereotype &lt;&lt;Implement&gt;&gt;. Each operation has the same signature (name and parameters) as the original operation, and you can only modify the code on its <i>Implementation</i> tab.</p> <div style="background-color: #ffffcc; padding: 10px;"> <p><b>i Note</b></p> <p>To automatically create the necessary operations in your class, click  <b>Tools</b>  <b>Model Options</b>, and select the <i>Auto-Implement Realized Interfaces</i> option.</p> </div> |

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

### 1.3.11.2 Operation Properties

To view or edit an operation's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property | Description   |
|----------|---|
| Parent   | Specifies the parent classifier to which the operation belongs. |

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | <p>Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <a href="#">Code</a> field.</p>   |
| Stereotype        | <p>Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.</p> <p>The following common stereotypes are available by default:</p> <ul style="list-style-type: none"> <li>• &lt;&lt;constructor&gt;&gt; - Operation called during the instantiation of an object that creates an instance of a class</li> <li>• &lt;&lt;create&gt;&gt; - Operation used by a class when instantiating an object</li> <li>• &lt;&lt;destroy&gt;&gt; - Operation used by a class that destroys an instance of a class</li> <li>• &lt;&lt;storedProcedure&gt;&gt; - Operation will become a stored procedure in the generated PDM</li> <li>• &lt;&lt;storedFunction&gt;&gt; - Operation will become a stored function in the generated PDM</li> <li>• &lt;&lt;EJBCreateMethod&gt;&gt; - EJB specific CreateMethod</li> <li>• &lt;&lt;EJBFinderMethod&gt;&gt; - EJB specific FinderMethod</li> <li>• &lt;&lt;EJBSelectMethod&gt;&gt; - EJB specific SelectMethod</li> </ul> <p>For more information on EJB specific methods, see <a href="#">Defining Operations for EJBs [page 309]</a>.</p> |
| Return Type       | <p>A list of values returned by a call of the operation. If none are returned, the return type value is null</p>   |
| Visibility        | <p>[class operators] Visibility of the operation, whose value denotes how it is seen outside its enclosing name space:</p> <ul style="list-style-type: none"> <li>• <b>Private</b> - Only to the class to which it belongs</li> <li>• <b>Protected</b> - Only to the class and its derived objects</li> <li>• <b>Package</b> - To all objects contained within the same package</li> <li>• <b>Public</b> - To all objects</li> </ul>   |
| Language event    | <p>When classes represent elements of interfaces, this box allows you to show an operation as triggered by a significant occurrence of an event</p>  |
| Static            | <p>The operation is associated with the class, as a consequence, static operations are shared by all instances of the class and have always the same value among instances</p>   |
| Array             | <p>Flag defining the return type of the operation. It is true if the value returned is a table</p>   |
| Abstract          | <p>The operation cannot be instantiated and thus has no direct instances</p>   |
| Final             | <p>The operation cannot be redefined</p>   |
| Read-only         | <p>Operation whose execution does not change the class instance</p>  |

| Property           | Description   |
|--------------------|---|
| Web service method | If displayed and selected, implies that the operation is used as a web service method   |
| Influent object    | Specifies the operation on which the current operation is based. In general, this is either a parent operation that is being overridden through a generalization link or an interface operation that is being implemented through a realization link. |
| Generic            | Specifies that the operation is a generic method (see <a href="#">Creating Generic Types [page 50]</a> ).   |
| Keywords           | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.   |

## Parameters Tab

The **Parameters** tab lists the parameters of your operation. Each parameter is a variable that can be changed, passed, or returned. A parameter has the following properties:

| Property          | Description  |
|-------------------|--|
| Parent            | Specifies the operation to which the parameter belongs.  |
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <b>Code</b> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Data type         | Set of instances sharing the same operations, abstract attributes, relationships, and semantics  |
| Array             | When selected, turns attributes into table format  |
| Array size        | Specifies an accurate array size when the attribute multiplicity is greater than 1.  |
| Variable Argument | Specifies that the method can take a variable number of parameters for a given argument. You can only select this property if the parameter is the last in the list.   |

| Property       | Description  |
|----------------|--|
| Parameter Type | <p>Direction of information flow for the parameter. Indicates what is returned when the parameter is called by the operation during the execution process. You can choose from the following:</p> <ul style="list-style-type: none"> <li>• <b>In</b> - Input parameter passed by value. The final value may not be modified and information is not available to the caller</li> <li>• <b>In/Out</b> - Input parameter that may be modified. The final value may be modified to communicate information to the caller</li> <li>• <b>Out</b> - Output parameter. The final value may be modified to communicate information to the caller</li> </ul> |
| Default value  | <p>Default value when a parameter is omitted. For example:</p> <p>Use an operation <code>oper (string param1, integer param2)</code>, and specify two arguments <code>oper(val1, val2)</code> during invocation. Some languages, like C++, allow you to define a default value that is then memorized when the parameter is omitted during invocation.</p> <p>If the declaration of the method is <code>oper(string param1, integer param2 = default)</code>, then the invocation <code>oper(val1)</code> is similar to <code>oper(val1, default)</code>.</p>  |
| WSDL data type | Only available with Web services. Defines the XML-Schema/SOAP type used during invocation of a Web method (using http or Soap)   |
| Keywords       | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## Implementation Tab

The *Implementation* tab allows you to specify the code that will be used to implement the operation, and contains the following sub-tabs at the bottom of the dialog:

| Items           | Description  |
|-----------------|--|
| Body            | Code of the implementation.  |
| Exceptions      | Signal raised in response to behavioral faults during system execution. Use the <i>Add Exception</i> tool to select an exception classifier to add at the cursor position. |
| Pre-conditions  | Constraint that must be true when the operation is invoked.  |
| Post-conditions | Constraints that must be true at the completion of the operation.  |
| Specification   | Similar to the pseudo code, it is a description of the normal sequence of actions.   |

The following tabs are also available:

- *Parameters* - lists the parameters of the operation. Each parameter is a variable that can be changed, passed, or returned (see [Parameters \(OOM\) \[page 83\]](#)).

- *Generic Parameters* - lets you specify the type parameters of a generic method (see [Creating Generic Types \[page 50\]](#)).
- *Related Diagrams* - lists and lets you add model diagrams that are related to the operation (see [Core Features Guide > Modeling with PowerDesigner > Diagrams, Matrices, and Symbols > Diagrams > Specifying Diagrams as Related Diagrams](#)).

### 1.3.11.3 Parameters (OOM)

A parameter is a variable that can be changed, passed, or returned. To view or edit a parameter's properties, select it on the *Parameters* tab of an operation or event and click the *Properties* tool.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Parent            | Specifies the operation or event to which the parameter belongs.   |
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Data type         | Set of instances sharing the same operations, abstract attributes, relationships, and semantics.   |
| Array             | Specifies that the data type is a table format.  |
| Array size        | Specifies an accurate array size when the attribute multiplicity is greater than 1.  |
| Variable Argument | Specifies that the method can take a variable number of parameters for a given argument. You can only select this property if the parameter is the last in the list.   |
| Parameter Type    | Direction of information flow for the parameter. You can choose from the following: <ul style="list-style-type: none"> <li>• <b>In</b> - Input parameter passed by value. The final value may not be modified and information is not available to the caller.</li> <li>• <b>In/Out</b> - Input parameter that may be modified. The final value may be modified to communicate information to the caller.</li> <li>• <b>Out</b> - Output parameter. The final value may be modified to communicate information to the caller.</li> </ul>          |

| Property       | Description   |
|----------------|---|
| Default value  | <p>Default value when a parameter is omitted. For example:</p> <p>Use an operation <code>oper (string param1, integer param2)</code>, and specify two arguments <code>oper(val1, val2)</code> during invocation. Some languages, like C++, allow you to define a default value that is then memorized when the parameter is omitted during invocation.</p> <p>If the declaration of the method is <code>oper(string param1, integer param2 = default)</code>, then the invocation <code>oper(val1)</code> is similar to <code>oper(val1, default)</code>.</p> |
| WSDL data type | Only available with Web services. Defines the XML-Schema/SOAP type used during invocation of a Web method (using http or Soap)  |
| Keywords       | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.   |

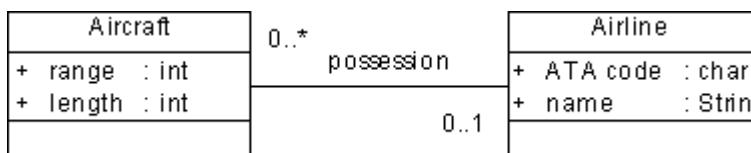
### 1.3.12 Associations (OOM)

An association represents a structural relationship between classes or between a class and an interface.

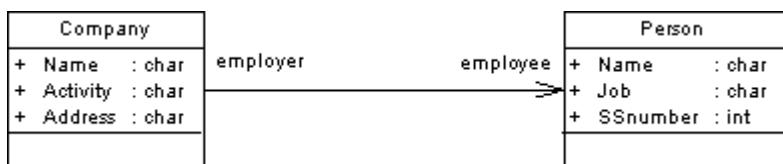
An association can be created in the following diagrams:

- Class Diagram
- Composite Structure Diagram

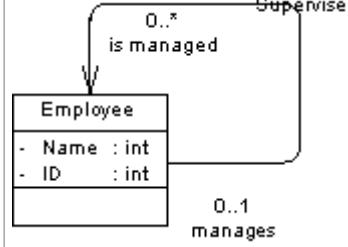
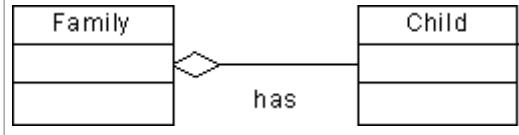
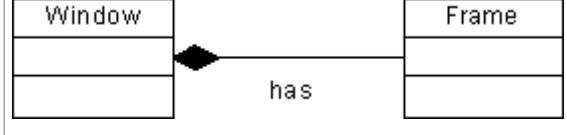
It is drawn as a solid line between the pair of objects.



In addition to naming the association itself, you can specify a role name for each end in order to describe the function of a class as viewed by the opposite class. For example, a person considers the company where he works as an employer, and the company considers this person as an employee.



The following special types of association are available:

|                       |   |
|-----------------------|---|
| Reflexive Association | <p>An association between a class and itself. In this example, the association <b>Supervise</b> expresses the fact that an employee can, at the same time, be a manager and someone to manage:</p>  <p>The <i>Dependencies</i> tab of the class lists two identical occurrences of the association to indicate that the association is reflexive and serves as origin and destination for the link.</p>  |
| Aggregation           | <p>An association in which one class represents a larger thing (a whole) made of smaller things (the parts). This is sometimes known as a "has-a" link, and allows you to represent the fact that an object of the whole has objects of the part. In this example, the family is the whole that can contain children:</p>  <p>You can create an aggregation directly using the <i>Aggregation</i> tool in the Toolbox.</p>                           |
| Composition           | <p>An aggregation in which the parts are strongly tied to the whole. In a composition, an object may be a part of only one composite at a time, and the composite object manages the creation and destruction of its parts. In this example, the frame is a part of a window. If you destroy the window object, the frame part also disappears:</p>  <p>You can create a composition directly using the <i>Composition</i> tool in the Toolbox.</p> |

You can define one of the roles of an association as being either an aggregation or a composition. The Container property needs to be defined to specify which of the two roles is an aggregation or a composition.

### 1.3.12.1 Creating an Association

You can create an association from the Toolbox, Browser, or *Model* menu.

- Use the *Association*, *Aggregation*, or *Composition* tool in the Toolbox.
- Select **Model > Associations** to access the List of Associations, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Association**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.3.12.2 Association Properties

To view or edit an association's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Class A/Class B   | Specifies the classes at each end of the association. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.   |
| Type              | Specifies the type of association. You can choose between: <ul style="list-style-type: none"><li>• <b>Association</b></li><li>• <b>Aggregation</b> – a part-whole relationship between a class and an aggregate class</li><li>• <b>Composition</b> – a form of aggregation but with strong ownership and coincident lifetime of parts by the whole</li></ul>   |
| Container         | If the association is an aggregation or a composition, the container radio buttons let you define which class contains the other in the association  |
| Association Class | Class related to the current association that completes the association definition   |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## Detail Tab

Each end of an association is called a role. You can define its multiplicity, persistence, ordering and changeability. You can also define its implementation.

| Property  | Description   |
|-----------|---|
| Role name | Name of the function of the class as viewed by the opposite class |

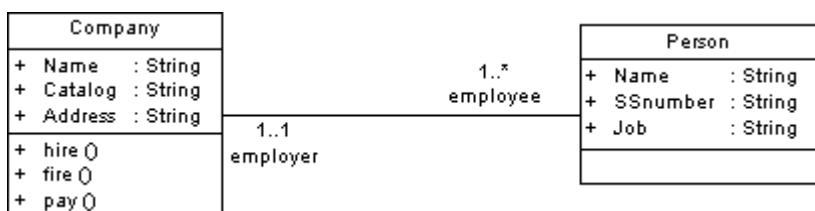
| Property      | Description  |
|---------------|--|
| Visibility    | <p>Specifies the visibility of the association role, how it is seen outside its enclosing namespace. When the role of an association is visible to another object, it may influence the structure or behavior of the object, or similarly, the other object can affect the properties of the association. You can choose between:</p> <ul style="list-style-type: none"> <li>• <b>Private</b> – only to the object itself</li> <li>• <b>Protected</b> – only to the object and its inherited objects</li> <li>• <b>Package</b> – to all objects contained within the same package</li> <li>• <b>Public</b> – to all objects (option by default)</li> </ul>   |
| Multiplicity  | <p>The allowable cardinalities of a role are called multiplicity. Multiplicity indicates the maximum and minimum cardinality that a role can have. You can choose between:</p> <ul style="list-style-type: none"> <li>• 0..1 – zero or one</li> <li>• 0..* – zero to unlimited</li> <li>• 1..1 – exactly one</li> <li>• 1..* – one to unlimited</li> <li>• * – none to unlimited</li> </ul> <p>An extended attribute exists for each role of an association. It allows you to choose how the association should be implemented. They are available in your current object language, from the <code>Profile\Association\ExtendedAttributes</code> category, under the <code>roleAContainer</code> and <code>roleBContainer</code> names. Such extended attributes are pertinent only for a 'many' multiplicity (represented by *), they provide a definition for collections of associations. For more information, see <i>Customizing and Extending PowerDesigner &gt; Object, Process, and XML Language Definition Files</i>.</p> |
| Array size    | Specifies an accurate array size when the multiplicity is greater than 1.  |
| Changeability | <p>Specifies if the set of links related to an object can be modified once the object has been initialized. You can choose between:</p> <ul style="list-style-type: none"> <li>• <b>Changeable</b> – Associations may be added, removed, and changed freely</li> <li>• <b>Read-only</b> – You are not allowed to modify the association</li> <li>• <b>Frozen</b> – Constant association</li> <li>• <b>Add-only</b> – New associations may be added from a class on the opposite end of the association</li> </ul>  |
| Ordering      | <p>The association is included in the ordering which sorts the list of associations by their order of creation. You can choose between:</p> <ul style="list-style-type: none"> <li>• <b>Sorted</b> – The set of objects at the end of an association is arranged according to the way they are defined in the model</li> <li>• <b>Ordered</b> – The set of objects at the end of an association is arranged in a specific order</li> <li>• <b>Unordered</b> – The end of an association is neither sorted nor ordered</li> </ul>   |
| Initial value | Specifies an instruction for initializing migrated attributes, for example 'new client ()'.  |

| Property             | Description   |
|----------------------|---|
| Navigable            | Specifies that information can be transmitted between the two objects linked by the relationship.   |
| Persistent           | Specifies that the instance of the association is preserved after the process that created this instance terminates.                          |
| Volatile             | Specifies that the corresponding migrated attributes are not members of the class, which is only defined by the getter and setter operations. |
| Container type       | Specifies a container collection for migrated attributes of complex types.  |
| Implementation class | Specifies the container implementation (see <a href="#">Association Implementation [page 88]</a> ).   |
| Migrated attribute   | Specifies the name of the migrated association role.  |

### 1.3.12.3 Association Implementation

Associations describe structural relationships between classes that become links between the instances of these classes. These links represent inter-object navigation, which permits one instance to retrieve another instance through the navigable link.

When an association end is navigable, you can retrieve the instance of the class it is linked to, this instance being displayed as a migrated attribute in the current instance of a class. The role name of this end can be used to clarify the structure used to represent the link. For example, for the association between class Company and class Person, navigation is possible in both directions to allow Company to retrieve a list of employees, and each employee to retrieve his company.



PowerDesigner supports different ways for implementing associations in each object language.

By default, migrated attributes use the class they come from as type. When the association multiplicity is greater than one, the type is usually an array of the class, displayed with [] signs. In our example, attribute employee in class Company is of type Person and has an array of values. When you instantiate class Company, you will have a list of employees to store for each company.

```

public class Company
{
    public String Name;
    public String Catalog;
    public String Address;

    public Person[] employee;
}
  
```

Your target language may provide other ways to implement migrated attributes. You can choose an implementation in the association property sheet *Detail* tab.

A container is a collection of objects that stores elements, and which provides more methods for accessing elements (test element existence, insert element in collection, and so on) and managing memory allocation dynamically. You can select a container type from the *Container Type* list, and this type will be used by migrated attributes and provides getter and setter functions used to define the implementation of the association, which are visible in the *Code Preview* tab, but do not appear in the list of operations (when you migrate roles to attributes (see [Migrating Association Roles \[page 91\]](#)) the generated code is identical).

Depending on the language and the libraries you are using, the container type may be associated with an implementation class. In this case, the container type is used as an interface for declaring the collection features, and the implementation class develops this collection. For example, if you select the container type `java.util.Set`, you should know that this collection contains no duplicate elements. You can then select an implementation class among the following: `HashSet`, `LinkedHashSet`, or `TreeSet`. For more information on container types and implementation classes, see the corresponding language documentation.

Documentation comment tags are used to manage generation and reverse-engineering:

- `pdRoleInfo` - to retrieve the classifier name, container type, implementation class, multiplicity and type of the association.
- `pdGenerated` - to flag automatically generated functions linked to association implementation, which should not be reverse-engineered.

These examples, show how the tags are used in various languages:

- Java - The javadoc tag syntax (`/**@tag value*/`) is used:

```
/**@pdRoleInfo name=Person coll=java.util.Collection impl=java.util.LinkedList
mult=1..* */
public java.util.Collection employee;
/** @pdGenerated default getter */
public java.util.Collection getEmployee()
{
    if (employee == null)
        employee = new java.util.HashSet();
    return employee;
}
...
```

- C# - The documentation tag syntax (`///) is used:`

```
///
public java.util.Collection employee;
///<pdGenerated> default getter </pdGenerated>
...
```

- VB.NET - The documentation tag "`<tag value />` is used:

```
"<pdRoleInfo name='Person' coll='System.CollectionsArrayList'
impl='java.util.LinkedList' mult='1..*' type='composition'/>
public java.util.Collection employee;
"<pdGenerated> default getter </pdGenerated>
...
```

## ⚠ Caution

Do not modify these tags in order to preserve round-trip engineering.

The default implementation is defined in the object language resource file under the `Profile\Association` category, and uses templates to define the migrated attribute generated syntax, the operations to generate, and other association details. For example, template `roleAMigratedAttribute` allows you to recover the visibility and initial value of the association role. You modify implementation details in the resource editor (see *Customizing and Extending PowerDesigner > Customizing Generation with GTL* ).

### 1.3.12.4 Creating an Association Class

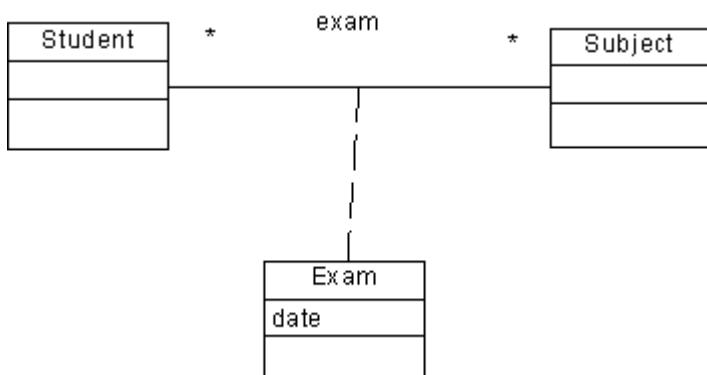
You can add properties to an association between classes or interfaces by creating an association class. It is used to further define the properties of an association by adding attributes and operations to the association.

#### Context

An association class is an association that has class properties, or a class that has association properties. In the diagram, the symbol of an association class is a connection between an association and a class. Association classes must be in the same package as the parent association; you cannot use the shortcut of a class to create an association class.

The class used to create an association class cannot be reused for another association class. However, you can create other types of links to and from this class.

In the following example, the classes `Student` and `Subject` are related by an association `exam`. However, this association does not specify the date of the exam. You can create an association class called `Exam` that will indicate additional information concerning the association.



#### Procedure

1. Right-click the association and select `Add Association Class` from the contextual menu.
2. Double-click the association to open its property sheet, and click the `Create` button to the right of the `Association class` listbox.

## Results

A dashed link is automatically added between the class and the association.

### 1.3.12.5 Migrating Association Roles

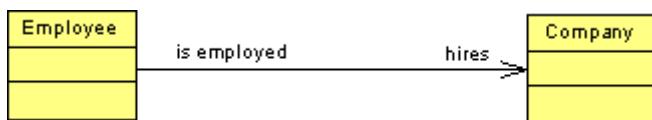
You can migrate association roles in a class diagram and create attributes before generation to permit data type customization and changing the attribute order in the list of attributes. The last feature is especially important in XML.

To migrate roles, right-click the association and select **Migrate > Migrate Navigable Roles**. An attribute is created in the classifier opposite the navigable role of the association, with the following properties:

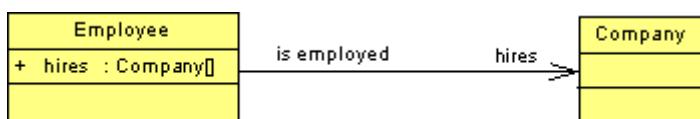
- **Name/Code** - association role if already set, or association name.
- **Data type** - code of the classifier linked by the association.
- **Multiplicity/Visibility** - role multiplicity, visibility.
- **Migrated from** - association name.

This migrated attribute remains synchronized with the objects from which it is derived and is updated if you make changes to them, unless you make changes to its properties. It is deleted if the association is deleted.

In this example, the **hires** role is navigable:



After migration, a new attribute is created in the Employee class:



## 1.3.12.6 Rebuilding Data Type Links

If a classifier data type is not linked to its original classifier, you can use the Rebuild Data Type Links feature to restore the link. This feature looks for all classifiers of the current model and links them, if needed, to the original classifier.

### Context

The Rebuild Data Type Links scans the following data types:

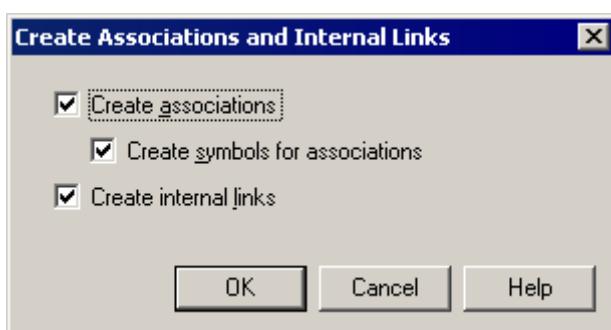
- Attribute data type: an association is created and the attribute is flagged as Migrated Attribute
- Parameter data type: an association is created and links the original classifier
- Operation return type: an association is created and links the original classifier

In some cases, for C++ in particular, this feature is very useful to keep the synchronization of the link even if the data type changes, so that it keeps referencing the original class.

The Rebuild Data Type Links contains the following options:

### Procedure

1. Select  *Tools*  to open the Create Associations and Internal Links window.



2. Set the following options as needed:

| Option              | Description  |
|---------------------|--|
| Create associations | Looks for attributes whose data type matches a classifier and links the attributes to the newly created association as migrated attributes |

| Option                          | Description  |
|---------------------------------|--|
| Create symbols for associations | Creates a symbol of the new association  |
| Create internal links           | Creates a link between the return type or parameter data type and the classifier it references |

3. Click **OK**.

### 1.3.12.7 Linking an Association to an Instance Link

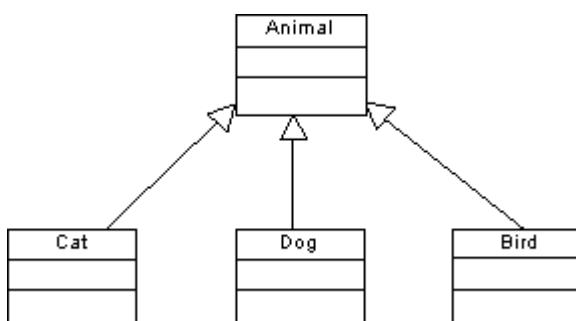
You can drag an association node from the Browser and drop it into the communication or object diagrams. This automatically creates two objects, and an instance link between them. Both objects are instances of the classes or interfaces, and the instance link is an instance of the association.

### 1.3.13 Generalizations (OOM)

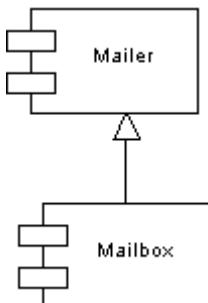
A generalization is a relationship between a general element (the parent) and a more specific element (the child). The more specific element is fully consistent with the general element and contains additional information. You create a generalization relationship when several objects have common behaviors.

A generalization can be created in the following diagrams:

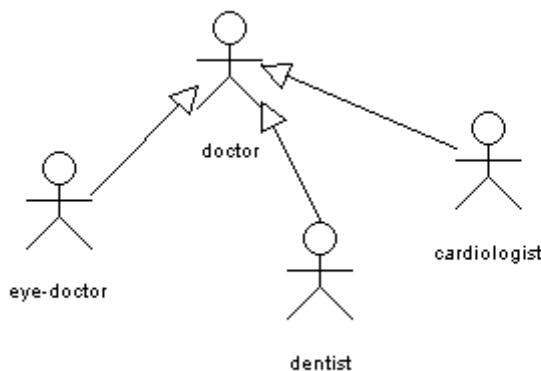
- Class or Composite Structure Diagram - between classes or interfaces. For example, an animal is a more general concept than a cat, a dog or a bird, which are more specific concepts. Animal is a super class. Cat, Dog and Bird are sub-classes of the super class.



- Component Diagram - between two components. For example:



- Use Case Diagram - between actors or use cases. For example two or more actors may have similarities, and communicate with the same set of use cases in the same way. Child actors inherit the roles and relationships to use cases held by the parent actor and include the attributes and operations of their parent. For example:



### 1.3.13.1 Creating a Generalization

You can create a generalization from the Toolbox, Browser, or *Model* menu.

- Use the *Generalization* tool in the Toolbox.
- Select **Model > Generalizations** to access the List of Generalizations, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Generalization**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.3.13.2 Generalization Properties

To view or edit a generalization's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property                       | Description  |
|--------------------------------|--|
| Name/Code/Comment              | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype                     | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Parent                         | Specifies the parent object. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.  |
| Child                          | Specifies the child object. Click the Properties tool to the right of this box to view the properties of the currently selected object.  |
| Visibility                     | Specifies the visibility of the object, how it is seen outside its enclosing namespace. You can choose between: <ul style="list-style-type: none"><li>• <b>Private</b> – only to the generalization itself</li><li>• <b>Protected</b> – only to the generalization and its inherited objects</li><li>• <b>Package</b> – to all objects contained within the same package</li><li>• <b>Public</b> – to all objects (option by default)</li></ul>  |
| Generate parent class as table | Selects the <i>Generate table</i> persistence option in the <i>Detail</i> tab of the parent class property sheet. If this option is not selected, the <i>Migrate columns</i> persistence option of the parent class is selected.   |
| Generate child class as table  | Selects the <i>Generate table</i> persistence option in the <i>Detail</i> tab of the child class property sheet. If this option is not selected, the <i>Migrate columns</i> persistence option of the child class is selected.   |
| Specifying Attribute           | Specifies a persistent attribute (with a stereotype of <<specifying>>) in the parent table. Click the <i>New</i> tool to create a new attribute. This attribute will only be generated if the child table is not.  |
| Keywords                       | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

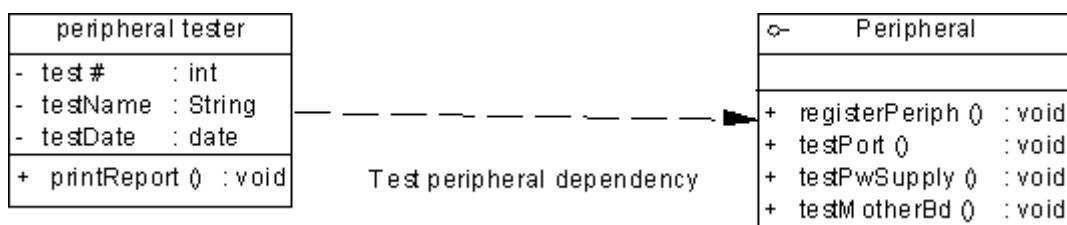
If the generalization is created in a use case diagram, you cannot change the type of objects linked by the generalization. For example, you cannot attach the dependency coming from a use case to a class, or an interface.

### 1.3.14 Dependencies (OOM)

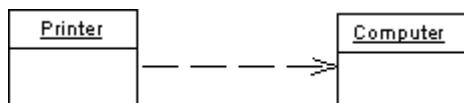
A dependency is a semantic relationship between two objects, in which a change to one object (the influent object) may affect the semantics of the other object (the dependent object). The dependency relationship indicates that one object in a diagram uses the services or facilities of another object. You can also define dependencies between a package and a modeling element.

A dependency can be created in the following diagrams:

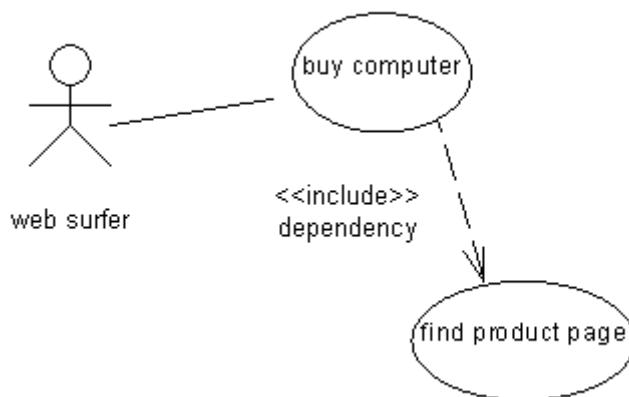
- Class or Composite Structure Diagram - between classes, interfaces, and classes and interfaces. For example:



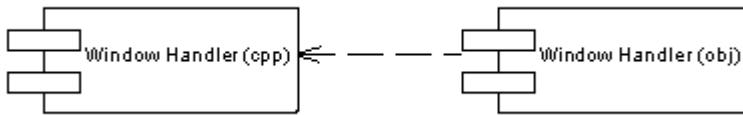
- Object Diagram - between objects. For example:



- Use Case Diagram - between actors, use cases, or actors and use cases. For example, buying a computer from a web site involves the activity of finding the product page within the seller's web site:

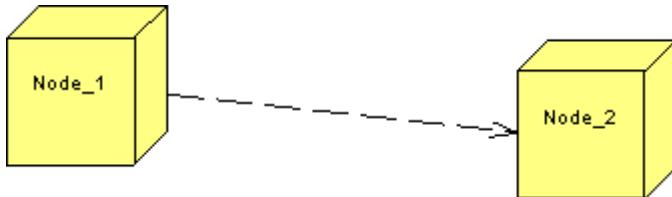


- Component Diagram - between components (you cannot create a dependency between a component and an interface):



When using a dependency, you can nest two components by using a stereotype.

- Deployment Diagram - between nodes, and component instances. For example:



### 1.3.14.1 Creating a Dependency

You can create a dependency from the Toolbox, Browser, or *Model* menu.

- Use the *Dependency* tool in the Toolbox.
- Select **Model > Dependencies** to access the List of Dependencies, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Dependency**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

### 1.3.14.2 Dependency Properties

To view or edit a dependency's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |

| Property   | Description   |
|------------|---|
| Stereotype | <p>Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.</p> <p>The following common stereotypes are provided by default:</p> <ul style="list-style-type: none"> <li>• &lt;&lt; Access &gt;&gt; - Public contents of the target package that can be accessed by the source package</li> <li>• &lt;&lt; Bind &gt;&gt; - Source object that instantiates the target template using the given actual parameters</li> <li>• &lt;&lt; Call&gt;&gt; - Source operation that invokes the target operation</li> <li>• &lt;&lt; Derive &gt;&gt; - Source object that can be computed from the target</li> <li>• &lt;&lt; Extend &gt;&gt; - (Use Case/Class) Target object extends the behavior of the source object at the given extension point</li> <li>• &lt;&lt; Friend&gt;&gt; - Source object that has special visibility towards the target</li> <li>• &lt;&lt; Import &gt;&gt; - Everything declared public in the target object becomes visible to the source object, as if it were part of the source object definition</li> <li>• &lt;&lt; Include &gt;&gt; - (Use Case/Class) Inclusion of the behavior of the first object into the behavior of the client object, under the control of the client object</li> <li>• &lt;&lt; Instantiate &gt;&gt; - Operations on the source class create instances of the target class</li> <li>• &lt;&lt; Refine &gt;&gt; - The target object has a greater level of detail than the source object</li> <li>• &lt;&lt; Trace &gt;&gt; - Historical link between the source object and the target object</li> <li>• &lt;&lt; Use &gt;&gt; - Semantics of the source object are dependent on the semantics of the public part of the target object</li> <li>• &lt;&lt; ejb-ref &gt;&gt; - (Java only) Used in Java Generation to create references to EJBs (entity beans and session beans) for generating the deployment descriptor</li> <li>• &lt;&lt; sameFile &gt;&gt; - (Java only) Used in Java Generation to generate Java classes of visibility protected or private within a file corresponding to a class of visibility public</li> </ul> |
| Influent   | Selected use case or actor influences the dependent object. Changes on the influent object affect the dependent object. Click the Properties tool to the right of the field to view the object property sheet   |
| Dependent  | Selected use case or actor depends on the influent object. Changes on the dependent object do not affect the influent object. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.  |
| Keywords   | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.   |

If the dependency is created in a use case diagram, you cannot change the objects linked by the dependency. For example, you cannot attach the dependency coming from a use case to a class or an interface.

### 1.3.15 Realizations (OOM)

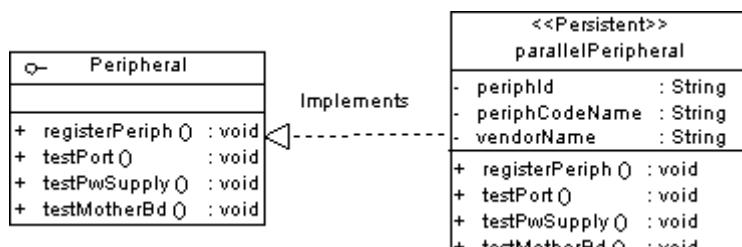
A realization is a relationship between a class or component and an interface.

A realization can be created in the following diagrams:

- Class Diagram
  - Composite Structure Diagram
  - Component Diagram

In a realization, the class implements the methods specified in the interface. The interface is called the specification element and the class is called the implementation element.

The arrowhead at one end of the realization always points towards the interface.



### 1.3.15.1 Creating a Realization

You can create a realization from the Toolbox, Browser, or *Model* menu.

- Use the *Realization* tool in the Toolbox.
  - Select *Model* *Realizations* to access the List of Realizations, and click the *Add a Row* tool.
  - Right-click the model (or a package) in the Browser, and select *New* *Realizations*

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.3.15.2 Realization Properties

To view or edit a realization's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Interface         | Name of the interface that carries out the realization. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.   |
| Class             | Name of the class for which the realization is carried out   |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## 1.3.16 Require Links (OOM)

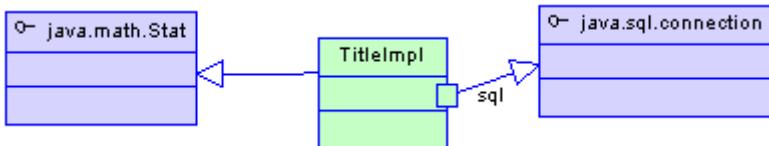
Require links connect classifiers and interfaces. A require link can connect a class, a component, or a port on the outside of one of these classifiers to an interface.

A require link can be created in the following diagrams:

- Class Diagram
- Composite Structure Diagram
- Component Diagram

### Require Links in a Class Diagram

In the example below, require links connect the class TitleImpl with the interfaces java.math.stat and java.sql.connection. Note how the require link can proceed from a port or directly from the class



### 1.3.16.1 Creating a Require Link

You can create a require link from the Toolbox, Browser, or *Model* menu.

- Use the *Require Link/Connector* tool in the Toolbox.
- Select **Model > Require Links** to access the List of Require Links, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Require Link**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

### 1.3.16.2 Require Link Properties

To view or edit a require link's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

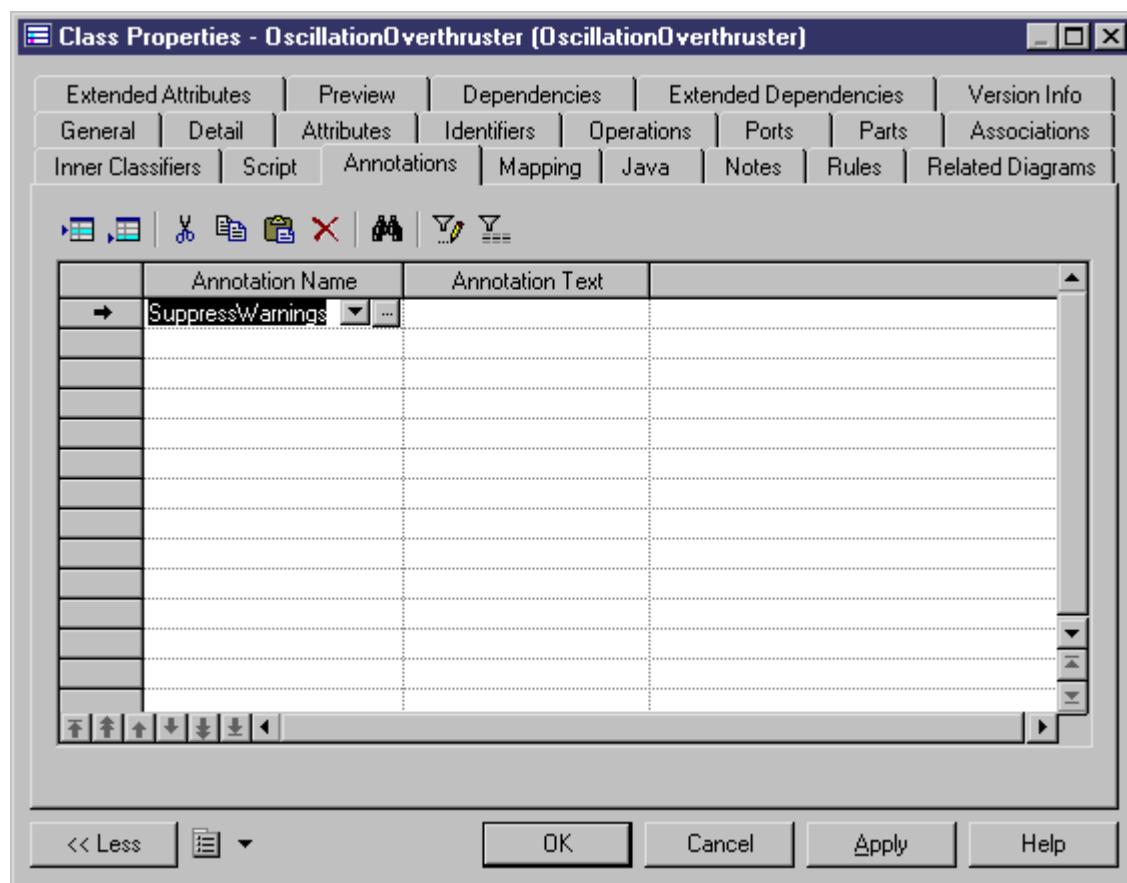
The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Interface         | Specifies the interface to be linked to. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.  |
| Client            | Specifies the class, port, or component to be linked to.   |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

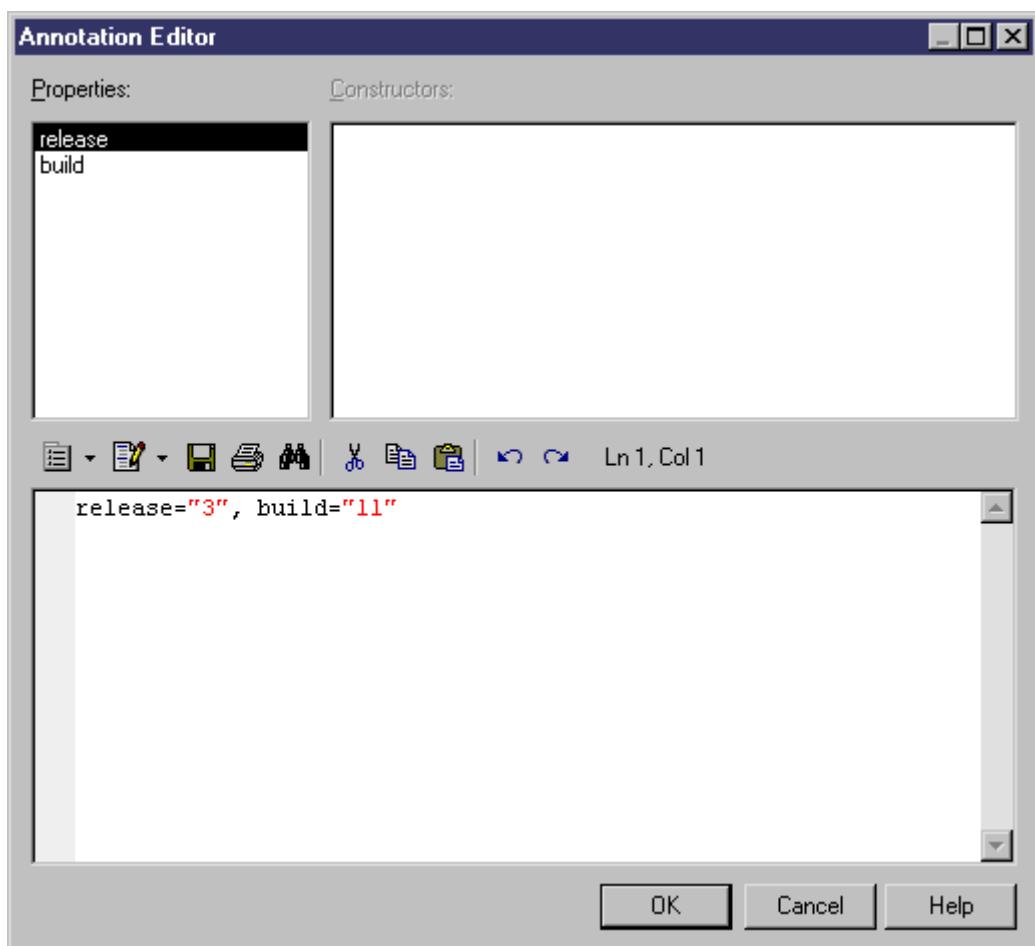
### 1.3.17 Annotations (OOM)

Java 5.0 and the .NET languages (C# 2.0 and VB 2005) provide methods for adding metadata to code, which can be accessed by post-processing tools or at run-time to vary the behavior of the system. PowerDesigner supports the Java 5.0 built-in annotations, the .NET 2.0 built-in custom attributes, and allows you to create your own. You can attach annotations to types and other model objects

To attach an annotation to a model object, open its property sheet, click the *Annotations* tab, click in the *Annotation Name* column, and select an annotation from the list:



If the annotation takes parameters, you can enter them directly in the Annotation Text column or click the ellipsis button to open the Annotation Editor. The top panes provide lists of available properties and constructors, which you can double-click to add them to the bottom, editing pane:

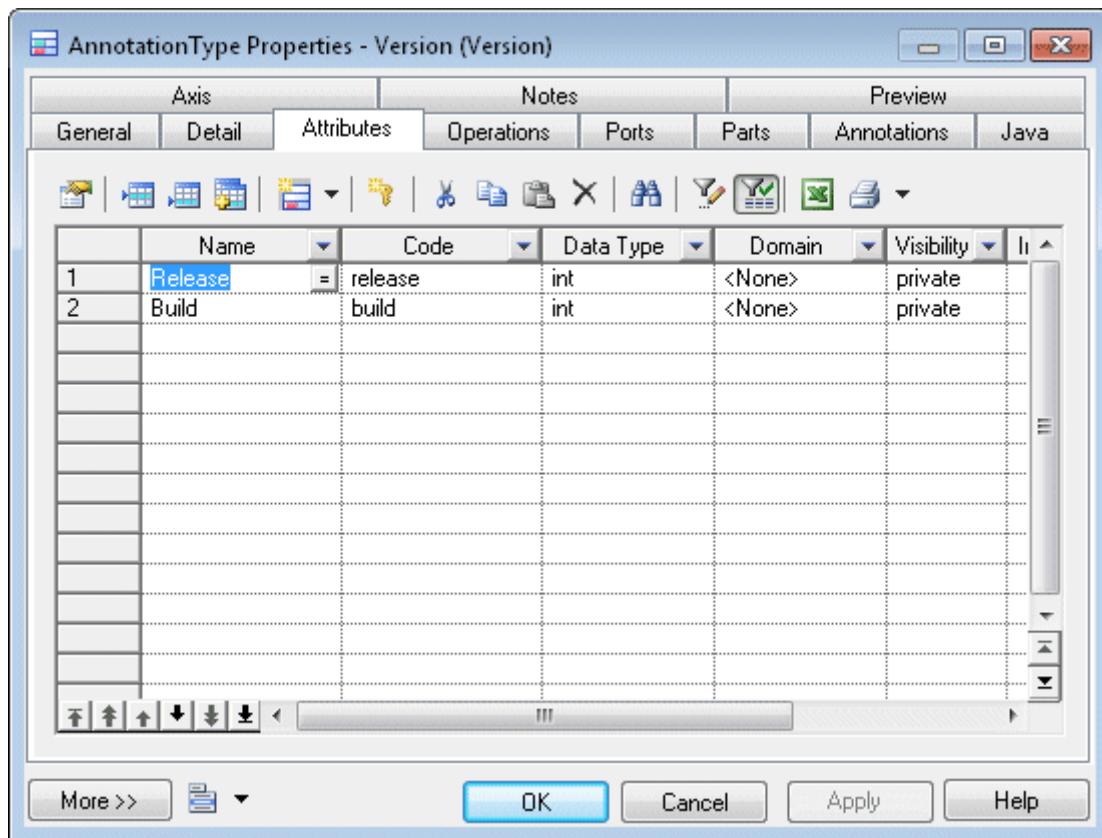


### 1.3.17.1 Creating a New Annotation Type

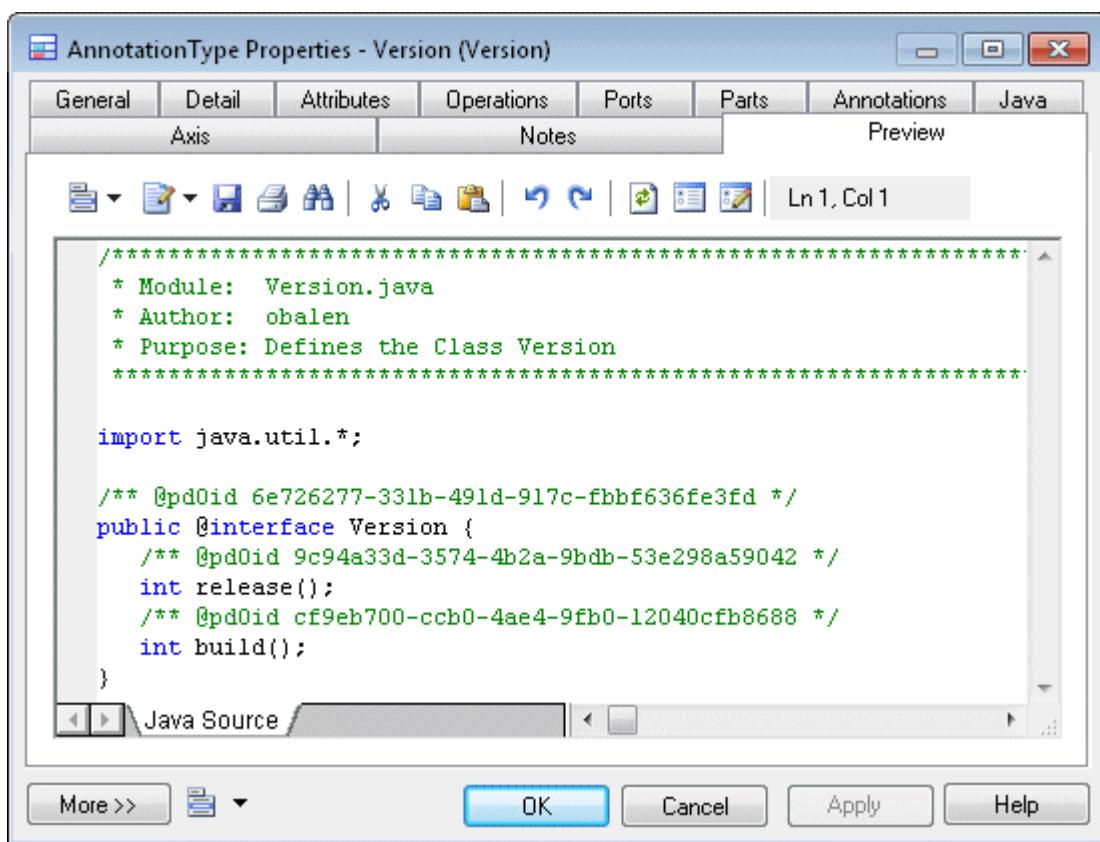
You can create new annotation types to attach to your objects.

#### Procedure

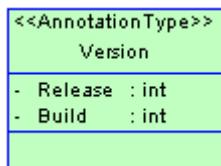
1. Create a class and then double-click it to access its property sheet.
2. On the *General* tab, in the *Stereotype* list:
  - For Java 5.0 - select *AnnotationType*
  - For .NET 2.0 - select *AttributeType*
3. Click the *Attributes* tab, and add an attribute for each parameter accepted by the annotation type.



4. [optional] Click the *Preview* tab to review the code to be generated for the annotation type:



5. Click **OK** to return to the diagram. The annotation type will be represented as follows:



### 1.3.18 Instance Links (OOM)

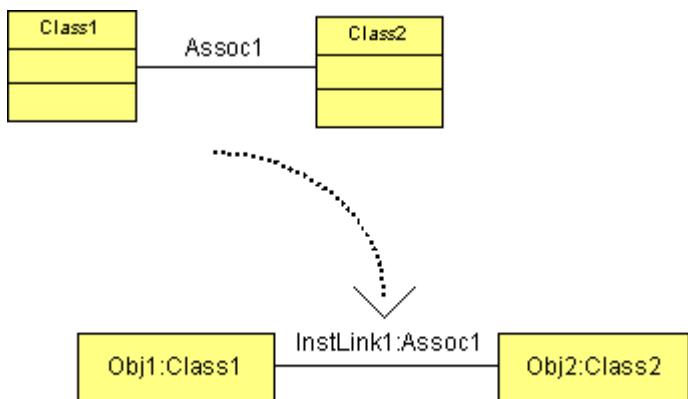
An instance link represents a connection between two objects. It is drawn as a solid line between two objects.

An instance link can be created in the following diagrams:

- Communication Diagram
- Object Diagram

## Instance Links in an Object Diagram

Instance links have a strong relationship with associations of the class diagram: associations between classes, or associations between a class and an interface can become instance links (instances of associations) between objects in the object diagram. Moreover, the instance link symbol in the object diagram is similar to the association symbol in the class diagram, except that the instance link symbol has no cardinalities.

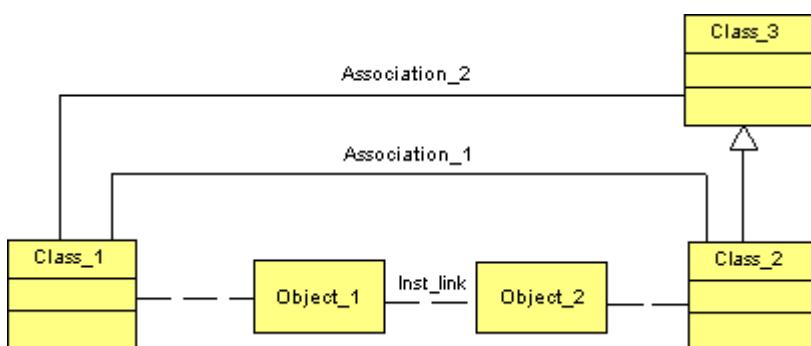


The roles of the instance link are duplicated from the roles of the association. An instance link can therefore be an aggregation or a composition, exactly like an association of the class diagram. If it is the case, the composition or aggregation symbol is displayed on the instance link symbol. The roles of the association are also displayed on the instance link symbol provided you select the Association Role Names display preference in the Instance Link category.

## Example

The following figure shows Object\_1 as instance of Class\_1, and Object\_2 as instance of Class\_2. They are linked by an instance link. It shows Class\_1 and Class\_2 linked by an association. Moreover, since Class\_2 is associated with Class\_1 and also inherits from Class\_3, there is an association between Class\_1 and Class\_3.

The instance link between Object\_1 and Object\_2 in the figure can represent Association\_1 or Association\_2.



You can also use shortcuts of associations, however you can only use it if the model to which the shortcut refers is open in the workspace.

## Instance Links Behavior

The following rules apply to instance links:

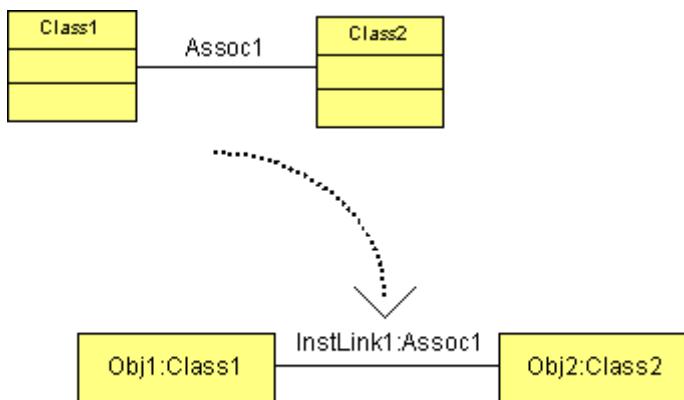
- When an association between classes becomes an instance link, both classes linked by the association, and both classes of the objects linked by the instance link must match (or the class of the object must inherit from the parent classes linked by the association). This is also valid for an association between a class and an interface
- Two instance links can be defined between the same source and destination objects (parallel instance links). If you merge two models, the Merge Model feature differentiates parallel instance links according to their class diagram associations
- You can use reflexive instance links (same source and destination object)

## Instance Links in a Communication Diagram

An instance link represents a connection between objects, it highlights the communication between objects, hence the name 'communication diagram'. It is drawn as a solid line between:

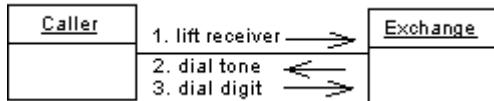
- Two objects
- An object and an actor (and vice versa)

An instance link can be an instance of an association between classes, or an association between a class and an interface.



The role of the instance link comes from the association. The name of an instance link comprises the names of both objects at the extremities, plus the name of the association.

The symbol of the instance link may contain several message symbols attached to it.



Instance links hold an ordered list of messages. The sequence numbers specify the order in which messages are exchanged between objects. For more information, see [Messages \(OOM\) \[page 131\]](#).

## Instance Links Behavior

The following rules apply to instance links:

- You can use a recursive instance link with an object (same source and destination object)
- Two instance links can be defined between the same source and destination objects (parallel instance links)
- When you delete an instance link, its messages are also deleted if no sequence diagram already uses them
- When an association between classes turns into an instance link, both classes linked by the association, and both classes of the objects linked by the instance link must match (or the class of the object must inherit from the parent classes linked by the association). This is also valid for an association between a class and an interface
- If you change one end of an association, the instance link that comes from the association is detached
- When you copy and paste, or move an instance link, its messages are automatically copied at the same time
- When the extremities of the message change, the message is detached from the instance link
- If you use the Show Symbols feature to display an instance link symbol, all the messages attached to the instance link are displayed

### 1.3.18.1 Creating an Instance Link

You can create an instance link from the Toolbox, Browser, or *Model* menu.

- Use the *Instance Link* tool in the Toolbox.
- Select **Model > Instance Links** to access the List of Instance Links, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Instance Link**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## Drag and Drop Associations in an Object Diagram

You can drag an association from the Browser and drop it into an object diagram. This creates an instance link and two objects, instances of these classes or interfaces.

## 1.3.18.2 Instance Link Properties

To view or edit an instance link's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description   |
|-------------------|---|
| Name/Code/Comment | Identify the object. The name and code are read-only. You can optionally add a comment to provide more detailed information about the object.   |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.  |
| Object A          | Name of the object at one end of the instance link. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.  |
| Object B          | Name of the object at the other end of the instance link. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.  |
| Association       | Association between classes (or association between a class and an interface) that the instance link uses to communicate between objects of these classes. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object. |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.   |

## 1.3.19 Domains (OOM)

Domains define the set of values for which an attribute is valid. They are used to enforce consistent handling of data across the system. Applying domains to attributes makes it easier to standardize data characteristics for attributes in different classes.

A domain can be created in the following diagrams:

- Class Diagram

In an OOM, you can associate the following properties with a domain:

- Data type
- Check parameters
- Business rules

## 1.3.19.1 Creating a Domain

You can create a domain from the Browser or *Model* menu.

- Select *Model* to access the List of Domains, and click the *Add a Row* tool.
- Right-click the model or package in the Browser, and select *New* .

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.3.19.2 Domain Properties

To view or edit a domain's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Data type         | Form of the data corresponding to the domain ; numeric, alphanumeric, Boolean, or others   |
| Multiplicity      | Specification of the range of allowable number of values attributes using this domain may hold. The multiplicity of a domain is useful when working with a multiple attribute for example. The multiplicity is part of the data type and both multiplicity and data type may come from the domain. You can choose between: <ul style="list-style-type: none"><li>• 0..1 – zero or one</li><li>• 0..* – zero to unlimited</li><li>• 1..1 – exactly one</li><li>• 1..* – one to unlimited</li><li>• * – none to unlimited</li></ul>                |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## Detail Tab

The *Detail* tab contains a Persistent groupbox whose purpose is to improve the generation of code and data types during generation of a CDM or a PDM from an object-oriented model, and which contains the following properties:

| Property   | Description  |
|------------|--|
| Persistent | Groupbox for valid generation of CDM or PDM persistent models. Defines a model as persistent (see <a href="#">Managing Object Persistence During Generation of Data Models [page 245]</a> ). |
| Data Type  | Specifies a persistent data type used in the generation of a persistent model, either a CDM or a PDM. The persistent data type is defined from default PowerDesigner conceptual data types   |
| Length     | Maximum number of characters of the persistent data type.  |
| Precision  | Number of places after the decimal point, for persistent data type values that can take a decimal point.   |

The following tabs are also available:

- Standard Checks - contains checks which control the values permitted for the domain (see [Setting Data Profiling Constraints \[page 73\]](#))
- Additional Checks - allows you to specify additional constraints (not defined by standard check parameters) for the domain.
- Rules - lists the business rules associated with the domain (see *Core Features Guide > Modeling with PowerDesigner > Objects > Business Rules*).

The tables below give details of the available data types:

## Numeric Data Types

| Data Type     | Content                               | Length | Mandatory Precision |
|---------------|---------------------------------------|--------|---------------------|
| Integer       | 32-bit integer                        | —      | —                   |
| Short Integer | 16-bit integer                        | —      | —                   |
| Long Integer  | 32-bit integer                        | —      | —                   |
| Byte          | 256 values                            | —      | —                   |
| Number        | Numbers with a fixed decimal point    | Fixed  |                     |
| Decimal       | Numbers with a fixed decimal point    | Fixed  |                     |
| Float         | 32-bit floating point numbers         | Fixed  | —                   |
| Short Float   | Less than 32-bit point decimal number | —      | —                   |

| Data Type  | Content                                       | Length | Mandatory Precision |
|------------|---|--------|---------------------|
| Long Float | 64-bit floating point numbers                 | —      | —                   |
| Money      | Numbers with a fixed decimal point            | Fixed  |                     |
| Serial     | Automatically incremented numbers             | Fixed  | —                   |
| Boolean    | Two opposing values (true/false; yes/no; 1/0) | —      | —                   |

## Character Data Types

| Data Type           | Content                     | Length  |
|---------------------|-----------------------------|---------|
| Characters          | Character strings           | Fixed   |
| Variable Characters | Character strings           | Maximum |
| Long Characters     | Character strings           | Maximum |
| Long Var Characters | Character strings           | Maximum |
| Text                | Character strings           | Maximum |
| Multibyte           | Multibyte character strings | Fixed   |
| Variable Multibyte  | Multibyte character strings | Maximum |

## Time Data Types

| Data Type   | Content                  |
|-------------|--------------------------|
| Date        | Day, month, year         |
| Time        | Hour, minute, and second |
| Date & Time | Date and time            |
| Timestamp   | System date and time     |

## Other Data Types

| Data Type   | Content                       | Length  |
|-------------|-------------------------------|---------|
| Binary      | Binary strings                | Maximum |
| Long Binary | Binary strings                | Maximum |
| Bitmap      | Images in bitmap format (BMP) | Maximum |
| Image       | Images                        | Maximum |
| OLE         | OLE links                     | Maximum |
| Other       | User-defined data type        | —       |
| Undefined   | Not yet defined data type     | —       |

### 1.3.19.3 Updating Attributes Using a Domain in an OOM

When you modify a domain, you can choose to automatically update the following properties for attributes using the domain:

#### Context

- Data type
- Check parameters
- Business rules

#### Procedure

1. Select   to display the list of domains.
2. Click a domain from the list, and then click the *Properties* tool to display its property sheet.

##### Note

You also access a domain property sheet by double-clicking the appropriate domain in the Browser.

3. Type or select domain properties as required in the tabbed pages and Click *OK*. If the domain is used by one or more attributes, an update confirmation box asks you if you want to update Data type and Check parameters for the attributes using the domain.

4. Select the properties you want to update for all attributes using the domain and click **Yes**.

## 1.4 Dynamic Diagrams

The diagrams in this chapter allow you to model the dynamic behavior of your system, how its objects interact at run-time. PowerDesigner provides four types of diagrams for modeling your system in this way:

- A communication diagram represents a particular use case or system functionality in terms of interactions between objects, while focusing on the object structure. For more information, see [Communication Diagrams \[page 114\]](#).
- A sequence diagram represents a particular use case or system functionality in terms of interactions between objects, while focusing on the chronological order of the messages sent. For more information, see [Sequence Diagrams \[page 116\]](#).
- A activity diagram represents a particular use case or system functionality in terms of the actions or activities performed and the transitions triggered by the completion of these actions. It also allows you to represent conditional branches. For more information, see [Activity Diagrams \[page 120\]](#).
- A statechart diagram represents a particular use case or system functionality in terms of the states that a classifier passes through and the transitions between them. For more information, see [Statechart Diagrams \[page 123\]](#).
- An interaction overview diagram provides a high level view of the interactions that occur in your system. For more information, see [Interaction Overview Diagrams \[page 126\]](#).

### 1.4.1 Communication Diagrams

A communication diagram is a UML diagram that provides a graphical view of the interactions between objects for a use case scenario, the execution of an operation, or an interaction between classes, with an emphasis on the system structure.

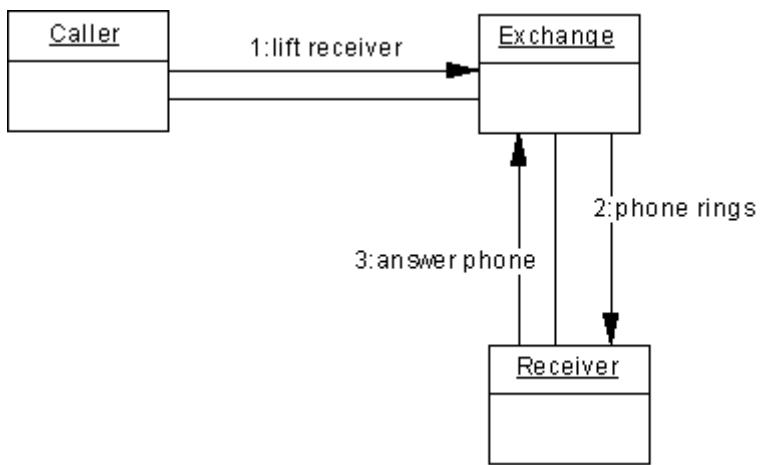
#### i Note

To create a communication diagram in an existing OOM, right-click the model in the Browser and select ► **New** ► **Communication Diagram** ». To create a new model, select ► **File** ► **New Model** », choose Object Oriented Model as the model type and **Communication Diagram** as the first diagram, and then click **OK**. To create a communication diagram that reuses the objects and messages from an existing sequence diagram (and creates instance links between the objects that communicate using messages, updating any message sequence numbers based upon the relative position of messages on the timeline), right-click in the sequence diagram and select **Create Default Communication Diagram**, or select ► **Tools** ► **Create Default Communication Diagram** ». Note that the two diagrams do not remain synchronized – changes made in one diagram will not be reflected in the other.

You can use one or more communication diagrams to enact a use case or to identify all the possibilities of a complex behavior.

A communication diagram conveys the same kind of information as a sequence diagram, except that it concentrates on the object structure in place of the chronology of messages passing between them.

A communication diagram shows actors, objects (instances of classes) and their communication links (called instance links), as well as messages sent between them. The messages are defined on instance links that correspond to a communication link between two interacting objects. The order in which messages are exchanged is represented by sequence numbers.



## Analyzing a Use Case

A communication diagram can be used to refine a use case behavior or description. This approach is useful during requirement analysis because it may help identify classes and associations that did not emerge at the beginning.

You can formalize the association between the use case and the communication diagram by adding the diagram to the Related Diagrams tab of the property sheet of the use case.

It is often necessary to create several diagrams to describe all the possible scenarios of a use case. In this situation, it can be helpful to use the communication diagrams to discover all the pertinent objects before trying to identify the classes that will instantiate them. After having identified the classes, you can then deduce the associations between them from the instance links between the objects.

The major difficulty with this approach consists in identifying the correct objects to transcribe the action steps of the use case. An extension to UML, "Robustness Analysis" can make this process easier. This method recommends separating objects into three types:

- Boundary objects are used by actors when communicating with the system; they can be windows, screens, dialog boxes or menus
- Entity objects represent stored data like a database, database tables, or any kind of transient object such as a search result
- Control objects are used to control boundary and entity objects, and represent the transfer of information

PowerDesigner supports the Robustness Analysis extension through an extension file (see *Customizing and Extending PowerDesigner > Extension Files > Example: Creating Robustness Diagram Extensions*).

## Analyzing a Class Diagram

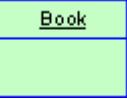
Building a communication diagram can also be the opportunity to test a static model at the conception level; it may represent a scenario in which classes from the class diagram are instantiated to create the objects necessary to run the scenario.

It complements the class diagram that represents the static structure of the system by specifying the behavior of classes, interfaces, and the possible use of their operations.

You can create the necessary objects and instance links automatically by selecting the relevant classes and associations in a class diagram, and then pressing [Ctrl] + [Shift] while dragging and dropping them into a an empty communication diagram. Then you have simply to add the necessary messages.

### 1.4.1.1 Communication Diagram Objects

PowerDesigner supports all the objects necessary to build communication diagrams.

| Object        | Tool  | Symbol   | Description  |
|---------------|---|--|--|
| Actor         |  | <br>Client      | An external person, process or something interacting with a system, sub-system or class. See <a href="#">Actors (OOM) [page 24]</a> .  |
| Object        |  | <br><u>Book</u> | Instance of a class. See <a href="#">Objects (OOM) [page 62]</a> .   |
| Instance link |  |                 | Communication link between two objects. See <a href="#">Instance Links (OOM) [page 105]</a> .  |
| Message       |  |                 | Interaction that conveys information with the expectation that action will ensue. It creates an instance link by default when no one exists. See <a href="#">Messages (OOM) [page 131]</a> . |

### 1.4.2 Sequence Diagrams

A sequence diagram is a UML diagram that provides a graphical view of the chronology of the exchange of messages between objects and actors for a use case, the execution of an operation, or an interaction between classes, with an emphasis on their chronology.

## Note

To create a sequence diagram in an existing OOM, right-click the model in the Browser and select **New > Sequence Diagram**. To create a new model, select **File > New Model**, choose Object Oriented Model as the model type and *Sequence Diagram* as the first diagram, and then click **OK**. To create a sequence diagram that reuses the objects and messages from an existing communication diagram, right-click in the communication diagram and select *Create Default Sequence Diagram*, or select **Tools > Create Default Sequence Diagram**. Note that the two diagrams do not remain synchronized – changes made in one diagram will not be reflected in the other.

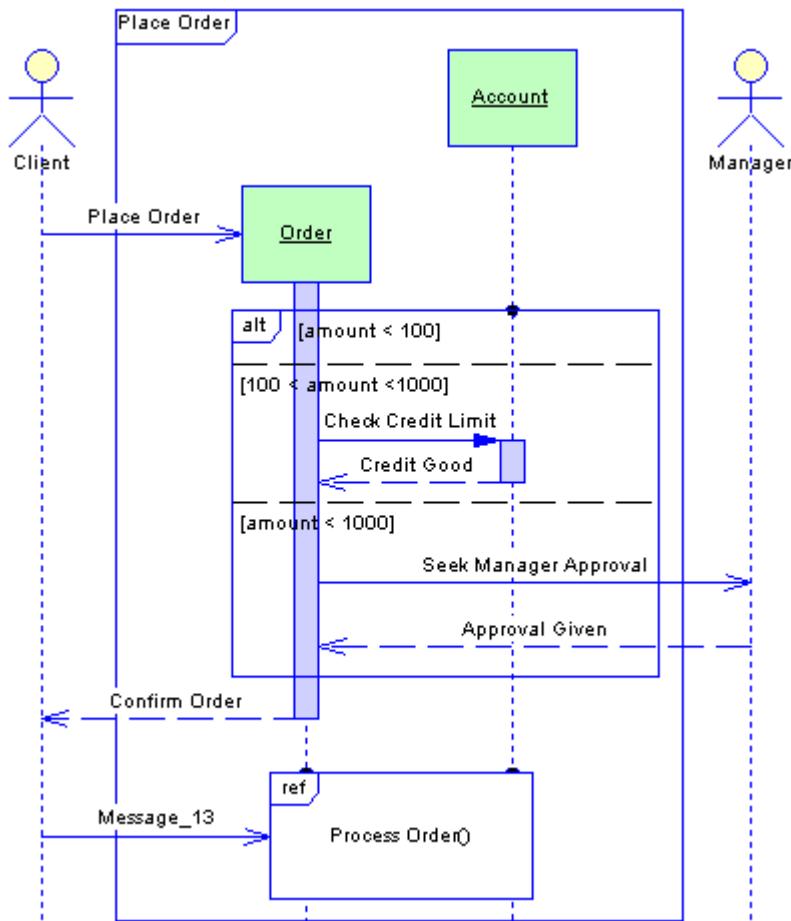
You can use one or more sequence diagrams to enact a use case or to identify all the possibilities of a complex behavior.

A sequence diagrams shows actors, objects (instances of classes) and the messages sent between them. It conveys the same kind of information as a communication diagram, except that it concentrates on the chronology of messages passing between the objects in place of their structure.

By default, PowerDesigner provides an "interaction frame", which surrounds the objects in the diagram and acts as the exterior of the system (or part thereof) being modeled. Messages can originate from or be sent to any point on the frame, and these gates can be used in place of actor objects (see [Messages and Gates \[page 142\]](#)). You can suppress the frame by clicking **Tools > Display Preferences**, selecting the *Interaction Frame* category and deselecting the *Interaction Symbol* option. For detailed information about using display preferences, see [Core Features Guide > Modeling with PowerDesigner > Diagrams, Matrices, and Symbols > Display Preferences](#).

One of the major advantages of a sequence diagram over a communication diagram is that you can reference common interactions and easily specify alternative or parallel scenarios using interaction fragments. Thus, you can describe in a single sequence diagram a number of related interactions that would require multiple communication diagrams.

In the following example, the Client actor places an order. The Place Order message creates an Order object. An interaction fragment handles various possibilities for checking the order. The Account object and Manager actor may interact with the order depending on its size. Once the Confirm Order message is sent, the Process Order interaction is initiated. This interaction is stored in another sequence diagram, and is represented here by an interaction reference:



## Analyzing a Use Case

A sequence diagram can be used to refine a use case behavior or description. This approach is useful during requirement analysis because it may help identify classes and associations that did not emerge at the beginning.

You can formalize the association between the use case and the sequence diagram by adding the diagram to the Related Diagrams tab of the property sheet of the use case.

It is often necessary to create several diagrams to describe all the possible scenarios of a use case. In this situation, it can be helpful to use the sequence diagrams to discover all the pertinent objects before trying to identify the classes that will instantiate them. After having identified the classes, you can then deduce the associations between them from the messages passing between the objects.

## Analyzing a Class Diagram

Building a sequence diagram can also be the opportunity to test a static model at the conception level; it may represent a scenario in which classes from the class diagram are instantiated to create the objects necessary to run the scenario.

It complements the class diagram that represents the static structure of the system by specifying the behavior of classes, interfaces, and the possible use of their operations.

A sequence diagram allows you to analyze class operations more closely than a communication diagram. You can create an operation in the class of an object that receives a message through the property sheet of the message. This can also be done in a communication diagram, but there is more space in a sequence diagram to display detailed information (arguments, return value, etc) about the operation.

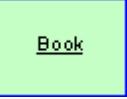
### Note

The Auto-layout, Align and Group Symbols features are not available in the sequence diagram.

When you use the Merge Models feature to merge sequence diagrams, the symbols of all elements in the sequence diagram are merged without comparison. You can either accept all modifications on all symbols or no modifications at all.

### 1.4.2.1 Sequence Diagram Objects

PowerDesigner supports all the objects necessary to build sequence diagrams.

| Object     | Tool  | Symbol  | Description  |
|------------|---|---|--|
| Actor      |  | <br>Client | An external person, process or something interacting with a system, sub-system or class. See <a href="#">Actors (OOM) [page 24]</a> .      |
| Object     |  | <br>Book   | Instance of a class. See <a href="#">Objects (OOM) [page 62]</a> .   |
| Activation |  |            | Execution of a procedure, including the time it waits for nested procedures to execute. See <a href="#">Activations (OOM) [page 138]</a> . |

| Object                 | Tool | Symbol | Description  |
|------------------------|------|--------|--|
| Interaction Reference  |      |        | Reference to another sequence diagram. See <a href="#">Interaction References and Interaction Activities (OOM) [page 127]</a> .  |
| Interaction Fragment   |      |        | Collection of associated messages. See <a href="#">Interaction Fragments (OOM) [page 128]</a> .  |
| Message                |      |        | Communication that conveys information with the expectation that action will ensue. See <a href="#">Messages (OOM) [page 131]</a> .  |
| Self Message           |      |        | Recursive message: the sender and the receiver are the same object. See <a href="#">Messages (OOM) [page 131]</a> .  |
| Procedure Call Message |      |        | Procedure call message with a default activation. See <a href="#">Messages (OOM) [page 131]</a> .  |
| Self Call Message      |      |        | Procedure call recursive message with a default activation. See <a href="#">Messages (OOM) [page 131]</a> .  |
| Return Message         |      |        | Specifies the end of a procedure. Generally associated with a Procedure Call, the Return message may be omitted as it is implicit at the end of an activation. See <a href="#">Messages (OOM) [page 131]</a> . |
| Self Return Message    |      |        | Recursive message with a Return control flow type. See <a href="#">Messages (OOM) [page 131]</a> .   |

### 1.4.3 Activity Diagrams

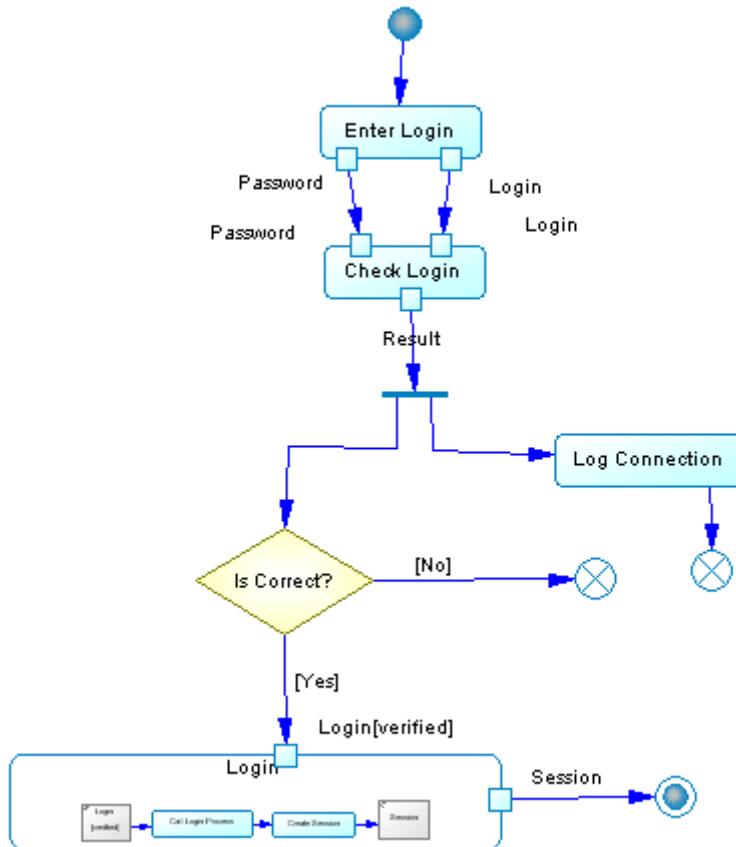
An activity diagram is a UML diagram that provides a graphical view of a system behavior, and helps you functionally decompose it in order to analyze how it will be implemented.

#### i Note

To create an activity diagram in an existing OOM, right-click the model in the Browser and select **New > Activity Diagram**. To create a new model, select **File > New Model**, choose Object Oriented Model as the model type and **Activity Diagram** as the first diagram, and then click **OK**.

Whereas a statechart diagram focuses on the implementation of operations in which most of the events correspond precisely to the end of the preceding activity, the activity diagram does not differentiate the states, the activities and the events.

The activity diagram gives a simplified representation of a process, showing control flows (called transitions) between actions performed in the system (called activities). These flows represent the internal behavior of a model element (use case, package, classifier or operation) from a start point to several potential end points.



You can create several activity diagrams in a package or a model. Each of those diagrams is independent and defines an isolated context in which the integrity of elements can be checked.

## Analyzing a Use Case

An activity diagram is frequently used to graphically describe a use case. Each activity corresponds to an action step and the extension points can be represented as conditional branches.

## Analyzing a Business Process

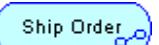
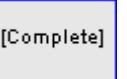
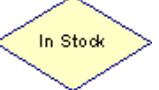
Beyond object-oriented modeling, activity diagrams are increasingly used to model the business processes of an enterprise. This kind of modeling takes place before the classic UML modeling of an application, and permits the

identification of the important processes of the enterprise and the domains of responsibility of each organizational unit within the enterprise.

For more information about business process modeling with PowerDesigner, see *Business Process Architecture*.

### 1.4.3.1 Activity Diagram Objects

PowerDesigner supports all the objects necessary to build activity diagrams.

| Object             | Tool  | Symbol  | Description   |
|--------------------|---|---|---|
| Start              |    |                  | Starting point of the activities represented in the activity diagram. See <a href="#">Starts and Ends (OOM) [page 167]</a> .                        |
| Activity           |    | <br>Find Item    | Invocation of an action. See <a href="#">Activities (OOM) [page 145]</a> .  |
| Composite activity | N/A   | <br>Ship Order | Complex activity decomposed to be further detailed. See <a href="#">Activities (OOM) [page 145]</a> .   |
| Object node        |  | <br>[Complete] | A specific state of an activity. See <a href="#">Object Nodes (OOM) [page 175]</a> .  |
| Organization unit  |  | <br>Sales      | A company, a system, a service, an organization, a user or a role. See <a href="#">Organization Units (OOM) [page 160]</a> .                        |
| Flow               |  |                | Path of the control flow between activities. See <a href="#">Flows (OOM) [page 173]</a> .   |
| Decision           |  | <br>In Stock   | Decision the control flow has to take when several flow paths are possible. See <a href="#">Decisions (OOM) [page 168]</a> .                        |
| Synchronization    |  |                | Enables the splitting or synchronization of control between two or more concurrent actions. See <a href="#">Synchronizations (OOM) [page 171]</a> . |

| Object | Tool | Symbol | Description  |
|--------|------|--------|--|
| End    |      |        | Termination point of the activities described in the activity diagram. See <a href="#">Starts and Ends (OOM)</a> [page 167]. |

## 1.4.4 Statechart Diagrams

A statechart diagram is a UML diagram that provides a graphical view of a State Machine, the public behavior of a classifier (component or class), in the form of the changes over time of the state of the classifier and of the events that permit the transition from one state to another.

### i Note

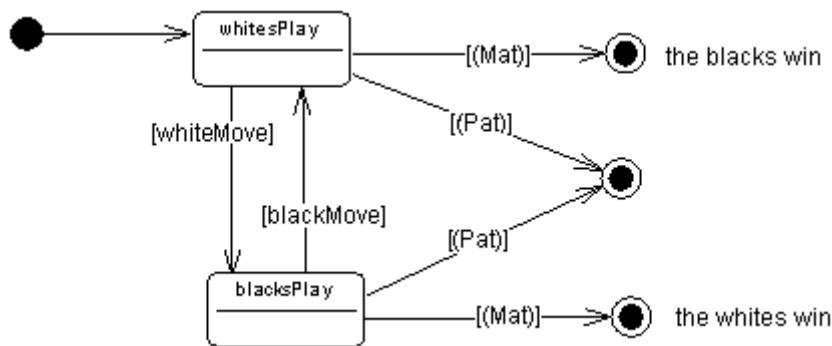
To create a statechart diagram in an existing OOM, right-click the model in the Browser and select [New](#) [Statechart Diagram](#). To create a new model, select [File](#) [New Model](#), choose Object Oriented Model as the model type and [Statechart Diagram](#) as the first diagram, and then click [OK](#).

It is assumed that the classifier has previously been identified in another diagram and that a finite number of states can be identified for it.

Unlike the interaction diagrams, the statechart diagram can represent a complete specification of the possible scenarios pertaining to the classifier. At any given moment, the object must be in one of the defined states.

You can create several statechart diagrams for the same classifier, but then the states and transitions represented should relate to a different aspect of its evolution. For example; a person can be considered on one hand as moving between the states of studying, working, being unemployed, and being retired, and on the other as transitioning between being single, engaged, married, and divorced.

Statechart diagrams show classifier behavior through execution rules explaining precisely how actions are executed during transitions between different states; these states correspond to different situations during the life of the classifier.



The example above shows the states of a game of chess.

The first step in creating a statechart diagram consists in defining the initial and final states and the set of possible states between them. Then you link the states together with transitions, noting on each, the event that sets off the transition from one state to another.

You can also define an action that executes at the moment of the transition. Similarly, the entry to or exit from a state can cause the execution of an action. It is even possible to define the internal events that do not change the state. Actions can be associated with the operations of the classifier described by the diagram.

It is also possible to decompose complex states into sub-states, which are represented in sub-statechart diagrams.

A statechart diagram requires the previous identification of a classifier. It can be used to describe the behavior of the classifier, and also helps you to discover its operations via the specification of actions associated with statechart events.

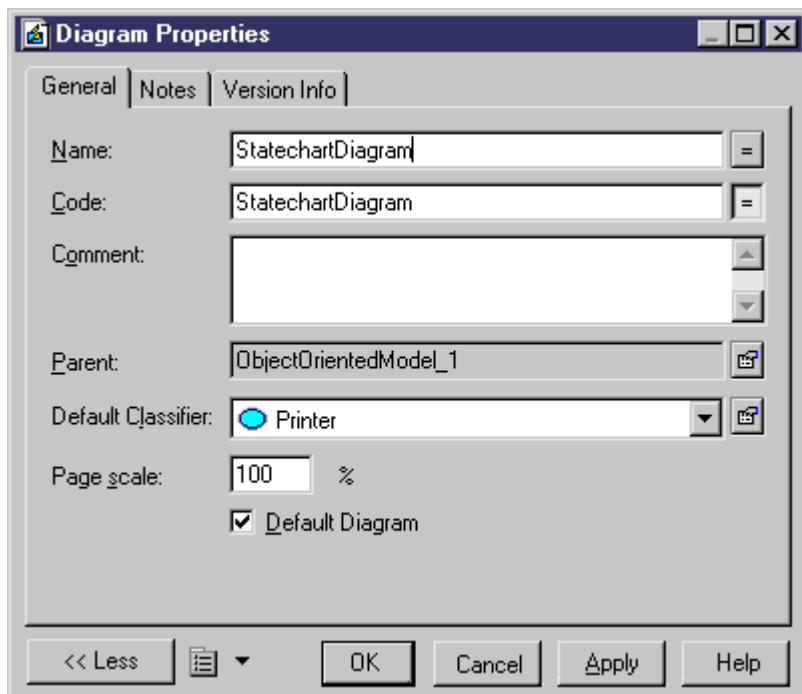
You can also use the transitions identified to establish the order in which operations can be invoked. This type of diagram is called a "protocol state machine".

Another potential use is the specification of a Graphic User Interface (GUI) where the states are the distinct screens available with possible transitions between them, all depending on keyboard and mouse events produced by the user.

#### 1.4.4.1 Defining a Default Classifier in a Statechart Diagram

You can define the classifier of a state using the Classifier list in the state property sheet. This allows you to link the state to a use case, a component or a class.

At the diagram level, you can also specify the context element of a state by filling in the Default Classifier list in the statechart diagram property sheet. As a result, each state that is created in a diagram using the State tool is automatically associated with the default classifier specified in the statechart diagram property sheet.



By default new diagrams are created with an empty value in the Default Classifier list, except sub-statechart diagrams that automatically share the same Classifier value defined on the parent decomposed state. The Default Classifier value is an optional value in the statechart diagram.

#### 1.4.4.2 Statechart Diagram Objects

PowerDesigner supports all the objects necessary to build statechart diagrams.

| Object          | Tool | Symbol | Description  |
|-----------------|------|--------|--|
| Start           |      |        | Starting point of the states represented in the statechart diagram. See <a href="#">Starts and Ends (OOM) [page 167]</a> .                                   |
| State           |      |        | The situation of a model element waiting for events. See <a href="#">States (OOM) [page 177]</a> .   |
| Action          | N/A  | N/A    | Specification of a computable statement. See <a href="#">Actions (OOM) [page 188]</a> .  |
| Event           | N/A  | N/A    | Occurrence of something observable, it conveys information specified by parameters. See <a href="#">Events (OOM) [page 185]</a> .                            |
| Transition      |      |        | Path on which the control flow moves between states. See <a href="#">Transitions (OOM) [page 182]</a> .  |
| Junction point  |      |        | Divides a transition between states. Used particularly when specifying mutually exclusive conditions. See <a href="#">Junction Points (OOM) [page 191]</a> . |
| Synchronization |      |        | Enables the splitting or synchronization of control between two or more concurrent states. See <a href="#">Synchronizations (OOM) [page 171]</a> .           |
| End             |      |        | Termination point of the states described in the statechart diagram. See <a href="#">Starts and Ends (OOM) [page 167]</a> .                                  |

## 1.4.5 Interaction Overview Diagrams

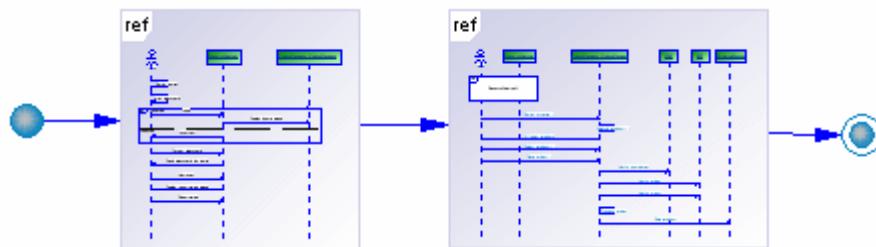
An interaction diagram is a UML diagram that provides a high-level graphical view of the control flow of your system as it is decomposed into sequence and other interaction diagrams.

### i Note

To create an interaction overview diagram in an existing OOM, right-click the model in the Browser and select **► New ► Interaction Overview Diagram**. To create a new model, select **► File ► New Model**, choose Object Oriented Model as the model type and *Interaction Overview Diagram* as the first diagram, and then click **OK**.

You can include references to sequence diagrams, communication diagrams, and other interaction diagrams.

In the following example, a control flow is shown linking two sequence diagrams:



### 1.4.5.1 Interaction Overview Diagram Objects

PowerDesigner supports all the objects necessary to build interaction overview diagrams.

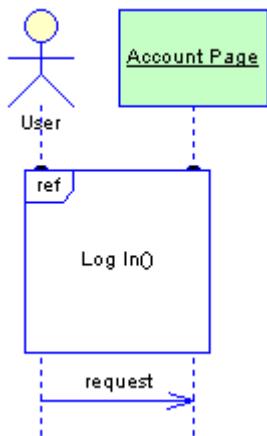
| Object               | Tool | Symbol | Description  |
|----------------------|------|--------|--|
| Start                |      |        | Starting point of the interactions represented in the diagram. See <a href="#">Starts and Ends (OOM)</a> [page 167].   |
| Interaction activity |      |        | Reference to a sequence diagram, communication diagram, or interaction overview diagram. See <a href="#">Interaction References and Interaction Activities (OOM)</a> [page 127]. |
| Flow                 |      |        | Flow of control between two interactions. See <a href="#">Flows (OOM)</a> [page 173].  |

| Object          | Tool | Symbol | Description  |
|-----------------|------|--------|--|
| Decision        |      |        | Decision the flow has to take when several paths are possible. See <a href="#">Decisions (OOM) [page 168]</a> .                        |
| Synchronization |      |        | Enables the splitting or synchronization of control between two or more flows. See <a href="#">Synchronizations (OOM) [page 171]</a> . |
| End             |      |        | Termination point of the interactions described in the diagram. See <a href="#">Starts and Ends (OOM) [page 167]</a> .                 |

## 1.4.6 Interaction References and Interaction Activities (OOM)

Interaction references allow you to represent one sequence diagram in the body of another to modularize and reuse commonly-used interactions across a range of sequence diagrams. Interaction activities allow you to represent a sequence, communication, or interaction overview diagram in an interaction overview diagram.

In this example, the user must log in before passing a request to the account page of a website. As the log in process will form a part of many interactions with the site, it has been abstracted to another sequence diagram called "Log In", and is represented here by an interaction reference:



When working in a sequence diagram, to reference another sequence diagram:

- Select the *Interaction Reference* tool in the Toolbox and click near the lifeline of an object or click and hold while drawing a box that will overlap and attach to several lifelines. When you release the mouse button, select the sequence diagram to reference (or select to create a new diagram) from the dialog, and click **OK**.
- Alternatively, drag the diagram from the browser and drop it into your diagram.

If you move or resize an interaction reference so that it overlaps an object lifeline, it attaches to it and displays a small bump on its top edge where it meets the lifeline. If you move or resize the symbol away, it detaches from the

lifeline. If you move an attached object, the reference symbol resizes to remain attached to the lifeline. You can attach or detach lifelines to a reference that passes over them by clicking the attachment point.

### i Note

An interaction reference cannot be copied or re-used in another diagram, but multiple references to the same diagram can be created.

When working in an interaction overview diagram, to reference a sequence or communication diagram or another interaction overview diagram:

- Select the *Interaction Activity* tool in the Toolbox and click in the diagram. When you release the mouse button, select the diagram to reference (or select to create a new diagram) from the dialog and click *OK*.
- Alternatively, drag the diagram from the browser and drop it into your diagram.

Right-click the activity and select ► *Composite View* ► *Read-only (Sub-Diagram)* ▾ to see the referenced diagram displayed in the symbol. Right-click and select ► *Composite View* ► *Adjust to read-only view* ▾ to resize the symbol to optimize the display of the referenced diagram.

## Interaction Reference and Interaction Activity Properties

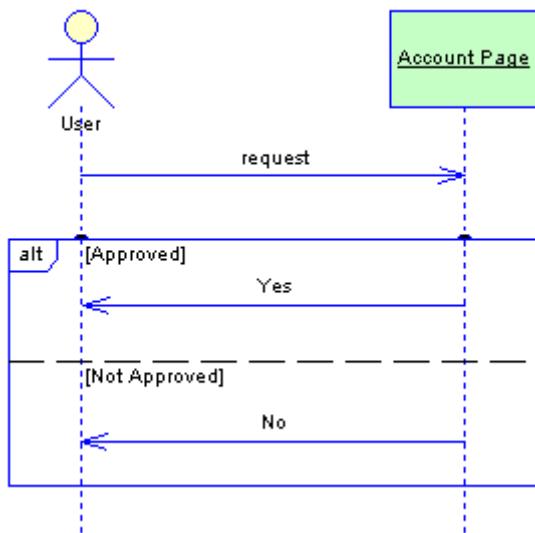
You can modify an object's properties from its property sheet. To open an interaction reference or interaction activity property sheet, double-click its diagram symbol in the top-left corner near the operator tag. The *General* tab contains the following properties:

| Property           | Description  |
|--------------------|--|
| Referenced Diagram | Specifies the diagram that will be represented in the current diagram. Click the <i>Create</i> tool to the right of the box to create a new diagram. |
| Stereotype         | Extends the semantics of the object beyond the core UML definition.  |
| Arguments          | Specifies the arguments to be passed to the first message in the referenced diagram.   |
| Return value       | Specifies the value to be returned by the last message in the referenced diagram.  |

## 1.4.7 Interaction Fragments (OOM)

An interaction fragment allows you to group related messages in a sequence diagram. Various predefined fragment types are available allowing you to specify alternate outcomes, parallel messages, or looping.

In the example below, the **User** sends a request to the **Account Page**. The two alternative responses and the conditions on which they depend, are contained within an interaction fragment.



To create an interaction fragment, select the *Interaction Fragment* tool in the Toolbox. Click near the lifeline of an object to create an interaction fragment attached to that lifeline, or click and hold while drawing a box that will overlap and attach to several lifelines.

### Note

An interaction fragment cannot be reused as a shortcut in another diagram.

If you move or resize an interaction fragment so that it overlaps an object lifeline, it attaches to it and displays a small bump on its top edge where it meets the lifeline. If you move or resize the symbol away, it detaches from the lifeline. If you move an attached object, the fragment symbol resizes to remain attached to the lifeline. You can attach or detach lifelines to a fragment that passes over them by clicking the attachment point.

You can move messages freely in and out of interaction fragments. If you move a message so that it is completely contained within a fragment, it will be attached to that fragment. Any message that is entirely enclosed within an interaction fragment will be moved with the fragment up and down the object lifeline to which it is attached. If you move a message so either of its ends is outside the fragment, then it is detached from the fragment.

When a fragment is split into two or more regions, you can move messages freely between regions, but you cannot move the dividing line between two regions over a message. You can resize a region by moving the dividing line below it. Such resizing will affect the total size of the fragment. To resize the last region, at the bottom of the fragment, select and move the bottom edge of the fragment. If you delete a region, then the space that it occupied and any messages it contained will be merged with the region above.

## Interaction Fragment Properties

You can modify an object's properties from its property sheet. To open an interaction fragment property sheet, double-click its diagram symbol in the top-left corner near the operator tag. The *General* tab contains the following properties:

| Property   | Description   |
|------------|---|
| Operator   | <p>Specifies the type of fragment. You can choose between:</p> <ul style="list-style-type: none"><li>• Alternative (alt) – the fragment is split into two or more mutually exclusive regions, each of which has an associated guard condition. Only the messages from one of these regions will be executed at runtime.</li><li>• Assertion (assert) – the interaction must occur exactly as indicated or it will be invalid.</li><li>• Break (break) – if the associated condition is true, the parent interaction terminates at the end of the fragment.</li><li>• Consider (consider) – only the messages shown are significant.</li><li>• Critical Region (critical) – no other messages can intervene until these messages are completed.</li><li>• Ignore (ignore) – some insignificant messages are not shown.</li><li>• Loop (loop) – the interaction fragment will be repeated a number of times.</li><li>• Negative (neg) – the interaction is invalid and cannot happen.</li><li>• Option (opt) – the interaction only occurs if the guard condition is satisfied.</li><li>• Parallel (par) – the fragment is split into two or more regions, all of which will be executed in parallel at runtime.</li><li>• Strict Sequencing (strict) – the ordering of messages is strictly enforced.</li><li>• Weak Sequencing (seq) – the ordering of messages is enforced on each lifeline, but not between lifelines.</li></ul> <p>The operator type is shown in the top left corner of the interaction fragment symbol.</p> |
| Stereotype | Extends the semantics of the object beyond the core UML definition.   |
| Condition  | [if the operator supports conditions] Specifies any condition associated with the fragment. This may be the evaluation of a variable, such as $X > 3$ , or, for a loop fragment, the specification of the minimum (and optionally maximum) number of times that the loop will run, such as $1, 10$ .<br><br>For the Consider or Ignore operators, this field lists the associated messages.   |

The following tabs may also be available:

- *Interaction Sub-Regions* - [if the operator supports multiple regions] Lists the regions contained within the fragment. You can add or delete regions and (if appropriate) specify conditions for them.

## 1.4.8 Messages (OOM)

A message is a communication between objects. The receipt of a message will normally have an outcome.

A message can be created in the following diagrams:

- Communication Diagram
- Sequence Diagram

Objects can cooperate by using several kinds of requests (send a signal, invoke an operation, create an object, delete an existing object, etc.). Sending a signal is used to trigger a reaction from the receiver in an asynchronous way and without a reply. Invoking an operation will apply an operation to an object in a synchronous or asynchronous mode, and may require a reply from the receiver. All these requests constitute messages. They correspond to stimulus in the UML language.

A message has a sender, a receiver, and an action. The sender is the object or actor that sends the message. The receiver is the object or actor that receives the message. The action is executed on the receiver. You can also create recursive messages, where the same object is the sender and receiver.

The message symbol is an arrow showing its direction, and can also display the following information:

- A sequence number indicating the order in which messages are exchanged (see [Message Sequence Numbers \[page 143\]](#))
- The message name (or the name of the associated operation)
- The condition
- The return value
- The argument

## Reusing Messages

The same message can be used in a sequence and a communication diagram or in multiple diagrams of either type. When you drag a message from one diagram to another, it is dropped with both extremities if they do not exist, and (in a communication diagram) it is attached to a default instance link.

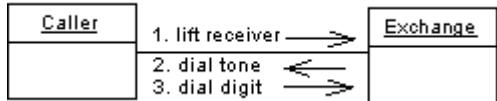
The sequence number attached to a message is identical in all diagrams if the message is reused.

When you copy a message, its name does not change. You can either keep its original name, or rename the message after copy.

Any change on the Action or Control Flow value of the message is reflected in all diagrams. However, if the change you want to perform is not valid, the change will not be possible. For example, you are not allowed to move a Create message if a Create message already exists between the sender and the receiver.

## Messages in a Communication Diagram

In a communication diagram, each message is associated with an instance link. An instance link may have several associated messages, but each message can be attached to only one instance link. The destruction of an instance link destroys all the messages associated with it.



## Messages in a Sequence Diagram

A message is shown as a horizontal solid arrow from the lifeline of one object or actor, to the lifeline of another. The arrow is labeled with the name of the message. You can also define a control flow type that represents both the relationship between an action and its preceding and succeeding actions, and the waiting semantics between them.

In a sequence diagram you can choose between the following types of messages:

- Message
- Self Message
- Procedure Call Message
- Self Call Message
- Return Message
- Self Return Message

You can create activations on the lifeline of an object to represent the period of time during which it is performing an action.

A message can be drawn from an actor to an object, or inversely. It is also possible to create a message between two actors but it will be detected, and displayed as a warning during the check model process.

### i Note

If you need to fully describe, or put a label on a message, you can write a note using the Note tool, and position the note close to the message.

### 1.4.8.1 Creating a Message

You can create a message from the Toolbox, Browser, or [Model](#) menu.

- Use the [Message](#) tool in the Toolbox. For sequence diagrams, the following specialized tools are available:

| Tool | Symbol | Description   |
|------|--------|---|
|      |        | <i>Message</i> - Communication that conveys information with the expectation that action will ensue.  |
|      |        | <i>Procedure Call Message</i> - Procedure call message with a default activation.   |
|      |        | <i>Return Message</i> - Specifies the end of a procedure. Generally associated with a <b>Procedure Call</b> , the <b>Return</b> message may be omitted as it is implicit at the end of an activation. |
|      |        | <i>Self Message</i> - Click the object lifeline to create a recursive message where the sender and the receiver are the same object.  |
|      |        | <pre> sequenceDiagram     participant LM as License Manager     participant A as Application     LM-&gt;&gt;LM: findAppl     LM-&gt;&gt;A: addLicense   </pre>  |
|      |        | <p><b>Note</b></p> <p>The <b>Create</b> and <b>Self-Destroy</b> actions, and the <b>Support delay</b> property are not available for recursive messages.</p>  |

| Tool | Symbol | Description   |
|------|--------|---|
|      |        | <p><i>Self Call Message</i> - Click the object lifeline to create a recursive procedure call message with a default activation.</p> |
|      |        | <p><i>Self Return Message</i> - Recursive message with a <b>Return</b> control flow type.</p>                                       |

- Select **Model > Messages** to access the List of Messages, and click the **Add a Row** tool.
- Right-click the model (or a package) in the Browser, and select **New > Message**.
- Open the property sheet of an instance link (in a communication diagram), click the Messages tab, and click the **Create a New Message** tool.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.4.8.2 Message Properties

To view or edit a message's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Sender            | Object the message starts from. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.   |
| Receiver          | Object the message ends on. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.. You can also reverse the direction of the message.<br><br><b>i Note</b><br>You can right-click a message in the diagram and select <i>Reverse</i> to reverse its direction. You cannot reverse the direction of a Create or Destroy message.   |
| Sequence number   | Allows you to manually add a sequence number to the message. It is mainly used in communication diagrams to describe the order of messages, but can also be used in sequence diagrams  |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## Detail tab

The *Detail* tab includes the following properties:

| Property     | Description  |
|--------------|--|
| Action       | <p>Specifies the type of message action. You can choose between:</p> <ul style="list-style-type: none"><li>• Create – the sender object instantiates and initializes the receiver object. A message with a create action is the first message between a sender and a receiver.</li><li>• Destroy – the sender object destroys the receiver object. A large X is displayed on the lifeline of the receiver object. A message with a destroy action is the last message between a sender and a receiver.</li><li>• Self-Destroy – (only available if the control flow property is set to "Return") the sender object warns the receiver object that it is destroying itself. A large X is displayed on the lifeline of the sender object. A message with a self-destroy action is the last message between a sender and a receiver.</li></ul>  |
| Control flow | <p>Specifies the mode in which messages are sent. You can choose between:</p> <ul style="list-style-type: none"><li>• Asynchronous – the sending object does not wait for a result, it can do something else in parallel. No-wait semantics</li><li>• Procedure Call – Call of a procedure. The sequence is complete before the next sequence resumes. The sender must wait for a response or the end of the activation. Wait semantics</li><li>• Return – Generally associated with a Procedure Call. The Return arrow may be omitted as it is implicit at the end of an activation</li><li>• Undefined – No control flow defined</li></ul>   |
| Operation    | <p>Links the message to an operation of a class. If the receiver of a message is an object, and the object has a class, the message, as a dynamic flow of information, invokes an operation. You can therefore link a message to an existing operation of a class but also operations defined on parent classes, or you can create an operation from the Operation list in the message property sheet.</p> <p>If an operation is linked to a message, you can replace the message name with the name of the method that one object is asking the other to invoke. This process can be very useful during implementation. To display the name of the operation instead of the name of the message, select the Replace by Operation Name display preference in the message category.</p> <p>You can link a Create message to a Constructor operation of a class if you wish to further detail a relation between a message and an operation. You are not allowed however to link a message with a Return control flow to an operation.</p> <p>If you change the generalization that exists between classes, the operation that is linked to the message may no longer be available. In this case, the operation is automatically detached from the message. The same occurs when you reverse the message direction, unless the new receiver object has the same class.</p> |
| Arguments    | Arguments of the operation   |
| Return value | Function return value stored in a variable and likely to be used by other functions  |

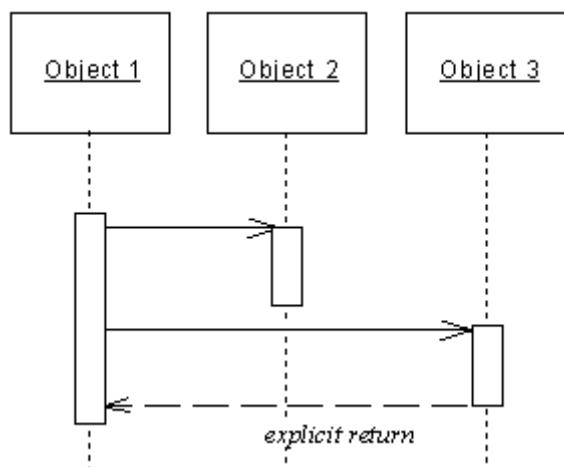
| Property         | Description   |
|------------------|---|
| Predecessor list | Made of a list of sequence numbers followed by " / ", the predecessor list defines which messages must be exchanged before the current message could be sent. Example: sequence numbers 1, 2, 4 before 3 = "1,2,4/ 3"   |
| Condition        | Condition attached to the message. May be specified by placing Boolean expressions in braces on the diagram. Example: condition for timing: [dialing time < 30 sec]   |
| Begin time       | User-defined time alias, used for defining constraints. Example: Begin time = t1, End time = t2, constraint = {t2 - t1 < 30 sec}  |
| End time         | User-defined time alias, used for defining constraints.   |
| Support delay    | <p>Specifies that the message may have duration. The message symbol may slant downwards.</p> <p>If this option is not selected, the message is instantaneous, or fast, and the message symbol is horizontal.</p> <p>You can specify Support delay as a default option in the Model Options dialog box.</p> <p>Support delay is not available with a recursive message: it is selected and grayed out.</p> |

## Control Flow

By default, a message has an Undefined control flow.

If you want to make a diagram more readable, you can draw the Return arrow to show the exact time when the action is returned back to the sender. It is an explicit return that results in returning a value to its origin.

In the example below, the explicit Return causes values to be passed back to the original activation.



You can combine message control flows and message actions according to the following table:

| Control flow   | Symbol | No action | Create | Destroy | Self-Destroy |
|----------------|--------|-----------|--------|---------|--------------|
| Asynchronous   |        |           |        |         | Yes          |
| Procedure Call |        |           |        |         | Yes          |
| Return         |        |           | Yes    | Yes     |              |
| Undefined      |        |           |        |         | Yes          |

#### i Note

You can access the Action and Control flow values of a message by right clicking the message symbol in the diagram, and selecting Action/Control flow from the contextual menu.

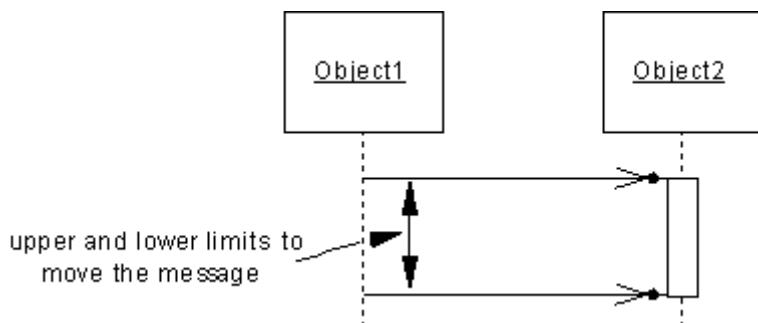
### 1.4.8.3 Activations (OOM)

Activations are optional symbols that are created on the lifeline of objects in sequence diagrams and represent the time required for an action to be performed. An activation is created by default when you create a procedure call message but you can associate any type of message, and create activations. Activations cannot be created on the lifelines of actors, and do not have properties.

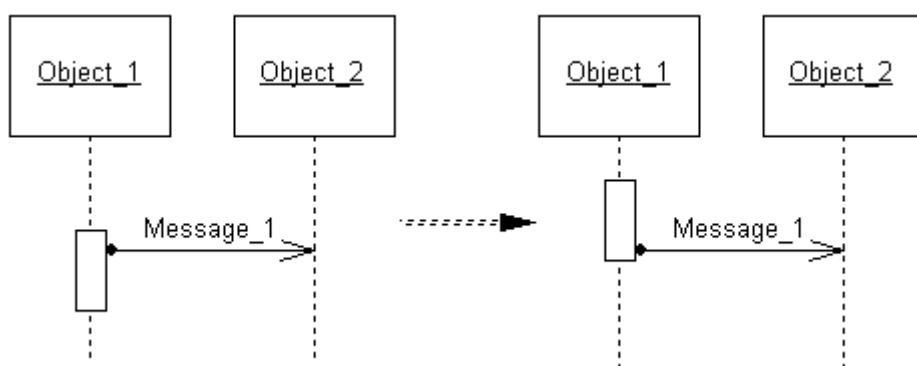
To create an activation, use one of the following tools:

| Tool | Description   |
|------|---|
|      | Activation - Click the lifeline of an object. If you create the activation on top of an existing message the message will be attached to the activation.  |
|      | Procedure Call Message (with a default activation). Click and hold on the lifeline of the sender object or actor, drag the cursor to the receiver object or actor lifeline and then release. An activation is created on the sender lifeline (unless it is an actor), from which a message is sent to a second activation on the receiver lifeline (unless it is an actor). |
|      | Self Call Message (with a default activation). Click on the lifeline of an object. One large activation is created and a second smaller activation is created on top of it with a message being sent from the first to the second activation.   |

To attach a message to an activation, press the CTRL key and drag the message onto the activation. When a message is attached to an activation, you cannot move it outside the limits of the activation symbol:



Procedure call messages are attached to the top of the receiver lifeline activation while return messages are attached to the bottom of the sender lifeline activation. These messages move with the activation when it is moved or resized to retain their positions. Other messages do not move with the activation when you move or resize it, and so you can only move the activation up and down until its top or bottom reaches the level of such a message:

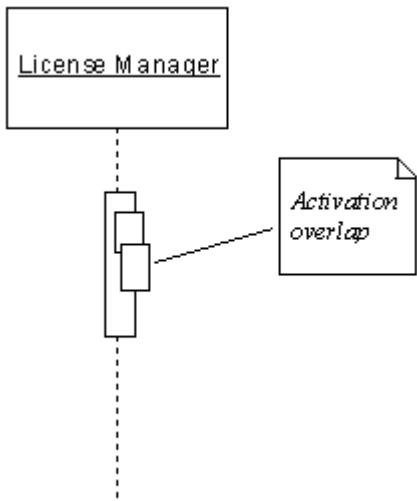


### i Note

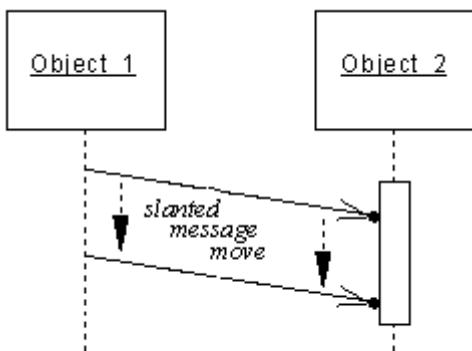
Messages that are covered by the activation after movement or resizing are not automatically attached to the activation. To attach such a message, select the **Tools > Display Preferences** and select the **Activation Attachment** display preference in the **Message** category.

Attachment points can be displayed to show that a message is attached to an activation. If they are not present in your diagram, select **Tools > Display Preferences**, and select the **Activation Attachment** display preference in the **Message** category).

Activations can be placed so that they overlap other activation to show concurrent activities. For example, you may want to represent an action in a loop that is done repeatedly until it reaches its goal, and have this loop start and finish at the same time while another action is happening:



If you move the endpoint of a message with the *Support Delay* property selected, the angle of the message flow is preserved:



To detach a message from an activation, press the `Ctrl` key and drag it off of the activation to another point on the lifeline. If you delete an activation with a message attached, the message will be detached from the activation but will not be deleted.

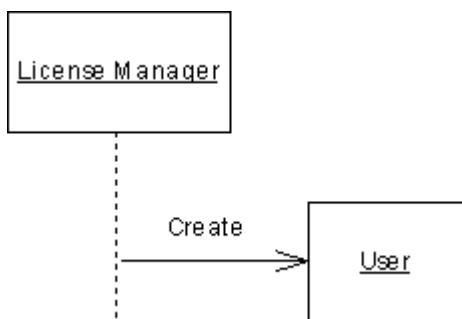
#### Note

An activation can only be created in sequence diagrams, but if messages associated with an activation are also displayed in a communication diagram, then they are given sub-numbers (see [Message Sequence Numbers \[page 143\]](#)). Thus an activation created by message 1, may give rise to messages 1.1 and 1.2.

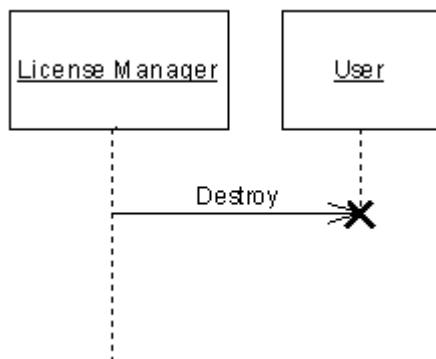
#### 1.4.8.4 Creating Create and Destroy Messages

To specify a message as a create, destroy, or self-destroy message, open its property sheet and select the appropriate action on the *Detail* tab.

- If the message is the first to be received by the receiver object and you specify the **Create** action, the receiver object symbol moves down to line up with the create message.

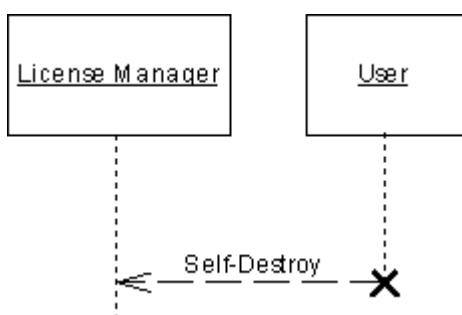


- If the message is the last to be received by the receiver object and you specify the **Destroy** action, an X is placed at the intersection point between the Destroy message arrow and the receiver object lifeline.



The object lifeline and any activation are stopped at this same point, and you cannot extend them further down. It does not destroy the object, but only represents this destruction in the diagram.

- If the message is after the last to be received by the sender object and you specify the **Self-Destroy** action, an X is placed at the intersection point between the Destroy message arrow and the sender object lifeline:



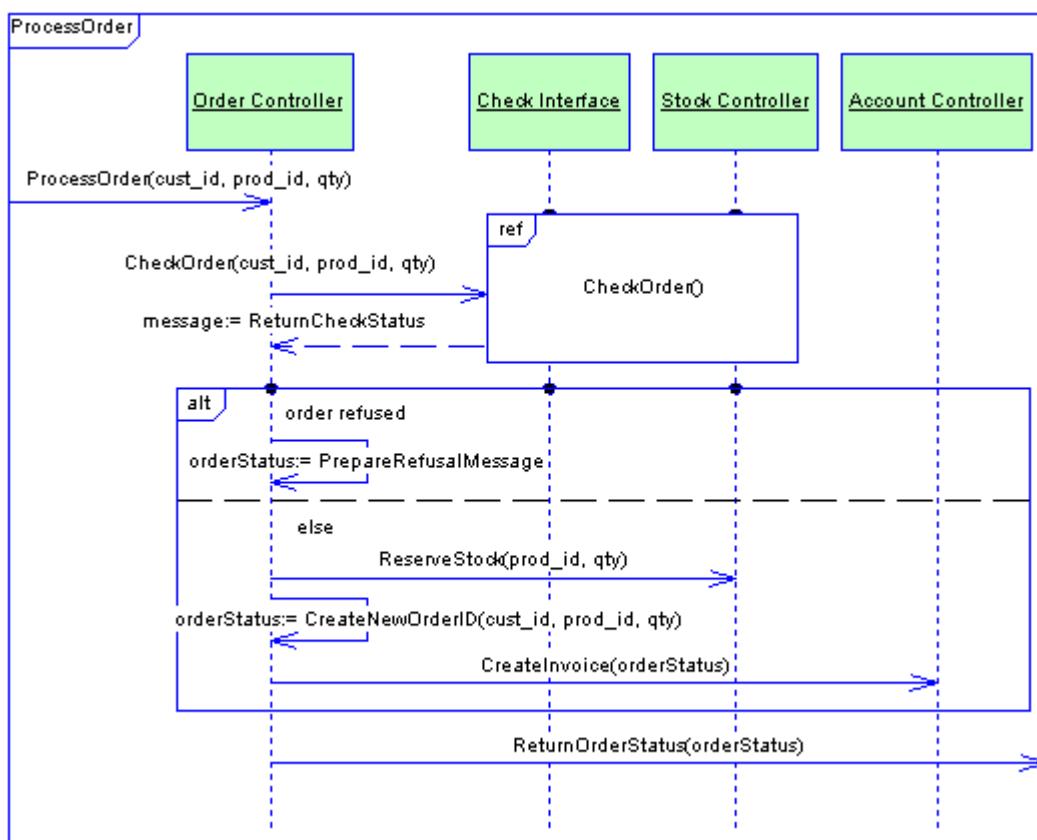
### **i Note**

You cannot create or destroy actors or use the create or destroy action with a recursive message.

## 1.4.8.5 Messages and Gates

In UML 2, you can send messages to and from the interaction frame that surrounds your sequence diagram. The frame represents the outer edge of the system (or of the part of the system) being modeled and can be used in place of an actor (actors are no longer used in UML 2 sequence diagrams, but continue to be supported for backwards compatibility in PowerDesigner). A message originating from a point on the frame is said to be sent from an input gate, while a message arriving there is received by an output gate.

In the example below, a high-level sequence diagram, ProcessOrder, shows a series of communications between a user and an sales system:

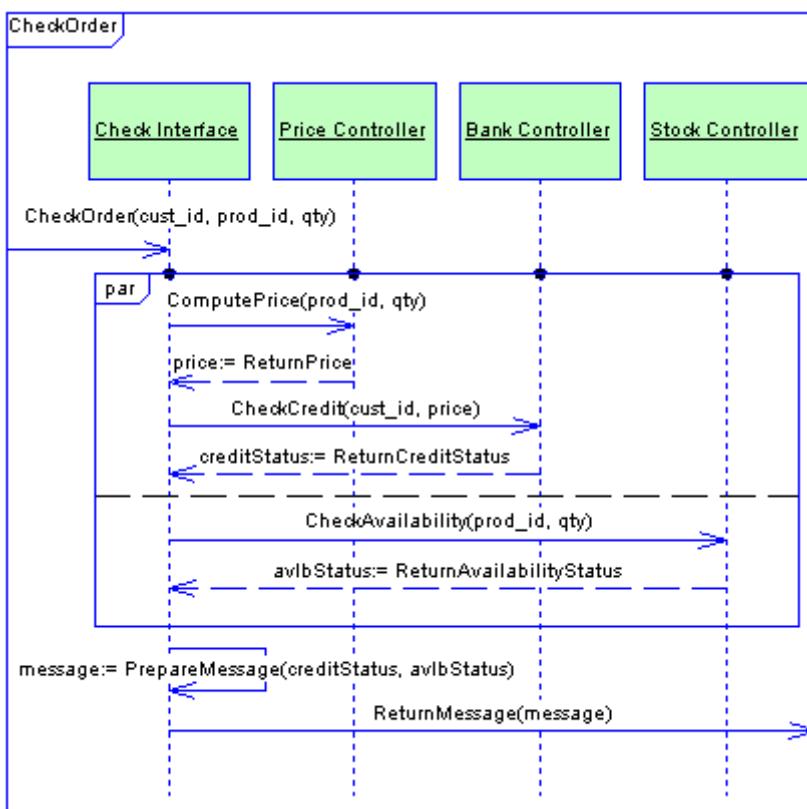


The message `ProcessOrder` originates from an input gate on the `ProcessOrder` interaction frame, and is received as an input message by the `Order Controller` object. Once the order processing is complete, the message `ReturnOrderStatus` is received by an output gate on the `ProcessOrder` interaction frame.

The message `CheckOrder` originates from the `Order Controller` object, and is received as an input message by an input gate on the `CheckOrder` interaction reference frame. Once the order checking is complete, the

ReturnCheckStatus message is sent from an output gate on the CheckOrder interaction reference frame and is received by the Order Controller object.

The following diagram shows the CheckOrder sequence diagram which illustrates the detail of the order checking process:



Here, the message `CheckOrder` originates from an input gate on the `CheckOrder` interaction frame, and is received as an input message by the `Check Interface` object. Once the order processing is complete, the message `ReturnMessage` is received by an output gate on the `CheckOrder` interaction frame.

#### i Note

PowerDesigner allows you to use actors and interaction frames in your diagrams in order to provide you with a choice of styles and to support backwards compatibility. However, since both represent objects exterior to the system being modeled, we recommend that you do not intermingle actors and frames in the same diagram. You cannot send messages between an actor and an interaction frame.

### 1.4.8.6 Message Sequence Numbers

Sequence numbers can be assigned to messages in both communication and sequence diagrams, but they are most important in communication diagrams where the default display preferences show them by default. The first message created in a communication diagram has a sequence number set to 1 by default, and each subsequent message number is incremented by one.

You can modify any sequence number and subsequently-created messages will be incremented from the last created or modified number. New numbers respect the syntax of the last-edited number, so that after setting a sequence number to 1.1, the next message created will be numbered 1.2 by default (if that number is available). To modify a sequence number:

- Click the message label in the diagram and then click the sequence number to edit it in place.

### Note

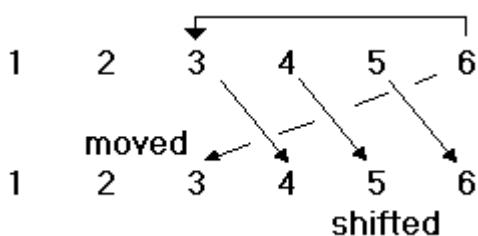
To increment or decrement the number by one, select the message arrow symbol and press CTRL and the numpad + or - key or right-click and select *Increase Number* or *Decrease Number*.

- Double-click the message in the diagram or Browser to open its property sheet and use the *Sequence number* property on the *General* tab.
- Select  *Model* > *Messages*  to open the List of Messages. If the *Sequence Number* column is not visible, click the *Customize Columns and Filters* tool to add it.

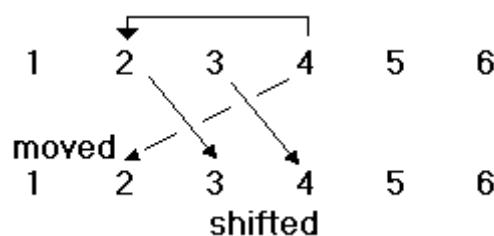
By convention, the addition of letters to sequence numbers signifies that the messages are parallel. For example, the messages with sequence numbers 3.1a and 3.1b are sent at the same time.

If you change the sequence number of a message to a value that is already used, other numbers are incremented or decremented to prevent duplication or to fill gaps:

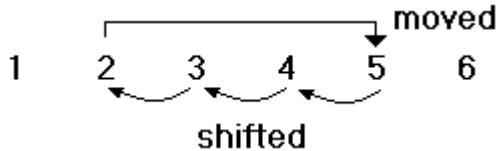
Change Message 6 to 3:



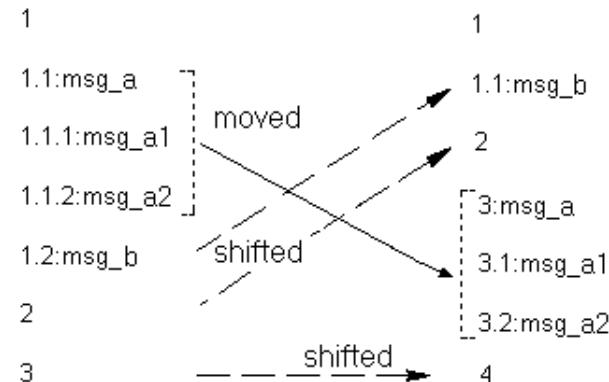
Change Message 4 to 2:



Change Message 2 to 5:



Change Message 1.1:msg\_a to 3:



---

### **i** Note

Use Undo if renumbering causes unexpected changes.

## 1.4.9 Activities (OOM)

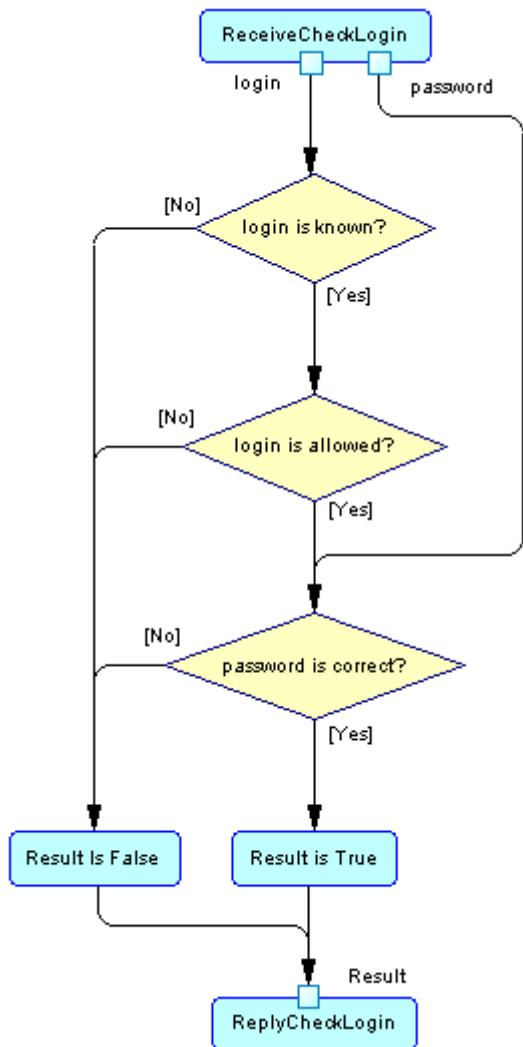
An activity is the invocation of a manual or automated action, such as "send a mail", or "increment a counter". When the activity gains control, it performs its action and then, depending on the result of the action, the transition (control flow) is passed to another activity.

An activity can be created in the following diagrams:

- Activity Diagram

PowerDesigner's support for UML 2 allows you a great deal of flexibility in the level of detail you provide in your activity diagrams. You can simply link activities together to show the high-level control flow, or refine your model by specifying the:

- parameters that are passed between the activities (see [Specifying Activity Parameters \[page 149\]](#))
- action type of the activity and associate it with other model objects (see [Specifying Action Types \[page 150\]](#))



In the example above, the ReceiveCheckLogin activity has an action type of "Accept call" (see [Specifying Action Types \[page 150\]](#)), and passes the two output parameters "login" and "password" (see [Specifying Activity Parameters \[page 149\]](#)) to a series of decisions that lead to the ReplyCheckLogin. This last activity has an input parameter called "Result" and an action type of Reply Call.

## Atomic and Decomposed Activities

An activity can be atomic or decomposed. Decomposed activities contain sub-activities, which are represented in a sub-diagram. For more information, see [Decomposed Activities and Sub-Activities \[page 157\]](#).

A PowerDesigner activity is equivalent to a UML activity (ActionState or SubactivityState) and an activity graph. In UML, an ActionState represents the execution of an atomic action, and the SubactivityState is the execution of an activity graph (which is, in turn, the description of a complex action represented by sub-activities).

The following table lists the mappings between UML and PowerDesigner terminology and concepts:

| UML Objects      | PowerDesigner Objects |
|------------------|-----------------------|
| ActionState      | Activity              |
| SubactivityState | Composite activity    |
| Activity Graph   | Composite activity    |

PowerDesigner combines a SubactivityState and an activity graph into a decomposed activity so that you can define sub-activities directly under the parent without defining an additional object. If you do need to highlight the difference, you can create activities directly under the model or the package, and use activity shortcuts to detail the activity implementation, so that the SubactivityState corresponds to the shortcut of a decomposed activity.

### 1.4.9.1 Creating an Activity

You can create an activity from the Toolbox, Browser, or *Model* menu.

- Use the *Activity* tool in the Toolbox.
- Select *Model* > *Activities* to access the List of Activities, and click the *Add a Row* tool.
- Right-click the model, package, or decomposed activity in the Browser, and select *New* > *Activity*

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

### 1.4.9.2 Activity Properties

To view or edit an activity's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |

| Property          | Description   |
|-------------------|---|
| Organization unit | <p>Specifies the organization unit (see <a href="#">Organization Units (OOM) [page 160]</a>) linked to the activity and also allows you to assign the Committee Activity value (see <a href="#">Displaying a Committee Activity [page 162]</a>) to a decomposed activity to graphically show the links between organization units designed as swimlanes and sub-activities. A Committee Activity is an activity realized by more than one organization unit.</p> <p>You can click the Ellipsis button beside the Organization unit list to create a new organization unit or click the Properties tool to display its property sheet.</p> |
| Composite status  | <p>Specifies whether the activity is decomposed into sub-activities. You can choose between:</p> <ul style="list-style-type: none"> <li>Atomic Activity – (default) The activity does not contain sub-activities</li> <li>Decomposed Activity – the activity can contain sub-activities. A Sub-Activities tab is displayed in the property sheet to list these sub-activities, and a sub-diagram is created below the activity in the Browser to display them.</li> </ul> <p>If you revert the activity from Decomposed to Atomic status, then any sub-activities that you have created will be deleted.</p>                              |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.   |

## Action Tab

The *Action* tab defines the nature, the type and the duration of an action that an activity executes. It contains the following properties:

| Property                                   | Description  |
|--|--|
| Action type                                | Specifies the kind of action that the activity executes. For more information, see <a href="#">Specifying Action Types [page 150]</a> .  |
| [action object]                            | Depending on the action type you choose, an additional field may be displayed, allowing you to specify an activity, classifier, attribute, event, expression, operation, or variable upon which the action acts. You can use the tools to the right of the list to create an object, browse the available objects or view the properties of the currently selected object. |
| Pre-Conditions / Actions / Post-Conditions | These sub-tabs provide a textual account of how the action is executed. For example, you can write pseudo code or information on the program to execute.   |
| Duration                                   | Specifies the estimated or statistic duration to execute the action. This information is for documentation purposes only; estimate on the global duration is not computed.   |
| Timeout                                    | Zero by default. If the value is not set to zero, it means that a timeout exception occurs if the execution of the activation takes more than the specified timeout limit. You can type any alphanumeric value in the Timeout box (example: 20 seconds).   |

## **Input Parameters and Output Parameters Tabs**

These tabs list the input and output parameters required by the activity (see [Specifying Activity Parameters \[page 149\]](#)).

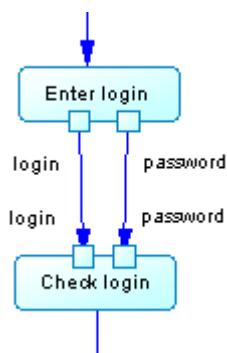
### **Sub-Activities Tab**

This tab is displayed only if the Composite status of the activity is set to Decomposed, and lists its sub-activities.

### **1.4.9.3 Specifying Activity Parameters**

Activity parameters are values passed between activities. They are represented as small squares on the edges of activity symbols. In this example, the parameters login and password are passed from the Enter login activity to the Check login activity.

#### **Context**



#### **Procedure**

1. Open the property sheet of an activity and click the Input Parameters or Output Parameters tab.
2. Use the tools to add an existing parameter or to create a new one.

## Results

### i Note

You can also create parameters as a part of specifying an activity action type. See [Specifying Action Types \[page 150\]](#).

Activity parameters can have the following properties:

| Property              | Description  |
|-----------------------|--|
| Parent                | Specifies the parent activity.   |
| Name/Code/<br>Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype            | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Data type             | Specifies the data type of the parameter. You can choose a standard data type or specify a classifier. You can use the tools to the right of the list to create a classifier, browse the available classifiers or view the properties of the currently selected classifier.  |
| State                 | Specifies the object state linked to the parameter. You can enter free text in the field, or use the tools to the right of the list to create a state, browse the available states or view the properties of the currently selected state.   |
| Keywords              | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

### 1.4.9.4 Specifying Action Types

You can add additional detail to your modeling of activities by specifying the type of action performed and, in certain cases, associating it with a specific model object that it acts upon, and the parameters that it passes.

## Procedure

1. Open the property sheet of an activity and click the *Action* tab.
2. Select an action type. The following list details the available action types, and specifies where appropriate, the required action object:

- **<Undefined>** [no object] - default. No action defined
  - **Reusable Activity** [no object] – a top-level container.
  - **Call** [operation or activity] – calls an operation or activity. See [Example: Using the Call Action Type \[page 151\]](#)
  - **Accept Call** [operation or activity] – waits for an operation or activity to be called.
  - **Reply Call** [operation or activity] – follows an Accept Call action, and responds to an operation or activity.
  - **Generate Event** [event] – generates an event. Can be used to raise an exception.
  - **Accept Event** [event] – waits for an event to occur.
  - **Create Object** [classifier] – creates a new instance of a classifier
  - **Destroy Object** [classifier] – destroys an instance of a classifier
  - **Read Attribute** [classifier attribute] – reads an attribute value from a classifier instance
  - **Write Attribute** [classifier attribute] – writes an attribute value to a classifier instance
  - **Read Variable** [variable] – writes a value to a local variable. The variable can be used to store an output pin provided by an action to reuse later in the diagram. See [Variable Properties \[page 156\]](#).
  - **Write Variable** [variable] - reads a value from a local variable. See [Variable Properties \[page 156\]](#).
  - **Evaluate Expression** [expression text] – evaluates an expression and returns the value as an output pin.
  - **Unmarshal** [no object] – breaks an input object instance into several outputs computed from it.
  - **Region** [no object] – a composite activity that isolates a part of the graph. Equivalent to the UML Interruptible Activity Region.
  - **For Each** [no object] – loops an input collection to execute a set of actions specified into the decomposed activity. Equivalent to the UML Expansion Region.
  - **Loop Node** [expression text] – expression text
3. If the action type requires an action object, an additional field will be displayed directly below the *Action Type* list, allowing you to specify an activity, classifier, attribute, event, expression, operation, or variable upon which the action acts. You can use the tools to the right of the list to create an object, browse the available objects or view the properties of the currently selected object.
4. Click **OK** to save your changes and return to the diagram.

#### 1.4.9.4.1 Example: Using the Call Action Type

One of the most common action types is **Call**, which allows an activity to invoke a classifier operation (or another activity).

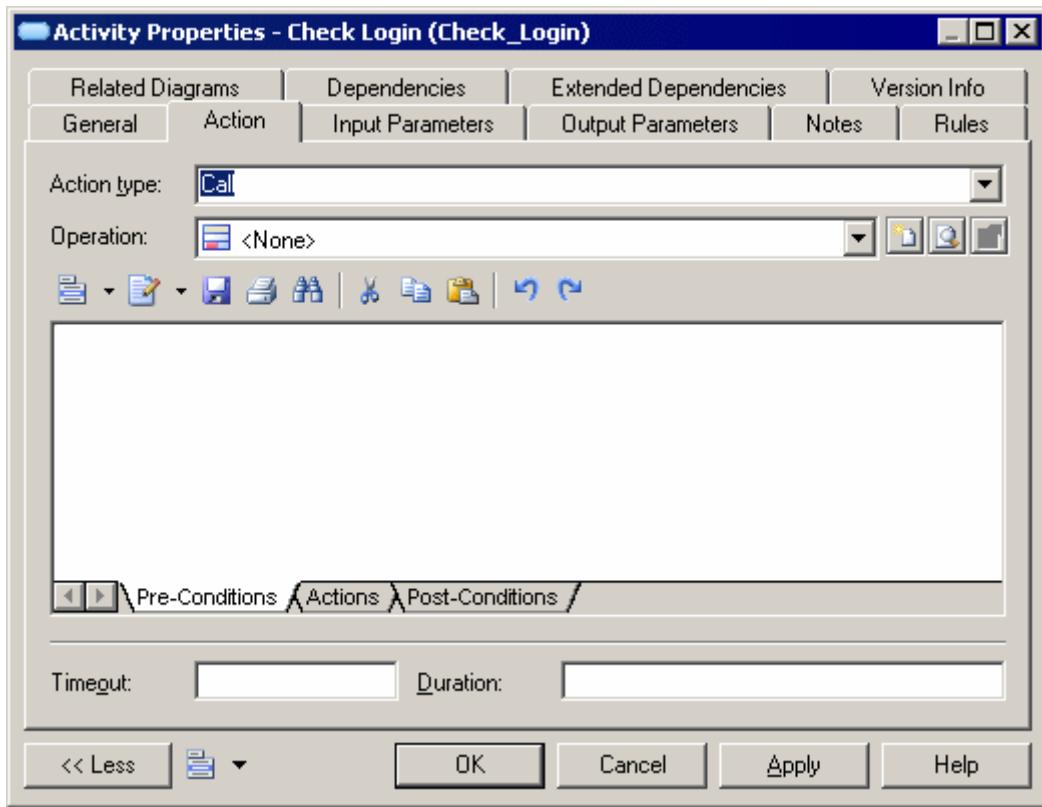
#### Procedure

1. Create an activity and call it **Check Login**.

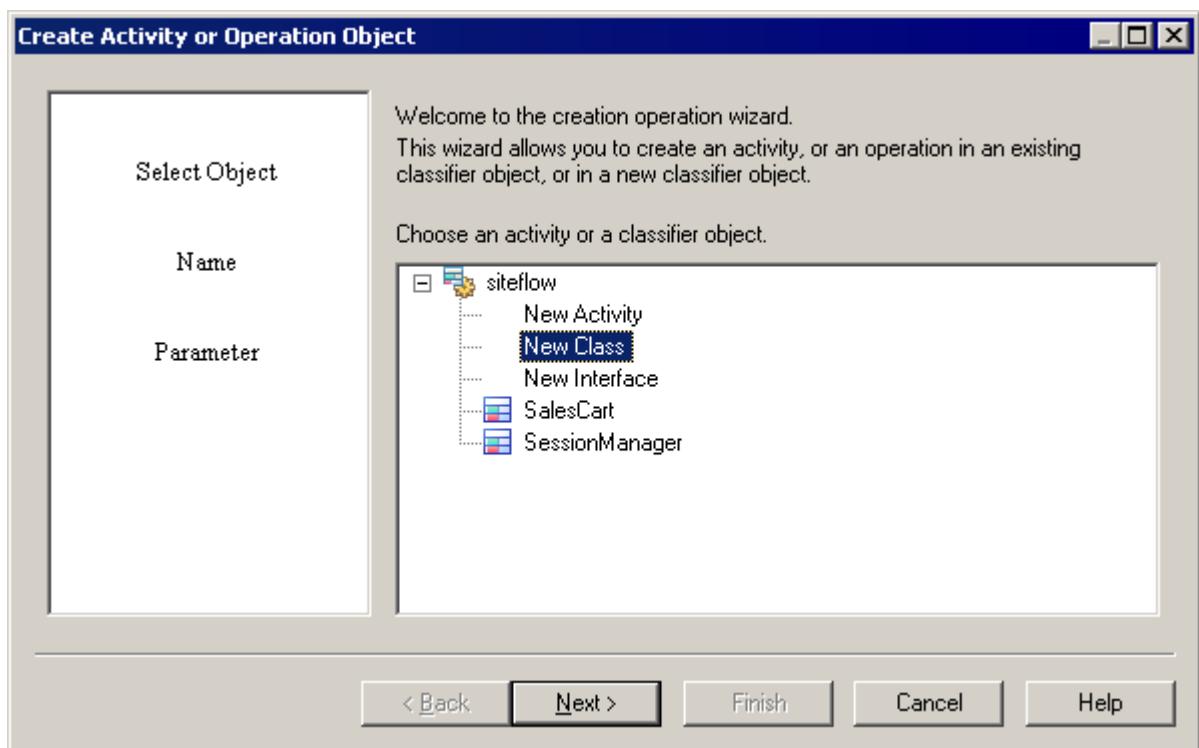


Check Login

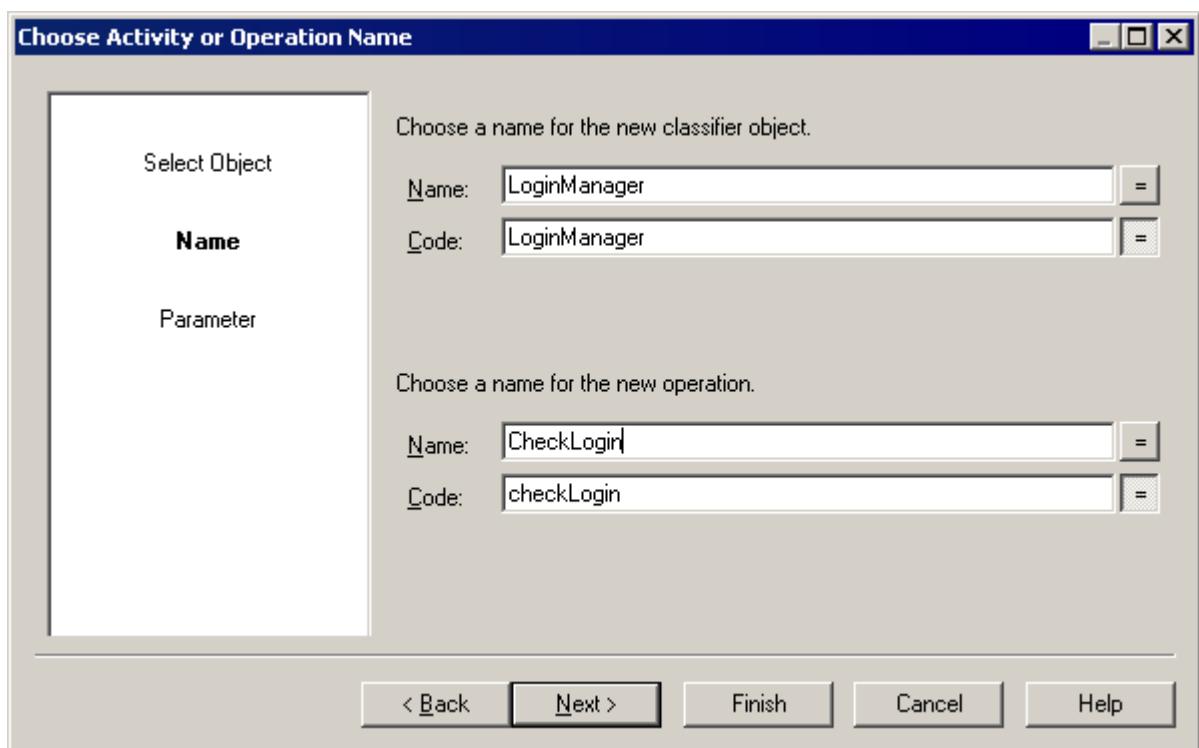
2. Open its property sheet, click the *Action* tab, and select **call** from the *Action type* list. The *Operation* field appears:



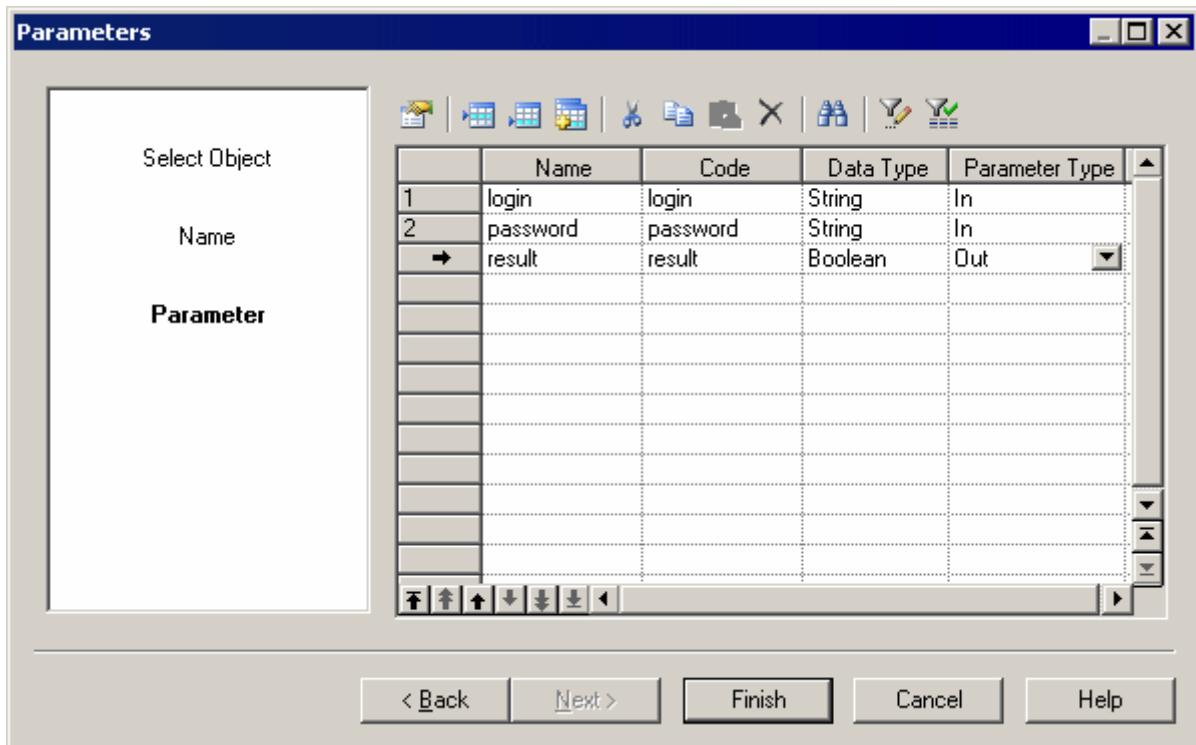
3. Click the *Create* tool to the right of the new field to open a wizard to choose an operation:



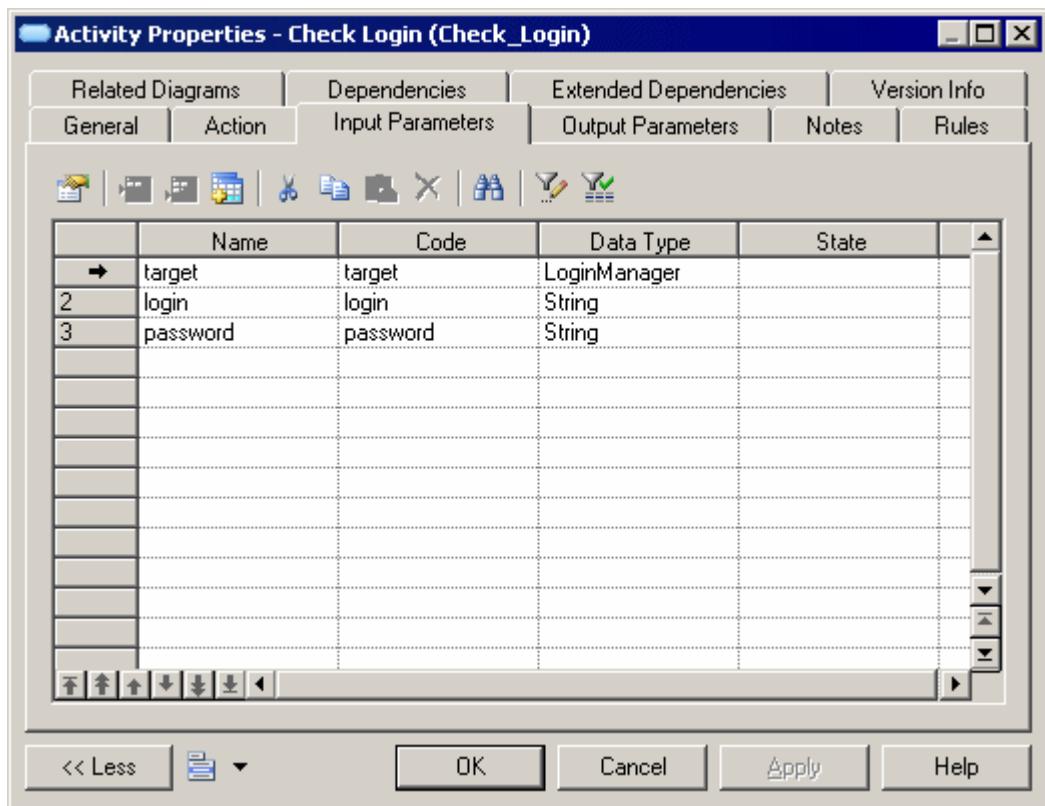
4. You can choose an existing classifier or activity, or select to create one. Select **New Class**, and then click **Next**:



5. Specify a name for the class and for the operation that you want to create, and then click *Next*:

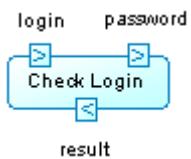


6. Create two input and one output parameter for the operation, and then click *Finish*. The property sheet of the new operation opens to allow you to further specify the operation. When you are finished, click *OK* to return to the activity property sheet and click the *Input Parameters* tab to view the parameters you have created:



Note that, in addition to the two input parameters, PowerDesigner has created a third, called "target", with the type of the new class.

7. Click **OK** to save the changes and return to the diagram:



The activity now displays its two input and one output parameter (the target parameter is hidden by default). The class and operation that you have created are available in the Browser for further use.

## 1.4.9.4.2 Example: Reading and Writing Variables

Variables hold temporary values that can be passed between activities. You can create and access variables using the Write Variable and Read Variables action types.

### Procedure

1. Open the property sheet of an activity and click the *Action* tab.
2. Select the appropriate action type:
  - **Read Variable** - then click the *Create* or *Select Object* tool to the right of the variable field to create or select the variable to read.
  - **Write Variable** - then click the *Create* or *Select Object* tool to the right of the variable field to create or select the variable to write to.
3. Specify the name and other properties of the variable and click *OK* to return to the activity property sheet.

### 1.4.9.4.2.1 Variable Properties

To view or edit a variable's properties, double-click its Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property              | Description  |
|-----------------------|--|
| Name/Code/<br>Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype            | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Data type             | Specifies the data type of the variable. You can choose a standard data type or specify a classifier. You can use the tools to the right of the list to create a classifier, browse the available classifiers or view the properties of the currently selected classifier.   |

| Property     | Description   |
|--------------|---|
| Multiplicity | <p>Specifies the number of instances of the variable. If the multiplicity is a range of values, it means that the number of variables can vary at run time.</p> <p>You can choose between:</p> <ul style="list-style-type: none"> <li>• * – none to unlimited</li> <li>• 0..* – zero to unlimited</li> <li>• 0..1 – zero or one</li> <li>• 1..* – one to unlimited</li> <li>• 1..1 – exactly one</li> </ul> |
| Keywords     | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.   |

### 1.4.9.5 Decomposed Activities and Sub-Activities

A decomposed activity is an activity that contains sub-activities. It is equivalent to a SubactivityState and an activity graph in UML. The decomposed activity behaves like a specialized package or container. A sub-activity can itself be decomposed into further sub-activities, and so on.

#### i Note

To display all activities in the model in the List of Activities, including those belonging to decomposed activities, click the Include Composite Activities tool.

You can decompose activities either directly in the diagram using an editable composite view or by using sub-diagrams. Sub-objects created in either mode can be displayed in both modes, but the two modes are not automatically synchronized. *Editable* composite view allows you to quickly decompose activities and show direct links between activities and subactivities, while *Read-only (Sub-Diagram)* mode favors a more formal decomposition and may be more appropriate if you decompose through many levels.

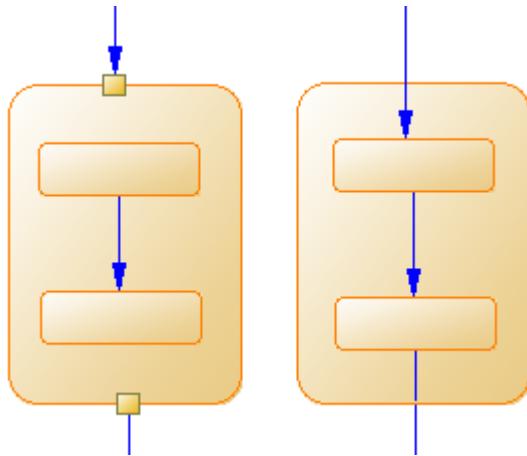
You can choose the mode for viewing composite activities on a per-object basis, by right-clicking the symbol and selecting the desired mode from the  *Composite View* menu.

You cannot create a package or any other UML diagram type in a decomposed activity, but you can use shortcuts to packages.

### Working in Editable Composite View Mode

You can decompose an activity and create sub-activities within it simply by creating or dragging another activity onto its symbol. You can resize the parent symbol as necessary and create any number of sub-activities inside it. You can decompose a sub-activity by creating or dragging another activity onto its symbol, and so on.

Flows can link activities at the same level, or can link activities in the parent diagram with sub-activities in the Live Composite View:

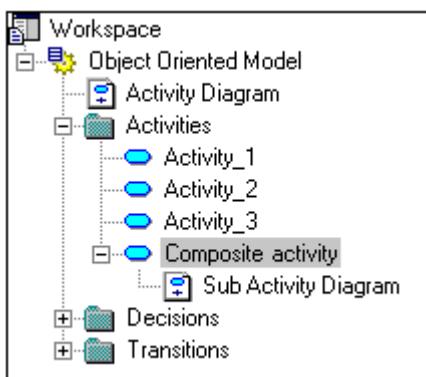


## Converting an Atomic Activity to a Decomposed Activity

You can convert an atomic activity to a decomposed activity in any of the following ways:

- Press **Ctrl** and double-click the activity symbol (this will open the sub-activity directly)
- Open the property sheet of the activity and, on the *General* tab, select the *Decomposed Activity* radio button
- Right-click the activity and select *Decompose Activity* from the contextual menu

When you create a decomposed activity, a sub-activity diagram, which is empty at first, is added below its entry in the browser:



To open a sub-activity diagram, press **Ctrl** and double-click on the decomposed activity symbol, or double-click the appropriate diagram entry in the Browser.

You can add objects to a sub-activity diagram in the same way as you add them to an activity diagram. Any activities that you add to a sub-activity diagram will be a part of its parent decomposed activity and will be listed under the decomposed activity in the Browser.

You can create several sub-activity diagrams within a decomposed activity, but we recommend that you only create one unless you want to design exception cases, such as error management.

**i Note**

You can locate any object or any diagram in the Browser tree view from the current diagram window. To do so, right-click the object symbol, or the diagram background and select ► *Edit* ► *Find in Browser* ▶.

### 1.4.9.5.1 Converting an Activity Diagram to a Decomposed Activity

You can convert an activity diagram to a decomposed activity using the Convert Diagram to Activity wizard. The conversion option is only available once objects have been created in the diagram. By converting a diagram to a decomposed activity, you can then use the decomposed activity in another activity diagram.

#### Procedure

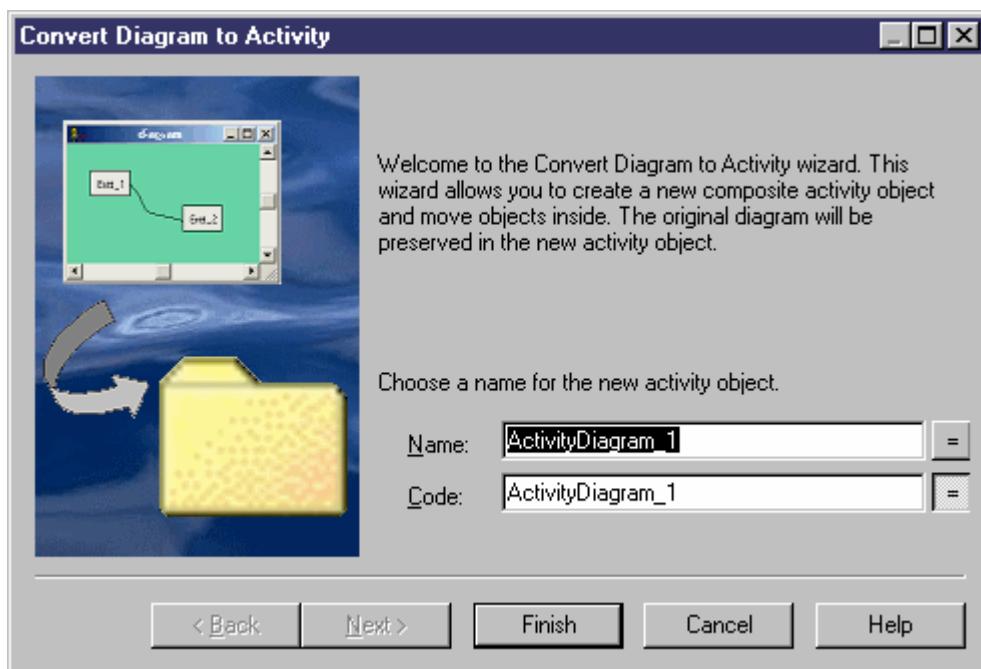
1. Right-click the diagram node in the Browser and select Convert to Composite Activity from the contextual menu.

or

Right-click the diagram background and select ► *Diagram* ► *Convert to Composite Activity* ▶ from the contextual menu.

or

Select ► *Tools* ► *Convert to Composite Activity* ▶.



2. Specify a name and a code in the Convert Diagram to Activity page, and then click Next to open the Selecting Objects to Move page.
3. Select the activities that you want to move to the new decomposed activity diagram. Activities that you select will be moved in the Browser to under the new decomposed activity. Those that you do not select will remain in their present positions in the Browser and will be represented in the new sub-activity diagram as shortcuts.
4. Click Finish to exit the wizard. The new decomposed activity and its sub-activity diagram will be created, and any objects selected to be moved will now appear beneath the decomposed object in the Browser

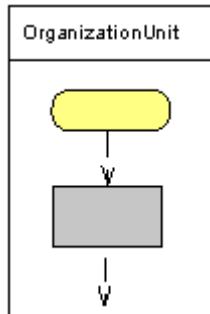
## 1.4.10 Organization Units (OOM)

An organization unit can represent a company, a system, a service, an organization, a user or a role, which is responsible for an activity. In UML, an organization unit is called a swimlane, while in the OOM, "swimlane" refers to the symbol of the organization unit.

### Note

To enable the display of organization unit swimlanes, select  [Tools > Display Preferences](#), and select the *Organization unit swimlane* checkbox on the [General](#) page, or right-click in the diagram background and select [Enable Swimlane Mode](#).

An organization unit can be created in an activity diagram and can contain any of the other activity diagram objects:



### 1.4.10.1 Creating an Organization Unit

Create an organization unit to show the participant responsible for the execution of activities.

In order to add Organization Unit Swimlanes to your diagrams, you must select  [Tools > Display Preferences](#) and select the *Organization Unit Swimlanes* checkbox.

- Use the *Organization Unit Swimlane* tool in the Toolbox. Click in or next to an existing swimlane or pool of swimlanes to add a swimlane to the pool. Click in space away from existing swimlanes to create a new pool.

- Select to access the List of Organization Units, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select .

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.4.10.2 Organization Unit Properties

To view or edit an organization unit's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property            | Description   |
|---------------------|---|
| Name/Code/Comment   | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field.  |
| Stereotype          | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.<br><br>An organization unit has the following predefined stereotypes: <ul style="list-style-type: none"> <li>• Role – specifies a role a user plays</li> <li>• User</li> <li>• Group – specifies a group of users</li> <li>• Company</li> <li>• Organization – specifies an organization as a whole</li> <li>• Division – specifies a division in a global structure</li> <li>• Service – specifies a service in a global structure</li> </ul> |
| Parent organization | Specifies another organization unit as the parent to this one.<br><br>For example, you may want to describe an organizational hierarchy between a department Dpt1 and a department manager DptMgr1 with DptMgr1 as the parent organization of Dpt1.<br><br>The relationship between parent and child organization units can be used to group swimlanes having the same parent (see <a href="#">Grouping Swimlanes [page 165]</a> ).   |
| Keywords            | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.   |

### 1.4.10.3 Attaching Activities to Organization Units

Attach activities to organization units to graphically assign responsibility for them. When activities are attached to an organization unit displayed in a swimlane, the organization unit name is displayed in the Organization Unit list of their property sheets.

You attach activities to an organization unit by creating them in (or moving existing ones into) the required swimlane. Alternately, you can select an organization unit name from the Organization Unit list of the activity property sheet, and click **OK** to attach it.

To detach activities from an organization unit, drag them outside the swimlane or select <None> in the activity property sheet.

### 1.4.10.4 Displaying a Committee Activity

A committee activity is a decomposed activity whose sub-activities are managed by several organization units.

#### Procedure

1. Open the property sheet of a decomposed activity.
2. Select *Committee Activity* from the Organization Unit list and click **OK**.

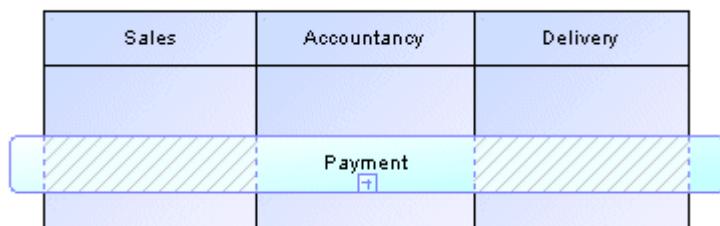
This value is only available for decomposed activities.

3. In the diagram, resize the decomposed activity symbol to cover all the appropriate swimlanes.

The symbol background color changes on the swimlanes depending on whether each is responsible for sub-activities.

#### Results

In the following example, all sub-activities of Payment are managed in the Accountancy organization unit:



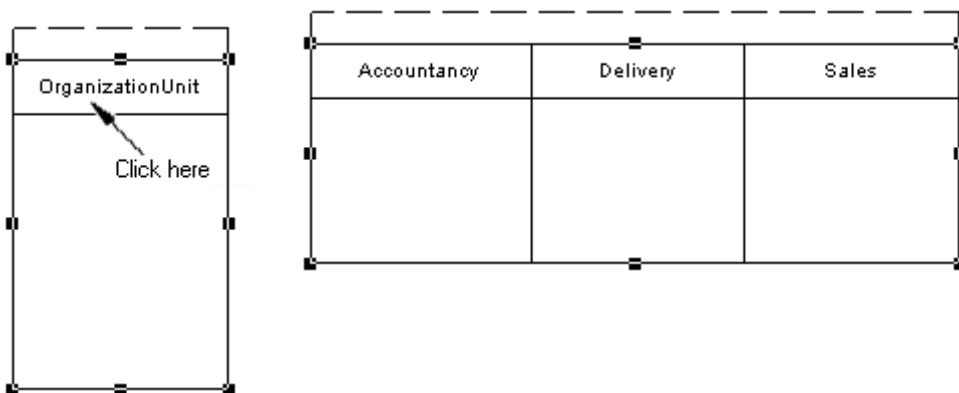
The symbol background of the committee activity is lighter and hatched on Sales and Delivery since they do not:

- Manage any sub-activities
- Have any symbol in the sub-activity diagram

Note that this display does not appear in composite view mode.

### 1.4.10.5 Moving, Resizing, Copying, and Pasting Swimlanes

Each group of one or more swimlanes forms a pool. You can create multiple pools in a diagram, and each pool is generally used to represent a separate organization. To select an individual swimlane in a pool, click its header. To select a pool, click any of its swimlanes or position the cursor above the pool, until you see a vertical arrow pointing to the frame, then click to display the selection frame.

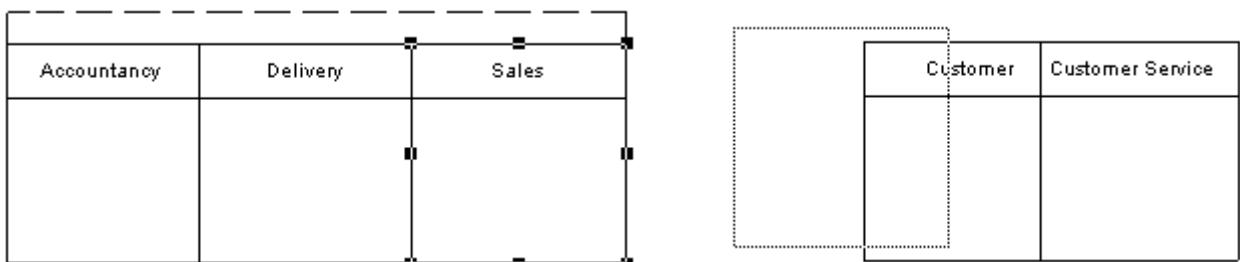


#### i Note

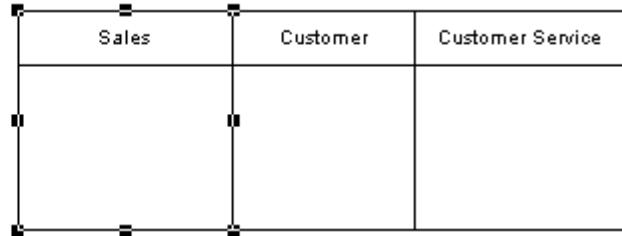
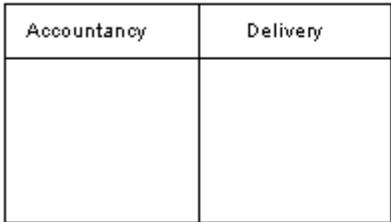
The auto-layout function is unavailable with organization units displayed as swimlanes.

If you move a swimlane or pool within the same diagram, all symbols inside the swimlane(s) are moved at the same time (even if some elements are not formally attached). If you move or copy a swimlane or pool to another diagram, the symbols inside the swimlane(s) are not copied.

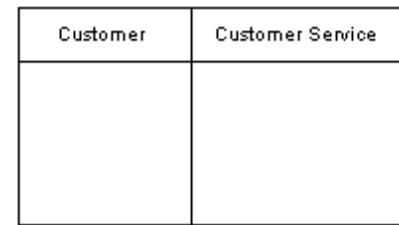
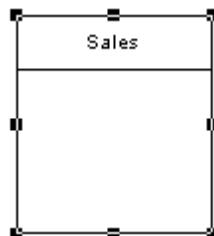
If a swimlane is dropped on or near another swimlane or pool, it joins the pool. In the following example, Sales forms a pool with Accountancy and Delivery:



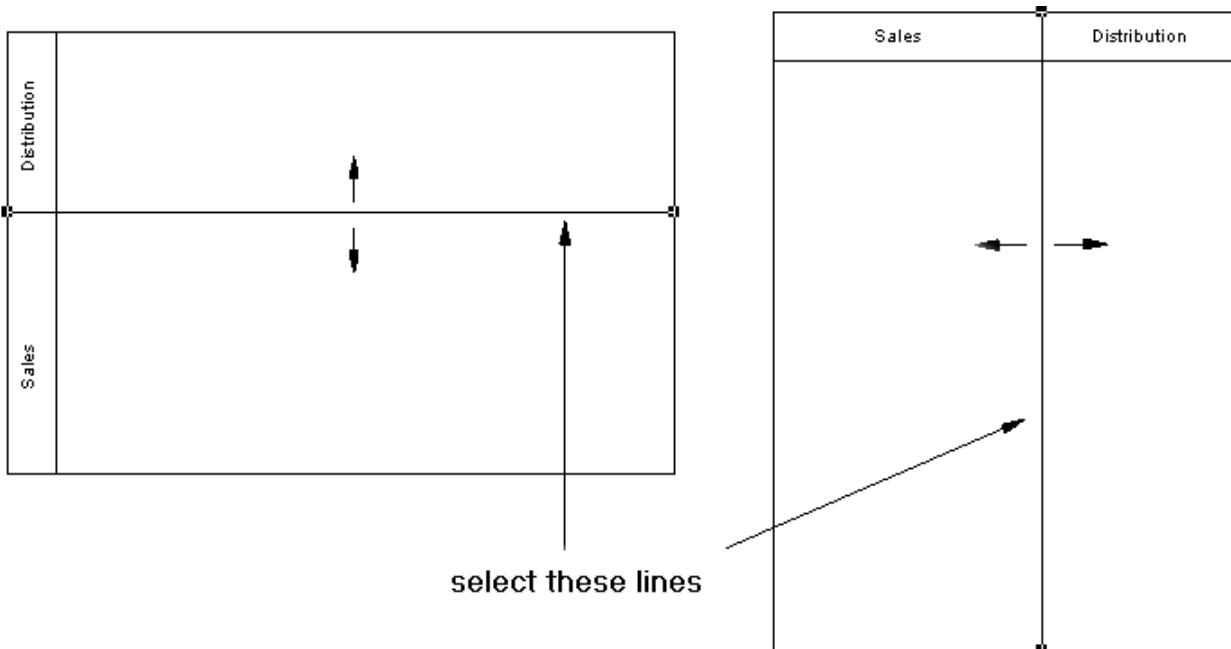
It is moved to another pool containing Customer and Customer Service



If the moved swimlane is dropped away from another swimlane or pool, it forms a new pool by itself:



You can resize swimlanes within a pool by clicking the dividing line between them and dragging it. You can resize a pool by selecting one of the handles around the pool, and dragging it into any direction. Any other pools your diagram may contain may also be resized to preserve the diagram layout.



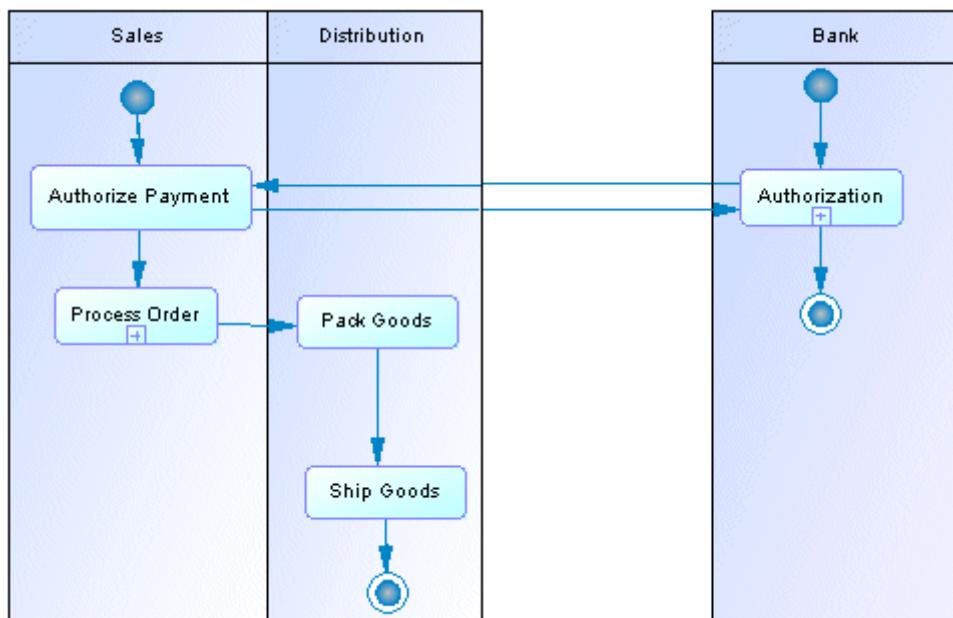
When you change the width or height of an individual swimlane, all activity symbols attached to the swimlane keep their position.

## 1.4.10.6 Creating Links Between Pools of Swimlanes

Create links between pools or between activities in separated pools to represent interactions between them.

To create links between pools of swimlanes, simply click the *Flow* tool in the Toolbox and drag a flow from one activity in a pool to another in a different pool or from one pool to another.

In the following example, flows pass between Authorize Payment in the Sales swimlane in one pool and Authorization in the Bank swimlane in another pool:



**i Note**

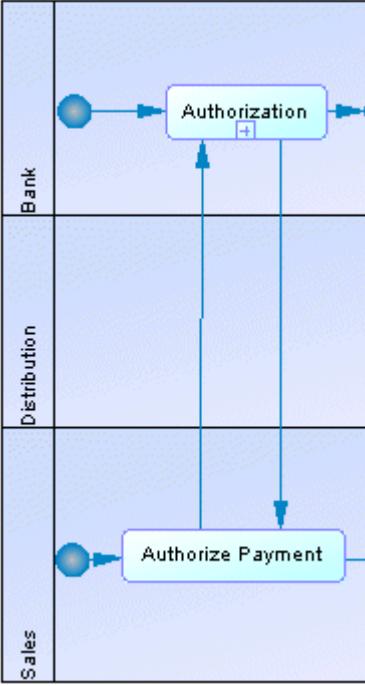
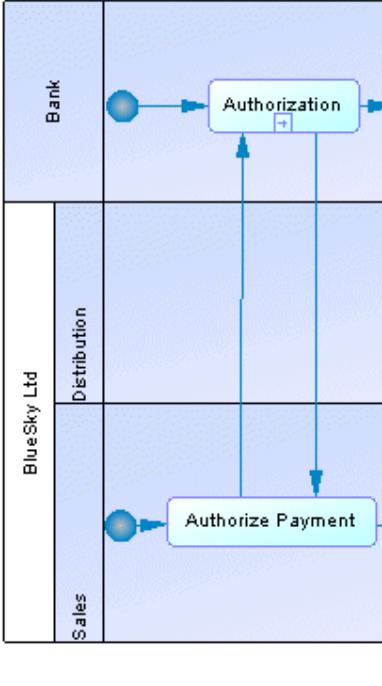
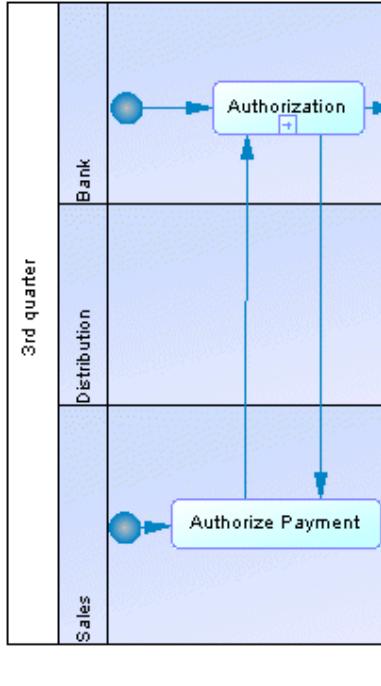
Such links between activities in separate pools are not visible when the swimlanes are not in composite view mode.

## 1.4.10.7 Grouping Swimlanes

Group organization unit swimlanes within a pool to organize them under a common parent or user-defined name.

To group swimlanes within a pool, select the pool, then right-click it and select ► *Swimlane Group Type* ▾, and then:

- *By Parent* - to assign the name of the immediate common parent for the group
- *User-Defined* - to assign a name of your choice for the group. Then, you must select at least two attached swimlanes, and select ► *Symbol* ▶ *Group Symbols* ▾ from the menu bar to display a default name that you can modify.

| No Group   | Parent Group   | User-Defined Group  |
|--|--|---|
| The three swimlanes are in a pool, without grouping:                               | Sales and Distribution are grouped by their parent:                                | The pool is assigned a user-defined group named 3rd quarter:                        |
|  |  |  |

To ungroup swimlanes, select *Ungroup Symbols* from the pool contextual menu or Select ► *Symbol* ► *Ungroup Symbols* ▾.

#### 1.4.10.8 Changing the Orientation and Format of Swimlanes

You can change the orientation of swimlanes so that they run vertically (from top to bottom) or horizontally (from left to right). All swimlanes in a diagram must have the same orientation.

#### Context

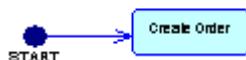
Select ► *Tools* ► *Display Preferences* ▾, select the appropriate radio button in the *Organization unit swimlane* groupbox, and click *OK*.

## 1.4.11 Starts and Ends (OOM)

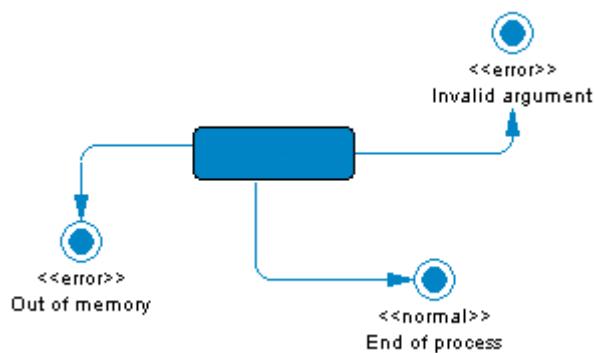
A start is a starting point of the flow represented in the diagram, and an end is a termination point of the flow.

A start can be created in the following diagrams:

- Activity Diagram - one or more per diagram
- Statechart Diagram - one or more per diagram
- Interaction Overview Diagram - one only per diagram



You can create several ends within the same diagram to show divergent end cases, such as error scenarios:



You should not use the same start or end in two diagrams, and you cannot create shortcuts of starts or ends.

If there is no end, the diagram contains an endless activity. However, a decomposed activity must always contain at least one end.

### i Note

The start is compared and merged when merging models to ensure that there is no additional start in decomposed activities.

### 1.4.11.1 Creating a Start or an End

You can create a start and an end from the Toolbox, Browser, or *Model* menu.

- Use the *Start* or *End* tool in the Toolbox.
- Select **Model > Starts** or **Model > Ends**, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Start** or **New > End**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.4.11.2 Start and End Properties

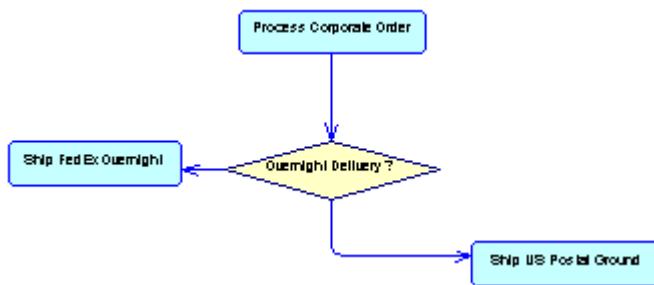
To view or edit a start or end's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property              | Description  |
|-----------------------|--|
| Name/Code/<br>Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype            | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Termination           | [ends only] Specifies whether the end is the termination of the entire activity or simply one possible flow.   |
| Keywords              | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## 1.4.12 Decisions (OOM)

A decision specifies which path to take, when several paths are possible. A decision can have one or more input flows and one or more output flows, each labeled with a distinct guard condition, which must be satisfied for its associated flow to execute some action. Your guard conditions should avoid ambiguity by not overlapping, yet should also cover all possibilities in order to avoid process freeze.



A decision can be created in the following diagrams:

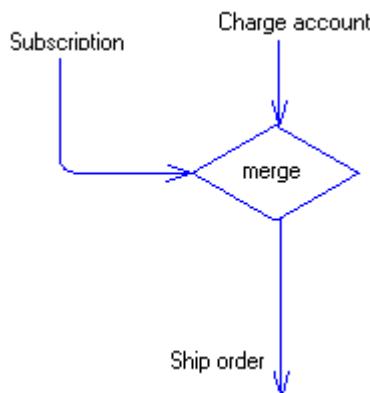
- Activity Diagram
- Interaction Overview Diagram

A decision can represent:

- A conditional branch: one input flow and several output flows. You can display a condition on the decision symbol in order to factorize the conditions attached to the flows:

| Without Condition on Symbol   | With Condition on Symbol  |
|---|---|
| <p>In this example, the control flow passes to the left if the age given in the application form is &lt;18, to the right if the age is &gt;65, and takes the another route if the age is not mentioned:</p> <pre> graph TD     A[Type in application form] --&gt; B{Conditional branch}     B -- "&lt;18" --&gt; C     B -- "&gt;65" --&gt; D     B -- "[else]" --&gt; E   </pre> | <p>In this, the condition <math>\text{Total} * \text{NB} + \text{VAT} &gt; 10.000</math> is entered in the Condition tab in the decision property sheet, and True and False are entered in the Condition tabs of the flows:</p> <pre> graph TD     A[Type in application form] --&gt; B{Total * NB + VAT &gt; 10.000}     B -- "[True]" --&gt; C     B -- "[False]" --&gt; D   </pre> |

- A merge: several input flows and one output flow. In the following example, the Subscription and Charge account flows merge to become the Ship order flow:



A decision allows you to create complex flows, such as:

- if ... then ... else ...
- switch ... case ...
- do ... while ...
- loop
- for ... next ...

### Note

You cannot attach two flows of opposite directions to the same corner of a decision symbol.

## 1.4.12.1 Creating a Decision

You can create a decision from the Toolbox, Browser, or *Model* menu.

- Use the *Decision* tool in the Toolbox
- Select  to access the List of Decisions, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select .

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.4.12.2 Decision Properties

To view or edit a decision's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property              | Description  |
|-----------------------|--|
| Name/Code/<br>Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype            | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Type                  | Calculated read-only value showing the type of the decision: <ul style="list-style-type: none"><li>• Incomplete - No input, no output flow, or only one input and one output flow.</li><li>• Conditional branch – One input and multiple outputs.</li><li>• Merge - Multiple inputs and one output.</li></ul>  |
| Keywords              | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

The following tabs are also available:

- *Condition* - contains the following properties:

| Property             | Description  |
|----------------------|--|
| Alias                | Specifies a short name for the condition, to be displayed next to its symbol in the diagram.   |
| Condition (text box) | Specifies a condition to be evaluated to determine how the decision should be traversed. You can enter any appropriate information in this box, as well as open, insert and save text files. You can open the Condition tab by right-clicking the decision symbol, and selecting Condition in the contextual menu. |

### 1.4.13 Synchronizations (OOM)

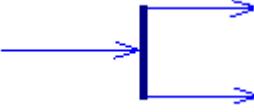
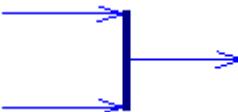
A synchronization enables the splitting or synchronization of control between two or more concurrent actions.

A synchronization can be created in the following diagrams:

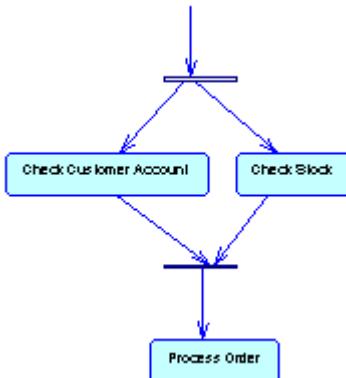
- Activity Diagram
- Statechart Diagram
- Interaction Overview Diagram

Synchronizations are represented as horizontal or vertical lines. To change the orientation of the symbol, right-click it and select *Change to Vertical* or *Change to Horizontal*.

A synchronization can be either a:

| Fork  | Join  |
|---|---|
| <p>Splits a single input flow into several output flows executed in parallel:</p>  | <p>Merges multiple input flows into a single output flow. All input flows must reach the join before the single output flow continues:</p>  |

In the following example, the flow entering the first synchronization is split into two flows, which pass through Check Customer Account and Check Stock. Then both flows are merged into a second synchronization giving a single flow, which leads to Process Order:



### 1.4.13.1 Creating a Synchronization

You can create a synchronization from the Toolbox, Browser, or *Model* menu.

- Use the Synchronization tool in the Toolbox.
- Select **Model > Synchronizations** to access the List of Synchronizations, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Synchronization**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

### 1.4.13.2 Synchronization Properties

To view or edit a synchronization's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property              | Description  |
|-----------------------|--|
| Name/Code/<br>Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype            | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Timeout               | Defines a timeout limit for waiting until all transitions end. It is empty when the value = 0.   |

| Property | Description   |
|----------|---|
| Type     | [read-only] Calculates the form of the synchronization: <ul style="list-style-type: none"> <li>Incomplete - No or only one input and output transition.</li> <li>Conditional branch – One input and multiple outputs.</li> <li>Merge - Multiple inputs and one output.</li> </ul> |
| Keywords | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.   |

## 1.4.14 Flows (OOM)

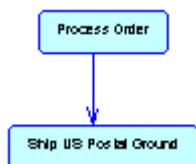
A flow is a route the control flow takes between objects . The routing of the control flow is made using guard conditions defined on flows. If the condition is true, the control is passed to the next object.

A flow can be created in the following diagrams:

- Activity Diagram
- Interaction Overview Diagram

A flow between an activity and an object node indicates that the execution of the activity puts an object in a specific state. When a specific event occurs or when specific conditions are satisfied, the control flow passes from the activity to the object node. A flow from an object node to an activity means that the activity uses this specific state in its execution. In both cases, the flow is represented as a simple arrow.

In the following example the flow links Process Order to Ship US Postal Ground:



A flow can link shortcuts. A flow accepts shortcuts on both extremities to prevent it from being automatically moved when a process is to be moved. In this case, the process is moved and leaves a shortcut, but contrary to the other links, the flow is not moved. Shortcuts of flows do not exist, and flows remain in place in all cases.

The following rules apply:

- Reflexive flows (same source and destination process) are allowed on processes.
- Two flows between the same source and destination objects are permitted, and called parallel flows.

### i Note

When flows are compared and merged by the Merge Model feature, they are matched by trigger event first, and then by their calculated name. When two flows match, the trigger actions automatically match because there cannot be more than one trigger action.

## 1.4.14.1 Creating a Flow

You can create a flow from the Toolbox, Browser, or *Model* menu.

- Use the *Flow/Resource Flow* tool in the Toolbox
- Select ► *Model* ► *Flows* ▾ to access the List of Flows, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select ► *New* ▾ *Flow* ▾.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.4.14.2 Flow Properties

To view or edit a flow's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property             | Description  |
|----------------------|--|
| Name/Code/Comment    | Identify the object. The name and code are read-only. You can optionally add a comment to provide more detailed information about the object.  |
| Stereotype           | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Source / Destination | Specify the objects that the flow leads from and to. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object. You can also open the property sheet of the source and destination objects by clicking the buttons in the top section of the flow property sheet.  |
| Flow type            | You can enter your own type of flow in the list, or choose one of the following values: <ul style="list-style-type: none"><li>• Success - defines a successful flow</li><li>• Timeout - defines the occurrence of a timeout limit</li><li>• Technical error</li><li>• Business error</li><li>• Compensation - defines a compensation flow</li></ul> The flow type is unavailable if you associate an event with the flow on the Condition tab. |
| Weight               | Specifies the number of objects consumed on each traversal.  |
| Keywords             | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

### i Note

You can view input and output flows of a process from its property sheet by clicking the *Input Flows* and *Output Flows* sub-tabs of the *Dependencies* tab.

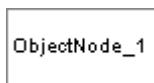
The following tabs are also available:

- *Parameters* - lists the parameters that are passed along the flow. The list is automatically completed if you draw the flow between two activity parameters.
- *Transformation* - specifies a data transformation to apply to input tokens. For example, it could extract a single attribute value from an input object.

## 1.4.15 Object Nodes (OOM)

An object node is the association of an object (instance of a class) and a state. It represents an object in a particular state.

Its symbol is a rectangle as shown below:



An object node can be created in the following diagrams:

- Activity Diagram

The same object can evolve after several actions defined by activities, have been executed. For example, a document can evolve from the state initial, to draft, to reviewed, and finally turn into a state approved.

You can draw flows from an activity to an object node and inversely:

- A flow from an activity to an object node - means that the execution of the activity puts the object in a specific state. It represents the result of an activity
- A flow from an object node to an activity - means that the activity uses this specific state in its execution. It represents a data flow between them

When an activity puts an object in a state and this object is immediately reused by another activity, it shows a transition between two activities with some data exchange, the object node representing the data exchange.

For example, the object nodes Order approved and Invoice edited, are linked to the classes Order and Invoice, which are represented in a separate class diagram:



### 1.4.15.1 Creating an Object Node

You can create an object node from the Toolbox, Browser, or *Model* menu.

- Use the *Object Node* tool in the Toolbox.

- Select to access the List of Object Nodes, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select .
- Drag and drop a classifier from the Browser onto an activity diagram. The new object node will be linked to and display the name of the classifier.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.4.15.2 Object Node Properties

To view or edit an object node's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Data type         | Specifies the data type of the object node. You can use the tools to the right of the list to create a classifier, browse the complete tree of available classifiers or view the properties of the currently selected classifier.  |
| State             | Specifies the state of the object node. You can type the name of a state here or, if a classifier has been specified as the data type, select one of its states from the list. You can use the tools to the right of the list to create a state, browse the complete tree of available states or view the properties of the currently selected state.  |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## 1.4.16 States (OOM)

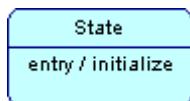
A state represents a situation during the life of a classifier that is usually specified by conditions. It can also be defined as the situation of a classifier waiting for events. Stability and duration are two characteristics of a state.

A state can be created in the following diagrams:

- Statechart Diagram

A state can be atomic or decomposed:

- An atomic state does not contain sub-states, and has the following symbol:



- A decomposed state contains sub-states, which are represented in a sub-diagram, and has the following symbol:



For more information on decomposed states, see [Decomposed States and Sub-states \[page 179\]](#).

Several states in a statechart diagram correspond to several situations during the life of the classifier.

Events and condition guards on output transitions define the stability of a state. Some actions can be associated with a state, especially when the object enters or exits the state. Some actions can also be performed when events occur inside the state; those actions are called internal transitions, they do not cause a change of state.

You cannot decompose shortcuts of states.

## Drag a Class, Use Case or Component in a Statechart Diagram

The statechart diagram describes the behavior of a classifier. To highlight the relationship between a classifier and a state, you can define the context classifier of a state using the Classifier list in the state property sheet. This links the state to a use case, a component or a class.

You can also move, copy and paste, or drag a class, use case or component and drop it into a statechart diagram to automatically create a state associated with the element that has been moved.

## 1.4.16.1 Creating a State

You can create a state from the Toolbox, Browser, or *Model* menu.

- Use the *State* tool in the Toolbox.
- Select ► *Model* ► *States* ▾ to access the List of States, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select ► *New* ► *State* ▾.
- Drag and drop a class, use case or component into the diagram.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.4.16.2 State Properties

To view or edit a state's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Classifier        | Classifier linked to the state. It can be a use case, a class or a component. When a classifier is selected, it is displayed in between brackets after the state name in the Browser. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.   |
| Composite status  | If you select the Decomposed state option, the state becomes a decomposed state. If you select the Atomic state option, the state becomes an atomic state, and all its child objects are deleted   |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## Actions Tab

You can specify a set of internal actions on an atomic or decomposed state on the *Actions* tab. These represent actions performed within the scope of the state when some events occur. You can create and define the properties

of the action from the *Actions* tab, or double-click the arrow at the beginning of a line to display the action property sheet.

**i Note**

You can open the Actions tab by right clicking the state symbol in the diagram, and selecting *Actions* from the contextual menu.

For more information on actions, see [Actions \(OOM\) \[page 188\]](#).

## Deferred Events Tab

The *Deferred Events* tab contains an *Add Objects* tool that allows you to add already existing events but not to create new events. This list is similar to the list of Business Rules that only reuse elements and does not create them.

The difference between an event and a deferred event is that an event is always instantaneous and dynamically handled by a state, whereas a deferred event is an event that occurs during a particular state in the object life cycle but it is not directly used up by the state.

A deferred event occurs in a specific state, is then handled in a queue, and is triggered by another state of the same classifier later.

## Sub-States Tab

The *Sub-States* tab is displayed when the current state is decomposed in order to display a list of child states. You can use the *Add a row* and *Delete* tools to modify the list of child states. The *Sub-States* tab disappears if you change the current state to atomic because this action deletes the children of the state.

### 1.4.16.3 Decomposed States and Sub-states

A decomposed state is a state that contains sub-states. The decomposed state behaves like a specialized package or container. A sub-state can itself be decomposed into further sub-states, and so on.

**i Note**

To display all states in the model in the List of States, including those belonging to decomposed states, click the *Include Composite States* tool.

You can decompose states either directly in the diagram using an editable composite view or by using sub-diagrams. Sub-objects created in either mode can be displayed in both modes, but the two modes are not automatically synchronized. *Editable* composite view allows you to quickly decompose states and show direct links between states and substates, while *Read-only (Sub-Diagram)* mode favors a more formal decomposition and may be more appropriate if you decompose through many levels.

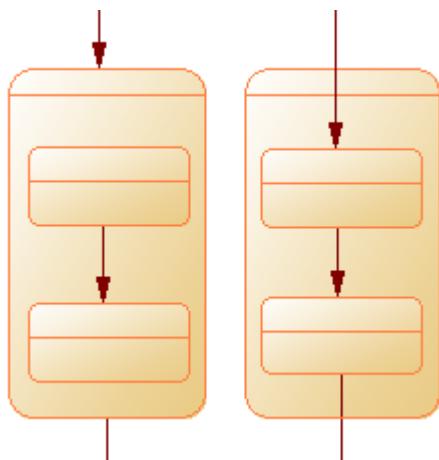
You can choose how to view composite states on a per-object basis, by right-clicking the symbol and selecting the desired mode from the  *Composite View* menu.

You cannot create a package or any other UML diagram type in a decomposed state, but you can use shortcuts to packages.

## Working in Editable Composite View Mode

You can decompose a state and create substates within it simply by creating or dragging another state onto its symbol. You can resize the parent symbol as necessary and create any number of substates inside it. You can decompose a substate by creating or dragging another state onto its symbol, and so on.

Transitions can link states at the same level, or can link states in the parent diagram with sub-states in the Editable Composite View mode:

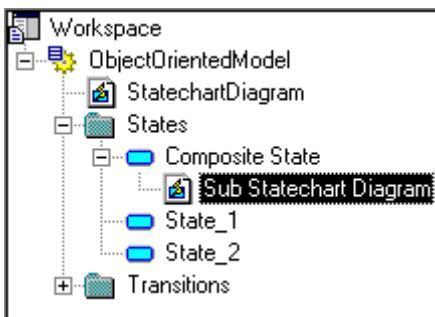


## Working with Sub-State Diagrams

You can convert an atomic state to a decomposed state in any of the following ways:

- Press **Ctrl** and double-click the state symbol (this will open the sub-state directly)
- Open the property sheet of the state and, on the *General* tab, select the *Decomposed State* radio button
- Right-click the state and select *Decompose State*

When you create a decomposed state, a sub-state diagram, which is empty at first, is added below its entry in the browser:



To open a sub-state diagram, press **[Ctrl]** and double-click on the decomposed state symbol, or double-click the appropriate diagram entry in the Browser.

You can add objects to a sub-state diagram in the same way as you add them to an state diagram. Any states that you add to a sub-state diagram will be a part of its parent decomposed state and will be listed under the decomposed state in the Browser.

You can create several sub-state diagrams within a decomposed state, but we recommend that you only create one unless you want to design exception cases, such as error management.

#### **i Note**

You can locate any object or any diagram in the Browser tree view from the current diagram window. To do so, right-click the object symbol, or the diagram background and select **► Edit ► Find in Browser**.

### **1.4.16.3.1 Converting a Statechart Diagram to a Decomposed State**

You can convert a statechart diagram to a decomposed state using the Convert Diagram to State wizard. The conversion option is only available once objects have been created in the diagram. By converting a diagram to a decomposed state, you can then use the decomposed state in another statechart diagram.

#### **Procedure**

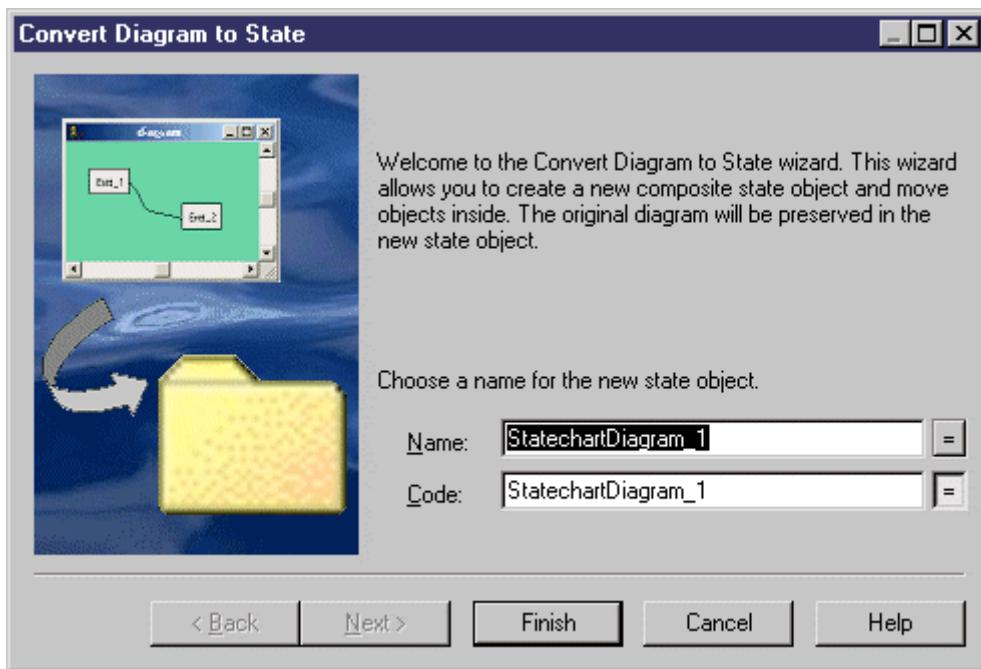
1. Right-click the diagram node in the Browser and select Convert to Decomposed State from the contextual menu.

*or*

Right-click the diagram background and select **► Diagram ► Convert to Decomposed State** from the contextual menu.

*or*

Select **► Tools ► Convert to Decomposed State**.



2. Specify a name and a code in the Convert Diagram to State page, and then click Next to open the Selecting Objects to Move page.
3. Select the states that you want to move to the new decomposed state diagram. States that you select will be moved in the Browser to under the new decomposed state. Those that you do not select will remain in their present positions in the Browser and will be represented in the new sub-state diagram as shortcuts.
4. Click Finish to exit the wizard. The new decomposed state and its sub-state diagram will be created, and any objects selected to be moved will now appear beneath the decomposed object in the Browser

### 1.4.17 Transitions (OOM)

A transition is an oriented link between states, which indicates that an element in one state can enter another state when an event occurs (and, optionally, if a guard condition is satisfied). The expression commonly used in this case is that a transition is fired.

A transition can be created in the following diagrams:

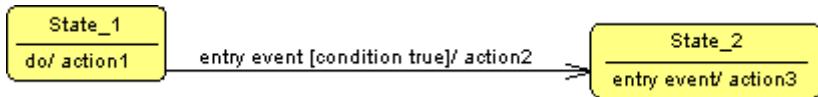
- Statechart Diagram

The statechart diagram transition is quite similar to the flow in the activity diagram, with the addition of a few properties:

- A trigger event: it is the event that triggers the transition (when you copy a transition, the trigger event is also copied)
- A trigger action: it specifies the action to execute when the transition is triggered

The activity diagram is a simplification of the statechart diagram in which the states have only one action and the transition has a triggered event corresponding to the end of the action.

The transition link is represented as a simple arrow. The associated event, the condition and the action to execute are displayed above the symbol.



The following rules apply:

- Reflexive transitions only exist on states
- A trigger event can only be defined if the source is a start or a state
- Two transitions can not be defined between the same source and destination objects (parallel transitions). The Merge Model feature forbids this.

#### **i Note**

When transitions are compared and merged by the Merge Model feature, they are matched by trigger event first, and then by their calculated name. When two transitions match, the trigger actions automatically match because there cannot be more than one trigger action.

### 1.4.17.1 Creating a Transition

You can create a transition from the Toolbox, Browser, or *Model* menu.

- Use the *Transition* tool in the Toolbox.
- Select **Model > Transitions** to access the List of Transitions, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Transition**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

### 1.4.17.2 Transition Properties

To view or edit a transition's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property              | Description  |
|-----------------------|--|
| Name/Code/<br>Comment | Identify the object. The name and code are read-only. You can optionally add a comment to provide more detailed information about the object.  |
| Stereotype            | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file. |

| Property    | Description  |
|-------------|--|
| Source      | Where the transition starts from. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.   |
| Destination | Where the transition ends on. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.   |
| Flow type   | Represents a condition that can be attached to the transition. You can choose between the following default types or create your own: <ul style="list-style-type: none"> <li>Success – Defines a successful flow</li> <li>Timeout – Defines a timeout limit</li> <li>Exception – Represents an exception case</li> </ul> |
| Keywords    | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## Trigger Tab

The *Trigger* tab contains the following properties:

| Property            | Description  |
|---------------------|--|
| Trigger event       | Specifies the event (see <a href="#">Events (OOM) [page 185]</a> ) that triggers the transition. You can click the Properties tool beside this box to display the event property sheet. It is available only for transitions coming from a state or a start and is not editable in other cases. When you define a trigger event, the inverse relationship is displayed in the Triggered Objects tab of the corresponding event property sheet. The Triggered Objects tab lists transitions that the event can trigger. |
| Event arguments     | Specifies a comma-separated list of event arguments (arg1, arg2,...).  |
| Trigger action      | Specifies the action to execute when the transition is triggered.  |
| Operation           | Read-only list that lists operations of the classifier associated with the state that is the source of the transition. It allows you to specify the action implementation using an operation. It is grayed and empty when the classifier is not a class  |
| Operation arguments | Arguments of an event defined on an operation  |

## Condition Tab

The *Condition* tab contains the following properties:

| Property             | Description  |
|----------------------|--|
| Alias                | Short name for the condition, to be displayed next to its symbol in the diagram.   |
| Condition (text box) | Specifies a condition to be evaluated to determine whether the transition should be traversed. You can enter any appropriate information in this field, as well as open, insert and save text files. |

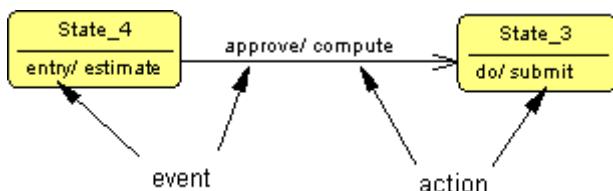
## 1.4.18 Events (OOM)

An event is the occurrence of something observable. The occurrence is assumed to be instantaneous and should not have duration.

An event can be created in the following diagrams:

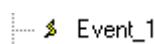
- Statechart Diagram

Events convey information specified by parameters. They are used in the statechart diagram in association with transitions: they are attached to transitions to specify which event fires the transition. They are also used in association with actions: the event can trigger the change of state of a classifier or the execution of an internal action on a state.



The same event can be shared between several transitions and actions. It is reusable by nature because it is not dependent on the context.

The event icon in the Browser is the following symbol:



## Predefined Events

You can select an event from the Trigger Event list in the action and transition property sheets. You can also select a predefined event value from the Trigger Event list if you define the event on an action.

The list of events contains the following predefined values:

- Entry: the action is executed when the state is entered
- Do: a set of actions is executed after the entry action
- Exit: the action is executed when the state is exited

## Examples

An event could be:

- A boolean expression becoming true
- The reception of a signal
- The invocation of an operation
- A time event, like a timeout or a date reached

You can display the arguments of an event in the statechart diagram.

For more information on arguments of an event, see [Defining Event Arguments \[page 187\]](#).

### 1.4.18.1 Creating an Event

You can create an event from the Browser, from the *Model* menu or from a transition property sheet.

- Select to access the List of Events, and click the *Add a Row* tool.
- Right-click the model or package in the Browser, and select .
- Double-click a transition to open its property sheet, click the *Trigger* tab, and then click the *Create* tool to the right of the *Trigger event* box.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

### 1.4.18.2 Event Properties

To view or edit an event's properties, double-click its Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <b>Code</b> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

The following tabs are also available:

- Parameters - lists the parameters of the event corresponding to the event signature (see [Parameters \(OOM\) \[page 83\]](#)).
- Dependencies - contains a Triggered Objects sub-tab that displays the actions on states and on transitions that are triggered by this event.

### 1.4.18.3 Defining Event Arguments

Event arguments are slightly different from event parameters. Event arguments are defined on the action or on the transition that receives the event, they are dependent on the particular context that follows this receipt.

It is a text field defined on the action or the transition. You can edit it and separate arguments with a comma, for example: arg1, arg2. There is no control of coherence between event parameters and event arguments in PowerDesigner.

#### Example

An event can have a parameter "person" that is for example, a person sending a request. Within the context of a transition triggered by this event, you may clearly know that this parameter is a customer, and then purposefully call it "customer" instead of "person".

## 1.4.19 Actions (OOM)

An action is a specification of a computable statement. It occurs in a specific situation and may comprise predefined events (entry, do and exit) and internal transitions.

An action can be created in the following diagrams:

- Statechart Diagram

Internal transitions can be defined on a state, they are internal to the state and do not cause a change of state; they perform actions when triggered by events. Internal transitions should not be compared to reflexive transitions on the state because the entry and exit values are not executed when the internal event occurs.

An action contains a Trigger Event property containing the specification of the event that triggers the action.



For more information on events, see [Events \(OOM\) \[page 185\]](#).

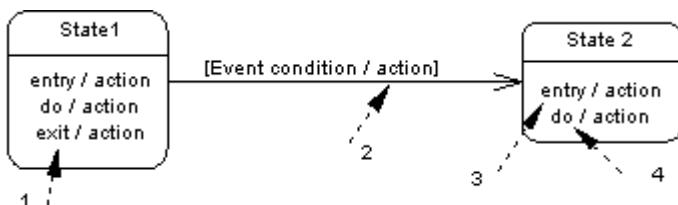
### Action on State and on Transition

In an OOM, an action is used in the statechart diagram in association with states: the action is executed in the state during entry or exit. It is also used in association with transitions: the action is executed when the transition is triggered.

In UML, the difference is that an action is displayed in interaction diagrams (in association with messages) and in statechart diagrams.

When you define an action on a state, you can define several actions without any limitation. When you define an action on a transition, there can only be one action as the transition can execute only one action. An action defined on a state can contain the event that triggers it: the action property sheet contains the event property sheet. An action defined on a transition does not contain the event that triggers it: you can only enter the action in a text field.

In the following figure, you can see actions defined on states, and actions defined on transitions together with the order of execution of actions:



The action icon in the Browser is a two-wheel symbol, it is defined within a state node but does not appear within a transition node.



### 1.4.19.1 Creating an Action

You can create an action from the property sheet of a state or transition.

- Open the *Actions* tab in the property sheet of a state, and click the *Add a Row* tool
- Open the *Trigger* tab in the property sheet of a transition, and type the action name in the *Trigger action* box

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

### 1.4.19.2 Action Properties

To view or edit an action's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |

| Property            | Description   |
|---------------------|---|
| Trigger event       | <p>Specifies the role an action plays for a state or the event that triggers its execution. You can:</p> <ul style="list-style-type: none"> <li>• Add an event to an action by choosing it from the list.</li> <li>• Add multiple events by clicking the ellipsis tool next to the list.</li> <li>• Create a new event by clicking the Create tool.</li> <li>• Select an event created in the current model or other models by clicking the Select Trigger Event tool.</li> </ul> <p>Click the Properties tool to display the event property sheet. When a trigger event is defined on an action, the inverse relationship is displayed in the Triggered Objects sub-tab of the Dependencies tab of the event property sheet (see <a href="#">Events (OOM) [page 185]</a>).</p> |
| Event arguments     | Arguments of an event defined on a state. Arguments are instances of parameters or names given to parameters in the context of executing an event. You can specify a list of event arguments (arg1, arg2,...) in this box   |
| Operation           | Read-only list that lists operations of the classifier associated with the state. It allows you to specify the action implementation using an operation. It is grayed and empty when the classifier is not a class  |
| Operation arguments | Arguments of an event defined on an operation   |
| Keywords            | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.   |

## Condition Tab

The **Condition** tab is available for actions defined on states. You can specify an additional condition on the execution of an action when the event specified by the trigger event occurs.

The Alias field allows you to enter a condition attached to an action. You can also use the text to detail the condition. For example, you can write information on the condition to execute, as well as open, insert and save any text files containing valuable information.

We recommend that you write an alias (short expression) when you use a long condition so as to display the alias instead of the condition in the diagram.

The condition of an action is displayed between brackets:



entry [timeout < 10 s]/ initialize

condition

## 1.4.20 Junction Points (OOM)

A junction point can merge and/or split several input and output transitions. It is similar to the decision in the activity diagram

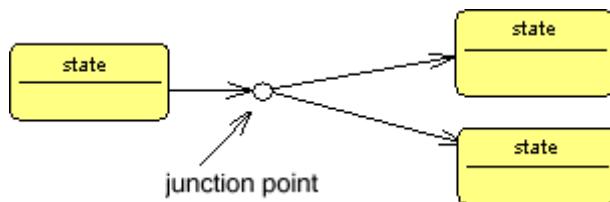
A junction point can be created in the following diagrams:

- Statechart Diagram

You are not allowed to use shortcuts of a junction point. A junction point may be dependent on event parameters if the parameters include some split or merge variables for example.

You can attach two transitions of opposite directions to the same junction point symbol.

The symbol of a junction point is an empty circle:



### 1.4.20.1 Creating a Junction Point

You can create a junction point from the Toolbox, Browser, or *Model* menu.

- Use the *Junction Point* tool in the Toolbox.
- Select **Model > Junction Points** to access the List of Junction Points, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Junction Point**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

### 1.4.20.2 Junction Point Properties

To view or edit a junction point's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Keywords          | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## 1.5 Implementation Diagrams

The diagrams in this chapter allow you to model the physical environment of your system, and how its components will be deployed. PowerDesigner provides two types of diagrams for modeling your system in this way:

- A component diagram represents your system decomposed into self-contained components or sub-systems. It can show the classifiers that make up these systems together with the artifacts that implement them, and exposes the interfaces offered or required by each component, and the dependencies between them. For more information, see [Component Diagrams \[page 192\]](#).
- A deployment diagram allows you to represent the execution environment for a project. It describes the hardware on which each of your components will run and how that hardware is connected together. For more information, see [Deployment Diagrams \[page 194\]](#).

### 1.5.1 Component Diagrams

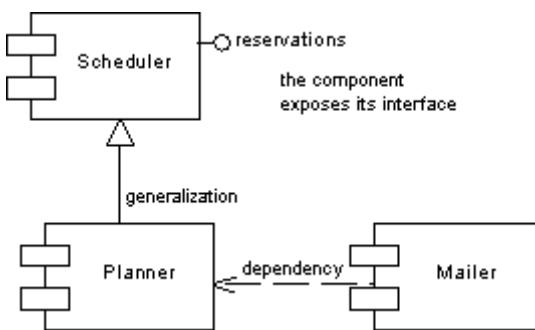
A component diagram is a UML diagram that provides a graphical view of the dependencies and generalizations among software components, including source code components, binary code components, and executable components.

#### i Note

To create a component diagram in an existing OOM, right-click the model in the Browser and select [New](#) [Component Diagram](#). To create a new model, select [File](#) [New Model](#), choose Object Oriented Model as the model type and [Component Diagram](#) as the first diagram, and then click *OK*.

For information about Java- and .NET-specific components, see [Java \[page 292\]](#) and [VB .NET \[page 376\]](#).

The following example shows relationships between components in a showroom reservation system:



Component diagrams are used to define object dependencies and relationships at a higher level than class diagrams.

Components should be designed in order to be reused for several applications, and so that they can be extended without breaking existing applications.

You use component diagrams to model the structure of the software, and show dependencies among source code, binary code and executable components so that the impact of a change can be evaluated.

A component diagram is useful during analysis and design. It allows analysts and project leaders to specify the components they need before having them developed and implemented. The component diagram provides a view of components and makes it easier to design, develop, and maintain components and help the server to deploy, catalog, and find components.

### 1.5.1.1 Component Diagram Objects

PowerDesigner supports all the objects necessary to build component diagrams.

| Object    | Tool | Symbol | Description  |
|-----------|------|--------|--|
| Component |      |        | Represents a shareable piece of implementation of a system. See <a href="#">Components (OOM) [page 196]</a> .  |
| Interface |      |        | Descriptor for the externally visible operations of a class, object, or other entity without specification of internal structure. See <a href="#">Interfaces (OOM) [page 60]</a> . |
| Port      |      |        | Interaction point between a classifier and its environment. See <a href="#">Ports (OOM) [page 44]</a> .  |
| Part      |      |        | Classifier instance playing a particular role within the context of another classifier. See <a href="#">Parts (OOM) [page 42]</a> .  |

| Object               | Tool | Symbol | Description   |
|----------------------|------|--------|---|
| Generalization       |      |        | A link between a general component and a more specific component that inherits from it and adds features to it. See <a href="#">Generalizations (OOM) [page 93]</a> .                       |
| Realization          |      |        | Semantic relationship between classifiers, in which one classifier specifies a contract that another classifier guarantees to carry out. See <a href="#">Realizations (OOM) [page 99]</a> . |
| Require Link         |      |        | Connects components to interfaces. See <a href="#">Require Links (OOM) [page 100]</a> .   |
| Assembly Connector   |      |        | Connects parts to each other. See <a href="#">Assembly and Delegation Connectors (OOM) [page 47]</a> .  |
| Delegation Connector |      |        | Connects parts to ports on the outside of components. See <a href="#">Assembly and Delegation Connectors (OOM) [page 47]</a> .  |
| Dependency           |      |        | Relationship between two modeling elements, in which a change to one element will affect the other element. See <a href="#">Dependencies (OOM) [page 96]</a> .                              |

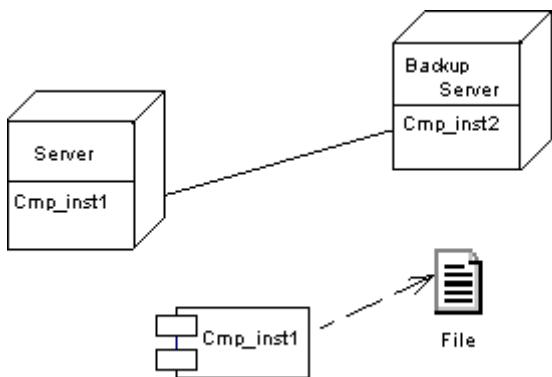
## 1.5.2 Deployment Diagrams

A deployment diagram is a UML diagram that provides a graphical view of the physical configuration of run-time elements of your system.

### i Note

To create a deployment diagram in an existing OOM, right-click the model in the Browser and select [New](#) [Deployment Diagram](#). To create a new model, select [File](#) [New Model](#), choose Object Oriented Model as the model type and [Deployment Diagram](#) as the first diagram, and then click [OK](#).

The deployment diagram provides a view of nodes connected by communication links. It allows you to design nodes, file objects associated with nodes that are used for deployment, and relationships between nodes. The nodes contain instances of components that will be deployed into and execute upon database, application or web servers.



Deployment diagrams are used for actual deployment of components into servers. A deployment can represent the ability to use instances.

You use the deployment diagram to establish the link to the physical architecture. It is suitable for modeling network topologies, for instance.

You can build a deployment diagram to show the following views, from a high level architecture that describes the material resources and the distribution of the software in these resources, to final complete deployment into a server:

- Identify the system architecture: use nodes and node associations only
- Identify the link between software and hardware: use component instances, split up their route, identify and select the servers
- Deploy components into the servers: include some details, add physical parameters

### 1.5.2.1 Deployment Diagram Objects

PowerDesigner supports all the objects necessary to build deployment diagrams.

| Object             | Tool | Symbol | Description  |
|--------------------|------|--------|--|
| Node               |      |        | Physical element that represents a processing resource, a physical unit (computer, printer, or other hardware units). See <a href="#">Nodes (OOM) [page 203]</a> . |
| Component instance |      |        | Instance of a deployable component that can run or execute on a node. See <a href="#">Component Instances (OOM) [page 205]</a> .                                   |

| Object           | Tool | Symbol | Description  |
|------------------|------|--------|--|
| Node association |      |        | An association between two nodes means that the nodes communicate to each other. See <a href="#">Node Associations (OOM) [page 209]</a> .                      |
| Dependency       |      |        | Relationship between two modeling elements, in which a change to one element will affect the other element. See <a href="#">Dependencies (OOM) [page 96]</a> . |

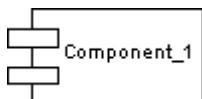
### 1.5.3 Components (OOM)

A component is a physical, replaceable part of a system that packages implementation, conforms to and provides the realization of a set of interfaces. It can represent a physical piece of implementation of a system, like software code (source, binary or executable), scripts, or command files. It is an independent piece of software developed for a specific purpose but not a specific application. It may be built up from the class diagram and written from scratch for the new system, or it may be imported from other projects and third party vendors.

A component can be created in the following diagrams:

- Component Diagram

The symbol of the component is as follows:



A component provides a 'black box' building block approach to software construction. For example, from the outside, a component may show two interfaces that describe it, whereas from the inside, it would reflect both interfaces realized by a class, both operations of the interfaces being the operations of the class.

A component developer has an internal view of the component: its interfaces and implementation classes, whereas one who assembles components to build another component or an application only has the external view (the interfaces) of these components.

A component can be implemented in any language. In Java, you can implement EJB, servlets, and JSP components, for example.

For more information on other types of components: EJB, servlets, JSP and ASP.NET, see [Java \[page 292\]](#) and [VB .NET \[page 376\]](#).

If you start developing a component with classes and interfaces in an OOM and you later want to store them in a database, it is possible to create a manual mapping of objects so that OOM objects correspond to PDM objects. Similarly, if you have an existing OOM and an existing PDM and both models must be preserved; you can handle the link between the object-oriented environment and the physical database through the object to relational mapping. Using this mapping, you can make your components communicate to each other and evolve in an object environment, as well as retrieve data stored in a database.

For more information on O/R Mapping, see [Object/Relational \(O/R\) Mapping \[page 430\]](#).

## 1.5.3.1 Creating a Component

You can create a component using a Wizard or from the Toolbox, Browser, or *Model* menu.

- Use the *Component* tool in the Toolbox.
- Select ► *Model* ► *Components* ▾ to access the List of Components, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select ► *New* ▷ *Component* ▾.
- Select ► *Tools* ▷ *Create Component* ▾ to access the Standard Component Wizard.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

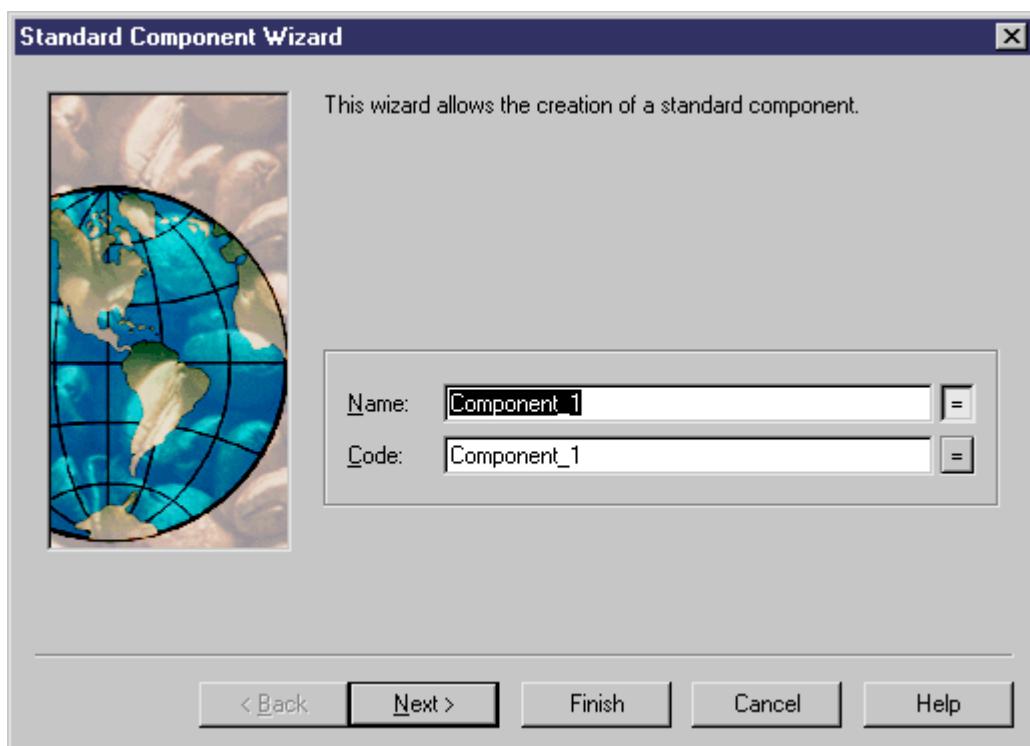
For more information on other types of components: EJB, servlets, JSP and ASP.NET, see [Java \[page 292\]](#) and [VB .NET \[page 376\]](#).

### 1.5.3.1.1 Using the Standard Component Wizard

PowerDesigner provides a wizard to help you in creating components from classes.

#### Procedure

1. Open a class or composite structure diagram and select the class or classes that you want to include in the new component.
2. Select ► *Tools* ▷ *Create Component* ▾ to open the Standard Component Wizard.



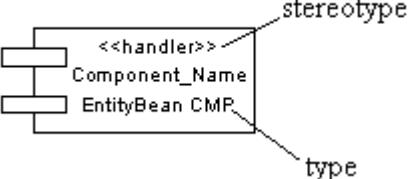
3. Type a name and a code for the component and click Next.
4. If you want the component to have a symbol and appear in a diagram, then select the Create Symbol In checkbox and specify the diagram in which you want it to appear (you can choose to create a new diagram). If you do not select this checkbox, then the component is created and visible from the Browser but will have no symbol.
5. If you want to create a new class diagram to regroup the classes selected, then select the Create Class Diagram for Component Classifiers.

### 1.5.3.2 Component Properties

To view or edit a component's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | <p>Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field.</p>  |
| Stereotype        | <p>Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.</p> <p>The following standard stereotypes are available by default:</p> <ul style="list-style-type: none"> <li>• &lt;&lt;Document&gt;&gt; - Generic file that is not a source file or an executable</li> <li>• &lt;&lt;Executable&gt;&gt; - Program file that can be executed on a computer system</li> <li>• &lt;&lt;File&gt;&gt; - Physical file in the context of the system developed</li> <li>• &lt;&lt;Library&gt;&gt; - Static or dynamic library file</li> <li>• &lt;&lt;Table&gt;&gt; - Database table</li> </ul> <p>You can modify an existing stereotype or create a new one in an object language or extension file.</p> |

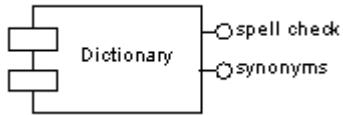
| Property      | Description  |
|---------------|--|
| Type          | <p>Specifies the type of component. You can choose between a standard component (if no specific implementation has been defined) or a specific component, such as EJB, JSP, Servlet or ASP.NET (see <a href="#">Web Services [page 211]</a>).</p> <p>To display the type of a component, select  <a href="#">Tools</a> <a href="#">Display Preferences</a> and select the Type option in the component category.</p>  <p>Whenever you change the type of a component after creation, the modification triggers a conversion from one type to another: all relevant interfaces, classes, and dependencies are automatically created and initialized. Such a change will affect some property sheets, the Check Model feature, and code generation.</p> <p>For example, if you convert a standard component to an EJB Entity Bean, it will automatically generate a Bean class and a primary key class of the EJB, as well as home and component interfaces. If you convert an EJB to a standard component, the classes and interfaces of the EJB are preserved in the model.</p> |
| Transaction   | Used for a component with transactional behavior.  |
| Class diagram | Specifies a diagram with classes and interfaces linked to the component, which is automatically created and updated (see <a href="#">Creating a Class Diagram for a Component [page 201]</a> ).  |
| Web service   | Indicates that the component is a Web service.   |
| Keywords      | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## Interfaces Tab

Each component uses one or several interfaces. It also uses or requires interfaces from other components. These interfaces are visible entry points and services that a component makes available to other software components and classes. If dependencies among components originate from interfaces, these components can be replaced by other components that use the same interfaces.

The Interfaces tab lists interfaces exposed and implemented by the component. Use the [Add Objects](#) tool to add existing interfaces or the [Create an Object](#) tool to create new interfaces for the component.

Component interfaces are shown as circles linked to the component side by an horizontal or a vertical line:



The symbol of a component interface is visible if you have selected the Interface symbols display preference from **Tools > Display Preferences**. The symbol of an interface can be moved around the component symbol, and the link from the component to the interface can be extended.

If you are working with EJB, some of the interfaces have a special meaning (local interface, remote interface, etc...). For more information, see [Defining Interfaces and Classes for EJBs \[page 307\]](#).

## Classes Tab

A component usually uses one implementation class as the main class, while other classes are used to implement the functions of the component. Typically, a component consists of many internal classes and packages of classes but it may also be assembled from a collection of smaller components.

The Classes tab lists classes contained within the component. Use the [Add Objects](#) tool to add existing classes or the [Create an Object](#) tool to create new classes for the component.

Classes are not visible in the component diagram.

The following tabs are also available:

- Components - lists the child components of the component. You can create components directly in this tab.
- Operations - lists the operations contained within the interfaces associated with the component. Use the filter in the toolbar to filter by a specific interfaces.
- Ports - lists the ports associated with the component. You can create ports directly in this tab (see [Ports \(OOM\) \[page 44\]](#)).
- Parts - lists the parts associated with the component. You can create parts directly in this tab (see [Parts \(OOM\) \[page 42\]](#)).
- Files - lists the files associated with the component. If files are attached to a component they are deployed to the server with the component (see [Files \(OOM\) \[page 207\]](#)).
- Related Diagrams - lists and lets you add model diagrams that are related to the component (see [Core Features Guide > Modeling with PowerDesigner > Diagrams, Matrices, and Symbols > Diagrams > Specifying Diagrams as Related Diagrams](#)).

### 1.5.3.3 Creating a Class Diagram for a Component

You can create a class diagram for a selected component to have an overall view of the classes and interfaces associated with the component. You can only create one class diagram per component.

#### Context

The Create/Update Class Diagram feature from the component contextual menu, acts as follows:

- Creates a class diagram if none exists
- Attaches a class diagram to a component
- Adds symbols of all related interfaces and classes in the class diagram
- Completes the links in the diagram

This feature also allows you to update a class diagram after you have made some modifications to a component.

The Open Class Diagram feature, available from the component contextual menu, opens the specific class diagram if it exists, or it creates a new default class diagram.

For EJB components for example, the Open Class Diagram feature opens the class diagram where the Bean class of the component is defined.

If you delete a component that is attached to a class diagram, the class diagram is also deleted. Moreover, the classes and interfaces symbols are deleted in the class diagram, but the classes and interfaces objects remain in the model.

#### Procedure

Right-click the component in the component diagram and select *Create/Update Class Diagram* from the contextual menu.

#### Results

A new class diagram, specific to the component, is displayed in the diagram window and the corresponding node is displayed under in the Browser. You can further create objects related to the component in the new class diagram.

##### Note

To open the class diagram for a component, right-click the component in the diagram and select *Open Class Diagram* from the contextual menu or press `Ctrl` and double-click the component.

## 1.5.3.4 Deploying a Component to a Node

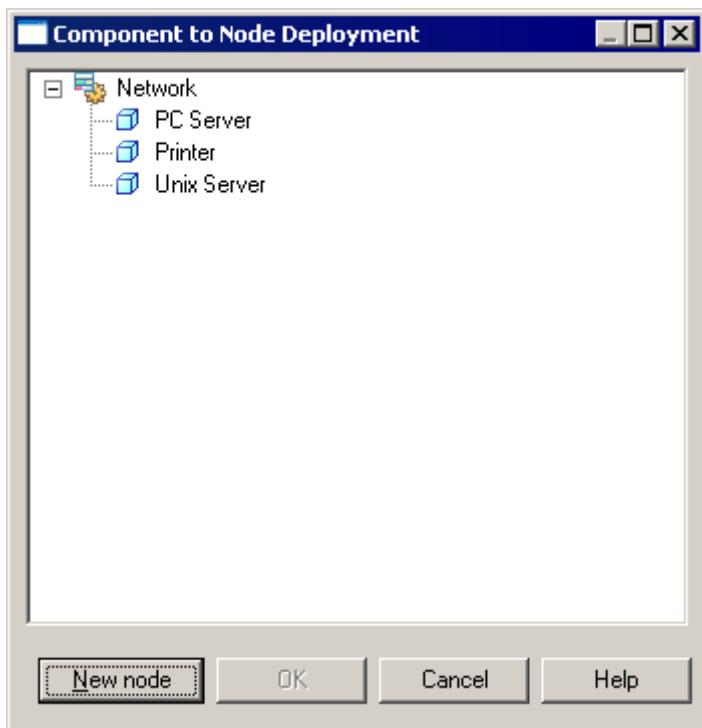
Deploying a component to a node allows you to set an instance of the component within a node. You can deploy a component from the component diagram or from the Browser. After deployment, a shortcut of the component and a new component instance are created within the deployment node.

### Context

You can only select one component to deploy at a time.

### Procedure

1. Right-click the component symbol and select Deploy Component to Node to open the Component to Node Deployment window:



2. Select either an existing node to deploy the component to or click the New Node button to create a new node and deploy the component to it.
3. Click OK to create a new component instance inside the selected node.

## 1.5.4 Nodes (OOM)

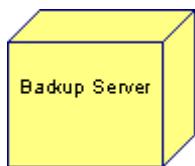
A node is the main element of the deployment diagram. It is a physical element that represents a processing resource, a real physical unit, or physical location of a deployment (computer, printer, or other hardware units).

In UML, a node is defined as Type or Instance. This allows you to define for example 'BackupMachine' as node Type, and 'Server:BackupMachine' as Instance. As a matter of simplification, PowerDesigner handles only one element, called node, which actually represents a node instance. If you need to designate the type, you can use a stereotype for example.

A node can be created in the following diagrams:

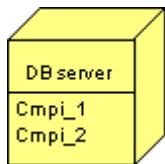
- Deployment Diagram

The symbol of a node is a cube:



A node cannot contain another node, however it can contain component instances and file objects: the software component instances and/or associated file objects that are executed within the nodes. You can use shortcuts of component as well.

You can add a component instance from the node property sheet. You can display the list of component instances in the node symbol as well, by selecting the option Components in the node display preferences.



### Composite View

You can add component instances and file objects to a node by dropping them onto the node symbol. By default, these sub-objects are displayed inside the symbol. To disable the display of these sub-objects, right click the node symbol and select **► Composite View ► None**. To redisplay them, select **► Composite View ► Editable**.

## 1.5.4.1 Creating a Node

You can create a node from the Toolbox, Browser, or *Model* menu.

- Use the *Node* tool in the Toolbox.
- Select ► *Model* ► *Nodes* ► to access the List of Nodes, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select ► *New* ► *Node* ►.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.5.4.2 Node Properties

To view or edit a node's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property              | Description  |
|-----------------------|--|
| Name/Code/<br>Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype            | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Cardinality           | Specific numbers of instances that the node can have, for example: 0...1.  |
| Network address       | Address or machine name.   |
| Keywords              | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

## Component Instances Tab

The *Component Instances* tab lists all instances of components that can run or execute on the current node (see *Component Instances (OOM)* [page 205]). You can specify component instances directly on this tab and they will be displayed within the node symbol.

### 1.5.4.3 Node Diagrams

You can create deployment diagrams within a node to visualize the component instances and file objects it contains.

To create a node diagram, press **Ctrl** and double-click the node symbol in the deployment diagram, or right-click the node in the Browser and select **New > Deployment Diagram**. The diagram is created under the node in the Browser and opens in the canvas pane.

To open a node diagram from the node symbol in a deployment diagram, press **Ctrl** and double-click on the node symbol or right-click the node symbol and select **Open Diagram**.

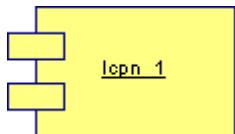
### 1.5.5 Component Instances (OOM)

A component instance is an instance of a component that can run or execute on a node. Whenever a component is processed into a node, a component instance is created. The component instance plays an important role because it contains the parameter for deployment into a server.

A component instance can be created in the following diagrams:

- Deployment Diagram

The component instance symbol is the same as the component symbol in the component diagram.



The component instance relationship with the node is similar to a composition; it is a strong relationship, whereas the file object relationship with the node is different because several nodes can use the same file object according to deployment needs.

### Drag and Drop a Component in a Deployment Diagram

You can drag a component from the Browser and drop it into a deployment diagram to automatically create a component instance linked to the component.

The component instance that inherits from the component automatically inherits its type: the type of the component is displayed in the property sheet of the component instance.

### Deploy Component to Node from the Component Diagram

You can create a component instance from a component. To do this, use the Deploy Component to Node feature. This feature is available from the contextual menu of a component (in the component diagram) or from the

Browser. This creates a component instance and attaches the component instance to a node. If you display the node symbol in a deployment diagram, the component instance name is displayed within the node symbol to which it is attached.

For more information, see [Deploying a Component to a Node \[page 202\]](#).

### 1.5.5.1 Creating a Component Instance

You can create a component instance from the Toolbox, Browser, or *Model* menu.

- Use the *Component Instance* tool in the Toolbox.
- Select *Model* to access the List of Component Instances, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select *New* .

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

### 1.5.5.2 Component Instance Properties

To view or edit a component instance's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property              | Description  |
|-----------------------|--|
| Name/Code/<br>Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field. |
| Stereotype            | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Cardinality           | Specific number of occurrences that the component instance can have, for example: 0...1.   |
| Component             | Component of which the component instance is an instance. If you change the component name in this box, the name of the component instance is updated in the model.  |
| Component type        | Read-only box that shows the type of the component from which the component instance is coming.  |
| Web service           | Indicates that the component instance is an instance of a Web service component.   |

| Property | Description   |
|----------|---|
| Keywords | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas. |

If you want to list all component instances of a component, click the Component Instances tabbed page in the Dependencies tab of the component property sheet.

## 1.5.6 Files (OOM)

A file object can be a bitmap file used for documentation, or it can be a file containing text that is used for deployment into a server.

A file can be created in the following diagrams:

- All Diagrams

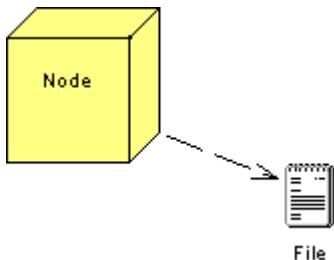
The symbol of a file object is as follows:



File\_1

The file object can have a special function in a deployment diagram, where it can be specified as an artifact (by selecting the Artifact property) and generated during the generation process.

When you want to associate a file object to a node, you can drag a dependency from the file object to the node:



You can also use **Ctrl** and double-click on the parent node symbol, then create the file object into the node diagram.

You can edit a file object by right-clicking its symbol in the deployment diagram and selecting *Open Document* or **Open With** > <text editor of your choice> from the contextual menu.

## 1.5.6.1 Creating a File Object

You can create a file object by drag and drop or from the Toolbox, Browser, or *Model* menu.

- Use the *File* tool in the Toolbox.
- Select *Model* to access the List of Files, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select *New* .
- Drag and drop a file from Windows Explorer to your diagram or the PowerDesigner Browser

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.5.6.2 File Object Properties

To view or edit a file object's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

| Property              | Description  |
|-----------------------|--|
| Name/Code/<br>Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field.   |
| Stereotype            | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Location type         | Specifies the nature of the file object. You can choose from the following: <ul style="list-style-type: none"><li>• <i>Embedded file</i> – the file is stored within the model and is saved when you save the model. If you subsequently change the type to external, you will be warned that the existing contents will be lost.</li><li>• <i>External file</i> – the file is stored in the Windows file system, and you must enter its path in the <i>Location</i> field. If you subsequently change the type to embedded, you will be prompted to import the contents of the file into the model.</li><li>• <i>URL</i> – the file is on the web and you must enter its URL in the <i>Location</i> field</li></ul> |
| Location              | [External and URL types only] Specifies the path or URL to the file.   |
| Extension             | Specifies the extension of the file object, which is used to associate it with an editor. By default, the extension is set to <code>txt</code> .   |
| Generate              | Specifies to generate the file object when you generate the model to another model.  |

| Property | Description  |
|----------|--|
| Artifact | <p>Specifies that the file object is not a piece of documentation, but rather forms an integral part of the application.</p> <p>If an artifact has an extension that is defined in the <i>Editors</i> page in the General Options dialog linked to the <i>&lt;internal&gt;</i> editor, a <i>Contents</i> tab is displayed in the artifact property sheet, which allows you to edit the artifact file in the PowerDesigner text editor.</p> |
| Keywords | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.  |

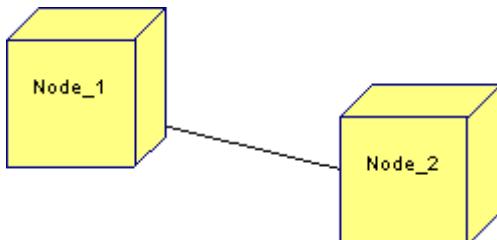
## 1.5.7 Node Associations (OOM)

You can create associations between nodes, called node associations. They are defined with a role name and a multiplicity at each end. An association between two nodes means that the nodes communicate with each other, for example when a server is sending data to a backup server.

A node association can be created in the following diagrams:

- Deployment Diagram

A node association symbol is as follows:



### 1.5.7.1 Creating a Node Association

You can create a node association from the Toolbox, Browser, or *Model* menu.

- Use the *Node Association* tool in the Toolbox.
- Select **Model > Node Associations** to access the List of Node Associations, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Node Association**.

For general information about creating objects, see *Core Features Guide > Modeling with PowerDesigner > Objects*.

## 1.5.7.2 Node Association Properties

To view or edit a node association's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

The *General* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Name/Code/Comment | Identify the object. The name should clearly convey the object's purpose to non-technical users, while the code, which is used for generating code or scripts, may be abbreviated, and should not normally include spaces. You can optionally add a comment to provide more detailed information about the object. By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the <i>Code</i> field.   |
| Stereotype        | Extends the semantics of the object beyond the core UML definition. You can enter a stereotype directly in this field, or add stereotypes to the list by specifying them in an extension file.   |
| Role A            | One side of a node association. Each role can have a name and a cardinality and be navigable.  |
| Node A            | Name of the node at one end of the node association. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object.  |
| Multiplicity A    | <p>Multiplicity indicates the maximum and minimum number of instances of the node association. You can choose between:</p> <ul style="list-style-type: none"><li>• 0..1 – zero or one</li><li>• 0..* – zero to unlimited</li><li>• 1..1 – exactly one</li><li>• 1..* – one to unlimited</li><li>• * – none to unlimited</li></ul> <p>For example, in a computer environment, there can be 100 clients and 100 machines but there is a constraint that says that a machine can accept at most 4 clients at the same time. In this case, the maximum number of instances is set to 4 in the Multiplicity box on the machine side:</p> <pre>graph LR; client[client] --- machine[machine];</pre> <p>The diagram shows two yellow 3D cube nodes. The left node is labeled "client" and the right node is labeled "machine". A horizontal line connects them. On the "client" node side, there is a multiplicity "1..*" below the line. On the "machine" node side, there is a multiplicity "4" above the line.</p> |
| Role B            | One side of a node association. Each role can have a name and a cardinality and be navigable.  |
| Node B            | Name of the node at the other end of the node association. You can use the tools to the right of the list to create an object, browse the complete tree of available objects or view the properties of the currently selected object.  |

| Property       | Description  |
|----------------|--|
| Multiplicity B | Multiplicity indicates the maximum and minimum number of instances of the node association. For more details, see Multiplicity A, above. |
| Keywords       | Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.                        |

## 1.6 Web Services

Web services are applications stored on Web servers that are accessed through standard Web protocols, receive requests, process them, and return responses. PowerDesigner supports the modeling of a Web service as an OOM component (EJB, servlet, or standard component) containing a Web service implementation class, which you define in class, component and deployment diagrams.

PowerDesigner supports the modeling of Web services for Java or .NET through a wizard or by reverse-engineering WSDL, and the generation of WSDL, server side Web service code, and client proxies for Java and .NET. When you create a Java, C#, or VB.NET OOM, a WSDL extension file is automatically attached to the model to support the definition of Web services.

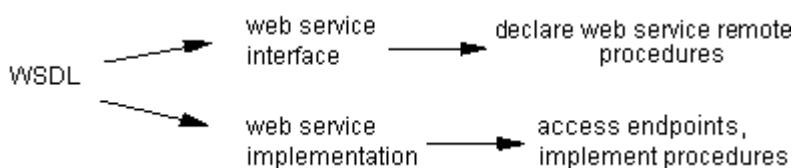
To work with Web services, you need a Java, C# or Visual Basic .NET compiler. For Java, you also need a WSDL-to-Java and a Java-to-WSDL tool to generate Java proxy code and JAX-RPC compliant server side code.

To generate client proxy code for .NET, you will need to use the WSDL.exe included in Visual Studio .NET and declare the path to the WSDL.exe in the General Options dialog box (▶ *Tools* ▶ *General Options* ▷) when you create the WSDL environment variables.

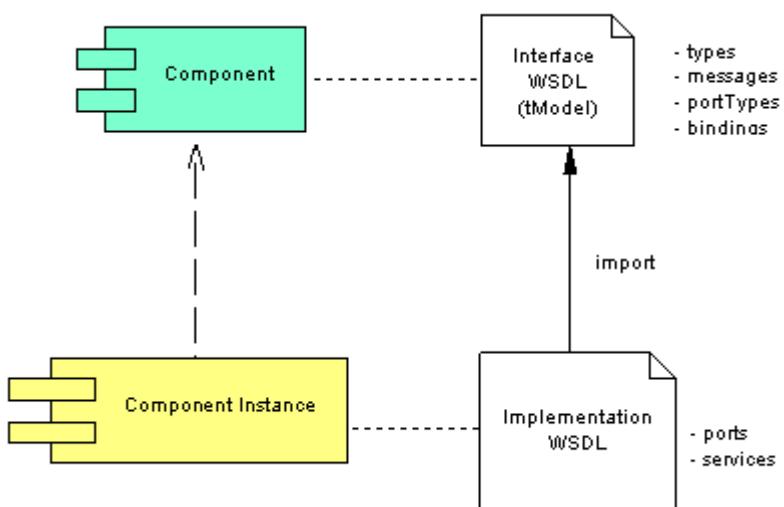
WSDL is a language that describes what a Web service is capable of and how a client can locate and invoke that service. The Web Services Description Language (WSDL) 1.1 document is available at <http://www.w3.org/TR/wsdl>, and the working group is located here: <http://www.w3.org/2002/ws/desc>. WSDL documents are made up of the following elements:

- Types - a container for data type definitions using some type system (such as XSD)
- Message - an abstract, typed definition of the data being communicated
- Operation - an abstract description of an action supported by the service
- Port Type - an abstract set of operations supported by one or more endpoints
- Binding - a concrete protocol and data format specification for a particular port type
- Port - a single endpoint defined as a combination of a binding and a network address
- Service - a collection of related endpoints

WSDL is used to define the Web service interface (the procedures that allow you to create a Web service), the Web service implementation (how to implement these procedures through services and ports (access endpoints URLs)), or both. As a result, it is possible to use two WSDL files, one for the interface and one for the implementation.



In an OOM, an interface WSDL is associated with a component, and an implementation WSDL is associated with a component instance. You can save both WSDL files within the model.



## 1.6.1 Web Service Components (OOM)

A Web service is represented as a component that you can display in a component diagram with the Web service interface and implementation code. To declare a component as a Web service, you select the [Web Service](#) option on the [General](#) tab of its property sheet .A component can be a Web service Interface or a Web service Implementation type.

You can also deploy Web service components to nodes to describe deployment of components into servers (see [Deploying a Component to a Node \[page 202\]](#)).

The following Web service types are supported for the Java language:

| Type             | Description  |
|------------------|--|
| Java Web Service | Exposes a Java class with the .jws extension as a Web service using Apache Axis. To deploy the Java class, simply copy the .jws file to an appropriate server. For example, for Apache Axis, you can copy the .jws file to the directory webapps\axis. |

| Type                 | Description  |
|----------------------|--|
| Axis RPC/ Axis EJB   | <p>For Axis RPC, PowerDesigner uses a Java class for implementation and Apache Axis for deployment. The supported provider type is Java:RPC. The supported provider styles are RPC, document and wrapped, available in the AxisProviderStyle property of the Web service component.</p> <p>For Axis EJB, PowerDesigner uses a Stateless Session Bean for the implementation, an application server for EJB deployment and Apache Axis for exposing the EJB as a Web service. To expose a Stateless Session Bean as Web service using Axis, you need to Generate the EJB code, compile and package the EJB, deploy it to a J2EE server, and expose it as a Web Service using Axis.</p> <p>To customize Axis deployment descriptor generation, change the appropriate Axis properties of the Web service component.</p> <p>A deploy.wsdd and an undeploy.wsdd are generated from the model or the package that contains Web service components. A single deploy.wsdd and undeploy.wsdd files are generated for all Web service components of the model or package.</p> |
| JAX-RPC              | <p>PowerDesigner uses the JAX-RPC model for implementation. You must generate and compile the Web Service Java class and interface code, run a JAX-RPC tool to generate server side artifacts and client side proxy to handle the Web Service, package all the compiled code, WSDL and deployment descriptor in a .WAR file and deploy it to the server.</p> <p>To invoke the wscompile.bat tool from PowerDesigner, you have to define an environment variable WSCOMPILE indicating the full path to the wscompile.bat file in the Variables category in General Options (▶ Tools ▶ General Options ▷). To run wscompile.bat, the jaxrpc-api.jar file must be in your CLASSPATH environment variable.</p>   |
| JAXM                 | <p>PowerDesigner uses the JAXM model for implementation.</p> <p>The JAXM Java class uses the onMessage() method to get the SOAP input message and return the output SOAP message. To generate correct WSDL, you have to define a Web Service method with the correct name, input message format and output message format but without implementation. The onMessage() method should not be defined as a Web Service method.</p> <p>To compile JAXM Web service components, you need the jaxm-api.jar, jaxp-api.jar and saaj-api.jar files in your CLASSPATH environment variable.</p>  |
| Web Service for J2EE | <p>Exposes a Stateless Session Bean as a Web service using the Web Service for J2EE (JSR109) model. PowerDesigner uses Web Services for J2EE specification for implementation.</p> <p>Developing Stateless Session Bean as Web Service is similar to JAX-RPC: you use a Bean class instead of a normal Java class. As for JAX-RPC, it is limited to simple message formats.</p> <p>In Java, Web services may be implemented either through JAX-RPC endpoints (Web components, they are represented as servlets in the OOM and are packaged into a WAR) or EJB stateless session bean components (packaged into an EJB JAR). Both of these implementations expose their Web methods through a service endpoint interface (SEI).</p> <p>In both cases, WSDL files, and the required deployment descriptors should be included in the WEB-INF or META-INF directories. You can refer to chapters 5 and 7 of the Web Services for J2EE specification for more information.</p>   |

## 1.6.1.1 Creating a Web Service Component

PowerDesigner provides a wizard to help you create a Web service component from a class diagram. You can create a component from an existing class or have the wizard create a new implementation class.

### Context

#### i Note

You can also create a Web service component manually in a component diagram by using the *Component* tool and selecting the *Web Service* option on the component property sheet *General* tab.

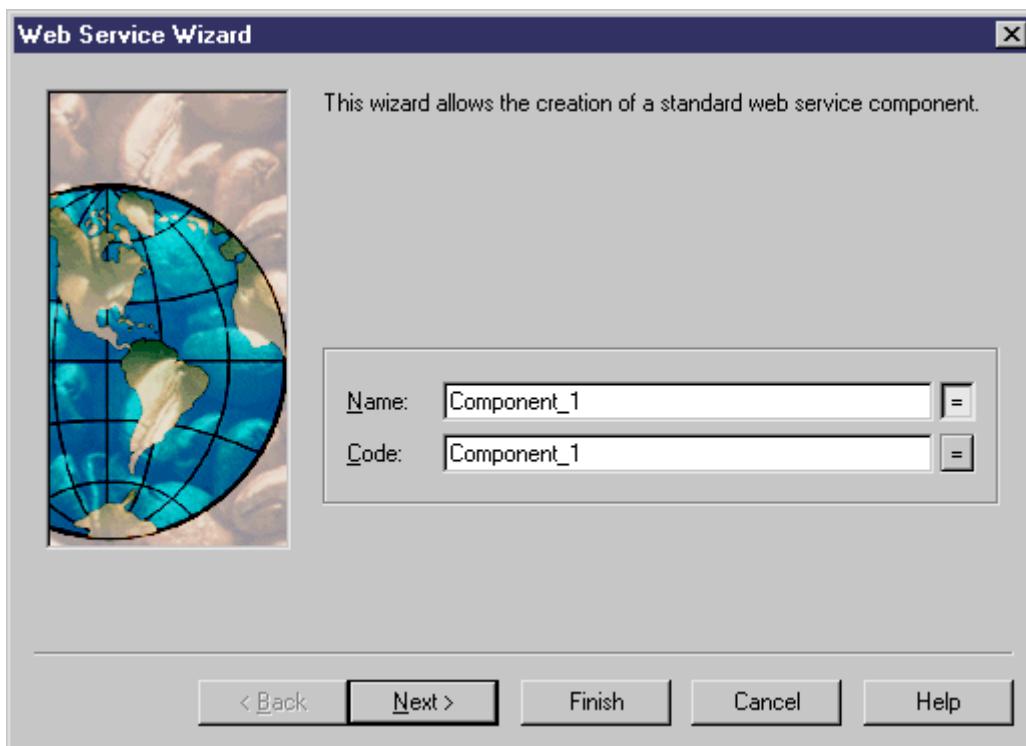
### Procedure

1. [optional] Select one or more classes in a class diagram to act as implementation classes.

#### i Note

We recommend that you create the Web service component within a package so that the package acts as a namespace.

2. Select  *Tools*  to open the Web Service Wizard:



3. Enter a name and code for the Web service component and click *Next*.
4. Select a Web service type (implementation or interface) and a component type and click *Next*.
5. [if you have not selected a class] Select an implementation class (if you select <None>, a new class will be created) and click *Next*.
6. Select whether you want to create a symbol for the new Web service component, and whether you want to create a class diagram to display the classes and interfaces associated with it, and then click *Finish*.

PowerDesigner performs the following actions:

- A Web service component is created.
- The class selected is converted to a Web service implementation class or a new class is created, named after the component.
- Depending on the component type, required interfaces or default Web method operations are created.

### 1.6.1.2 Web Service Component Properties

Web service component property sheets contain all the standard component properties, and some additional tabs.

#### Component Web Service Tab

The *Web Service* tab includes the following properties:

| Property              | Description   |
|-----------------------|---|
| Web service class     | Specifies the Web service class name. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected class. If the Web service component is a Stateless Session Bean, the Web service class is also the bean class. |
| Application namespace | Specifies the application namespace, which is used to generate the URL for the Web service in the server. By default, the component package or model code is used, but you can override it here.  |
| WSDL URL              | Specifies where the WSDL is published on the web.   |
| Web service type      | Specifies the type of Web service. An interface is a component that defines the service interface only. An implementation is a component that implements a service interface.   |
| Use external WSDL     | Specifies that the WSDL is published at a specific URL and that the original WSDL will be preserved. When you import a WSDL, this option is selected by default.  |

## Component WSDL Tab

The *WSDL* tab includes the following properties:

| Property         | Description  |
|------------------|--|
| Target namespace | Specifies a URL linked to a method that ensures the uniqueness of the Web service and avoids conflicts with other Web services of the same name.<br><br>By default, this field is set to <code>http://tempuri.org</code> for .NET and <code>urn:%Code%Interface</code> for Java, but we recommend that you change it to ensure the service name uniqueness |
| Prefix           | Specifies the target namespace prefix  |
| Encoding style   | Specifies the kind of encoding, either SOAP ( <code>soap:xxx</code> ) or XML-Schema ( <code>xsd:xxx</code> ) for the WSDL  |
| Comment          | Provides a description of the WSDL file.   |
| WSDL editor      | Allows you to edit the contents of the WSDL. If you make edits, then the User-Defined tool is pressed.   |

## Component WSDL Schema Tab

The *WSDL Schema* tab in the component property sheet includes a text zone that contains some shared schema definitions from the WSDL schema. This part of the schema defines the data types used by the input, output and fault messages.

The other part of the schema is defined within the different Web methods (operations) as SOAP input message data types, SOAP output message data types, and SOAP fault data types (see [Defining SOAP Data Types of the WSDL Schema \[page 224\]](#)).

## Component UDDI/Extended Attributes Tab

These tabs include the following properties:

| Name                          | Description  |
|-------------------------------|--|
| SOAP binding style/ transport | Specify the SOAP binding style and transport URI.<br><br>Scripting name: <code>SoapBindingStyle</code> , <code>SoapBindingTransport</code> |
| SOAP body namespace           | Specifies the namespace of the XML schema data types in the WSDL<br><br>Scripting name: <code>SoapBodyNamespace</code>                     |

| Name                            | Description  |
|---------------------------------|--|
| Business name/ description/ key | Specifies the name, description, and key of the business found in a UDDI registry.<br>Scripting name: BusinessName, BusinessDescription, BusinessKey |
| Namespaces                      | Stores additional namespace that are not automatically identified by PowerDesigner.<br>Scripting name: Namespaces                                    |
| Service name                    | Stores the name, description, and key of the service found in a UDDI registry.<br>Scripting name: ServiceName, ServiceDescription, ServiceKey        |
| tModel name                     | Stores the name, key, and URL of the tModel found in a UDDI registry.<br>Scripting name: t modelName, t modelKey, t modelURL                         |
| UDDI operator URL               | Stores the URL of the UDDI registry operator URL used to find the WSDL<br>Scripting name: UDDIOperatorURL  |

You can use the *Preview* tab of the component to preview the code that will be generated for the Web service component.

### 1.6.1.3 Web Service Implementation Class Properties

A Web service requires one implementation class. An implementation class can only be associated with one Web service component. In .NET languages, the implementation class can be generated inside the .asmx file or outside. In Java, a Web service class can have a serialization and a deserialization class.

Web service implementation classes contain the following additional properties on the Detail tab:

| Property              | Description  |
|-----------------------|--|
| Web service component | Specifies the Web service component linked to the Web service implementation class. Click the Properties tool to the right of this field to open the component property sheet. |
| Serialization class   | Specifies the class used to convert an object into a text or binary format. Click the Properties tool to the right of this field to open the class property sheet.             |
| Deserialization class | Specifies the class used to convert a text, XML or binary format into an object. Click the Properties tool to the right of this field to open the class property sheet.        |

Click the Preview tab to preview:

- The Web service implementation class in Java
- The .ASMX file in .NET
- The interface WSDL generated from the component and implementation class in Java and .NET (read-only)

## 1.6.1.4 WSDL Data Types

WSDL uses XML Schema to define data types for message structures. To generate WSDL, it is necessary to map Java or .NET types to XML types.

Three data type maps are defined in the WSDL extension file.

- `WSDL2Local` - converts WSDL types to Java or .NET types
- `Local2SOAP` - converts Java or .NET types to SOAP types
- `Local2XSD` - converts Java or .NET types to XML Schema types

You can select WSDL basic data types for XML Schema encoding or SOAP encoding from the lists on the *Web Method* tab of operation property sheets or the *General* tab of parameter property sheets. As long as the WSDL data type is not manually changed, it is synchronized with the Java or .NET data type. You can use the class property sheet *Preview* tab to verify the code at any time.

### i Note

Classes used as data types are declared as Complex Types inside the <types> section of WSDL.

## 1.6.2 Web Service Methods (OOM)

You can specify one or more methods as part of your Web Service. PowerDesigner models Web service methods as operations with the *Web Service Method* property selected. Web service methods can belong to the component implementation class or to component interfaces.

To create a Web service method, open the property sheet of the Web Service class or interface and click the *Operation* tab, then click the *Insert a row* tool, click the *Properties* tool to open the operation property sheet, and select the *Web Service method* property on the *General* tab

The Web Method tab in the operation property sheet includes the following properties:

| Property             | Description  |
|----------------------|--|
| SOAP extension class | Used for .NET. At creation of the class, new default functions are added. In .NET, a method can have a SOAP extension class to handle the serialization and de-serialization for the method and to handle security of other SOAP extensions features. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected object. |
| WSDL data type       | Data type for the return type. It includes basic data types from the object language and complex data types from the WSDL Schema. You can click the Properties tool beside this box to display the WSDL Schema tab of the component property sheet. This tab shows the contents of the WSDL schema   |

The following tabs are also displayed:

- SOAP Input/ Output/ Fault - Specify the names and schemas of the SOAP input, output, and fault message elements.
- WSDL - Allows you to customize the WSDL generation by modifying input and output message names, SOAP operation style, port types to generate, SOAP action, Web method type, and so on. The SOAPPortType,

HttpGetPortType andHttpPostPortType properties are used to decide which Port Type should be generated. If a Web method is created in an interface, only the SOAPPortType attribute is set to True. This method is automatically added to the implementation class of the component.

A web service method can call other methods that are not exposed as Web service methods. In this case, these internal methods are not generated in the WSDL.

Interfaces linked to a Web service component can be used to design different groups of methods representing different port types.

A component interface containing at least one operation with the Web Service Method property selected is considered as a port type.

For JAXM Web Service component, the implementation of the Web Service must be done in the onMessage() method. To be able to generate the correct WSDL, you have to declare a Web Service method without implementation to define the input SOAP message and the output SOAP message.

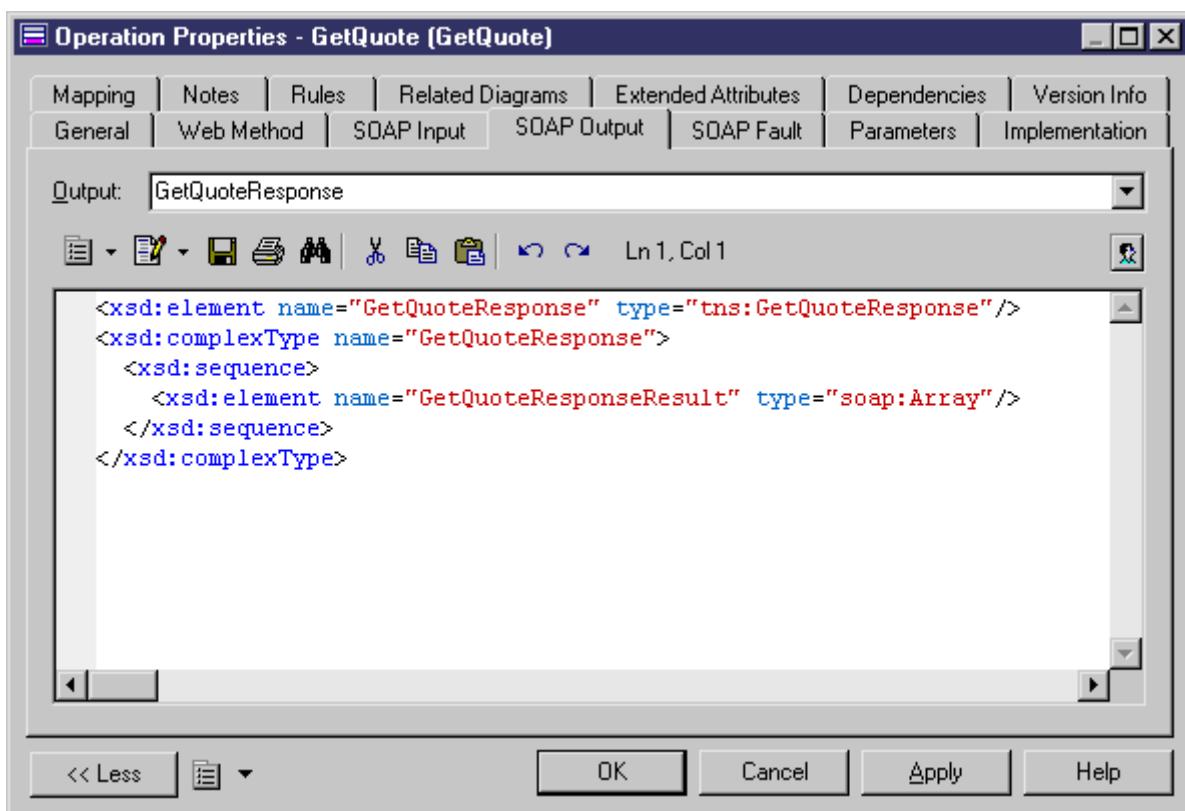
For more information on method implementation see [Implementing a Web Service Method in Java \[page 219\]](#) and [Implementing a Web Service Method in .NET \[page 223\]](#).

### 1.6.2.1 Implementing a Web Service Method in Java

To implement a Web service method, you have to define the operation return type, parameters, and implementation.

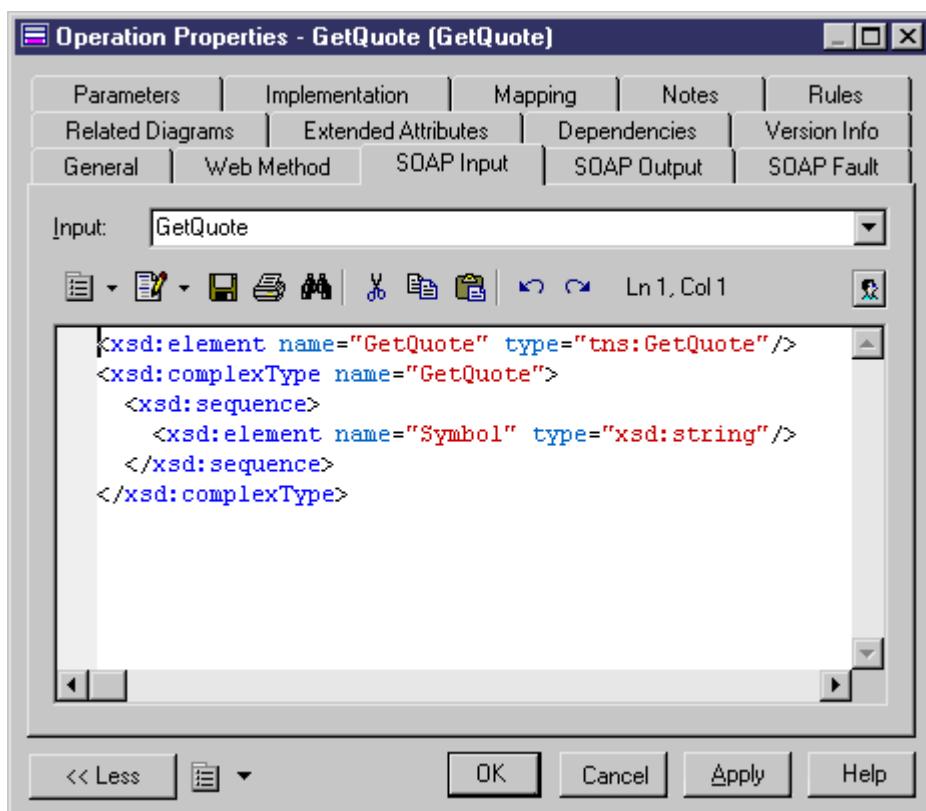
To define the return type of an operation you have to specify:

- The return type for the Java method on the *General* tab of the operation property sheet. If the type is a Java class or an array of Java class, PowerDesigner will generate an output message type based on the return class structure.
- The output message type for WSDL. For simple return values, this can be defined on the *Web Method* tab of the operation property sheet. For more complex return types, you can manually define the output message schema using the *SOAP Output* tab of the operation property sheet:

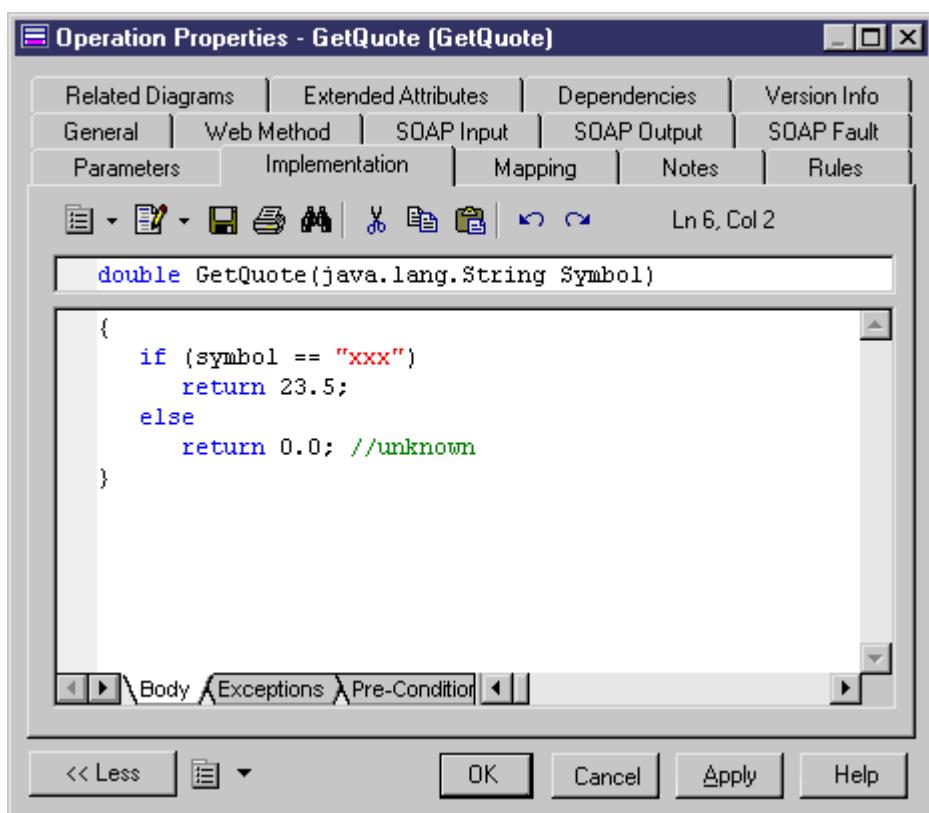


To define the parameters of an operation, you have to specify:

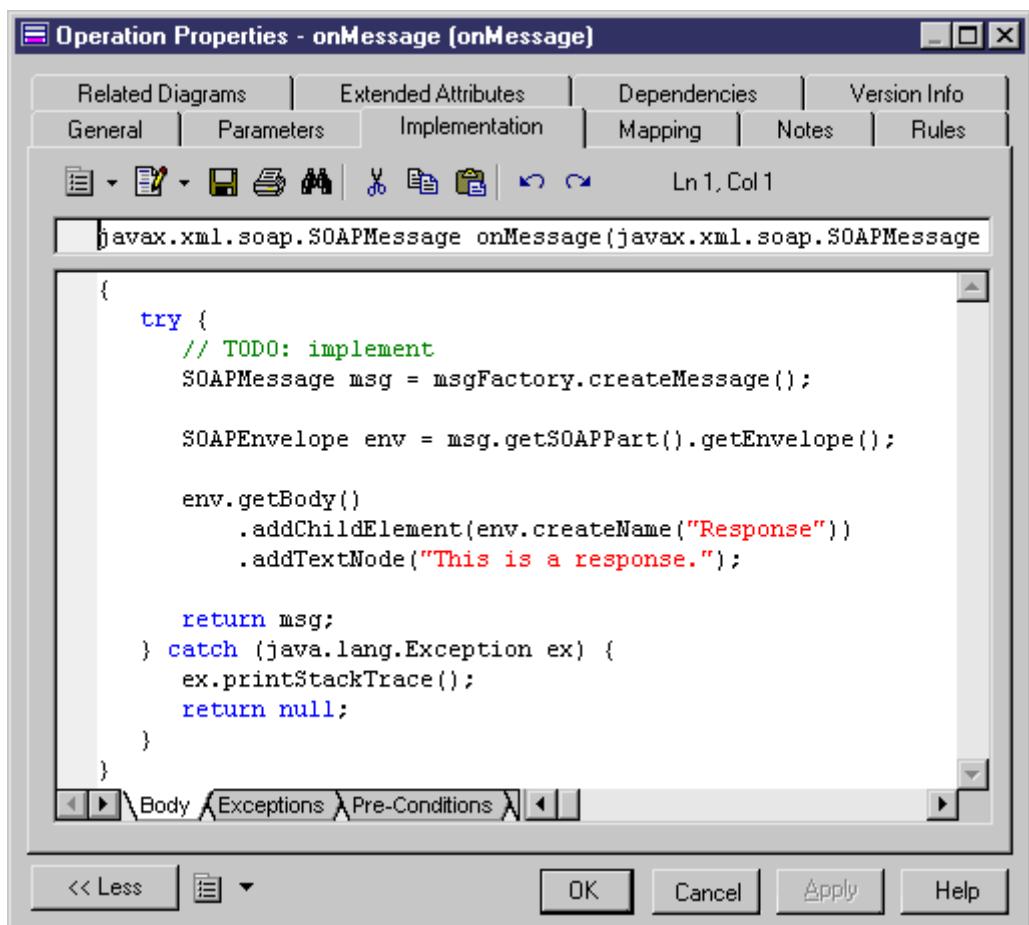
- The parameter type for the Java method using the *Data Type* list on the *General* tab of the Parameter property sheet.
- The input message type for WSDL using the *WSDL Data Type* list on the *General* tab of the Parameter property sheet. For simple parameter values, the input message type is generated from the WSDL types of the parameters. If a parameter is a class or an array of class, PowerDesigner will only generate SOAP binding. For more complex parameter types and input message types, you can manually define the input message schema using the SOAP Input tab in the operation property sheet:



You implement a Web Service method as a normal Java method. The following example, shows the implementation of Web service method GetQuote:



For a Web Service method in a JAXM Web Service, you have to implement the onMessage() method. You have to process the input SOAP message, generate an output SOAP message and return the output message:



### 1.6.2.2 Implementing a Web Service Method in .NET

To implement a Web service method in .NET, you have to define input parameters and return type.

These procedures are described in [Implementing a Web Service Method in Java \[page 219\]](#).

By default, PowerDesigner generates the C# or VB .NET Web Service class inside the .asmx file. If you want to generate the C# or VB .NET class in a separate file and use the CodeBehind mode for the .asmx file, you have to modify a generation option: in the Options tab of the Generation dialog box, set the value of Generate Web Service code in .asmx file to False.

You can preview the .asmx file code and the WSDL code from the Preview tab of the class property sheet.

### 1.6.2.3 Defining SOAP Data Types of the WSDL Schema

Each Web method has an input, output and fault data type to be defined. A data type has a name and a schema definition.

For reverse-engineered Web services, input, output, and fault data types are set to the value found in the reversed WSDL schema. Data types that are not associated with any input, output, or fault are considered as shared schema definitions and are available in the component. They are displayed in the WSDL Schema tab of the component property sheet.

For newly-created operations, input and output data types are set to a default value, and are synchronized with parameter changes. Default data type name and schema are defined in the WSDL extension file and can be customized. However once modified, a data type becomes user-defined and cannot be synchronized any more. Fault data types are always user-defined.

You can reuse an existing data type defined in another operation. A check is available on components to make sure that no data type has different names inside the same component (see [Checking an OOM \[page 251\]](#)).

When generating, the WSDL schema is composed of the shared schema definitions from the component, and a computed combination of all SOAP input, output, and fault definitions from the operations.

You can type the SOAP input, SOAP output and SOAP fault data type names in the appropriate tabs in the operation property sheet. Each tab contains a box, and a text zone in which you can edit the data type definition from the WSDL schema.

### 1.6.3 Web Service Component Instances (OOM)

A component instance defines the ports, the access type, and the access endpoint that is the full URL to invoke the Web service. PowerDesigner models the deployment of Web services in a deployment diagram, which can show the deployment of a Web service into one or several servers, displaying network addresses, access endpoints, and access types.

Select the *Web Service* property on the *General* tab of the component instance property sheet to indicate that the component instance is an instance of a Web service component. For more information on the deployment diagram, see [Implementation Diagrams \[page 192\]](#).

When the *Web Service* check box is selected, additional tabs are displayed in the component instance property sheet:

- The *Web Service* tab contains the following properties:

| Property          | Description  |
|-------------------|--|
| Access point URL  | <p>Displays the full URL to invoke the Web service. It is a calculated value that uses the network address located in the node. You can also type your own URL by using the <a href="#">User-Defined</a> tool to the right of the box.</p> <p>For .NET, the default syntax is:</p> <pre>accesstype://machine_networkaddress:port/application_namespace/ webservice_code.asmx</pre> <p>For example:</p> <pre>http://doc.acme.com:8080/WebService1/StockQuote.asmx</pre> <p>For Java, the default syntax is:</p> <pre>accesstype://machine_networkaddress:port/application_namespace/ webservice_code</pre> <p>For example:</p> <pre>http://doc.acme.com/WebService1/StockQuote</pre> <p>Computed attributes <code>AccessType</code> and <code>PortNumber</code> for code generator &amp; VBScript are computed from the access point URL. For example: http, https.</p> |
| WSDL URL          | <p>Indicates where the WSDL should be published on the web. You can type your own URL by using the <a href="#">User-Defined</a> tool to the right of the box.</p> <p>For .NET, the default syntax is:</p> <pre>accesstype://machine_networkaddress:port/application_namespace/ Webservice_code.asmx?WSDL</pre> <p>For Java, the default syntax is:</p> <pre>accesstype://machine_networkaddress:port/application_namespace/ wsdl_file</pre>  |
| Use external WSDL | Indicates that the WSDL is published at a specific URL   |

- The [WSDL](#) tab contains the following properties:

| Property              | Description  |
|-----------------------|--|
| Target namespace      | URL linked to a method that ensures the uniqueness of the Web service and avoids conflicts with other Web services of the same name. By default, it is: <code>http://tempuri.org/</code> for .NET, and <code>urn:%Component.targetNamespace%</code> for Java |
| Import interface WSDL | When selected, means that the implementation WSDL imports the existing interface WSDL  |

| Property    | Description  |
|-------------|--|
| Comment     | Used as the description of the WSDL file during WSDL generation and WSDL publishing in UDDI  |
| WSDL editor | You can also use a text zone below the Comment area to display the contents of the WSDL. When you click the <i>User-Defined</i> tool among the available tools, you make the contents user-defined. Once clicked, the contents can be overridden |

In addition the node property sheet contains the following property, specific to Web services:

| Property        | Description   |
|-----------------|---|
| Network address | Address or machine name. For example: doc.acme.com, or 123.456.78.9<br>Since a machine can be used for several services and each service may have a different access type, port number and path, the machine Network address is only used as a default value. You can redefine the real URL of each component instance in the property sheet of the component instance at any time. |

## 1.6.4 Generating Web Services for Java

You can generate client side or server side implementation classes, interfaces, deployment descriptor, JAR, WAR, or EAR from the Generate object language command in the Language menu. PowerDesigner supports generating JAXM, JAX-RPC, Web Service for J2EE (JSR 109), AXIS RPC, EJB, and Java Web Service (JWS) Web services.

### Context

Web services server side code generation consists in generating the following items:

- Generate Web service implementation classes and interfaces (Java class, Stateless Session Bean, Servlet, etc.)
- Generate Web services deployment descriptor for Java using the Web Service for J2EE (JSR109) specification. The deployment descriptor is an XML file that must be in every WAR archive, it is named WEB.xml by convention and contains information needed for deploying a Web service
- Generate interface WSDL and implementation WSDL files (WSDL files can be generated separately because they can be used for UDDI or client applications)

In general, once a Web service is deployed, the server is capable of generating an implementation WSDL.

Web services client side code generation consists in generating proxy classes.

## Procedure

1. Select  [Language](#)  [Generate Java Code](#)  to open the Generation dialog.
2. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see [Checking an OOM \[page 251\]](#)).
3. [optional] Click the [Selection](#) tab and specify the objects that you want to generate from. By default, all objects are generated.
4. [optional] Click the [Options](#) tab and set any appropriate generation options:
5. [optional] Click the [Tasks](#) tab and specify any appropriate generation tasks to perform:
  - For Axis EJB, select the commands in the following order: Package J2EE application in an EAR file, Deploy J2EE application, Expose EJB as Web Services.
  - For JAXM, select the command Java: Package J2EE application in an EAR file. This command will create a .WAR file and a .EAR file.
  - For JAX-RPC or stateless session beans, to generate server side code, select the command WSDL: Compile and PackageWeb Service Server-Side Code into an archive. To generate client proxy, select the command WSDL: Compile and PackageWeb Service Client Proxy into an archive.
6. Click [OK](#) to begin generation.

When generation is complete, the Generated Files dialog opens, listing the files that have been generated to the specified directory. Select a file in the list and click [Edit](#) to open it in your associated editor, or click [Close](#) to exit the dialog.

7. There are various methods for testing a Java Web Service:
  - Send a SOAP message. Write a Java program to send a SOAP message to the Web service and process the returned output SOAP message using the SAAJ API.
  - Use the Dynamic Invocation method defined by JAX-RPC.
  - Use the Dynamic Proxy method defined by JAX-RPC.
  - Use a client proxy to invoke a Web Service. If you use the JWSDP, you can use the wscompile.bat tool to generate a client proxy. If you use Apache Axis, you can use the java org.apache.axis.wsdl.WSDL2Java Java class

## 1.6.5 Generating Web Services for .NET

PowerDesigner supports generating Web services for C# and VB.NET.

## Context

The following items are generated:

- An .ASMX file - an ASP.NET file, it contains the code of the C# or VB.NET Web service class.
- An implementation class (C# or VB.NET) with special super class and WebMethod property for the methods. If you disable the Generate Web Service C# code in .asmx file option, a C# or VB .NET class will also be generated for each Web Service. It is not necessary to define the super class (also known as WebService) for the Web service classes; if it is not defined, the code generator adds it to the code.

## Procedure

1. Select **Language** or to open the Generation dialog.
2. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see [Checking an OOM \[page 251\]](#)). You can select a Microsoft Internet Information Server (IIS) directory for generation, for example, C:\Inetpub\wwwroot\StockQuote. If you have defined your Web services inside a package, you can generate the Web service code in the C:\Inetpub\wwwroot directory. Each package will create a subdirectory.
3. [optional] Click the **Selection** tab and specify the objects that you want to generate from. By default, all objects are generated.
4. [optional] Click the **Options** tab and set any appropriate generation options:  
Set the Generate Web Service C# code in .asmx file option to false if you want to generate the C# or VB .NET class outside a .asmx file.
5. [optional] Click the **Tasks** tab and specify any appropriate generation tasks to perform:

| Option                                  | Description   |
|---|---|
| Compile source files                    | Compiles the generated code.  |
| Generate Web service proxy code         | Generates the Web Service proxy class for a Web Service component instance.<br><br>You need to define a component instance for the Web Service deployment URL<br><br><b>Note</b><br>You must define a WSDL variable to indicate where the wsdl.exe program is located in the <b>Tools</b> <b>General Options</b> <b>Variables</b> category. |
| Open the solution in Visual Studio .NET | If you selected the Generate Visual Studio .NET project files option, this task allows to open the solution in the Visual Studio .NET development environment   |

6. Click **OK** to begin generation.  
  
The code generation process creates a subdirectory under wwwroot using the package name, and creates a <WebServiceName>.ASMX file within the subdirectory.
7. To deploy a .NET Web Service, install Microsoft Internet Information Server (IIS) along with the .NET Framework. Then, simply copy the .asmx file and the C# or VB .NET class files under the IIS directory C:\Inetpub\wwwroot\<PackageName>. For example: C:\Inetpub\wwwroot\StockQuote.
8. To test the Web service, enter the URL of the Web service in the browser: http://[HostName]/[PackageName]/[ServiceName].asmx. For example: http://localhost/StockQuote/StockQuote.asmx.

If the input parameters and the return value use simple data types, the IIS Web server will generate a testing tab to let you test the deployed Web service. To test Web services with complex data types, create a testing program using Web service proxy or use a tool to send a SOAP message to the Web service.

## 1.6.6 Importing WSDL Files

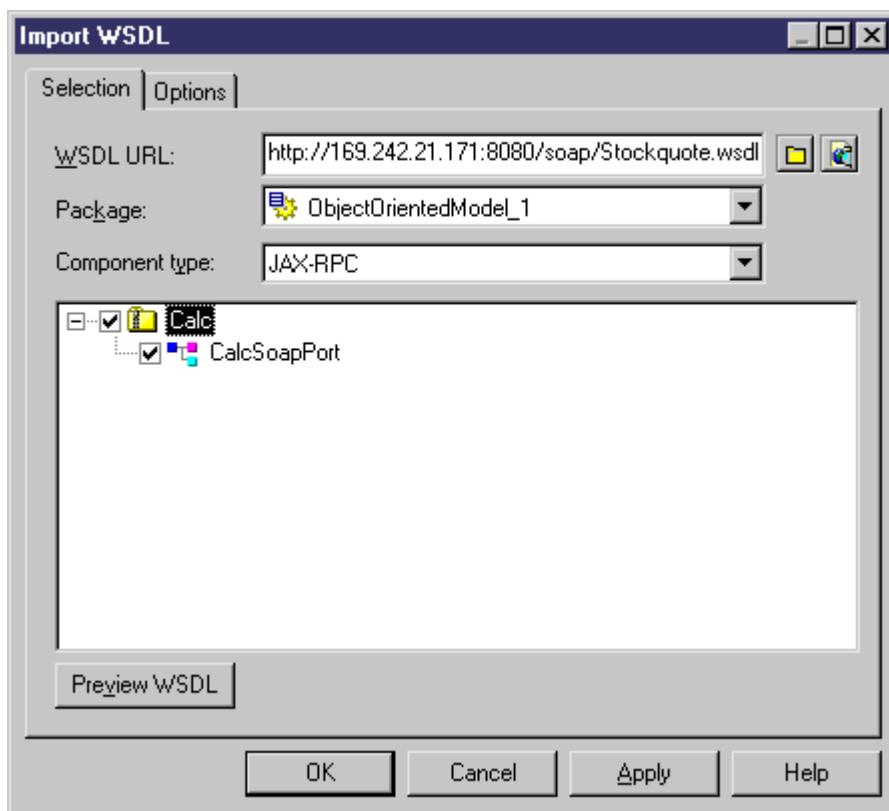
PowerDesigner can import WSDL files for .NET and Java.

### Procedure

1. Select **Language > Import WSDL** to open the Import WSDL dialog.
2. On the *Selection* tab, complete the following fields:

| Item           | Description   |
|----------------|---|
| WSDL URL       | Indicates the location of the WSDL file. You can complete this field by: <ul style="list-style-type: none"><li>o Entering the location directly in the field</li><li>o Clicking the <i>Browse File</i> tool to browse on your local file system</li><li>o Clicking the Browse UDDI tool to search on a UDDI server (see <a href="#">Browsing WSDL Files from UDDI [page 230]</a>)</li></ul> |
| Package        | Specifies the package and namespace where the component and the Web service class will be created.  |
| Component type | [Java only] Specifies the type of the component to create.  |

3. Select the Web services and port types you want to import.



Each Web service selected will be imported as a component and an implementation class. Each port type selected in a selected Web service generates an interface.

4. [optional] Click the *Preview WSDL* button to preview the WSDL and the unique key used to locate the UDDI.
5. [optional] Click the *Options* tab, which allows you to specify in which diagrams PowerDesigner should create the symbols for the imported objects. Deselecting an option will suppress the creation of a symbol, but the object will still be imported.
6. Click *OK* to begin the import.

A progress box is displayed. If the model in which you are reverse engineering already contains data, the Merge Models dialog opens to allow you to select how the imported objects will be merged with your model

For detailed information about merging models, see *Core Features Guide > Modeling with PowerDesigner > Comparing and Merging Models*.

Each Web service selected will be imported as a component and an implementation class. Each port type selected in a selected Web service generates an interface.

#### i Note

If the WSDL contains a section prefixed with <!-- service -->, a component instance is created. This section is displayed in the WSDL tab in the property sheet of the component instance.

### 1.6.6.1 Browsing WSDL Files from UDDI

UDDI is an XML-based registry for businesses worldwide, which lists all Web services on the Internet and handles their addresses. PowerDesigner provides an interface for browsing for a WSDL directly on a UDDI server.

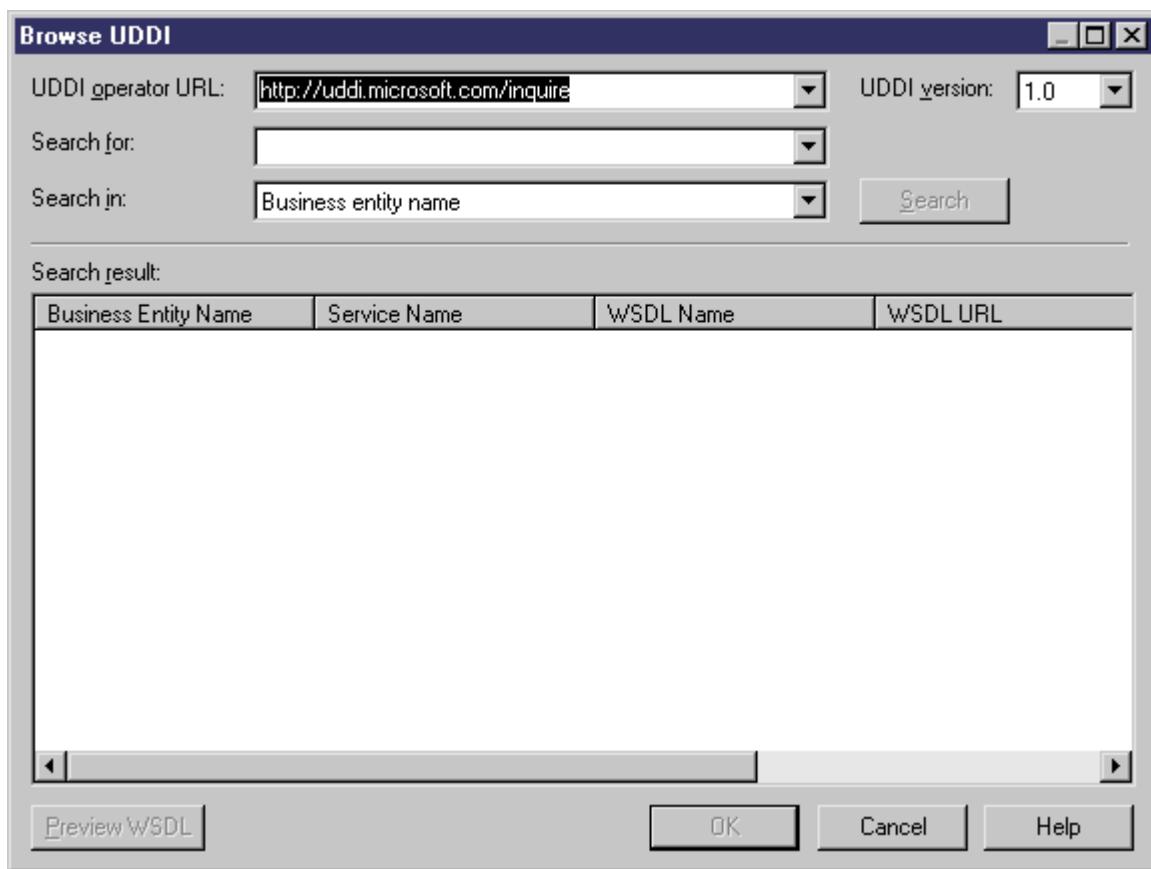
#### Context

In UDDI, an organization or a company, called a businessEntity usually publishes a WSDL to describe Web services interfaces as tModel. Another company may implement it, and then publish the following items that describe how to invoke the Web service in the UDDI:

- The company, called businessEntity
- The service, called businessService
- The access endpoints, called the bindingTemplate
- The specification, called the tModel

#### Procedure

1. Click the *Browse UDDI* tool to the right of the WSDL URL field on the *Selection* tab of the Import WSDL dialog.



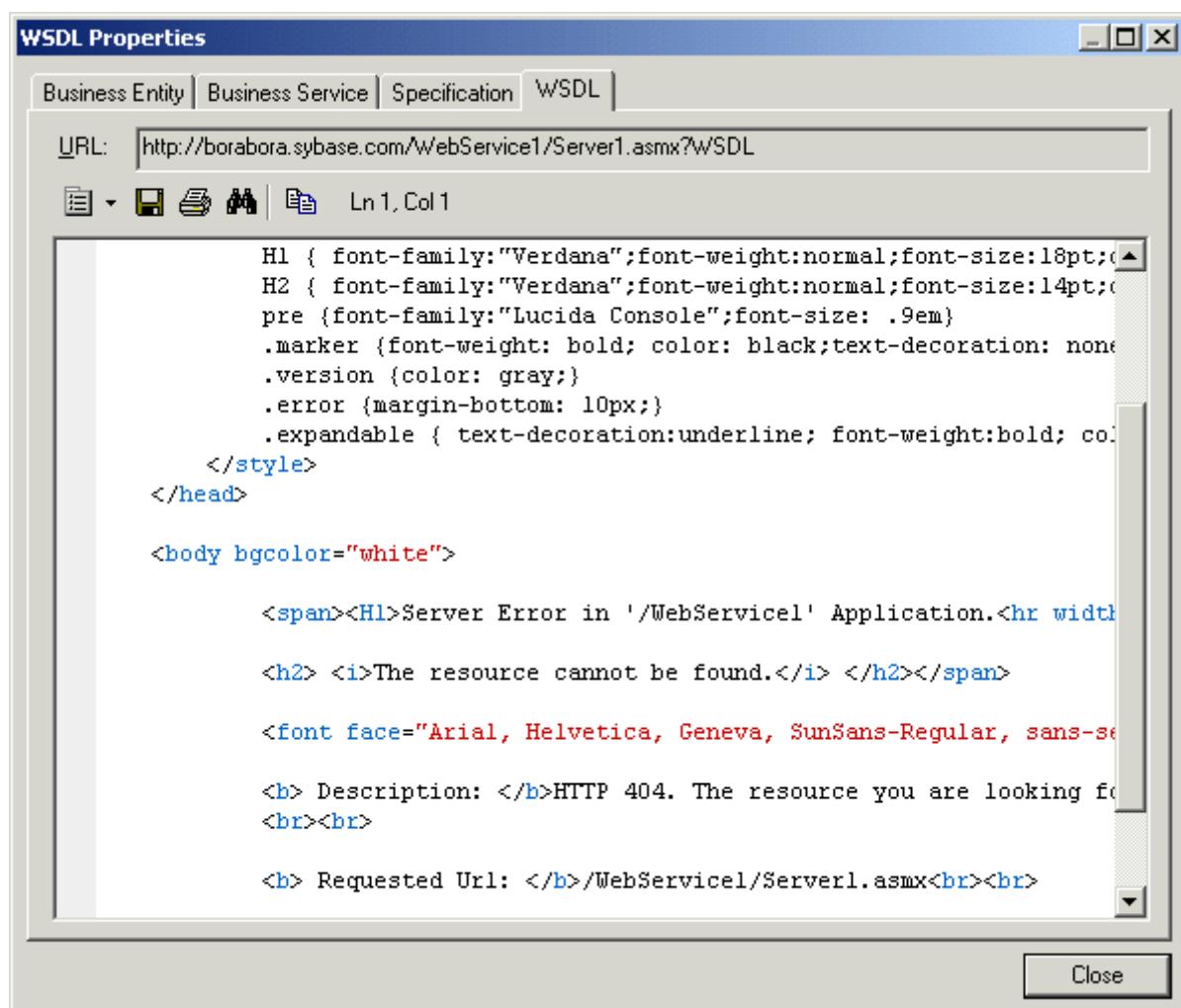
2. Complete the following fields to specify your search criteria:

| Item              | Description  |
|-------------------|--|
| UDDI operator URL | Choose from a list of default UDDI operator URLs, or enter your own URL.                         |
| UDDI version      | Specify the correct UDDI version for the URL.  |
| Search for        | Specify the name of the item to search for.  |
| Search in         | Specify whether to search on the business entity (company name), Web service name, or WSDL name. |

3. Click the *Search* button.

The result is displayed in the Search Result window.

4. [optional] Click the *Preview WSDL* button to open the WSDL property sheet, which contains various tabs allowing you to view information about the business entity and service, the specification and the WSDL code:



5. Click **Close** to return to the Browse UDDI dialog.
6. Click **OK** to return to the Import WSDL dialog to complete the import.

## 1.7 Generating and Reverse Engineering OO Source Files

PowerDesigner can generate and reverse engineer source files from and to an OOM.

### 1.7.1 Generating OO Source Files from an OOM

PowerDesigner provides a standard interface for generating source files for all the supported OO languages. For details of language-specific options and generation tasks, see the appropriate language chapter.

#### Context

By default, PowerDesigner supports the generation of the following types of objects for the languages supported by the OOM:

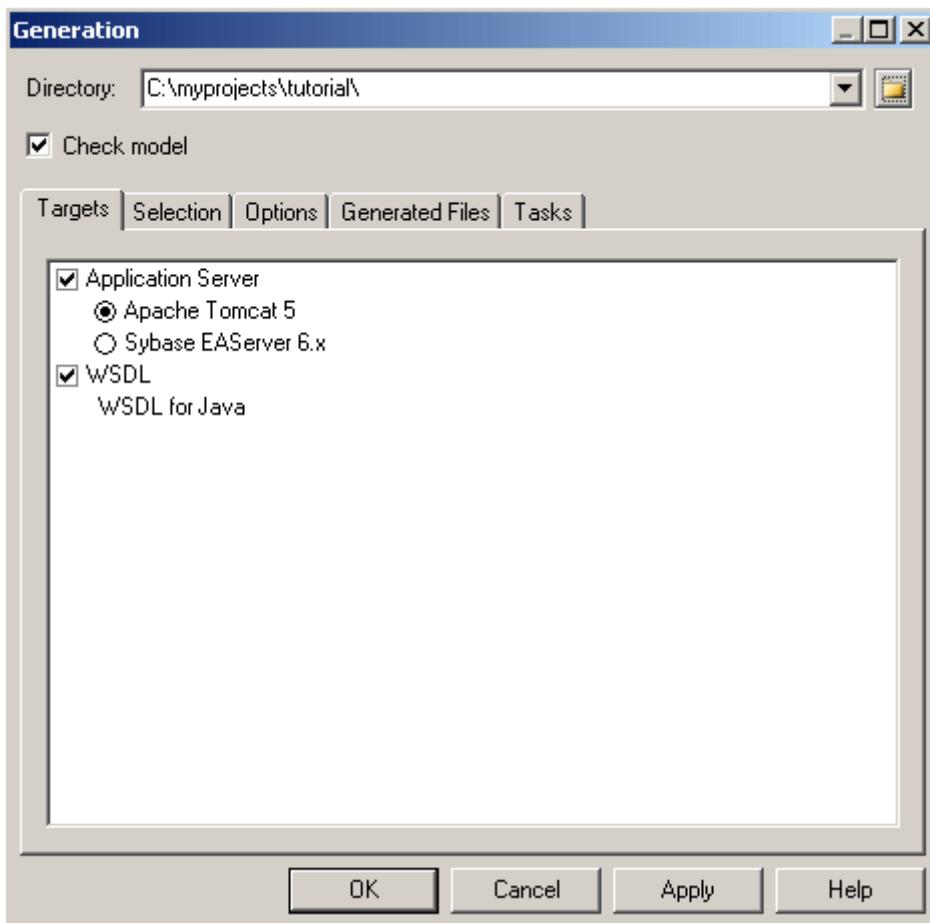
| Object language  | What is generated   |
|------------------|---|
| Analysis         | No files generated as this language is mainly used for modeling purpose               |
| C#               | .CS definition files  |
| C++              | C++ definition files (.h and .cpp)  |
| IDL-CORBA        | IDL-CORBA definition files  |
| Java             | Java files from classes and interfaces of the model. Includes support of EJB and J2EE |
| PowerBuilder     | .PBL application or .SRU files from classes of the model                              |
| Visual Basic.Net | .VB files   |
| XML-DTD          | .DTD files  |
| XML-Schema       | .XSD files. Includes standard XML language properties                                 |

#### i Note

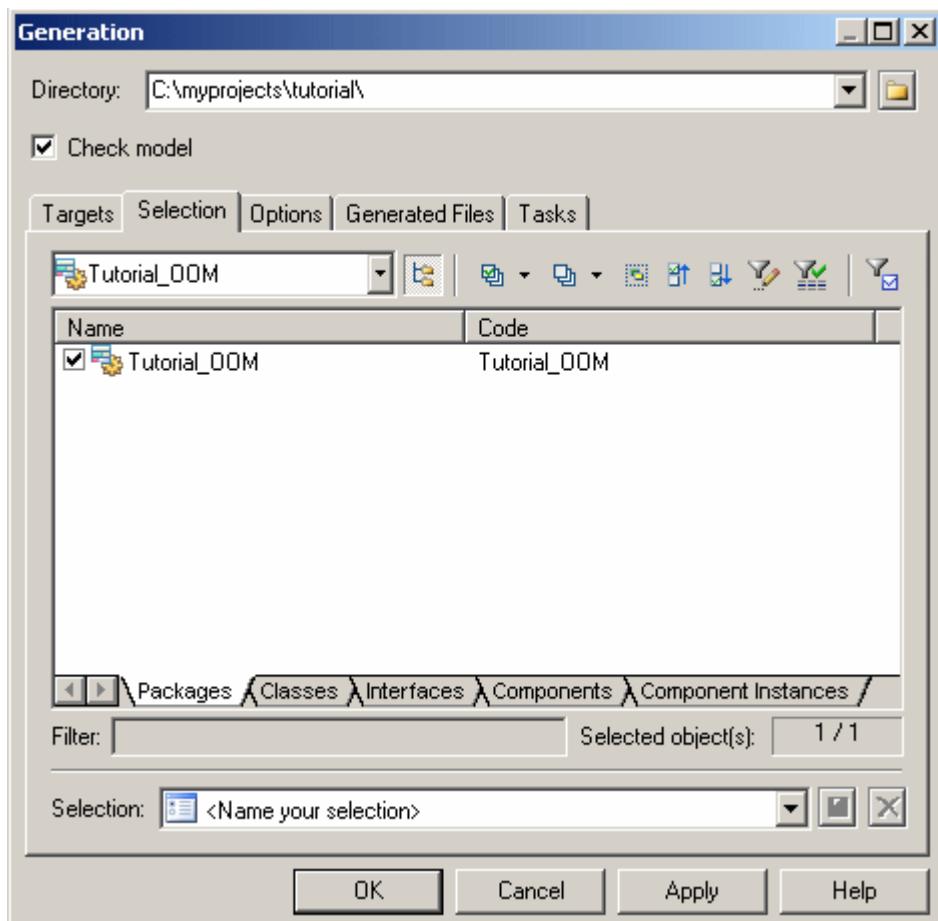
The PowerDesigner generation system is extremely customizable through the use of extensions (see [Extending your Modeling Environment \[page 19\]](#)). For detailed information about customizing generation, including adding generation targets, options, and tasks, see *Customizing and Extending PowerDesigner > Extension Files*.

## Procedure

1. Select **Language > Generate <language> Code** to open the Generation dialog box:



2. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see [Checking an OOM \[page 251\]](#)).
3. [optional] Select any additional targets to generate for. These targets are defined by any extensions that may be attached to your model (see [Working with Generation Targets \[page 236\]](#)).
4. [optional] Click the **Selection** tab and specify the objects that you want to generate from. By default, all objects are generated, and PowerDesigner remembers for any subsequent generation the changes you make.



- [optional] Click the *Options* tab and set any necessary generation options. For more information about these options, see the appropriate language chapter.

**i Note**

For information about modifying the options that appear on this and the *Tasks* tab and adding your own options and tasks, see *Customizing and Extending PowerDesigner > Object, Process, and XML Language Definition Files > Generation Category*.

- [optional] Click the *Generated Files* tab and specify which files will be generated. By default, all files are generated.

For information about customizing the files that will be generated, see *Customizing and Extending PowerDesigner > Extension Files > Generated Files (Profile)*.

- [optional] Click the *Tasks* tab and specify any additional language-specific generation tasks to perform.
- Click *OK* to begin generation.

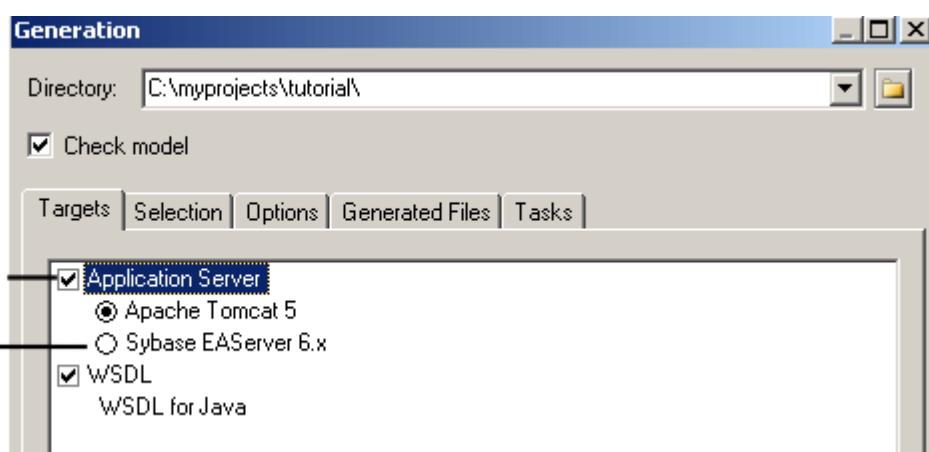
When generation is complete, the Generated Files dialog opens, listing the files that have been generated to the specified directory. Select a file in the list and click *Edit* to open it in your associated editor, or click *Close* to exit the dialog.

## 1.7.1.1 Working with Generation Targets

The Targets tab of the Generation dialog box allows you to specify additional generation targets, which are defined by extension files.

PowerDesigner provides many extensions, which can extend the object language for use with a particular server, framework, etc. You can modify these extensions or create your own. For information about attaching extensions to your model, see [Extending your Modeling Environment \[page 19\]](#).

The Generation dialog Targets tab groups targets by category. For each category, it is only possible to select one extension at a time.



For detailed information about editing and creating extensions, see *Customizing and Extending PowerDesigner > Extension Files* .

## 1.7.1.2 Defining the Source Code Package

For those languages that support the concept of packages and/or namespaces, classes must be generated in packages that are used as qualifying namespace. You can define these qualifying packages one by one in the model as necessary, or insert a base structure automatically via the Add Package Hierarchy command.

### Procedure

1. Right-click the Model in the Browser, and select Add Package Hierarchy from the contextual menu.
2. Enter a package hierarchy in the text field, using periods or slashes to separate the packages. For example:

## Results

```
com.mycompany.myproduct.oom
```

or

```
com/mycompany/myproduct/oom
```

The corresponding package hierarchy will be created in the Browser. All diagrams and objects (except global objects) existing in the model will be moved to the lowest level package of the hierarchy.

## 1.7.2 Reverse Engineering OO Source Files into an OOM

Reverse engineering is the process of extracting data or source code from a file and using it to build or update an OOM. You can reverse engineer objects to a new model, or to an existing model.

You can reverse the following types of files into an OOM:

- Java
- IDL
- PowerBuilder
- XML - PowerDesigner uses a parser developed by the Apache Software Foundation (<http://www.apache.org> ).
- C#
- VB
- VB.NET

## Inner Classifiers

When you reverse a language containing one or more inner classifiers (see [Creating Composite and Inner Classifiers \[page 54\]](#)) into an OOM, one class is created for the outer class, and one class is created for each of the inner classifiers, and an inner link is created between each inner classifier and the outer class.

## Symbol Creation

If you select the Create Symbols reverse option, the layout of the symbols in the diagram is automatically arranged. When reverse engineering a large number of objects with complex interactions, auto-layout may create synonyms of objects to improve the diagram readability.

## 1.7.2.1 Reverse Engineering OO Files into a New OOM

You can reverse engineer object language files to create a new OOM.

### Procedure

1. Select  *File*  *Reverse Engineer*  to open the New Object-Oriented Model dialog box.
2. Select an object language in the list and click the *Share* radio button.
3. [optional] Click the *Select Extensions* tab, and select any extensions you want to attach to the new model.

For detailed information about working with extensions, see *Customizing and Extending PowerDesigner > Extension Files*.

4. Click *OK* to go to the appropriate, language-specific Reverse Engineering window. For detailed information about this window for your language see the appropriate language chapter.
5. Select the files that you want to reverse and the options to set, and then click *OK* to start reverse engineering.

A progress box is displayed. The classes are added to your model.

### Results

#### Note

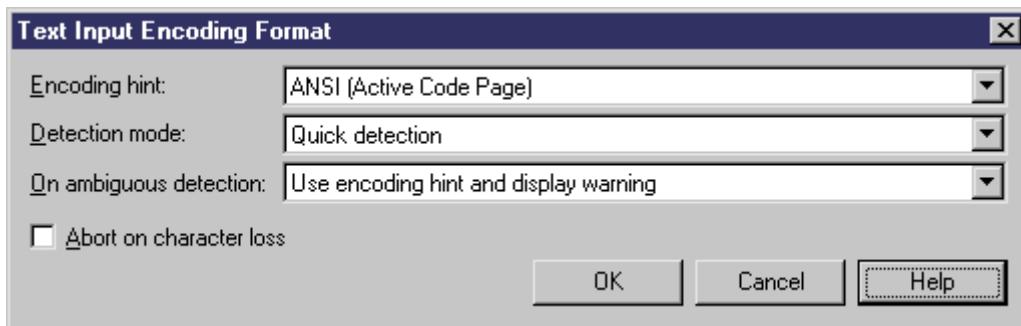
This product includes XML4C 3.0.1 software developed by the Apache Software Foundation (<http://www.apache.org> 

Copyright (c) 1999 The Apache Software Foundation. All rights reserved. THE XML4C 3.0.1 SOFTWARE ("SOFTWARE") IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### 1.7.2.1.1 Reverse Engineering Encoding Format

If the applications you want to reverse contain source files written with Unicode or MBCS (Multibyte character set), you should use the encoding parameters provided to you in the File Encoding box.

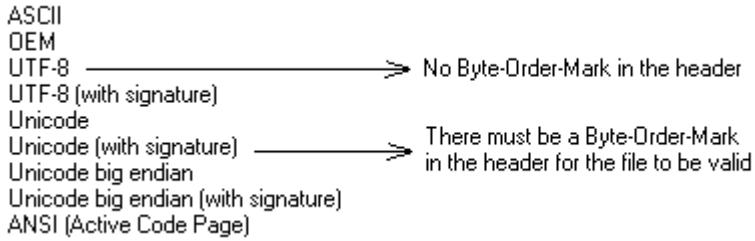
If you want to change these parameters because you know which encoding is used within the sources, you can select the appropriate encoding parameter by clicking the Ellipsis button beside the File Encoding box. This opens the Text Input Encoding Format dialog box in which you can select the encoding format of your choice.



The Text Input Encoding Format dialog box includes the following options:

| Option                  | Description  |
|-------------------------|--|
| Encoding hint           | Encoding format to be used as hint when reversing the file   |
| Detection mode          | Indicates whether text encoding detection is to be attempted and specifies how much of each file should be analyzed. You can select from the following options: <ul style="list-style-type: none"><li>No detection - Turns off the detection feature. Select this option when you know what the encoding format is</li><li>Quick detection - Analyzes a small buffer to perform detection. Select this option when you think that the encoding format will be easy to detect</li><li>Full detection - Analyzes the whole file to perform detection. Select this option when you think that the number of characters that determine the encoding format is very small</li></ul> |
| On ambiguous detection  | Specifies what action should be taken in case of ambiguity. You can select from the following options: <ul style="list-style-type: none"><li>Use encoding hint and display warning - the encoding hint is used and a warning message is displayed in the Reverse tab of the Output window</li><li>Use encoding hint - uses the encoding format selected in the Encoding Hint box, if possible. No warning message is displayed</li><li>Use detected encoding - Uses the encoding format detected by PowerDesigner</li></ul>  |
| Abort on character loss | Allows you to stop reverse engineering if characters cannot be identified and are to be lost in current encoding   |

Here is an example on how to read encoding formats from the list:



## 1.7.2.2 Reverse Engineering into an Existing OOM

You can reverse engineer source files to add objects to an existing OOM.

### Procedure

1. Select *Language* *Reverse Engineer* to display Reverse Engineering dialog box.
2. Select to reverse engineer files or directories from the *Reverse Engineering* list.
3. Click the add button in the *Selection* tab to display a standard Open dialog box.
4. Select the files or directory to reverse engineer and click *Open*.
5. Click *OK* to begin reverse engineering.

A message in the Output window indicates that the specified file is fully reverse engineered and the Merge Models window opens.

6. Review the objects that you will be importing, and the changes that they will make to the model.

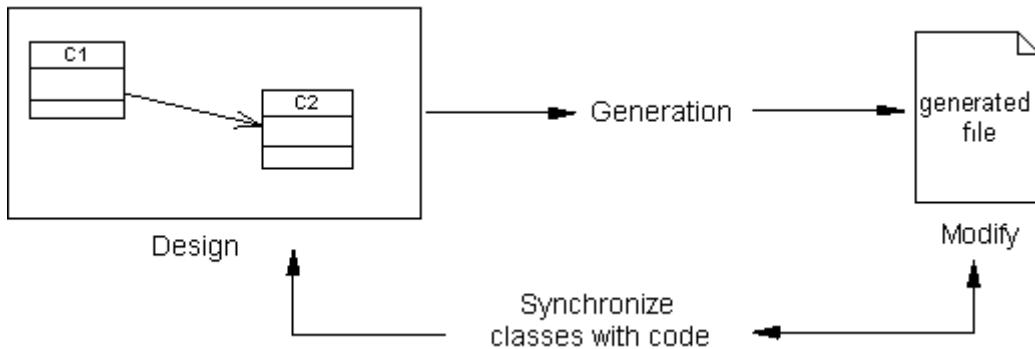
For more information on merging models, see *Core Features Guide > Modeling with PowerDesigner > Comparing and Merging Models*.

7. Click *OK* to merge the selected changes into your model.

## 1.7.3 Synchronizing a Model with Generated Files

You can design your system in PowerDesigner, use the generation process, then visualize and modify the generated file in your code editor, synchronize the classifiers with the source code and then go back to the model. With this feature, you can modify the generated file and reverse in the same generated file.

### Context



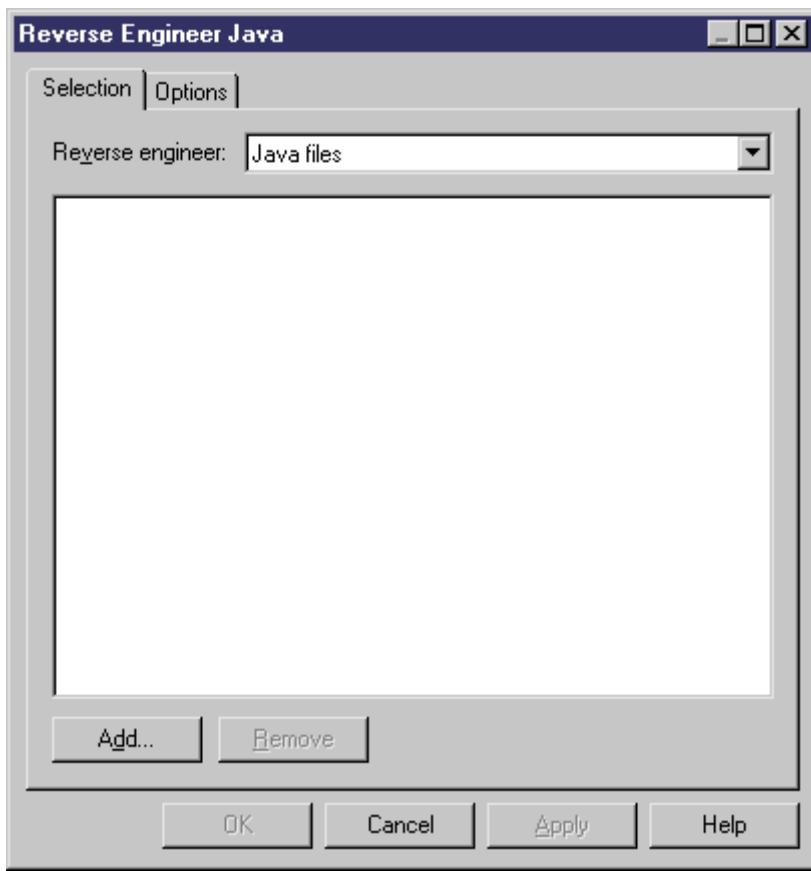
The synchronization launches a reverse engineering dialog box, pre-selects option, and fills the list of classifiers with the classifiers selected in the class diagram.

You can then easily locate the files that should be taken into account for synchronization. If there is no classifier selected, the reverse feature pre-selects directories and adds the current directory to the list.

## Procedure

1. Select **Language > Synchronize with generated files** to display the Reverse dialog box.

The **Selection** tab is displayed.



2. Select to reverse engineer files or directories from the Reverse Engineering list.
3. Click the *Add* button to open the Browse for Folder dialog box.
4. Select the appropriate directory, and click *OK* to open the Reverse Java dialog box you need.
5. Click *OK* to begin synchronization.

A progress box is displayed, followed by the Merge Models dialog box.

**i Note**

The Merge Models dialog box shows the *From Model* (source directory) in the left pane, and the *To Model* (current model) in the right pane. You can expand the nodes in the *To Model* pane to verify that the merge actions selected correspond to what you want to perform.

6. Review the objects that you will be importing, and the changes that they will make to the model, and then click *OK*.

The *Reverse* tab of the Output window displays the changes which occurred during synchronization and the diagram window displays the synchronized model.

## 1.8 Generating Other Models from an OOM

You can generate conceptual and physical data models and XML models from an OOM.

### Context

The following table details how OOM objects are generated to other models:

| OOM   | CDM  | PDM  | XSM   |
|---|--|--|---|
| Domain  | Domain   | Domain   | Simple Type                                   |
| Class (Persistent and Generate check boxes selected).                             | Entity   | Table (The cardinality of a class becomes the number of records of a table.)                           | Element                                       |
| Abstract class  | Entity   | Table  | Complex type                                  |
| Attribute (Persistent check box selected)   | Attribute  | Column   | Attribute or element (see generation options) |
| Identifier  | Identifier   | Identifier   | Key   |
| Composition   | -  | -  | New level in the element hierarchy            |
| Operation with <<storedProcedure>> stereotype (parent class generated as a table) | -  | Stored procedure attached to the table, with the operation body as a body in the procedure definition. | -   |
| Operation with <<storedFunction>> stereotype (parent class generated as a table)  | -  | Stored function attached to the table, with the operation body as a body in the function definition.   | -   |
| Association   | Relationship or association  | Table (if many-to-many multiplicity) or reference. Role names become migrated foreign keys.            | KeyRef constraints                            |
| Association class   | Entity/relationship notation: entity with two associations.<br>Merise notation: association, and two association links | A table and two associations between the end points of the association class                           | -   |

| OOM            | CDM         | PDM       | XSM   |
|----------------|-------------|-----------|---|
| Dependency     | -           | -         | -   |
| Realization    | -           | -         | -   |
| Generalization | Inheritance | Reference | Complex type derivation (XSD) or attribute migration to child element (DTD) |

## Procedure

1. Select *Tools*, and then one of the following to open the appropriate Model Generation Options Window:
  - o *Generate Conceptual Data Model...* `Ctrl` + `Shift` + `C` - For example, to translate OOM classes into CDM entities. You will then be able to further refine your model and eventually generate a Physical Data Model (PDM) from the CDM.
  - o *Generate Physical Data Model...* `Ctrl` + `Shift` + `P` - For example, to translate the design of your system to your database. This allows you to model the objects in the world they live in and to automate the translation to database tables and columns.
  - o *Generate Object-Oriented Model...* `Ctrl` + `Shift` + `O` - For example, to transform an analytical OOM (designed with the Analysis object language) to implementation OOMs designed for Java, C#, and any other of the object languages supported by PowerDesigner.
  - o *Generate XML Model...* `Ctrl` + `Shift` + `M` - For example, to generate a message format from your class structure.
2. On the *General* tab, select a radio button to generate a new or update an existing model, and complete the appropriate options.
3. [optional] Click the *Detail* tab and set any appropriate options. We recommend that you select the Check model checkbox to check the model for errors and warnings before generation.
4. [optional] Click the *Target Models* tab and specify the target models for any generated shortcuts.
5. [optional] Click the *Selection* tab and select or deselect objects to generate.
6. Click *OK* to begin generation.

## Results

### i Note

For detailed information about the options available on the various tabs of the Generation window, see *Core Features Guide > Linking and Synchronizing Models > Generating Models and Model Objects*.

## 1.8.1 Managing Object Persistence During Generation of Data Models

Developers use object-oriented programming languages to develop business objects that will be stored in a database. PowerDesigner provides various properties to give you precise control over the generation of persistent objects into a data model.

Sometimes, the class and attribute codes in object-oriented programming languages (specified in the *Code* field under the *Name* field on the *General* tab of their property sheets) are different to the codes used for tables and columns in the data model representing the database.

In these cases, you can specify an alternative, persistent, code in the *Code* field in the *Persistent* groupbox on the *Detail* tab of classes and attributes. These codes will be used in place of the standard codes during generation of a data model and also facilitate round-trip engineering by recovering object codes from the database.

The other properties in these *Persistent* groupboxes, help you control how classes will be generated in data models (see [Class Properties \[page 39\]](#)) and the data types to be used for attributes (see [Attribute Properties \[page 69\]](#))

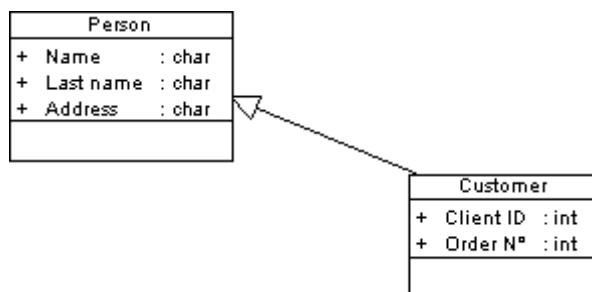
### i Note

You can also create object-to-relational mappings between OOM and CDM or PDM objects using the mapping editor (see *Core Features Guide > Linking and Synchronizing Models > Object Mappings*).

## 1.8.2 Managing Persistence for Generalizations

You can control the generation of classes connected by a generalization link into CDM entities or PDM tables using the *Generate table* and *Migrate columns* options in the *Persistent* groupbox on the *Detail* tab.

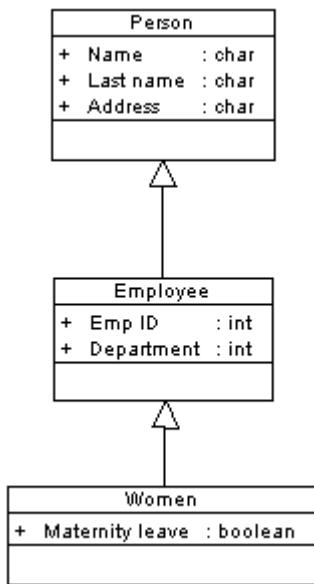
In the following example, **Customer** is set to *Generate table* and inherits, via a generalization link, from **Person**, which is set to *Migrate columns*:



Customer inherits the attributes of the parent class in the generated PDM:

| Customer  |         |
|-----------|---------|
| Client ID | integer |
| Order N°  | integer |
| Name      | char    |
| Last name | char    |
| Address   | char    |

Derived classes are created to improve the readability of a model but add no semantic information and are not generated in a PDM, their attributes being migrated to the parent. In the following example, **Women** and **Person** are both set to *Migrate columns*, while **Employee** is set to *Generate table*:



In the generated PDM, **Employee** inherits from both its parent class and the derived class:

| Employee        |          |
|-----------------|----------|
| Emp ID          | integer  |
| Department      | integer  |
| Name            | char     |
| Last name       | char     |
| Address         | char     |
| Maternity leave | smallint |

For more information, see [Class Properties \[page 39\]](#) and [Generalizations \(OOM\) \[page 93\]](#).

### 1.8.3 Managing Persistence for Complex Data Types

When you specify a class as the data type of an attribute, you can control its generation to a CDM or PDM using the *Generate table*, *Value Type* and *Generate ADT* (PDM only) options in the *Persistent* groupbox on the *Detail* tab.

In the following example, **Customer** contains an attribute, **address**, for which the class **Address**, has been selected as data type (see [Specifying a Classifier as a Data Type or Return Type \[page 55\]](#)):

| Address   | Customer   |
|---|--|
| + line1 : String<br>+ line2 : String<br>+ city : String<br>+ zipCode : String<br>+ country : String | + number : int<br>+ lastName : String<br>+ company : String<br>+ address : ADDRESS |

Customer is specified as persistent, and the *Generate table* option is selected. You can generate the class **Address** in any of the following ways:

- As an embedded class - by selecting *Value Type* in the *Persistent* groupbox on the *Detail* tab (see [Class Properties \[page 39\]](#)):

In a PDM, **Customer** is generated as a table with all the attributes of **Address** embedded in it as columns prefixed by **address\_**:

| Customer        |              |
|-----------------|--------------|
| number          | integer      |
| lastName        | long varchar |
| company         | long varchar |
| address_line1   | long varchar |
| address_line2   | long varchar |
| address_city    | long varchar |
| address_zipCode | long varchar |
| address_country | long varchar |

In a CDM, both classes are generated as entities, and **Customer** includes all the attributes of **Address** embedded in it as attributes prefixed by **address\_**:

| Address |     |
|---------|-----|
| line1   | TXT |
| line2   | TXT |
| city    | TXT |
| zipCode | TXT |
| country | TXT |

| Customer        |     |
|-----------------|-----|
| number          | I   |
| lastName        | TXT |
| company         | TXT |
| address_line1   | TXT |
| address_line2   | TXT |
| address_city    | TXT |
| address_zipCode | TXT |
| address_country | TXT |

### i Note

If you specify a multiplicity (see [Attribute Properties \[page 69\]](#)) for the attribute using a complex data type, attributes are generated the maximum number of times required by the multiplicity, but only if the maximum is set to a fixed value. In the following example, attribute multiplicity is 0..2, so attributes will be embedded twice:

| Address |      |
|---------|------|
| line1   | LONG |
| line2   | LONG |
| city    | LONG |
| zipCode | LONG |
| country | LONG |

| Customer         |         |
|------------------|---------|
| number           | INTEGER |
| lastName         | LONG    |
| company          | LONG    |
| address1_line1   | LONG    |
| address1_line2   | LONG    |
| address1_city    | LONG    |
| address1_zipCode | LONG    |
| address1_country | LONG    |
| address2_line1   | LONG    |
| address2_line2   | LONG    |
| address2_city    | LONG    |
| address2_zipCode | LONG    |
| address2_country | LONG    |

- As an Abstract Data Type class (PDM only) - by selecting *Generate ADT* in the *Persistent* groupbox:

In a PDM, **Customer** is generated as a table and **Address** is generated as an abstract data type (which does not have a symbol), which is referenced by the column **address**:

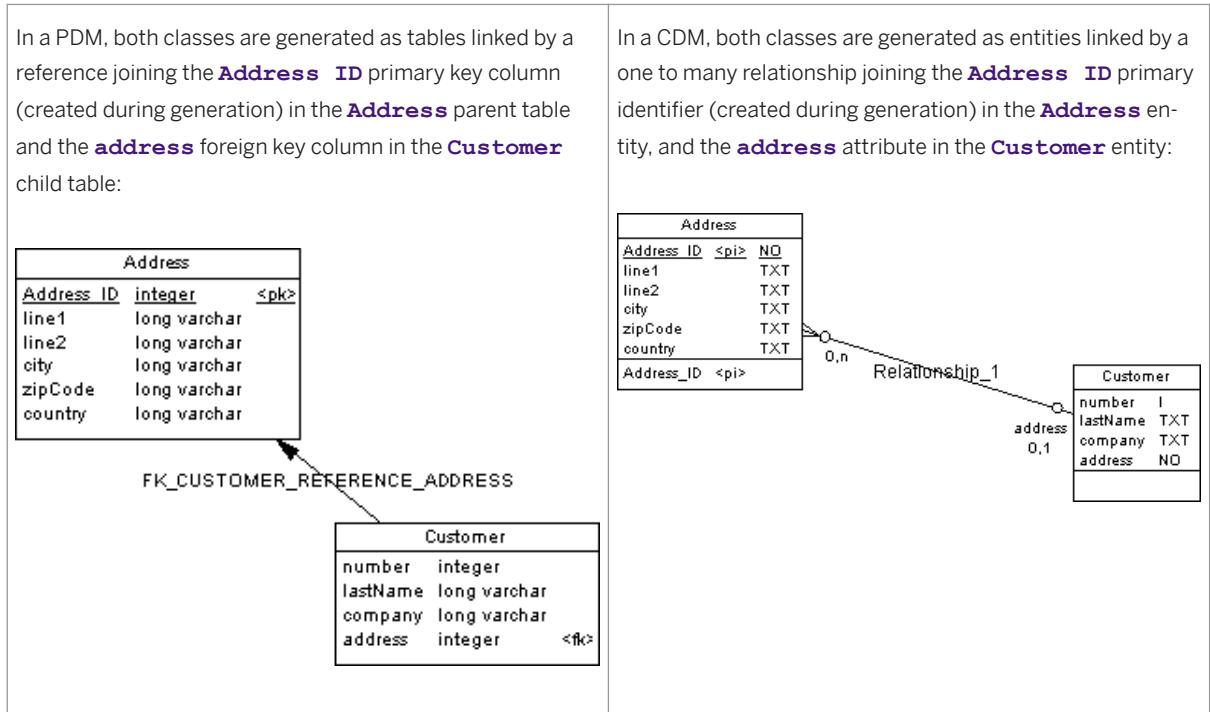
| Customer |         |
|----------|---------|
| number   | INTEGER |
| lastName | LONG    |
| company  | LONG    |
| address  | Address |

### i Note

If you specify a multiplicity for the attribute using a complex data type, for DBMSs that support ARRAY and LIST for abstract data types, multiplicity affects generation as follows:

- 0..n or 1..n - generate as an abstract data type of type LIST (example: TABLE for Oracle).
- 1..2 or 0..10 - generate as an abstract data type of type ARRAY (example: VARRAY for Oracle).
- 0..1 or 1..1 - generate as an abstract data type of type OBJECT.

- As a persistent class - by drawing an association between the classes, right-clicking the association and selecting *Migrate Navigable Roles*, and selecting *Generate table* in the *Persistent* groupbox:



#### **i Note**

If you specify a multiplicity, the multiplicity is generated as a cardinality on the reference between the tables.

### 1.8.4 Customizing XSM Generation for Individual Objects

When generating an XSM from a PDM or OOM, you can specify global generation options to generate tables/classes as elements with or without complex types and columns/attributes as elements or attributes. You can override these options for individual objects by attaching the **PDM XML Generation** or **OOM XML Generation** extension to your source model and selecting from their XML generation options.

#### Context

##### **i Note**

The extension provides new property sheet tabs for setting generation options for individual objects, but you can also set these options with or without the extension by selecting **Model > <objects>** to open the appropriate object list, clicking the **Customize Columns and Filter** tool, and selecting to display the **XML Generation Mode** column.

For example, if you want to generate the majority of your table columns to an XSM as XML attributes, but want to generate certain columns as elements, you should:

- Modify the XML generation options for those columns that you want to generate as elements.
- Select to generate columns as attributes on the Model Generation Options *Detail* tab.

## Procedure

1. Select to open the List of Extensions, and click the *Attach an Extension* tool.
2. On the *General Purpose* tab, select **PDM XML Generation** or **OOD XML Generation** and click **OK** to attach the extension to your model and **OK** to close the List of Extensions.

These extension files enable the display of the **XML** tab in all table and column or class and attribute property sheets.
3. Open the property sheet of the table, column, class, or attribute whose generation you want to customize, and click the **XML** tab.
4. Use the radio buttons to specify how you want to generate the object in an XSM.
  - For tables and classes, you can specify to generate them as:
    - Elements - the table/class is generated as an untyped element directly linked to its columns/attributes generated as attributes or sub-elements.
    - Elements with complex types - the table/class is generated as an element typed by a complex type, generated in parallel, to contain the columns/attributes.
    - Default - generation of the table/class is controlled by the option selected in the *XML Generation* group box on the Model Generation Options *Detail* tab.
  - For tables, you can additionally specify to generate keys as:
    - Key - [default] The primary key columns are generated and also KEY and KEYREF wherever the table is referenced.
    - ID attribute - The primary key columns are not generated and an ID attribute, **id**, is generated to replace them.

Wherever the table is referenced, an IDREF attribute is generated to reference the appropriate element. If the reference role name is assigned, this attribute is given this name. Otherwise, the referenced table name is used and the standard renaming mechanism is enforced.
    - Key and ID attribute - In many cases the primary key columns have significant data and you may want to generate them, as well as an ID attribute. In this case an ID attribute is generated for the element and IDREF is used systematically for any reference to the table:

The following rules apply to the generation of keys:

- If a Table generates an ID, all its child tables will generate an ID attribute.
- If a Table generates Key columns, all its child tables will generate Key columns.
- If a child table is flagged to generate PK only, ID Attribute will be automatically generated.
- If a table generates ID attribute, No Key nor KeyRef will be generated, and ALL references will generate IDREF attribute.. (Even if the table generates also Key Columns)
- If a table generates ID attribute ONLY, All Foreign Key Columns referencing its Key columns will be systematically removed and replaced by an IDREF attribute
- For columns and attributes, you can specify to generate them as:

- Elements - [default] the column/attribute is generated as an sub-element of its table/class element or complex type.
  - Attributes - the column/attribute is generated as an attribute of its table/class element or complex type.
  - Default - generation of the column/attribute is controlled by the option selected in the *XML Generation* group box on the Model Generation Options *Detail* tab.
5. Modify the XML generation options for any other objects that you want to generate in a different manner.
  6. Select *Tools* *Generate XML Model*, ensure that the appropriate options are set in the *XML Generation* group box on the Model Generation Options *Detail* tab, and start your generation.

## 1.9 Checking an OOM

The object-oriented model is a very flexible tool, which allows you quickly to develop your model without constraints. You can check the validity of your OOM at any time.

A valid OOM conforms to the following kinds of rules:

- Each object name in a OOM must be unique
- Each class in a OOM must have at least one attribute and operation
- Each start or end must be linked to an object of the diagram

### Note

We recommend that you check your object-oriented model before generating code or another model from it . If the check encounters errors, generation will be stopped. The *Check model* option is enabled by default in the Generation dialog box.

You can check your model in any of the following ways:

- Press F4, or
- Select *Tools* *Check Model*, or
- Right-click the diagram background and select Check Model from the contextual menu

The Check Model Parameters dialog opens, allowing you to specify the kinds of checks to perform, and the objects to apply them to. The following sections document the OOM-specific checks available by default. For information about checks made on generic objects available in all model types and for detailed information about using the Check Model Parameters dialog, see *Core Features Guide > Modeling with PowerDesigner > Objects > Checking Models* .

## 1.9.1 Domain Checks

PowerDesigner provides default model checks to verify the validity of domains.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary                  | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: None.</li></ul>  |
| Name/Code contains synonyms of glossary terms             | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: Replaces synonyms with their associated glossary terms.</li></ul>   |
| Name/Code uniqueness                                      | Object names must be unique in the namespace. <ul style="list-style-type: none"><li>• Manual correction: Modify the duplicate name or code.</li><li>• Automatic correction: Appends a number to the duplicate name or code.</li></ul>   |
| Inconsistency between default values and check parameters | The values entered in the check parameters tab are inconsistent for numeric and string data types: default does not respect minimum and maximum values, or default does not belong to list of values, or values in list are not included in minimum and maximum values, or minimum is greater than maximum value. Check parameters must be defined consistently. <ul style="list-style-type: none"><li>• Manual correction: Modify default, minimum, maximum or list of values in the check parameters tab</li><li>• Automatic correction: None</li></ul> |

## 1.9.2 Data Source Checks

PowerDesigner provides default model checks to verify the validity of data sources.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: None.</li></ul>                                      |
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: Replaces synonyms with their associated glossary terms.</li></ul> |

| Check   | Description and Correction   |
|---|--|
| Name/Code uniqueness                                    | <p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>  |
| Existence of model                                      | <p>A data source must have at least one physical data model in its definition.</p> <ul style="list-style-type: none"> <li>Manual correction: Add a physical data model from the Models tab of the property sheet of the data source</li> <li>Automatic correction: Deletes data source without physical data model</li> </ul>    |
| Data source containain models with different DBMS types | <p>The models in a data source represent a single database. This is why the models in the data source should share the same DBMS.</p> <ul style="list-style-type: none"> <li>Manual correction: Delete models with different DBMS or modify the DBMS of models in the data source</li> <li>Automatic correction: None</li> </ul> |

### 1.9.3 Package Checks

PowerDesigner provides default model checks to verify the validity of packages.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | <p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul>                                      |
| Name/Code contains synonyms of glossary terms | <p>[if glossary enabled] Names and codes must not contain synonyms of glossary terms.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul> |
| Circular inheritance                          | <p>Objects cannot be dependent on each other. Circular links must be removed.</p> <ul style="list-style-type: none"> <li>Manual correction: Remove circular generalization links</li> <li>Automatic correction: None</li> </ul>   |
| Circular dependency                           | <p>Classes are dependent on each other through association class and/or generalization links. Circular links must be removed.</p> <ul style="list-style-type: none"> <li>Manual correction: Remove circular links</li> <li>Automatic correction: None</li> </ul>  |

| Check  | Description and Correction  |
|--|---|
| Shortcut code uniqueness                                     | <p>Two shortcuts with the same code cannot be in the same namespace.</p> <ul style="list-style-type: none"> <li>Manual correction: Change the code of one of the shortcuts</li> <li>Automatic correction: None</li> </ul>   |
| Shortcut potentially generated as child table of a reference | <p>The package should not contain associations with an external shortcut as child class. Although this can be tolerated in the OOM, the association will not be generated in a PDM if the external shortcut is generated as a shortcut.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the design of your model in order to create the association in the package where the child class is defined</li> <li>Automatic correction: None</li> </ul> |

## 1.9.4 Actor/Use Case Checks

PowerDesigner provides default model checks to verify the validity of actors and use cases.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | <p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul>                                      |
| Name/Code contains synonyms of glossary terms | <p>[if glossary enabled] Names and codes must not contain synonyms of glossary terms.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul> |
| Name/Code uniqueness                          | <p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>   |
| Actor/use case not linked to any object       | <p>Actors and use cases should be linked to at least one object in the model.</p> <ul style="list-style-type: none"> <li>Manual correction: Create a link between the actor and a use case, or an object</li> <li>Automatic correction: None</li> </ul>   |

## 1.9.5 Class Checks

PowerDesigner provides default model checks to verify the validity of classes.

| Check  | Description and Correction  |
|--|---|
| Name/Code contains terms not in glossary       | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: None.</li></ul>                                      |
| Name/Code contains synonyms of glossary terms  | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: Replaces synonyms with their associated glossary terms.</li></ul> |
| Name/Code uniqueness                           | Object names must be unique in the namespace. <ul style="list-style-type: none"><li>• Manual correction: Modify the duplicate name or code.</li><li>• Automatic correction: Appends a number to the duplicate name or code.</li></ul>   |
| Empty classifier                               | Attributes and operations are missing for this classifier. <ul style="list-style-type: none"><li>• Manual correction: Add attributes or operations to the classifier</li><li>• Automatic correction: None</li></ul>   |
| Persistent class without persistent attributes | All attributes of a persistent class cannot be non-persistent. <ul style="list-style-type: none"><li>• Manual correction: Define at least one attribute as persistent</li><li>• Automatic correction: None</li></ul>  |
| Association class with identifier(s)           | An associated class should not have identifiers. <ul style="list-style-type: none"><li>• Manual correction: Remove the identifier</li><li>• Automatic correction: None</li></ul>  |
| Classifier visibility                          | A private or protected classifier should be inner to another classifier. <ul style="list-style-type: none"><li>• Manual correction: Change classifier visibility to public or package</li><li>• Automatic correction: Changes the visibility to public or package</li></ul>                             |
| Class constructor return type                  | A constructor cannot have a return type. <ul style="list-style-type: none"><li>• Manual correction: Remove current return type of the constructor</li><li>• Automatic correction: Removes current return type of the constructor</li></ul>  |
| Class constructor modifiers                    | A constructor cannot be static, abstract, or final. <ul style="list-style-type: none"><li>• Manual correction: Remove the static, abstract, or final property of the constructor</li><li>• Automatic correction: Removes the static, abstract or final property of the constructor</li></ul>            |

| Check                                      | Description and Correction   |
|--|--|
| Operation implementation                   | <p>When there is a realization between a class and an interface, you must implement the operations of the interface within the class. To do so, click the <i>Operations</i> tab in the class property sheet and click the <i>To be Implemented</i> button at the bottom of the tab to implement the missing operations.</p> <ul style="list-style-type: none"> <li>Manual correction: Implement the operations of the interface within the class</li> <li>Automatic correction: None</li> </ul>  |
| Role name assignment                       | <p>A navigable role will be migrated as an attribute into a class. The code of the association is used if the role has no name.</p> <ul style="list-style-type: none"> <li>Manual correction: Assign a role name for the association role</li> <li>Automatic correction: None</li> </ul>   |
| Role name uniqueness                       | <p>The name of the role is used by another role or by another attribute.</p> <ul style="list-style-type: none"> <li>Manual correction: Change the name of the duplicate role</li> <li>Automatic correction: None</li> </ul>  |
| JavaBean without a BeanInfo                | <p>Bean implementors that provide explicit information about their beans must provide a BeanInfo class.</p> <ul style="list-style-type: none"> <li>Manual correction: Create a BeanInfo class</li> <li>Automatic correction: None</li> </ul>   |
| BeanInfo without a JavaBean class          | <p>A BeanInfo class must depend on a JavaBean class.</p> <ul style="list-style-type: none"> <li>Manual correction: Create a JavaBean class and recreate its BeanInfo, or delete the BeanInfo</li> <li>Automatic correction: None</li> </ul>  |
| Enumeration type parent                    | <p>A enum may not have children.</p> <ul style="list-style-type: none"> <li>Manual correction: Remove links to child classes.</li> <li>Automatic correction: None</li> </ul>   |
| Bean class definition                      | <p>The Bean class must be defined as public. It must define a public constructor that takes no arguments and cannot define the finalize() method. It must be abstract for CMP Entity Beans but cannot be abstract or final for BMP Entity, Session and Message-driven Beans.</p> <ul style="list-style-type: none"> <li>Manual correction: Change the class visibility to public, define a public constructor with no arguments, do not define the finalize() method</li> <li>Automatic correction: Changes the class visibility to public, defines a public constructor with no arguments and removes the finalize() method. Corrects to set final = false, and set abstract = false</li> </ul> |
| Bean class Business methods implementation | <p>For each method defined in the component interface(s), there must be a matching method in the Bean class that has the same name, number, return type and types of arguments.</p> <ul style="list-style-type: none"> <li>Manual correction: Add a method with the same name, number, return type and types of arguments in the Bean class</li> <li>Automatic correction: Adds a method with the appropriate values in the Bean class</li> </ul>  |

| Check  | Description and Correction   |
|--|--|
| Bean class Home interface methods implementation | <p>For each create&lt;METHOD&gt; method of the bean Home Interface(s), there must be a matching ejbCreate&lt;METHOD&gt; method in the Bean class with the same method arguments. For each home method of the Home Interface(s), there must be a matching ejbHome&lt;METHOD&gt; method in the Bean class with the same number and types of arguments, and the same return type.</p> <p>The following check applies to Entity Beans only.</p> <p>For each ejbCreate&lt;METHOD&gt; method of the Bean class, there must be a matching ejbPostCreate&lt;METHOD&gt; method in the Bean class with the same number and types of arguments.</p> <ul style="list-style-type: none"> <li>Manual correction: Add a method with the same name and types of arguments in the Bean class</li> <li>Automatic correction: Adds a method with the appropriate values in the Bean class</li> </ul> <p>The following check applies to BMP Entity Beans only.</p> <p>For each find&lt;METHOD&gt; finder method defined in the bean Home Interface(s), there must be a corresponding ejbFind&lt;METHOD&gt; method with the same number, return type, and types of arguments.</p> <ul style="list-style-type: none"> <li>Manual correction: Add a method with the same number, return type and types of arguments in the bean Home Interface(s)</li> <li>Automatic correction: Adds a method with the appropriate values in the bean Home Interface(s)</li> </ul> |
| Bean class ejbCreate methods                     | <p>ejbCreate&lt;METHOD&gt; methods must be defined as public, and cannot be final nor static.</p> <p>The following check applies to Entity Beans only.</p> <p>The return type of an ejbCreate() method must be the primary key type.</p> <ul style="list-style-type: none"> <li>Manual correction: Select the primary key in the <i>Return type</i> list of the operation property sheet</li> <li>Automatic correction: Selects the primary key as return type</li> </ul> <p>The following check applies to Session Beans and Message Driven Beans. and Message Driven Beans.</p> <p>The return type of an ejbCreate() method must be void.</p> <ul style="list-style-type: none"> <li>Manual correction: Select void in the <i>Return type</i> list of the operation property sheet</li> <li>Automatic correction: Changes the return type to void</li> </ul> <p>The following check applies to Message Driven Beans only.</p> <p>The Bean class must define an ejbCreate() method that takes no arguments.</p> <ul style="list-style-type: none"> <li>Manual correction: Add a method with no argument in the Bean class</li> <li>Automatic correction: Adds a method with no argument in the Bean class</li> </ul>  |
| Bean class ejbPostCreate methods                 | <p>The following check applies to Entity Beans only.</p> <p>ejbPostCreate&lt;METHOD&gt; methods must be defined as public, and cannot be final nor static. Their return type must be void.</p> <ul style="list-style-type: none"> <li>Manual correction: Change the method visibility to public, deselect the final and static check boxes and select void in the Return type list of the Operation property sheet</li> <li>Automatic correction: Changes the method visibility to public, changes the final and static check boxes and changes the return type to void</li> </ul>   |

| Check                        | Description and Correction  |
|------------------------------|---|
| Bean class ejbFind methods   | <p>BMP Entity Bean specific.</p> <p>ejbFind&lt;METHOD&gt; methods must be defined as public and cannot be final nor static. Their return type must be the entity bean primary key type or a collection of primary keys.</p> <ul style="list-style-type: none"> <li>Manual correction: Change the method visibility to public and deselect the static check box</li> <li>Automatic correction: Changes the method visibility to public and deselects the static and final check boxes. Forces the return type of ejbFind&lt;METHOD&gt; to the primary key type</li> </ul>  |
| Bean class ejbHome methods   | <p>ejbHome&lt;METHOD&gt; methods must be defined as public and cannot be static.</p> <ul style="list-style-type: none"> <li>Manual correction: Change the method visibility to public and deselect the static check box</li> <li>Automatic correction: Changes the method visibility to public and deselects the static check box</li> </ul>  |
| Bean class ejbSelect methods | <p>The following check applies to CMP Entity Beans only.</p> <p>EjbSelect &lt;METHOD&gt; methods must be defined as public and abstract. Their throws clause must include the javax.ejb.FinderException.</p> <ul style="list-style-type: none"> <li>Manual correction: Change the method visibility to public, select the abstract check box, and include the javax.ejb.FinderException</li> <li>Automatic correction: Changes the method visibility to public, selects the abstract check box, and includes the javax.ejb.FinderException</li> </ul>   |
| Primary key class definition | <p>The following check applies to Entity Beans only.</p> <p>The primary key class must be declared as public and must define a public constructor that takes no arguments.</p> <ul style="list-style-type: none"> <li>Manual correction: Change the method visibility to public and add a default constructor in the primary key class</li> <li>Automatic correction: Changes the method visibility to public and adds a default constructor in the primary key class</li> </ul>  |
| Primary key class attributes | <p>All primary key class attributes must be declared as public. In addition, each primary key class attribute must have a corresponding cmp-field in the Bean class.</p> <ul style="list-style-type: none"> <li>Manual correction: Change the visibility to public, and create a cmp-field in the Bean class that has the same name and the same data type as the attribute of the primary key class</li> <li>Automatic correction: Changes the visibility to public and creates a cmp-field in the Bean class that has the same name and the same data type as the attribute of the primary key class</li> </ul> |
| Primary key class existence  | <p>If the bean class has more than one primary key attribute then a primary key class must exist. If there is only one primary key attribute, it cannot have a standard data type, but must have an object data type (ex: java.lang.Long).</p> <ul style="list-style-type: none"> <li>Manual correction: If there are many primary key attributes, create a primary key class. If there is only one primary key attribute, select an object/classifier data type</li> <li>Automatic correction: Creates a primary key class, or selects the appropriate object/classifier data type</li> </ul>                    |

| Check                         | Description and Correction   |
|-------------------------------|--|
| Class mapping not defined     | <p>The class must be mapped to tables or views in the data source.</p> <ul style="list-style-type: none"> <li>Manual correction: Define the mapping from the <a href="#">Mapping</a> tab of the class property sheet (<a href="#">Class Sources</a> tab), or remove the data source</li> <li>Automatic correction: Removes the data source from the <a href="#">Mapping For</a> list in the class <a href="#">Mapping</a> tab</li> </ul> <p>For more information about O/R mapping, see <a href="#">Core Features Guide &gt; Linking and Synchronizing Models &gt; Object Mappings &gt; The Mapping Editor &gt; Object to Relational (O/R) Mapping</a>.</p>  |
| Attribute mapping not defined | <p>The attribute must be mapped to columns in the data source.</p> <ul style="list-style-type: none"> <li>Manual correction: Define the mapping from the <a href="#">Mapping</a> tab of the class property sheet (<a href="#">Attributes Mapping</a> tab), or remove the data source</li> <li>Automatic correction: Removes the data source from the <a href="#">Mapping For</a> list in the class <a href="#">Mapping</a> tab</li> </ul> <p>For more information about O/R mapping, see <a href="#">Core Features Guide &gt; Linking and Synchronizing Models &gt; Object Mappings &gt; The Mapping Editor &gt; Object to Relational (O/R) Mapping</a>.</p> |
| Incomplete bound classifier   | <p>A classifier that is of type "Bound" must be bound to a generic classifier.</p> <ul style="list-style-type: none"> <li>Manual correction: Specify a generic classifier in the field to the right of the type list on the <a href="#">General</a> tab of the bound classifier's property sheet. You can also connect it to the generic classifier by way of a dependency with stereotype &lt;&lt;bind&gt;&gt;.</li> <li>Automatic correction: None</li> </ul>  |
| Invalid generation mode       | <p>If a class has its persistence mode set to <a href="#">Migrate Columns</a>, it must have a persistent parent or child to which to migrate the columns</p> <ul style="list-style-type: none"> <li>Manual correction: Link the class to a persistent parent or child, or change its persistence mode on the <a href="#">Detail</a> tab of its property sheet.</li> <li>Automatic correction: None</li> </ul>  |

## 1.9.6 Identifier Checks

PowerDesigner provides default model checks to verify the validity of identifiers.

| Check                                    | Description and Correction   |
|--|--|
| Name/Code contains terms not in glossary | <p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul> |

| Check   | Description and Correction   |
|---|--|
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul> |
| Name/Code uniqueness                          | Object names must be unique in the namespace. <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>   |
| Existence of attribute                        | Identifiers must have at least one attribute. <ul style="list-style-type: none"> <li>Manual correction: Add an attribute to the identifier, or delete the identifier</li> <li>Automatic correction: None</li> </ul>  |
| Identifier inclusion                          | Two identifiers should not use the same attributes. <ul style="list-style-type: none"> <li>Manual correction: Remove the unnecessary identifier</li> <li>Automatic correction: None</li> </ul>   |

## 1.9.7 Interface Checks

PowerDesigner provides default model checks to verify the validity of interfaces.

| Check   | Description and Correction   |
|---|--|
| Name/Code contains terms not in glossary      | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul>                                      |
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul> |
| Name/Code uniqueness                          | Object names must be unique in the namespace. <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>   |
| Empty classifier                              | Attributes and operations are missing for this classifier. <ul style="list-style-type: none"> <li>Manual correction: Add attributes or operations to the classifier</li> <li>Automatic correction: None</li> </ul>   |

| Check                         | Description and Correction   |
|-------------------------------|--|
| Classifier visibility         | <p>A private or protected classifier should be inner to another classifier.</p> <ul style="list-style-type: none"> <li>Manual correction: Change classifier visibility to public or package</li> <li>Automatic correction: Changes the visibility to public or package</li> </ul>  |
| Interface constructor         | <p>An interface cannot be instantiated. Thus a constructor cannot be defined for an interface.</p> <ul style="list-style-type: none"> <li>Manual correction: Remove the constructor</li> <li>Automatic correction: None</li> </ul>   |
| Interface navigability        | <p>Navigation is not allowed from an interface.</p> <ul style="list-style-type: none"> <li>Manual correction: Deselect navigability on the class side of the association</li> <li>Automatic correction: Deselects navigability on the class side of the association</li> </ul>   |
| Home interface create methods | <p>The return type for create&lt;METHOD&gt; methods must be the bean component interface type. The throws clause must include the javax.ejb.CreateException together with all exceptions defined in the throws clause of the matching ejbCreate&lt;METHOD&gt; and ejbPostCreate&lt;METHOD&gt; methods of the Bean class.</p> <ul style="list-style-type: none"> <li>Manual correction: Include the javax.ejb.CreateException and all exceptions defined in the throws clause of the matching ejbCreate&lt;METHOD&gt; and ejbPostCreate&lt;METHOD&gt; methods of the Bean class, or remove exceptions from the ejbPostCreate&lt;METHOD&gt; method</li> <li>Automatic correction: Includes the javax.ejb.CreateException and all exceptions defined in the throws clause of the matching ejbCreate&lt;METHOD&gt; and ejbPostCreate&lt;METHOD&gt; methods of the Bean class</li> </ul>  |
| Home interface finder methods | <p>The return type for find&lt;METHOD&gt; methods must be the bean component interface type (for a single-object finder) or a collection of primary keys thereof (for a multi-object finder). The throws clause must include the javax.ejb.FinderException.</p> <ul style="list-style-type: none"> <li>Manual correction: Include the javax.ejb.FinderException in the throws clause</li> <li>Automatic correction: Includes the javax.ejb.FinderException in the throws clause, and sets Return Type to be the component interface type</li> </ul> <p>The following check applies to BPM Entity Beans only.</p> <p>The throws clause must include all exceptions defined in the throws clause of the matching ejbFind&lt;METHOD&gt; methods of the Bean class.</p> <ul style="list-style-type: none"> <li>Manual correction: Include all exceptions defined in the throws clause of the matching ejbFind&lt;METHOD&gt; methods of the Bean class, or remove exceptions from the ejbFind&lt;METHOD&gt; method</li> <li>Automatic correction: Includes all exceptions defined in the throws clause of the matching ejbFind&lt;METHOD&gt; methods of the Bean class</li> </ul> |
| Remote Home interface methods | <p>The throws clause of the Remote Home interface methods must include the java.rmi.RemoteException.</p> <ul style="list-style-type: none"> <li>Manual correction: Include the java.rmi.RemoteException</li> <li>Automatic correction: Includes the java.rmi.RemoteException</li> </ul>  |

| Check                                | Description and Correction  |
|--------------------------------------|---|
| Component interface business methods | <p>The throws clause of the component interface business methods must include all exceptions defined in the throws clause of the matching method of the Bean class. The throws clause of the Remote interface methods must include the java.rmi.RemoteException.</p> <ul style="list-style-type: none"> <li>Manual correction: Include the java.rmi.RemoteException</li> <li>Automatic correction: Includes the java.rmi.RemoteException</li> </ul>         |
| Incomplete bound classifier          | <p>A classifier that is of type "Bound" must be bound to a generic classifier.</p> <ul style="list-style-type: none"> <li>Manual correction: Specify a generic classifier in the field to the right of the type list on the <i>General</i> tab of the bound classifier's property sheet. You can also connect it to the generic classifier by way of a dependency with stereotype &lt;&gt;bind&gt;&lt;/&gt;.</li> <li>Automatic correction: None</li> </ul> |

## 1.9.8 Class/Interface Attribute Checks

PowerDesigner provides default model checks to verify the validity of class and interface attributes.

| Check  | Description and Correction  |
|--|---|
| Name/Code contains terms not in glossary       | <p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul>  |
| Name/Code contains synonyms of glossary terms  | <p>[if glossary enabled] Names and codes must not contain synonyms of glossary terms.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul>   |
| Name/Code uniqueness                           | <p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>   |
| Detect inconsistencies within check parameters | <p>The values entered in the check parameters tab are inconsistent for numeric and string data types: default does not respect minimum and maximum values, or default does not belong to list of values, or values in list are not included in minimum and maximum values, or minimum is greater than maximum value. Check parameters must be defined consistently.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify default, minimum, maximum or list of values in the check parameters tab</li> <li>Automatic correction: None</li> </ul> |

| Check                             | Description and Correction   |
|-----------------------------------|--|
| Data type assignment              | <p>The data type of an attribute should be defined. Moreover, its type cannot be void.</p> <ul style="list-style-type: none"> <li>Manual correction: Assign a valid data type to the attribute</li> <li>Automatic correction: None</li> </ul>  |
| Initial value for final attribute | <p>The final attribute of a classifier must be initialized.</p> <ul style="list-style-type: none"> <li>Manual correction: Give a default value to the final attribute</li> <li>Automatic correction: None</li> </ul>   |
| Domain divergence                 | <p>The definition of the attribute definition is diverging from the definition of the domain.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify attribute type to respect domain properties</li> <li>Automatic correction: Corrects attribute type to prevent divergence from domain</li> </ul> <p>For more information about domain divergence, see <a href="#">Setting OOM Model Options [page 15]</a>.</p> |
| Event parameter data type         | <p>[VB 2005] An interface attribute with a stereotype of Event must have a delegate as its data type.</p> <ul style="list-style-type: none"> <li>Manual correction: set the data type to an appropriate delegate</li> <li>Automatic correction: None</li> </ul>  |

## 1.9.9 Class/Interface Operation Checks

PowerDesigner provides default model checks to verify the validity of class and interface operations.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | <p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul>                                      |
| Name/Code contains synonyms of glossary terms | <p>[if glossary enabled] Names and codes must not contain synonyms of glossary terms.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul> |
| Name/Code uniqueness                          | <p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>   |
| Return type assignment                        | <p>The return type of an operation should be defined.</p> <ul style="list-style-type: none"> <li>Manual correction: Assign a return type to the operation</li> <li>Automatic correction: Assigns a void return type to the operation</li> </ul>   |

| Check   | Description and Correction   |
|---|--|
| Parameter data type assignment                | <p>The data type of a parameter should be defined. Moreover, its type cannot be void.</p> <ul style="list-style-type: none"> <li>Manual correction: Choose a valid data type for the parameter</li> <li>Automatic correction: None</li> </ul>  |
| Abstract operation body                       | <p>[classes] Abstract operations in a class cannot be implemented.</p> <ul style="list-style-type: none"> <li>Manual correction: Remove the body or the abstract property of the operation</li> <li>Automatic correction: None</li> </ul>  |
| Abstract operation in a instantiable class    | <p>[classes] Abstract operations must be declared in abstract classes only.</p> <ul style="list-style-type: none"> <li>Manual correction: Set the class to abstract, or remove the abstract property of the operation</li> <li>Automatic correction: Removes the abstract property in the operation property sheet</li> </ul>  |
| Overloading operations signature              | <p>[classes] Overloaded operations with the same name and same parameters data type cannot have different return types in a class.</p> <p>Overloading an operation refers to using the same method name but performing different operations based on different parameter number or type.</p> <ul style="list-style-type: none"> <li>Manual correction: Change the operation name, parameter data type, or return type</li> <li>Automatic correction: None</li> </ul> |
| Overriding operations                         | <p>[classes] When overriding a parent operation in a class, it is impossible to change its modifiers.</p> <p>Overriding an operation means that an operation defined in a given class is redefined in a child class, in this case the operation of the parent class is said to be overriden.</p> <ul style="list-style-type: none"> <li>Manual correction: Keep the same modifiers for child operation</li> <li>Automatic correction: None</li> </ul>                |
| Enum: Constants must overload abstract method | <p>[classes] You can give each enum constant a different behavior by declaring an abstract method in the enum type and overloading it with a concrete method for each constant. In this case, each constant must overload the abstract method.</p> <ul style="list-style-type: none"> <li>Manual correction: Make sure each constant is associated with a concrete method that overloads the abstract method.</li> <li>Automatic correction: None</li> </ul>         |

## 1.9.10 Realization Checks

PowerDesigner provides default model checks to verify the validity of realizations.

| Check   | Description and Correction   |
|---|--|
| Redundant realizations                            | Only one realization is allowed between two given objects. <ul style="list-style-type: none"><li>• Manual correction: Remove redundant realizations</li><li>• Automatic correction: None</li></ul>   |
| Realization generic missing child type parameters | A child of a generic classifier must resolve all of the type parameters defined by its parent. <ul style="list-style-type: none"><li>• Manual correction: Resolve the missing type parameters.</li><li>• Automatic correction: None.</li></ul> |
| Realization generic child cannot be bound         | A bound classifier cannot be the child of any classifier other than its generic parent. <ul style="list-style-type: none"><li>• Manual correction: Remove the additional links.</li><li>• Automatic correction: None.</li></ul>                |

## 1.9.11 Generalization Checks

PowerDesigner provides default model checks to verify the validity of generalizations.

| Check                      | Description and Correction  |
|----------------------------|---|
| Redundant generalizations  | Only one generalization is allowed between two classes or two interfaces. <ul style="list-style-type: none"><li>• Manual correction: Remove redundant generalizations</li><li>• Automatic correction: None</li></ul>  |
| Class multiple inheritance | The following check applies only to Java and PowerBuilder.<br>Multiple inheritance is accepted in UML but not in this language. <ul style="list-style-type: none"><li>• Manual correction: Keep single inheritance</li><li>• Automatic correction: None</li></ul> |
| Extend final class         | A final class cannot be extended. <ul style="list-style-type: none"><li>• Manual correction: Remove the generalization link, or remove the final property in the parent class</li><li>• Automatic correction: None</li></ul>                                      |

| Check                               | Description and Correction  |
|-------------------------------------|---|
| Non-persistent specifying attribute | If a generalization has a specifying attribute, the attribute must be marked as persistent. <ul style="list-style-type: none"> <li>Manual correction: Select the Persistent checkbox on the Detail tab of the specifying attribute property sheet.</li> <li>Automatic correction: None</li> </ul> |
| Generic: Child type parameters      | A child of a generic classifier must resolve all of the type parameters defined by its parent. <ul style="list-style-type: none"> <li>Manual correction: Resolve the missing type parameters.</li> <li>Automatic correction: None.</li> </ul>   |
| Generic: Child cannot be bound      | A bound classifier cannot be the child of any classifier other than its generic parent. <ul style="list-style-type: none"> <li>Manual correction: Remove the additional links.</li> <li>Automatic correction: None.</li> </ul>  |

## 1.9.12 Object Checks

PowerDesigner provides default model checks to verify the validity of objects.

| Check   | Description and Correction   |
|---|--|
| Name/Code contains terms not in glossary      | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul>  |
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul>   |
| Name/Code uniqueness                          | Object names must be unique in the namespace. <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>   |
| Isolated object                               | An object should not be isolated in the model. <ul style="list-style-type: none"> <li>Manual correction: Create a relationship to or from the object. The relationship can be a message, an instance link, or a dependency or Link the object to an object node in the activity diagram</li> <li>Automatic correction: None</li> </ul> <p>Note: the Check Model feature takes the object into account and not the symbol of the object to perform this check; if the object is already associated with an instance link or an object node in your model, the Check Model feature will not return an error message.</p> |

## 1.9.13 Instance Link Checks

PowerDesigner provides default model checks to verify the validity of instance links.

| Check                    | Description and Correction  |
|--------------------------|---|
| Redundant instance links | <p>Two instance links between the same objects should not have the same association.</p> <ul style="list-style-type: none"><li>Manual correction: Remove one of the redundant instance links</li><li>Automatic correction: None</li></ul> |

## 1.9.14 Message Checks

PowerDesigner provides default model checks to verify the validity of messages.

| Check                                  | Description and Correction  |
|--|---|
| Message without sequence number        | <p>A message should have a sequence number.</p> <ul style="list-style-type: none"><li>Manual correction: Enter a sequence number on the message</li><li>Automatic correction: None</li></ul>  |
| Message used by several instance links | <p>A message should not be attached to several instance links.</p> <ul style="list-style-type: none"><li>Manual correction: Detach the message from the instance link</li><li>Automatic correction: None</li></ul>  |
| Message between actors                 | <p>An actor cannot send a message to another actor in the model. Messages are allowed between two objects, and between objects and actors.</p> <ul style="list-style-type: none"><li>Manual correction: Create a message between two objects or between an actor and an object</li><li>Automatic correction: None</li></ul> |

## 1.9.15 State Checks

PowerDesigner provides default model checks to verify the validity of states.

| Check                                    | Description and Correction  |
|--|---|
| Name/Code contains terms not in glossary | <p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"><li>Manual correction: Modify the name or code to contain only glossary terms.</li><li>Automatic correction: None.</li></ul> |

| Check   | Description and Correction  |
|---|---|
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul>  |
| Name/Code uniqueness                          | Object names must be unique in the namespace. <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>  |
| Input transition missing                      | Each state must have at least one input transition. A state without an input transition cannot be reached. <ul style="list-style-type: none"> <li>Manual correction: Add a transition linked to the state</li> <li>Automatic correction: None</li> </ul>  |
| Composite state does not have any start       | A composite state details the state behavior in a sub-statechart diagram. To be complete, this sub-statechart diagram requires a start. <ul style="list-style-type: none"> <li>Manual correction: Add a start in the sub-statechart diagram, or deselect the Composite check box in the state property sheet</li> <li>Automatic correction: None</li> </ul>   |
| Incorrect action order                        | The entry trigger events must be the first in the list of actions on a state. The exit trigger events must be the last in the list. All other actions can be ordered without any constraint. <ul style="list-style-type: none"> <li>Manual correction: Move all entry at the top of the list and all exit at the bottom</li> <li>Automatic correction: Moves all entry at the top of the list and all exit at the bottom</li> </ul> |

## 1.9.16 State Action Checks

PowerDesigner provides default model checks to verify the validity of state actions.

| Check   | Description and Correction   |
|---|--|
| Name/Code contains terms not in glossary      | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul>                                      |
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul> |

| Check                     | Description and Correction  |
|---------------------------|---|
| Name/Code uniqueness      | <p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>   |
| Unspecified trigger event | <p>Each action on a state must have a trigger event specified. This trigger event indicates when the action is executed.</p> <p>Note that this check does not apply to actions defined on transitions because transitions have an implicit event corresponding to the end of execution of internal actions (of the source state).</p> <ul style="list-style-type: none"> <li>Manual correction: Specify a trigger event in the action property sheet</li> <li>Automatic correction: None</li> </ul> |
| Duplicated occurrence     | <p>Two distinct actions of a same state should not occur simultaneously. The occurrence of an action is defined by combining a trigger event and a condition.</p> <ul style="list-style-type: none"> <li>Manual correction: Change the trigger event or the condition of the action</li> <li>Automatic correction: None</li> </ul>  |

### 1.9.17 Event Checks

PowerDesigner provides default model checks to verify the validity of events.

| Check   | Description and Correction   |
|---|--|
| Name/Code contains terms not in glossary      | <p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul>   |
| Name/Code contains synonyms of glossary terms | <p>[if glossary enabled] Names and codes must not contain synonyms of glossary terms.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul>    |
| Name/Code uniqueness                          | <p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>  |
| Unused event                                  | <p>An event is useful to trigger an action defined on a state or on a transition. An event alone is useless.</p> <ul style="list-style-type: none"> <li>Manual correction: Delete the event or use it within an action on a state or on a transition</li> <li>Automatic correction: Deletes the event</li> </ul> |

## 1.9.18 Junction Point Checks

PowerDesigner provides default model checks to verify the validity of junction points.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: None.</li></ul>  |
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: Replaces synonyms with their associated glossary terms.</li></ul>               |
| Name/Code uniqueness                          | Object names must be unique in the namespace. <ul style="list-style-type: none"><li>• Manual correction: Modify the duplicate name or code.</li><li>• Automatic correction: Appends a number to the duplicate name or code.</li></ul>   |
| Incomplete junction point                     | A junction point represents a split or a merge of transition paths. That is why a junction point must have at least one input and one output transitions. <ul style="list-style-type: none"><li>• Manual correction: Add any missing transitions on the junction point</li><li>• Automatic correction: None</li></ul> |

## 1.9.19 Activity Checks

PowerDesigner provides default model checks to verify the validity of activities.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: None.</li></ul>                                      |
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: Replaces synonyms with their associated glossary terms.</li></ul> |
| Name/Code uniqueness                          | Object names must be unique in the namespace. <ul style="list-style-type: none"><li>• Manual correction: Modify the duplicate name or code.</li><li>• Automatic correction: Appends a number to the duplicate name or code.</li></ul>   |

| Check                                      | Description and Correction   |
|--|--|
| Input or Output transition missing         | <p>Each activity must have at least one input transition and at least one output transition.</p> <ul style="list-style-type: none"> <li>Manual correction: Add a transition linked to the activity</li> <li>Automatic correction: None</li> </ul>  |
| Composite activity does not have any start | <p>A composite activity details the activity execution in a sub-activity diagram. To be complete, this sub-activity diagram requires a start connected to other activities, or requires a start at the beginning.</p> <ul style="list-style-type: none"> <li>Manual correction: Add a start in the sub-activity diagram, or deselect the Composite check box in the activity property sheet</li> <li>Automatic correction: None</li> </ul> |
| Non-Reusable Activity Calls                | <p>Only activities with an action type of &lt;undefined&gt; or Reusable activity may be reused by other activities with action types of Call, Accept Call, or Reply Call.</p> <ul style="list-style-type: none"> <li>Manual correction: Change the action type of the referenced activity, or remove any references to it.</li> <li>Automatic correction: None</li> </ul>  |

## 1.9.20 Decision Checks

PowerDesigner provides default model checks to verify the validity of decisions.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | <p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul>                                      |
| Name/Code contains synonyms of glossary terms | <p>[if glossary enabled] Names and codes must not contain synonyms of glossary terms.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul> |
| Name/Code uniqueness                          | <p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>   |

| Check               | Description and Correction  |
|---------------------|---|
| Incomplete decision | <p>A decision represents a conditional branch when a unique transition is split into several output transitions, or it represents a merge when several input transitions are merged into a unique output transition. That is why a decision must have more than one input transition or more than one output transition.</p> <ul style="list-style-type: none"> <li>Manual correction: Add any missing transitions on the decision</li> <li>Automatic correction: None</li> </ul> |

## 1.9.21 Object Node Checks

PowerDesigner provides default model checks to verify the validity of object nodes.

| Check   | Description and Correction   |
|---|--|
| Name/Code contains terms not in glossary      | <p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul>   |
| Name/Code contains synonyms of glossary terms | <p>[if glossary enabled] Names and codes must not contain synonyms of glossary terms.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul>                |
| Name/Code uniqueness                          | <p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>  |
| Object node with undefined object             | <p>An object node represents a particular state of an object. That is why an object must be linked to an object node.</p> <ul style="list-style-type: none"> <li>Manual correction: In the property sheet of the object node, select or create an object from the Object list</li> <li>Automatic correction: None</li> </ul> |
| Object Node Without Data Type                 | <p>An object node conveys no information if it does not have a data type.</p> <ul style="list-style-type: none"> <li>Manual correction: Select a Data type in the object node property sheet.</li> <li>Automatic correction: None</li> </ul>   |

## 1.9.22 Organization Unit Checks

PowerDesigner provides default model checks to verify the validity of organization units.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: None.</li></ul>                                      |
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: Replaces synonyms with their associated glossary terms.</li></ul> |
| Name/Code uniqueness                          | Object names must be unique in the namespace. <ul style="list-style-type: none"><li>• Manual correction: Modify the duplicate name or code.</li><li>• Automatic correction: Appends a number to the duplicate name or code.</li></ul>   |
| Circular dependency                           | The organization unit cannot be parent of itself or parent of another child organization unit. <ul style="list-style-type: none"><li>• Manual correction: Change the organization unit in the Parent box in the organization unit property sheet</li><li>• Automatic correction: None</li></ul>         |

## 1.9.23 Start/End Checks

PowerDesigner provides default model checks to verify the validity of starts and ends.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: None.</li></ul>                                      |
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: Replaces synonyms with their associated glossary terms.</li></ul> |
| Name/Code uniqueness                          | Object names must be unique in the namespace. <ul style="list-style-type: none"><li>• Manual correction: Modify the duplicate name or code.</li><li>• Automatic correction: Appends a number to the duplicate name or code.</li></ul>   |

| Check              | Description and Correction  |
|--------------------|---|
| Missing transition | <p>Starts and ends must be linked to an object of the statechart or activity diagram.</p> <ul style="list-style-type: none"> <li>Manual correction: Create a transition from the start and/or to the end</li> <li>Automatic correction: None</li> </ul> |

## 1.9.24 Synchronization Checks

PowerDesigner provides default model checks to verify the validity of synchronizations.

| Check   | Description and Correction   |
|---|--|
| Name/Code contains terms not in glossary      | <p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul>   |
| Name/Code contains synonyms of glossary terms | <p>[if glossary enabled] Names and codes must not contain synonyms of glossary terms.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul>  |
| Name/Code uniqueness                          | <p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>  |
| Incomplete synchronization                    | <p>A synchronization represents a fork when a unique transition is split into several output transitions executed in parallel, or it represents a join when several input transitions are joined and they wait until all transitions reach the join before continuing as a unique output transition. That is why a synchronization must have more than one input transition, or more than one output transition.</p> <ul style="list-style-type: none"> <li>Manual correction: Add any missing transitions to the synchronization</li> <li>Automatic correction: None</li> </ul> |

## 1.9.25 Transition and Flow Checks

PowerDesigner provides default model checks to verify the validity of transitions and flows.

| Check  | Description and Correction   |
|--|--|
| Transition / flow without source or destination                          | The transition or flow has no source or destination. <ul style="list-style-type: none"><li>• Manual correction: Select or create an object as source or destination.</li><li>• Automatic correction: None</li></ul>  |
| Useless condition  | If there is only one output transition/flow, there is no reason to have a condition or type on the transition/flow. <ul style="list-style-type: none"><li>• Manual correction: Remove the unnecessary condition or type, or create another transition/flow with another condition or type.</li><li>• Automatic correction: None</li></ul>  |
| Missing condition  | If an object has several output transitions/flows, or if the transition/flow is reflexive, each transition/flow must contain a condition.<br><br>In a statechart diagram, a transition must contain an event or condition. <ul style="list-style-type: none"><li>• Manual correction: Define a condition, or create a synchronization to specify a parallel execution</li><li>• Automatic correction: None</li></ul> |
| Duplicated transition between states/ Duplicated flow between activities | Two parallel transitions (with the same extremities) must not occur simultaneously, but must rather be governed by conditions (and, for transitions, trigger events). <ul style="list-style-type: none"><li>• Manual correction: Change one of the trigger events or conditions</li><li>• Automatic correction: None</li></ul>   |

## 1.9.26 Component Checks

PowerDesigner provides default model checks to verify the validity of components.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: None.</li></ul>                                      |
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: Replaces synonyms with their associated glossary terms.</li></ul> |

| Check                              | Description and Correction   |
|------------------------------------|--|
| Name/Code uniqueness               | <p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>  |
| Isolated component                 | <p>A component should not be isolated in the model. It should be linked to a class or an interface.</p> <ul style="list-style-type: none"> <li>Manual correction: Attach some classes or interfaces to the component</li> <li>Automatic correction: None</li> </ul>  |
| EJB component attached classifiers | <p>Entity and Session Beans must provide either a remote or a local client view, or both.</p> <ul style="list-style-type: none"> <li>Manual correction: Complete existing view(s), or create a remote view if no interface has been exposed</li> <li>Automatic correction: Completes existing view(s), or creates a remote view if no interface has been exposed</li> </ul>  |
| SOAP message redefinition          | <p>You cannot have the same SOAP input and SOAP output data type inside the same component.</p> <ul style="list-style-type: none"> <li>Manual correction: Change the name of the input data type, or change the name of the output data type in the SOAP Input or SOAP Output tabs in the operation property sheet</li> <li>Automatic correction: None</li> </ul> <p>The definition of the SOAP data types is available in the schema part of the WSDL that you can display in the WSDL tab of the component property sheet.</p> |

## 1.9.27 Node Checks

PowerDesigner provides default model checks to verify the validity of nodes.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | <p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul>                                      |
| Name/Code contains synonyms of glossary terms | <p>[if glossary enabled] Names and codes must not contain synonyms of glossary terms.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul> |
| Name/Code uniqueness                          | <p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>   |

| Check      | Description and Correction   |
|------------|--|
| Empty node | A node is said to be empty when it does not contain any component instance. <ul style="list-style-type: none"> <li>• Manual correction: Add at least one component instance to the node</li> <li>• Automatic correction: None</li> </ul> |

## 1.9.28 Data Format Checks

PowerDesigner provides default model checks to verify the validity of data formats.

| Check            | Description and Correction   |
|------------------|--|
| Empty expression | Data formats must have a value entered in the <i>Expression</i> field. <ul style="list-style-type: none"> <li>• Manual correction: Specify an expression for the data format.</li> <li>• Automatic correction: None</li> </ul> |

## 1.9.29 Component Instance Checks

PowerDesigner provides default model checks to verify the validity of component instances.

| Check   | Description and Correction   |
|---|--|
| Name/Code contains terms not in glossary      | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"> <li>• Manual correction: Modify the name or code to contain only glossary terms.</li> <li>• Automatic correction: None.</li> </ul>  |
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"> <li>• Manual correction: Modify the name or code to contain only glossary terms.</li> <li>• Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul>   |
| Name/Code uniqueness                          | Object names must be unique in the namespace. <ul style="list-style-type: none"> <li>• Manual correction: Modify the duplicate name or code.</li> <li>• Automatic correction: Appends a number to the duplicate name or code.</li> </ul>   |
| Component instance without component          | An instance of a component has been created but no component is attached. You must attach it to a component. <ul style="list-style-type: none"> <li>• Manual correction: Attach an existing component to the component instance, or create a component from the Create tool in the property sheet of the component instance</li> <li>• Automatic correction: None</li> </ul> |

| Check                         | Description and Correction  |
|-------------------------------|---|
| Duplicate component instances | <p>Several instances of the same component exist in the same node.</p> <ul style="list-style-type: none"> <li>Manual correction: Delete the duplicate component instance or attach it to the right component</li> <li>Automatic correction: None</li> </ul> |
| Isolated component instance   | <p>A component instance should not be created outside of a node. It will not be deployed.</p> <ul style="list-style-type: none"> <li>Manual correction: Attach it to a node</li> <li>Automatic correction: None</li> </ul>                                  |

### 1.9.30 Interaction Reference Checks

PowerDesigner provides default model checks to verify the validity of interaction references.

| Check                                 | Description and Correction  |
|---------------------------------------|---|
| Missing referenced diagram            | <p>An interaction reference object must reference a sequence diagram object.</p> <ul style="list-style-type: none"> <li>Manual correction: Open the interaction reference property sheet and specify the sequence diagram to reference.</li> <li>Automatic correction: None</li> </ul>  |
| Attached lifelines consistency        | <p>The interaction reference symbol has a list of attached lifelines, which correspond to actors and objects. These actors and objects must match a subset of the ones displayed in the referenced sequence diagram. This corresponds to a subset because some lifelines in referenced diagram could not be displayed in the diagram of the interaction reference. The current check verifies the following consistency rules:</p> <p>The number of attached lifelines cannot be greater than the number of lifelines in the referenced diagram</p> <p>If one attached lifeline corresponds to an object, and if this object has an associated metaclass, then there must be at least one object in the referenced sequence diagram that is associated with the same metaclass</p> <ul style="list-style-type: none"> <li>Manual correction: Change the list of attached lifelines for the interaction reference object. This can be done simply by resizing the interaction reference symbol or by clicking with the pointer tool on the intersection of the interaction reference symbol and the lifeline. The tool cursor changes on this area and allows you to detach the interaction reference symbol from (or attach it to) the lifeline.</li> <li>Automatic correction: None</li> </ul> |
| Too many input messages for reference | <p>The interaction reference has more incoming than outgoing messages.</p> <ul style="list-style-type: none"> <li>Manual correction: Delete incoming messages or add outgoing messages, until the numbers are equal.</li> <li>Automatic correction: None</li> </ul>   |

| Check                                  | Description and Correction  |
|--|---|
| Too many output messages for reference | <p>The interaction reference has more outgoing than incoming messages.</p> <ul style="list-style-type: none"> <li>Manual correction: Delete outgoing messages or add incoming messages, until the numbers are equal.</li> <li>Automatic correction: None</li> </ul> |

### 1.9.31 Class Part Checks

PowerDesigner provides default model checks to verify the validity of class parts.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | <p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: None.</li> </ul>  |
| Name/Code contains synonyms of glossary terms | <p>[if glossary enabled] Names and codes must not contain synonyms of glossary terms.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the name or code to contain only glossary terms.</li> <li>Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul>           |
| Name/Code uniqueness                          | <p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> <li>Manual correction: Modify the duplicate name or code.</li> <li>Automatic correction: Appends a number to the duplicate name or code.</li> </ul>   |
| Class part classifier type                    | <p>A class part must have a data type that is a classifier linked to its owner classifier by an association.</p> <ul style="list-style-type: none"> <li>Manual correction: Specify a data type for the part and connect the relevant classifier to its owner classifier.</li> <li>Automatic correction: None</li> </ul> |
| Class part association type                   | <p>The composition property of a part must match the type of the association between its owner and its data type.</p> <ul style="list-style-type: none"> <li>Manual correction: Enable or disable the Composition property.</li> <li>Automatic correction: The Composition property is enabled or disabled.</li> </ul>  |

## 1.9.32 Class/Component Port Checks

PowerDesigner provides default model checks to verify the validity of class and component ports.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: None.</li></ul>                                      |
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: Replaces synonyms with their associated glossary terms.</li></ul> |
| Name/Code uniqueness                          | Object names must be unique in the namespace. <ul style="list-style-type: none"><li>• Manual correction: Modify the duplicate name or code.</li><li>• Automatic correction: Appends a number to the duplicate name or code.</li></ul>   |
| Class or component port isolated ports        | Class and component ports must have a data type, or must have a provided or required interface. <ul style="list-style-type: none"><li>• Manual correction: Specify a data type or interface</li><li>• Automatic correction: None</li></ul>  |

## 1.9.33 Class/component Assembly Connector Checks

PowerDesigner provides default model checks to verify the validity of class and component assembly connectors.

| Check   | Description and Correction  |
|---|---|
| Name/Code contains terms not in glossary      | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: None.</li></ul>                                      |
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"><li>• Manual correction: Modify the name or code to contain only glossary terms.</li><li>• Automatic correction: Replaces synonyms with their associated glossary terms.</li></ul> |
| Name/Code uniqueness                          | Object names must be unique in the namespace. <ul style="list-style-type: none"><li>• Manual correction: Modify the duplicate name or code.</li><li>• Automatic correction: Appends a number to the duplicate name or code.</li></ul>   |

| Check   | Description and Correction  |
|---|---|
| Component assembly connector null interface connector | An interface must be defined for each assembly connector. <ul style="list-style-type: none"> <li>• Manual correction: Define an interface for the assembly connector.</li> <li>• Automatic correction: None.</li> </ul>   |
| Component assembly connector interfaces               | The interface defined for an assembly connector must be provided by the supplier and required by the client. <ul style="list-style-type: none"> <li>• Manual correction: Ensure that the supplier and client are correctly defined.</li> <li>• Automatic correction: None.</li> </ul> |

### 1.9.34 Association Checks

PowerDesigner provides default model checks to verify the validity of associations.

| Check                          | Description and Correction  |
|--------------------------------|---|
| Generic: Child type parameters | In a navigable association, if the parent is generic, the child must redefine all the type parameters of the parent.<br><br>If the parent is a partially bound classifier (where some type parameters are not resolved) then the child must redefine all the unresolved type parameters. <ul style="list-style-type: none"> <li>• Manual correction: Resolve the missing type parameters.</li> <li>• Automatic correction: None.</li> </ul> |
| Generic: Child cannot be bound | A bound classifier cannot be the child of any navigable association other than its generic parent. <ul style="list-style-type: none"> <li>• Manual correction: Remove the additional links.</li> <li>• Automatic correction: None.</li> </ul>   |

### 1.9.35 Activity Input and Output Parameter Checks

PowerDesigner provides default model checks to verify the validity of activity input and output parameters.

| Check                                    | Description and Correction  |
|--|---|
| Name/Code contains terms not in glossary | [if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none"> <li>• Manual correction: Modify the name or code to contain only glossary terms.</li> <li>• Automatic correction: None.</li> </ul> |

| Check   | Description and Correction   |
|---|--|
| Name/Code contains synonyms of glossary terms | [if glossary enabled] Names and codes must not contain synonyms of glossary terms. <ul style="list-style-type: none"> <li>• Manual correction: Modify the name or code to contain only glossary terms.</li> <li>• Automatic correction: Replaces synonyms with their associated glossary terms.</li> </ul> |
| Name/Code uniqueness                          | Object names must be unique in the namespace. <ul style="list-style-type: none"> <li>• Manual correction: Modify the duplicate name or code.</li> <li>• Automatic correction: Appends a number to the duplicate name or code.</li> </ul>   |

## 1.10 Importing a Rational Rose Model into an OOM

You can import Rational Rose (.MDL) models created with version 98, 2000, and 2002 into an OOM.

### Context

#### i Note

A Rose model can support one or several languages whereas a PowerDesigner OOM can only have a single object language. When you import a multi-language Rose model into an OOM, the OOM will have only one of the object languages of the Rose model. The following table shows how Rose languages are converted to PowerDesigner languages:

| Rose Language  | PowerDesigner Language |
|--|------------------------|
| CORBA  | IDL-CORBA              |
| Java   | Java                   |
| C++  | C++                    |
| VC++   | C++                    |
| XML-DTD  | XML-DTD                |
| Visual Basic   | Visual Basic 6         |
| Analysis, Oracle 8, Ada, COM, Web Modeler, and all other languages | Analysis               |

## Procedure

1. Select *File* > *Import* > *Rational Rose File* , and browse to the directory that contains the Rose file.
2. Select Rational Rose Model (\*.MDL) file from the Files of Type list, and then select the file to import.
3. Click *Open*.

The import process begins, and the default diagram of the model opens in the canvas. General Rose objects are imported as follows:

| Rose Object | OOM Object  |
|-------------|---|
| Package     | Package   |
|             | <b>Note</b><br>The Global property is not imported. |
| Diagram     | Diagram   |
| Note        | Note  |
| Note link   | Note link   |
| Text        | Text  |
| File        | File  |

Only the following properties are imported for the Rose model object:

| Rose Property | OOM Property |
|---------------|--------------|
| Documentation | Comment      |
| Zoom          | Page scale   |
| Stereotype    | Stereotype   |

## 1.10.1 Importing Rational Rose Use Case Diagrams

PowerDesigner can import the most important objects in Rose use case diagrams.

Only the listed properties are imported:

| Rose Object  | OOM Object  |
|--|---|
| Class with <<actor>>, <<business actor>>, or <<business worker>> stereotype                    | Implementation class contained by an actor                                    |
| Class with <<boundary>>, <<business entity>>, <<control>>, <<entity>>, or <<table>> stereotype | Class (without symbol, as classes are not permitted in OOM use case diagrams) |
| Use case: <ul style="list-style-type: none"><li>• Diagrams</li></ul>                           | Use case: <ul style="list-style-type: none"><li>• Related diagrams</li></ul>  |
| Association: <ul style="list-style-type: none"><li>• Navigable</li></ul>                       | Association: <ul style="list-style-type: none"><li>• Orientation</li></ul>    |

### i Note

Dependencies between a use case and an actor are not imported.

## 1.10.2 Importing Rational Rose Class Diagrams

PowerDesigner can import the most important objects in Rose class diagrams.

Only the listed properties are imported:

| Rose Object   | OOM Object   |
|---|--|
| Class: <ul style="list-style-type: none"><li>• 'Class utility' Type</li><li>• Export Control</li><li>• Implementation</li><li>• Cardinality: 0..n, 1..n</li><li>• Nested class</li><li>• Persistence</li><li>• Abstract</li></ul> | Class: <ul style="list-style-type: none"><li>• 'Class' Type</li><li>• Visibility</li><li>• Package</li><li>• Cardinality: 0..*, 1..*</li><li>• Inner classifier</li><li>• Persistence</li><li>• Abstract</li></ul> |

| Rose Object  | OOM Object   |
|--|--|
| Interface: <ul style="list-style-type: none"><li>• Export Control</li><li>• Implementation</li><li>• Nested class</li></ul>  | Interface: <ul style="list-style-type: none"><li>• Visibility</li><li>• Package</li><li>• Inner classifier</li></ul>   |
| Attribute: <ul style="list-style-type: none"><li>• Export Control</li><li>• Implementation</li><li>• Initial value</li><li>• Static</li><li>• Derived</li></ul>  | Attribute: <ul style="list-style-type: none"><li>• Visibility</li><li>• Package</li><li>• Initial value</li><li>• Static</li><li>• Derived</li></ul>   |
| Operation: <ul style="list-style-type: none"><li>• Export Control</li><li>• Implementation</li></ul>   | Operation: <ul style="list-style-type: none"><li>• Visibility</li><li>• Package</li></ul>  |
| Generalization: <ul style="list-style-type: none"><li>• Export Control</li><li>• Implementation</li><li>• Virtual inheritance</li><li>• Multi inheritance</li></ul>  | Generalization: <ul style="list-style-type: none"><li>• Visibility</li><li>• Package</li><li>• Extended attribute</li><li>• Multi inheritance</li></ul>  |
| Association: <ul style="list-style-type: none"><li>• Role name</li><li>• Export Control</li><li>• Navigable</li><li>• Cardinality</li><li>• Aggregate (class A or B)</li><li>• Aggregate (by reference)</li><li>• Aggregate (by value)</li></ul> | Association: <ul style="list-style-type: none"><li>• Role name</li><li>• Visibility</li><li>• Navigable</li><li>• Multiplicity</li><li>• Container</li><li>• Aggregation</li><li>• Composition</li></ul> |
| Dependency   | Dependency   |

### 1.10.3 Importing Rational Rose Collaboration Diagrams

PowerDesigner can import the most important objects in Rose collaboration diagrams.

Only the listed properties are imported:

| Rose Object   | OOM Object   |
|---|--|
| Object/class instance: <ul style="list-style-type: none"><li>• Class</li><li>• Multiple instances</li></ul>   | Object: <ul style="list-style-type: none"><li>• Class or interface</li><li>• Multiple</li></ul>  |
| Link/object link: <ul style="list-style-type: none"><li>• Assoc</li><li>• Messages list</li></ul>   | Instance link: <ul style="list-style-type: none"><li>• Association</li><li>• Messages</li></ul>  |
| Actor   | Actor  |
| Message: <ul style="list-style-type: none"><li>• Simple stereotype</li><li>• Synchronous stereotype</li><li>• Asynchronous stereotype</li><li>• Balking</li></ul> | Message: <ul style="list-style-type: none"><li>• Undefined</li><li>• Procedure call</li><li>• Asynchronous</li><li>• Condition</li></ul> |

### 1.10.4 Importing Rational Rose Sequence Diagrams

PowerDesigner can import the most important objects in Rose sequence diagrams.

Only the listed properties are imported:

| Rose Object   | OOM Object   |
|---|--|
| Object/class instance: <ul style="list-style-type: none"><li>• Persistence</li><li>• Multiple instances</li></ul> | Object: <ul style="list-style-type: none"><li>• Persistence</li><li>• Multiple</li></ul> |
| Actor   | Actor  |

| Rose Object   | OOM Object  |
|---|---|
| <p>Message:</p> <ul style="list-style-type: none"> <li>• Simple stereotype</li> <li>• Synchronous stereotype</li> <li>• Asynchronous stereotype</li> <li>• Balking</li> <li>• Return message</li> <li>• Destruction marker</li> </ul> | <p>Message:</p> <ul style="list-style-type: none"> <li>• Undefined</li> <li>• Procedure call</li> <li>• Asynchronous</li> <li>• Condition</li> <li>• Return</li> <li>• Recursive message with Destroy action</li> </ul> |

## 1.10.5 Importing Rational Rose Statechart Diagrams

PowerDesigner can import the most important objects in Rose statechart diagrams.

In Rose, activity and statechart diagrams are created in the Use Case or Logical View:

- At the root level
- In an activity
- In a state

A UML State Machine is automatically created, which contains statechart and activity diagrams with their relevant objects.

In PowerDesigner, statechart diagrams are created at the model level or in a composite state: the parent package or the model is considered the State Machine.

Rose statechart diagrams that are at the root level, or in a state are imported, but those that are in an activity are not imported.

Only the listed properties are imported:

| Rose Object  | OOM Object   |
|--|--|
| <p>State:</p> <ul style="list-style-type: none"> <li>• When action</li> <li>• OnEntry action</li> <li>• OnExit action</li> <li>• Do action</li> <li>• OnEvent action</li> <li>• Event action</li> <li>• Event arguments</li> </ul> | <p>State or object node:</p> <ul style="list-style-type: none"> <li>• Trigger event</li> <li>• Entry</li> <li>• Exit</li> <li>• Do</li> <li>• Event</li> <li>• Trigger event</li> <li>• Event arguments</li> </ul> |

| Rose Object  | OOM Object   |
|--|--|
| <p>State transition:</p> <ul style="list-style-type: none"> <li>• &lt;No name&gt;</li> <li>• &lt;No code&gt;</li> <li>• Event</li> <li>• Arguments</li> <li>• Guard condition</li> <li>• Action</li> </ul> | <p>Transition:</p> <ul style="list-style-type: none"> <li>• Calculated name</li> <li>• Calculated Code</li> <li>• Trigger Event</li> <li>• Event arguments</li> <li>• Condition</li> <li>• Trigger action</li> </ul> |

## 1.10.6 Importing Rational Rose Activity Diagrams

PowerDesigner can import the most important objects in Rose activity diagrams.

Only the listed properties are imported:

| Rose Object  | OOM Object  |
|--|---|
| <p>Activity:</p> <ul style="list-style-type: none"> <li>• Actions</li> </ul> | <p>Activity:</p> <ul style="list-style-type: none"> <li>• Action</li> </ul> |
| Object (associated with a state)   | Object node   |
| State  | State (no symbol in activity diagram)                                       |
| Start state  | Start   |
| Self Transition/Object Flow  | Transition  |
| Synchronization  | Synchronization   |
| Decision   | Decision  |
| End state  | End   |
| Swimlane   | Organization unit/swimlane  |

### i Note

- PowerDesigner does not support multiple actions on an activity. After import, the Action tab in the OOM activity property sheet displays <<Undefined>> and the text zone reproduces the list of imported actions.
- PowerDesigner does not manage Rose Subunits as separate files, but rather imports \*.CAT and \*.SUB files into the model that references them. If a .CAT or a .SUB file does not exist in the specified path, PowerDesigner searches in the same directory as the file containing the model.
- In Rose, you can associate an Object (instance of a class) with a State. The Rose Object is imported as an object without symbol. If it is associated with a State, an object node with a symbol is created with the name, stereotype and comment of the State. If the Rose diagram that contains the symbol of the Object is

in a composite activity, a shortcut of the object is created in the imported composite activity, because PowerDesigner does not support decomposition of object nodes.

## 1.10.7 Importing Rational Rose Component Diagrams

PowerDesigner can import the most important objects in Rose component diagrams.

Only the listed properties are imported:

| Rose Object  | OOM Object  |
|--|---|
| Component: <ul style="list-style-type: none"><li>• Interface of realize</li><li>• Class of realize</li><li>• File</li><li>• URL file</li><li>• Declaration</li></ul> | Component: <ul style="list-style-type: none"><li>• Interface</li><li>• Class</li><li>• External file in Traceability Links</li><li>• URL file in Traceability Links</li><li>• <i>Description</i> in <i>Definition</i> tab</li></ul> |

The following types of components, which have Rose predefined stereotypes and different symbols are imported. The stereotypes are preserved, but each will have a standard OOM component symbol:

- Active X
- Applet
- Application
- DLL
- EXE
- Generic Package
- Generic Subprogram
- Main Program
- Package Body
- Package Specification
- Subprogram Body
- Subprogram Specification
- Task Body
- Task Specification

## 1.10.8 Importing Rational Rose Deployment Diagrams

PowerDesigner can import the most important objects in Rose deployment diagrams.

Only the listed properties are imported:

| Rose Object   | OOM Object  |
|---|---|
| Node: <ul style="list-style-type: none"><li>• Device</li><li>• Processor</li><li>• Device characteristics</li><li>• Processor characteristics</li></ul> | Node: <ul style="list-style-type: none"><li>• Node</li><li>• Node</li><li>• Description</li><li>• Description</li></ul> |
| File objects: <ul style="list-style-type: none"><li>• File</li><li>• URL definition</li></ul>   | File objects: <ul style="list-style-type: none"><li>• File object linked to the package</li><li>• File object</li></ul> |
| Node association: <ul style="list-style-type: none"><li>• Connection</li><li>• Characteristics</li></ul>  | Node association: <ul style="list-style-type: none"><li>• Node association</li><li>• Description</li></ul>              |

## 1.11 Importing and Exporting an OOM in XMI Format

PowerDesigner supports the import and export of XML Metadata Interchange (XMI) UML v2.x files, an open format that allows you to transfer UML models between different tools. All of the OOM objects can be imported and exported.

### 1.11.1 Importing XMI Files

PowerDesigner supports the import of an OOM from an XMI v2.x file. Since XMI only supports the transfer of objects, PowerDesigner assigns default symbols to imported objects and assigns them to default diagrams.

#### Context

## Procedure

1. Select **File > Import > XMI File** to open the New Model dialog.
2. Select an object language, specify the first diagram in the model and click **OK**.
3. Select the .XML or .XMI file to import and click **Open**.

The **General** tab in the Output window shows the objects being imported. When the import is complete, your specified first diagram opens in the canvas.

## 1.11.2 Exporting XMI Files

PowerDesigner supports the export of an OOM to an XMI v2.x file. Since XMI only supports the transfer of objects, any associated PowerDesigner symbols and diagrams are not preserved during the export.

## Context

PowerDesigner exports to UML container objects as follows:

| UML Container | PowerDesigner Object | PowerDesigner Diagram  |
|---------------|----------------------|--|
| Activity      | Composite Activity   | Activity Diagram   |
| State Machine | Composite State      | State Diagram  |
| Interaction   | Package              | Sequence Diagram,<br>Communication Diagram, or<br>Interaction Overview Diagram |

## Procedure

1. Select **File > Export > XMI File** to open a standard Save As dialog box.
2. Enter a filename for the file to be exported, select the appropriate type, and click Save.

The **General** tab in the Output window shows the objects being exported. You can open the resulting XMI file in any modeling tool that supports this exchange format or a code generator such as Java, CORBA, or C++.

## 2 Object Language Definition Reference

The chapters in this part provide information specific to the object languages supported by PowerDesigner.

### 2.1 Java

PowerDesigner supports the modeling and round-trip reverse-engineering and generation of Java 5 code including annotations and generics.

**i** Note

Support for earlier versions, through Java 1.4 is deprecated.

For information specific to modeling for Java in the Eclipse environment, see *Core Features Guide > Modeling with PowerDesigner > The PowerDesigner Plugin for Eclipse*.

#### 2.1.1 Java Public Classes

Java allows the creation of multiple classes in a single file but one class, and only one, has to be public.

In PowerDesigner, you should create one public class and several dependent classes and draw dependency links with stereotype <<sameFile>> between them. This type of link is handled during generation and reverse engineering.

#### 2.1.2 Java Enumerated Types (Enums)

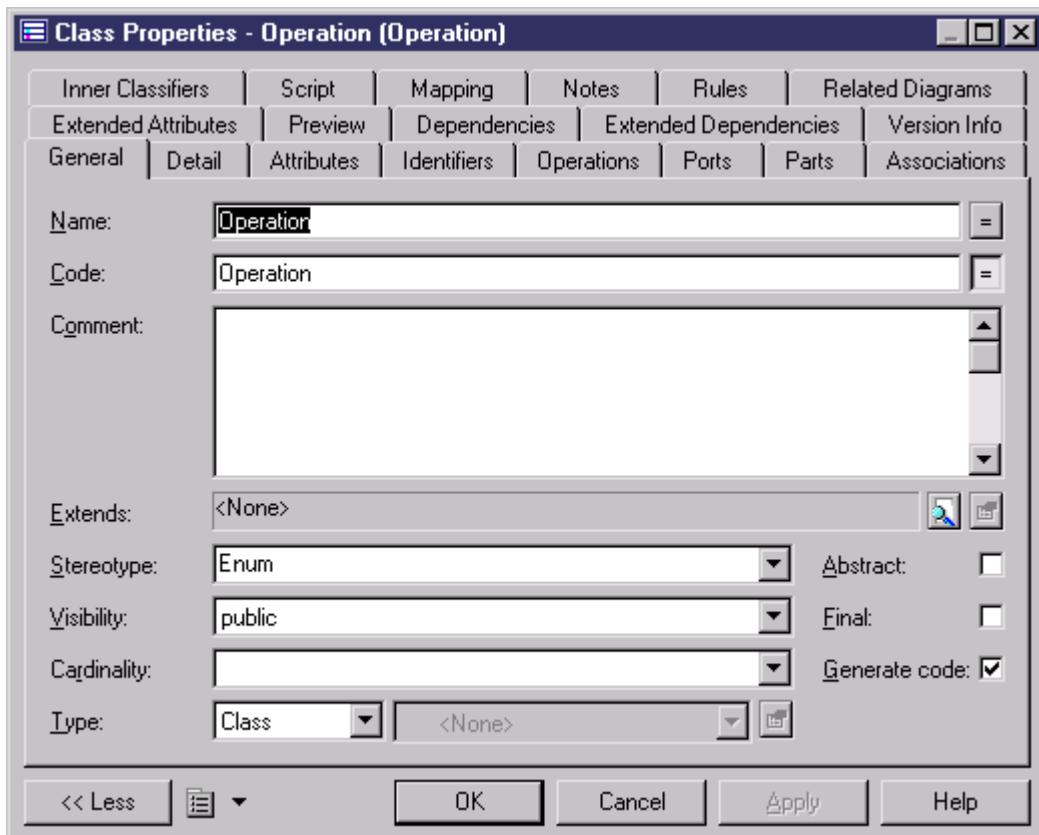
Java 5 supports enumerated types. These replace the old typesafe enum pattern, and are much more compact and easy to maintain. They can be used to list such collections of values as the days of the week or the suits of a deck of cards, or any fixed set of constants, such as the elements in a menu.

### Context

An enum is represented by a class with an <<enum>> stereotype.

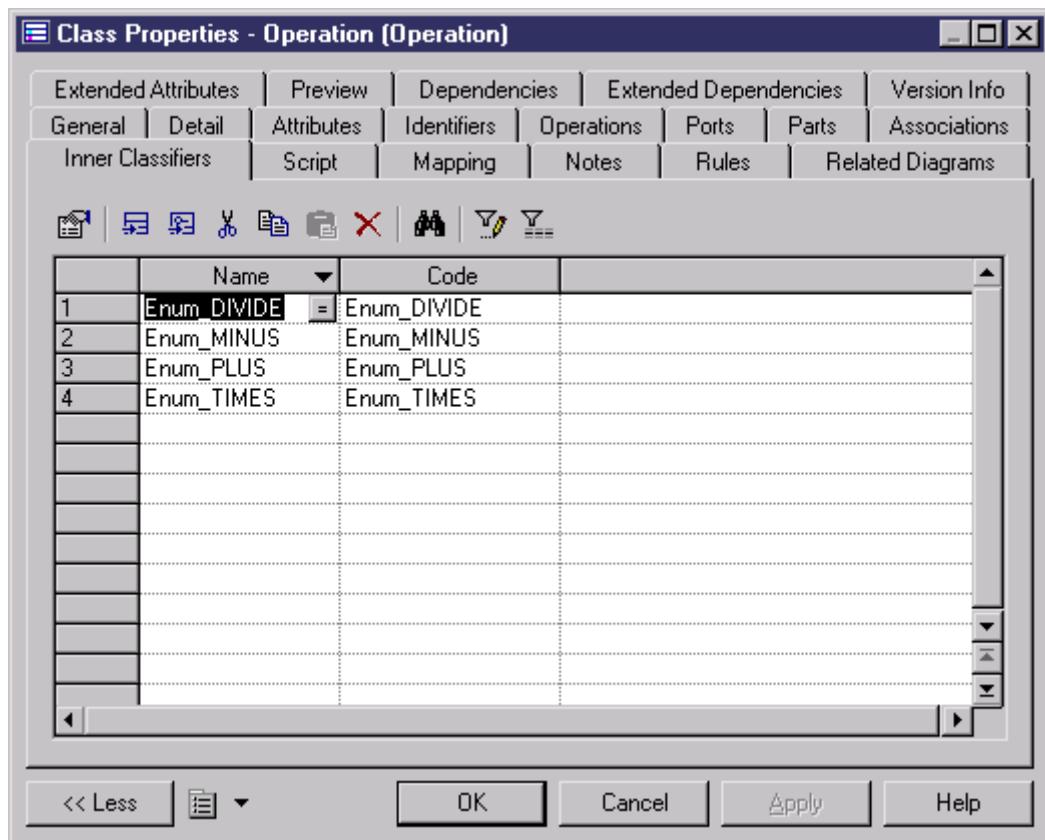
## Procedure

1. Create a class in a class diagram or composite structure diagram, and double-click it to open its property sheet.
2. On the *General* tab, select <>enum>> from the *Stereotype* list.

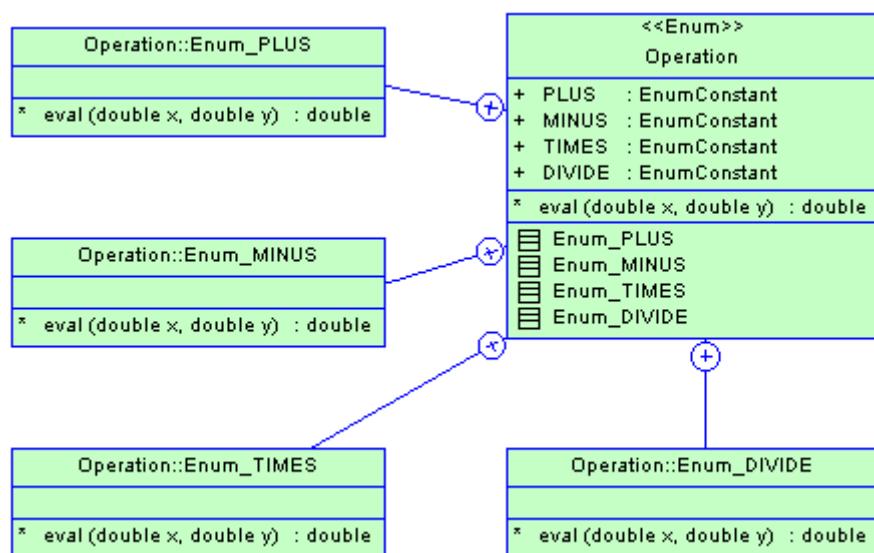


3. Click the *Attributes* tab, and add as many attributes as necessary. These attributes have, by default, a data type of *EnumConstant*. For example, to create an enum type that contained standard mathematical operations, you would create four *EnumConstants* with the names "PLUS", "MINUS", "TIMES", and "DIVIDE".  
Note that, since a Java enum is a full featured class, you can also add other kinds of attributes to it by clicking in the *Data Type* column and selecting another type from the list.
4. [optional] You can create an anonymous class for an *EnumConstant* by selecting its row on the *Attributes* tab and clicking the *Properties* tool to open its property sheet. On the *General* tab, click the *Create* tool next to the *Enum class* box to create the internal class.

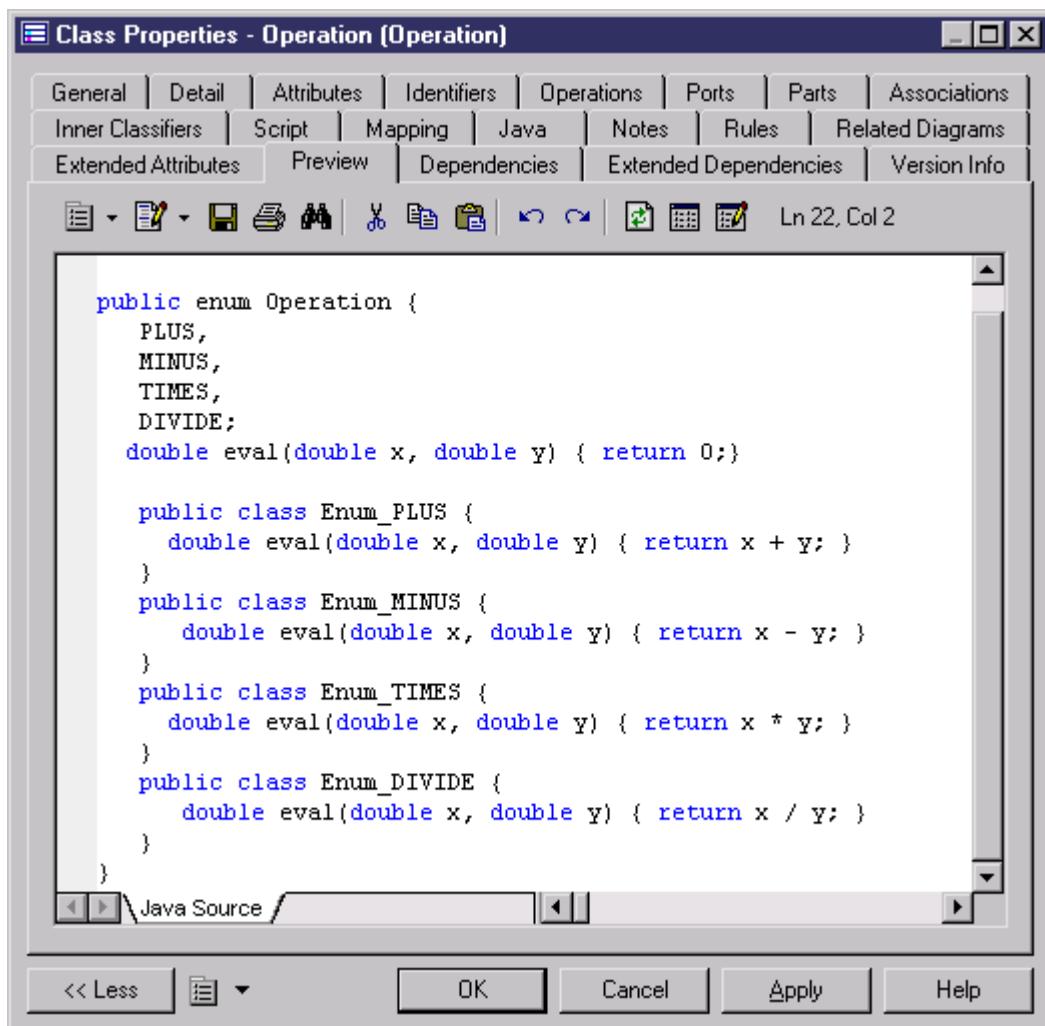
These anonymous classes will be displayed on the *Inner Classifiers* tab of the *Enum class* property sheet:



The Operation enum class, with an anonymous class for each of its EnumConstants to allow for the varied arithmetic operations, could be represented in a class diagram as follows:



The equivalent code would be like the following:



The screenshot shows a software interface for managing class properties. The title bar reads "Class Properties - Operation (Operation)". The main area displays Java source code for an enum named "Operation". The code defines four constants: PLUS, MINUS, TIMES, and DIVIDE. It also contains nested classes named after these constants, each overriding the "eval" method to perform addition, subtraction, multiplication, or division respectively. The code is color-coded, with keywords in blue and identifiers in black. The interface includes tabs for General, Detail, Attributes, Identifiers, Operations, Ports, Parts, Associations, Inner Classifiers, Script, Mapping, Java, Notes, Rules, Related Diagrams, Extended Attributes, Preview, Dependencies, Extended Dependencies, and Version Info. A toolbar with various icons is at the top, and a status bar at the bottom indicates "Ln 22, Col 2".

```
public enum Operation {
    PLUS,
    MINUS,
    TIMES,
    DIVIDE;
    double eval(double x, double y) { return 0; }

    public class Enum_PLUS {
        double eval(double x, double y) { return x + y; }
    }
    public class Enum_MINUS {
        double eval(double x, double y) { return x - y; }
    }
    public class Enum_TIMES {
        double eval(double x, double y) { return x * y; }
    }
    public class Enum_DIVIDE {
        double eval(double x, double y) { return x / y; }
    }
}
```

### 2.1.3 JavaDoc Comments

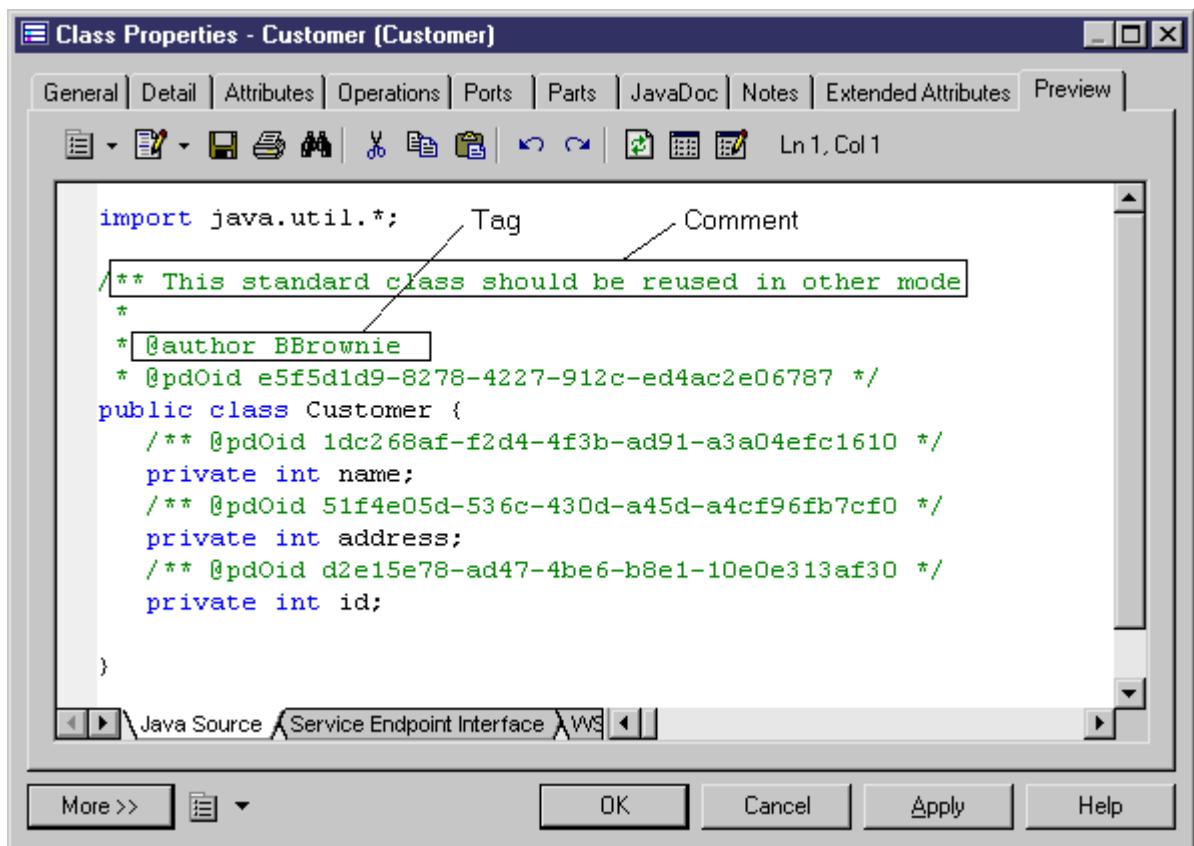
Javadoc is a tool delivered in the JDK that parses the declarations and documentation comments in a set of Java source files and produces a corresponding set of HTML pages describing model objects.

Javadoc comments are included in the source code of an object, immediately before the declaration of any object, between the characters `/**` and `*/`.

A Javadoc comment can contain:

- A description after the starting delimiter `/**`. This description corresponds to a Comment in OOM objects
- Tags prefixed by the `@` character

For example, in the following code preview page, you can read the tag `@author`, and the comment inserted from the Comment box in the General page of the class property sheet.



The following table summarizes the support of Javadoc comments in PowerDesigner:

| Javadoc     | Description  | Applies to                             | Corresponding extended attribute |
|-------------|--|--|----------------------------------|
| %comment%   | Comment box. If Javadoc comments are not found, standard comments are reversed instead | Class, interface, operation, attribute | —                                |
| @since      | Adds a "Since" heading with the specified since-text to the generated documentation    | Class, interface, operation, attribute | Javadoc since                    |
| @deprecated | Adds a comment indicating that this API should no longer be used                       | Class, interface, operation, attribute | Javadoc deprecated               |

| Javadoc      | Description  | Applies to                             | Corresponding extended attribute   |
|--------------|--|--|--|
| @author      | Adds an Author entry   | Class, interface                       | Javadoc author.<br><br>If the tag is not defined, the user name from the Version Info page is used, otherwise the defined tag value is displayed |
| @version     | Adds a Version entry, usually referring to the version of the software                           | Class, interface                       | Javadoc version  |
| @see         | Adds a "See Also" heading with a link or text entry that points to reference                     | Class, interface, operation, attribute | Javadoc see  |
| @return      | Adds a "Returns" section with the description text   | Operation                              | Javadoc misc   |
| @throws      | Adds a "Throws" subheading to the generated documentation  | Operation                              | Javadoc misc. You can declare operation exceptions   |
| @exception   | Adds an "Exception" subheading to the generated documentation                                    | Operation                              | Javadoc misc. You can declare operation exceptions   |
| @serialData  | Documents the types and order of data in the serialized form                                     | Operation                              | Javadoc misc   |
| @serialField | Documents an Object-StreamField component of a Serializable class' serialPersistentFields member | Attribute                              | Javadoc misc   |
| @serial      | Used in the doc comment for a default serializable field   | Attribute                              | Javadoc misc   |
| @param       | Adds a parameter to the Parameters section   | Attribute                              | Javadoc misc   |

## 2.1.3.1 Defining Values for Javadoc Tags

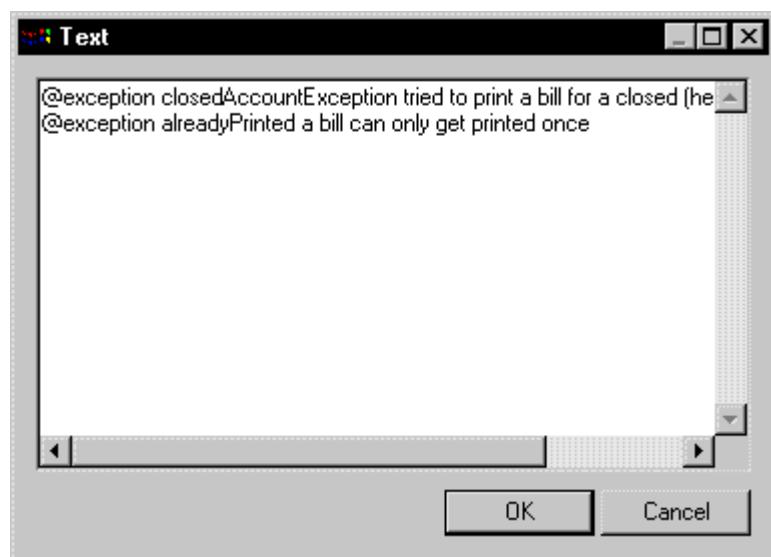
You can define values for Javadoc tags from the *JavaDoc* tab of an object property sheet.

To do so, you have to select a Javadoc tag in the list of extended attributes and click the ellipsis button in the *Value* column. The input dialog box that is displayed allows you to create values for the selected Javadoc tag. For example, if the data type is set to (Color), you can select another color in the *Color* dialog box by clicking the ellipsis button in the *Value* column.

Note: You define values for the @return, @exception, @throws, @serialData, @serialField, @serial JavaDoc, and @param comments in the Javadoc@misc extended attribute.

Javadoc tags are not generated if no extended attribute is valued.

Do not forget to repeat the Javadoc tag before each new value, some values being multi-line. If you do not repeat the Javadoc tag, the values will not be generated.



When you assign a value to a Javadoc tag, the tag and its value appear in the code preview page of the corresponding object.

### @author

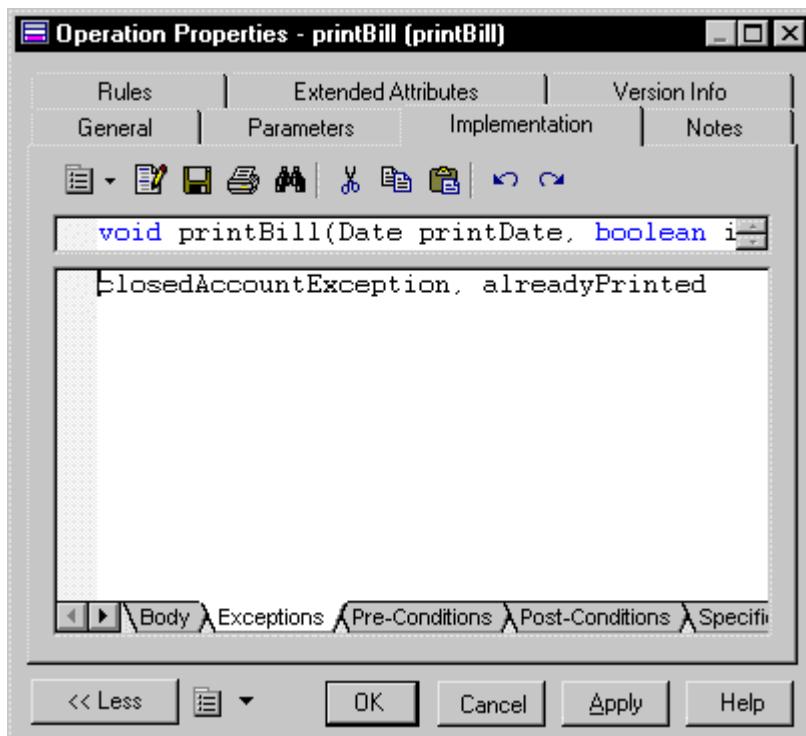
@author is not generated if the extended attribute has no value.

### @exceptions and @throws

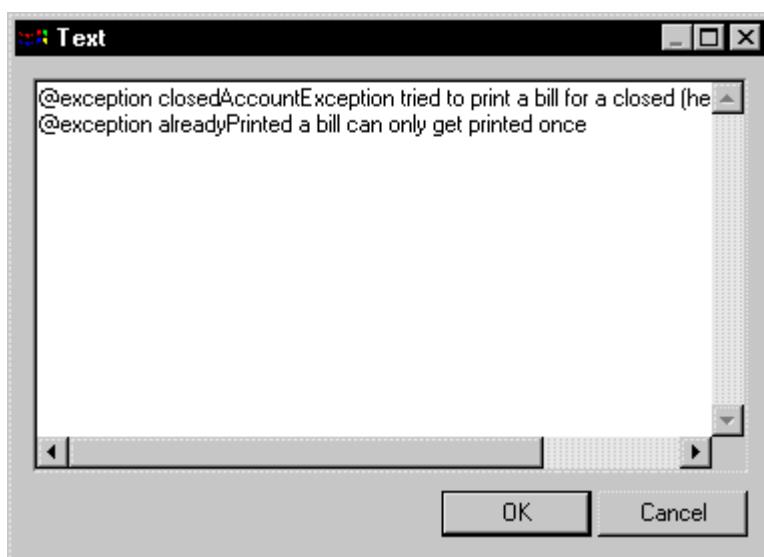
@exceptions and @throws are synonymous Javadoc tags used to define the exceptions that may be thrown by an operation.

To use these tags you can proceed as follows:

- From the operation property sheet, click the *Implementation* tab and click the *Exceptions* tab to open the Exceptions page. You can type exceptions in this page.

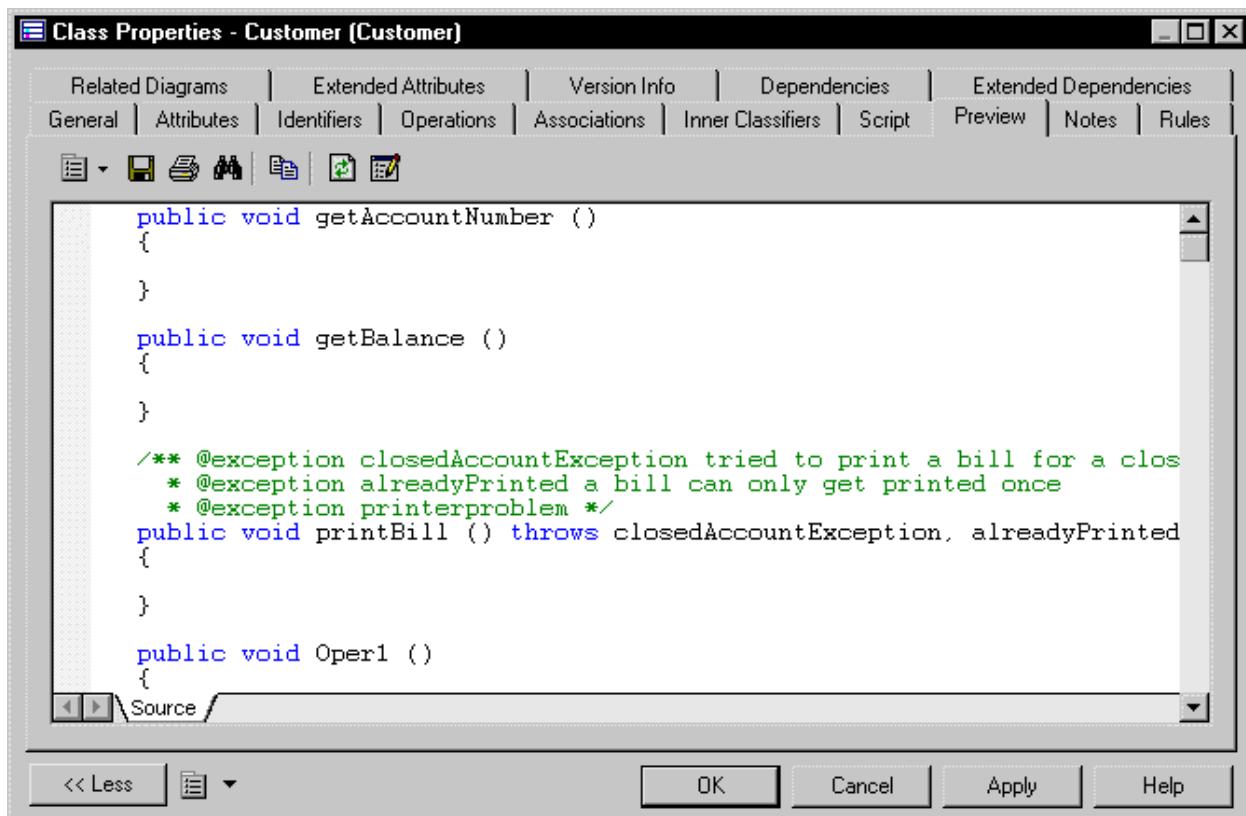


- In the same property sheet, click the *Extended Attributes* tab, select the Javadoc@exception line in the list and click the ellipsis button in the *Value* column. Type values for each declared exception, do not forget to repeat @exception or @throws before each exception.



It is also possible to type values directly after the @exception or the @throws tags in the *Extended Attributes* page. These comments describe exceptions that are not listed in the operation exceptions, and do not appear after the throws parameter in the *Preview* page of the class.

When you preview the generated code, each exception is displayed with its value:



The screenshot shows the 'Class Properties - Customer (Customer)' dialog box. The 'Script' tab is selected in the top navigation bar. The main area displays Java code with Javadoc comments. The code includes methods for getting account number and balance, and a method for printing a bill that includes exception annotations.

```
public void getAccountNumber ()  
{  
}  
  
public void getBalance ()  
{  
}  
  
/** @exception closedAccountException tried to print a bill for a clos  
 * @exception alreadyPrinted a bill can only get printed once  
 * @exception printerproblem */  
public void printBill () throws closedAccountException, alreadyPrinted  
{  
}  
  
public void Oper1 ()  
{
```

### 2.1.3.2 Javadoc Comments Generation and Reverse Engineering

You recover Javadoc comments in classes, interfaces, operations, and attributes during reverse engineering. Javadoc comments are reverse engineered only if they exist in the source code.

The Reverse Javadoc Comments feature is very useful for round-trip engineering: you keep Javadoc comments during reverse engineering and you can regenerate the code with preserved Javadoc comments. The generation process generates object comments compliant with Javadoc

For more information on the generation of Javadoc comments, see [Javadoc Comments Generation and Reverse Engineering \[page 300\]](#).

## 2.1.4 Java 5.0 Annotations

PowerDesigner provides full support for Java 5.0 annotations, which allow you to add metadata to your code. This metadata can be accessed by post-processing tools or at run-time to vary the behavior of the system.

You can use built-in annotations, such as those listed below, and also create your own annotations, to apply to your types.

There are three types of annotations available:

- Normal annotations – which take multiple arguments
- Single member annotations – which take only a single argument, and which have a more compact syntax
- Marker annotations – which take no parameters, and are used to instruct the Java compiler to process the element in a particular way

PowerDesigner supports the seven built-in Java 5.0 annotations:

- `java.lang.Override` - specifies that a method declaration will override a method declaration in a superclass, and will generate a compile-time error if this is not the case.
- `java.lang.Deprecated` – specifies that an element is deprecated, and generates a compile-time warning if it is used in non-deprecated code.
- `java.lang.SuppressWarning` – specifies compile-time warnings that should be suppressed for the element.
- `java.lang.annotation.Documented` – specifies that annotations with a type declaration are to be documented by javadoc and similar tools by default to become part of the public API of the annotated elements.
- `java.lang.annotation.Inherited` – specifies that an annotation type is automatically inherited for a superclass
- `java.lang.annotation.Retention` – specifies how far annotations will be retained during processing. Takes one of the following values:
  - `SOURCE` – annotations are discarded at compile-time
  - `CLASS` – [default] annotations are retained by the compiler, but discarded at run-time
  - `RUNTIME` – annotations are retained by the VM at run-time
- `java.lang.annotation.Target` – restricts the kind of program element to which an annotation may be applied and generates compile-time errors. Takes one of the following values:
  - `TYPE` – class, interface, or enum declaration
  - `FIELD` – including enum constants
  - `METHOD`
  - `PARAMETER`
  - `CONSTRUCTOR`
  - `LOCAL_VARIABLE`
  - `PACKAGE`

For general information about modeling this form of metadata in PowerDesigner, see [Attributes \(OOM\) \[page 67\]](#).

## 2.1.5 Java Strictfp Keyword

Floating-point hardware may calculate with more precision and with a greater range of values than required by the Java specification. The strictfp keyword can be used with classes or operations in order to specify compliance with

the Java specification (IEEE-754). If this keyword is not set, then floating-point calculations may vary between environments.

## Context

To enable the Strictfp keyword:

## Procedure

1. Double-click a class to open its property sheet.
2. Click the *Extended Attributes* tab, and then click the *Java* sub-tab.
3. Find the strictfp entry in the *Name* column, and set the Value to "true".
4. Click *OK*. The class will be generated with the strictfp keyword, and will perform operations in compliance with the Java floating-point specification

## 2.1.6 Enterprise Java Beans (EJBs) V2

The Java TM 2 Platform, Enterprise Edition (J2EE TM) is a Java platform that defines the standard for developing multi-tier enterprise applications. J2EE simplifies enterprise applications by basing them on standardized, reusable modular components, it provides a complete set of services to those components, and handles many details of application behavior automatically without complex programming.

PowerDesigner supports the EJB 2.0 specification, with special emphasis on entity beans (both CMP and BMP), and allows you to take full advantage of the tight integration between the PDM and OOM.

To work with EJB 2.0, you require Java 2 SDK Standard Edition (J2SE TM) 1.3 (final release), Java 2 SDK Enterprise Edition (J2EE TM) 1.3 (final release), a Java IDE or a text editor and a J2EE application server supporting EJB 2.0.

We recommend that you add the `JAVA_HOME` and `J2EE_HOME` system variables in your environment as follows:

In CLASSPATH:

```
%JAVA_HOME%\lib;%J2EE_HOME%\lib\j2ee.jar;%J2EE_HOME%\lib
```

In Path:

```
%JAVA_HOME%\bin;%J2EE_HOME%\bin
```

For detailed information on EJB, see the Oracle Java Web site, at <http://www.oracle.com/technetwork/java/index.html>.

## 2.1.6.1 Using EJB Types

You can define the following types of EJB components:

| Type                                   | Definition  |
|--|---|
| Entity Beans                           | Designed to represent data in the database; they wrap data with business object semantics, read and update data automatically. Entity beans include: CMP (Container Managed Persistence) With CMP Entity Beans, persistence is handled by the component server (also known as Container) BMP (Bean Managed Persistence) With BMP Entity Beans, persistence management is left to the bean developer |
| Session Beans (Stateful and Stateless) | Encapsulate business logic and provide a single entry point for client users. A session bean will usually manage, and provide indirect access to several entity beans. Using this architecture, network traffic can be substantially reduced. There are Stateful and Stateless beans (see below)  |
| Message Driven Beans                   | Anonymous beans that cannot be referenced by a given client, but rather respond to JMS asynchronous messages. Like Session Beans, they provide a way of encapsulating business logic on the server side   |

### Entity Beans

Entity beans are used to represent underlying objects. The most common application for entity beans is their representation of data in a relational database. A simple entity bean can be defined to represent a database table where each instance of the bean represents a specific row. More complex entity beans can represent views of joined tables in a database. One instance represents a specific customer and all of that customer's orders and order items.

The code generated is different depending on the type of Entity Bean (Container Managed Persistence or Bean Managed Persistence).

### Stateful and Stateless Session Beans

A session bean is an EJB in which each instance of a session bean is created through its home interface and is private to the client connection. The session bean instance cannot be easily shared with other clients, this allows the session bean to maintain the client's state. The relationship between the client and the session bean instance is one-to-one.

Stateful session beans maintain conversational state when used by a client. A conversational state is not written to a database, it is a state kept in memory while a client uses a session.

Stateless session beans do not maintain any conversational state. Each method is independent, and uses only data passed in its parameters.

## Message Driven Beans

They are stateless, server side components with transactional behavior that process asynchronous messages delivered via the Java Message Service (JMS). Applications use asynchronous messaging to communicate by exchanging messages that leave senders independent from receivers.

### 2.1.6.2 EJB Properties

The *EJB* tab in the component property sheet provides additional properties.

| Property              | Description   |
|-----------------------|---|
| Remote home interface | Defines methods and operations used in a remote client view. Extends the javax.ejb.EJBHome interface                              |
| Remote interface      | Provides the remote client view. Extends the javax.ejb.EJBObject interface  |
| Local home interface  | Defines methods and operations used locally in a local client view. Extends the javax.ejb.EJBLocalHome interface                  |
| Local interface       | Allows beans to be tightly coupled with their clients and to be directly accessed. Extends the javax.ejb.EJBLocalObject interface |
| Bean class            | Class implementing the bean business methods  |
| Primary key class     | Class providing a pointer into the database. It is linked to the Bean class. Only applicable to entity beans                      |

#### i Note

You can open the EJB page by right clicking the EJB component symbol, and selecting *EJB*.

For more information on interface methods and implementation methods, see [Understanding Operation Synchronization \[page 312\]](#).

## Previewing the Component Code

You can see the relation between an EJB and its classes and interfaces from the *Preview* tab without generating any file. To preview the code of an EJB, click the *Preview* tab in the component property sheet (see [Previewing Object Code \[page 11\]](#)). The various sub-tabs show the code for each interface and class of the EJB. In the model or package property sheet, the Preview page describes the EJB deployment descriptor file with the name of the generated EJB and its methods.

## 2.1.6.3 Creating an EJB with the Wizard

You can create an EJB component with the wizard that will guide you through the creation of the component. It is only available if the language is Java.

### Context

The wizard is invoked from a class diagram. You can either create an EJB without selecting any class, or select a class first and start the wizard from the contextual menu of the class.

You can also create several EJBs of the same type by selecting several classes at the same time. The wizard will automatically create one EJB per class. The classes you have selected in the class diagram become the Bean classes. They are renamed to fit the naming conventions standard, and they are linked to their component.

If you have selected classes or interfaces before starting the wizard, they are automatically linked to the new EJB component.

When an interface or a class is already stereotyped, like <<EJBEntity>> for example, it is primarily used to be the interface or the class of the EJB.

For more information on stereotyped EJB interface or class, see section [Defining Interfaces and Classes for EJBs \[page 307\]](#).

The EJB creation wizard lets you define the following parameters:

| Property              | Description  |
|-----------------------|--|
| Name                  | Name of the EJB component  |
| Code                  | Code of the EJB component  |
| Component type        | Entity Bean CMP, Entity Bean BMP, Message Driven Bean, Session Bean Stateful, or Session Bean Stateless For more information on the different types of EJB, see section <a href="#">Using EJB Types [page 303]</a> |
| Bean class            | Class implementing the bean business methods   |
| Remote interface      | Extends the javax.ejb.EJBObject interface and provides the remote client view  |
| Remote home interface | Defines methods and operations used in a remote client view. It extends the javax.ejb.EJBHome interface  |
| Local interface       | Extends the javax.ejb.EJBLocalObject interface, and allows beans to be tightly coupled with their clients and to be directly accessed  |
| Local home interface  | Defines methods and operations used locally in a local client view. It extends the javax.ejb.EJBLocalHome interface  |
| Primary key class     | Class providing a pointer into the database. It is only applicable to entity beans   |

| Property                                       | Description   |
|--|---|
| Transaction                                    | Defines what transaction support is used for the component. The transaction is important for distribution across a network on a server. The transaction support value is displayed in the deployment descriptor. This information is given by the deployment descriptor to the server when generating the component |
| Create symbol                                  | Creates a component symbol in the diagram specified beside the <i>Create symbol In</i> check box. If a component diagram already exists, you can select one from the list. You can also display the diagram properties by selecting the <i>Properties</i> tool  |
| Create Class Diagram for component classifiers | Creates a class diagram with a symbol for each class and interface. If you have selected classes and interfaces before starting the wizard, they are used to create the component. This option allows you to display these classes and interfaces in a diagram  |

The *Transaction support* groupbox contains the following values, as per the Enterprise JavaBeans 2.0 specification:

| Transaction value | Description   |
|-------------------|---|
| Not Supported     | The component does not support transaction, it does not need any transaction and if there is one, it ignores it |
| Supports          | The component is awaiting a transaction, it uses it   |
| Required          | If there is no transaction, one is created  |
| Requires New      | The component needs a new transaction at creation, the server must provide it with a new transaction            |
| Mandatory         | If there is no transaction, an exception is thrown  |
| Never             | There is no need for a transaction  |

The EJB deployment descriptor supports transaction type for each method: you can specify a transaction type for each method of EJB remote and local interface.

You can define the transaction type for each method using an extended attribute from the Profile/Operation/Extended Attributes/EJB folder of the Java object language. If the transaction type of the operation is not specified (it is empty), the default transaction type defined in the component is used instead.

## Procedure

1. Select  *Tools*  *Create Enterprise JavaBean*  from a class diagram to open the Enterprise JavaBean Wizard dialog.

### i Note

If you have selected classes before starting the wizard, some of the following steps may be omitted because the different names are created by default according to the names of the selected classes.

2. Enter a name and code for the component and click *Next*.
3. Select the component type and click *Next*.
4. Select the Bean class name and click *Next*.
5. Select the remote interface and the remote home interface names and click *Next*.
6. Select the local interface and the local home interface names and click *Next*.
7. Select the primary key class name and click *Next*.
8. Select the transaction support and click *Next*.
9. At the end of the wizard, you have to define the creation of symbols and diagrams.

## Results

When you have finished using the wizard, the following actions are executed:

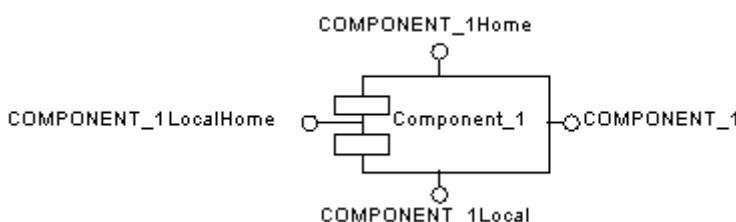
- An EJB component is created
- Classes and interfaces are associated with the component, and any missing classes or interfaces associated with the component are added
- Any diagrams associated with the component are created or updated
- Depending on the EJB type, the EJB primary key class, its interfaces and dependencies are automatically created and visible in the Browser. In addition to this, all dependencies between remote interfaces, local interfaces and the Bean class of the component are created
- The EJB created is named after the original class if you have selected a class before starting the wizard. Classes and interfaces are also prefixed after the original class name to preserve coherence

### 2.1.6.4 Defining Interfaces and Classes for EJBs

An EJB comprises a number of specific interfaces and implementation classes. Interfaces of an EJB are always exposed, you define a public interface and expose it. You can attach an interface or class to only one EJB at a time.

## Context

EJB component interfaces are shown as circles linked to the EJB component side by an horizontal or a vertical line:



Interfaces provide a remote view (Remote home interface Remote interface), or a local view (Local home interface Local interface).

Classes have no symbol in the component diagram, but the relationship between the class and the EJB component is visible from the *Classes* page of the EJB component property sheet, and from the *Components* tabbed page in the *Dependencies* page of the class property sheet.

The following table displays the stereotypes used to automatically identify EJB interfaces and classes:

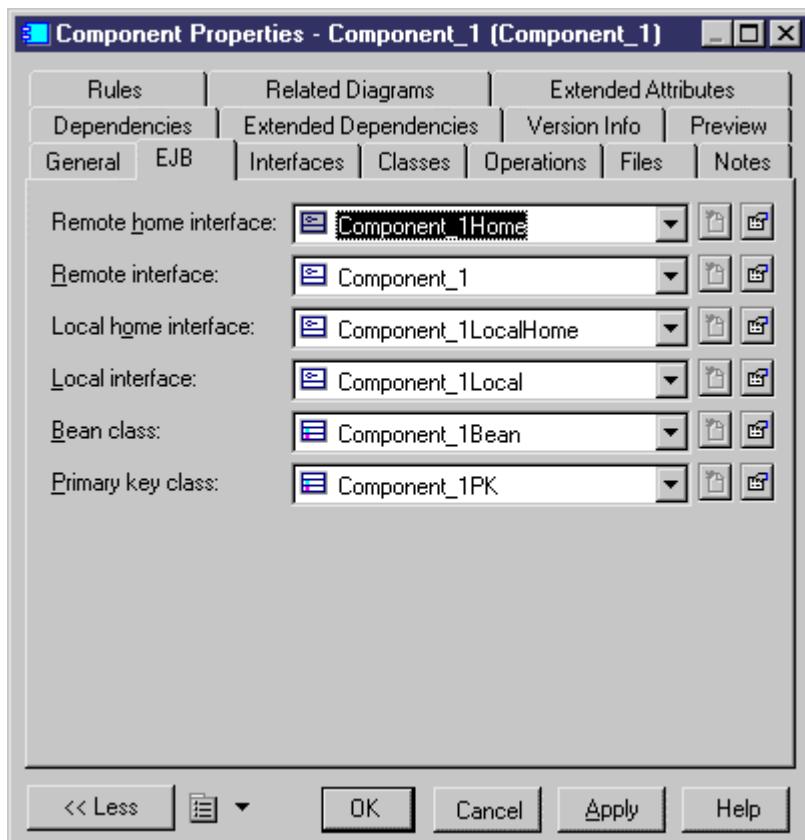
| Stereotype           | Describes                               |
|----------------------|---|
| <<EJBRMoteHome>>     | The remote home interface               |
| <<EJBRMote>>         | The remote interface                    |
| <<EJBLocalHome>>     | The local home interface                |
| <<EJBLocal>>         | The local interface                     |
| <<EJBEntity>>        | The bean class of an entity bean        |
| <<EJBSession>>       | The bean class of a session bean        |
| <<EJBMessageDriven>> | The bean class of a message driven bean |
| <<EJBPrimaryKey>>    | The primary key class of an entity bean |

Template names are instantiated with respect to the corresponding component and assigned to the newly created objects. If an unattached interface or class, matching a given name and classifier type already exists in the model, it is automatically attached to the EJB.

## Procedure

1. Right-click the EJB component in the diagram and select EJB from the contextual menu.

The component property sheet opens to the EJB page. Interfaces and classes are created and attached to the EJB.



You can use the *Create* tool beside the interface or the class name to recreate an interface or a class if it is set to <None>.

2. Click the *Properties* button beside the interface or the class name box that you want to define.

The interface or the class property sheet is displayed.

3. Select properties as required.

The interfaces and classes definitions are added to the current EJB component definition.

## 2.1.6.5 Defining Operations for EJBs

You can create the following types of operations for an EJB from the property sheet of the Bean class or EJB interfaces:

- EJB Business Method (local)
- EJB Business Method (remote)
- EJB Create Method (local)
- EJB Create Method (remote)
- EJB Finder Method (local)
- EJB Finder Method (remote)
- EJB Select Method

### **i** Note

You cannot create an operation from the Operations page of the component property sheet as this page is only used to view the operations of the EJB component. You view operations of an EJB from the Operations page in the component property sheet

## Stereotypes

The following standard stereotypes, as defined for EJBs, are assigned to these operations:

- <<EJBCreateMethod>>
- <<EJBFinderMethod>>
- <<EJBSelectMethod>>

## CMP Entity Beans

You can create the following operations for CMP Entity Beans only:

- EJB Create(...) Method (local)
- EJB Create(...) Method (remote)

When you create an EJB Create(...) Method (local), the method is created in the local home interface with all the persistent attributes as parameters. When you create an EJB Create(...) Method (remote), the method is created in the remote home interface with all the persistent attributes as parameters.

Moreover, all linked methods ejbCreate(...) and ejbPostCreate(...) are created automatically in the Bean class with all the persistent attributes as parameters. Parameters are synchronized whenever a change is applied.

### **i** Note

If you need to modify the methods of an interface, you can do so from the Operations page of the interface property sheet.

## 2.1.6.5.1 Adding an Operation to the Bean Class

After creation of the Bean class, you may need to create a method that is missing at this stage of the process, such as an internal method.

### Procedure

1. Double-click the Bean class to display its property sheet.
2. Click the *Operations* tab, then click a blank line in the list.

An arrow is displayed at the beginning of the line.

3. Double-click the arrow at the beginning of the line.

A confirmation box asks you to commit the object creation.

4. Click *Yes*.

The operation property sheet is displayed.

5. Type a name and code for your operation.
6. Click the *Implementation* tab.

The *Implementation* page opens to the *Body* tabbed page.

7. Add the method code in this page.
8. Click *OK*.

You return to the class property sheet.

9. Click the *Preview* tab.

You can now validate the to-be-generated Java code for the Bean class.

10. Click *OK*.

## 2.1.6.5.2 Adding an Operation to an EJB Interface

You can add an operation to an EJB interface from the interface property sheet or from the Bean class property sheet using the Add button in the Operations page.

### Context

When you add an operation to an interface, an implementation operation is automatically created in the Bean class because the interface operation has a linked method. This ensures operation synchronization (see [Understanding Operation Synchronization \[page 312\]](#)).

## Procedure

1. Open the Bean class property sheet and click the *Operations* tab.
2. Click the *Add...* tool and select the required EJB operation from the list.

The requested operation is created at the end of the list of operations in the Bean class property sheet, and you can also verify that the new operation is displayed in the interface operation list.

### 2.1.6.5.3 Understanding Operation Synchronization

Synchronization maintains the coherence of the whole model whenever a change is applied on operations, attributes, and exceptions. Synchronization occurs progressively as the model is modified.

#### Operation Synchronization

Synchronization occurs from interface to Bean class. Interface operations have linked methods in the Bean class with name/code, return type and parameters synchronized with the interface operation. When you add an operation to an interface, you can verify that the corresponding linked method is created in the Bean class (it is grayed in the list). Whereas no operation is created in an interface if you add an operation to a Bean class.

For example, double-click the Bean class of a component, click the Operations tab, click the Add button at the bottom of the Operations page, and select EJB Create method (local): PowerDesigner adds this operation to the interface and automatically creates the ejbCreate and ejbPostCreate operations in the Bean class.

#### Exception Synchronization

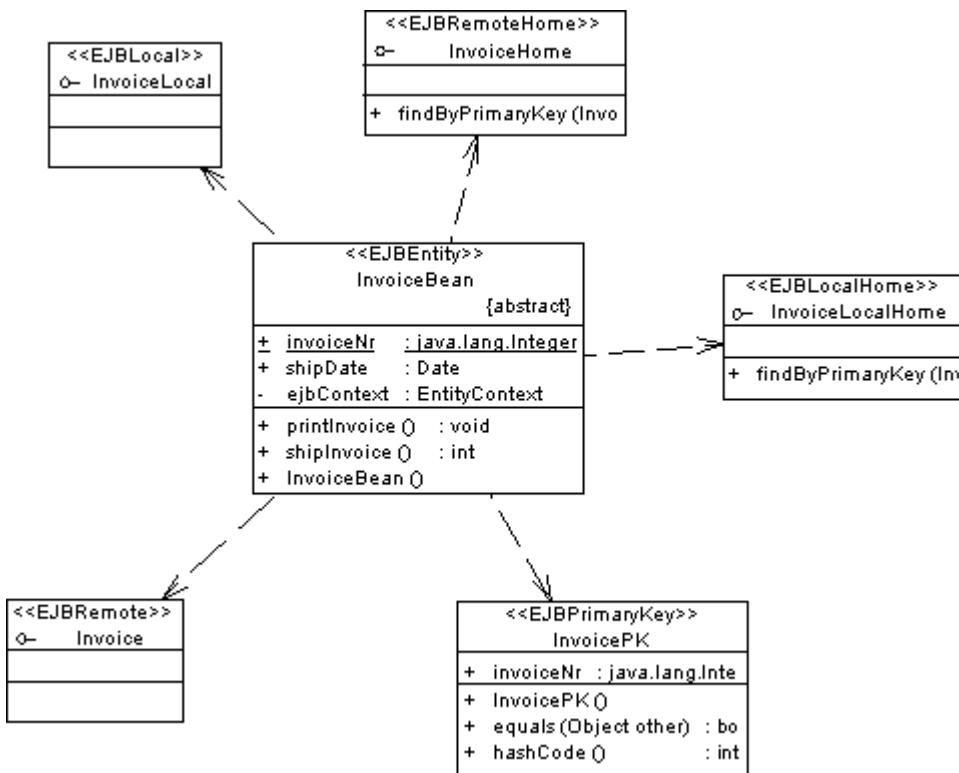
Exceptions are synchronized from the Bean class to interfaces. The exception list of the home interface create method is a superset of the union of the exception lists of the matching ejbCreate and ejbPostCreate implementation operations in the bean class.

The interface exception attributes are thus updated whenever the exception list of the bean class implementation method is modified.

### 2.1.6.6 Understanding EJB Support in an OOM

PowerDesigner simplifies the development process by transparently handling EJB concepts and enforcing the EJB programming contract.

- Automatic initialization - When creating an EJB, the role of the initialization process is to initialize classes and interfaces and their methods with respect to the EJB programming contract. PowerDesigner automatically does this whenever a class or interface is attached to an EJB



- Synchronization - maintains the coherence of the whole model whenever a change is applied:
  - Operations - Synchronization occurs from interface to Bean class with linked methods ([Understanding Operation Synchronization \[page 312\]](#)).
  - Exceptions - Synchronization occurs from Bean class to interface. ([Understanding Operation Synchronization \[page 312\]](#)).
  - Primary identifier attribute - Synchronization occurs from Bean class to Primary key class, when attribute is primary identifier in Bean class it is automatically migrated to primary key class.
- Model checks: the Check Model feature validates a given model and complements synchronization by offering auto-fixes. You can check your model at any time using the Check Model feature from the **Tools** menu (see [Checking an OOM \[page 251\]](#)).
- Template based code generation - EJB-specific templates are available in the Profile/Component/Templates category of the Java object language resource file. The generation of EJB classes and interfaces creates the following inheritance links (for Entity beans CMP):
  - The local home interface inherits `javax.ejb.EJBLocalHome`
  - The local interface inherits `javax.ejb.EJBLocalObject`
  - The remote home interface inherits `javax.ejb.EJBHome`
  - The remote interface inherits `javax.ejb.EJBObject`

**Interface Properties - INVOICE (INVOICE)**

Related Diagrams | Extended Attributes | Dependencies | Extended Dependencies | Version Info  
General | Attributes | Operations | Inner Classifiers | Script | Preview | Notes | Rules

Ln 1, Col 1

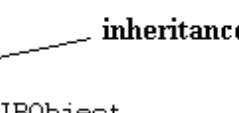
```
/*
 * Module: INVOICE.java
 * Author: lpelleta
 * Modified: Wednesday, September 10, 2002 16:17:56
 * Purpose: Defines the Interface INVOICE
 */

import java.rmi.RemoteException;
import java.util.*;

public interface INVOICE extends javax.ejb.EJBObject
{
    public java.lang.Integer getINVOICENR() throws java.rmi.RemoteException;
    public Date getSHIPDATE() throws java.rmi.RemoteException;
    public void setSHIPDATE(Date SHIPDATE) throws java.rmi.RemoteException;
}
```

Java Source

<< Less | OK | Cancel | Apply | Help



It creates the following realization links:

- The primary key class implements java.io.Serializable
- The bean class implements javax.ejb.EntityBean

**Class Properties - INVOICEBean (INVOICEBean)**

Notes | Rules | Related Diagrams | Extended Attributes | Dependencies | Extended Dependencies | Version Info  
General | Detail | Attributes | Identifiers | Operations | Associations | Inner Classifiers | Script | Preview | Mapping

Ln 1, Col 1

```
* Module: INVOICEBEAN.java
* Author: lpelleta
* Created: Wednesday, September 10, 2002 16:27:06
* Purpose: Defines the Class INVOICEBEAN
*****
import javax.ejb.*;
import java.util.*;

public abstract class INVOICEBEAN implements javax.ejb.EntityBean
{
    private EntityContext ejbContext;

    public abstract java.lang.Integer getINVOICENR();
    public abstract void setINVOICENR(java.lang.Integer INVOICENR)

    public abstract Date getSHIPDATE();
    public abstract void setSHIPDATE(Date SHIPDATE);
```

Java Source

<< Less | OK | Cancel | Apply | Help

It transforms cmp-fields (attributes flagged as Persistent) and cmr-fields (attributes that are migrated from associations) into getter and setter methods:

```

import javax.ejb.*;
import java.util.*;

public abstract class INVOICEBEAN implements javax.ejb.EntityBean
{
    private EntityContext ejbContext;

    public abstract java.lang.Integer getINVOICENR();
    public abstract void setINVOICENR(java.lang.Integer INVOICENR);

    public abstract Date getSHIPDATE();
    public abstract void setSHIPDATE(Date SHIPDATE);

    public void PRINTINVOICE()
    {
        // TODO: implement
    }
}

```

At a higher level, PowerDesigner supports different approaches to assist you in the component development process including:

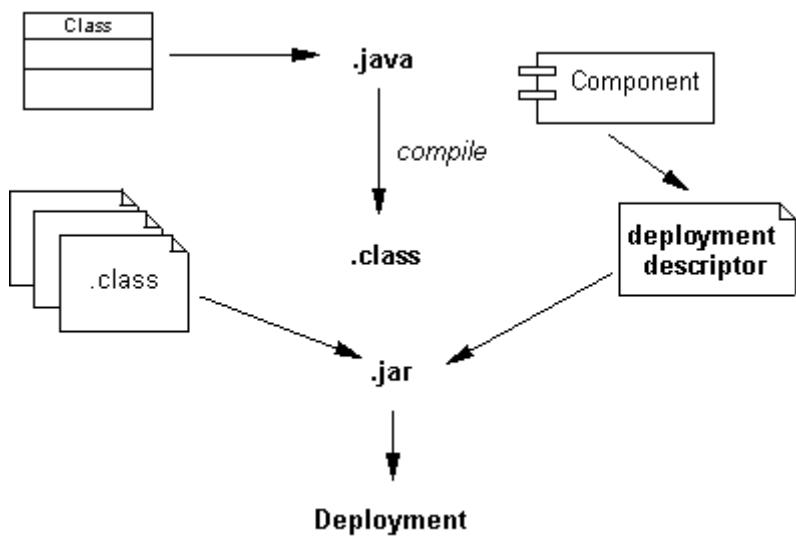
- Forward engineering: from an OOM to a PDM. It provides the ability to create and reverse EJBs in a OOM, generate the corresponding PDM, O/R mapping and generate code
- Reverse engineering: from a PDM (database) to an OOM. It provides the ability to create and reverse tables in a PDM, generate corresponding classes, create EJB from given classes and generate code

## 2.1.6.7 Previewing the EJB Deployment Descriptor

An EJB deployment descriptor describes the structure and properties of one or more EJBs in an XML file format. It is used for deploying the EJB in the application server. It declares the properties of EJBs, the relationships and the dependencies between EJBs. One deployment descriptor is automatically generated per package or model, it describes all EJBs defined in the package.

### Context

The role of the deployment descriptor, as part of the whole process is shown in the following figure:



The EJB deployment descriptor and the compiled Java classes of the EJBs should be packaged in a JAR file.

The EJB deployment tool of the application server imports the Java classes from the JAR file and configures the EJBs in the application server based on the description of EJBs contained in the EJB deployment descriptor.

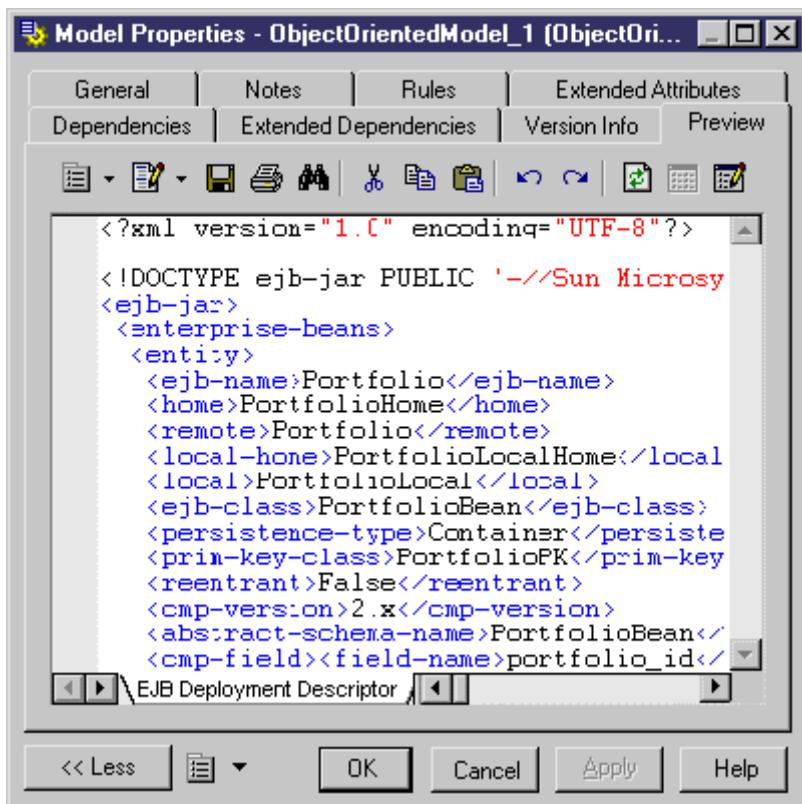
You can see the deployment descriptor from the *Preview* page of the package or model property sheet.

You can customize the EJB deployment descriptor by modifying the templates in the Java object language.

For more information on customizing the object language, see *Customizing and Extending PowerDesigner > Object, Process, and XML Language Definition Files* .

## Procedure

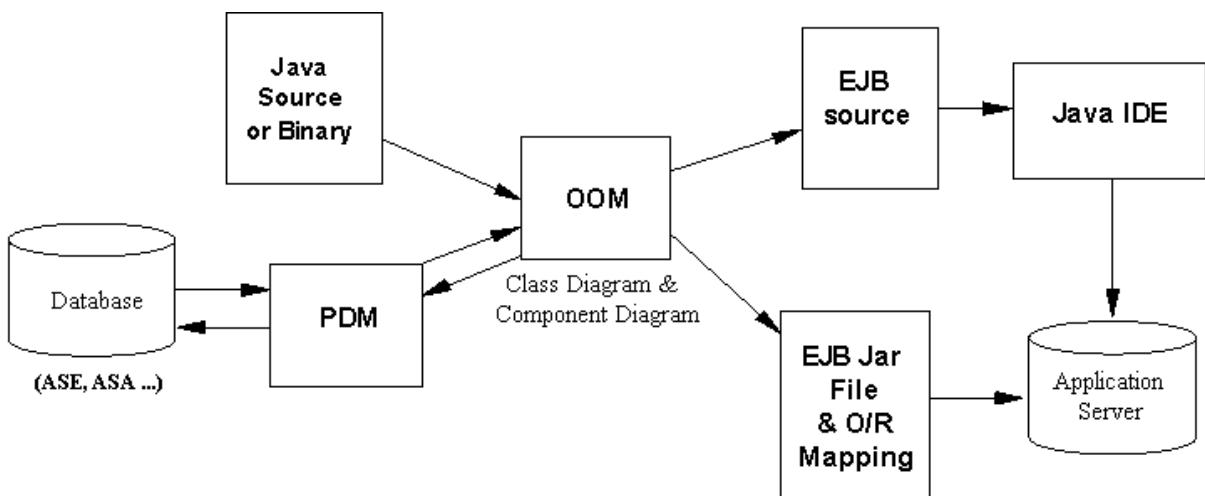
1. Right-click the model in the Browser and select *Properties* to open the model property sheet.
2. Click the *Preview* tab.



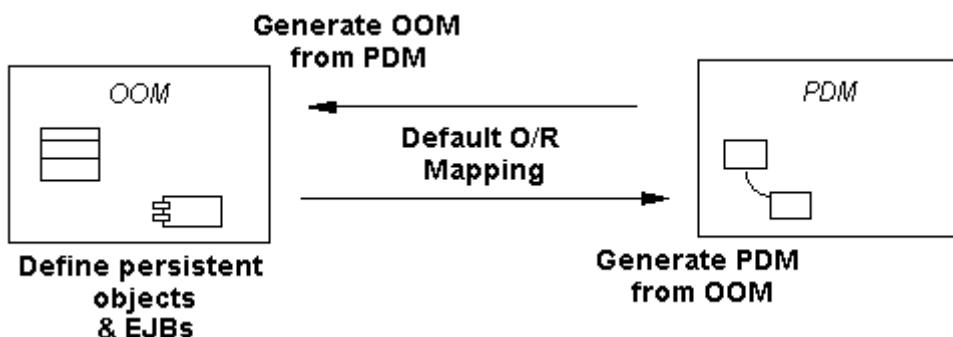
## 2.1.6.8 Generating EJBs

The EJB generation process allows you to generate EJB source code that is compliant with J2EE 1.3 (EJB 2.0). The code generator (Java, EJB, XML, etc...) is based on templates and macros. You can customize the generated code by editing the Java object language from **Tools > Resources > Object Languages**.

The following picture illustrates the overall EJB development process.



The following picture focuses on the PowerDesigner part, and highlights the role of O/R Mapping generation. The O/R mapping can be created when generating a PDM from an OOM or generating an OOM from a PDM.



For more information on object mapping, see [Object/Relational \(O/R\) Mapping \[page 430\]](#).

### 2.1.6.8.1 What Kind of Generation to Use?

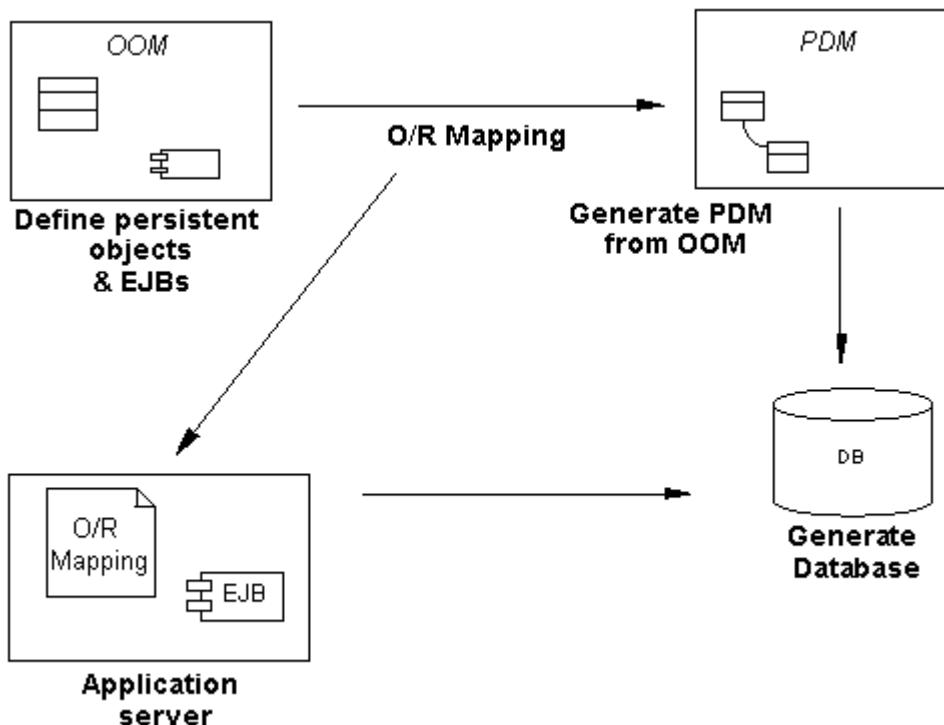
You may face two situations when generating EJB source code:

- You have no database
- You already have a database

You want to generate EJB source code when creating a database. If there is no database, the development process is the following:

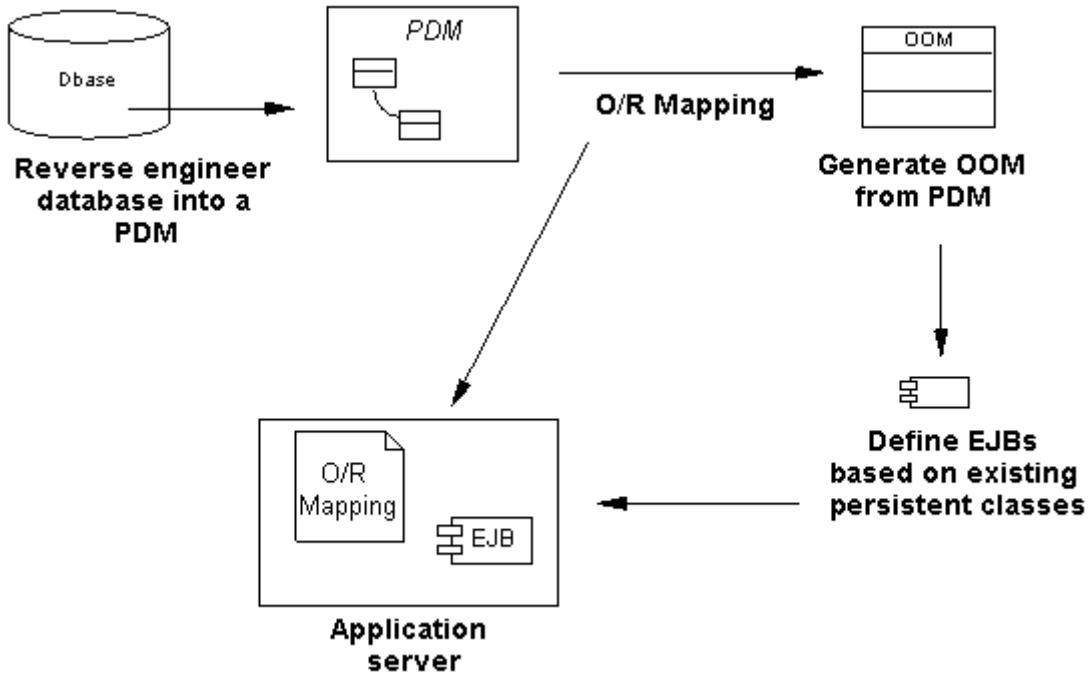
- Define persistent objects in the class diagram, and define EJBs in the component diagram
- Generate the Physical Data Model, with creation of an O/R Mapping during generation
- Create the database

- Generate EJB source code and deployment descriptor for deployment in the application server



You want to generate EJB source code for an existing database. If the database already exists, you may want to wrap the existing database into EJBs and use EJBs to access the database. The development process is the following:

- Reverse engineer the PDM into an OOM, and retrieve the O/R mapping generated during reverse
- Define EJBs in the OOM based on persistent classes
- Generate the EJB source code and deployment descriptor for deployment in the application server



You can also use generation of EJB source code when managing persistence of EJBs with an existing database. For example, it may be necessary to manage the persistence of an already defined EJB with an existing database. Since you cannot change the definition of EJB nor the database schema, you need a manual object to relational mapping in this case.

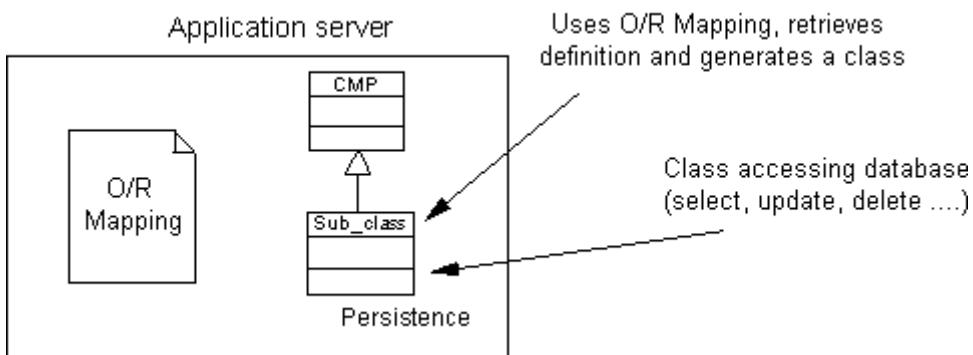
For more information on object mapping, see [Object/Relational \(O/R\) Mapping \[page 430\]](#).

### 2.1.6.8.2 Understanding EJB Source and Persistence

You generate persistence management methods based on the object language.

Depending if the EJB is of a CMP or BMP type, the deployment descriptor file is displayed different:

- A CMP involves the application server. It includes the EJB and the O/R mapping descriptor (.XML). The server retrieves both EJB and O/R mapping descriptor to generate the code  
If the application server does not support an O/R Mapping descriptor, the mapping must be done manually. If the O/R mapping descriptor of your application server is not supported yet, you can create your own by creating a new extension file. For detailed information about working with extension files, see *Customizing and Extending PowerDesigner > Extension Files* .
- A BMP involves a manual process. It includes the EJB source, without any O/R mapping descriptor (O/R mapping is not necessary). The BMP developer should implement the persistence management him/herself by implementing the ejbStore(), and ejbLoad() methods, the application server only supports its functions. An implementation class inherits from the BMP Bean class, handles persistence data and communicates with the database
- You can also define an EJB as CMP, then generate it as BMP when generating the code. The code generator generates an implementation class (sub-class) for the Bean class that contains its methods, and uses an O/R mapping and a persistent template to implement the persistence



For more information on defining O/R mapping, see [Object/Relational \(O/R\) Mapping \[page 430\]](#).

You can use different methods to generate an EJB CMP into an EJB BMP. You can either copy the Java object language delivered in PowerDesigner as a reference to a new object language, and describe how to generate implementation classes of the EJB CMP in your own object language, or you can create an extension file that includes these implementation classes.

You could also write a VB script to convert the EJB CMP into an EJB BMP. To do this, you must generate the EJB as CMP, then launch the VB script that will go through all objects of the model and generate an implementation class for each identified class.

### 2.1.6.8.3 Generating EJB Source Code and the Deployment Descriptor

When generating an EJB, classes and interfaces of the EJB are directly selected. The generation retrieves the classes used by the component, as well as the interfaces associated with the classes.

#### Procedure

1. Select **Language > Generate Java Code** to open the Generation dialog.
2. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see [Checking an OOM \[page 251\]](#)).
3. [optional] Click the *Selection* tab and select the objects that you want to generate. By default, all objects are generated.
4. [optional] Click the *Options* tab and select any appropriate generation options (see [Generating Java Files \[page 352\]](#)).
5. [optional] Click the *Tasks* tab and select any appropriate task to perform during generation (see [Generating Java Files \[page 352\]](#)).
6. Click **OK** to begin generation.

A progress box is displayed, followed by a Result list. You can use the **Edit** button in the Result list to edit the generated files individually.

7. Click *Close*.

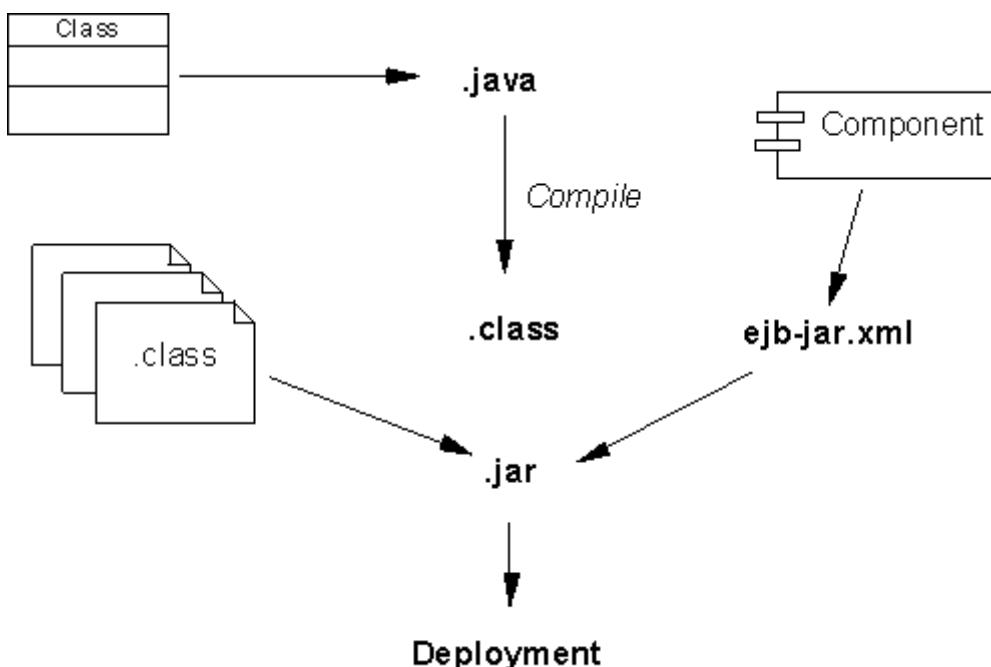
The ejb-jar.xml deployment descriptor is created in the META-INF directory and all files are generated in the generation directory.

## 2.1.6.9 Generating JARs

In order to package the EJB, the bean classes, interfaces and the deployment descriptor are placed into a .JAR file. This process is common to all EJB components.

You can generate .JAR files from the Tasks page of the Generation dialog box ( *Language* > *Generate Java Code*).

For example, one of the task allows you to compile .JAVA files using a compiler, to create a .JAR file including the compiled Java classes, and to complete the .JAR file with the deployment descriptor and icons.



### Caution

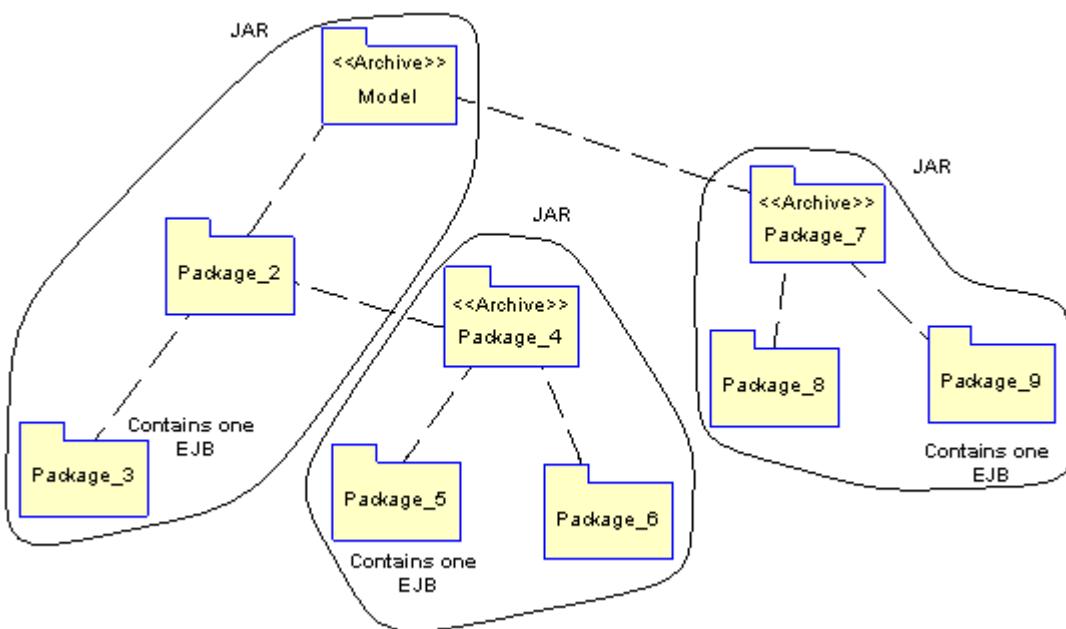
You must set the values of the commands used during generation from the Variables section of the General Options dialog box in order to enable them. For example, you can set the javac.exe and jar.exe executables at this location.

For more information on how to set these variables, see *Core Features Guide > Modeling with PowerDesigner > Customizing Your Modeling Environment > General Options > Environment Variables*.

There is no constraint over the generation of one JAR per package. Only packages with the <<archive>> stereotype will generate a JAR when they (or one of their descendant package not stereotyped <<archive>>) contain one EJB.

The newly created archive contains the package and all of its non-stereotyped descendants. The root package (that is the model) is always considered as being stereotyped <<archive>>.

For example, if a model contains several EJB components in different sub-packages but that none of these packages is stereotyped <<archive>>, a single JAR is created encompassing all packages.



## 2.1.6.10 Reverse Engineering EJB Components

PowerDesigner can reverse engineer EJB components located in .JAR files.

### Procedure

1. Select **Language > Reverse Engineer Java** to open the Reverse Engineer Java dialog.
2. On the **Selection** tab, select **Archives** from the **Reverse** list.
3. Click the **Add** button, navigate to and select the objects that you want to reverse, and then click **Open** to add them to the list.
4. Click the **Options** tab and select the **Reverse Engineer Deployment Descriptor** check box.
5. Click **OK**.

A progress box is displayed. If the model in which you are reverse engineering already contains data, the Merge Models dialog box is displayed.

For more information on merging models, see *Core Features Guide > Modeling with PowerDesigner > Comparing and Merging Models*.

6. Click **OK**.

The **Reverse** page of the Output window displays the changes that occurred during reverse and the diagram window displays the updated model.

## 2.1.7 Enterprise Java Beans (EJBs) V3

The specification for EJB 3.0 attempts to simplify the complexity of the EJB 2.1 architecture by decreasing the number of programming artefacts that developers need to provide, minimizing the number of required callback methods and reducing the complexity of the entity bean programming model and the O/R mapping model.

The two most significant changes in the proposed EJB 3.0 specification are:

- An annotation-based EJB programming model - all kinds of enterprise beans are just plain old Java objects (POJOs) with appropriate annotations. A configuration-by-exception approach uses intuitive defaults to infer most common parameters. Annotations are used to define the bean's business interface, O/R mappings, resource references, and other information previously defined through deployment descriptors or interfaces. Deployment descriptors are not required; the home interface is gone, and it is no longer necessary to implement a business interface (the container can generate it for you).  
For example, you declare a stateless session bean by using the `@Stateless` annotation. For stateful beans, the `@Remove` annotation is marked on a particular method to indicate that the bean instance should be removed after a call to the marked method completes.
- The new persistence model for entity beans - The new entity beans are also just POJOs with a few annotations and are not persistent entities by birth. An entity instance becomes persistent once it is associated with an EntityManager and becomes part of a persistence context, which is loosely synonymous with a transaction context and implicitly coexists with a transaction's scope.  
The entity relationships and O/R mapping is defined through annotations, using the open source Hibernate framework (see [Generating Persistent Objects for Java and JSF Pages \[page 463\]](#)).

There are also several side effects to these proposals, such as a new client-programming model, use of business interfaces, and an entity bean life cycle.

### **i** Note

The EJB 2.1 programming model (with deployment descriptors and home/remote interfaces) is still valid and supported by PowerDesigner. The new simplified model, which is only available with Java 5.0, does not entirely replace the EJB 2.1 model.

## 2.1.7.1 Creating an EJB 3.0 with the Enterprise JavaBean Wizard

To create an EJB3, launch the Enterprise JavaBean Wizard from a class diagram.

### Context

The following types of EJB3 beans are available:

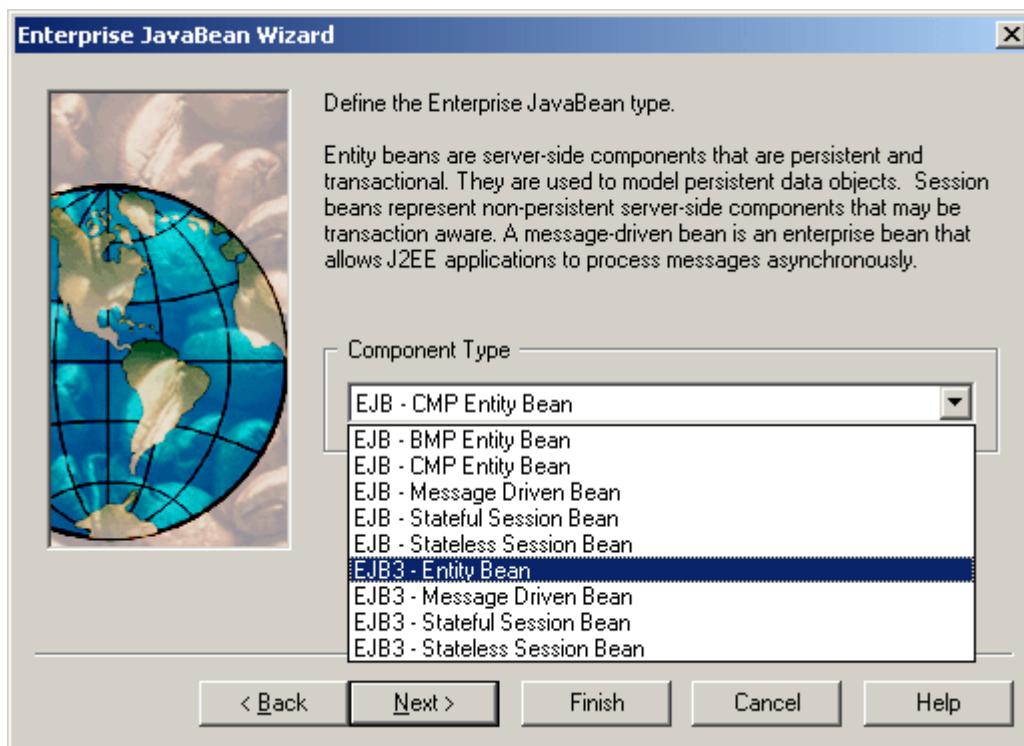
- Entity Bean – generated with an @Entity annotation
- Message Driven Bean – generated with a @MessageDriven annotation
- Stateful Session Bean – generated with an @Stateful annotation
- Stateless Session Bean – generated with an @Stateless annotation

### Procedure

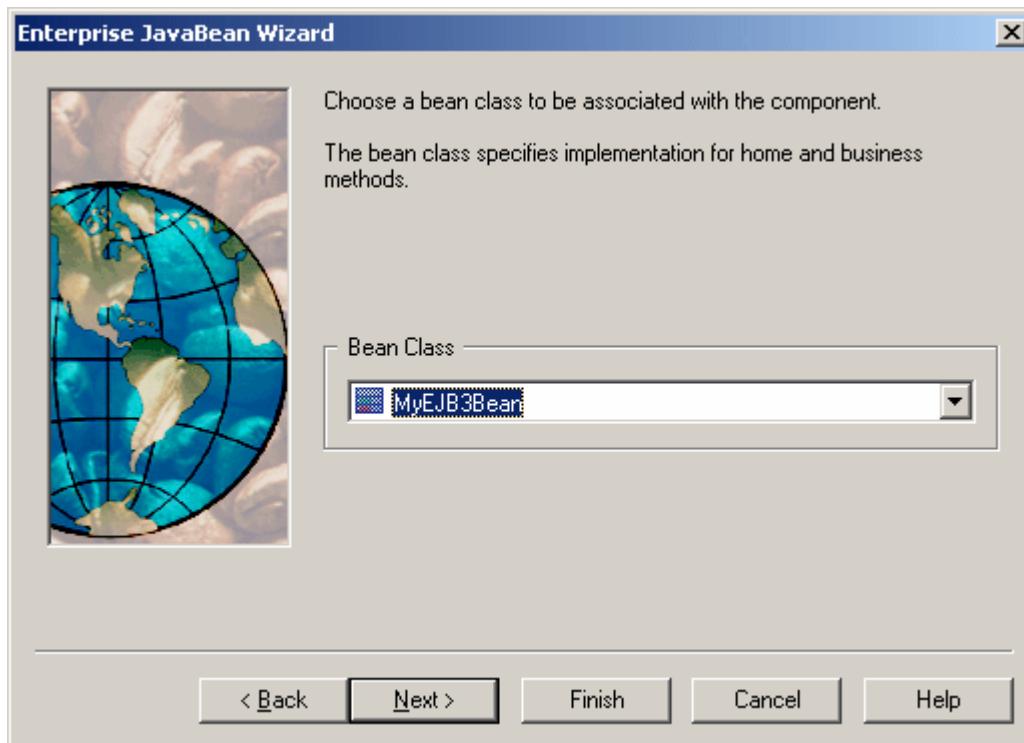
1. If you have already created a class to serve as the BeanClass, then right click it and select *Create Enterprise JavaBean* from the contextual menu. Otherwise, to create an EJB 3.0 along with a new BeanClass, Select  *Tools* > *Create Enterprise JavaBean*. In either case, the Enterprise JavaBean Wizard opens:



2. Specify a name for the EJB, and then click [Next](#) to go to the next page:



3. Choose a type of EJB3, and then click [Next](#) to go to the next page:



4. Choose a Bean Class. If you have not selected a class before launching the wizard, a default class with the same name as the component will be suggested. Otherwise the original class will be selected. Then click [Next](#) to go to the next page:



5. Choose the desired level of transaction support, and then click [Next](#) to go to the next page:



6. Select the appropriate checkboxes if you want to create diagrams for the component and/or the component classifiers, and then click *Finish* to instruct PowerDesigner to create them.

### 2.1.7.2 EJB 3.0 BeanClass Properties

A BeanClass is the primary class contained within an EJB 3.0 component. EJB 3.0 BeanClass property sheets contain all the standard class tabs along with the EJB3 tab.

The EJB3 tab contains the following properties:

| Property               | Description   |
|------------------------|---|
| Transaction Management | <p>Specifies the method of transaction management for a Session Bean or a Message Driven Bean. You can choose between:</p> <ul style="list-style-type: none"> <li>• Bean</li> <li>• Container</li> </ul> <p>Generated as a @TransactionManagement annotation.</p> |

| Property                     | Description   |
|------------------------------|---|
| Transaction Attribute Type   | <p>Transaction Attribute Type for the Bean Class</p> <p>Specifies the transaction attribute type for a Session Bean or a Message Driven Bean. You can choose between:</p> <ul style="list-style-type: none"> <li>• Not Supported</li> <li>• Supports</li> <li>• Required</li> <li>• Requires New</li> <li>• Mandatory</li> <li>• Never</li> </ul> <p>Generated as a @TransactionAttribute annotation.</p> |
| Exclude Default Interceptors | <p>Specifies that the invocation of default interceptor methods is excluded.</p> <p>Generated as a @ExcludeDefaultInterceptors annotation.</p>  |
| Exclude Superclass Listeners | <p>Specifies that the invocation of superclass listener methods is excluded.</p> <p>Generated as a @ExcludeSuperclassListeners annotation.</p>  |
| Mapped Name                  | <p>Specifies a product specific name.</p> <p>Generated as a @MappedName annotation.</p>   |
| Run-As                       | <p>Specifies the bean's run-as property (security role).</p> <p>Generated as a @RunAs annotation.</p>   |
| Declare Roles                | <p>Specifies references to security roles.</p> <p>Generated as a @DeclareRoles annotation.</p>  |
| Roles Allowed                | <p>Specifies the roles allowed for all bean methods.</p> <p>Generated as a @RolesAllowed annotation.</p>  |
| Permit All                   | <p>Specifies that all roles are allowed for all bean business methods.</p> <p>Generated as a @PermitAll annotation.</p>   |

## 2.1.7.3 EJB 3.0 Component Properties

EJB 3.0 component property sheets contain all the standard component tabs along with the *EJB* tab.

The *EJB* tab contains the following properties:

| Property              | Description   |
|-----------------------|---|
| Bean class            | Specifies the associated bean class.  |
| Remote home interface | [session beans only] Specifies an optional remote home interface (for earlier EJB clients). |
| Local home interface  | [session beans only] Specifies the local home interface (for earlier EJB clients).          |

### 2.1.7.3.1 Adding Further Interfaces and Classes to the EJB

In addition to the bean class and remote and home interfaces defined on the EJB tab, you can link supplementary classes and interfaces to the EJB3.

#### Context

You can link the following supplementary classes and interfaces to the EJB3:

- An optional collection of remote interfaces with <<EJBRemote>> stereotypes. Generated as a @Remote annotation.
- An optional collection of local interfaces with <<EJBLocal>> stereotypes. Generated as a @Local annotation.
- An optional collection of interceptor classes with <<EJBInterceptor>> stereotypes. Generated as an @Interceptors annotation.
- An optional collection of entity listener classes with <<EJBEntityListener>> stereotypes. Generated as an @EntityListeners annotation.

You add these interfaces and classes to the EJB3 component via the Interfaces and Classes tabs. For example, you can add an <<EJBInterceptor>> Interface to an EJB3:

#### Procedure

1. Open the property sheet of the EJB3 and click the *Interfaces* tab.
2. Click the *Create a New Object* tool to create a new interface and open its property sheet.
3. On the *General* tab, select <<EJBInterceptor>> from the list of stereotypes.
4. Complete the remaining properties as required and then click *OK* to return to the EJB3 property sheet.

## 2.1.7.4 EJB 3.0 Operation Properties

EJB operation 3.0 property sheets contain all the standard operation tabs along with the *EJB3* tab.

The *EJB3* tab contains the following properties:

| Property                     | Description   |
|------------------------------|---|
| Initialize Method            | Specifies an initialize method.<br>Generated as a @Init annotation.   |
| Remove Method                | Specifies an remove method.<br>Generated as a @Remove annotation.   |
| Post-Construct               | Specifies a post construct method.<br>Generated as a @PostConstruct annotation.   |
| Post-Activate                | Specifies a post activate method.<br>Generated as a @PostActivate annotation.   |
| Pre-Passivate                | Specifies a pre passivate method.<br>Generated as a @PrePassivate annotation.   |
| Pre-Destroy                  | Specifies a pre destroy method.<br>Generated as a @PreDestroy annotation.   |
| Interceptor Method           | Specifies an interceptor method.<br>Generated as a @AroundInvoke annotation.  |
| Timeout Method               | Specifies a timeout method.<br>Generated as a @Timeout annotation.  |
| Exclude Default Interceptors | Excludes invocation of default interceptor for the method.<br>Generated as a @ExcludeDefaultInterceptors annotation.    |
| Exclude Class Interceptors   | Excludes invocation of class-level interceptors for the method.<br>Generated as a @ExcludeClassInterceptors annotation. |
| Transaction Attribute Type   | Specifies a Transaction Attribute Type for the method.<br>Generated as a @TransactionAttribute annotation.              |
| Permit All Roles             | Specifies that all roles are permitted for the method.<br>Generated as a @PermitAll annotation.                         |

| Property       | Description   |
|----------------|---|
| Deny All Roles | Specifies that method may not be invoked by any security role.<br>Generated as a @DenyAll annotation. |
| Roles Allowed  | Specifies the roles allowed for the method.<br>Generated as a @RolesAllowed annotation.               |

## 2.1.8 Java Servlets

Servlets are programs that help in building applications that generate dynamic Web pages (HTML, XML). Servlets are a Java-equivalent of the CGI scripts and can be thought of as a server-side counterpart to the client-side Java applets.

Servlets are Java classes that implement a specific interface and produce HTML in response to GET/POST requests.

In an OOM, a servlet is represented as a component, it is linked to a servlet class that implements the required interface and provides the servlet implementation.

When you set the type of the component to Servlet, the appropriate servlet class is automatically created, or attached if it already exists. The servlet class is initialized so that operations are automatically added.

### 2.1.8.1 Servlet Page of the Component

When you set the type of the component to Servlet, the [Servlet](#) page is automatically displayed in the component property sheet.

The [Servlet](#) page in the component property sheet includes the following properties:

| Property      | Description  |
|---------------|--|
| Servlet class | Class that implements the required interface. You can click the <a href="#">Properties</a> tool beside this box to display the property sheet of the class, or click the <a href="#">Create</a> tool to create a class                                   |
| Servlet type  | HttpServlet supports the http protocol, it is the most commonly used. GenericServlet extends the servlet generic class. User-defined implies some customization as it does not implement anything. The methods vary if you change the servlet type value |

## 2.1.8.2 Defining Servlet Classes

Servlet classes are identified using the <<ServletClass>> stereotype.

The servlet class name is synchronized with the component name following the convention specified in the Value box of the Settings/Namings/ServletClassName entry in the Java object language.

## 2.1.8.3 Creating a Servlet with the Wizard

You can create a servlet with the wizard that will guide you through the creation of the component. The wizard is invoked from a class diagram. It is only available if the language is Java.

### Context

You can either create a servlet without selecting any class, or select a class beforehand and start the wizard from the contextual menu of the class.

You can also create several servlets of the same type by selecting several classes at the same time. The wizard will automatically create one servlet per class. The classes you have selected in the class diagram become servlet classes. They are renamed to fit the naming conventions standard, and they are linked to the new servlet component.

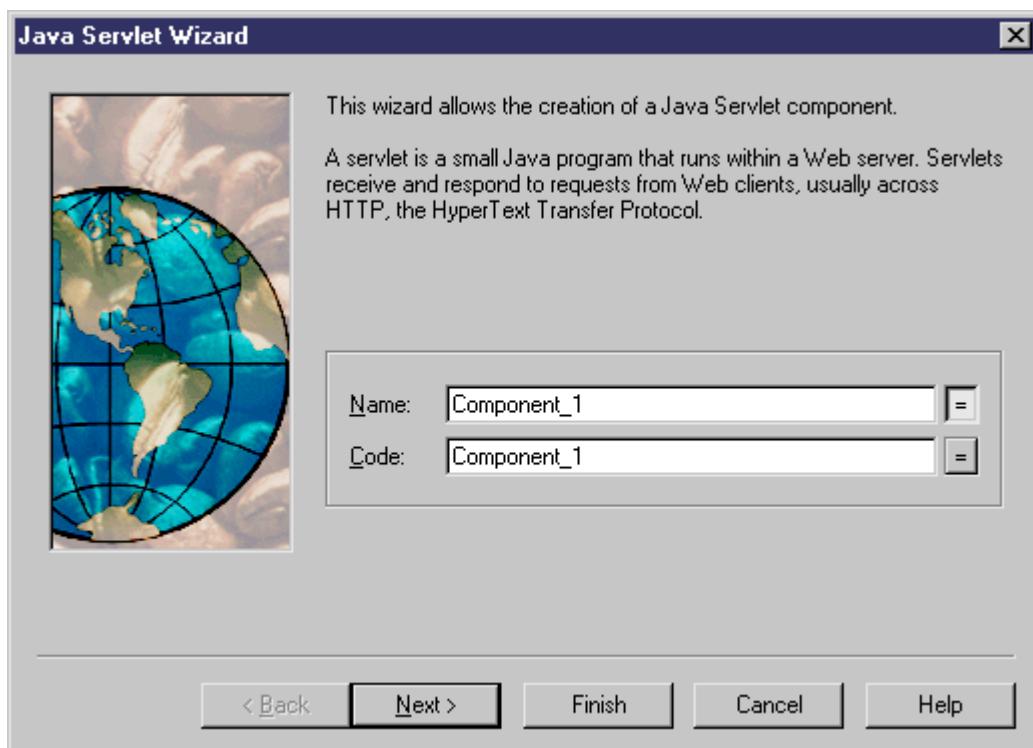
The wizard for creation of a servlet lets you define the following parameters:

| Wizard page                                    | Description  |
|--|--|
| Name   | Name of the servlet component  |
| Code   | Code of the servlet component  |
| Servlet type                                   | You can select the following types: HttpServlet that supports the http protocol (most commonly used), GenericServlet that extends the servlet generic class, or user-defined that implies some customization as it does not implement anything                 |
| Servlet class                                  | Class that provides the Servlet implementation.  |
| Create symbol                                  | Creates a component symbol in the diagram specified beside the <i>Create symbol In</i> check box. If a component diagram already exists, you can select one from the list. You can also display the diagram properties by selecting the <i>Properties</i> tool |
| Create Class Diagram for component classifiers | Creates a class diagram with a symbol for each class. If you have selected classes before starting the wizard, they are used to create the component. This option allows you to display these classes in a diagram   |

## Procedure

1. Select *Tools* *Create Servlet* from a class diagram.

The Java Servlet Wizard dialog box is displayed.



### Note

If you have selected classes before starting the wizard, some of the following steps are omitted because the different names are created by default according to the names of the selected classes.

2. Select a name and code for the servlet component and click *Next*.
3. Select a servlet type and click *Next*.
4. Select the servlet class name and click *Next*.
5. At the end of the wizard, you have to define the creation of symbols and diagrams.

## Results

When you have finished using the wizard, the following actions are executed:

- A servlet component is created
- The servlet class is created and visible in the Browser. It is named after the original class if you have selected a class before starting the wizard

- If you have not selected a class beforehand, it is prefixed after the original default component name to preserve coherence
- Any diagrams associated with the component are created or updated

## 2.1.8.4 Understanding Servlet Initialization and Synchronization

When creating a servlet, the initialization process instantiates the servlet class together with its methods.

The role of the synchronization is to maintain the coherence of the whole model whenever a change is applied. It occurs progressively between classes already attached to a component. PowerDesigner successively performs several actions to complete synchronization as the model is modified.

The initialization and synchronization of the servlet class works in a similar way as with Message Driven bean classes:

- When the servlet class is attached to a servlet component, implementation methods for operations defined in the javax.servlet.Servlet interface are added by default. This interface is the base interface for all servlets, it may be included in the code depending on the servlet type selected. For HttpServlet and GenericServlet, the servlet class being directly derived from a class that implements it, it does not need to reimplement the interface. On the contrary, for user-defined servlets, this interface is implemented. You can see it from the Preview page of the servlet class.
- Implementation methods are removed when the class is detached if their body has not been altered
- The actual set of predefined methods vary depending on the servlet type

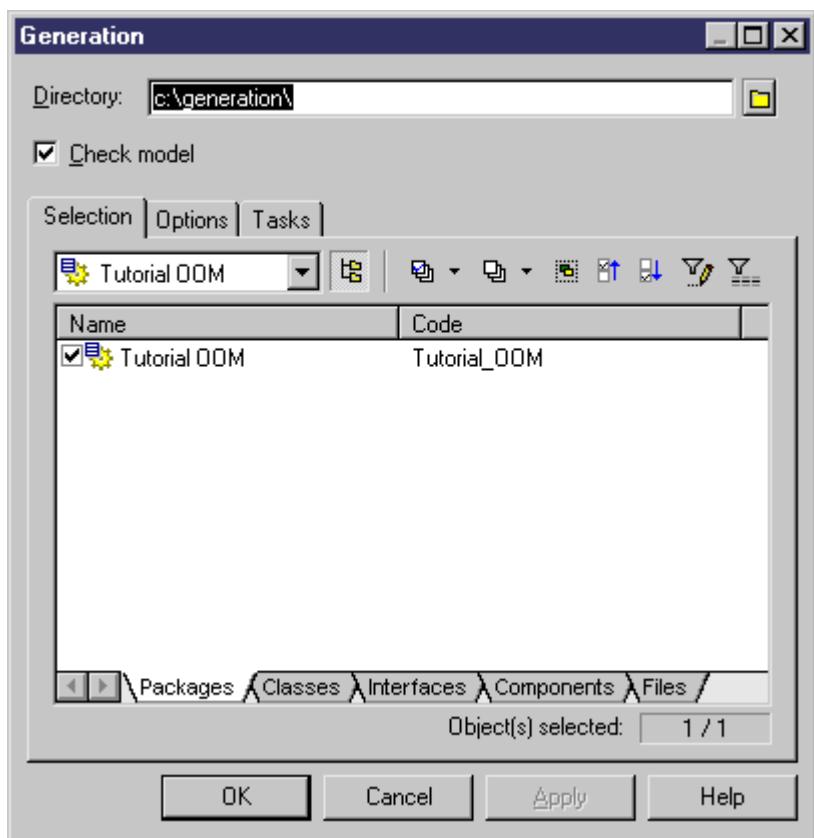
You can use the Check Model feature at any time to validate your model and complement the synchronization by selecting  [Tools > Check Model](#).

## 2.1.8.5 Generating Servlets

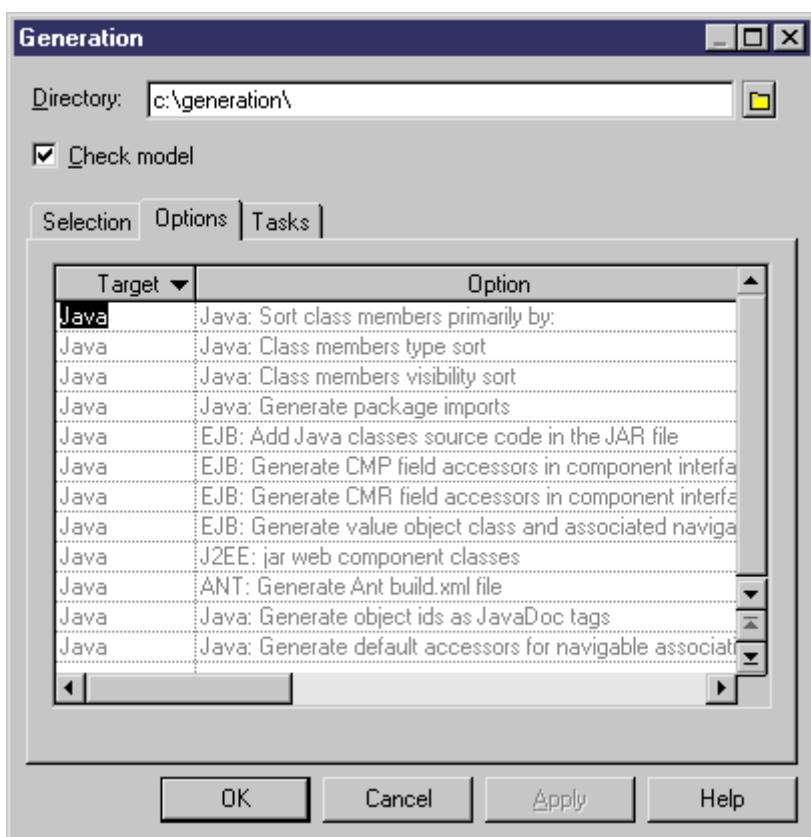
The generation process retrieves all classes used by the servlet components to generate the servlets.

### Procedure

1. Select  [Language > Generate Java Code](#) to open the Generation dialog.
2. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see [Checking an OOM \[page 251\]](#)).
3. Click the [Selection](#) tab, then select the objects you need in the different tabbed pages.

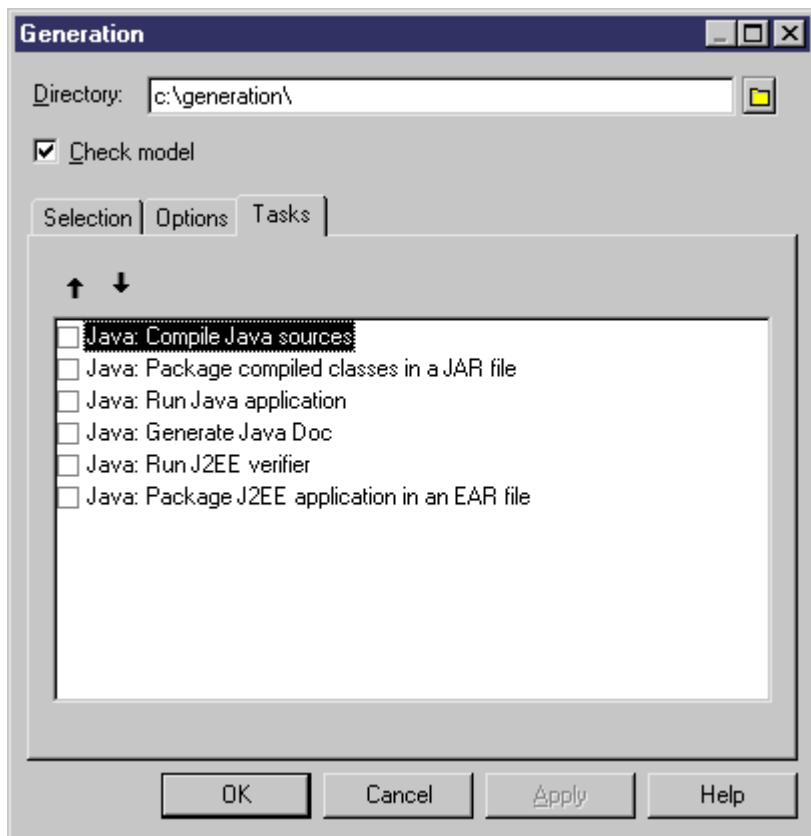


4. Click *Apply*.
5. Click the *Options* tab, specify your generation options.



For more information on the generation options, see [Generating Java Files \[page 352\]](#).

6. Click *Apply*.
7. Click the *Tasks* tab, then select the commands you want to perform during generation.



For more information on the generation tasks, see [Generating Java Files \[page 352\]](#).

You must beforehand set your environment variables from ► [Tools](#) ► [General Options](#) ► [Variables](#) in order to activate them in this page.

For more information on how to set these variables, see *Core Features Guide > Modeling with PowerDesigner > Customizing Your Modeling Environment > General Options > Environment Variables*.

8. Click [OK](#) to close the dialog.

A progress box is displayed, followed by a Result list. You can use the [Edit](#) button in the Result list to edit the generated files individually.

9. Click [OK](#) to begin generation.

The web.XML file is created in the WEB-INF directory and all files are generated in the generation directory.

## 2.1.8.5.1 Generating Servlet Web Deployment Descriptor

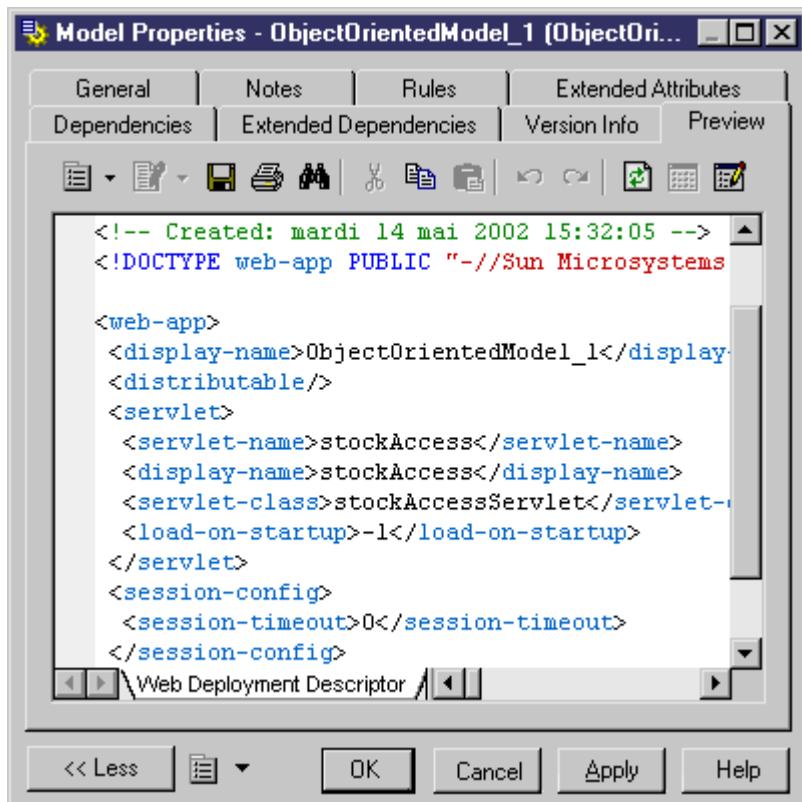
The Web deployment descriptor is an XML file, called web.XML. It is generated in the WEB-INF directory and is independent from the application server.

For more information on generating the Web deployment descriptor, see [Generating Servlets \[page 336\]](#).

The Web application deployment descriptor is generated per package. A WAR command available in the Tasks page of the Generation dialog box allows you to build a Web Archive that contains the Web deployment descriptor,

in addition to all classes and interfaces referenced by servlet components. At the model level, an EAR command is also provided to group all WAR and JAR files generated for a given model inside a single enterprise archive. The EAR contains an additional deployment descriptor generated per model that is called application.XML.

The Web deployment descriptor contains several servlets to be deployed, it is available from the Preview page of the package, or the model property sheet.



## 2.1.8.6 Generating WARs

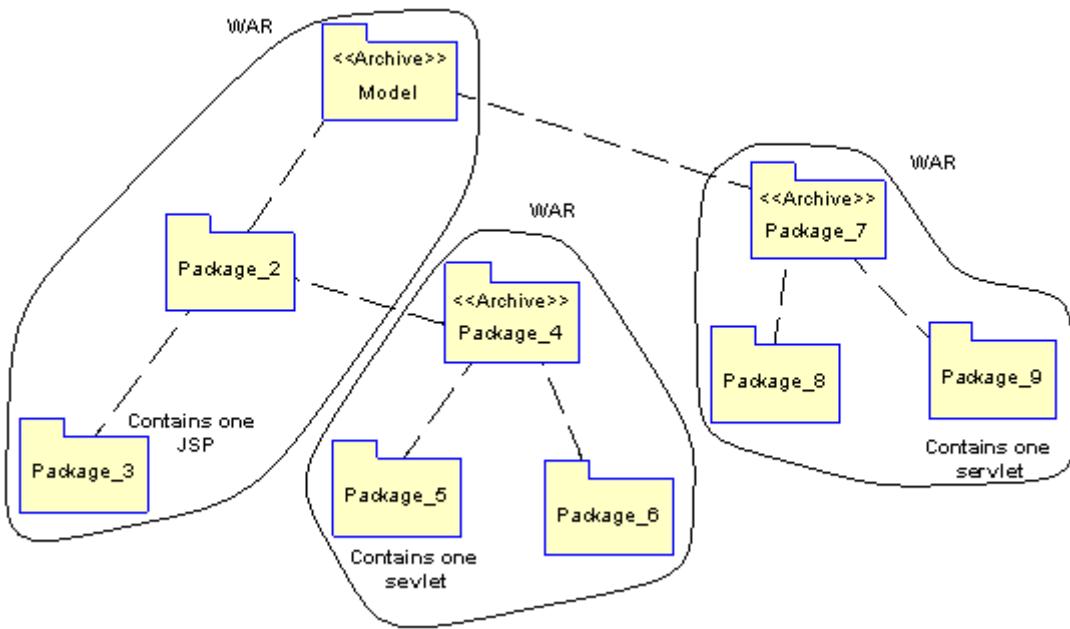
You can package servlets and JSPs into a .WAR file.

You can generate .WAR files from the Tasks page of the Generation dialog box (▶ [Language](#) ▶ [Generate Java Code](#) ▶).

There is no constraint over the generation of one WAR per package. Only packages with the <<archive>> stereotype will generate a WAR when they (or one of their descendant package not stereotyped <<archive>>) contain one servlet, or one JSP.

The newly created archive contains the package and all of its non-stereotyped descendants. The root package (that is the model) is always considered as being stereotyped <<archive>>.

For example, if a model contains several Web components in different sub-packages but that none of these packages is stereotyped <<archive>>, a single WAR is created encompassing all packages.



For more information on generating WARs, see [Generating Java Files \[page 352\]](#).

## 2.1.8.7 Reverse Engineering Servlets

You can reverse engineer servlet code and deployment descriptor into an OOM. The reverse engineering feature reverses the Java class as a servlet class, it reverses the servlet as a component, and associates the servlet class with this component. The reverse engineering feature also reverses the deployment descriptor and all the files inside the WAR.

### Context

You start reverse engineering servlet from [Language](#) [Reverse Engineer Java](#).

Select one of the following Java formats from the Selection page, and select the Reverse Engineer Deployment Descriptor check box from the Options page:

- Java directories
- Class directories
- Archive (.JAR file)

For more information on the Java formats, see [Reverse Engineering Java Code \[page 355\]](#).

## Procedure

1. Select [Language](#) [Reverse Engineer Java](#).

The Reverse Java dialog box is displayed.

2. Select Archive from the Reverse list in the [Selection](#) page.

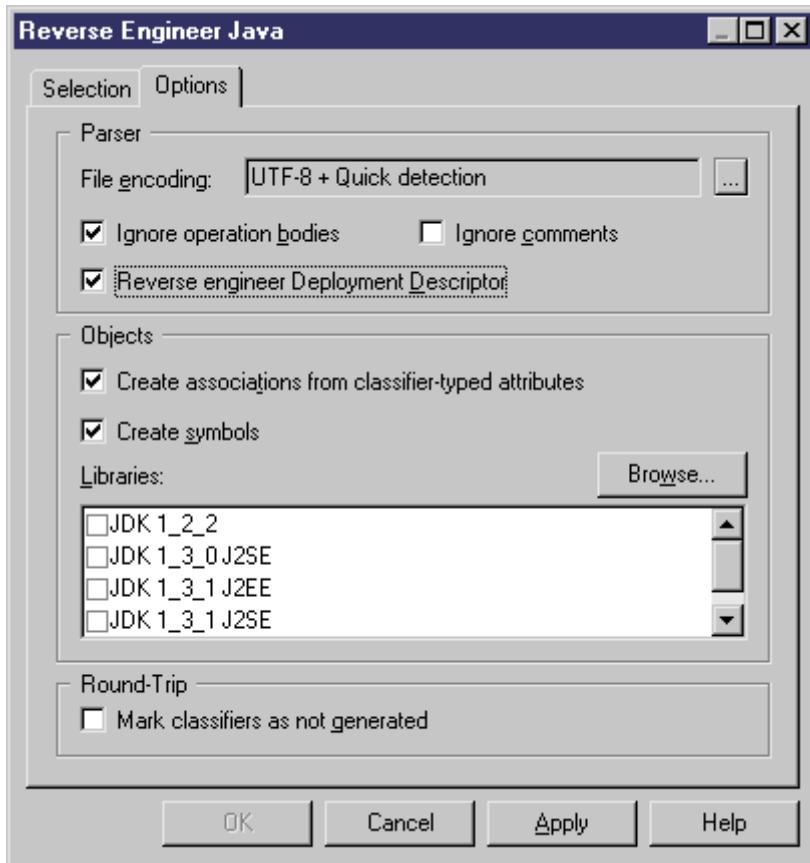
3. Click the [Add](#) button.

A standard Open dialog box is displayed.

4. Select the items you want to reverse and click [Open](#).

The Reverse Java dialog box displays the items you selected.

5. Click the [Options](#) tab, then select the [Reverse Engineer Deployment Descriptor](#) check box.



6. Click [OK](#).

A progress box is displayed. If the model in which you are reverse engineering already contains data, the Merge Models dialog box is displayed.

For more information on merging models, see [Core Features Guide > Modeling with PowerDesigner > Comparing and Merging Models](#).

7. Click [OK](#).

The Reverse page of the Output window displays the changes that occurred during reverse and the diagram window displays the updated model.

## 2.1.9 Java Server Pages (JSPs)

Java Server Page (JSP) is an HTML Web page that contains additional bits of code that execute application logic to generate dynamic content.

In an OOM, a JSP is represented as a file object and is linked to a component - of type JSP -. The Java Server Page (JSP) component type allows you to identify this component. Components of this type are linked to a single file object that defines the page.

When you set the type of the component to JSP, the appropriate JSP file object is automatically created, or attached if it already exists. You can see the JSP file object from the Files page in the component property sheet.

### 2.1.9.1 JSP Page of the Component

When you set the type of the component to JSP, the JSP page is automatically displayed in the component property sheet.

The JSP page in the component property sheet includes the following properties:

| Property         | Description   |
|------------------|---|
| JSP file         | File object that defines the page. You can click the Properties tool beside this box to display the property sheet of the file object, or click the Create tool to create a file object |
| Default template | Extended attribute that allows you to select a template for generation. Its content can be user defined or delivered by default   |

To modify the default content, edit the current object language from [Language](#) [Edit Current Object Language](#) and modify the following item: Profile/FileObject/Criteria/JSP/Templates/DefaultContent<%is(DefaultTemplate)%>. Then create the templates and rename them as DefaultContent<%is(<name>)%> where <name> stands for the corresponding DefaultContent template name.

To define additional DefaultContent templates for JSPs, you have to modify the JSPTemplate extended attribute type from Profile/Share/Extended Attribute Types and add new values corresponding to the new templates respective names.

For more information on the default template property, see the definition of TemplateContent in [Creating a JSP with the Wizard \[page 344\]](#).

### 2.1.9.2 Defining File Objects for JSPs

The file object content for JSPs is based on a special template called DefaultContent defined with respect to the FileObject metaclass. It is located in the Profile/FileObject/Criteria/JSP/Templates category of the Java object

language. This link to the template exists as a basis, therefore if you edit the file object, the link to the template is lost - the mechanism is similar to that of operation default bodies.

For more information on the Criteria category, see *Customizing and Extending PowerDesigner > Extension Files > Criteria (Profile)*.

Java Server Page files are identified using the JSPFile stereotype. The server page name is synchronized with the JSP component name following the convention specified in the Value box of the Settings/Namings/JSPFileName entry of the Java object language.

You can right-click a file object, and select  *Open With*  from the contextual menu to display the content of the file object.

### 2.1.9.3 Creating a JSP with the Wizard

You can create a JSP with the wizard that will guide you through the creation of the component. The wizard is invoked from a class diagram. It is only available if the language is Java.

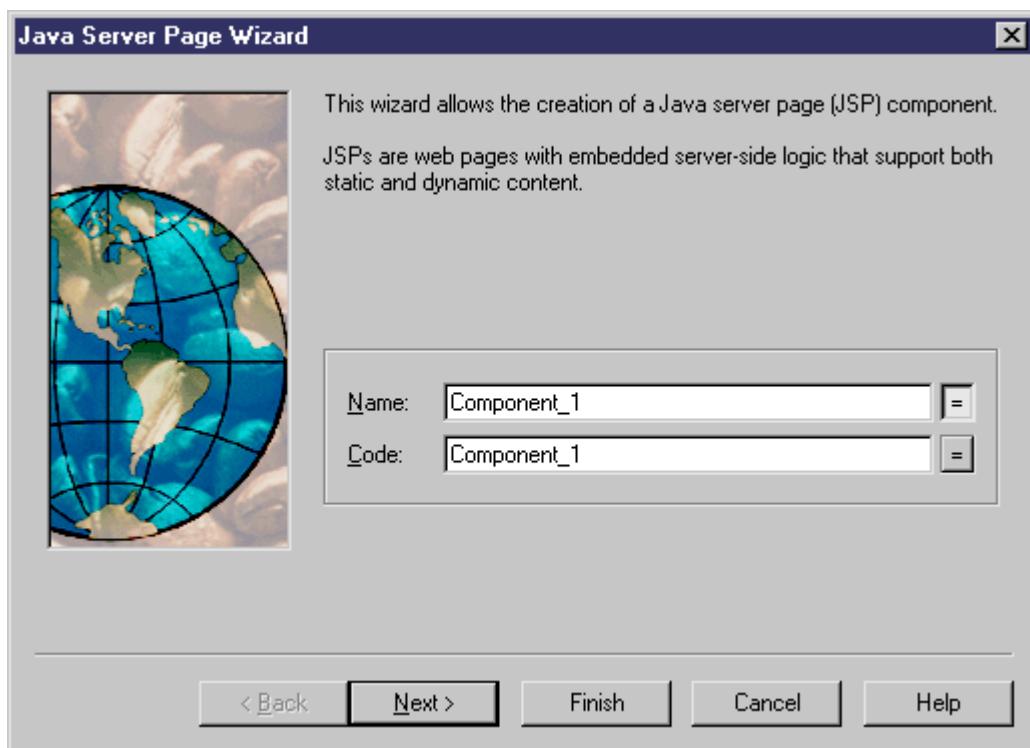
#### Context

You can either create a JSP without selecting any file object, or select a file object beforehand and start the wizard from the *Tools* menu.

You can also create several JSP of the same type by selecting several file objects at the same time. The wizard will automatically create one JSP per file object: the file objects you have selected in the class diagram become .JSP files.

#### Procedure

1. Select  *Tools*  from a class diagram.



2. Select a name and code for the JSP component and click *Next*.
3. Select the default template of the JSP file object. **TemplateContent** is an extended attribute located in the *Profile/Component/Criteria/J2EE-JSP* category of the Java object language. If you do not modify the content of the file object, the default content remains. All templates are available in the *Profile/FileObject/Criteria/JSP/templates* category of the Java object language.
4. Click *Next* to go to the final page of the wizard, where you can select *Create symbol* to create a component symbol in the specified diagram.

## Results

When you have finished using the wizard, the following actions are executed:

- A JSP component and a file object with an extension .JSP are created and visible in the Browser. The file object is named after the original default component name to preserve coherence
- If you open the property sheet of the file object, you can see that the *Artifact* property is selected. For more information on artifact file objects, see [File Object Properties \[page 208\]](#).
- You can edit the file object directly in the internal editor of PowerDesigner, if its extension corresponds to an extension defined in the *Editors* page of the General Options dialog box, and if the `<internal>` keyword is defined in the *Editor Name* and *Editor Command* columns for this extension

## 2.1.9.4 Generating JSPs

The generation process generates only file objects with the Artifact property selected.

### 2.1.9.4.1 Generating JSP Web Deployment Descriptor

The Web deployment descriptor is an XML file, called web.XML. It is generated in the WEB-INF directory and is independent from the application server.

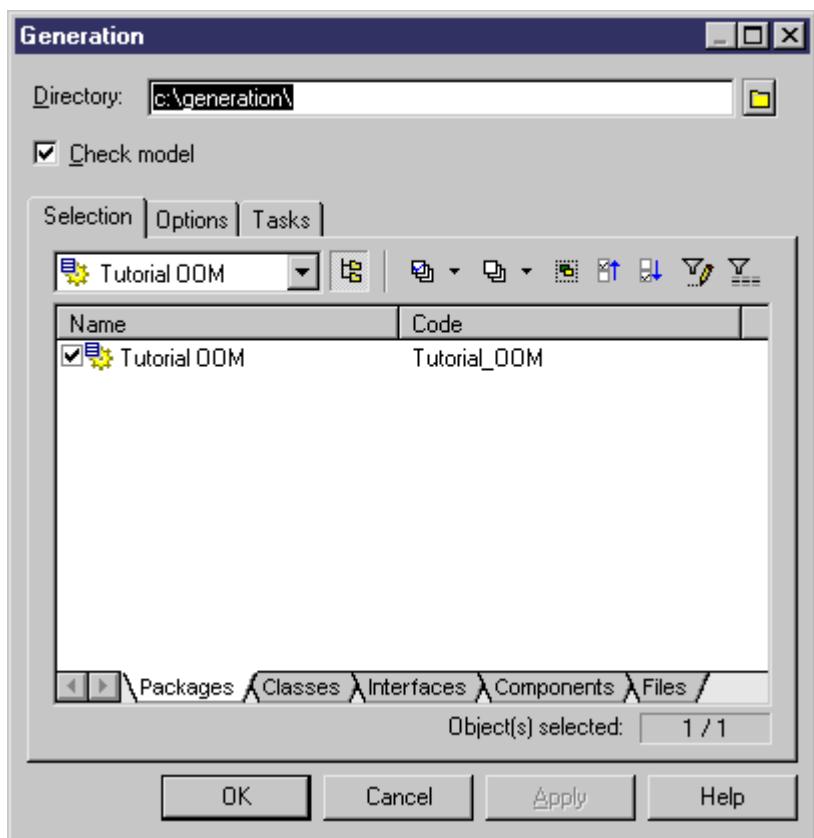
#### Context

The Web application deployment descriptor is generated per package. A WAR command available in the Tasks page of the Generation dialog box allows you to build a Web Archive that contains the Web deployment descriptor, in addition to all classes and file objects referenced by JSP components. At the model level, an EAR command is also provided to group all WAR and JAR files generated for a given model inside a single enterprise archive. The EAR contains an additional deployment descriptor generated per model that is called application.XML.

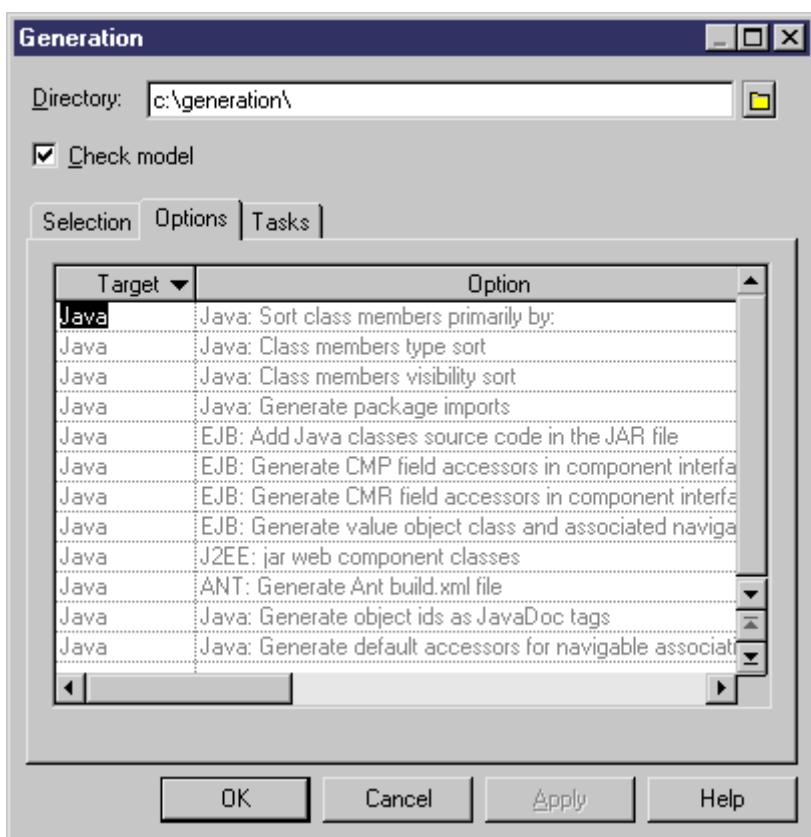
The Web deployment descriptor is available from the Preview page of the package, or the model property sheet.

#### Procedure

1. Select  [Language](#)  [Generate Java Code](#)  to open the Generation dialog.
2. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see [Checking an OOM \[page 251\]](#)).
3. Click the [Selection](#) tab, then select the objects you need in the different tabbed pages.

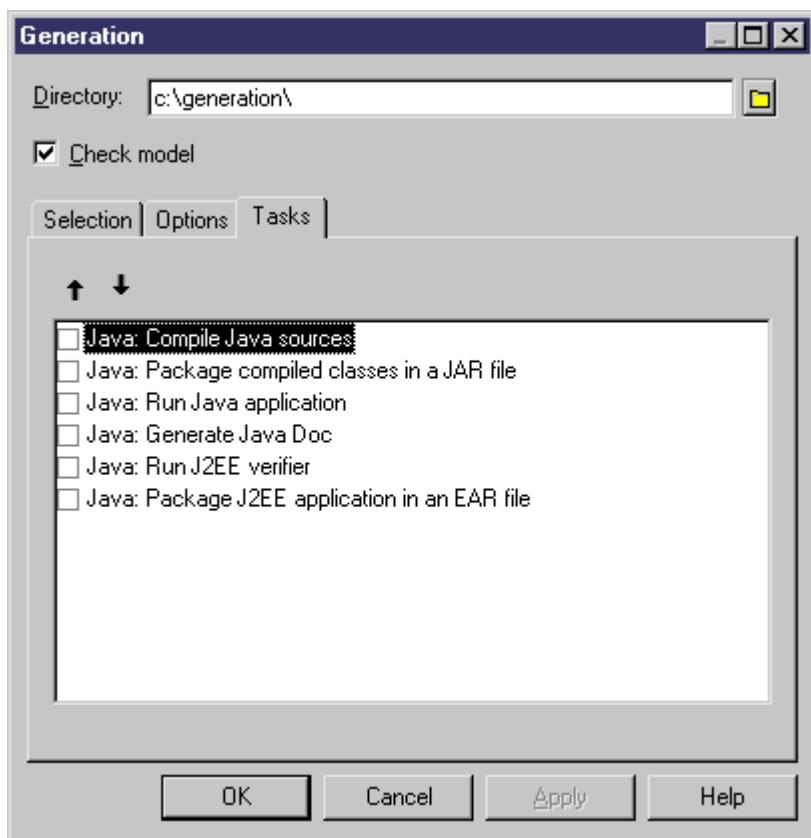


4. Click *Apply*.
5. Click the *Options* tab, then specify your generation options in the *Options* page.



For more information on the generation options, see [Generating Java Files \[page 352\]](#).

6. Click *Apply*.
7. Click the *Tasks* tab to display, then select the commands you want to perform during generation in the *Tasks* page.



For more information on the generation tasks, see [Generating Java Files \[page 352\]](#).

You must beforehand set the environment variables from the Variables tab of the General Options diaog box in order to activate them in this page.

For more information on how to set these variables, see *Core Features Guide > Modeling with PowerDesigner > Customizing Your Modeling Environment > General Options > Environment Variables*.

8. Click **OK** to begin generation.

A progress box is displayed, followed by a Result list. You can use the **Edit** button in the Result list to edit the generated files individually.

9. Click **Close**.

The web.XML file is created in the WEB-INF directory and all files are generated in the generation directory.

## 2.1.9.5 Reverse Engineering JSPs

You can reverse engineer JSPs code and deployment descriptor into an OOM. The reverse engineering feature reverses the files to create JSP components and reverses the deployment descriptor inside the WAR.

### Context

You start reverse engineering JSPs from  [Language](#)  [Reverse Engineer Java](#). Select one of the following Java formats from the Selection page, and select the *Reverse Engineer Deployment Descriptor* check box from the *Options* page:

- Java directories
- Class directories
- Archive (.JAR file)

For more information on the Java formats, see [Reverse Engineering Java Code \[page 355\]](#).

### Procedure

1. Select  [Language](#)  [Reverse Engineer Java](#).

The Reverse Java dialog box is displayed.

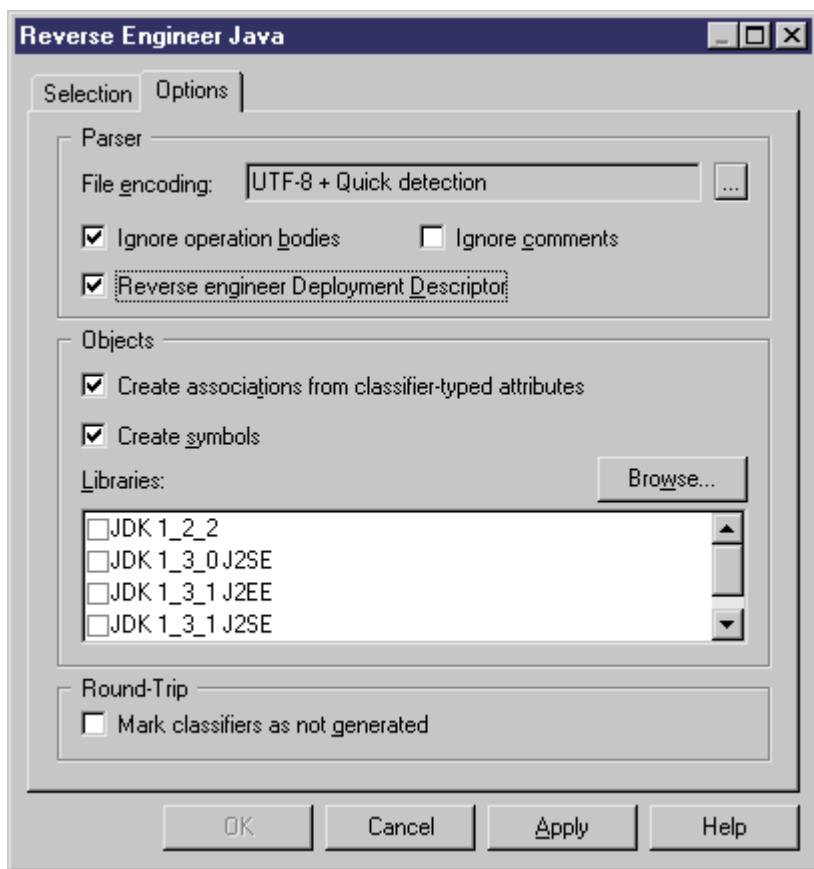
2. Select Archive from the Reverse list in the *Selection* page.
3. Click the *Add* button.

A standard Open dialog box is displayed.

4. Select the items you want to reverse and click *Open*.

The Reverse Java dialog box displays the items you selected.

5. Click the *Options* tab, then select the *Reverse Engineer Deployment Descriptor* check box.



6. Click [OK](#).

A progress box is displayed. If the model in which you are reverse engineering already contains data, the Merge Models dialog box is displayed.

For more information on merging models, see *Core Features Guide > Modeling with PowerDesigner > Comparing and Merging Models*.

7. Click [OK](#).

The Reverse page of the Output window displays the changes that occurred during reverse and the diagram window displays the updated model.

## 2.1.10 Generating Java Files

You generate Java source files from the classes and interfaces of a model. A separate file, with the file extension .java, is generated for each class or interface that you select from the model, along with a generation log file. You can only generate Java files from one model at a time.

### Context

The following PowerDesigner variables are used in the generation of Java source files:

| Variable  | Description   | Default      |
|-----------|---|--------------|
| J2EEVERIF | Batch program for verifying if the deployment jar for an EJB is correct | verifier.bat |
| JAR       | Command for archiving java files  | jar.exe      |
| JAVA      | Command for running JAVA programs                                       | java.exe     |
| JAVAC     | Command for compiling JAVA source files                                 | javac.exe    |
| JAVADOC   | Command for defining JAVA doc comments                                  | javadoc.exe  |

To review or edit these variables, select  [Tools](#)  [General Options](#) and click the [Variables](#) category. For example, you could add the JAVACLASSPATH variable in this table in order to override your system's CLASSPATH environment variable.

### Procedure

1. Select  [Language](#)  to open the Generation dialog.
2. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see [Checking an OOM \[page 251\]](#)).
3. [optional] Select any additional targets to generate for. These targets are defined by any extensions that may be attached to your model (see [Working with Generation Targets \[page 236\]](#)).
4. [optional] Click the [Selection](#) tab and specify the objects that you want to generate from. By default, all objects are generated.
5. [optional] Click the [Options](#) tab and set any appropriate generation options:

| Option  | Description  |
|---|--|
| Java: Sort class members primarily by   | <p>Sorts attributes and operations by:</p> <ul style="list-style-type: none"> <li>o Visibility- [default] Public attributes and operations are generated before private ones</li> <li>o Type - Attributes and operations are sorted by type whatever their visibility</li> </ul>   |
| Java: Class members type sort   | <p>Sorts attributes and operations in the order:</p> <ul style="list-style-type: none"> <li>o Attributes – Operations - Attributes are generated before the operations</li> <li>o Operations – Attributes - Operations are generated before the attributes</li> </ul>  |
| Java: Class members visibility sort   | <p>Sorts attributes and operations in the order:</p> <ul style="list-style-type: none"> <li>o Public – Private - Public attributes and operations are generated before private ones</li> <li>o Private – Public - Private attributes and operations are generated before public attributes and operations</li> <li>o None - Attributes and operations order remains unchanged</li> </ul> |
| Java: Generate package imports  | <p>When a class is used by another class, it is referenced by a class import:</p> <pre>import package1.package2.class.</pre> <p>This options allows you to declare import of the whole package, and saves time whenever many classes of the same package are referenced:</p> <pre>import package1.package2.*;</pre>  |
| Java: Generate object ids as Java-Doc tags  | <p>Generates information used for reverse engineering like object identifiers (@pdoid) that are generated as documentation tags. If you do not want these tags to be generated, you have to set this option to False</p>   |
| Java: Generate default accessors for navigable associations                             | <p>Generates the getter and setter methods for navigable associations</p>  |
| Ant: Generate Ant build.xml file  | <p>Generates the build.xml file. You can use this file if you have installed Ant</p>   |
| EJB: Generate CMP field accessors in component interfaces                               | <p>Generates CMP fields getter and setter operations to EJB interfaces</p>   |
| EJB: Generate CMR field accessors in component interfaces                               | <p>Generates CMR fields getter and setter declarations in EJB interfaces</p>   |
| EJB: Add Java classes source code in the JAR file                                       | <p>Includes Java classes code in the JAR</p>   |
| EJB: Generate value object class and associated navigation methods for CMP Entity Beans | <p>Generates an additional class named %Component.Code%ValueObject for each CMP bean class and declares all the CMP fields as public attributes. In addition, a getter and a setter are generated in the bean class for each CMR relationship</p>  |

| Option                          | Description                             |
|---------------------------------|---|
| J2EE: Jar Web component classes | Archives Web component classes in a Jar |

**i Note**

For information about modifying the options that appear on this and the *Tasks* tab and adding your own options and tasks, see *Customizing and Extending PowerDesigner > Object, Process, and XML Language Definition Files > Generation Category*.

- [optional] Click the *Generated Files* tab and specify which files will be generated. By default, all files are generated.

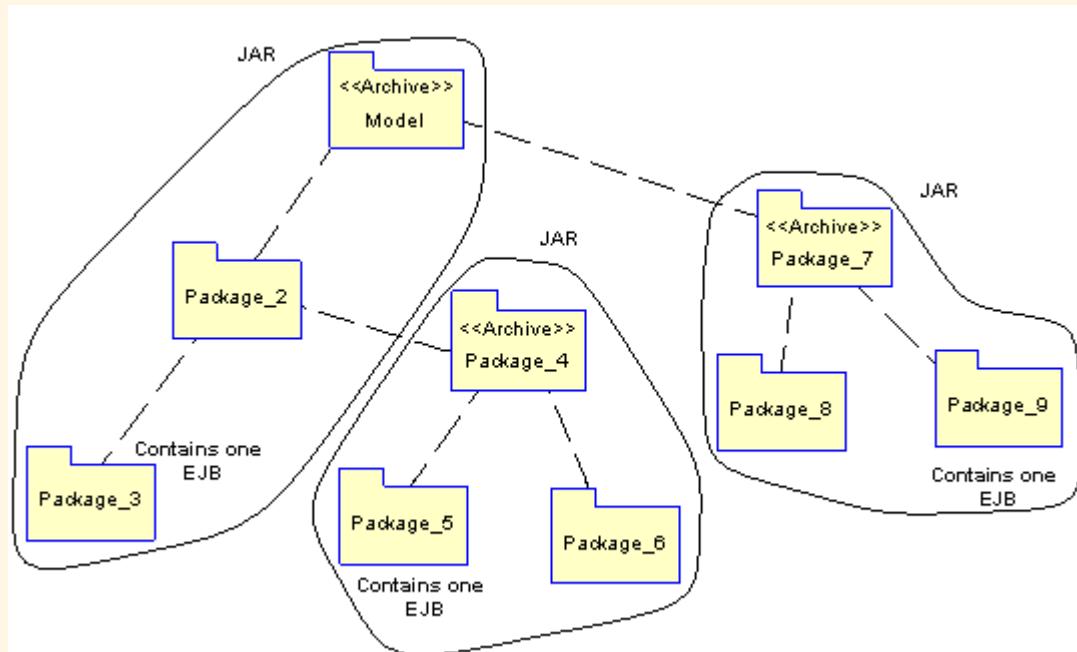
For information about customizing the files that will be generated, see *Customizing and Extending PowerDesigner > Extension Files > Generated Files (Profile)*.

- [optional] Click the *Tasks* tab and specify any appropriate generation tasks to perform:

| Task   | Description   |
|--|---|
| Java: Compile Java sources   | Starts a compiler using the javac command to compile Java source files.   |
| Java: Package compiled classes in a JAR file                           | Compiles source files and package them in a JAR file  |
| Java: Run Java application   | Compiles source files and run the Java application using the java command   |
| Java: Generate Javadoc   | Generates Javadoc   |
| Java: Package J2EE application in an EAR file                          | Calls commands for building EJB component source, creating a JAR file for Java classes and a deployment descriptor, building the Web component source code, creating an EAR file for Web component classes and a deployment descriptor, and creating an EAR archive containing all generated JAR/WAR files  |
| Java: Run J2EE verifier  | Calls commands for building EJB component source code, creating a JAR file for Java classes and a deployment descriptor, building the Web component source code, creating a WAR file for Web component classes and a deployment descriptor, creating an EAR archive containing all generated JAR/WAR files, and running the J2EE verifier on generated archives |
| WSDL: Compile and package Web Service server-side code into an archive | Calls commands for building EJB and Web component source code, running the WSCompile tool, creating a WAR file for Web component classes and deployment descriptor, and creating a JAR file for Java classes and deployment descriptor  |
| WSDL: Compile and package Web Service client proxy into an archive     | Calls commands for building EJB and Web component source code, running the WSCompile tool, and creating a WAR file for client-side artifacts  |

### Note

Packages with the <<archive>> stereotype will generate a JAR (when they or one of their descendant packages not stereotyped <<archive>> contain one EJB) or a WAR (when they contain a servlet or JSP). Each archive contains the package and all of its non-stereotyped descendants. The model acts as the root package and is considered to be stereotyped <<archive>>.



8. Click **OK** to begin generation.

When generation is complete, the Generated Files dialog opens, listing the files that have been generated to the specified directory. Select a file in the list and click **Edit** to open it in your associated editor, or click **Close** to exit the dialog.

## 2.1.11 Reverse Engineering Java Code

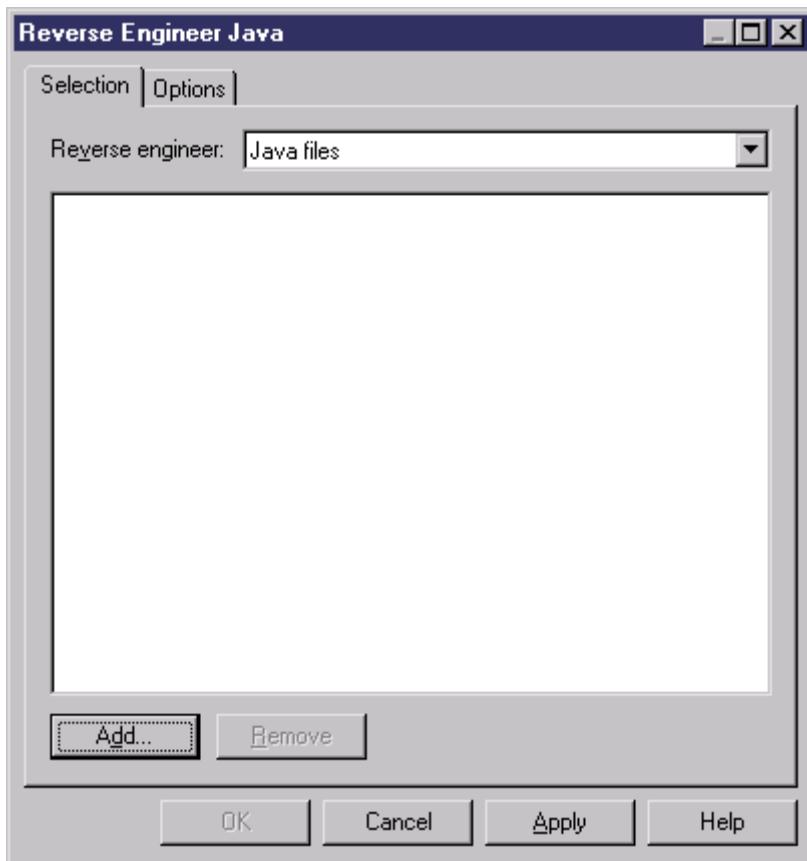
You can reverse engineer files that contain Java classes into an OOM. For each existing class in a Java file, a corresponding class is created in the OOM, with the same name and containing the same information.

### Context

When you reverse engineer a Java class that already exists in a model, a Merge Model window will open, allowing you to specify whether to replace existing classes, or to retain the existing class definitions in the model.

## Procedure

1. Select **Language > Reverse Engineer Java** to open the Reverse Engineer Java dialog box:



2. Select one of the following file formats from the Reverse engineer list:
  - Java files (.java) - Files contains one or several class definitions.
  - Java directories - Folders containing Java files. All the .java files, including those contained in sub-directories will be reverse engineered. Each sub-directory becomes a package within the model. As Java files in the same directory are often interdependent, if you do not reverse engineer all the files in the directory, your model may be incomplete.
  - Class files (.class) – Compiled files containing the definition of a single class with the same name as the file. Each sub-directory becomes a package within the model.
  - Class directories – Folders containing class files. All the .class files, including those contained in sub-directories will be reverse engineered.
  - Archives (.zip, .jar) - Compressed files containing definitions of one or several classes. PowerDesigner creates a class for each class definition in the .jar or .zip file. The following files are not reverse engineered: manifest.mf, web.xml, ejb-jar.xml, and \*.jsp. Other files are reverse engineered as files with the Artifact property set to true so that they can be generated later. Files are reverse engineered in packages corresponding to the directory structure found in the archive.
3. Click the **Add** button to browse to and select the files or directories to reverse, and then click **Open** to return to the Reverse Java dialog box, which now displays the selected files.

You can repeat this step as many times as necessary to select files or directories from different locations.

You can right-click any of the files and select *Edit* from the contextual menu to view its contents in an editor.

4. [optional] Click the *Options* tab and specify any appropriate reverse engineering options. For more information about these options, see [Reverse Engineer Java Options Tab \[page 357\]](#)

 Note

You can choose to reverse .java source files without their code body for visualization or comparison purposes, or to limit the size of your model if you have a very large number of classes to reverse engineer. To do this, select the Ignore operation body option.

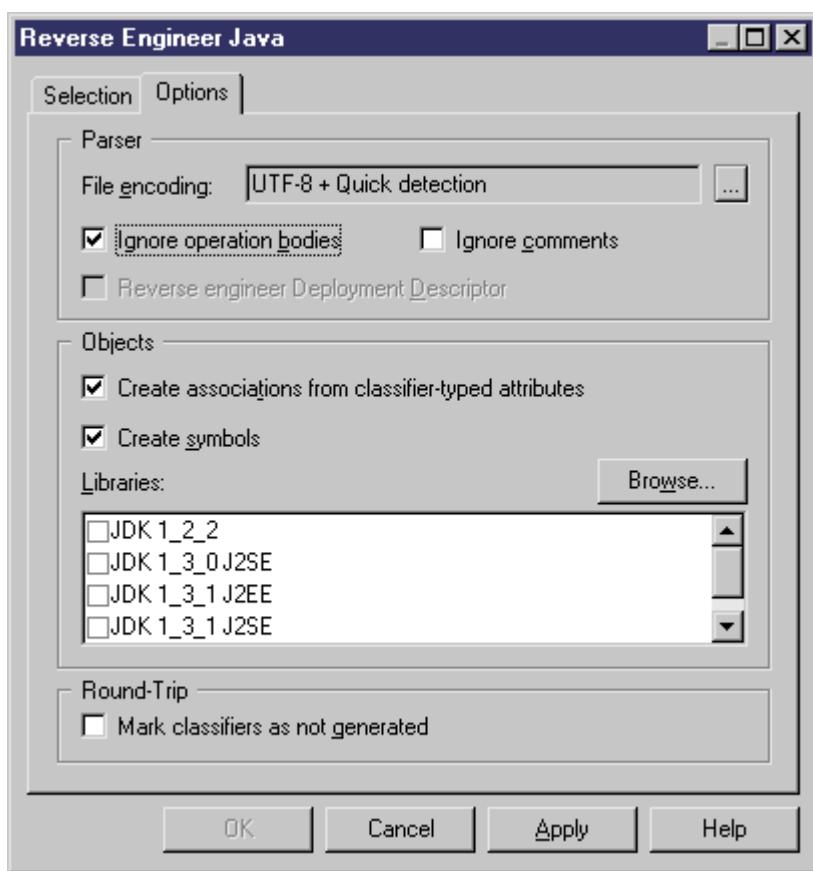
5. Click *OK* to begin the reverse engineering process. If the model in which you are reverse engineering already contains data, the Merge Models dialog box will open to allow you to specify whether to control whether existing objects will be overwritten.

For more information on merging models, see *Core Features Guide > Modeling with PowerDesigner > Comparing and Merging Models*.

PowerDesigner creates a class in the model for each class definition in the reversed files. The classes are visible in the Browser and, by default, symbols are created in one or more diagrams

### 2.1.11.1 Reverse Engineer Java Options Tab

The options tab allows you to specify various reverse engineering options.



The following Java reverse engineering options are available. Note that some may be disabled depending on the type of Java files being reversed:

| Option   | Result of selection   |
|--|---|
| File encoding  | Specifies the default file encoding of the files to reverse engineer.   |
| Ignore operation bodies                              | Reverses classes without including the body of the code. This can be useful when you want to reverse objects for visualization or comparison purposes, or to limit the size of your model if you have a very large number of classes to reverse.              |
| Ignore comments                                      | Reverses classes without including code comments.   |
| Reverse engineer Deployment Descriptor               | Reverses components with deployment descriptor. For more information, see <a href="#">Reverse Engineering EJB Components [page 324]</a> , <a href="#">Reverse Engineering Servlets [page 341]</a> , and <a href="#">Reverse Engineering JSPs [page 350]</a> . |
| Create associations from classifier-typed attributes | Creates associations between classes and/or interfaces.   |
| Create symbols                                       | Creates a symbol for each object in the diagram. If this option is not selected, reversed objects are only visible in the browser.  |

| Option                            | Result of selection  |
|-----------------------------------|--|
| Libraries                         | <p>Specifies a list of library models to be used as references during reverse engineering.</p> <p>The reverse engineered model may contain shortcuts to objects defined in a library. If you specify the library here, the link between the shortcut and its target object (in the library) will be preserved and the library will be added to the list of target models in the reverse engineered model.</p> <p>You can drag and drop the libraries in the list in order to specify a hierarchy among them. PowerDesigner will seek to resolve shortcuts found in the reverse engineered model against each of the specified libraries in turn. Thus, if library v1.1 is displayed in the list above library v1.0, PowerDesigner will first attempt to resolve shortcuts against library v1.1 and will only parse library v1.0 if unresolved shortcuts remain.</p> <p>You should use the List of Target Models to manage libraries related to the reverse engineered model, for example, you can change the library version (see <i>Core Features Guide &gt; Linking and Synchronizing Models &gt; Shortcuts and Replicas &gt; Working with Target Models</i>).</p> |
| Mark classifiers as not generated | Specifies that reversed classifiers (classes and interfaces) will not be generated from the model. To subsequently generate the classifier, you must select the Generate check box in its property sheet.  |

### 2.1.11.1.1 Reverse Engineering Java Code Comments

When you reverse engineer Java files, some comments may change form or position within the code.

| Comment in original Java file              | After reverse   |
|--|---|
| Before the import declarations             | Goes to header  |
| Beginning with /*                          | Begins with //  |
| At the end of the file below all the code  | Goes to footer  |
| Within a class but not within an operation | Is attached to the attribute or operation that immediately follows it |

## 2.2 Technical Architecture Modeling (TAM)

Technical Architecture Modeling is the SAP internal standard for architecture modeling and combines elements of FMC and UML.

PowerDesigner supports the modeling of the following TAM diagrams:

- Use case diagram - see [Use Case Diagrams \[page 20\]](#).
- Class diagram - see [Class Diagrams \[page 31\]](#).
- Package diagram - see [Package Diagrams \[page 35\]](#).
- Sequence diagram - see [Sequence Diagrams \[page 116\]](#).
- Activity diagram - see [Activity Diagrams \[page 120\]](#).
- Statechart diagram - see [Statechart Diagrams \[page 123\]](#).
- Component diagram - see [Component Diagrams \[page 192\]](#).
- Block diagram - see [Block Diagrams \(TAM\) \[page 360\]](#).

To create a TAM OOM, use the Technical Architecture Modeling (TAM) category on the [Categories](#) tab of the New Model dialog, or create an OOM on the [Model types](#) tab targeting the Technical Architecture Modeling (TAM) object language.

## 2.2.1 Block Diagrams (TAM)

Block diagrams show the compositional structure of any system that processes information and illustrate how agents access data in storages and communicate over channels. PowerDesigner supports block diagrams as component diagrams with additional TAM tools.

### Note

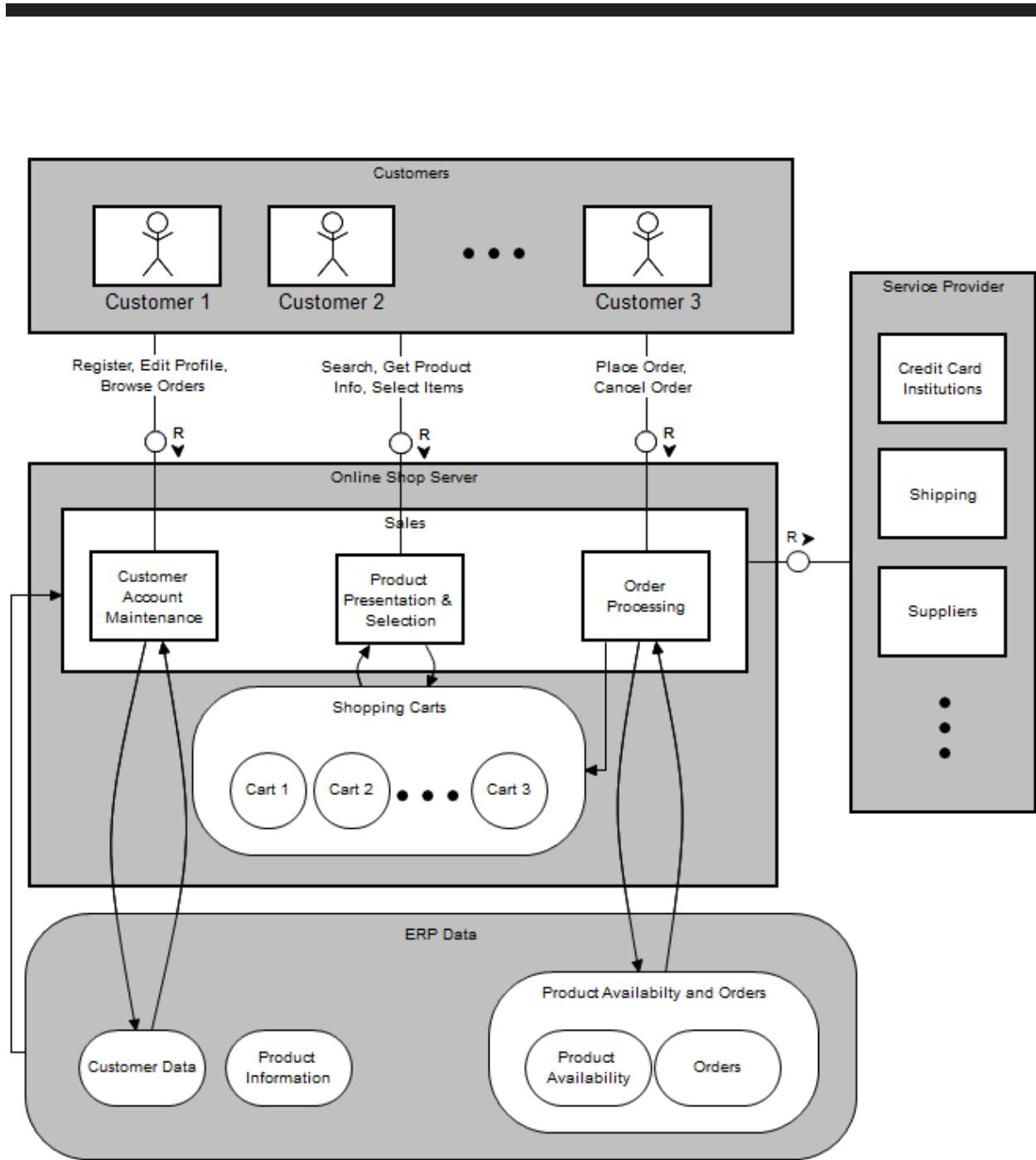
To create a block diagram in an existing TAM OOM, right-click the model in the Browser and select  [New](#)  [Component Diagram](#). To create a new model, select   [File](#)  [New Model](#), click the Categories button, select the Technical Architecture Modeling (TAM) category, click the Block Diagram item, specify a model name, and then click [OK](#).

The following tools are available in the Technical Architecture Modeling (TAM) toolbox in a block diagram:

| Tool  | Description   |
|---|---|
|   | Agents, Human Agents - Represent the active parts of the system. Only agents can do something, all other elements in a block diagram are passive. You can nest agents and human agents inside agents. Human agents do not allow nesting.<br><br>You can use nesting to show the inner structure of agents and storages. You can group elements that belong to one system (part), are executed on one platform, are located on the same database, and so on, to reduce the number of channels between agents, or accesses between agents and storages. |
|    | Storages - Contain information that can be processed by an agent. To specify the type of the storage and the format in which the data is stored, open its property sheet and use the <a href="#">Storage type</a> and <a href="#">Storage format</a> properties. You can select values from the list or enter your own.   |
|    | Common Feature Areas - Provide logical spaces for grouping any other elements. To add objects to the area, create them directly on the area or drag them onto it. Areas can be nested.  |

| Tool  | Description  |
|---|--|
|    | <p>Read, Write, and Modify Accesses - Connect agents with storages.</p> <p>To change the type of access after creation, right-click the access and select the appropriate <i>Change to...</i> command, or open its property sheet and use the <i>Access type</i> property.</p> <p>Modify access can be displayed as single lines or as double arcs. To control the symbol type, right-click the access and select the <i>Display as Single Line</i> or <i>Display as Double Arc</i> command, or use the <i>Display as double-arc</i> property.</p>   |
|    | <p>Request/Response, Unidirectional, and Bidirectional Communication Channels - Represent telephone lines, network connections, pipes, or simple procedure calls and connect two or more agents, enabling them to communicate.</p> <p>To change the type of channel after creation, right-click the channel and select the appropriate <i>Change to...</i> command, or open its property sheet and use the <i>Communication type</i> property.</p> <p>Arrows indicate the direction of information flow. An "R" with a small arrow represents a request-response channel pair, with the arrow indicating the direction of the request from client to server.</p> <p>To reverse the direction, right-click the channel and select the <i>Change Channel Direction</i>. To control the display of arrows, right-click the channel and select the <i>Show Arrows</i> or <i>Hide Arrows</i> command, or use the <i>Show arrows</i> property.</p> |
|  | <p>Multiple Dots - [in Predefined Symbols toolbox] Can be placed in an agent or storage symbol to indicate multiple instances.</p>   |
|  | <p>Boundary Line, Protocol Boundary - [use the Line tool in the Free Symbols toolbox] Identifies a system boundary. To change the format of a boundary line, right-click it and select the <i>Format</i> command. For example, to obtain a dashed line, select the appropriate style from the list on the <i>Line Style</i> tab of the Format dialog.</p>  |

The following example shows customer agents communicating with an online shop server, in which sales components access shopping cart and ERP storages:



## 2.3 Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework (EMF), is a modeling framework and code generation facility for building tools and other applications based on a structured data model. An EMF model provides a simple model of the classes and data of an application and is used as a metadata definition framework in many Eclipse-based tools.

PowerDesigner supports modeling in the EMF language including round-trip engineering.

PowerDesigner supports the following EMF objects:

- EAnnotations - As standard UML classes, with an `AnnotationType` stereotype. EAnnotation property sheets contain all the standard class tabs (see [Classes \(OOM\) \[page 38\]](#)) along with the `EMF` tab, containing the following properties:

| Property   | Description                              |
|------------|--|
| References | Specifies the EMF annotation references. |
| URI        | Specifies the EMF annotation source.     |

- EAttributes and EEnumLiterals (EAttributes belonging to EEnums) - As standard UML attributes. These objects' property sheets contain all the standard attribute tabs (see [Attributes \(OOM\) \[page 67\]](#)) along with the `EMF` tab, containing the following properties:

| Property   | Description   |
|------------|---|
| Unique     | Specifies that the attribute may not have duplicates. |
| Unsettable | Generates an unset method to undo the set operation.  |
| Ordered    | Specifies that the attribute is ordered.              |

- Eclasses, EEnums, and EDataTypes - As standard UML classes. EEnums and EDataTypes bear the `Enum` and `EDataType` stereotype, respectively. These objects property sheets contain all the standard class tabs (see [Classes \(OOM\) \[page 38\]](#)) along with the `EMF` tab, containing the following properties:

| Property            | Description                                  |
|---------------------|--|
| Instance Class Name | Specifies the data type instance class name. |

- EPackages - As standard UML packages. EPackage property sheets contain all the standard package tabs (see [Packages \(OOM\) \[page 57\]](#)) along with the `EMF` tab, containing the following properties:

| Property          | Description  |
|-------------------|--|
| Namespace prefix  | Used when references to instances of the classes in this package are serialized.   |
| Namespace URI     | Appears in the <code>xmlns</code> tag to identify this package in an XMI document. |
| Base package name | Contains the generated code for the model.   |

- EOperations and EParameters - As standard UML operations and operation parameters. These objects property sheets contain all the standard operation or parameter tabs (see [Operations \(OOM\) \[page 77\]](#)).
- EReferences - As standard UML associations. EReference property sheets contain all the standard association tabs (see [Associations \(OOM\) \[page 84\]](#)) along with the `EMF` tab, containing the following properties:

| Property        | Description   |
|-----------------|---|
| Unique          | Specifies that the selected role may not have duplicates. |
| Resolve proxies | Resolves proxies if the selected role is in another file. |
| Unsettable      | Specifies that the selected role cannot be set.           |

For more information on EMF, see the EMF documentation and tutorials at <http://www.eclipse.org/emf>.

## 2.3.1 Generating EMF Files

PowerDesigner can generate EMF .ecore and .genmodel files.

### Procedure

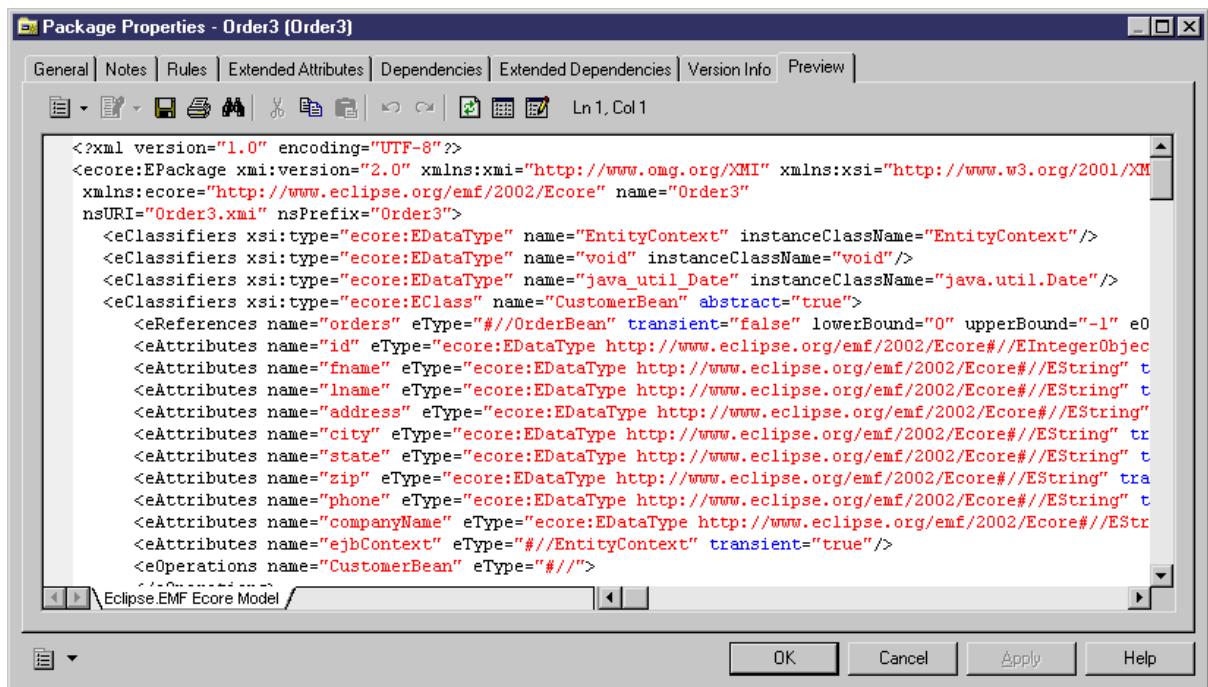
1. Select  *Language*  *Generate EMF Code*  to open the Generation dialog box:
2. Enter a directory in which to generate the files and specify whether you want to perform a model check.
3. [optional] On the *Selection* tab, specify the objects that you want to generate from. By default, all objects are generated, and PowerDesigner remembers for any subsequent generation the changes you make.

#### Note

Although you can create all the standard UML diagrams and their associated objects, you can only generate packages, classes, and interfaces.

4. [optional] Click the *Options* tab and specify the EMF version that you want to generate for.
5. Click *OK* to begin generation.

A Progress box is displayed. The Result list displays the files that you can edit. The result is also displayed in the *Generation* tab of the Output window, located in the bottom part of the main window.



## 2.3.2 Reverse Engineering EMF Files

In order to reverse engineer EMF files into an OOM, you must use the PowerDesigner Eclipse plugin, and also have installed the EMF plugin.

### Procedure

1. Select **Language > Reverse Engineer EMF** to open the Reverse Engineer OOM from EMF file dialog box.
2. Click one of the following buttons to browse to .ecore, .emof, or .genmodel files to reverse engineer:
  - Browse File System
  - Browse Workspace
3. Select the files to reverse engineer and click Open (or OK) to add them to the Model URIs list.
4. Click Next to go to the Package Selection page, and select the packages to reverse engineer.
5. Click Finish to begin the reverse engineering.

If the model in which you are reverse engineering already contains data, the Merge Models dialog box is displayed.

For more information on merging models, see *Core Features Guide > Modeling with PowerDesigner > Comparing and Merging Models*.

The packages are added to your model.

## 2.4 PowerBuilder

PowerDesigner supports the modeling of PowerBuilder programs including round-trip engineering.

PowerBuilder is an object oriented development tool. Most of its components are designed as objects with properties, methods and events that can be mapped to UML classes bearing specific stereotypes. PowerDesigner supports PowerBuilder objects stored in a .PBL file. Dynamic PowerBuilder Libraries (PBD) are not supported.

### i Note

If you have multiple versions of PowerBuilder installed on your machine PowerDesigner uses the most recent version by default. If you want to work on an earlier version of PowerBuilder, click the *Change* button to the right of the *PowerBuilder version* field in the generation or reverse engineering dialog.

### 2.4.1 PowerBuilder Objects

This section describes the mapping between PowerBuilder objects and PowerDesigner OOM objects.

## Applications

You design a PowerBuilder application using a class with the <<application>> stereotype. Application properties are defined as follow:

| PowerBuilder        | PowerDesigner   |
|---------------------|---|
| Instance variable   | Attribute   |
| Shared variable     | Static attribute  |
| Global variable     | Attribute with <<global>> stereotype                                      |
| Property            | Attribute with <<property>> stereotype                                    |
| External function   | Operation with <<external>> stereotype                                    |
| Function            | Operation   |
| Event               | Operation with <<event>> stereotype or operation with non-null event name |
| Structure in object | Inner class with <<structure>> stereotype                                 |

## Structures

You design a PowerBuilder structure using a class with the <<structure>> stereotype. The members of the structure are designed with class attributes.

## Functions

You design a PowerBuilder function using a class with the <<function>> stereotype. This class should also contain one operation. The structures in a function are designed with <<structure>> inner classes linked to the class.

## User Objects

You design a PowerBuilder user object using a class with the <<userObject>> stereotype. User objects properties are defined as follow:

| PowerBuilder        | PowerDesigner   |
|---------------------|---|
| Control             | inner class with <<control>> stereotype                                   |
| Instance variable   | Attribute   |
| Shared variable     | Static attribute  |
| Property            | attribute with <<property>> stereotype                                    |
| Function            | Operation   |
| Event               | operation with <<event>> stereotype or operation with non-null event name |
| Structure in object | inner class with <<structure>> stereotype                                 |

## Proxies

You design a PowerBuilder proxy using a class with the <<proxyObject>> stereotype. Instance variables of the proxy are designed with class attributes, and proxy functions are designed with operations.

## Windows

You design a PowerBuilder window using a class with the <<window>> stereotype. Window properties are defined as follow:

| PowerBuilder        | PowerDesigner   |
|---------------------|---|
| Control             | inner class with <<control>> stereotype                                   |
| Instance variable   | Attribute   |
| Shared variable     | Static attribute  |
| Property            | Attribute with <<property>> stereotype                                    |
| Function            | Operation   |
| Event               | Operation with <<event>> stereotype or operation with non-null event name |
| Structure in object | Inner class with <<structure>> stereotype                                 |

## Operations

If the operation extended attribute GenerateHeader is set to true, the operation header will be generated. This attribute is set to true for any new operation. You can force header generation for all operations in a model by setting the ForceOperationHeader extended attribute to true. Operation headers are generated in the following way:

```
//<FuncType>: <Operation signature>
//Description: <Operation comment line1>
//          <Operation comment line2>
//Access: <visibility>
//Arguments: <parameter1 name> - <parameter1 comment line1>
//          <parameter1 comment line2>
//          <parameter2 name> - <parameter2 comment>
//Returns: <Return comment>
//          <Return comment2>
```

| Header item | PowerDesigner Object or Property                                      |
|-------------|---|
| FuncType    | Function, Subroutine or Event   |
| Description | Comment typed in operation property sheet                             |
| Access      | Visibility property in operation property sheet                       |
| Arguments   | Parameter(s) name and comment   |
| Returns     | Value of ReturnComment extended attribute in operation property sheet |

| Header item          | PowerDesigner Object or Property   |
|----------------------|--|
| User-defined comment | Value of UserDefinedComment extended attribute in operation property sheet |

## Events

To generate a:

- Standard event handler - create an operation and select an event value in the Language Event list in the operation property sheet
- User-defined event handler - create an operation and select the <>event<> stereotype. The Language Event list must remain empty
- Custom event handler - create an operation and set a value to the EventID extended attribute. If this extended attribute has a value, the operation is generated as a custom event handler, even if it has a name defined in the Language Event list or the <>event<> stereotype.

## Other Objects

These PowerBuilder objects are reverse engineered as classes with the corresponding PowerBuilder stereotype. Their properties are not mapped to PowerDesigner class properties, and their symbol is a large PowerBuilder icon.

| PowerBuilder | PowerDesigner        |
|--------------|----------------------|
| Query        | <>query<> class      |
| Data window  | <>dataWindow<> class |
| Menu         | <>menu<> class       |
| Project      | <>project<> class    |
| Pipe line    | <>pipeLine<> class   |
| Binary       | <>binary<> class     |

For more information about PowerBuilder reverse engineering, see [Reverse Engineering PowerBuilder \[page 371\]](#).

## 2.4.2 Generating PowerBuilder Objects

You can generate PowerBuilder objects to an existing PowerBuilder application or as source files. Each class bearing a stereotype is generated as the appropriate PowerBuilder object. Classes without stereotypes are

generated as user objects. Objects not fully supported by PowerDesigner have all their properties removed and only the header is generated.

## Procedure

1. Select   to open the PowerBuilder Generation dialog.
2. [optional] Click the *Selection* tab and specify the objects that you want to generate from. By default, all objects are generated.
3. [optional] Click the *Options* tab and set any appropriate generation options:

| Option             | Description   |
|--------------------|---|
| Check model        | Launches a model check before generation (see <a href="#">Checking an OOM [page 251]</a> ).   |
| Using libraries    | <p>This mode is only available if you have PowerBuilder installed on your machine.</p> <p>Specify a PowerBuilder version and select a target or application from the Target/Application list. Objects are generated as follows:</p> <ul style="list-style-type: none"><li>○ Package with specified library path (defined in an extended attribute during reverse engineering) is generated in corresponding library from target/application library list</li><li>○ Package at the root of the model without library path is generated in a new library at the same level as target/application library</li><li>○ Child package without library path is generated in parent package</li><li>○ Object at the root of the model is generated in the target/application library</li></ul> |
| Using source files | <p>Specify a directory to which to generate the files. Objects are generated as follows:</p> <ul style="list-style-type: none"><li>○ Classes Defined at the Model Level are generated as source files in the specified directory.</li><li>○ Classes Defined in Packages are generated as source files in sub-directories.</li></ul> <p>You must import the generated objects into PowerBuilder.</p>   |

4. Click **OK** to begin generation.

The files are generated to the specified application or directory.

## 2.4.3 Reverse Engineering PowerBuilder

This section explains how PowerBuilder objects are reverse engineered and how to define reverse engineering options for PowerBuilder.

### 2.4.3.1 Reverse Engineered Objects

You can reverse engineer into an OOM, objects stored in a .PBL file or exported by PowerBuilder into files. Some of the reverse engineered objects support a full-featured mapping with an OOM class, some do not.

#### Libraries

Each PowerBuilder library is reversed as a package in the resulting OOM. The path of the library is stored in an extended attribute attached to the package.

Objects reverse engineered from a library are created into the corresponding package in PowerDesigner.

#### Full-featured Mapping

During reverse engineering, new classes with stereotype corresponding to the PowerBuilder objects they come from are created. The symbol of these classes displays an icon in the upper left corner:



For more information on fully supported PowerBuilder objects, [PowerBuilder Objects \[page 366\]](#).

#### Minimal Mapping

PowerBuilder objects not fully supported in PowerDesigner are reverse engineered as classes with the corresponding PowerBuilder stereotype. However, their properties are not mapped to PowerDesigner class properties, and their symbol is a large PowerBuilder icon.



The source code of these objects is retrieved without any parsing and stored in the class header, as displayed in the Script\Header tab of the class; it will be used in the same way during generation.

For more information on partially supported PowerBuilder objects, see [PowerBuilder Objects \[page 366\]](#).

### 2.4.3.1.1 Operation Reversed Header

Reverse engineering processes the first comment block of the function between two lines of slash characters.

```
//////////  
// <FuncType>: <Operation signature>  
// Description: <Operation comment line1>  
//     <Operation comment line2>  
// Access: <visibility>  
// Arguments: <parameter1 name> - <parameter1 comment line1>  
//             <parameter1 comment line2>  
//             <parameter2 name> - <parameter2 comment>  
// Returns: <Return comment>  
//           <Return comment2>  
//////////
```

If all generated keywords are found, the block will be removed and relevant attributes will be set:

| Keywords attribute          | Corresponding operation attribute               |
|-----------------------------|---|
| FuncType, Subroutine, Event | Name  |
| Description                 | Operation comment                               |
| Access                      | Visibility property                             |
| Arguments                   | Parameter(s) name and comment                   |
| Returns                     | Value for ReturnComment extended attribute      |
| User-defined comment        | Value for UserDefinedComment extended attribute |
| GenerateHeader              | Set to True                                     |
| Other function comments     | Kept in operation body                          |

Otherwise, the function comments are kept in the operation body and the GenerateHeader extended attribute set to false.

## 2.4.3.1.2 Overriding Attributes

When a class inherits from another class, non-private inherited attributes can be defined as properties of the child class, allowing the user to define initial values in the child class.

### Procedure

1. Open the property sheet of a child class, and click the *Attributes* tab.
2. Click the *Override Inherited Attributes* tool to display the list of attributes available from the parent class.
3. Select one or more attributes in the list and click *OK*.

The attributes appear in the child class list of attributes. You can modify their initial value in the corresponding column.

## 2.4.3.2 PowerBuilder Reverse Engineering Process

When you reverse engineer objects from PowerBuilder, you can select to reverse engineer libraries, files or directories.

### Reverse Engineering Libraries

This mode allows you to select a PowerBuilder target/application from the Target/Application list. When a target or an application is selected, the libraries used by the target or application are automatically displayed in the list. By default all objects of all libraries are selected. You can deselect objects and libraries before starting reverse engineering.

If PowerBuilder is not installed on your machine, the Target/Application list remains empty.

### Reverse Engineering Source Files

This mode allows you to select PowerBuilder object source files to reverse engineer. The extension of the source file determines the type of the reversed object.

You can right-click the files to reverse engineer and select the Edit command to view the content of your files. To use this command you have to associate the file extension with an editor in the General Options\Editor dialog box.

## Reverse Engineering Directories

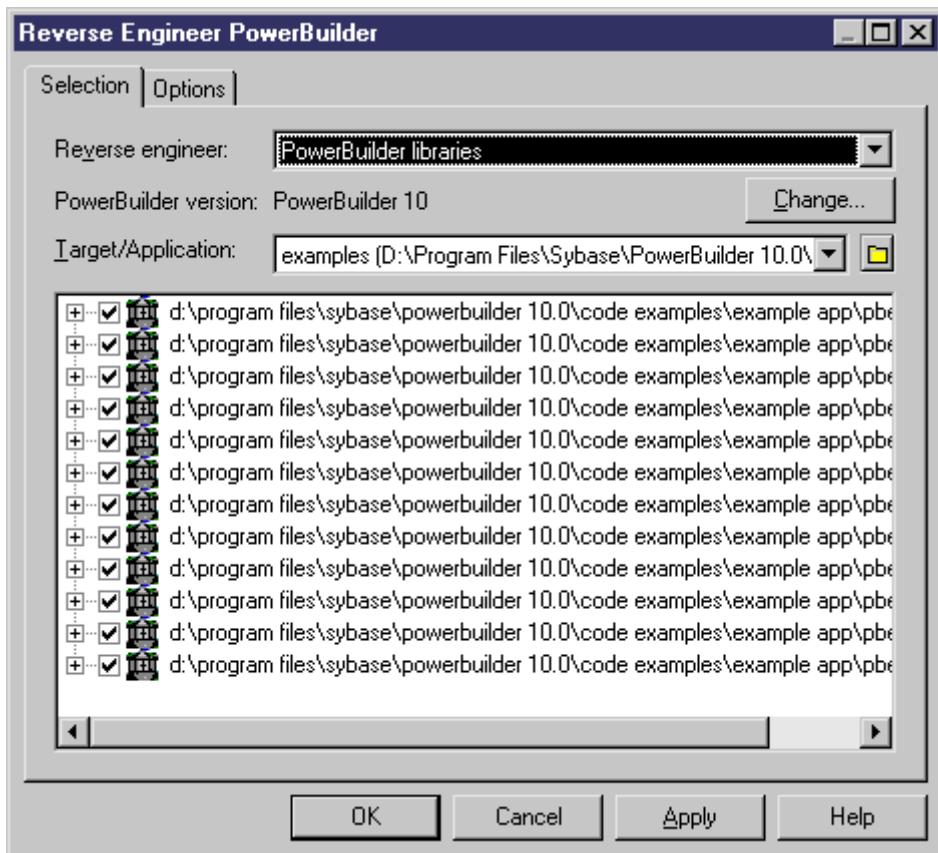
This mode allows you to select a PowerBuilder directory to reverse engineer. When you select a directory, you cannot select individual target or application. Use the Change button to select a directory.

### 2.4.3.2.1 Reverse Engineering PowerBuilder Objects

You can reverse engineer PowerBuilder objects by selecting **► Language ► Reverse Engineer PowerBuilder**.

#### Procedure

1. Select **► Language ► Reverse Engineer PowerBuilder** to display the Reverse Engineer PowerBuilder dialog box.
2. Select a file, library or directory in the Reverse Engineering box.
3. When available, select a target or application in the list.



4. Click the Options tab and set any appropriate options.

| Option                               | Description   |
|--------------------------------------|---|
| Ignore operation body                | Reverses PowerBuilder objects without including the body of the code  |
| Ignore comments                      | Reverses PowerBuilder objects without including code comments   |
| Create symbols                       | Creates a symbol in the diagram for each object. Otherwise, reversed objects are visible only in the browser  |
| Create inner classes symbols         | Creates a symbol in the diagram for each inner class  |
| Mark classifiers not to be generated | Reversed classifiers (classes and interfaces) will not be generated from the model. To generate the classifier, you must select the <a href="#">Generate</a> check box in its property sheet  |
| Create Associations                  | Creates associations between classes and/or interfaces  |
| Libraries                            | <p>Specifies a list of library models to be used as references during reverse engineering.</p> <p>The reverse engineered model may contain shortcuts to objects defined in a library. If you specify the library here, the link between the shortcut and its target object (in the library) will be preserved and the library will be added to the list of target models in the reverse engineered model.</p> <p>You can drag and drop the libraries in the list in order to specify a hierarchy among them. PowerDesigner will seek to resolve shortcuts found in the reverse engineered model against each of the specified libraries in turn. Thus, if library v1.1 is displayed in the list above library v1.0, PowerDesigner will first attempt to resolve shortcuts against library v1.1 and will only parse library v1.0 if unresolved shortcuts remain.</p> <p>You should use the List of Target Models to manage libraries related to the reverse engineered model, for example, you can change the library version. See <i>Core Features Guide &gt; Linking and Synchronizing Models &gt; Shortcuts and Replicas &gt; Working with Target Models</i>.</p> |

5. Click [OK](#).

A progress box is displayed. If the model in which you are reverse engineering already contains data, the Merge Models dialog (see *Core Features Guide > Modeling with PowerDesigner > Comparing and Merging Models*) is displayed.

The classes are added to your model. They are visible in the diagram and in the Browser. They are also listed in the Reverse tab of the Output window, located in the lower part of the main window.

### Note

Some standard objects like windows or structures, inherit from parent classes defined in the system libraries. If these libraries are not loaded in the workspace, PowerDesigner no longer creates an unresolved class to represent the standard object parent in the model. The link between standard object and parent will be recreated after generation thanks to the standard object stereotype.

### **2.4.3.3 Loading a PowerBuilder Library Model in the Workspace**

When you reverse engineer PowerBuilder files, you can, at the same time, load one of the PowerBuilder models that contains the class libraries of a particular version of PowerBuilder. The Setup program installs these models in the Library directory.

#### **Context**

You can select to reverse a PowerBuilder library model from the Options tab of the Reverse Engineer PowerBuilder dialog box.

You can open a PowerBuilder library model in the workspace from the Library directory.

#### **Procedure**

1. Select  *File* > *Open*  to display the Open dialog box.
2. Select or browse to the Library directory.

The available library files are listed. Each PB file corresponds to a particular version of PowerBuilder.

3. Select the file corresponding to the version you need.

This file contains all the library class files of the PowerBuilder version that you have chosen.

4. Click Open.

The OOM opens in the workspace.

## **2.5 VB .NET**

PowerDesigner supports the modeling of VB .NET programs including round-trip engineering.

### **2.5.1 Inheritance & Implementation**

You design VB .NET inheritance using a generalization link between classes.

You design VB .NET implementation using a realization link between a class and an interface.

## 2.5.2 Namespace

You define a VB .NET namespace using a package.

PowerDesigner models namespaces as standard packages with the *Use Parent Namespace* property deselected.

In the following example, class Architect is declared in package Design which is a sub-package of Factory. The namespace declaration is the following:

```
Namespace Factory.Design
    Public Class Architect
    ...
    ...End Class
End Namespace ' Factory.Design
```

Classifiers defined directly at the model level fall into the VB .NET global namespace.

## 2.5.3 Project

You can reverse engineer VB .NET projects when you select VB .NET projects from the Reverse Engineer list in the Reverse Engineer VB .NET dialog box.

Make sure you reverse engineer each project into a separate model.

Assembly properties are reverse engineered as follow:

| VB .NET assembly properties  | PowerDesigner equivalent                     |
|------------------------------|--|
| Title                        | Name of the model                            |
| Description                  | Description of the model                     |
| Company                      | AssemblyCompany extended attribute           |
| Copyright                    | AssemblyCopyright extended attribute         |
| Product                      | AssemblyProduct extended attribute           |
| Trademark                    | AssemblyTrademark extended attribute         |
| Version                      | AssemblyVersion extended attribute           |
| AssemblyInformationalVersion | Stored inCustomAttributes extended attribute |
| AssemblyFileVersion          |  |
| AssemblyDefaultAlias         |  |

Project properties are reverse engineered as extended attributes whether they have a value or not. For example, the default HTML page layout is saved in extended attribute DefaultHTMLPageLayout.

You can use the Ellipsis button in the Value column to modify the extended attribute value, however you should be very cautious when performing such changes as they may jeopardize model generation.

## 2.5.4 Accessibility

To define accessibility for a class, an interface, an attribute or a method, you have to use the visibility property in PowerDesigner.

The following accessibility attributes are supported in PowerDesigner:

| VB .NET accessibility  | PowerDesigner visibility |
|--|--------------------------|
| Public (no restriction)  | Public                   |
| Protected (accessible by derived classes)  | Protected                |
| Friend (accessible within the program that contains the declaration of the class)                                  | Friend                   |
| Protected Friend (accessible by derived classes and within the program that contains the declaration of the class) | Protected Friend         |
| Private (only accessible by the class)   | Private                  |

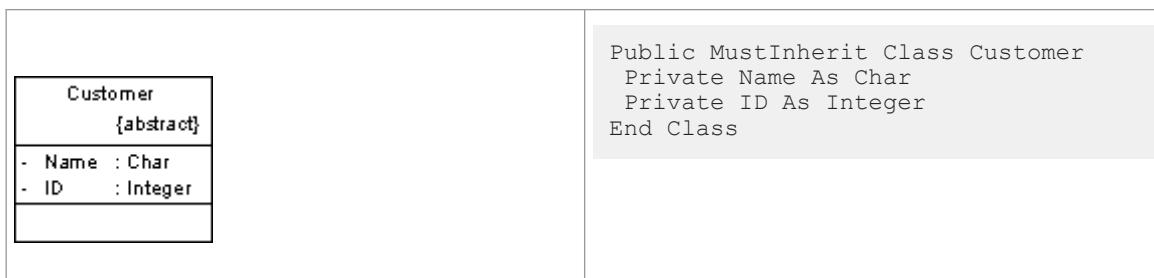
In the following example, the visibility of class Customer is friend:

```
Friend Class Customer
```

## 2.5.5 Classes, Interfaces, Structs, and Enumerations

You design a VB .NET class using a class in PowerDesigner. Structures are classes with the <<structure>> stereotype, and enumerations are classes with the <<enumeration>> stereotype.

- VB .NET classes can contain events, variables, constants, methods, constructors and properties. The following specific kinds of classes are also supported:
  - MustInherit class is equivalent to an abstract class. To design this type of class you need to create a class and select the Abstract check box in the *General* tab of the class property sheet.



- NotInheritable class is equivalent to a final class. To design this type of class, you need to create a class and select the Final check box in the *General* tab of the class property sheet.

|  |   |
|--|---|
| <pre> <b>FinalClass</b> - At1 : Object - At2 : Object </pre> | <pre> Public NotInheritable Class FinalClass     Private At1 As Object     Private At2 As Object End Class </pre> |
|--|---|

### i Note

You design a VB .NET nested type using an inner class or interface.

- VB .NET interfaces are modeled as standard interfaces. They can contain events, properties, and methods; they do not support variables, constants, and constructors.
- Structures can implement interfaces but do not support inheritance; they can contain events, variables, constants, methods, constructors, and properties. The following structure contains two attributes and a constructor operation:

|   |   |
|---|---|
| <pre> <b>Point</b> # Y : Integer # X : Integer + &lt;&lt;Constructor&gt;&gt; New() </pre> | <pre> ... Public Class Point     Protected Y As Integer     Protected X As Integer     Public Sub New()      End Sub End Class ... </pre> |
|---|---|

- Enumeration class attributes are used as enumeration values. The following items must be set:
  - Data Type - using the EnumDataType extended attribute of the enumeration (for example Byte, Short, or Long)
  - Initial Expression - using the Initial Value field of an enum attribute

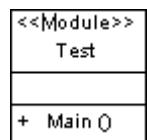
For example:

|  |   |
|--|---|
| <pre> &lt;&lt;enumeration&gt;&gt; Day - Monday - Tuesday - Wednesday - Thursday - Friday - Saturday - Sunday - FirstDay = Monday - LastDay = Sunday </pre> | <pre> Public Enum Day     Monday     Tuesday     Wednesday     Thursday     Friday     Saturday     Sunday     FirstDay = Monday     LastDay = Sunday End Enum </pre> |
|--|---|

## 2.5.6 Module

You design a VB .NET module using a class with the <<Module>> stereotype and attributes, functions, subs and events.

In the following example, you define a module Test using a class with the <<Module>> stereotype. Test contains a function. To design this function you have to create an operation called Main and empty the return type property. You can then define the function body in the implementation tab of this operation.

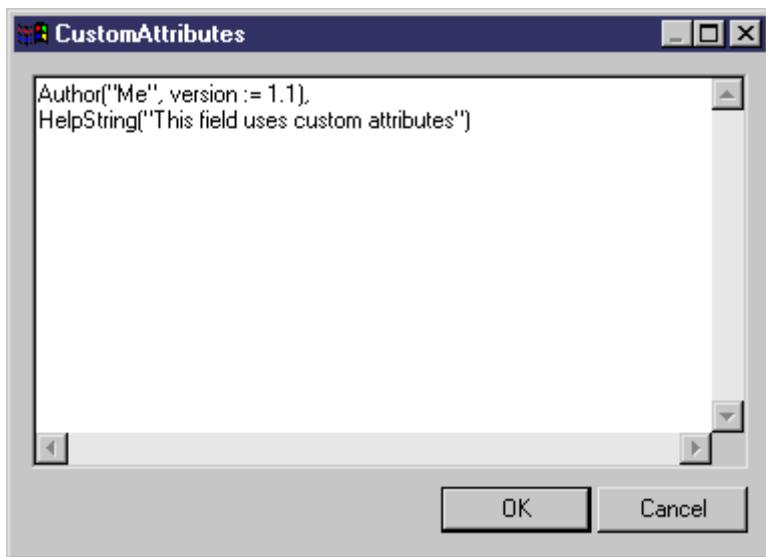


```
...
Public Module Test
    Public Sub Main()
        Dim val1 As Integer = 0
        Dim val1 As Integer = val1
        val2 = 123
        Dim ref1 As New Class1 ()
        Dim ref1 As Class1 = ref1
        ref2.Value = 123
        Console.WriteLine ("Value: "& val1, "& val2)
        Console.WriteLine ("Refs: "&ref1.Value &, "& ref2.Value)

    End Sub
End Module
...
```

## 2.5.7 Custom Attributes

To define custom attributes for a class, an interface, a variable, a parameter or a method, you have to use the Custom attributes extended attribute in PowerDesigner. You can use the Custom attributes input box to type all the custom attributes you wish to add using the correct VB .NET syntax.



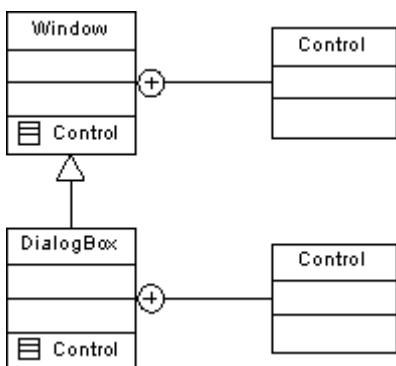
## Custom Attributes for Return Types

You can use the Return type custom attribute extended attribute to define custom attributes for the return type of a property attribute or a method.

### 2.5.8 Shadows

Shadows indicates that an inherited element hides a parent element with the same name. To design a shadows class or interface, you have to set the class or interface Shadows extended attribute to True.

In the following example, class DialogBox inherits from class Window. Class Window contains an inner classifier Control, and so does class DialogBox. You do not want class DialogBox to inherit from the control defined in Window, to do so, you have to set the Shadows extended attribute to True, in the Control class inner to DialogBox:



```

Public Class DialogBox
    Inherits Window
    Public Shadows Class Control
    End Class
End Class
...

```

## 2.5.9 Variables

You design a VB .NET variable using an attribute in PowerDesigner.

The following table summarizes the different types of VB .NET variables and attributes supported in PowerDesigner:

| VB .NET variable    | PowerDesigner equivalent                                 |
|---------------------|--|
| ReadOnly variable   | Attribute with <i>Read-only</i> changeability            |
| Const variable      | Attribute with <i>Frozen</i> changeability               |
| Shared variable     | Attribute with <i>Static</i> property selected           |
| Shadowing variable  | Extended attribute Shadowing set to Shadows or Overloads |
| WithEvents variable | Attribute with <i>withEvents</i> stereotype              |
| Overrides variable  | Extended attribute Overrides set to True                 |
| New variable        | Extended attribute AsNew set to True                     |

- Data Type: You define the data type of a variable using the attribute Data Type property
- Initial Value: You define the initial value of a variable using the attribute Initial Value property
- Shadowing: To define a shadowing by name set the Shadowing extended attribute to Shadows. To define a shadowing by name and signature set the Shadowing extended attribute to Overloads. See [Method \[page 384\]](#) for more details on shadowing

## 2.5.10 Property

To design a VB .NET property you have to design an attribute with the <<Property>> stereotype, another attribute with the <<PropertyImplementation>> stereotype is automatically created, it is displayed with an underscore sign in the list of attributes. The corresponding getter and setter operations are also automatically created.

You can get rid of the implementation attribute.

If you remove the getter operation, the *ReadOnly* keyword is automatically generated. If you remove the setter operation, the *WriteOnly* keyword is automatically generated. If you remove both getter and setter operations, the attribute no longer has the <<Property>> stereotype.

When you define a <<Property>> attribute, the attribute changeability and the getter/setter operations are tightly related as explained in the following table:

| Operations  | Property attribute changeability |
|---|----------------------------------|
| If you keep both getter and setter operations               | Property is Changeable           |
| If you remove the setter operation of a changeable property | Property becomes Read-only       |
| If you remove the getter operation of a changeable property | Property becomes Write-only      |

On the other hand, if you modify the property changeability, operations will reflect this change, for example, if you turn a changeable property into a read-only property, the setter operation is automatically removed.

In the following example, class Button contains a property Caption. The Getter operation has been removed which causes the WriteOnly keyword to appear in the property declaration line:

| Button                                |
|---------------------------------------|
| - captionValue : String               |
| + <<Property>> Caption : String       |
| + <<Setter>> SetCaption(String Value) |

```

Public Class Button
    Private captionValue As String
    Public WriteOnly Property Caption() As String
        Set (ByVal Value As String)
            captionValue = value
            Repaint()
        End Set
    End Property
End Class

```

- Must override: Set the Must override extended attribute of the property to True to express that the property in a base class must be overridden in a derived class before it can be used
- Overridable: Set the Overridable extended attribute of the property to True to express that the property can be overridden in a derived class
- Overrides: Set the Overrides extended attribute of the property to True to express that a property overrides a member inherited from a base class
- Parameters: Type a value in the value box of the Property parameters extended attribute to specify which value of the property attribute is to be used as parameter. In the following example, class Person contains property attribute ChildAge. The parameter used to sort the property is ChildName:

| Person   |
|--|
| - <<Property>> ChildAge : Integer                |
| - <<PropertyImplementation>> _ChildAge : Integer |
| + <<Setter>> set_ChildAge (Integer newChildAge)  |
| + <<Getter>> get_ChildAge()                      |

```

Public Class Person
    Private _ChildAge As Integer

    Private Property ChildAge(ChildName as String) As Integer
        Get
            return _ChildAge
        End Get
        Set (ByVal Value ChildAge As Integer)
            If (_ChildAge <> newChildAge)
                _ChildAge = newChildAge
            End If
        End Set
    End Property
End Class

```

## 2.5.11 Method

You design a VB .NET method using an operation. Methods can be functions or subs.

You design a function using an operation with a return value.

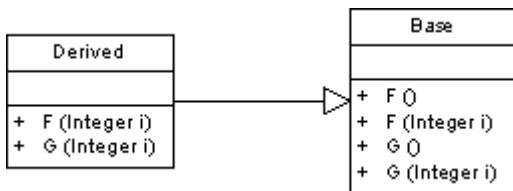
You design a sub using an operation with an empty return type.

The following table summarizes the different methods supported in PowerDesigner:

| VB .NET method                | PowerDesigner equivalent  |
|-------------------------------|---|
| Shadowing or Overloads method | Select <i>Shadows</i> or <i>Overloads</i> from the <i>Shadowing</i> list on the <i>VB.NET</i> tab of the operation property sheet |
| Shared method                 | Select the <i>Static</i> check box on the <i>General</i> tab of the operation property sheet                                      |
| NotOverridable method         | Select the <i>Final</i> check box on the <i>General</i> tab of the operation property sheet                                       |
| Overridable method            | Select the <i>Overridable</i> check box on the <i>VB.NET</i> tab of the operation property sheet                                  |
| MustOverride method           | Select the <i>Abstract</i> check box on the <i>General</i> tab of the operation property sheet                                    |
| Overrides method              | Select the <i>Overrides</i> check box on the <i>VB.NET</i> tab of the operation property sheet                                    |

## Shadowing

To define a shadowing by name, select *Shadows* from the *Shadowing* list on the *VB.NET* tab of the operation property sheet. To define a shadowing by name and signature select *Overloads*. In the following example, class Derived inherits from class Base:



Operation F in class Derived overloads operation F in class Base; and operation G in class Derived shadows operation G in class Base:

```

Public Class Derived
Inherits Base
Public Overloads Sub F(ByVal i As Integer)
End Sub
Public Shadows Sub G(ByVal i As Integer)
End Sub
End Class

```

## Method Parameters

You define VB .NET method parameters using operation parameters.

You can define the following parameter modifiers in PowerDesigner:

| VB .NET modifier | PowerDesigner equivalent   |
|------------------|--|
| ByVal            | Select In in the Parameter <i>Type</i> box on the parameter property sheet <i>General</i> tab            |
| ByRef            | Select In/Out or Out in the <i>Parameter Type</i> box on the parameter property sheet <i>General</i> tab |
| Optional         | Set the Optional extended attribute on the <i>Extended Attributes</i> tab to True                        |
| ParamArray       | Select the Variable Argument checkbox on the parameter property sheet <i>General</i> tab                 |

## Method Implementation

Class methods are implemented by the corresponding interface operations. To define the implementation of the methods of a class, you have to use the To be implemented button in the *Operations* tab of a class property sheet, then click the *Implement* button for each method to implement. The method is displayed with the <<Implement>> stereotype.

## 2.5.12 Constructor & Destructor

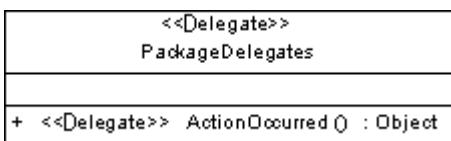
You design VB .NET constructors and destructors by clicking the button in the list of operations of a class. This automatically creates a constructor called New with the Constructor

stereotype, and a destructor called Finalize with the Destructor stereotype. Both constructor and destructor are grayed out in the list, which means you cannot modify their definition, but you can still remove them from the list.

## 2.5.13 Delegate

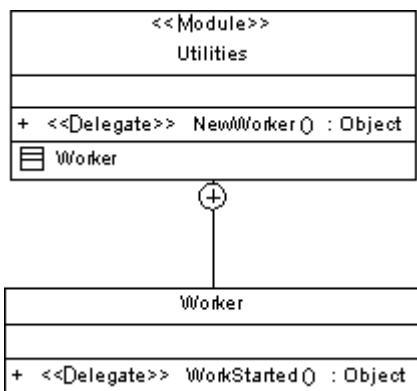
You can design the following types of VB .NET delegates:

- To create a delegate at the namespace level, create a class with the <<Delegate>> stereotype, and add an operation with the <<Delegate>> stereotype to this class and define a visibility for this operation. This visibility becomes the visibility of the delegate



```
...
Public Delegate Function ActionOccurred () As Object
...
```

- To create a delegate in a class, module, or structure, you just have to create an operation with the <<Delegate>> stereotype. In the following example, class Worker is inner to module Utilities. Both contain internal delegates designed as operations with the <<Delegate>> stereotype



```
...
Public Module Utilities
    Public Delegate Function NewWorker () As Object
    Public Class Worker
        Public Delegate Function WorkStarted () As Object
    End Class
End Module
...
```

## 2.5.14 Event

To define an event in VB .NET you must declare its signature. You can either use a delegate as a type for this event or define the signature on the event itself. Both declarations can be mixed in a class.

The delegate used as a type is represented by an attribute with the <<Event>> stereotype. You define the delegate name using the attribute data type.

| Printer                                 |
|---|
| - <<Event>> PaperJam : EventHandler     |
| - <<Event>> OutOfPaper : EventHandler   |
| - <<Event>> JobOK : PrinterGoodDelegate |
|   |

```
Public Class Printer
    Public PaperJam As EventHandler
    Public OutOfPaper As EventHandler
    Public JobOK As PrinterGoodDelegate
End Class
```

When you define the signature on the event itself, you have to use an operation with the <<Event>> stereotype. The signature of this operation then becomes the signature of the event.

| Printer   |
|---|
|   |
| + <<event>> PaperJam(Printer p, EventArgs e) : Object |
| + <<event>> JobOK(Object p) : Object                  |

```
Public Class Printer
    Public Event PaperJam(ByVal p As Printer, ByVal e As EventArgs)
    Public Event JobOK(ByVal p As Object)
End Class
```

## Event Implementation

To design the implementation clause of a delegate used as a type you have to type a clause in the implements extended attribute of the <<Event>> attribute.

For <<Event>> operations, you have to use the To Be Implemented feature in the list of operations of the class.

## 2.5.15 Event Handler

To define a VB .NET event handler you should already have an operation with the <<event>> stereotype in your class. You then have to create another operation, and type the name of the <<event>> operation in the Handles extended attribute Value box.

|  |
|--|
| Printer  |
|  |
| + <<event>> Print() : Object<br>+ Operation_2() : Object |

```
...  
Public Function Operation_2() As Object Handles Print  
End Function  
...
```

## 2.5.16 External Method

You define a VB .NET external method using an operation with the <<External>> stereotype. External methods share the same properties as standard methods.

You can also define the following specific properties for an external method:

- Alias clause: you can use the Alias name extended attribute to specify numeric ordinal (prefixed by a @ character) or a name for an external method
- Library clause: you can use the Library name extended attribute to specify the name of the external file that implements the external method
- ANSI, Unicode and Automodifiers used for calling the external method can be defined using the Character set extended attribute of the external method

## 2.5.17 Generating VB.NET Files

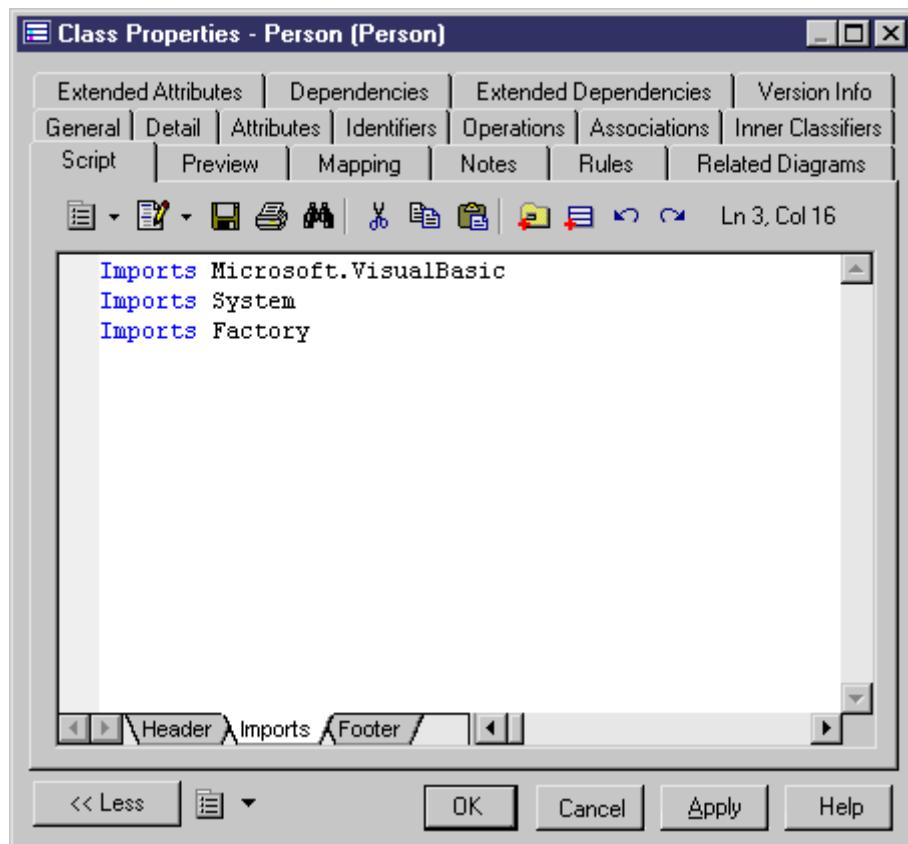
You generate VB.NET source files from the classes and interfaces of a model. A separate file, with the file extension .vb, is generated for each class or interface that you select from the model, along with a generation log file.

### Context

During VB .NET generation, each top object, that is to say class, interface, module, and so on, generates a source file with the .vb extension. Inner classifiers are generated in the source of the container classifier.

The Imports directive can appear at the beginning of the script of each generated file.

You can define imports in PowerDesigner in the *Script\Imports* sub-tab of the property sheet of a main object. You can type the import statement or use the *Import Folder* or *Import Classifier* tools in the *Imports* sub-tab.



Options appear in the generated file header. You can define the following options for main objects:

- Compare: type the value Text or Binary in the value box of the Compare extended attribute of the generated top object
- Explicit: select True or False in the value box of the Explicit extended attribute of the generated top object
- Strict: select True or False in the in the value box of the Strict extended attribute of the generated top object

The following PowerDesigner variables are used in the generation of VB.NET source files:

| Variable | Description   |
|----------|---|
| VBC      | VB .NET compiler full path. For example, C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705\vbc.exe                               |
| WSDL     | Web Service proxy generator full path. For example, C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Bin\wsdl.exe |

To review or edit these variables, select **Tools > General Options** and click the **Variables** category.

## Procedure

1. Select  [Language](#)  [Generate VB.NET Code](#) to open the VB.NET Generation dialog.
2. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see [Checking an OOM \[page 251\]](#)).
3. [optional] Select any additional targets to generate for. These targets are defined by any extensions that may be attached to your model (see [Working with Generation Targets \[page 236\]](#)).
4. [optional] Click the [Selection](#) tab and specify the objects that you want to generate from. By default, all objects are generated.
5. [optional] Click the [Options](#) tab and set any appropriate generation options:

| Options   | Description   |
|---|---|
| Generate VB .NET Web Service code in .ASMX file instead of .VB file | Generates the Visual Basic code in the .ASMX file   |
| Generate Visual Studio .NET project files                           | Generates the files of the Visual Studio .NET project. A solution file is generated together with several project files, each project corresponding to a model or a package with the <>Assembly>> stereotype      |
| Generate object ids as documentation tags                           | Generates information used for reverse engineering like object identifiers (@pdoid) that are generated as documentation tags. If you do not want these tags to be generated, you have to set this option to False |
| Visual Studio .NET version  | Indicates the version number of Visual Studio .NET  |

### Note

For information about modifying the options that appear on this and the [Tasks](#) tab and adding your own options and tasks, see *Customizing and Extending PowerDesigner > Object, Process, and XML Language Definition Files > Generation Category* .

6. [optional] Click the [Generated Files](#) tab and specify which files will be generated. By default, all files are generated.

For information about customizing the files that will be generated, see *Customizing and Extending PowerDesigner > Extension Files > Generated Files (Profile)* .

7. [optional] Click the [Tasks](#) tab and specify any appropriate generation tasks to perform:

| Task                                   | Description               |
|--|---------------------------|
| Generate Web service proxy code (WSDL) | Generates the proxy class |
| Compile Visual Basic .NET source files | Compiles the source files |

| Task                                    | Description   |
|---|---|
| Open the solution in Visual Studio .NET | If you selected the Generate Visual Studio .NET project files option, this task allows to open the solution in the Visual Studio .NET development environment |

- Click **OK** to begin generation.

When generation is complete, the Generated Files dialog opens, listing the files that have been generated to the specified directory. Select a file in the list and click **Edit** to open it in your associated editor, or click **Close** to exit the dialog.

## 2.5.18 Reverse Engineering VB .NET

You can reverse engineer VB .NET files into an OOM.

In the **Selection** tab, you can select to reverse engineer files, directories or projects.

You can also define a base directory. The base directory is the common root directory for all the files to reverse engineer. This base directory will be used during regeneration to recreate the exact file structure of the reverse engineered files.

### Edit Source

You can right-click the files to reverse engineer and select the **Edit** command to view the content of your files. To use this command you have to associate the file extension with an editor in the General Options\Editor dialog box.

### 2.5.18.1 Selecting VB .NET Reverse Engineering Options

You define the following VB .NET reverse engineering option from the Reverse Engineer VB .NET dialog box:

| Option   | Result of selection   |
|--|---|
| File encoding  | Allows you to modify the default file encoding of the files to reverse engineer                           |
| Ignore operation body                                | Reverses classes without including the body of the code   |
| Ignore comments                                      | Reverses classes without including code comments  |
| Create Associations from classifier-typed attributes | Creates associations between classes and/or interfaces  |
| Create symbols                                       | Creates a symbol for each object in the diagram, if not, reversed objects are only visible in the browser |

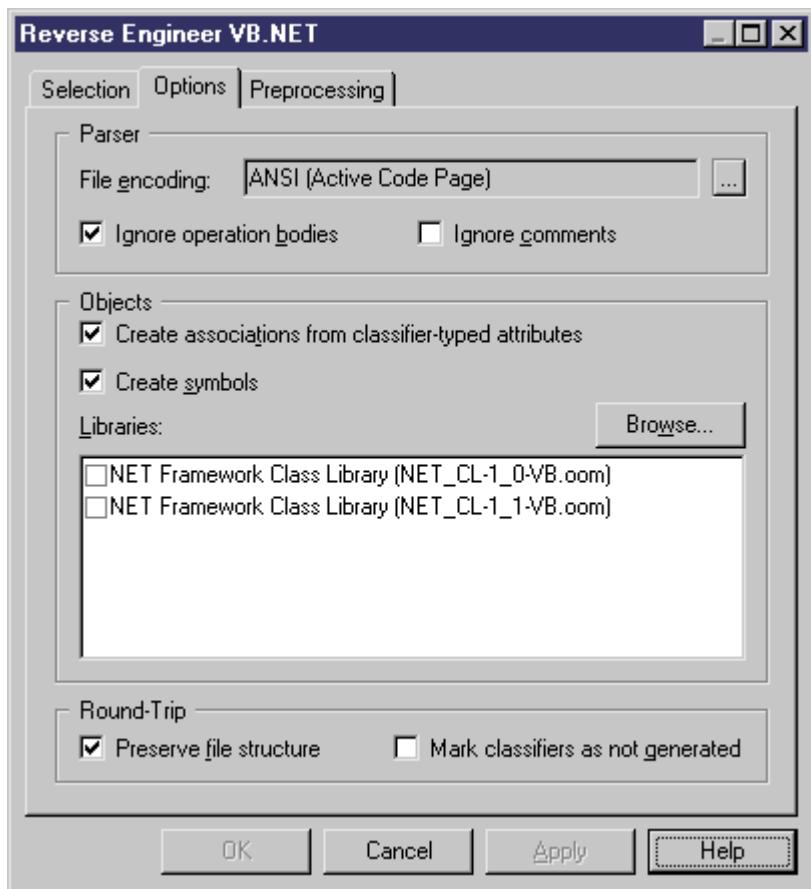
| Option                               | Result of selection  |
|--------------------------------------|--|
| Libraries                            | <p>Specifies a list of library models to be used as references during reverse engineering.</p> <p>The reverse engineered model may contain shortcuts to objects defined in a library. If you specify the library here, the link between the shortcut and its target object (in the library) will be preserved and the library will be added to the list of target models in the reverse engineered model.</p> <p>You can drag and drop the libraries in the list in order to specify a hierarchy among them. PowerDesigner will seek to resolve shortcuts found in the reverse engineered model against each of the specified libraries in turn. Thus, if library v1.1 is displayed in the list above library v1.0, PowerDesigner will first attempt to resolve shortcuts against library v1.1 and will only parse library v1.0 if unresolved shortcuts remain.</p> <p>You should use the List of Target Models to manage libraries related to the reverse engineered model, for example, you can change the library version (see <i>Core Features Guide &gt; Linking and Synchronizing Models &gt; Shortcuts and Replicas &gt; Working with Target Models</i>).</p> |
| Preserve file structure              | Creates an artifact during reverse engineering in order to be able to regenerate an identical file structure   |
| Mark classifiers not to be generated | Reversed classifiers (classes and interfaces) will not be generated from the model. To generate the classifier, you must select the <i>Generate</i> check box in its property sheet  |

## 2.5.18.1.1 Defining VB .NET Reverse Engineering Options

To define VB .NET reverse engineering options:

### Procedure

1. Select  [Language](#)  [Reverse Engineer VB .NET](#) .
2. Click the Options tab to display the Options tab.



3. Select or clear reverse engineering options.
4. Browse to the Library directory, if required.
5. Click Apply and Cancel.

## 2.5.18.2 VB .NET Reverse Engineering Preprocessing

VB .NET files may contain conditional code that needs to be handled by preprocessing directives during reverse engineering. A preprocessing directive is a command placed within the source code that directs the compiler to do a certain thing before the rest of the source code is parsed and compiled. The preprocessing directive has the following structure:

```
#directive symbol
```

Where # is followed by the name of the directive, and symbol is a conditional compiler constant used to select particular sections of code and exclude other sections.

In VB .NET symbols have values.

In the following example, the #if directive is used with symbols FrenchVersion and GermanVersion to output French or German language versions of the same application from the same source code:

```
#if FrenchVersion Then
```

```

' <code specific to French language version>.
#ElseIf GermanVersion Then
    ' <code specific to French language version>.
#Else
    ' <code specific to other language version>.
#End If

```

You can declare a list of symbols for preprocessing directives. These symbols are parsed by preprocessing directives: if the directive condition is true the statement is kept, otherwise the statement is removed.

## 2.5.18.2.1 VB .NET Supported Preprocessing Directives

The following directives are supported during preprocessing:

| Directive | Description   |
|-----------|---|
| #Const    | Defines a symbol  |
| #If       | Evaluates a condition, if the condition is true, the statement following the condition is kept otherwise it is ignored  |
| #Else     | If the previous #If test fails, source code following the #Else directive will be included  |
| #Else If  | Used with the #if directive, if the previous #If test fails, #Else If includes or exclude source code, depending on the resulting value of its own expression or identifier |
| #End If   | Closes the #If conditional block of code  |

Note: #Region, #End Region, and #ExternalSource directives are removed from source code.

## 2.5.18.2.2 Defining a VB .NET Preprocessing Symbol

You can define VB .NET preprocessing symbols and values in the preprocessing tab of the reverse engineering dialog box.

### Context

Symbol names are not case sensitive but they must be unique. Make sure you do not type reserved words like true, false, if, do and so on. You must always assign a value to a symbol, this value can be a string (no " " required), a numeric value, a boolean value or Nothing.

The list of symbols is saved in the model and will be reused when you synchronize your model with existing code using the Synchronize with Generated Files command.

For more information on the Synchronize with Generated Files command see [Synchronizing a Model with Generated Files \[page 240\]](#).

You can use the Set As Default button to save the list of symbols in the registry.

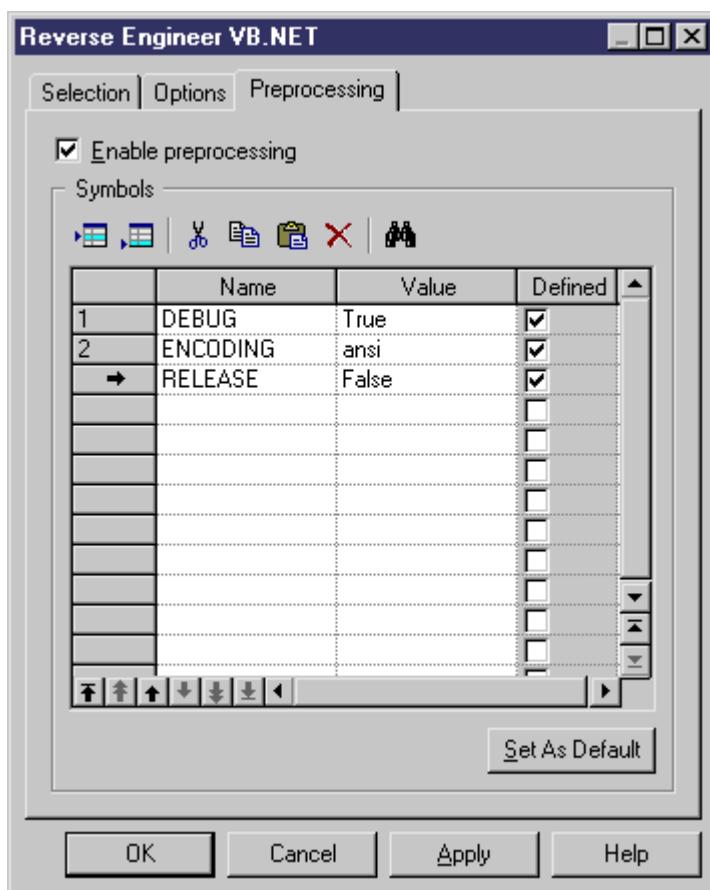
## Procedure

1. Select  *Language* > *Reverse engineering VB .NET* >.

The Reverse Engineering VB .NET dialog box is displayed.

2. Click the *Preprocessing* tab, then click the *Add a Row* tool to insert a line in the list.
  3. Type symbol names in the *Name* column.
  4. Type symbol value in the *Value* column.

The *Defined* check box is automatically selected for each symbol to indicate that the symbol will be taken into account during preprocessing.



5. Click **Apply**.

### **2.5.18.2.3 VB .NET Reverse Engineering with Preprocessing**

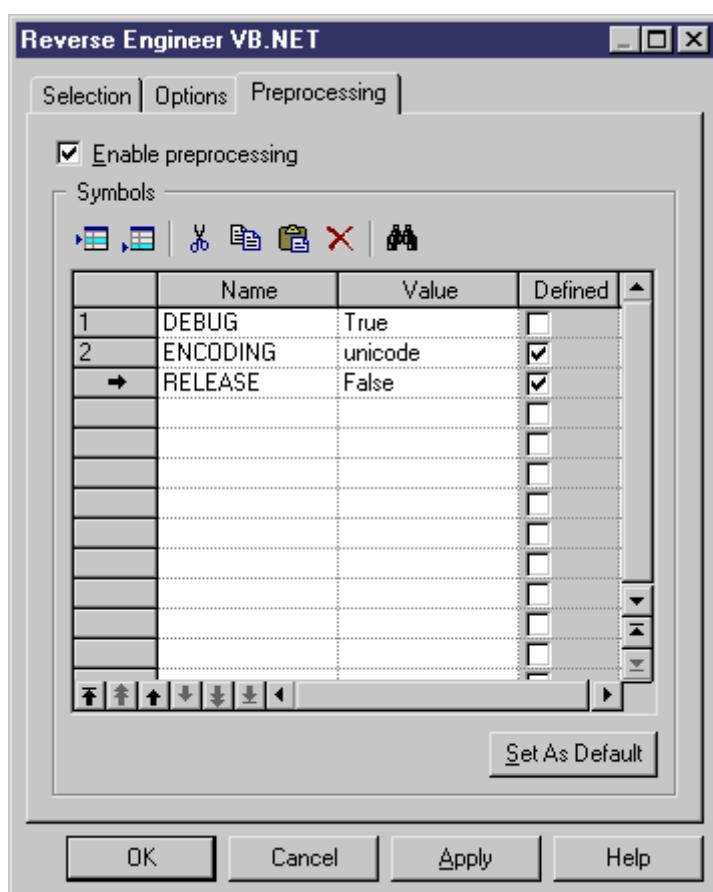
Preprocessing is a reverse engineering option you can enable or disable.

## Procedure

1. Select  *Language*  *Reverse engineering VB .NET* .

The Reverse Engineering VB .NET dialog box is displayed.

  2. Select files to reverse engineer in the *Selection* tab.
  3. Select reverse engineering options in the *Options* tab.
  4. Select the *Enable preprocessing* check box in the *Preprocessing* tab.
  5. Select symbols in the list of symbols.



6. Click **OK** to start reverse engineering.

When preprocessing is over the code is passed to reverse engineering.

### 2.5.18.3 Reverse Engineering VB .NET Files

You can reverse engineer VB .NET files.

#### Procedure

1. Select  [Language](#) > [Reverse Engineer VB .NET](#) to display the Reverse Engineer VB .NET dialog box.
2. Select to reverse engineer files or directories from the [Reverse Engineer](#) list.
3. Click the [Add](#) button in the [Selection](#) tab.  
A standard Open dialog box is displayed.
4. Select the items or directory you want to reverse engineer.

##### Note

You select several files simultaneously using the `Ctrl` or `Shift` keys. You cannot select several directories.

The Reverse VB .NET dialog box displays the files you selected.

5. Click [OK](#).

A progress box is displayed. If the model in which you are reverse engineering already contains data, the Merge Models dialog box is displayed.

For more information on merging models, see *Core Features Guide > Modeling with PowerDesigner > Comparing and Merging Models*.

The classes are added to your model. They are visible in the diagram and in the Browser. They are also listed in the [Reverse](#) tab of the Output window, located in the lower part of the main window.

### 2.5.19 Working with ASP.NET

An Active Server Page (ASP) is an HTML page that includes one or more scripts (small embedded programs) that are interpreted by a script interpreter (such as VBScript or JScript) and that are processed on a Microsoft Web server before the page is sent to the user. An ASP involves programs that run on a server, usually tailoring a page for the user. The script in the Web page at the server uses input received as the result of the user's request for the page to access data from a database and then builds or customizes the page on the fly before sending it to the requestor.

ASP.NET (also called ASP+) is the next generation of Microsoft Active Server Page (ASP). Both ASP and ASP.NET allow a Web site builder to dynamically build Web pages on the fly by inserting queries to a relational database in the Web page. ASP.NET is different than its predecessor in two major ways:

- It supports code written in compiled languages such as Visual Basic, VB .NET, and Perl
- It features server controls that can separate the code from the content, allowing WYSIWYG editing of pages

ASP.NET files have a .ASPX extension. In an OOM, an ASP.NET is represented as a file object and is linked to a component (of type ASP.NET). The component type Active Server Page (ASP.NET) allows you to identify this component. Components of this type are linked to a single file object that defines the page.

When you set the type of the component to ASP.NET, the appropriate ASP.NET file object is automatically created, or attached if it already exists. You can see the ASP.NET file object from the Files tab in the component property sheet.

## 2.5.19.1 ASP Tab of the Component

When you set the type of the component to ASP.NET, the [ASP](#) tab is automatically displayed in the component property sheet.

The [ASP](#) tab includes the following properties:

| Property         | Description   |
|------------------|---|
| ASP file         | File object that defines the page. You can click the <a href="#">Properties</a> tool beside this box to display the property sheet of the file object, or click the <a href="#">Create</a> tool to create a file object |
| Default template | Extended attribute that allows you to select a template for generation. Its content can be user defined or delivered by default   |

To modify the default content, edit the current object language from [Language](#) [Edit Current Object Language](#) and modify the following item: Profile/FileObject/Criteria/ASP/Templates/DefaultContent<%is(DefaultTemplate)%>. Then create the templates and rename them as DefaultContent<%is(<name>)%> where <name> stands for the corresponding DefaultContent template name.

To define additional DefaultContent templates for ASP.NET, you have to modify the ASPTemplate extended attribute type from Profile/Share/Extended Attribute Types and add new values corresponding to the new templates respective names.

For more information on the default template property, see the definition of TemplateContent in [Creating an ASP.NET with the Wizard \[page 399\]](#).

## 2.5.19.2 Defining File Objects for ASP.NET

The file object content for ASP is based on a special template called DefaultContent defined with respect to the FileObject metaclass. It is located in the Profile/FileObject/Criteria/ASP/Templates category of the C# and VB.NET object languages. This link to the template exists as a basis, therefore if you edit the file object, the link to the template is lost - the mechanism is similar to that of operation default bodies.

For more information on the Criteria category, see [Customizing and Extending PowerDesigner > Extension Files > Criteria \(Profile\)](#).

Active Server Page files are identified using the ASPFile stereotype. The server page name is synchronized with the ASP.NET component name following the convention specified in the Value box of the Settings/Namings/ASPFileName entry of the C# and VB.NET object languages.

You can right-click a file object, and select *Open With* <text editor> from the contextual menu to display the content of the file object.

### 2.5.19.3 Creating an ASP.NET with the Wizard

You can create an ASP.NET with the wizard that will guide you through the creation of the component. The wizard is invoked from a class diagram. It is only available if the language is C# or VB.NET.

#### Context

You can either create an ASP.NET without selecting any file object, or select a file object beforehand and start the wizard from the Tools menu.

You can also create several ASP.NET of the same type by selecting several file objects at the same time. The wizard will automatically create one ASP.NET per file object: the file objects you have selected in the class diagram become ASP.NET files.

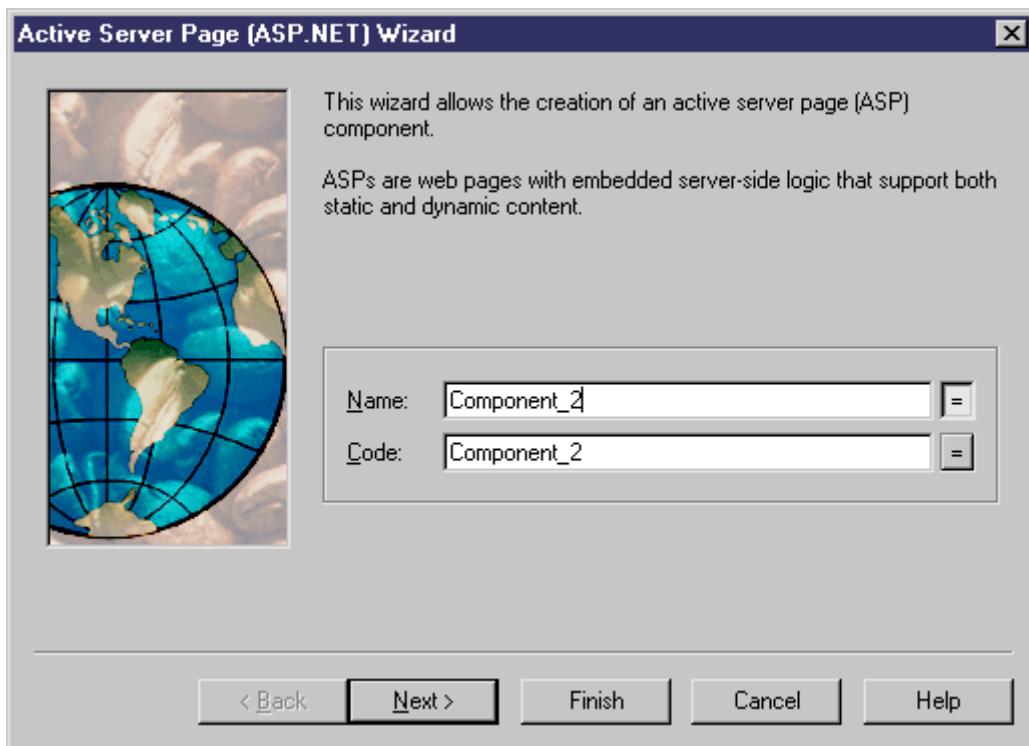
The wizard for creation of an ASP.NET lets you define the following parameters:

| Wizard page     | Description  |
|-----------------|--|
| Name            | Name of the ASP.NET component  |
| Code            | Code of the ASP.NET component  |
| TemplateContent | Allows you to choose the default template of the ASP.NET file object. The TemplateContent is an extended attribute located in the Profile/Component/Criteria/ASP category of the C# and VB.NET object languages. If you do not modify the content of the file object, the default content remains (see the Contents tab of the file object property sheet). All templates are available in the FileObject/Criteria/ASP/templates category of the current object language |
| Create symbol   | Creates a component symbol in the diagram specified beside the Create symbol In check box. If a component diagram already exists, you can select one from the list. You can also display the diagram properties by selecting the Properties tool   |

#### Procedure

1. Select *Tools* *Create ASP* from a class diagram.

The Active Server Page Wizard dialog box is displayed.



2. Select a name and code for the ASP.NET component and click Next.
3. Select an ASP.NET template and click Next.
4. At the end of the wizard, you have to define the creation of symbols.

## Results

When you have finished using the wizard, the following actions are executed:

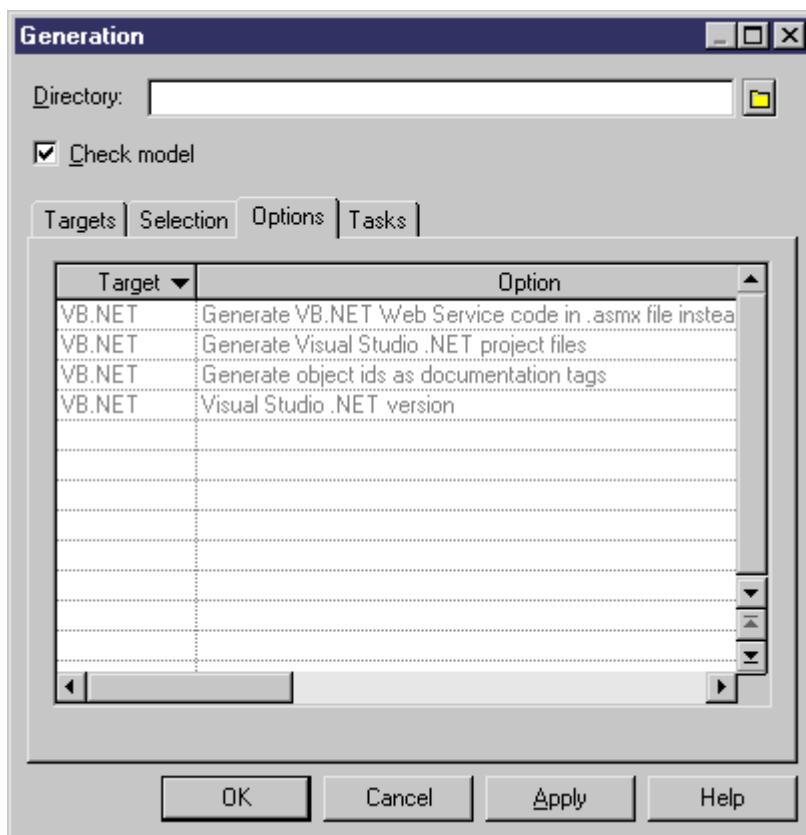
- An ASP.NET component and a file object with an extension .ASPX are created and visible in the Browser. The file object is named after the original default component name to preserve coherence
- If you open the property sheet of the file object, you can see that the Artifact property is selected  
For more information on artifact file objects, see [File Object Properties \[page 208\]](#).
- You can edit the file object directly in the internal editor of PowerDesigner, if its extension corresponds to an extension defined in the Editors page of the General Options dialog box, and if the <internal> keyword is defined in the Editor Name and Editor Command columns for this extension

## 2.5.19.4 Generating ASP.NET

The generation process generates only file objects having the Artifact property selected.

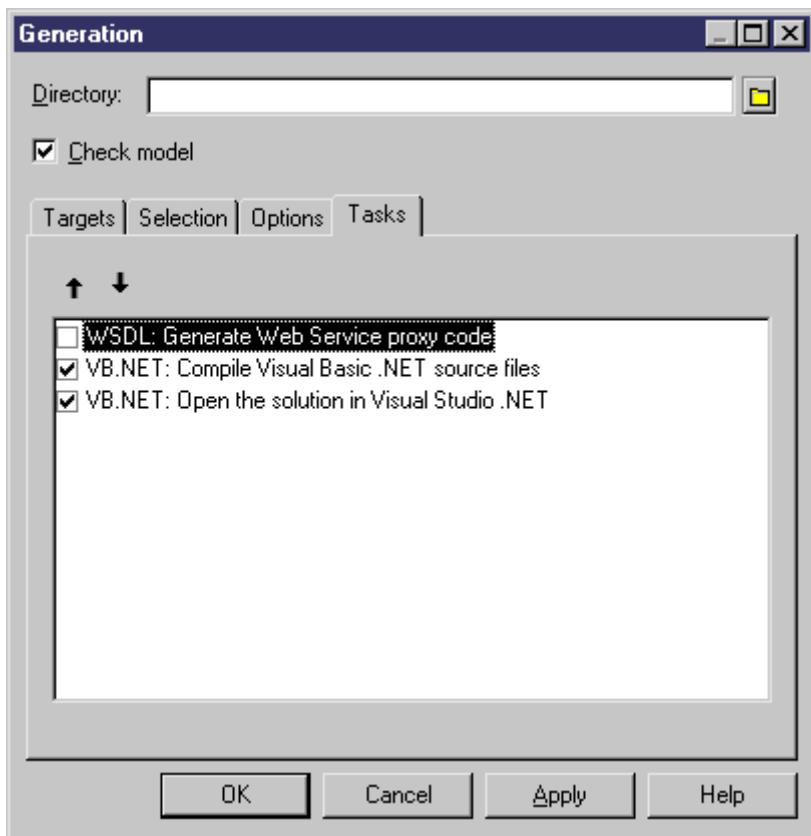
### Procedure

1. Select ► *Language* ► *Generate C# or VB.NET code* ▶ to display the Generation dialog box.
2. Select or browse to a directory that will contain the generated files.
3. Click the *Selection* tab, then select the objects you need in the different sub-tabs.
4. Click *Apply*.
5. Click the *Options* tab, then specify your generation options.



For more information on the generation options, see [Generating VB.NET Files \[page 388\]](#).

6. Click *Apply*.
7. Click the *Tasks* tab.
8. Select the commands you want to perform during generation in the *Tasks* tab.



For more information on the generation tasks, see [Generating VB.NET Files \[page 388\]](#).

You must beforehand set the environment variables from General Options diaog box (Variables section) in order to activate them in this tab.

For more information on how to set these variables, see *Core Features Guide > Modeling with PowerDesigner > Customizing Your Modeling Environment > General Options > Environment Variables*.

9. Click [OK](#).

A progress box is displayed, followed by a Result list. You can use the [Edit](#) button in the Result list to edit the generated files individually.

10. Click [Close](#).

The files are generated in the generation directory.

## 2.6 C# 2.0

PowerDesigner provides full support for modeling all aspects of C# 2.0 including round-trip engineering.

C# 2.0 is a modern, type-safe, object-oriented language that combines the advantages of rapid development with the power of C++.

In addition to PowerDesigner's standard pallets, the following custom tools are available to help you rapidly develop your class and composite structure diagrams:

| Icon | Tool   |
|------|--|
|      | Assembly – a collection of C# files (see <a href="#">C# 2.0 Assemblies [page 403]</a> ).           |
|      | Custom Attribute – for adding metadata (see <a href="#">C# 2.0 Custom Attributes [page 419]</a> ). |
|      | Delegate – type-safe reference classes (see <a href="#">C# 2.0 Delegates [page 410]</a> ).         |
|      | Enum – sets of named constants (see <a href="#">C# 2.0 Enums [page 411]</a> ).                     |
|      | Struct – lightweight types (see <a href="#">C# 2.0 Structs [page 409]</a> ).                       |

## 2.6.1 C# 2.0 Assemblies

An assembly is a collection of C# files that forms a DLL or executable. PowerDesigner provides support for both single-assembly models (where the model represents the assembly) and multi-assembly models (where each assembly appears directly below the model in the Browser tree and is modeled as a standard UML package with a stereotype of <>Assembly<>).

### Creating an Assembly

PowerDesigner supports both single-assembly and multi-assembly models.

By default, when you create a C# 2.0 OOM, the model itself represents an assembly. To continue with a single-assembly model, insert a type or a namespace in the top-level diagram. The model will default to a single-module assembly, with the model root representing the assembly.

To create a multi-assembly model, insert an assembly in the top-level diagram in any of the following ways:

- Use the [Assembly](#) tool in the C# 2.0 Toolbox.
- Select [Model](#) [Assembly Objects](#) to access the List of Assembly Objects, and click the [Add a Row](#) tool.
- Right-click the model (or a package) in the Browser, and select [New](#) [Assembly](#).

#### Note

If these options are not available to you, then you are currently working with a single-assembly model.

## Converting a Single-Assembly Model to a Multi-Assembly Model

To convert to a multi-assembly model, right-click the model in the Browser and select [Convert to Multi-Assembly Model](#), enter a name for the assembly that will contain all the types in your model in the Create an Assembly dialog, and click [OK](#).

PowerDesigner converts the single-assembly model into a multi-assembly model by inserting a new assembly directly beneath the model root to contain all the types present in the model. You can add new assemblies as necessary but only as direct children of the model root.

## Assembly Properties

Assembly property sheets contains all the standard package tabs along with the following C#-specific tabs:

The [Application](#) tab contains the following properties:

| Property              | Description   |
|-----------------------|---|
| Generate Project File | Specifies whether to generate a Visual Studio 2005 project file for the assembly.   |
| Project Filename      | Specifies the name of the project in Visual Studio. The default is the value of the assembly code property.   |
| Assembly Name         | Specifies the name of the assembly in Visual Studio. The default is the value of the assembly code property.  |
| Root Namespace        | Specifies the name of the root namespace in Visual Studio. The default is the value of the assembly code property.  |
| Output Type           | Specifies the type of application being designed. You can choose between: <ul style="list-style-type: none"><li>• Class library</li><li>• Windows Application</li><li>• Console Application</li></ul> |
| Project GUID          | Specifies a unique GUID for the project. This field will be completed automatically at generation time.   |

The [Assembly Information](#) tab contains the following properties:

| Property                      | Description   |
|-------------------------------|---|
| Generate Assembly Information | Specifies whether to generate an assembly manifest file.  |
| Title                         | Specifies a title for the assembly manifest. This field is linked to the Name field on the <a href="#">General</a> tab. |
| Description                   | Specifies an optional description for the assembly manifest.  |

| Property                  | Description   |
|---------------------------|---|
| Company                   | Specifies a company name for the assembly manifest.   |
| Product                   | Specifies a product name for the assembly manifest.   |
| Copyright                 | Specifies a copyright notice for the assembly manifest.   |
| Trademark                 | Specifies a trademark for the assembly manifest.  |
| Culture                   | Specifies which culture the assembly supports.  |
| Version                   | Specifies the version of the assembly.  |
| File Version              | Specifies a version number that instructs the compiler to use a specific version for the Win32 file version resource. |
| GUID                      | Specifies a unique GUID that identifies the assembly.   |
| Make assembly COM-Visible | Specifies whether types within the assembly will be accessible to COM.  |

## 2.6.2 C# 2.0 Compilation Units

By default, PowerDesigner generates one source file for each class, interface, delegate, or other type, and bases the source directory structure on the namespaces defined in the model.

You may want instead to group multiple classifiers in a single source file and/or construct a directory structure independent of your namespaces.

A compilation-unit allows you to group multiple types in a single source file. It consists of zero or more using-directives followed by zero or more global-attributes followed by zero or more namespace-member-declarations.

PowerDesigner models compilation units as artifacts with a stereotype of <<Source>> and allows you to construct a hierarchy of source directories using folders. Compilation units do not have diagram symbols, and are only visible inside the Artifacts folder in the Browser.

You can preview the code that will be generated for your compilation unit at any time, by opening its property sheet and clicking the *Preview* tab.

### Creating a Compilation Unit

To create an empty compilation unit from the Browser, right-click the model or the Artifacts folder and select **► New ► Source**, enter a name (being sure to retain the .cs extension), and then click **OK**.

### Note

You can create a compilation unit and populate it with a type from the *Generated Files* tab of the property sheet of the type by clicking the *New* tool in the *Artifacts* column.

## Adding a Type to a Compilation Unit

You can add types to a compilation unit by:

- Dragging and dropping the type diagram symbol or browser entry onto the compilation unit browser entry.
- Opening the compilation unit property sheet to the *Objects* tab and using the *Add Production Objects* tool.
- Opening the type property sheet to the *Generated Files* tab and using the *Add/Remove* tool in the *Artifacts* column. Types that are added to multiple compilation units will be generated as partial types and you can specify the compilation unit in which each of their attributes and methods will be generated.

## Creating a Generation Folder Structure

You can control the directory structure in which your compilation units will be generated by using artifact folders:

1. Right-click the model or a folder inside the Browser Artifacts folder, and select  *New*  *Artifact Folder*.
2. Specify a name for the folder, and then click *OK* to create it.
3. Add compilation units to the folder by dragging and dropping their browser entries onto the folder browser entry, or by right-clicking the folder and selecting  *New*  *Source*.

### Note

Folders can only contain compilation units and other folders. To place a type in the generation folder hierarchy, you must first add it to a compilation unit.

### 2.6.2.1 Partial Types

Partial types are types that belong to more than one compilation unit. They are prefixed with the *partial* keyword.

```
public partial class Server
{
    private int start;
}
```

In this case, you can specify to which compilation unit each field and method will be assigned, using the *Compilation Unit* box on the *C#* or *VB* tab of their property sheets.

For partial types that contain inner types, you can specify the compilation unit to which each inner type will be assigned as follows:

1. Open the property sheet of the container type and click the *Inner Classifiers* tab.
2. If the *CompilationUnit* column is not displayed, click the *Customize Columns and Filter* tool, select the column from the selection box, and then click *OK* to return to the tab.
3. Click in the *CompilationUnit* column to reveal a list of available compilation units, select one, and click *OK* to close the property sheet.

### 2.6.3 C# 2.0 Namespaces

Namespaces restrict the scope of an object's name. Each class or other type must have a unique name within the namespace.

PowerDesigner models namespaces as standard packages with the *Use Parent Namespace* property deselected. For information about creating and working with packages, see [Packages \(OOM\) \[page 57\]](#).

In the following example, class Architect is declared in package Design which is a sub-package of Factory. The namespace declaration is the following:

```
namespace Factory.Design
{
    public class Architect
    {
    }
}
```

This structure, part of the NewProduct model, appears in the PowerDesigner Browser as follows:



Classifiers defined directly at the model level fall into the C# global namespace.

### 2.6.4 C# 2.0 Classes

PowerDesigner models C# 2.0 classes as standard UML classes, but with additional properties.

For information about creating and working with classes, see [Classes \(OOM\) \[page 38\]](#).

In the following example, class DialogBox inherits from class Window, which contains an inner classifier Control, as does class DialogBox:



In the following example, the class Client is defined as abstract by selecting the *Abstract* check box in the *General* tab of the class property sheet:



In the following example, the class SealedClient is defined as sealed by selecting the *Final* check box in the *General* tab of the class property sheet:



## C# Class Properties

C# class property sheets contain all the standard class tabs along with the **C#** tab, the properties of which are listed below:

| Property | Description  |
|----------|--|
| Static   | Specifies the static modifier for the class declaration. |
| Sealed   | Specifies the sealed modifier for the class declaration. |
| New      | Specifies the new modifier for the class declaration.    |
| Unsafe   | Specifies the unsafe modifier for the class declaration. |

## 2.6.5 C# 2.0 Interfaces

PowerDesigner models C# 2.0 interfaces as standard UML interfaces, with additional properties.

For information about creating and working with interfaces, see [Interfaces \(OOM\) \[page 60\]](#).

C# interfaces can contain events, properties, indexers and methods; they do not support variables, constants, and constructors.

## C# Interface Properties

C# interface property sheets contain all the standard interface tabs along with the C# tab, the properties of which are listed below:

| Property | Description  |
|----------|--|
| New      | Specifies the new modifier for the interface declaration.    |
| Unsafe   | Specifies the unsafe modifier for the interface declaration. |

## 2.6.6 C# 2.0 Structs

Structs are lightweight types that make fewer demands on the operating system and on memory than conventional classes. PowerDesigner models C# 2.0 structs as classes with a stereotype of <<Structure>>.

For information about creating and working with classes, see [Classes \(OOM\) \[page 38\]](#).

A struct can implement interfaces but does not support inheritance; it can contain events, variables, constants, methods, constructors, and properties.

In the following example, the struct contains two attributes:

|  |       |               |               |                         |  |
|--|-------|---------------|---------------|-------------------------|--|
| <table border="1"><tr><td>Point</td></tr><tr><td># Y : Integer</td></tr><tr><td># X : Integer</td></tr><tr><td>+ &lt;&lt;Constructor&gt;&gt; New()</td></tr></table> | Point | # Y : Integer | # X : Integer | + <<Constructor>> New() | {<br>public struct Point<br>{<br>public int New()<br>{<br>return 0;<br>}<br>private int x;<br>private int y;<br>}<br>} |
| Point  |       |               |               |                         |  |
| # Y : Integer  |       |               |               |                         |  |
| # X : Integer  |       |               |               |                         |  |
| + <<Constructor>> New()  |       |               |               |                         |  |

## Creating a Struct

You can create a struct in any of the following ways:

- Use the *Struct* tool in the *C# 2.0* Toolbox.
- Select **Model > Struct Objects** to access the List of Struct Objects, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Struct**.

## Struct Properties

Struct property sheets contains all the standard class tabs along with the *C#* tab, the properties of which are listed below:

| Property | Description   |
|----------|---|
| New      | Specifies the new modifier for the struct declaration.    |
| Unsafe   | Specifies the unsafe modifier for the struct declaration. |

## 2.6.7 C# 2.0 Delegates

Delegates are type-safe reference types that provide similar functions to pointers in other languages. PowerDesigner models delegates as classes with a stereotype of **<<Delegate>>** with a single operation code-named "**<signature>**". The visibility, name, comment, flags and attributes are specified on the class object whereas the return-type and parameters are specified on the operation.

A type-level (class or struct) delegate is modeled either as an operation bearing the **<<Delegate>>** stereotype, or as a namespace-level delegate in which the class representing the delegate is inner to the enclosing type.

For information about creating and working with classes, see [Classes \(OOM\) \[page 38\]](#).

|   |  |
|---|--|
| <pre> &lt;&lt;Delegate&gt;&gt; PackageDelegates  + &lt;&lt;Delegate&gt;&gt; ActionOccurred() : int </pre> | <pre> {     public delegate int ActionOccurred(); } </pre> |
|---|--|

## Creating a Delegate

You can create a delegate in any of the following ways:

- Use the *Delegate* tool in the C# 2.0 Toolbox.
- Select **Model > Delegate Objects** to access the List of Delegate Objects, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select **New > Delegate**.

## Delegate Properties

Delegate property sheets contains all the standard class tabs along with the *C#* tab, the properties of which are listed below:

| Property | Description   |
|----------|---|
| New      | Specifies the new modifier for the delegate declaration.    |
| Unsafe   | Specifies the unsafe modifier for the delegate declaration. |

## 2.6.8 C# 2.0 Enums

Enums are sets of named constants. PowerDesigner models enums as classes with a stereotype of *<<Enum>>*.

|  |  |
|--|--|
| <pre> &lt;&lt;enumeration&gt;&gt; Color  - Red   : Object - Blue  : Object - Green : Object - Max   : Object = Blue </pre> | <pre> {     public enum Color : colors     {         Red,         Blue,         Green,         Max = Blue     } } </pre> |
|--|--|

For information about creating and working with classes, see [Classes \(OOM\) \[page 38\]](#).

## Creating an Enum

You can create an enum in any of the following ways:

- Use the *Enum* tool in the C# 2.0 Toolbox.
- Select to access the List of Enum Objects, and click the *Add a Row* tool.
- Right-click the model (or a package) in the Browser, and select .

## Enum Properties

C# Enum property sheets contain all the standard class tabs along with the *C#* tab, the properties of which are listed below:

| Property           | Description  |
|--------------------|--|
| Base Integral Type | Specifies the base integral type for the enum.       |
| New                | Specifies the new modifier for the enum declaration. |

## 2.6.9 C# 2.0 Fields

PowerDesigner models C# fields as standard UML attributes.

For information about creating and working with attributes, see [Attributes \(OOM\) \[page 67\]](#).

## Field Properties

C# Field property sheets contain all the standard attribute tabs along with the *C#* tab, the properties of which are listed below:

| Property         | Description  |
|------------------|--|
| Compilation Unit | Specifies the compilation unit in which the field will be stored. This field is only available if the parent type is a partial type (allocated to more than one compilation unit). |
| New              | Specifies the new modifier for the field declaration.  |
| Unsafe           | Specifies the unsafe modifier for the field declaration.   |
| Const            | Specifies the const modifier for the field declaration.  |

| Property | Description  |
|----------|--|
| Readonly | Specifies the readonly modifier for the field declaration. |

## 2.6.10 C# 2.0 Methods

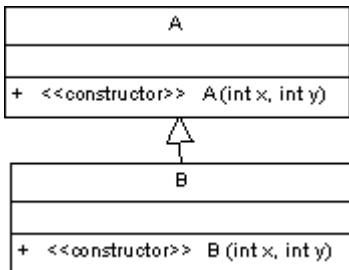
PowerDesigner models C# methods as operations.

For information about creating and working with operations, see [Operations \(OOM\) \[page 77\]](#).

### Method Properties

Method property sheets contain all the standard operation tabs along with the **C#** tab, the properties of which are listed below:

| Property         | Description   |
|------------------|---|
| Compilation Unit | Specifies the compilation unit in which the method will be stored. This field is only available if the parent type is a partial type (allocated to more than one compilation unit).   |
| Extern           | Specifies the extern modifier for the method declaration.   |
| New              | Specifies the new modifier for the method declaration. When a class inherits from another class and contains methods with identical signature as in the parent class, this field is selected automatically to make the child method prevail over the parent method. |
| Override         | Specifies the override modifier for the method declaration.   |
| Unsafe           | Specifies the unsafe modifier for the method declaration.   |
| Virtual          | Specifies the virtual modifier for the method declaration.  |
| Scope            | Specifies the scope of the method.  |

| Property         | Description  |
|------------------|--|
| Base Initializer | <p>Creates an instance constructor initializer of the form base, causing an instance constructor from the base class to be invoked.</p> <p>In the following example, class B inherits from class A. You define a Base Initializer in the class B constructor, which will be used to initialize the class A constructor:</p>  <pre> internal class B : A {     public B(int x, int y) : base(x + y, x - y)     {} } </pre> |
| This Initializer | Creates an instance constructor initializer, causing an instance constructor from the class itself to be invoked.  |

## Constructors and Destructors

You design C# constructors and destructors by clicking the *Add Default Constructor/Destructor* button on the class property sheet *Operations* tab. This automatically creates a constructor with the Constructor stereotype, and a destructor with the Destructor stereotype. Both constructor and destructor are grayed out in the list, which means you cannot modify their definition.

## Method Implementation

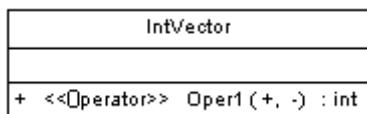
Class methods are implemented by the corresponding interface operations. To define the implementation of the methods of a class, you have to use the *To be implemented* button on the class property sheet *Operations* tab, then click the *Implement* button for each method to implement. The method is displayed with the <<Implement>> stereotype.

## Operator Method

You design a C# operator using an operation with the <<Operator>> stereotype. Make sure the <<Operator>> operation has Public visibility and the Static property selected.

To define an external operator, you have to set the extern extended attribute of the operation to True. The new, virtual and override extended attributes are not valid for operators.

The operator token (like +, -, !, ~, or ++ for example) is the name of the method.



## Conversion Operator Method

You design a C# conversion operator using an operation with the <<ConversionOperator>> stereotype.

You also need to declare the conversion operator using the explicit or implicit keywords. You define the conversion operator keyword by selecting the implicit or explicit value of the scope extended attribute.

In the following example, class Digit contains one explicit conversion operators and one implicit conversion operator:

A UML class diagram showing a class named "Digit". It has one attribute "- value : byte" and three operations: a constructor "+ <<constructor>> Digit(byte value)", an explicit conversion operator "+ <<ConversionOperator>> byte(Digit d) : byte", and an implicit conversion operator "+ <<ConversionOperator>> Digit(byte b) : Digit".

```
public struct Digit
{
    public Digit(byte value)
    {
        if (value < 0 || value > 9) throw
new ArgumentException();
        this.value = value;
    }
    public static implicit operator
byte(Digit d)
    {
        return d.value;
    }

    public static explicit operator
Digit(byte b)
    {
        return new Digit(b);
    }
    private byte value;
}
```

## 2.6.11 C# 2.0 Events, Indexers, and Properties

PowerDesigner represents C# events, indexers, and properties as standard UML attributes with additional properties.

For general information about creating and working with attributes, see [Attributes \(OOM\) \[page 67\]](#).

### Creating an Event, Indexer, or Property

To create an event, indexer, or property, open the property sheet of a type, click the *Attributes* tab, click the *Add* button at the bottom of the tab, and select the appropriate option.

These objects are created as follows:

- Events – stereotype <<Event>> with one or two linked operations representing the add and/or remove handlers
- Indexers – stereotype <<Indexer>> with one or two linked operations representing the get and/or set accessors
- Properties – stereotype <<Property>> with one or two linked operations representing the get and/or set accessors. In addition, you should note that:
  - The visibility of the property is defined by the visibility of the get accessor operation if any, otherwise by that of the set accessor operation.
  - When an attribute becomes a property, an implementation attribute is automatically created to store the property value. The implementation attribute is not persistent and has a private visibility. It has the stereotype <<PropertyImplementation>> and has the same name than the property but starting with a lowercase character. If the property name already starts with a lower case its first character will be converted to uppercase.
  - The implementation attribute can be removed for properties not needing it. (calculated properties for instance)
  - If the boolean-valued extended attribute Extern is set to true, no operations should be linked to the property.
  - When a property declaration includes an extern modifier, the property is said to be an external property. Because an external property declaration provides no actual implementation, each of its accessor-declarations consists of a semicolon.

### Event, Indexer, and Property Properties

Event, indexer, and property property sheets contain all the standard attribute tabs along with the *C#* tab, the properties of which are listed below:

| Property         | Description  |
|------------------|--|
| Compilation Unit | Specifies the compilation unit in which the attribute will be stored. This field is only available if the parent type is a partial type (allocated to more than one compilation unit). |

| Property | Description  |
|----------|--|
| New      | Specifies the new modifier for the attribute declaration.      |
| Unsafe   | Specifies the unsafe modifier for the attribute declaration.   |
| Abstract | Specifies the abstract modifier for the attribute declaration. |
| Extern   | Specifies the extern modifier for the attribute declaration.   |
| Override | Specifies the override modifier for the attribute declaration. |
| Sealed   | Specifies the sealed modifier for the attribute declaration.   |
| Virtual  | Specifies the virtual modifier for the attribute declaration.  |

## Event Example

The following example shows the Button class, which contains three events:

| Button                           |
|----------------------------------|
| - <<Event>> Click : Object       |
| - <<Event>> DoubleClick : Object |
| - <<Event>> RightClick : Object  |

## Property Example

In the following example, class Employee contains 2 properties. The Setter operation has been removed for property TimeReport:

| Employee |                            |                                |
|----------|----------------------------|--------------------------------|
| -        | <<Property>>               | Function : int                 |
| -        | <<Property>>               | TimeReport : int               |
| -        | <<PropertyImplementation>> | _Function : int                |
| -        | <<PropertyImplementation>> | _TimeReport : int              |
| +        | <<Setter>>                 | set_Function(int value) : void |
| +        | <<Getter>>                 | get_Function() : int           |
| +        | <<Getter>>                 | get_TimeReport() : int         |

```
{  
public class Employee  
{  
    private int _Function;  
    private int _TimeReport;  
    // Property Function  
    private int Function  
    {  
        get  
        {  
            return _Function;  
        }  
        set  
        {  
            if (this._Function != value)  
                this._Function = value;  
        }  
    }  
    // Property TimeReport  
    private int TimeReport  
    {  
        get  
        {  
            return _TimeReport;  
        }  
    }  
}
```

## Indexer Example

In the following example, class Person contains indexer attribute Item. The parameter used to sort the property is String Name:

| Person |  |
|--------|--|
| -      | <<Indexer>> Item : int                           |
| -      | <<IndexerImplementation>> _childAges : Hashtable |
| +      | <<Setter>> set_Item (int value) : void           |
| +      | <<Getter>> get_Item () : int                     |

```
public class Person
{
    private Hashtable _childAges;
    // Indexer Item
    private int this[String name]
    {
        get
        {
            return (int)_ChildAges[name];
        }
        set
        {
            _ChildAges[name] = value;
        }
    }
    Person someone;
    someone ["Alice"] = 3;
    someone ["Elvis"] = 5;
}
```

## 2.6.12 C# 2.0 Inheritance and Implementation

PowerDesigner models C# inheritance links between types as standard UML generalizations.

For more information about generalizations, see [Generalizations \(OOM\) \[page 93\]](#).

PowerDesigner models C# implementation links between types and interfaces as standard UML realizations. For more information, see [Realizations \(OOM\) \[page 99\]](#).

## 2.6.13 C# 2.0 Custom Attributes

PowerDesigner provides full support for C# 2.0 custom attributes, which allow you to add metadata to your code. This metadata can be accessed by post-processing tools or at run-time to vary the behavior of the system.

You can use built-in custom attributes, such as System.Attribute and System.ObsoleteAttribute, and also create your own custom attributes to apply to your types.

For general information about modeling this form of metadata in PowerDesigner, see [Annotations \(OOM\) \[page 102\]](#).

## 2.6.14 Generating C# 2.0 Files

You generate C# 2.0 source files from the classes and interfaces of a model. A separate file, with the file extension .cs, is generated for each class or interface that you select from the model, along with a generation log file.

### Context

The following PowerDesigner variables are used in the generation of C# 2.0 source files:

| Variable | Description  |
|----------|--|
| CSC      | C# compiler full path. For example, C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705\csc.exe                                     |
| WSDL     | Web Service proxy generator full path. For example, C:\Program Files\Microsoft Visual Studio .NET \FrameworkSDK\Bin\wsdl.exe |

To review or edit these variables, select  [Tools](#)  and click the *Variables* category.

### Procedure

1. Select  [Language](#)  to open the C# 2.0 Generation dialog.
2. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see [Checking an OOM \[page 251\]](#)).
3. [optional] Select any additional targets to generate for. These targets are defined by any extensions that may be attached to your model (see [Working with Generation Targets \[page 236\]](#)).
4. [optional] Click the [Selection](#) tab and specify the objects that you want to generate from. By default, all objects are generated.
5. [optional] Click the [Options](#) tab and set any appropriate generation options:

| Option                                    | Description   |
|---|---|
| Generate object ids as documentation tags | Specifies whether to generate object ids for use as documentation tags.   |
| Sort class members primarily by           | Specifies the primary method by which class members are sorted: <ul style="list-style-type: none"><li>○ Visibility</li><li>○ Type</li></ul> |

| Option  | Description   |
|---|---|
| Class members type sort                               | Specifies the order by which class members are sorted in terms of their type: <ul style="list-style-type: none"> <li>○ Methods – Properties - Fields</li> <li>○ Properties – Methods - Fields</li> <li>○ Fields – Properties - Methods</li> </ul> |
| Class members visibility sort                         | Specifies the order by which class members are sorted in terms of their visibility: <ul style="list-style-type: none"> <li>○ Public - Private</li> <li>○ Private – Public</li> <li>○ None</li> </ul>  |
| Generate Visual Studio 2005 project files             | Specifies whether to generate project files for use with Visual Studio 2005.  |
| Generate Assembly Info File                           | Specifies whether to generate information files for assemblies.   |
| Generate Visual Studio Solution File                  | Specifies whether to generate a solution file for use with Visual Studio 2005.  |
| Generate Web Service code in .asmx file               | Specifies whether to generate web services in a .asmx file.   |
| Generate default accessors for navigable associations | Specifies whether to generate default accessors for navigable associations.   |

### Note

For information about modifying the options that appear on this and the [Tasks](#) tab and adding your own options and tasks, see *Customizing and Extending PowerDesigner > Object, Process, and XML Language Definition Files > Generation Category* .

- [optional] Click the [Generated Files](#) tab and specify which files will be generated. By default, all files are generated.

For information about customizing the files that will be generated, see *Customizing and Extending PowerDesigner > Extension Files > Generated Files (Profile)* .

- [optional] Click the [Tasks](#) tab and specify any appropriate generation tasks to perform:

| Task  | Description   |
|---|---|
| WSDLDotNet: Generate Web service proxy code | Generates the proxy class   |
| Compile source files                        | Compiles the source files   |
| Open the solution in Visual Studio          | Depends on the Generate Visual Studio 2005 project files option. Opens the generated project in Visual Studio 2005. |

- Click **OK** to begin generation.

When generation is complete, the Generated Files dialog opens, listing the files that have been generated to the specified directory. Select a file in the list and click *Edit* to open it in your associated editor, or click *Close* to exit the dialog.

## 2.6.15 Reverse Engineering C# 2.0 Code

You can reverse engineer C# files into an OOM.

### Procedure

1. Select  *Language*  *Reverse Engineer C#* to open the Reverse Engineer C# dialog box.
2. Select what form of code you want to reverse engineer. You can choose between:
  - C# files (.cs)
  - C# directories
  - C# projects (.csproj)
3. Select files, directories, or projects to reverse engineer by clicking the *Add* button.

#### Note

You can select multiple files simultaneously using the `ctrl` or `shift` keys. You cannot select multiple directories.

The selected files or directories are displayed in the dialog box and the base directory is set to their parent directory. You can change the base directory using the buttons to the right of the field.

4. [optional] Click the *Options* tab and set any appropriate options:

| Option   | Description   |
|--|---|
| File encoding  | Specifies the default file encoding of the files to reverse engineer                                      |
| Ignore operation body                                | Reverses classes without including the body of the code   |
| Ignore comments                                      | Reverses classes without including code comments  |
| Create Associations from classifier-typed attributes | Creates associations between classes and/or interfaces  |
| Create symbols                                       | Creates a symbol for each object in the diagram, if not, reversed objects are only visible in the browser |

| Option                               | Description  |
|--------------------------------------|--|
| Libraries                            | <p>Specifies a list of library models to be used as references during reverse engineering. If the reverse engineered model contains shortcuts to objects defined in a library and you specify the library here, the link between the shortcut and its target object (in the library) will be preserved and the library will be added to the list of target models in the reverse engineered model.</p> <p>You can drag and drop the libraries in the list to specify the order in which PowerDesigner will seek to resolve shortcuts found in the reverse engineered model against them. Use the List of Target Models to manage libraries related to the reverse engineered model, for example, you can change the library version.</p> |
| Preserve file structure              | Creates an artifact during reverse engineering in order to be able to regenerate an identical file structure   |
| Mark classifiers not to be generated | Specifies that reversed classifiers (classes and interfaces) will not be generated from the model. To generate the classifier, you must select the <i>Generate</i> check box in its property sheet   |

- [optional] Click the *Preprocessing* tab and set any appropriate preprocessing symbols (see [C# Reverse Engineering Preprocessing Directives \[page 423\]](#)).

- Click **OK** to begin the reverse engineering.

A progress box is displayed. If the model in which you are reverse engineering already contains data, the Merge Models dialog box is displayed.

For more information on merging models, see *Core Features Guide > Modeling with PowerDesigner > Comparing and Merging Models*.

The classes are added to your model. They are visible in the diagram and in the Browser, and are also listed in the Reverse tab of the Output window, located in the lower part of the main window.

## 2.6.15.1 C# Reverse Engineering Preprocessing Directives

C# files may contain conditional code that needs to be handled by preprocessing directives during reverse engineering. A preprocessing directive is a command placed within the source code that directs the compiler to do a certain thing before the rest of the source code is parsed and compiled.

The preprocessing directive has the structure:`#directive symbol`, where # is followed by the name of the directive, and symbol is a conditional compiler constant used to select particular sections of code and exclude other sections.

In C#, symbols have no value, and can only be true or false.

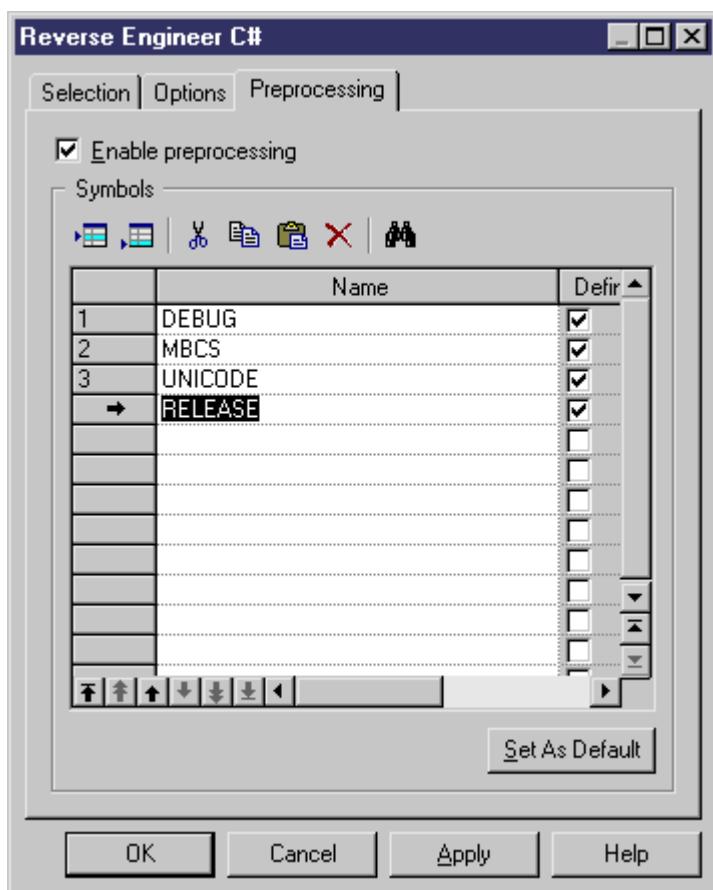
In the following example, the `#if` directive is used with symbol DEBUG to output a certain message when DEBUG symbol is true, if DEBUG symbol is false, another output message is displayed.

```
using System;
public class MyClass
{
    public static void Main()
```

```
{  
    #if DEBUG  
        Console.WriteLine("DEBUG version");  
    #else  
        Console.WriteLine("RELEASE version");  
    #endif  
}  
}
```

You can declare a list of symbols for preprocessing directives. These symbols are parsed by preprocessing directives: if the directive condition is true the statement is kept, otherwise the statement is removed.

To define a C# preprocessing symbol on the Reverse Engineering dialog *Preprocessing* tab, click the [Add a Row](#) tool and enter the symbol name in the *Name* column.



The **Defined** check box is selected for each symbol to indicate that it will be taken into account during preprocessing. The list of symbols is saved in the model and will be reused when you synchronize your model with existing code using the Synchronize with Generated Files command (see [Synchronizing a Model with Generated Files \[page 240\]](#)).

## i Note

Symbol names are case sensitive and must be unique. Do not enter reserved words like true, false, if, do and so on. PowerDesigner does not support the default namespace in a Visual Studio project. If you define default namespaces in your projects, you should avoid reverse engineering the entire solution. It is better to reverse engineer each project separately.

The following directives are supported:

| Directive          | Description  |
|--------------------|--|
| #define, #undefine | #define defines a symbol, and #undefine removes a previous definition of the symbol.   |
| #if, #elif, #endif | #if evaluates a condition, if the condition is true, the statement following the condition is kept otherwise it is ignored. If an #if test fails, #elif can include or exclude source code, depending on the resulting value of its own expression or identifier. #endif closes the conditional block. |
| #warning, #error   | Displays a warning or error message if the condition is true   |

### 2.6.15.1.1 C# Supported Preprocessing Directives

The following directives are supported during preprocessing:

| Directive | Description  |
|-----------|--|
| #define   | Defines a symbol   |
| #undefine | Removes a previous definition of the symbol  |
| #if       | Evaluates a condition, if the condition is true, the statement following the condition is kept otherwise it is ignored   |
| #elif     | Used with the #if directive, if the previous #if test fails, #elif includes or exclude source code, depending on the resulting value of its own expression or identifier |
| #endif    | Closes the #if conditional block of code   |
| #warning  | Displays a warning message if the condition is true  |
| #error    | Displays an error message if the condition is true   |

Note: #region, #endregion, and #line directives are removed from source code.

### 2.6.15.1.2 Defining a C# Preprocessing Symbol

You can define C# preprocessing symbols in the preprocessing tab of the reverse engineering dialog box.

## Context

Symbol names are case sensitive and must be unique. Make sure you do not type reserved words like true, false, if, do and so on.

The list of symbols is saved in the model and will be reused when you synchronize your model with existing code using the Synchronize with Generated Files command.

For more information on the synchronize with generated files command see [Synchronizing a Model with Generated Files \[page 240\]](#).

You can use the Set As Default button to save the list of symbols in the registry.

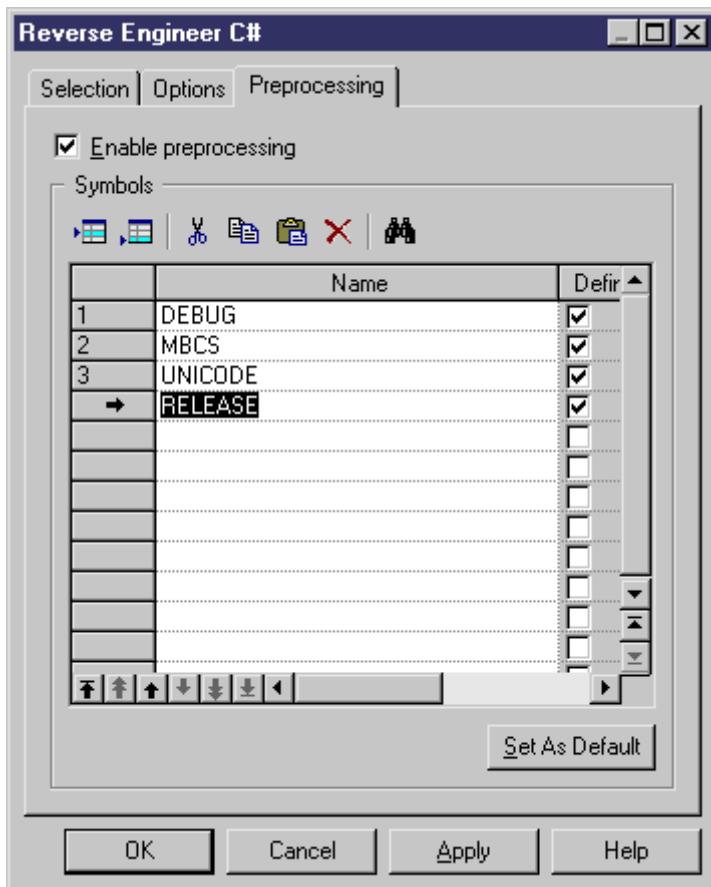
## Procedure

1. Select ► *Language* > *Reverse engineering C#* ▾.

The Reverse Engineering C# dialog box is displayed.

2. Click the *Preprocessing* tab to display the corresponding tab.
3. Click the *Add a Row* tool to insert a line in the list.
4. Type symbol names in the *Name* column.

The *Defined* check box is automatically selected for each symbol to indicate that the symbol will be taken into account during preprocessing.



5. Click *Apply*.

## Results

PowerDesigner does not support the default namespace in a Visual Studio project. If you define default namespaces in your projects, you should avoid reverse engineering the entire solution. It is better to reverse engineer each project separately.

## 2.7 C++

PowerDesigner supports the modeling of C++ programs including round-trip engineering.

### 2.7.1 Designing for C++

This section explains how to design C++ objects in the PowerDesigner Object Oriented Model.

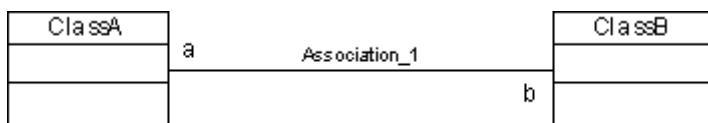
#### Namespace Declaration for Classifiers

The extended attribute UseNamespace allows you to generate a classifier inside a namespace declaration. You should set the extended attribute value to True.

#### Bidirectional Associations Management

The problem of bidirectional associations is addressed by using forward declarations instead of includes.

Consider a bidirection association between ClassA and ClassB.



The generated code in A.h would be the following:

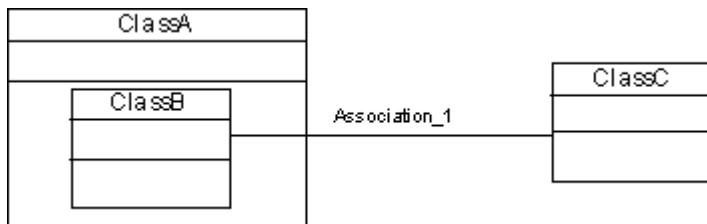
```
#if !defined(_A_h)
#define _A_h
class B; // forward declaration of class B
class A
{
public:
    B* b;
protected:
private:
```

```
};  
#endif
```

The generated code in B.h would be the following:

```
#if !defined(__B_h)  
#define __B_h  
class A; // forward declaration of class A  
class B  
{  
public:  
    A* a;  
protected:  
private:  
};  
#endif
```

This approach will not work if one of the classes is an inner class because it is not possible to forward-declare inner classes in C++.



If such a situation occurs, a warning message is displayed during generation, and the corresponding code is commented out.

## Unsupported ANSI Features

PowerDesigner does not support the following C++ features:

- Templates
- Enums
- Typedefs
- Inline methods

## 2.7.2 Generating for C++

When generating with C++, the files generated are generated for classes and interfaces.

### Context

A header file with the .h extension, and a source file with the .cpp extension are generated per classifier.

A generation log file is also created after generation.

### Procedure

1. Select  [Language](#)  [Generate C++ Code](#)  to display the Generation dialog box.
2. Type a destination directory for the generated file in the [Directory](#) box.

or

Click the [Select a Path](#) button to the right of the [Directory](#) box and browse to select a directory path.

3. Select the objects to include in the generation from the tabbed pages at the bottom of the [Selection](#) page.

#### Note

All classes of the model, including those grouped into packages, are selected and displayed by default. You use the selection tools to the right of the list to modify the selection. The [Include Sub-Packages](#) tool allows you to include all classes located within packages.

4. Click the [Options](#) tab to display the [Options](#) page.
5. <optional> Select the [Check Model](#) check box if you want to verify the validity of your model before generation.
6. Select a value for each required option.
7. Click the [Tasks](#) tab, then select the required task(s).
8. Click [OK](#) to generate.

A Progress box is displayed. The Result list displays the files that you can edit. The result is also displayed in the [Generation](#) page of the Output window, located in the bottom part of the main window.

All C++ files are generated in the destination directory.

## 2.8 Object/Relational (O/R) Mapping

PowerDesigner supports and can automatically generate and synchronize O/R Mappings between OOM and PDM objects.

The following table lists object mappings in these two model types:

| OOM Element   | PDM Element   |
|---|---|
| Domain  | Domain  |
| Class (if the Persistent checkbox and Generate table option are selected) | Table   |
| Attribute Column (if the Persistent checkbox is selected)                 | Column  |
| Identifier  | Identifier  |
| Association   | Reference or table  |
| Association class   | Table with two associations between the end points of the association class |
| Generalization  | Reference   |

You can define mappings between these two model types in any of the following ways:

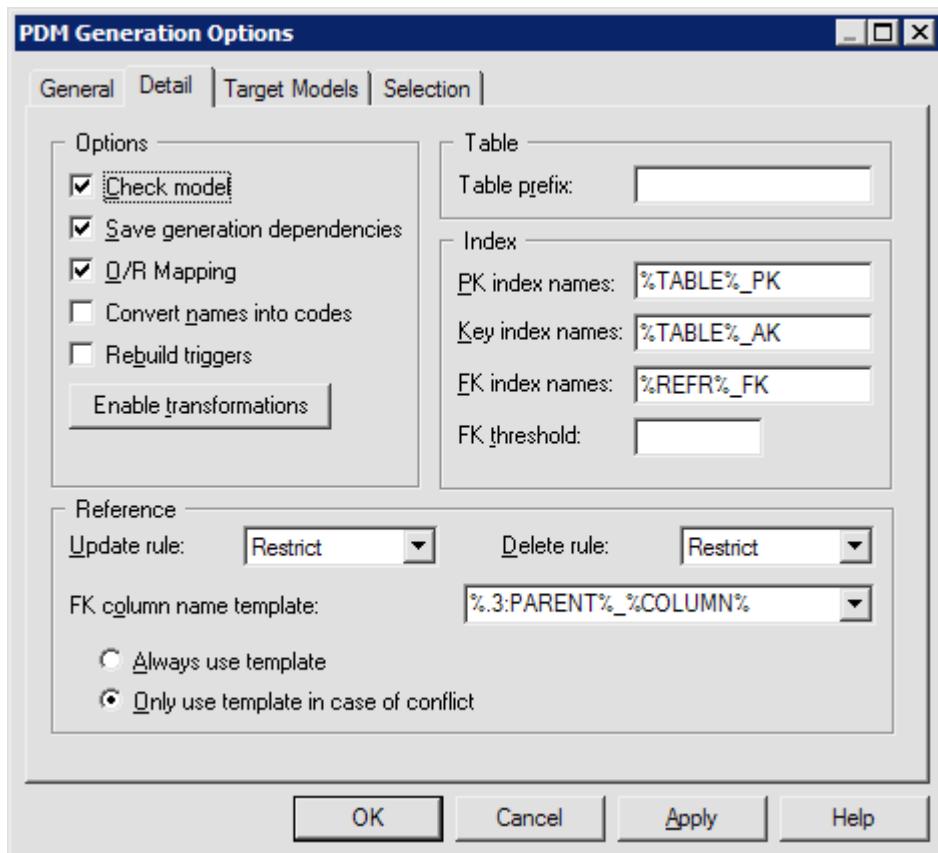
- Top-down – generate tables and other PDM objects from OOM classes
- Bottom-up – generate classes and other OOM objects from PDM tables
- Meet-in-the-middle – manually define mappings between classes and tables using the visual mapping editor

### 2.8.1 Top-Down: Mapping Classes to Tables

PowerDesigner provides default transformation rules for generating physical data models from object-oriented models. You can customize these rules with persistence settings and generation options.

#### Procedure

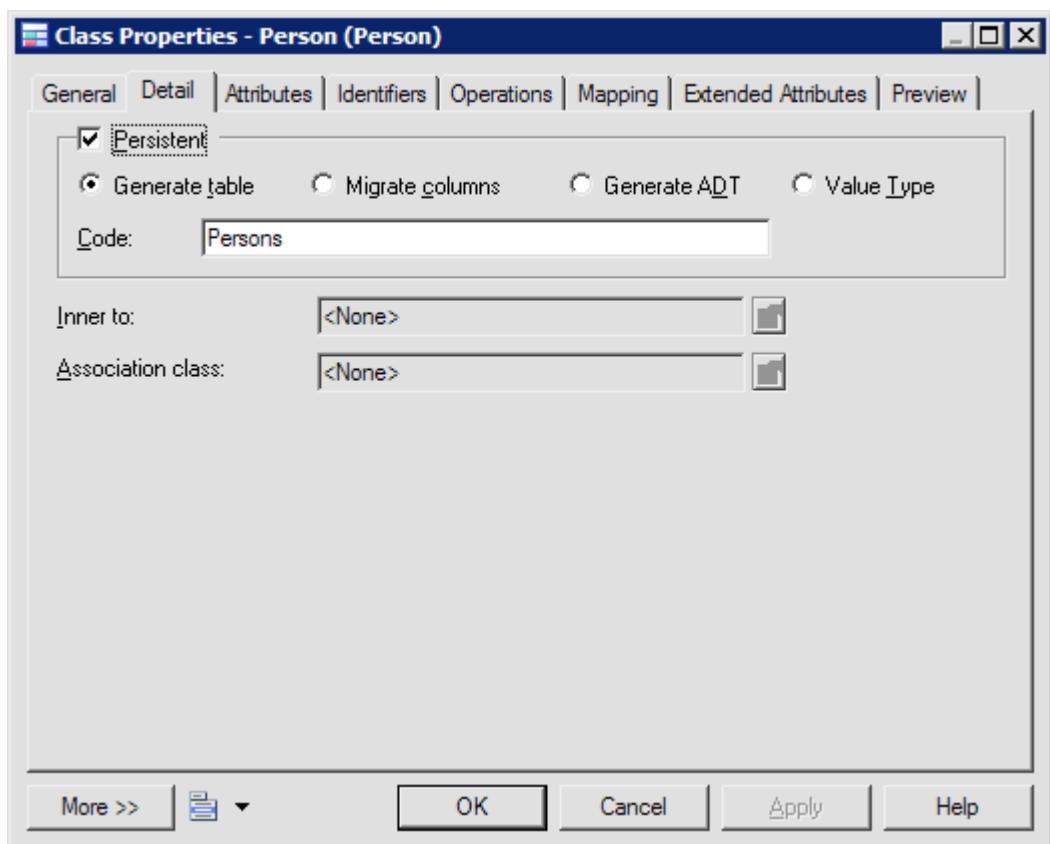
1. Create your OOM, and populate it with persistent classes (see [Entity Class Transformation \[page 431\]](#)), inheritance links and associations etc, to define the structure of your model domain.
2. Select **Tools** **Generate Physical Data Model** to open the PDM Generation Options dialog.
3. On the *General* tab, specify the DBMS type and the name and code of the PDM to generate (or select an existing PDM to update).
4. Click the *Detail* tab and select the *O/R Mapping* checkbox. You can optionally also specify a table prefix that will be applied to all generated tables.



5. Click the *Selection* tab and the select the OOM objects that you want to transform into PDM objects.
6. Click *OK* to generate (or update) your PDM.

### 2.8.1.1 Entity Class Transformation

To transform a class into a table, select the *Persistent* option on the *Detail* tab of its property sheet and then specify the type in the *Persistent* groupbox.



Persistent classes are classes with one of the following persistent types:

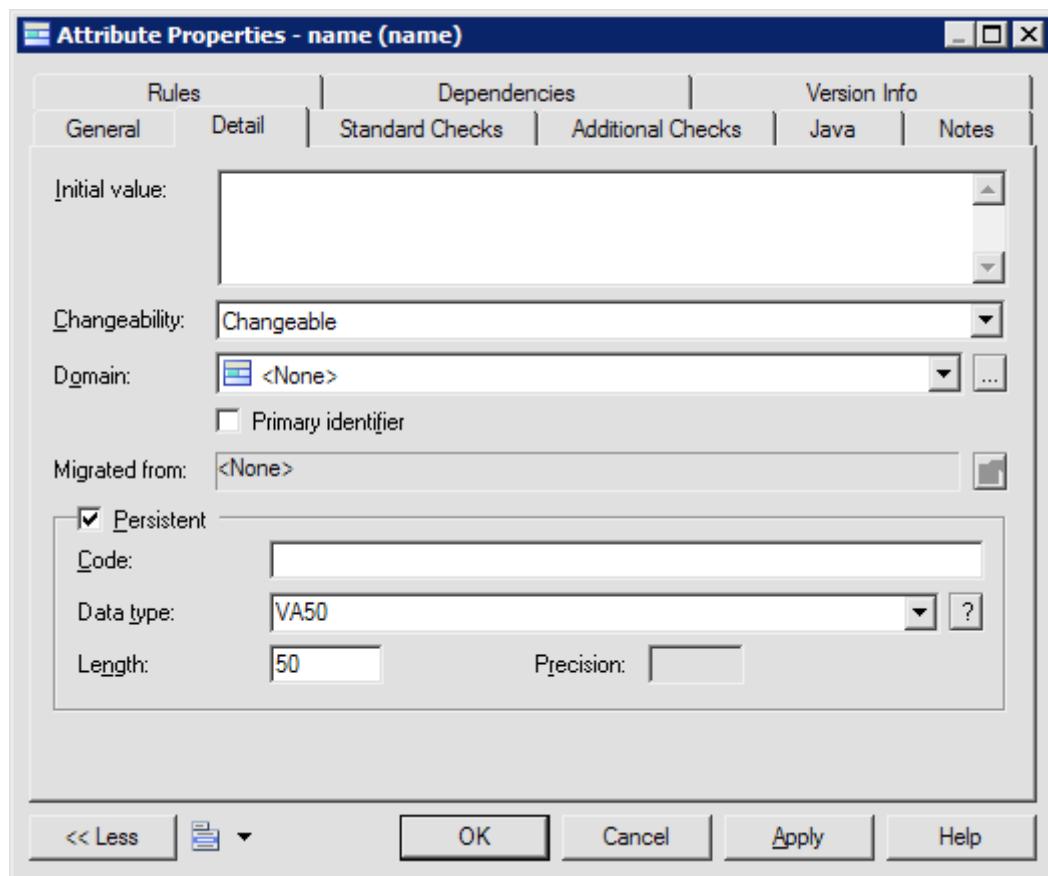
- Generate table - These classes are called Entity classes, and will be generated as separate tables. You can customize the code of the generated tables in the Code box in the Persistent groupbox. Only one table can be generated for each entity class with this type, but you can manually map an entity class to multiple tables (see [Entity Class Mapping \[page 443\]](#)).
- Migrate columns - These classes are called Entity classes, but no separate table will be generated for them. This persistent type is used in inheritance transformation, and its attributes and associations are migrated to the generated parent or child table.
- Generate ADT - These classes are generated as abstract data types, user-defined data types that can encapsulate a range of data values and functions. This option is not used when you define O/R Mapping.
- Value Type - These classes are called Value type classes. No separate table will be generated for the class; its persistent attributes will be transformed into columns that are embedded in other table(s)

#### i Note

Identifiers of persistent classes, where the generation type is not set to Value type are transformed into table keys. Primary identifiers are transformed into primary keys or part of primary keys (see [Primary Identifier Mapping \[page 448\]](#)). Persistent attributes contained in primary identifiers are transformed into columns of primary keys.

## 2.8.1.2 Attribute Transformation

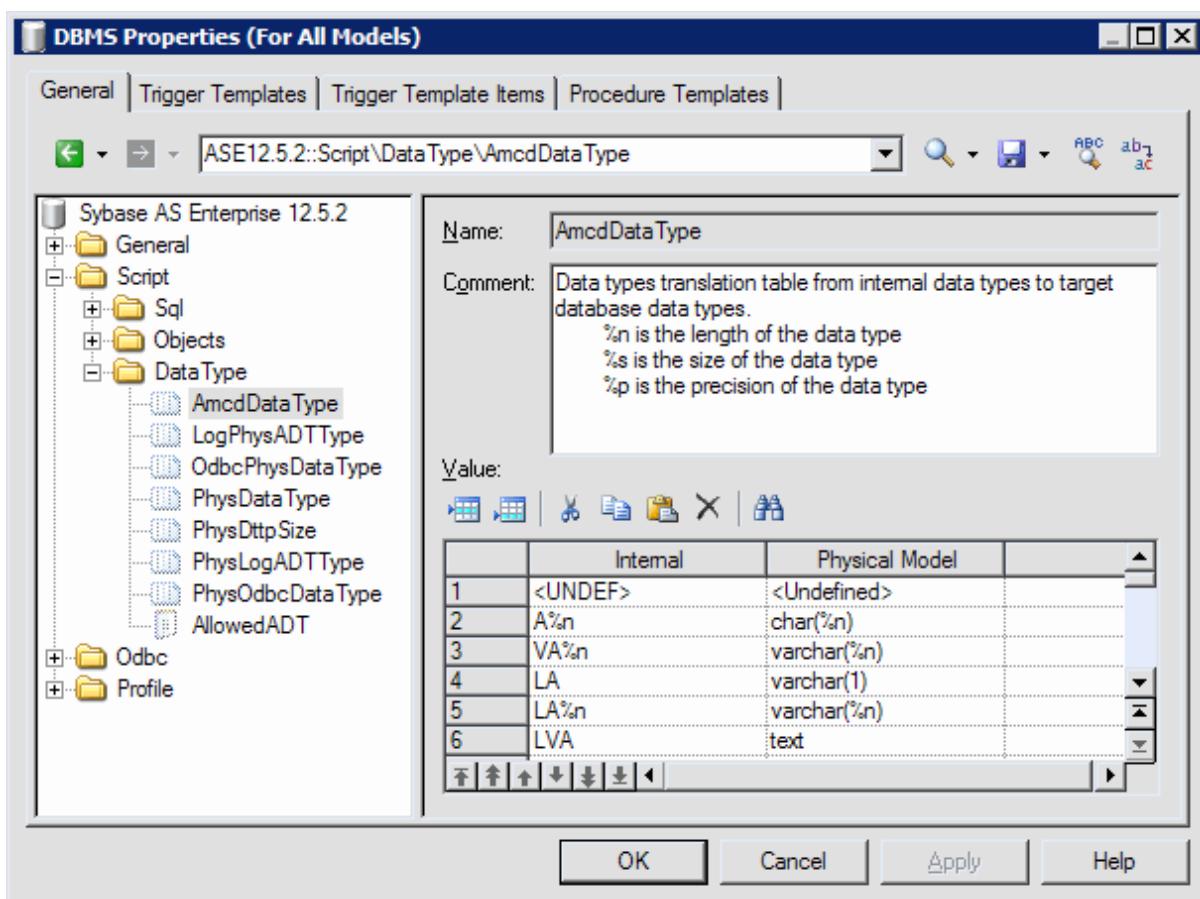
To transform an attribute into a column, select the *Persistent* option on the *Detail* tab of its property sheet.



Persistent attributes can have simple data types or complex data types:

- Simple Data Type - such as int, float, String, Date etc.

Each persistent attribute is transformed into one column. Its data type is converted into an internal standard data type, which is then mapped to the appropriate data type in the DBMS. These transformations are controlled by the table of values in the AMCDDataType entry in the Data Type folder of the DBMS definition:



- Complex Data Type - based on a classifier. The transformation depends on the persistent settings of the classifier. The classifier is generally used as a value type class (see [Value Type Transformation \[page 434\]](#)).

You can also customize the code of the generated data types in the Code box of the Persistent groupbox. You can also customize the code of the generated columns.

### 2.8.1.3 Value Type Transformation

PowerDesigner supports fine-grained persistence model. Multiple classes can be transformed into single table.

Given two classes, *Person* and *Address*, where the class *Person* contains an attribute *address* whose data type is *Address*, the classes can be transformed into one table if the transformation type of Class *Address* is set to Value type. The columns transformed from persistent attributes of the class *Address* will be embedded in the table transformed from the class *Person*.

| Classes   | Table  |
|---|--|
| <p>Person</p> <pre>- pid : int - name : String - email : String - age : int - address : Address</pre><br><p>Address</p> <pre>- country : String - state : String - city : String - address : String - postalCode : String</pre> | <p>Person</p> <pre>pid int &lt;pk&gt; name varchar(50) email varchar(254) age int country varchar(254) state varchar(254) city varchar(254) address varchar(254) postalCode varchar(254)</pre> |

## 2.8.1.4 Association Transformation

Association defined between entity classes will be transformed into reference keys or reference tables. Associations with Value type classes as target or source will be ignored.

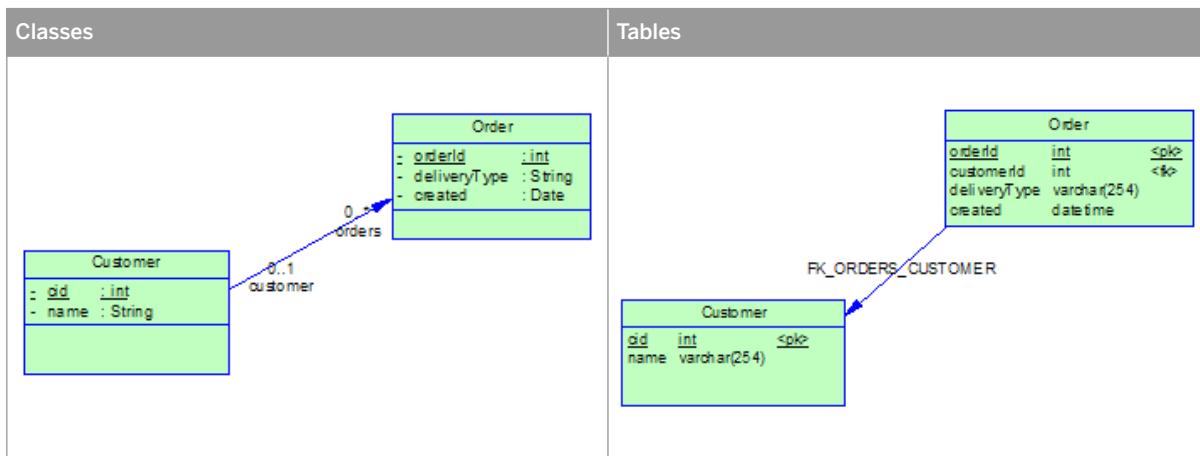
Transformation rules differ according to the type of the association:

- One-to-one - one foreign key will be generated with the same direction as the association. The primary key of parent table will also migrate into child table as its foreign key.

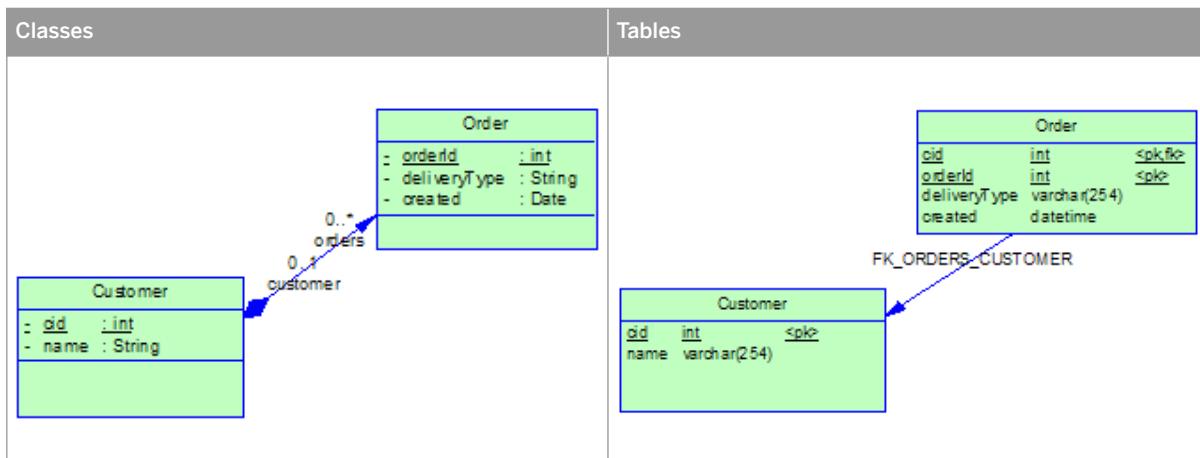
| Classes  | Tables   |
|--|--|
| <p>Person</p> <pre>- pid : int - name : String - email : String - age : int</pre><br><p>Account</p> <pre>- aid : int - lastAccess : Date - expired : boolean</pre> | <p>Account</p> <pre>aid int &lt;pk&gt; lastAccess datetime expired bit</pre><br><p>FK_PERSON_ACCOUNT</p> <p>Person</p> <pre>pid int &lt;pk&gt; aid int &lt;fk&gt; name varchar(254) email varchar(254) age int</pre> |

The generated foreign key has the same direction as the association direction. If the association is bidirectional (can navigate in two ways), foreign keys with both directions will be generated since PowerDesigner does not know which table is the parent table. You need to delete one manually.

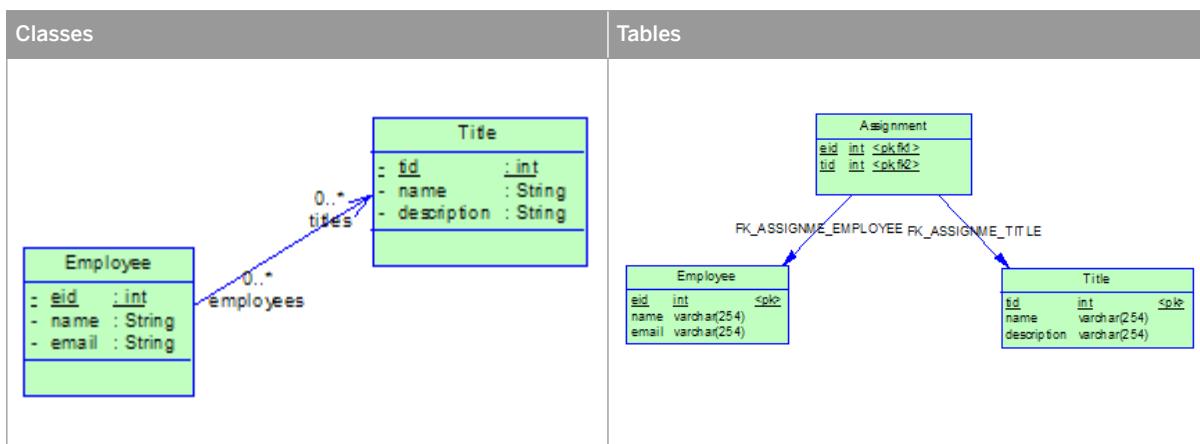
- One-to-many association - just one foreign key will be generated for each one-to-many association, whatever its direction (bidirectional or unidirectional). The reference key navigates from the table generated from the entity class on multiple-valued side to the table generated from the entity class on single-valued side.



- One-to-many composition - PowerDesigner can generate a primary key of a parent table as part of the primary key of the child table if you define the association as composition with the class on single-valued side containing the class on multiple-valued side:



- Many-to-many - each many-to-many association will be transformed into one middle table and two reference keys that navigate from the middle table to the tables generated from the two entity classes.



For most O/R Mapping frameworks, one unidirectional one-to-many association (see [One-to-Many Association Mapping Strategy \[page 453\]](#)) will usually be mapped to a middle table and two references navigating from the middle table to the tables mapped by the two entity classes.

### Note

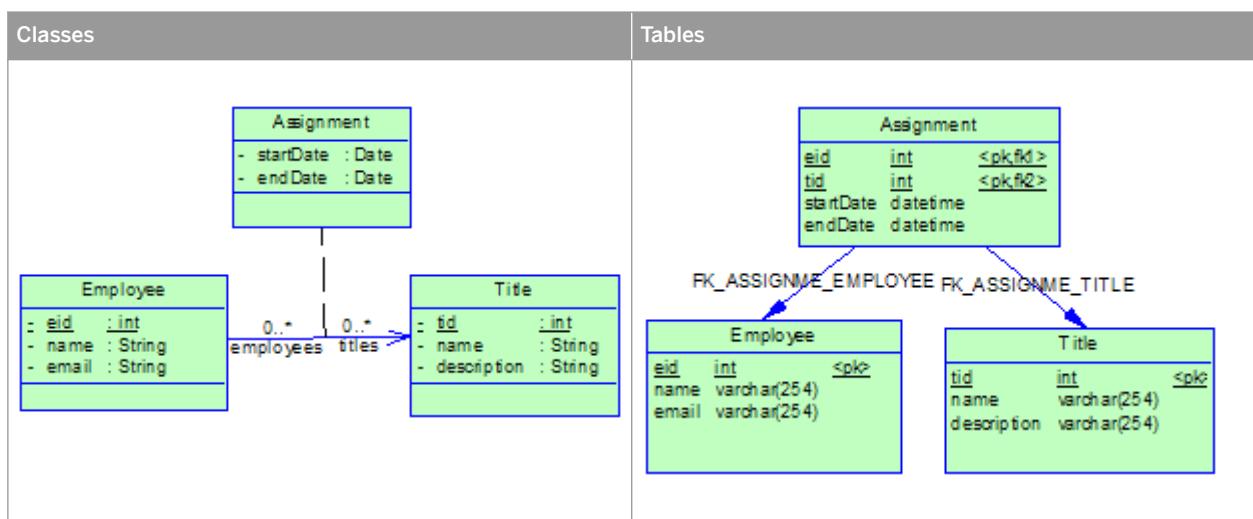
The minimal multiplicity of association ends can affect the Mandatory property of the generated reference keys:

- For one-to-one associations if the minimal multiplicity of side that is transformed to parent table is more than one, the generated foreign key will be mandatory.
- For one-to-many associations, if the minimal multiplicity on single-valued side is more than one, the generated foreign key will be mandatory.

## 2.8.1.4.1 Association Class Transformation

In O/R Mapping, association classes are only meaningful for many-to-many associations. Persistent attributes in the association entity class will be transformed into columns of the middle table.

In the following example, we have defined an association class to hold ultra information for the association:



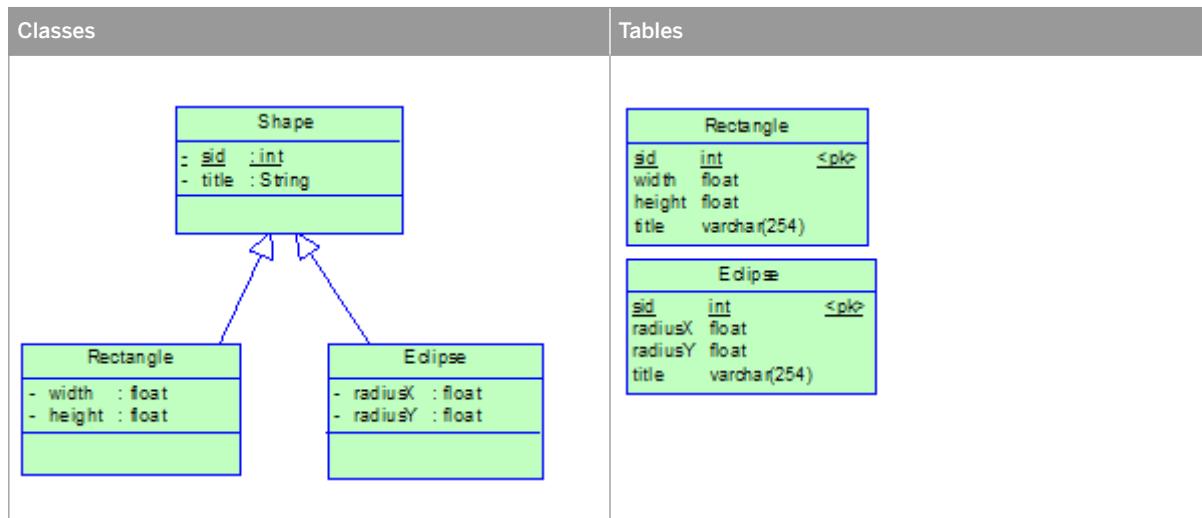
## 2.8.1.5 Inheritance Transformation

PowerDesigner supports various mapping strategies for inheritance persistence. Each strategy has its pros and cons, and you should select the most appropriate one for your needs. You can also apply mixed strategies, but this may not be well supported by your persistence framework.

- Table per class hierarchy. All the classes in a hierarchy are mapped to a single table. The table has a column that serves as a "discriminator column". The value of this column identifies the specific subclass to which the instance that is represented by the row belongs.

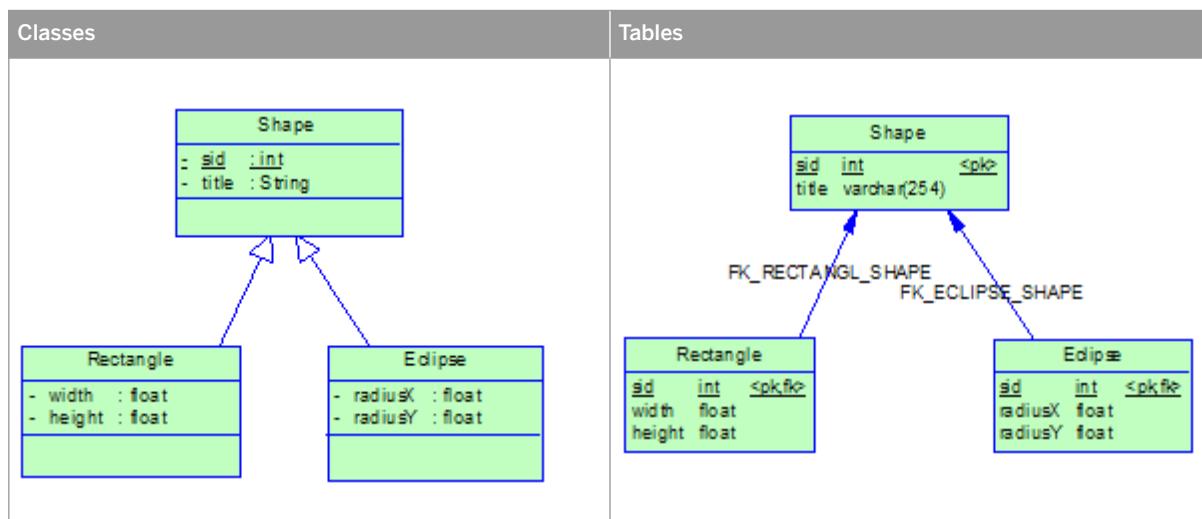
In order to apply this kind of strategy, you should set the transformation type of leaf classes to *Generate table* and the transformation type of the other classes in the hierarchy to *Migrate columns*. PowerDesigner will only

generate the tables for leaf classes. If you want to map other classes to tables, you need to create them manually.



- Joined subclass. The root class of the class hierarchy is represented by a single table. Each subclass is represented by a separate table. This table contains the fields that are specific to the subclass (not inherited from its super class), as well as the column(s) that represent its primary key. The primary key column(s) of the subclass table serves as a foreign key to the primary key of the super class table.

In order to apply this kind of strategy, you should set the transformation type of all the classes to *Generate table*. You can also, optionally, define a discriminator.

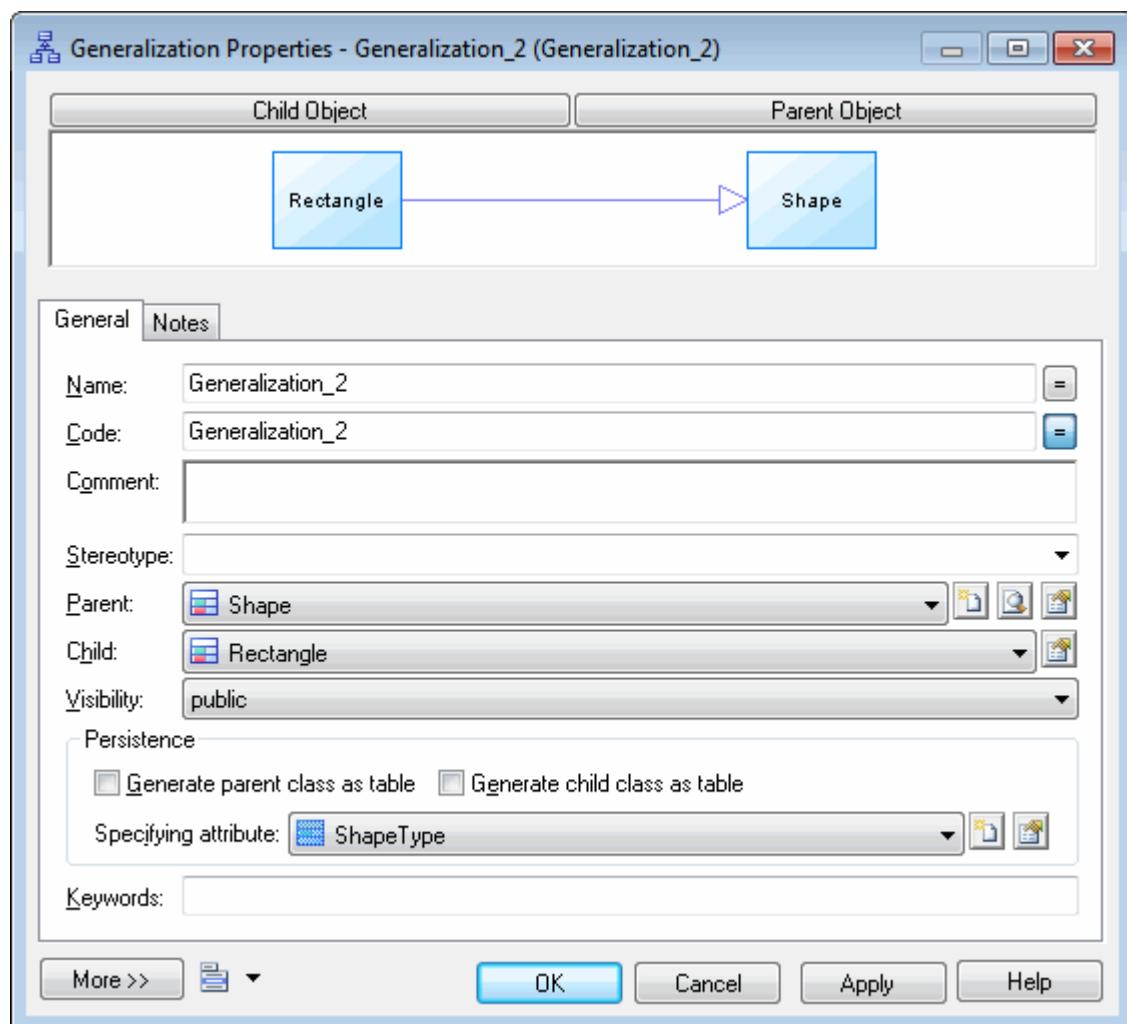


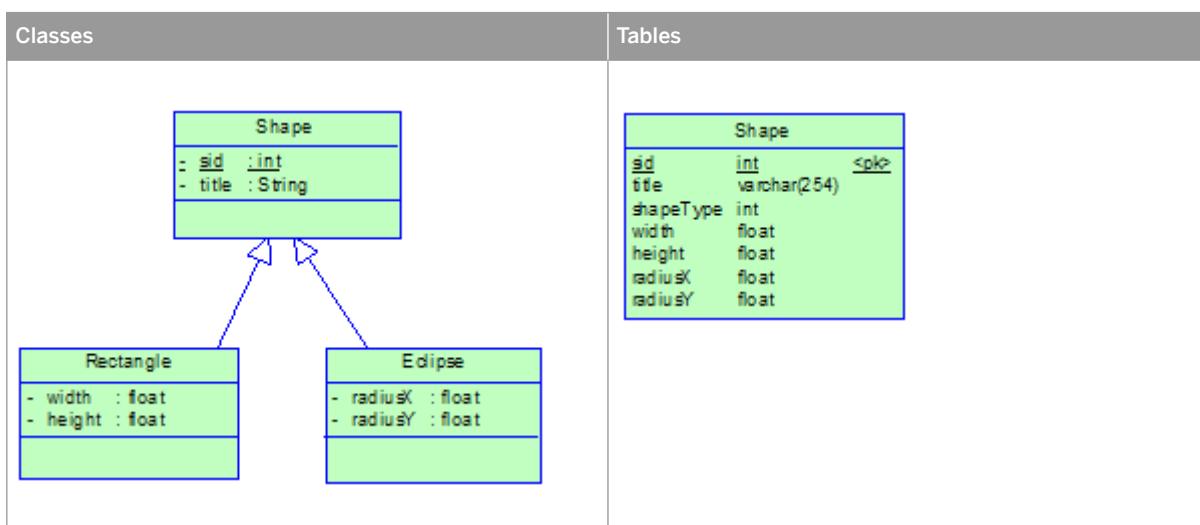
- Table per class. Each class is mapped to a separate table. All properties of the class, including inherited properties, are mapped to columns of the table for the class.

In order to apply this kind of strategy, you should set the transformation type of the root class to *Generate table* and the transformation type of other classes in the class hierarchy to *Migrate columns*.

For each class hierarchy, a discriminator is needed to distinguish between different class instances. You need to select one of the attributes of the root class in the *Specifying Attribute* list located in the property sheet of one of the children inheritance links of the root class. The attribute will be transformed into a discriminator

column. In the following example, we define one extra attribute *shapeType* in *Shape* and select it as discriminator attribute:





- Mixed Strategy - You can apply more than one strategy in the same inheritance hierarchy. The transformation of entity classes with the *Generate table* transformation type will not change, but the transformation of those set to *Migrate columns* will be slightly different. If entity classes set to *Migrate columns* have both their super-class and sub-classes set to *Generate table*, the columns transformed from their persistent attributes will be migrated into tables transformed from sub-classes. The migration to sub-classes has higher priority.

## 2.8.2 Bottom-Up: Mapping Tables to Classes

PowerDesigner provides default transformation rules for generating object-oriented models from physical data models. You can enhance the generated mappings manually using the Mapping Editor.

### Context

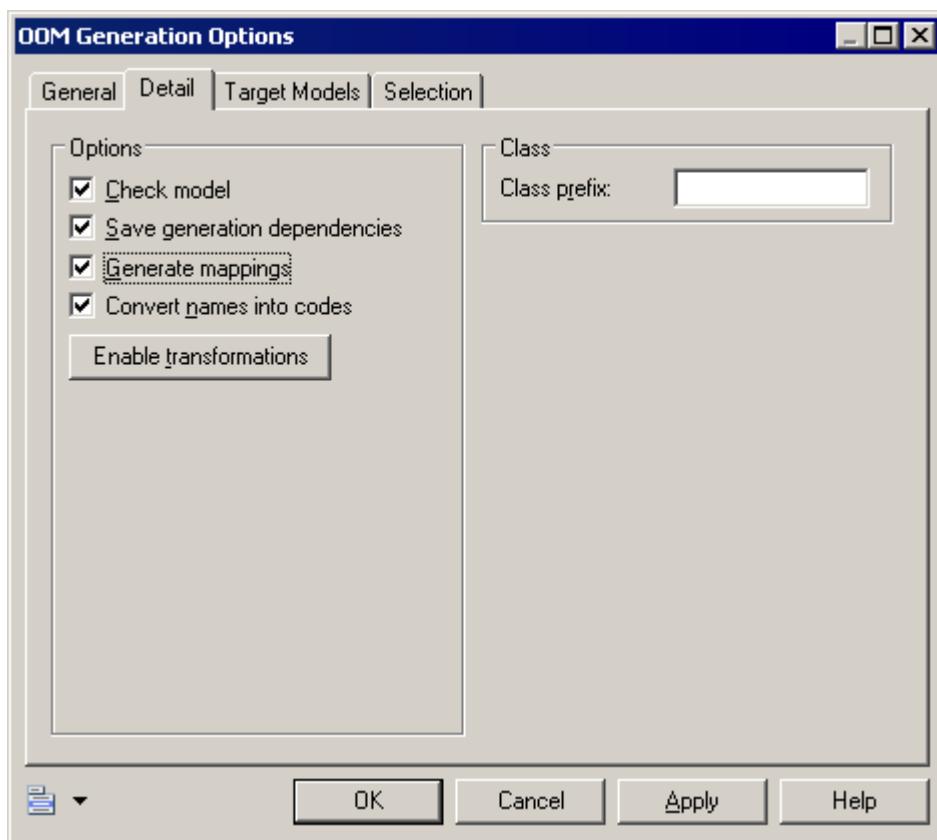
When you generate an object-oriented model from a data model:

- Each selected table is transformed into a persistent entity class and its:
  - Columns are transformed into persistent attributes.
  - Keys are transformed into identifiers.
  - Primary keys are transformed into primary identifiers.
- Reference keys have, by default, a cardinality of 0..\* and will be transformed into bidirectional many-to-many associations. To generate a one-to-one association, you need to set the maximum cardinality to 1 (cardinality 0..1 or 1..1). If the reference key is mandatory, the minimal multiplicity of one side of the generated association will be 1.

You cannot generate inheritance links from reference keys and tables.

## Procedure

1. Create your PDM (perhaps by reverse-engineering a database) and populate it with the appropriate tables and references.
2. Select **Tools > Generate Object-Oriented Model** to open the OOM Generation Options dialog.
3. On the *General* tab, specify the Object language type and the name and code of the OOM to generate (or select an existing OOM to update).
4. Click the *Detail* tab and select the **Generate Mappings** checkbox. You can optionally also specify a class prefix that will be applied to all generated classes.



5. Click the *Selection* tab and the select the tables that you want to transform into classes.
6. Click **OK** to generate (or update) your OOM.

## 2.8.3 Meet in the Middle: Manually Mapping Classes to Tables

If you have an existing OOM and PDM, you can define mappings between them manually using the Mapping Editor.

### Context

There are no constraints on the way you map your persistent classes. However, there are some well-defined mapping strategies, which are supported by most of O/R Mapping technologies. You should follow these strategies in order to build correct O/R Mapping models. However, minor differences still reside between them which we will raise when necessary.

Note: when your O/R Mapping models are related with a specific technology, for example when you are modeling for EJB 3.0 persistence, there will be some constraints and we provide model checks to help you check the syntax of the mappings you have defined.

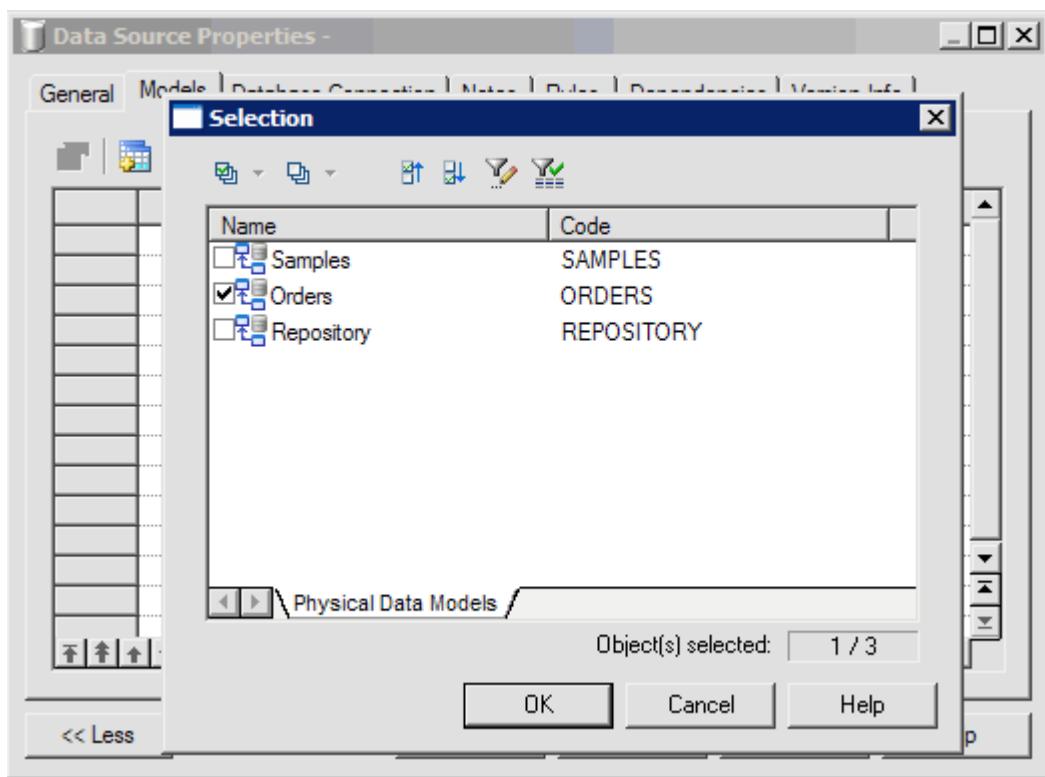
In order to define basic mappings, you have to define a data source for your OOM. Then you can define the mapping using the Mapping tab of the OOM object you want to map to a PDM object or using the Mapping Editor.

### Procedure

1. In the OOM, select **Model** **Data Sources** to open the corresponding list.
2. Click the *Add a Row* tool to create a data source.

You can create multiple data sources in the model.

3. Double-click the data source in the list to open its property sheet.
4. On the *Models* tab, click the *Add Models* tool to select one or more PDMS from the available open PDM as source models for the data source.



5. Define mappings using the *Mapping* tab or the Mapping Editor.

The Mapping Editor is more convenient to use as you can define all the mappings in one place just by some drag and drop actions. However, it is easy to understand the correspondence between OOM elements and PDM elements by using the *Mapping* tab in objects property sheet. So we will introduce you how to use *Mapping* definition tab to define mappings in the following sections.

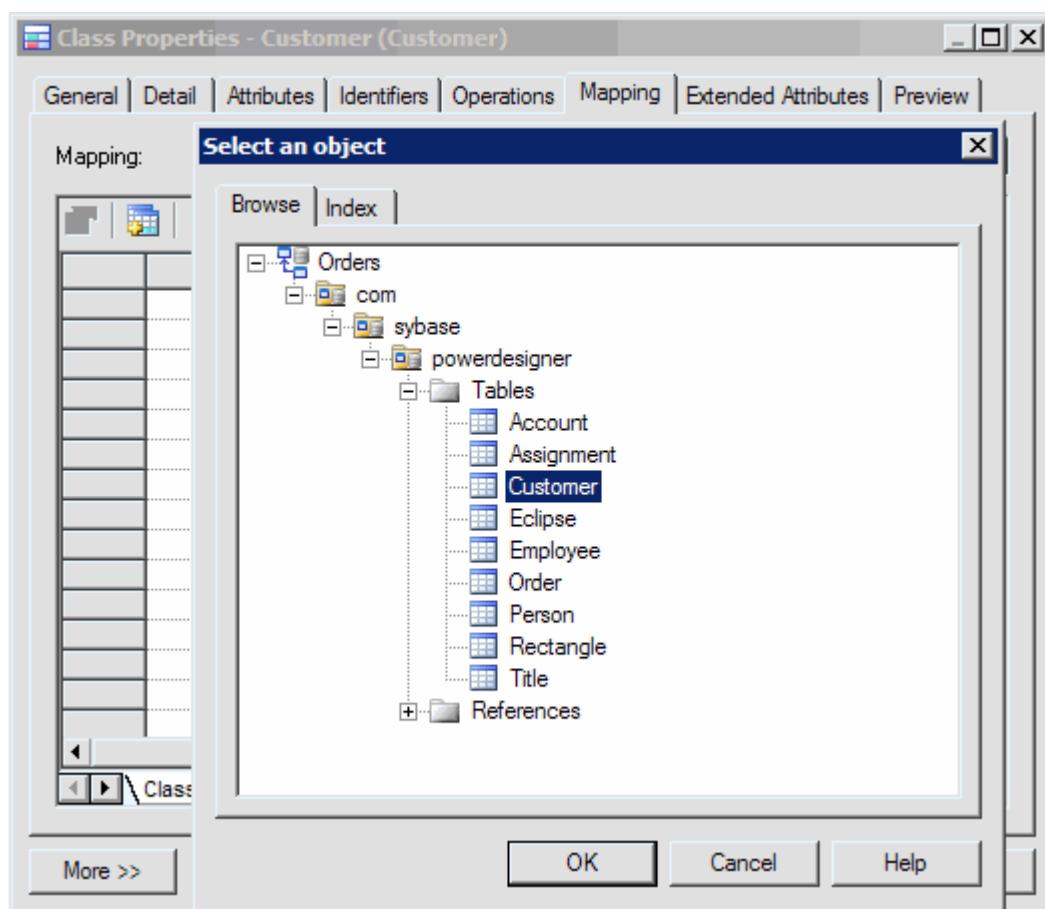
When you are familiar with O/R Mapping concepts, you can use the Mapping Editor.

### 2.8.3.1 Entity Class Mapping

In order to define mapping for entity classes, you have to:

- Open the Mapping tab of a class property sheet
- Click the Create Mapping to create a new class mapping
- In the Select an object dialog box, add a data model element as mapping source

You can also click the Add objects tool in the Class Sources sub-tab of the Mapping tab after you created the class mapping.



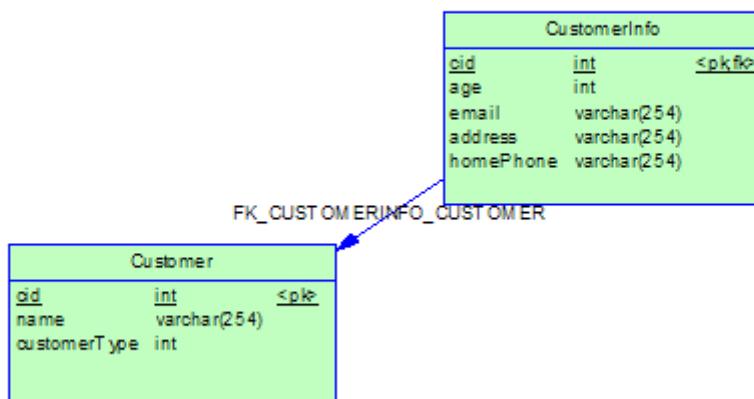
You can add tables, views and references as mapping sources. There are some constraints on views as mapping sources, as some views cannot be updated. When you add references as mapping sources, tables at the two ends will also be added.

You can add multiple tables as mapping sources. Usually, the first table you add is called the primary table. Other tables are called secondary tables. Each secondary table should have reference key referring to primary table, which is joined on its primary key.

Given the following class *Customer*:

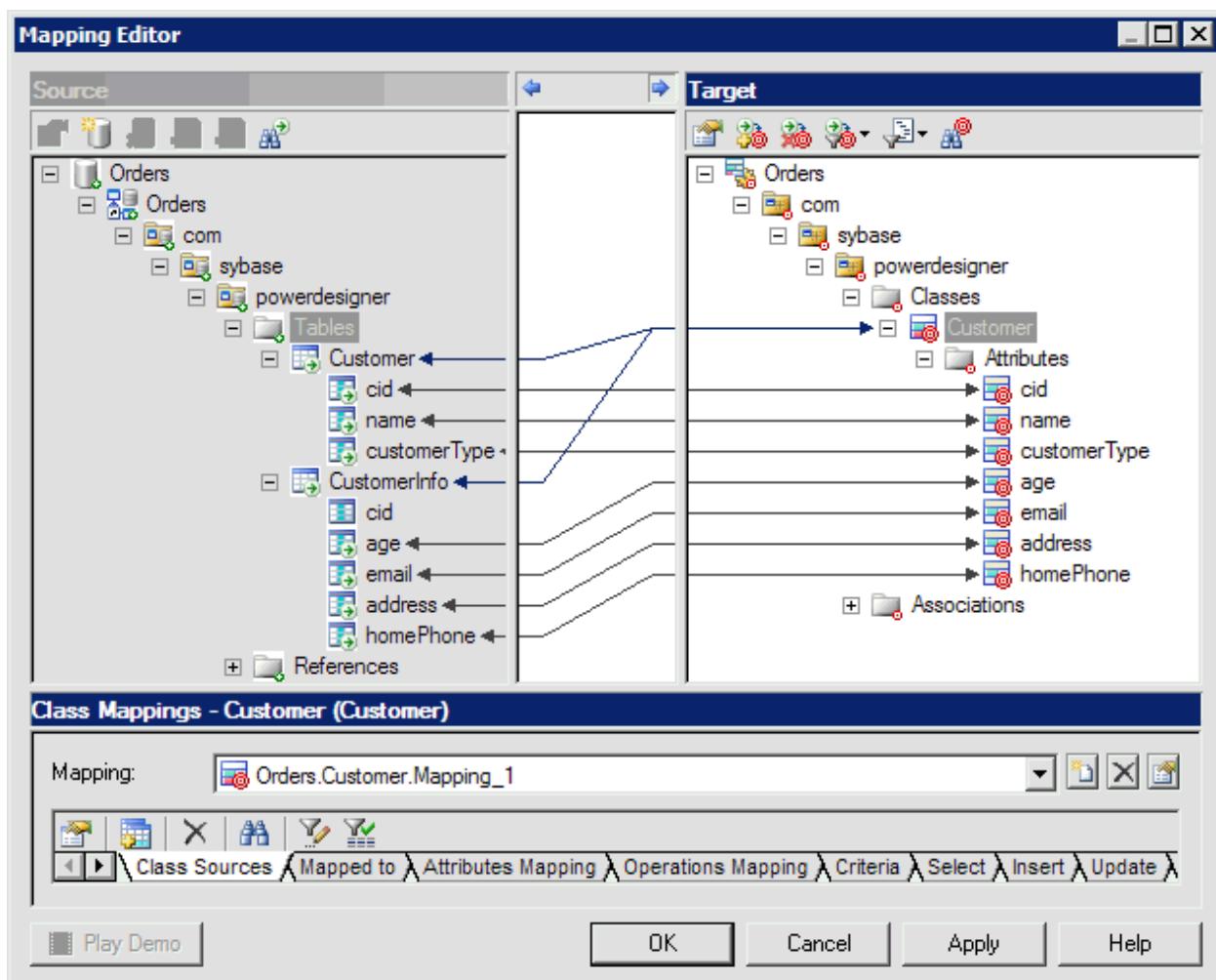
| Customer       |          |  |
|----------------|----------|--|
| - id           | : int    |  |
| - name         | : String |  |
| - customerType | : int    |  |
| - age          | : int    |  |
| - email        | : String |  |
| - address      | : String |  |
| - homePhone    | : String |  |

It can be mapped to two tables:



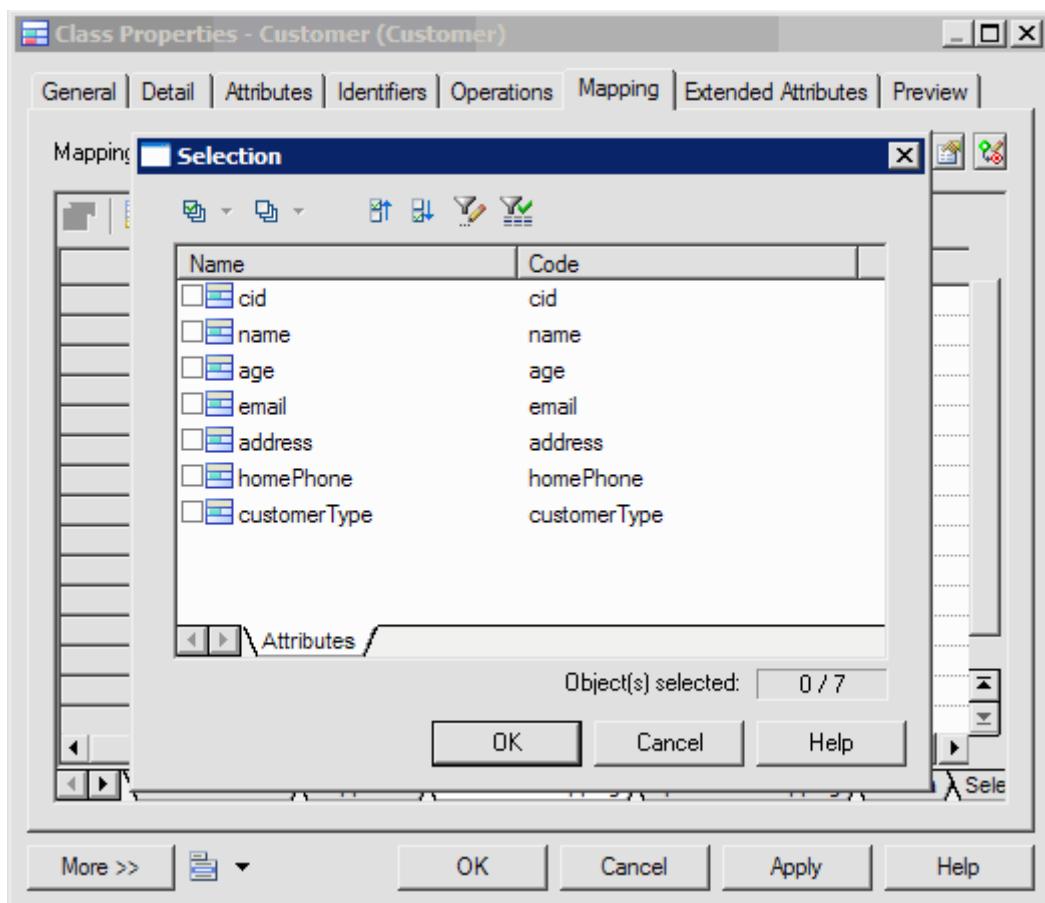
The *Customer* table is the primary table. The *CutomerInfo* table is the secondary table and it has one reference key referring to the primary table, which is joined on its primary key.

With the Mapping Editor, you just have to drag the two tables and drop them to class *Customer* to define class mappings.

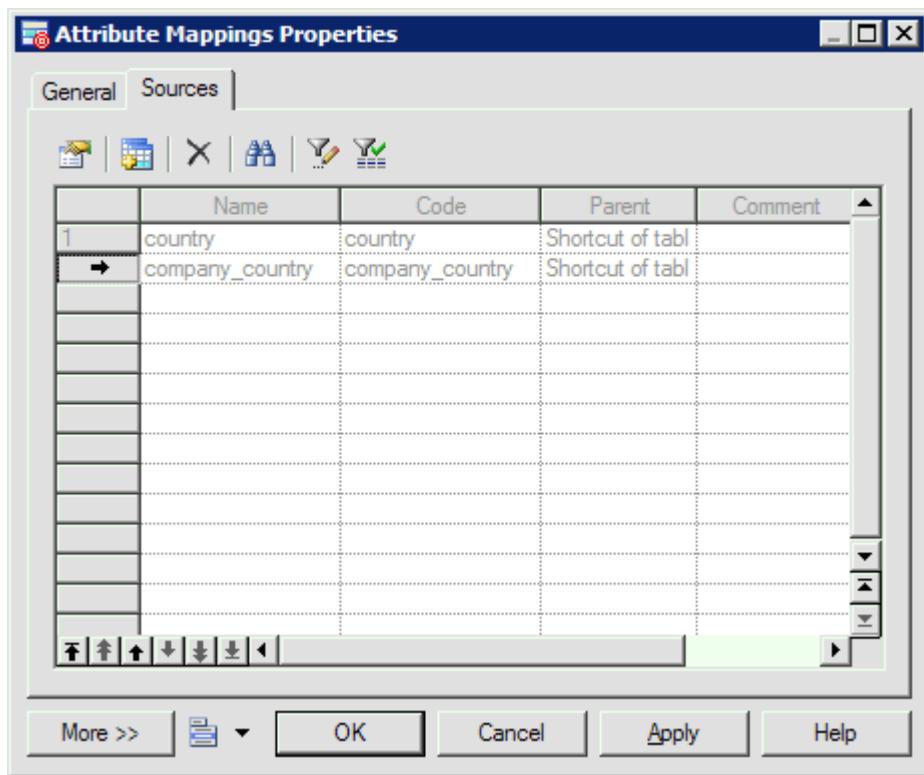


### 2.8.3.2 Attribute Mapping

After you have defined class mapping, you can define attribute mappings for the class in the *Attributes Mapping* sub-tab of the *Mapping* tab. PowerDesigner will generate some attribute mappings by matching their names with the column names. Click the *Add Mappings* tool and select the attributes you want to be mapped from the list.



For each attribute, you can select the column to which it is mapped from the list in the *Mapped to* column. Usually you just have to map each attribute to one column. However, you may need to map the attribute to multiple columns when you define attribute mappings for Value type class for example. In this case, you can open the attribute mappings property sheet and select the *Sources* tab to add multiple columns.



You can also map the attribute to a formula expression by defining it in the *Mapped to* box in the *General* tab. You can construct the formula using the SQL editor.

When an attribute has a Value type class as type, you do not need to define attribute mappings for it. You should instead define mapping for the Value type class.

### **2.8.3.3 Primary Identifier Mapping**

Columns of primary keys should be mapped to persistent attributes. Like primary keys for tables, you need to set these persistent attributes as primary identifiers of entity classes. The mapped primary keys should be primary keys of primary tables.

There are three types of primary identifier mapping:

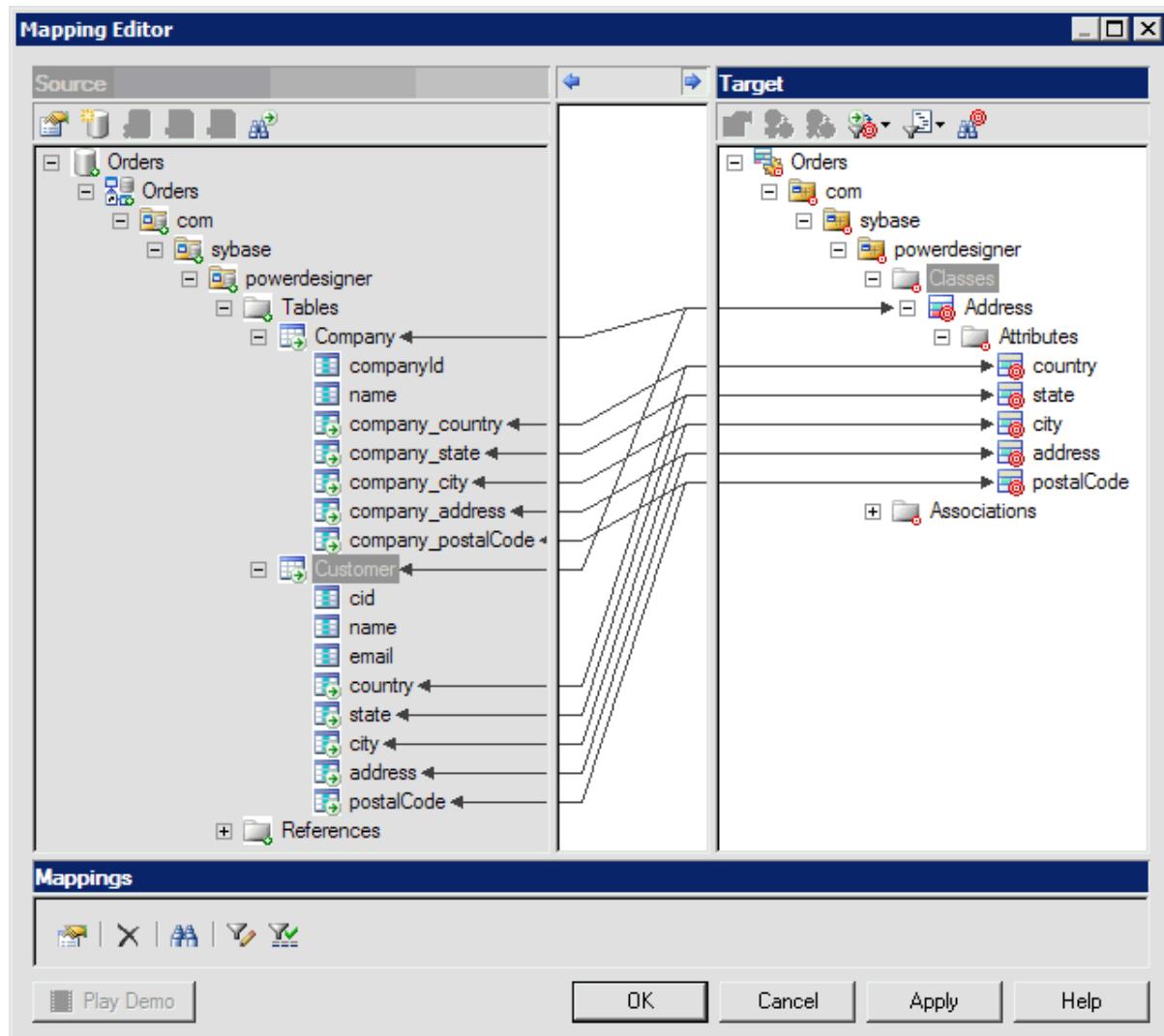
- Simple primary identifier mapping - the primary key is associated with only one column and the mapped primary identifier has one persistent attribute mapped to the column.
  - Composite primary identifier mapping - the primary key is associated with more than one column and the mapped primary identifier has the same number of persistent attributes mapped to the columns.  
Column(s) of primary keys can be mapped to associations (see [Association Transformation \[page 435\]](#)). They are migrated from primary keys of other tables.
  - Component primary identifier mapping - multiple persistent attributes are encapsulated into a value type class, and the mapped primary identifier contains one attribute whose type is the Value type class.  
Attributes of value type classes are mapped to columns, which are embedded in primary tables mapped by other entity classes. So you have to add primary tables of the containing classes as value type classes'

mapping sources. If the value type class is used in more than one entity class, you should map each of its persistent attributes to multiple columns of tables of these classes.

For example, Value type class *Address* is used as attribute type for two classes, *Product* and *Customer*. The attributes of the Value type class *Address* can be mapped to columns of two tables, *Company* table and *Customer* table:

| Classes   | Tables   |
|---|--|
| <pre> Customer + id : int - name : String - email : String - address : Address </pre> <pre> Company + id : int - name : String - address : Address </pre> | <pre> Customer id int &lt;pk&gt; name varchar(254) email varchar(254) country varchar(254) state varchar(254) city varchar(254) address varchar(254) postalCode varchar(254) </pre> <pre> Company companyId int &lt;pk&gt; name varchar(254) company_country varchar(254) company_state varchar(254) company_city varchar(254) company_address varchar(254) company_postalCode varchar(254) </pre> |

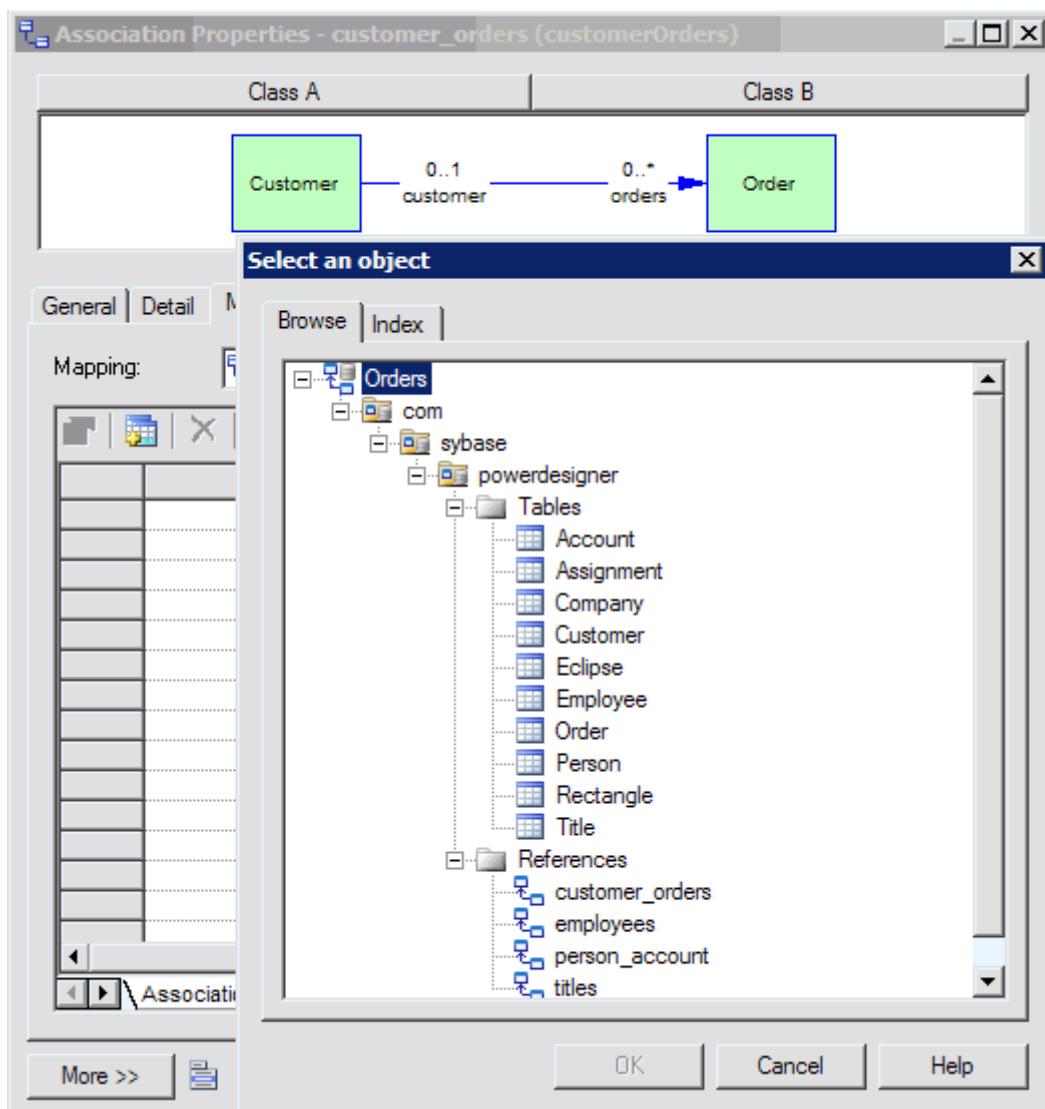
The mapping is easier to visualize in the Mapping Editor.



Primary identifier mapping is mandatory for entity classes.

### 2.8.3.4 Association Mapping

You can define association mapping in the Mapping tab of the association property sheet and select the Add Objects tool to add mapping sources.



Associations defined between entity classes can be mapped to reference keys or tables. In order to define association mapping, you have to add the references keys or tables as mapping sources. When you add reference keys, the tables on their ends will also be added.

Associations can be classified as one-to-one, one-to-many and many-to-many according to multiplicities of ends.

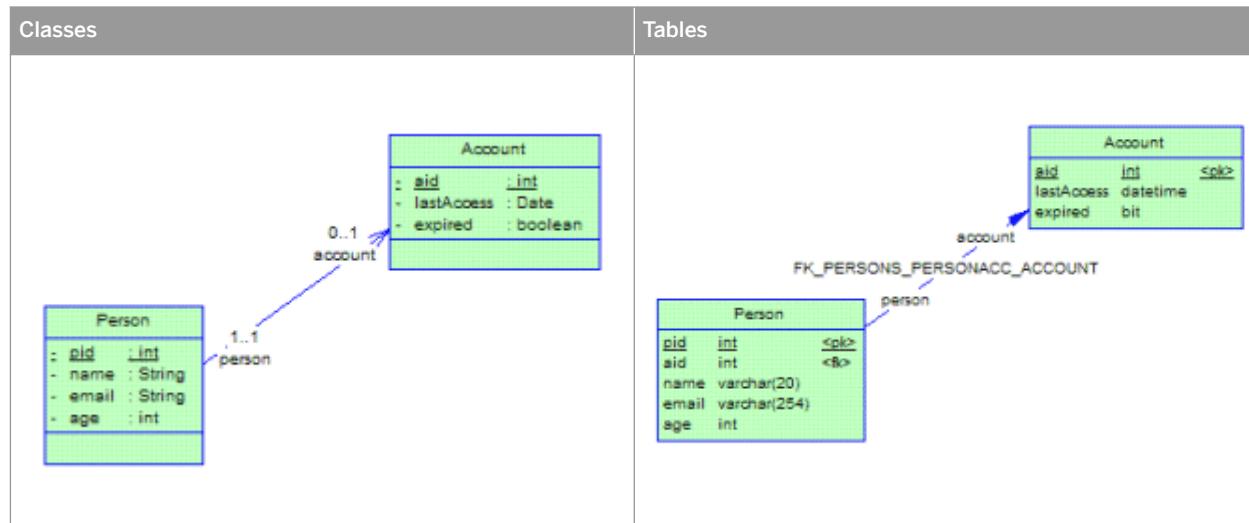
And associations can be classified as unidirectional and bi-directional according to navigability of both ends.

Associations of different types should be mapped in different ways. We will introduce them in detail in the following sections.

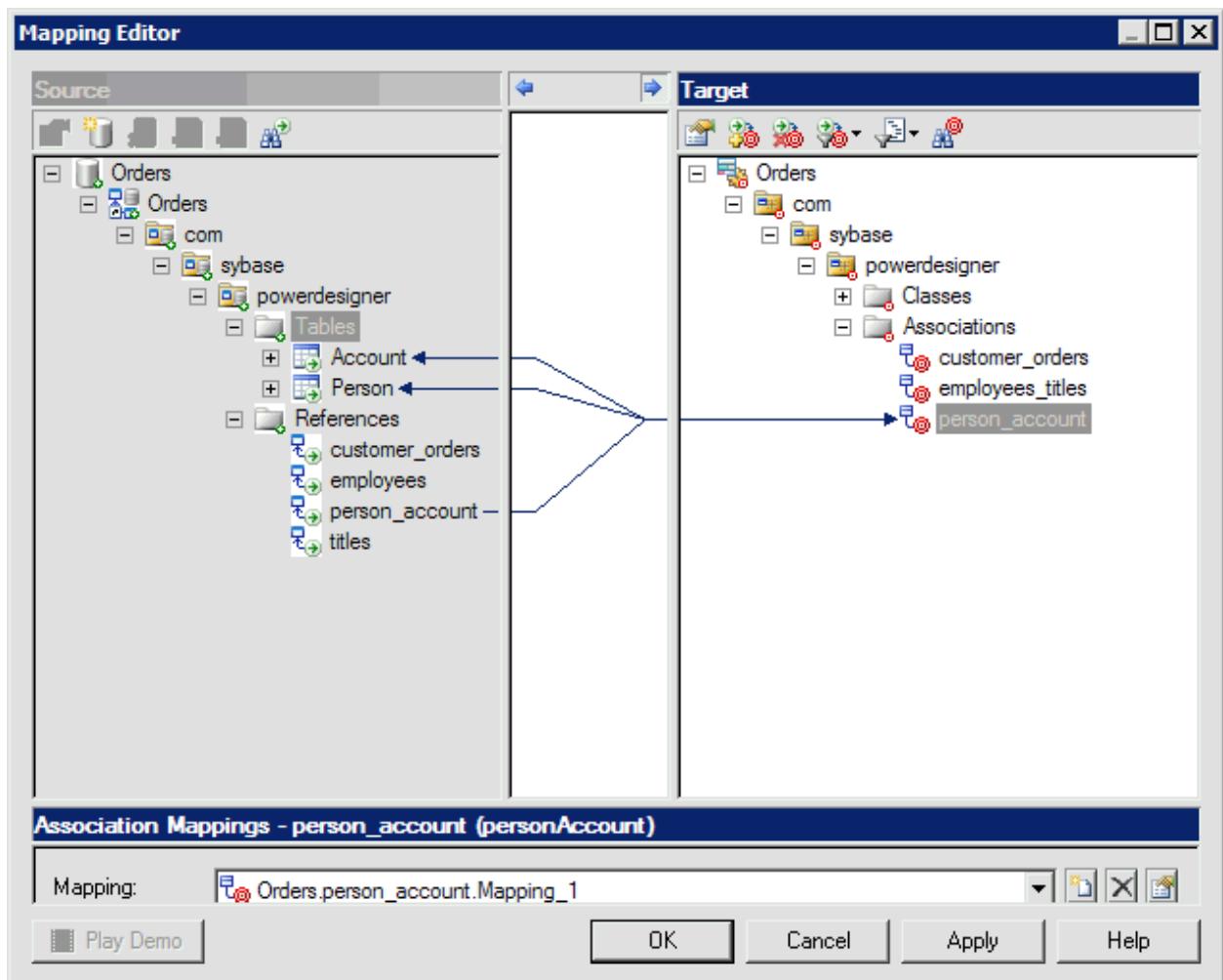
## 2.8.3.4.1 One-to-One Association Mapping Strategy

You can map each unidirectional one-to-one association to a reference key. The foreign key should have the same direction as the association.

In the following example, there are two entity classes, *Person* and *Account*, and a one-to-one association between them. The association is unidirectional and navigates from the entity class *Person* to the entity class *Account*.



The association and the reference key are linked in the Mapping Editor.

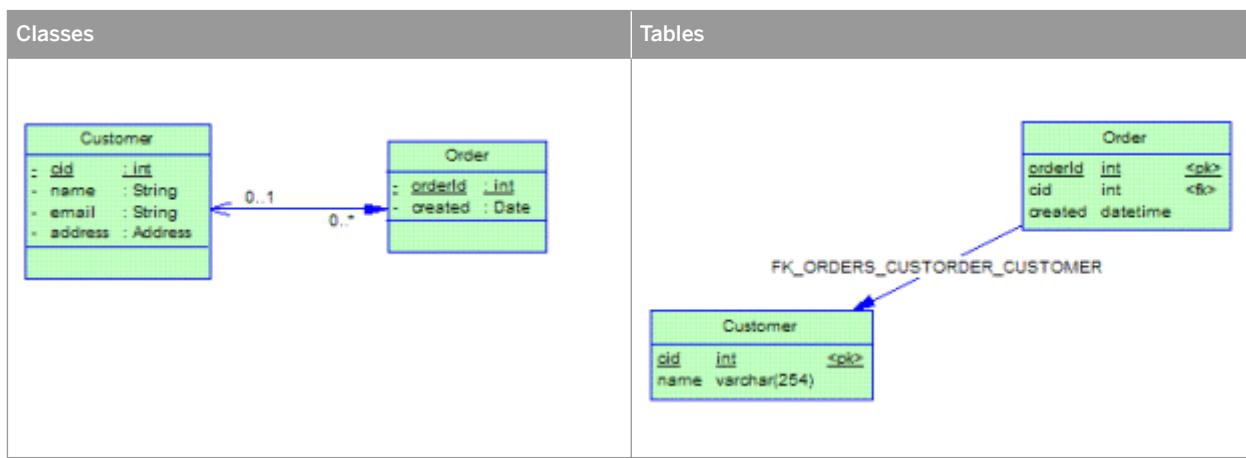


For a bi-directional one-to-one association, you also just can map it to one reference key. But the reference can navigate in either direction.

### 2.8.3.4.2 One-to-Many Association Mapping Strategy

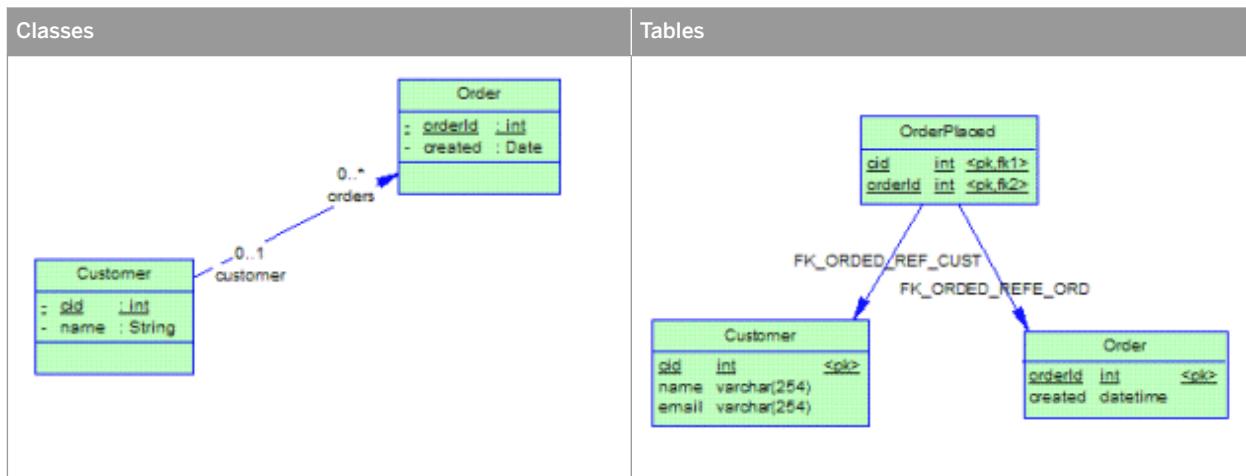
Each unidirectional many-to-one association is mapped to a reference that has the same direction as the association.

In the following example, a unidirectional many-to-one association defined between the class *Customer* and the class *Order* is mapped to the reference key:



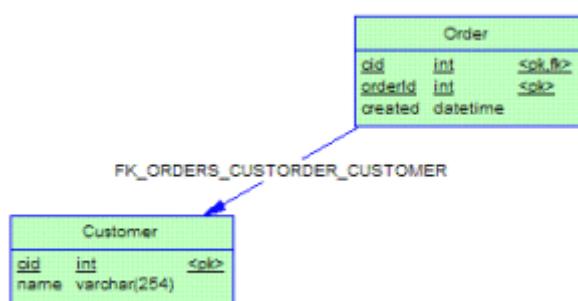
Each unidirectional one-to-many association should be mapped to a middle table and two references that refer to tables mapped by the entity classes on both ends.

In the following example, the association defined between *Customer* and *Order* is a unidirectional one-to-many association mapped to a middle table and reference keys:

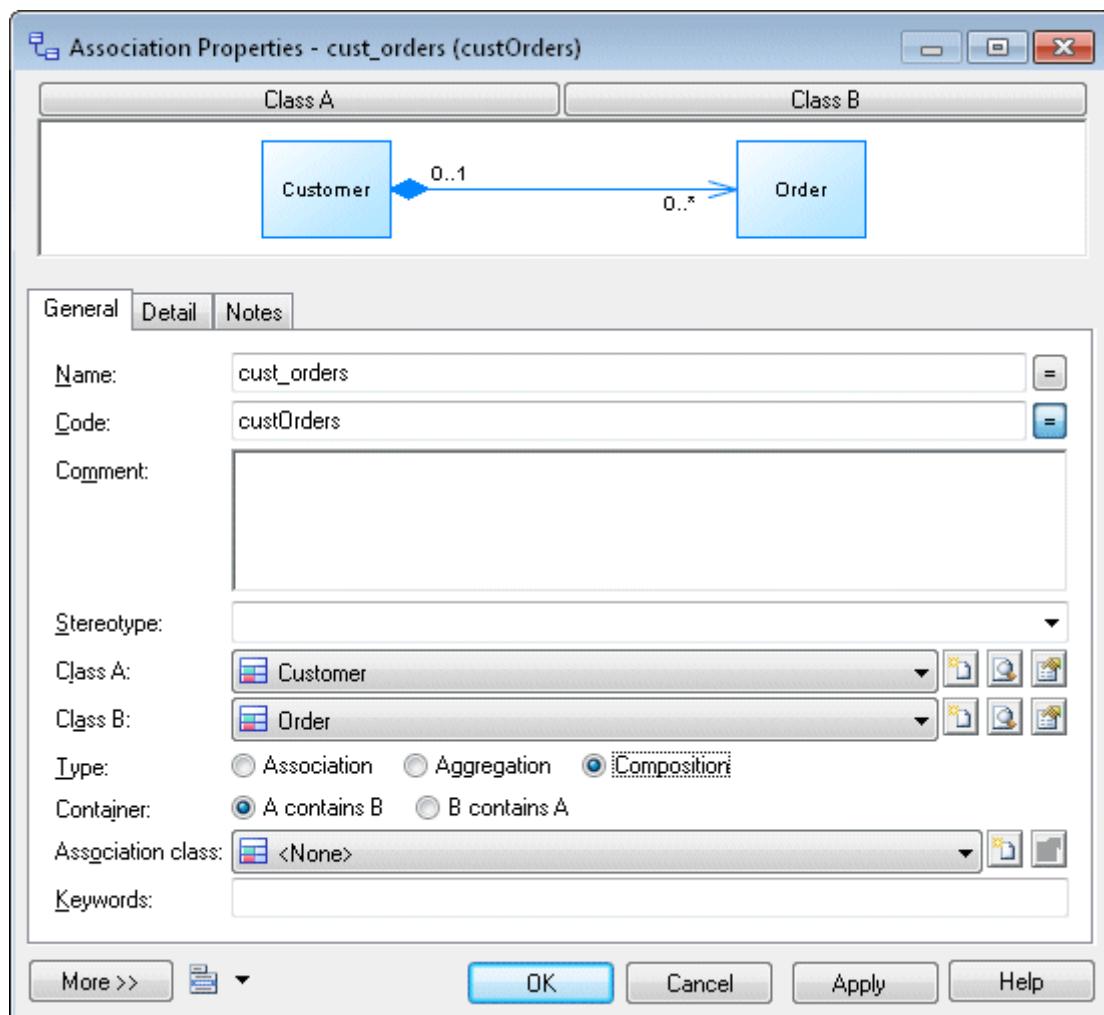


You can map a bi-directional one-to-many association as unidirectional many-to-one association. The reference just can navigate from primary table of class on multiple-valued side to primary table of class on single-valued side.

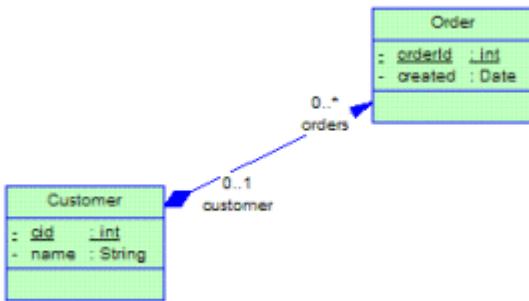
Sometimes we want to make the primary key of parent table be part of primary key of the child table and reference key join on the migrated column(s). For example we can map *Customer*, *Order* and bi-directional one-to-many association to tables and reference key as follows:



In order to define such type of association mapping, you have to define the association as composition with the class on single-valued side containing the class on multiple-valued side first.



The association is the following:

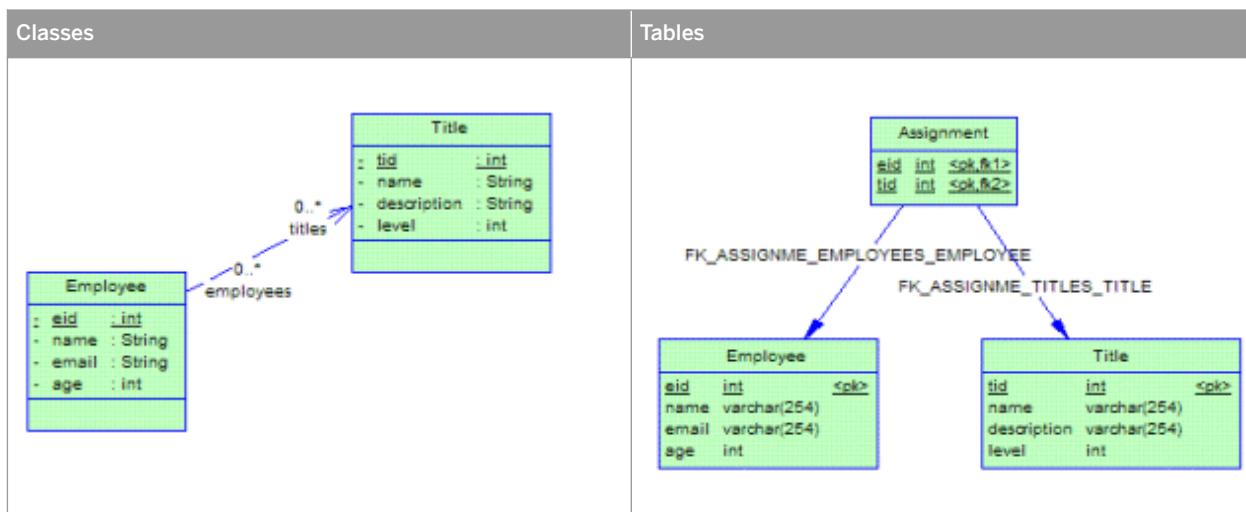


Then add the reference as mapping sources. You just can define the same way association mapping for bi-directional one-to-many association.

### 2.8.3.4.3 Many-to-Many Association Mapping Strategy

Each many-to-many association is mapped to a middle table and two reference keys that refer to tables mapped by entity classes on the two ends.

In the following example a many-to-many association defined between the class *Employee* and the class *Title* is mapped to a middle table and references:



## 2.8.3.5 Defining Inheritance Mapping

Inheritance can be mapped using a table per class, joined subclass, or table per class hierarchy inheritance mapping strategy. You can apply any of these inheritance mapping strategies or mix them. You should define primary identifier on the entity class that is the root of the entity hierarchy.

### 2.8.3.5.1 Table Per Class Hierarchy Inheritance Mapping Strategy

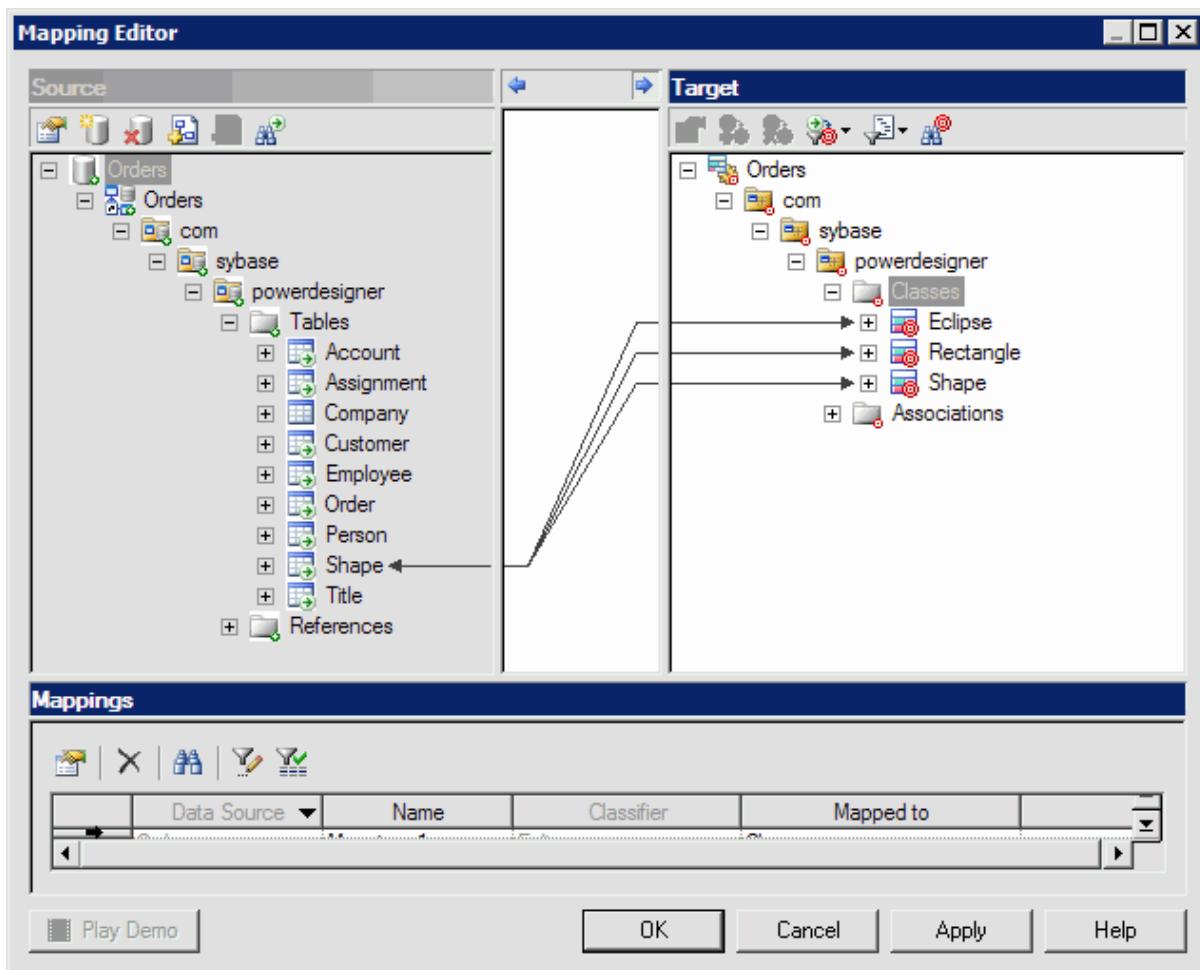
The whole class hierarchy is mapped to one table. One character based type or integer type discriminator column is defined to distinguish instances of difference classes in the hierarchy.

#### Context

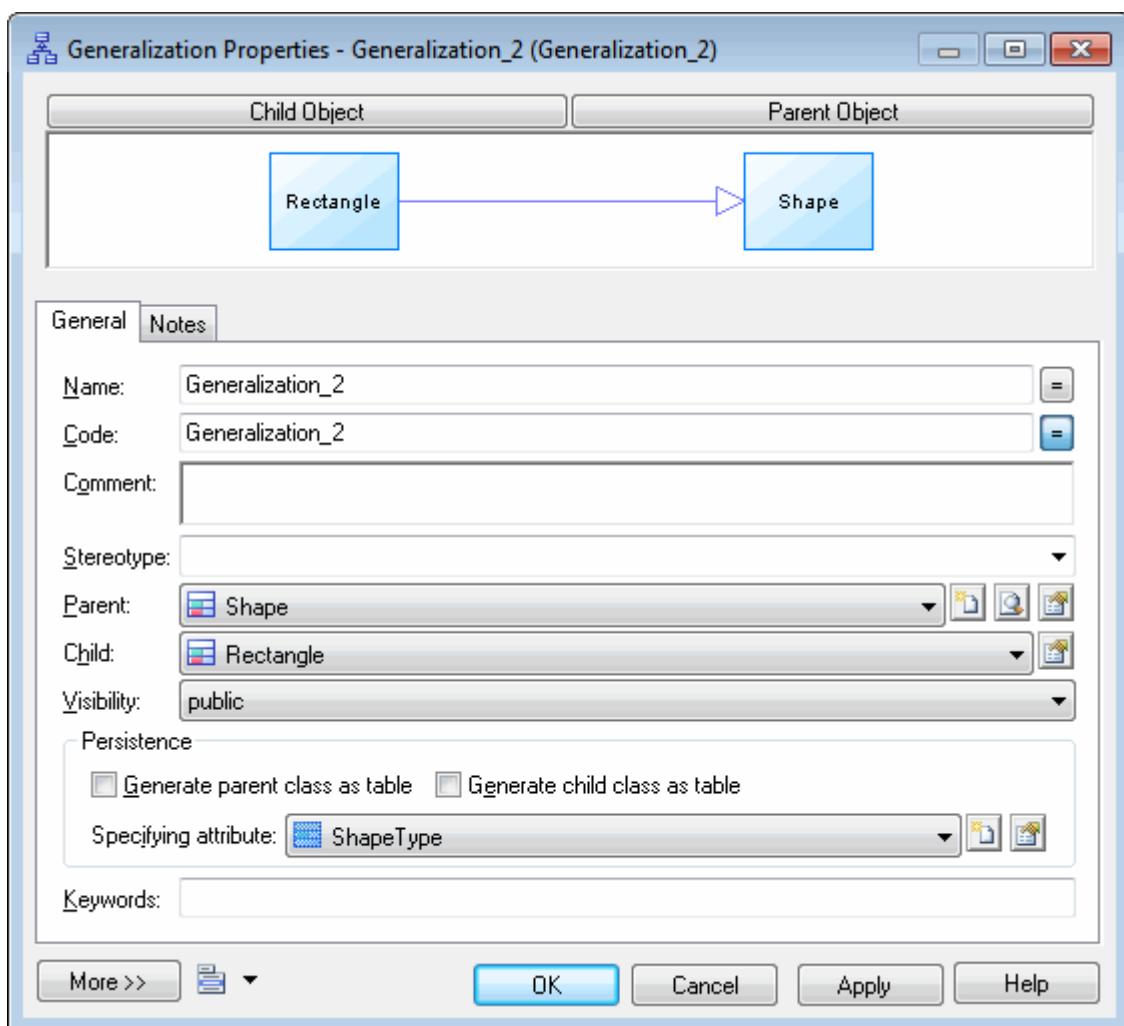
| Classes  | Tables   |
|--|--|
| <pre>classDiagram     class Shape {         +id : int         +title : String     }     class Rectangle {         -width : float         -height : float     }     class Eclipse {         -radiusX : float         -radiusY : float     }     Shape &lt; -- Rectangle     Shape &lt; -- Eclipse</pre> | <pre>tableShape {     id Integer &lt;pk&gt;     title varchar(254)     shapeType varchar(254)     width float     height float     radiusX float     radiusY float }</pre> |

#### Procedure

1. Define class mappings for each class in the hierarchy so that all the classes have the same primary table. They can also be mapped to other secondary tables:



2. Define identifier mapping in the root class.
3. Define attribute mappings or association mappings for each class.
4. Select one of the attributes in the root class, as the Specifying Attribute in the property sheet of one of the children inheritance links of the root class to specify it as a discriminator column, which is used to distinguish between different class instances. In the following example, we define one extra attribute *shapeType* in *Shape* and select it as discriminator attribute:

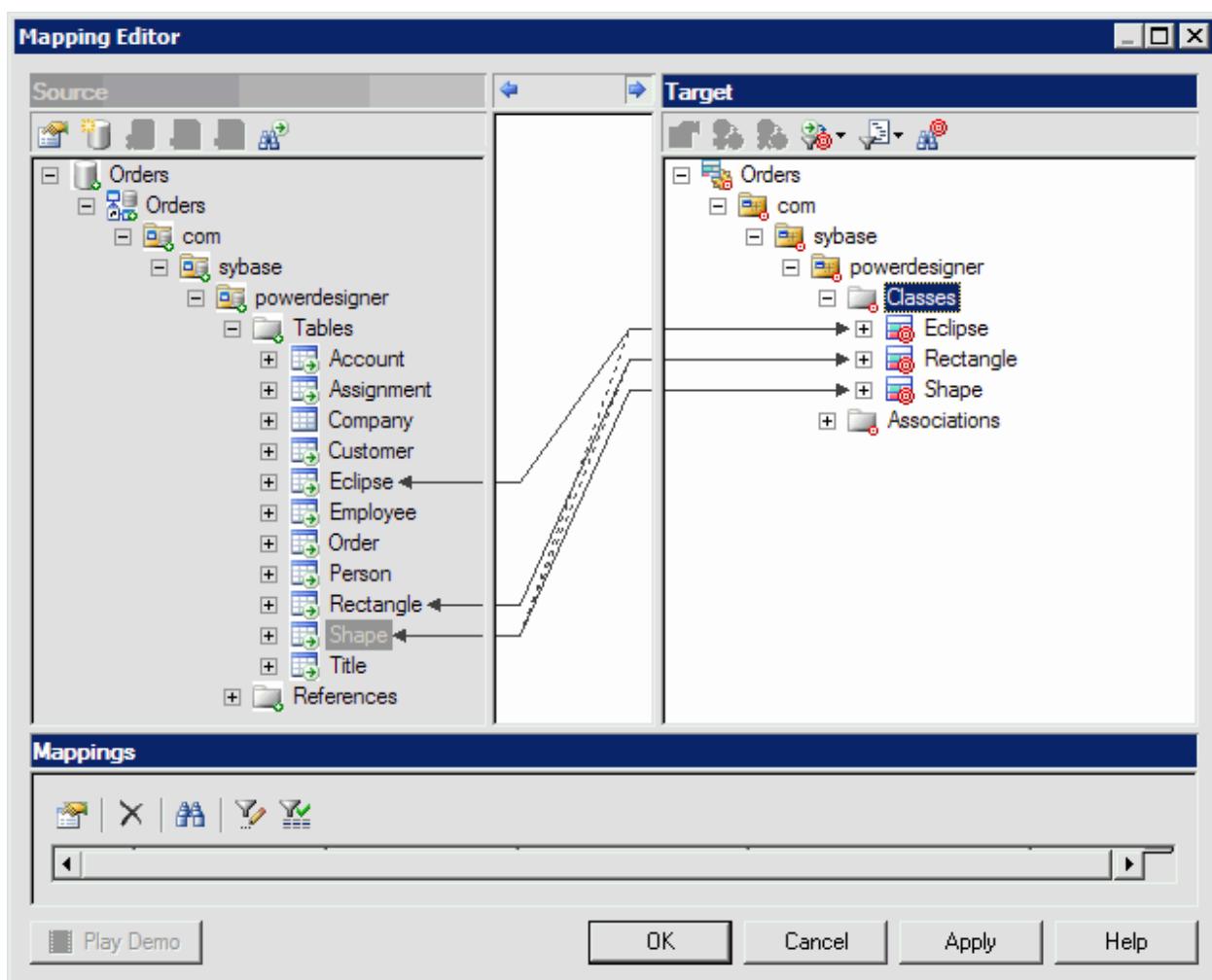


5. Define persistence generation type for each class. Define the persistence generation type of the root class as *Generate table* and all the other classes as *Migrate columns*.

### 2.8.3.5.2 Joined Subclass Inheritance Mapping Strategy

Each entity class is mapped to its own primary table. Each primary table has a reference key referring to a primary table of its parent class except for the primary table of the root class. The reference key should join on the primary key of the primary table.

#### Context



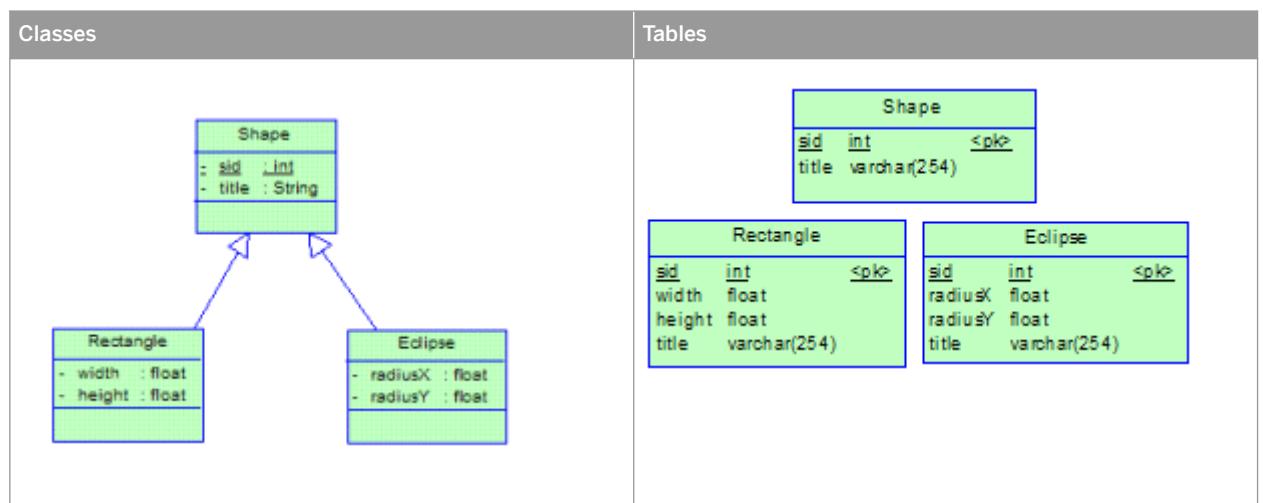
## Procedure

1. Define class mappings for each class in the hierarchy. Each class is mapped to its own primary table.
2. Define identifier mapping in the root class.
3. Define attribute mappings or association mappings for each class.
4. Define persistence generation type for each class.
5. Define persistence generation type of all the classes as *Generate table*.

### 2.8.3.5.3 Table Per Class Inheritance Mapping Strategy

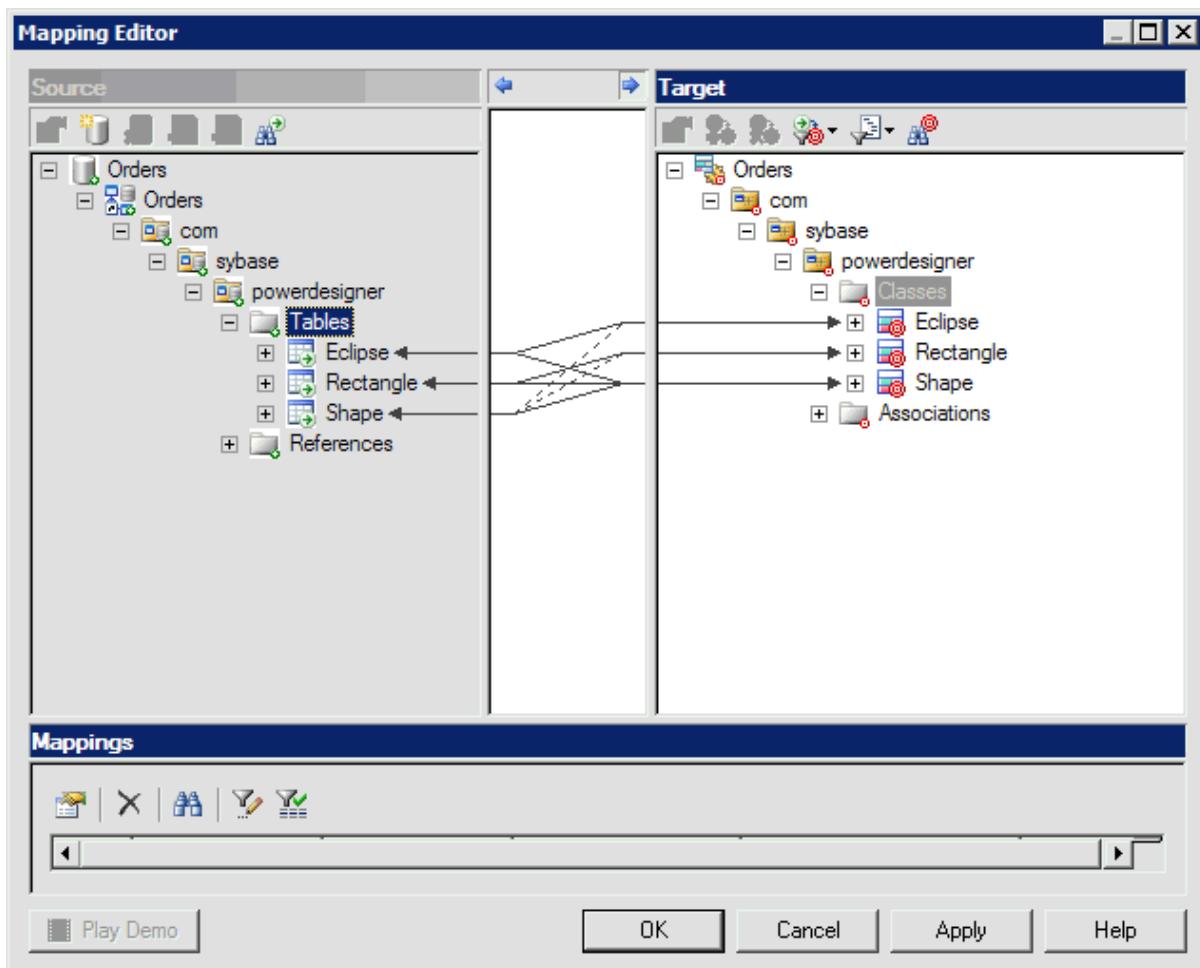
Each class is mapped to its own primary table. All persistent attributes of the class, including inherited persistent attributes, are mapped to columns of the table.

#### Context



#### Procedure

1. Define entity class mappings for each class in the hierarchy, mapping each class to its own primary table:



2. Define attribute mappings and association mappings for each class.
3. Define identifier mapping in the root class.
4. Define persistence generation type for each class.
5. Define persistence generation type of leaf classes as *Generate table* and all the other classes as *Migrate columns*.

#### **i** Note

Super classes can be also mapped to primary tables of subclasses if inherited persistent attributes are mapped in different ways for subclasses, for example to different columns. The other primary table can just be secondary tables. PowerDesigner will generate these secondary tables for super classes.

For this kind of strategy, some super classes can have no table mapped. These classes are used to define state and mapping information that can be inherited by their subclasses.

## 2.9 Generating Persistent Objects for Java and JSF Pages

PowerDesigner supports the generation of persistent objects for Hibernate and EJB3, as well as JavaServer Faces for Hibernate.

### 2.9.1 Generating Hibernate Persistent Objects

Hibernate is an open source project developed by JBoss, which provides a powerful, high performance and transparent object/relational persistence and query solution for Java. PowerDesigner can generate Hibernate O/R mapping files for you from your Java OOM.

To enable Hibernate extensions in your model, select Model > Extensions, click the *Attach an Extension* tool, select the `Hibernate` file on the *O/R Mapping* tab, and click *OK* to attach it.

Hibernate lets you develop persistent objects using POJO (Plain Old Java Object). All the common Java idioms, including association, inheritance, polymorphism, composition, and the Java collections framework are supported. Hibernate allows you to express queries in its own portable SQL extension (HQL), as well as in native SQL, or with Java-based Criteria and Example objects.

PowerDesigner supports the design of Java classes, database schema and Object/Relational mapping (O/R mapping). Using these metadata, PowerDesigner can generate Hibernate persistent objects including:

- Persistent Java classes (domain specific objects)
- Configuration file
- O/R mapping files
- DAO factory
- Data Access Objects (DAO)
- Unit test classes for automated test

#### 2.9.1.1 Defining the Hibernate Default Options

You can define these options in the model or a package property sheet.

#### Procedure

1. Open the model or a package property sheet, and click the *Hibernate Default Options* tab:
2. Define the model or package level default options.

## Results

| Option                        | Description  |
|-------------------------------|--|
| Auto import                   | Specifies that users may use an unqualified class name in queries. |
| Default access                | Specifies the default class attribute access type.                 |
| Specifies the default cascade | Specifies the default cascade type.                                |
| Schema name                   | Specifies the default database schema name.                        |
| Catalog name                  | Specifies the default database catalog name.                       |

### 2.9.1.2 Defining the Hibernate Database Configuration Parameters

Hibernate can support multiple databases. You need to define database configuration parameters. The database configuration parameters are stored in the configuration file, hibernate.cfg.xml .

## Procedure

1. Open the model property sheet and click the *Hibernate Configuration* tab.
2. Define the type of database, JDBC driver, connection URL, JDBC driver jar file path, user name, password, etc.

## Results

| Option            | Description   |
|-------------------|---|
| Dialect           | Specifies the dialect, and hence the type of database.<br>Scripting name: dialect |
| JDBC driver class | Specifies the JDBC driver class.<br>Scripting name: connection.driver_class       |
| Connection URL    | Specifies the JDBC connection URL string.<br>Scripting name: connection.url       |

| Option               | Description  |
|----------------------|--|
| JDBC driver jar      | Specifies the JDBC driver jar file path.<br>Scripting name: N/A  |
| User name            | Specifies the database user name.<br>Scripting name: <code>connection.username</code>                    |
| Password             | Specifies the database user password.<br>Scripting name: <code>connection.password</code>                |
| Show SQL             | Specifies that SQL statements should be shown in the log.<br>Scripting name: <code>show_sql</code>       |
| Auto schema export   | Specifies the mode of creation from tables.<br>Scripting name: <code>hbm2ddl.auto</code>                 |
| Package prefix       | Specifies a namespace prefix for all the packages in the model.<br>Scripting name: N/A                   |
| Connection pool size | Specifies the maximum number of pooled connections.<br>Scripting name: <code>connection.pool_size</code> |

You can verify the configuration parameters in the [Preview](#) tab.

### 2.9.1.3 Defining Hibernate Basic O/R Mappings

There are two kinds of classes in Hibernate, entity classes and value type classes. Entity classes have their own database identities, mapping files and life cycles, while value type classes don't have. Value type classes depend on entity classes. Value type classes are also called component classes.

Hibernate uses mapping files to define the mapping metadata. Each mapping file `<Class>.hbm.xml` can contain metadata for one or many classes. PowerDesigner uses the following grouping strategy:

- A separate mapping file is generated for each single entity class that is not in an inheritance hierarchy.
- A separate mapping file is generated for each inheritance hierarchy that has a unique mapping strategy. All mappings of subclasses are defined in the mapping file. The mapping file is generated for the root class of the hierarchy. See [Defining Hibernate Inheritance Mappings \[page 477\]](#) for details about how the mapping strategy is determined.
- No mapping file is generated for a single value type class that is not in an inheritance hierarchy. Its mapping is defined in its owner's mapping file.

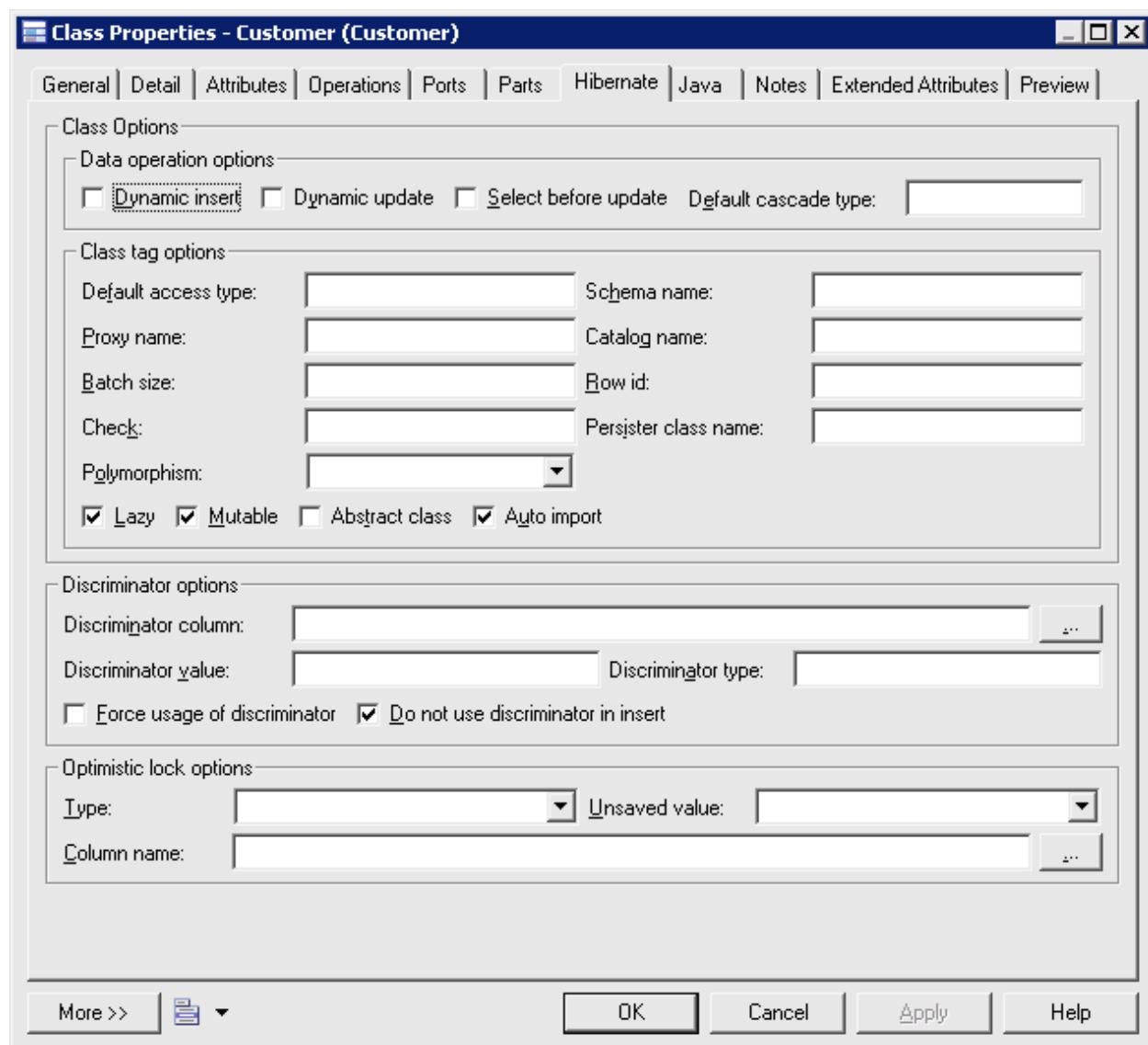
## 2.9.1.3.1 Defining Hibernate Class Mapping Options

Classes can be mapped to tables or views. Since views have many constraints and limited functionality (for example they do not have primary keys and reference keys), some views cannot be updated, and the mappings may not work properly in some cases.

There are some conditions that need to be met in order to generate mapping for a specific class:

- The Java source can be generated. This may not be possible if, for example, the visibility of the class is private.
- The class is persistent.
- The generation mode is not set to Generate ADT (abstract data type).
- If the class is an inner class, it must be static, and have public visibility. Hibernate should then be able to create instances of the class.

Hibernate-specific class mapping options can be defined in the *Hibernate* tab of the class property sheet:



| Option               | Description  |
|----------------------|--|
| Dynamic insert       | <p>Specifies that INSERT SQL should be generated at runtime and will contain only the columns whose values are not null.</p> <p>Scripting name: <code>dynamic-insert</code></p>      |
| Dynamic update       | <p>Specifies that UPDATE SQL should be generated at runtime and will contain only the columns whose values have changed.</p> <p>Scripting name: <code>dynamic-update</code></p>      |
| Select before update | <p>Specifies that Hibernate should never perform a SQL UPDATE unless it is certain that an object is actually modified.</p> <p>Scripting name: <code>select-before-update</code></p> |
| Default cascade type | <p>Specifies the default cascade style.</p> <p>Scripting name: <code>default-cascade</code></p>  |
| Default access type  | <p>Specifies the default access type (field or property)</p> <p>Scripting name: <code>default-access</code></p>  |
| Proxy name           | <p>Specifies an interface to use for lazy initializing proxies.</p> <p>Scripting name: <code>proxy</code></p>  |
| Batch size           | <p>Specifies a "batch size" for fetching instances of this class by identifier.</p> <p>Scripting name: <code>batch-size</code></p>   |
| Check                | <p>Specifies a SQL expression used to generate a multi-row check constraint for automatic schema generation.</p> <p>Scripting name: <code>check</code></p>                           |
| Polymorphism         | <p>Specifies whether implicit or explicit query polymorphism is used.</p> <p>Scripting name: <code>polymorphism</code></p>   |
| Schema name          | <p>Specifies the name of the database schema.</p> <p>Scripting name: <code>schema</code></p>   |
| Catalog name         | <p>Specifies the name of the database catalog.</p> <p>Scripting name: <code>catalog</code></p>   |
| Row id               | <p>Specifies that Hibernate can use the ROWID column on databases which support it (for example, Oracle).</p> <p>Scripting name: <code>rowid</code></p>                              |

| Option                             | Description  |
|------------------------------------|--|
| Persister class name               | <p>Specifies a custom persistence class.</p> <p>Scripting name: <code>persister</code></p>   |
| Lazy                               | <p>Specifies that the class should be lazy fetching.</p> <p>Scripting name: <code>lazy</code></p>  |
| Mutable                            | <p>Specifies that instances of the class are mutable.</p> <p>Scripting name: <code>mutable</code></p>  |
| Abstract class                     | <p>Specifies that the class is abstract.</p> <p>Scripting name: <code>abstract</code></p>  |
| Auto import                        | <p>Specifies that an unqualified class name can be used in a query</p> <p>Scripting name: <code>Auto-import</code></p>   |
| Discriminator column               | <p>Specifies the discriminator column or formula for polymorphic behavior in a one table per hierarchy mapping strategy.</p> <p>Scripting name: <code>discriminator</code></p> |
| Discriminator value                | <p>Specifies a value that distinguishes individual subclasses, which are used for polymorphic behavior.</p> <p>Scripting name: <code>discriminator-value</code></p>            |
| Discriminator type                 | <p>Specifies the discriminator type.</p> <p>Scripting name: <code>type</code></p>  |
| Force usage of discriminator       | <p>Forces Hibernate to specify allowed discriminator values even when retrieving all instances of the root class.</p> <p>Scripting name: <code>force</code></p>                |
| Do not use discriminator in insert | <p>Forces Hibernate to not include the column in SQL INSERTs</p> <p>Scripting name: <code>insert</code></p>  |
| Optimistic lock type               | <p>Specifies an optimistic locking strategy.</p> <p>Scripting name: <code>optimistic-lock</code></p>   |
| Optimistic lock column name        | <p>Specifies the column used for optimistic locking. A field is also generated if this option is set.</p> <p>Scripting name: <code>version/ timestamp</code></p>               |
| Optimistic lock unsaved value      | <p>Specifies whether an unsaved value is null or undefined.</p> <p>Scripting name: <code>unsaved-value</code></p>  |

## 2.9.1.3.2 Defining Primary Identifier Mappings

Primary identifier mapping is mandatory in Hibernate. Primary identifiers of classes are mapped to primary keys of master tables in data sources. If not defined, a default primary identifier mapping will be generated, but this may not work properly.

There are three kinds of primary identifier mapping in Hibernate:

- Simple identifier mapping
- Composite identifier mapping
- Component identifier mapping

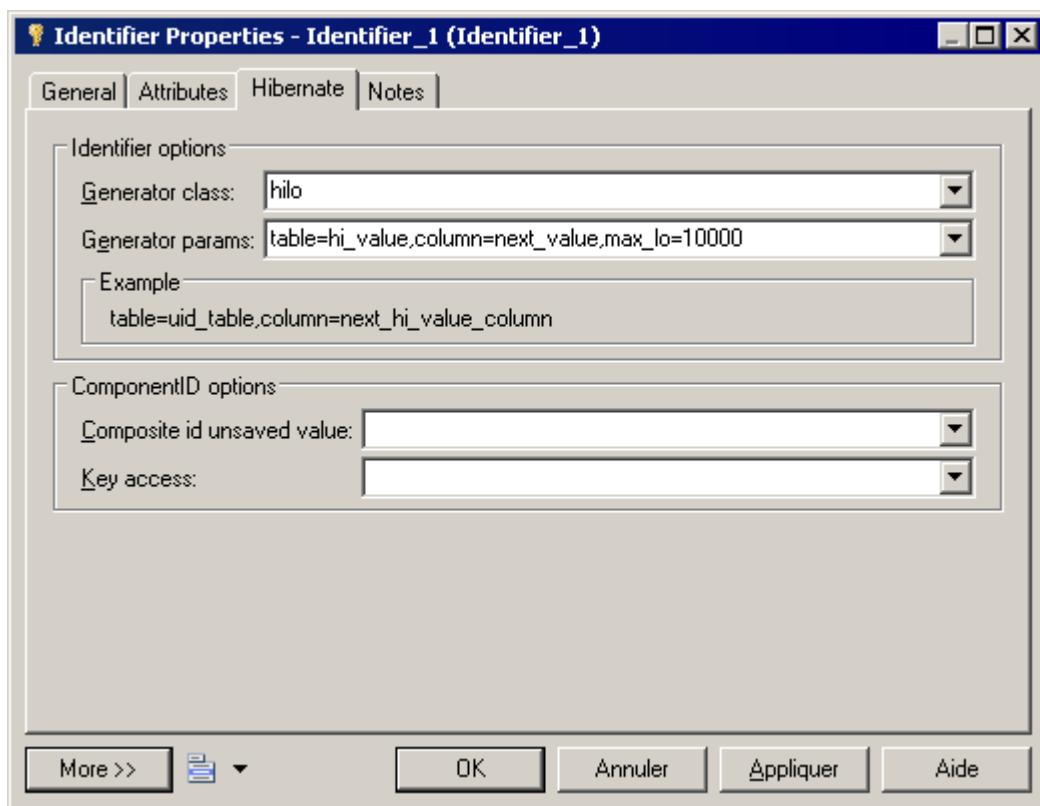
Mapped classes must declare the primary key column of the database table. Most classes will also have a Java-Beans-style property holding the unique identifier of an instance.

### 2.9.1.3.2.1 Simple Identifier Mapping

When a primary key is attached to a single column, only one attribute in the primary identifier can be mapped. This kind of primary key can be generated automatically. You can define the generator class and parameters. There are many generator class types, such as increment, identity, sequence, etc. Each type of generator class may have parameters that are meaningful to it. See your Hibernate documentation for detailed information.

#### Context

You can define the generator class and parameters in the *Hibernate* tab of the primary identifier property sheet. The parameters take the form of param1=value1; param2=value2.



## Procedure

1. Open the class property sheet and click the *Attributes* tab.
2. Create an attribute and set it as the *Primary identifier*.
3. Click the *Identifiers* tab and double-click the entry to open its property sheet.
4. Click the *Hibernate* tab, select a generator class and define its parameters.

Example parameters:

- Select hilo in the *Generator class* list
- Enter "table=hi\_value,column=next\_value,max\_lo=10000" in the *Generator params* box. You should use commas to separate the parameters.

5. You can check the code in the *Preview* tab:

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- Hibernate XML Mapping File -->
<!-- Author: xwang -->
<!-- Modified: Tuesday, October 25, 2005 12:36:51 AM -->
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="package1" auto-import="true" default-cas
<class name="Customer" mutable="true" dynamic-update="false" dyna
<id name="id">
  <column name="id" sql-type="int" not-null="true"/>
  <generator class="hilo">
    <param name="table">hi_value</param>
    <param name="column">next_value</param>
    <param name="max_lo">10000</param>
  </generator>
</id>
<property name="name" insert="true" update="true" optimistic-
  <column name="name" sql-type="varchar(254)" length="254"/>
</property>
<property name="email" insert="true" update="true" optimistic-
  <column name="email" sql-type="varchar(254)" length="254"/>
</property>
</class>
</hibernate-mapping>

```

Note that, if there are several primary identifier attributes, the generator is not used.

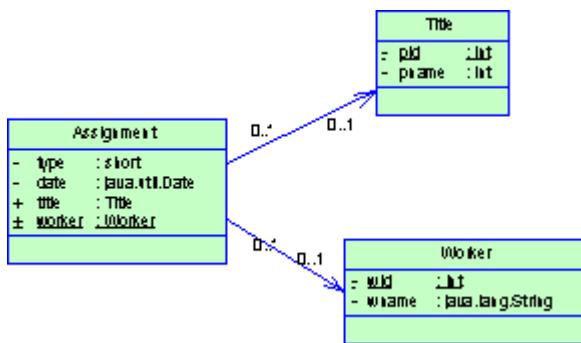
### 2.9.1.3.2.2 Composite Identifier Mapping

If a primary key comprises more than one column, the primary identifier can have multiple attributes mapped to these columns. In some cases, the primary key column could also be the foreign key column.

#### Procedure

1. Define association mappings.
2. Migrate navigable roles of associations.
3. Add these migrated attributes in primary identifier. The migrated attributes need not to be mapped.

#### Results



In the above example, the Assignment class has a primary identifier with three attributes: one basic type attribute and two migrated attributes. The primary identifier mapping is as follows:

```

<composite-id>
    <key-property name="type">
        <column name="type" sql-type="smallint"
            not-null="true"/>
    </key-property>
    <key-many-to-one name="title">
    </key-many-to-one>
    <key-many-to-one name="worker">
    </key-many-to-one>
</composite-id>

```

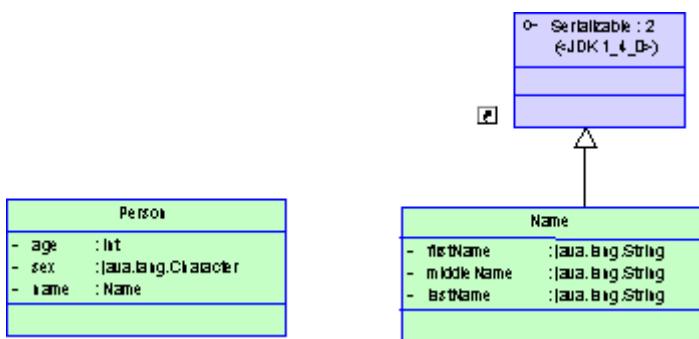
### 2.9.1.3.2.3 Component Primary Identifier Mapping

For more convenience, a composite identifier can be implemented as a separate value type class. The primary identifier has just one attribute with the class type. The separate class should be defined as a value type class. Component class mapping will be generated then.

#### Procedure

1. Define a primary identifier attribute.
2. Define the type of the attribute as a value type class.
3. Set the Class generation property of the primary identifier attribute to Embedded.
4. Set the ValueType of the primary identifier class to true.
5. Define a mapping for the primary identifier class.

#### Results



In the example above, three name attributes are grouped into one separate class Name. It is mapped to the same table as Person class. The generated primary identifier is as follows:

```

<composite-id name="name" class="identifier.Name">
    <key-property name="firstName">
        <column name="name(firstName"
            sql-type="text"/>
    </key-property>
    <key-property name="middleName">
        <column name="name(middleName"
            sql-type="text"/>
    </key-property>
    <key-property name="lastName">
        <column name="name(lastName"
            sql-type="text"/>
    </key-property>
</composite-id>

```

Note: The value type class must implement the `java.io.Serializable` interface, which implements the `equals()` and `hashCode()` methods.

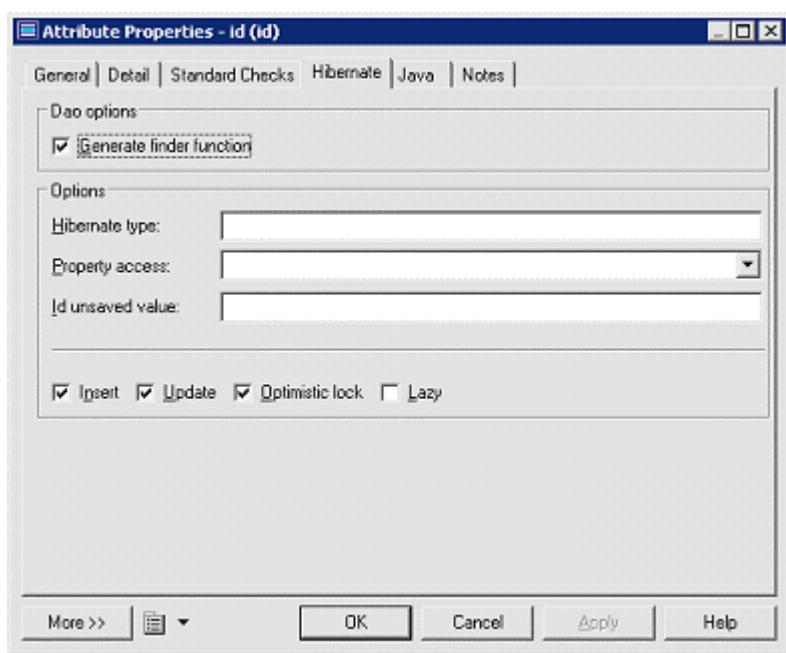
### 2.9.1.3.3 Defining Attribute Mappings

Attributes can be migrated attributes or ordinary attributes. Ordinary attributes can be mapped to columns or formulas. Migrated attributes do not require attribute mapping.

The following types of mapping are possible:

- Map attribute to formula - When mapping an attribute to a formula, you should ensure that the syntax is correct. There is no column in the source table of the attribute mapping.
- Component attribute mapping - A component class can define the attribute mapping as for other classes, except that there is no primary identifier.
- Discriminator mapping - In inheritance mapping with a one-table-per-hierarchy strategy, the discriminator needs to be specified in the root class. You can define the discriminator in the Hibernate tab of the class property sheet.

Hibernate-specific attribute mapping options are defined in the Hibernate tab of the Attribute property sheet.



| Option                   | Description   |
|--------------------------|---|
| Generate finder function | Generates a finder function for the attribute.  |
| Hibernate type           | Specifies a name that indicates the Hibernate type.   |
| Property access          | Specifies the strategy that Hibernate should use for accessing the property value.  |
| Id unsaved value         | Specifies the value of an unsaved id.   |
| Insert                   | Specifies that the mapped columns should be included in any SQL INSERT statements.  |
| Update                   | Specifies that the mapped columns should be included in any SQL UPDATE statements.  |
| Optimistic lock          | Specifies that updates to this property require acquisition of the optimistic lock.   |
| Lazy                     | Specifies that this property should be fetched lazily when the instance variable is first accessed (requires build-time byte code instrumentation). |

## 2.9.1.4 Hibernate Association Mappings

Hibernate supports one-one, one-to-many/many-to-one, and many-to-many association mappings. The mapping modeling is same with standard O/R Mapping Modeling. However, Hibernate provides special options to define its association mappings, which will be saved into <Class>.hbm.xml mapping file. PowerDesigner allows you to define standard association attributes like Container Type implementation class, role navigability, array size and specific extended attributes for Hibernate association mappings.

The following properties are available on the association property sheet *Hibernate Collection* tab:

| Field              | Description  |
|--------------------|--|
| Sort               | Specifies a sorted collection with natural sort order, or a given comparator class.<br>Scripting name: <code>sort</code>   |
| Order by           | Specifies a table column (or columns) that define the iteration order of the Set or bag, together with an optional asc or desc.<br>Scripting name: <code>order-by</code> |
| Access             | Specifies the strategy Hibernate should use for accessing the property value.<br>Scripting name: <code>access</code>   |
| Cascade            | Specifies which operations should be cascaded from the parent object to the associated object.<br>Scripting name: <code>cascade</code>                                   |
| Collection type    | Specifies a name that indicates the Hibernate type.<br>Scripting name: <code>type</code>   |
| Batch size         | Specifies the batch load size.<br>Scripting name: <code>batch-size</code>  |
| Not found          | Specifies how foreign keys that reference missing rows will be handled: ignore will treat a missing row as a null association.<br>Scripting name: <code>not-found</code> |
| Inverse collection | Specifies that the role is the inverse relation of the opposite role.<br>Scripting name: <code>inverse</code>  |

The following properties are available on the association property sheet *Hibernate Persistence* tab:

| Field        | Description   |
|--------------|---|
| Schema       | Specifies the name of the schema.<br>Scripting name: <code>schema</code>  |
| Catalog      | Specifies the name of the catalog.<br>Scripting name: <code>catalog</code>  |
| Where clause | Specifies an arbitrary SQL WHERE condition to be used when retrieving objects of this class.<br>Scripting name: <code>where</code>              |
| Check        | Specifies a SQL expression used to generate a multi-row check constraint for automatic schema generation.<br>Scripting name: <code>check</code> |

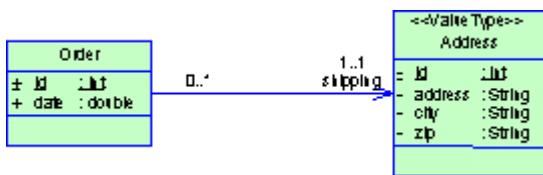
| Field           | Description  |
|-----------------|--|
| Fetch type      | Specifies outer-join or sequential select fetching.<br>Scripting name: <code>fetch</code>  |
| Persister class | Specifies a custom persistence class.<br>Scripting name: <code>persister</code>  |
| Subselect       | Specifies an immutable and read-only entity to a database subselect.<br>Scripting name: <code>subselect</code>                           |
| Index column    | Specifies the column name if users use list or array collection type.<br>Scripting name: <code>index</code>                              |
| Insert          | Specifies that the mapped columns should be included in any SQL INSERT statements.<br>Scripting name: <code>insert</code>                |
| Update          | Specifies that the mapped columns should be included in any SQL UPDATE statements.<br>Scripting name: <code>update</code>                |
| Lazy            | Specifies that this property should be fetched lazily when the instance variable is first accessed.<br>Scripting name: <code>lazy</code> |
| Optimistic lock | Specifies that a version increment should occur when this property is dirty.<br>Scripting name: <code>optimistic-lock</code>             |
| Outer join      | Specifies to use an outer-join.<br>Scripting name: <code>outer-join</code>   |

## 2.9.1.4.1 Mapping Collections of Value Type

If there is a value type class on the navigable role side of an association with a multiplicity of one, PowerDesigner will embed the value type in the entity type as a composite attribute.

### Procedure

1. Create an entity type class and another class for value type. Open the property sheet of the second class, click the *Detail* tab, and select the *Value type* radio button.
2. Create an association between the value type class and an entity type class. On the value type side, set the multiplicity to one and the navigability to true.



- Generate the PDM with O/R mapping, open the property sheet of the entity class and click the *Preview* tab and verify the mapping file.

A composite entity class may contain components, using the <nested-composite-element> declaration.

### 2.9.1.4.2 Defining the Association Collection Type

You can define the association collection type for one-to-many or many-to-many associations.

#### Procedure

- Open the association property sheet, click the *Detail* tab, and specify a *Multiplicity* on both sides.
- Specify either unidirectional or bi-directional navigability, and role names, if necessary.
- If one role of the association is navigable and the multiplicity is many, you can set the collection container type and batch loading size.
- If you select `java.util.List` or `<none>`, it implies that you want to use an array or list-indexed collection type. Then you should define an index column to preserve the objects collection order in the database.

Note: The Java collection container type conditions the Hibernate collection type.

| Collection Container Type         | Hibernate Collection Type                             |
|-----------------------------------|---|
| <code>&lt;None&gt;</code>         | <code>array</code>                                    |
| <code>java.util.Collection</code> | <code>bag</code> or <code>idbag</code> (many-to-many) |
| <code>java.util.List</code>       | <code>list</code>                                     |
| <code>java.util.Set</code>        | <code>set</code>                                      |

### 2.9.1.5 Defining Hibernate Inheritance Mappings

Hibernate supports the three basic inheritance mapping strategies:

- Table per class hierarchy
- Table per subclass

- Table per concrete class
- There are not any special different from standard inheritance mapping definition in O/R Mapping Modeling. However, a separate mapping file is generated for each inheritance hierarchy that has a unique mapping strategy. All mappings of subclasses are defined in the mapping file. The mapping file is generated for the root class of the hierarchy.

## 2.9.1.6 Generating Code for Hibernate

PowerDesigner can generate code for Hibernate. You must have installed Hibernate 3.0 or higher.

### Procedure

1. Select **Tools > General Options**, click the *Variables* node, and add a variable `HIBERNATE_HOME` with your Hibernate home directory path. For example, `D:\Hibernate-3.0`.
2. Select **Tools > Check Model** (or press **F4**) to verify if there are errors or warnings in the model. If there are errors, you must fix them before generating code.
3. Select **Language > Generate Java Code** to open the Generation dialog.
4. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see [Checking an OOM \[page 251\]](#)).
5. [optional] Click the *Selection* tab and specify the objects that you want to generate from. By default, all objects are generated.
6. [optional] Click the *Options* tab and set any appropriate generation options:
  - [optional] To use DAO, set the *Generate DAO sources* option to true.
  - [optional] To use Eclipse to compile and test the Java classes, set the *Generate Eclipse project artifacts* option to true.
  - [optional] To use unit test classes to test the Hibernate persistent objects, set the *Generate unit test sources* option to true.
7. [optional] Click the *Generated Files* tab and specify which files will be generated. By default, all files are generated.

For information about customizing the files that will be generated, see [Customizing and Extending PowerDesigner > Extension Files > Generated Files \(Profile\)](#) .

8. Click **OK** to begin generation.

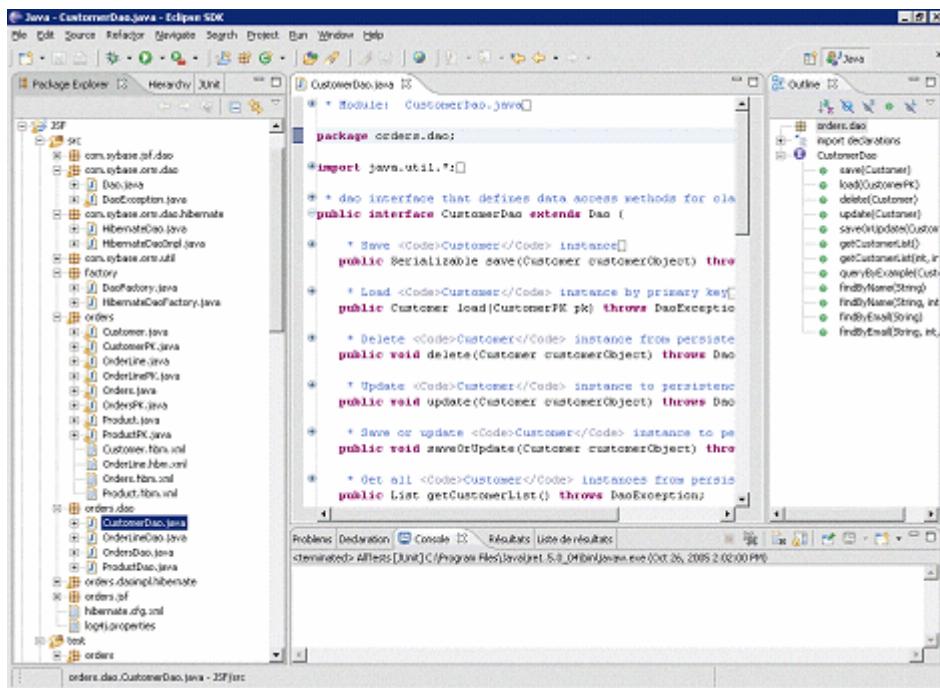
#### **i** Note

If you are working in the PowerDesigner Eclipse plugin then, after the code generation, the project is automatically imported or refreshed in Eclipse.

9. [optional - if you are working in the desktop PowerDesigner and have selected the Generate Eclipse project artifacts generation option] In Eclipse, select **File > Import > Existing Projects into Workspace**.

Eclipse will automatically compile all the Java classes. If there are errors, you should check that all the required Jar files are included in the .classpath file, and that the JDK version is the right one. If you use Java as the language in OOM, you need to use the JDK 5.0 to compile the code.

You can modify the generated code, compile, run the unit test and develop your application.



### 2.9.1.6.1 Running Unit Tests in Eclipse

If the generated Java classes are compiled without error, you can run the unit tests within Eclipse or using Ant. The unit tests generate random data, and create, update, delete or find objects, and return whether the result is as expected or not. Eclipse integrates JUnit.

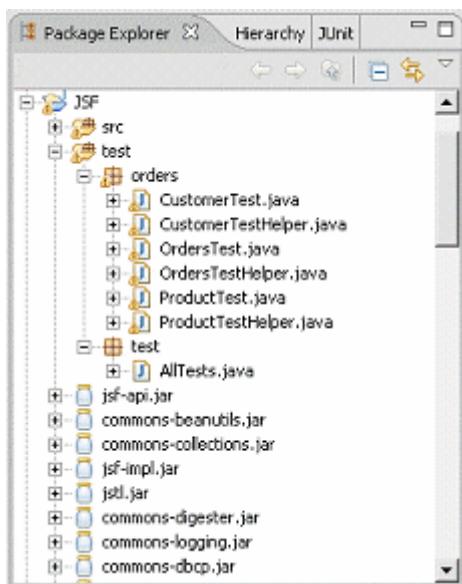
#### Context

##### i Note

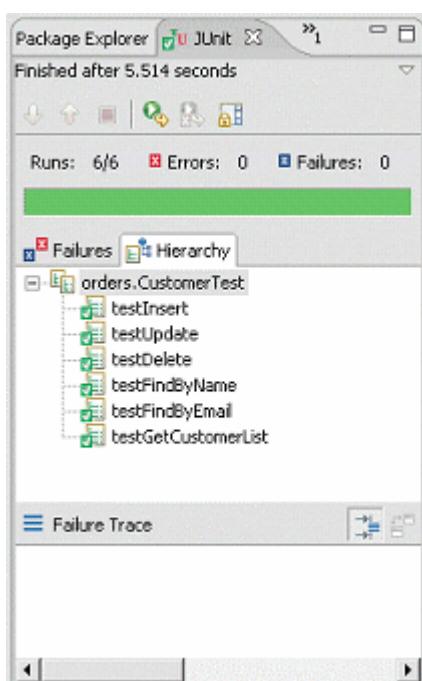
#### Procedure

1. Create a database file, define an ODBC connection, generate the database from the PDM using the ODBC connection, give the test user the permission to connect, and start the database.

- Open the Java perspective, and expand the test package in the Package Navigator



- Right-click a single test case (for example, CustomerTest.java) or the test suite (to run all the tests), and select **Run As > JUnit Test**
- Select the JUnit view to verify the result



If there are 0 errors, then the test has succeeded. If there are errors, you need to check the Console view to locate the sources of them. The problem could be:

- The database is not started or the user name or password is wrong.

- The database is not generated, or the mapping is wrong.

## 2.9.1.6.2 Running Unit Tests with Ant

To generate the Ant build.xml file, you need to select the Generate Ant build.xml file in the Java code generation window.

### Context

To use Ant, you need to download it from <http://www.apache.org>, install it, and:

- Define an environment variable **ANT\_HOME** and set it to your Ant installation directory.
- Copy junit-3.8.1.jar from the **HIBERNATE\_HOME/lib** directory to the **ANT\_HOME/lib** directory.
- Make sure that the Hibernate Jar files and the JDBC driver Jar file of your database are defined in the **build.xml** file or in the **CLASSPATH** environment variable.

### Procedure

1. Select  *Language*  *Generate Java Code*  to open the Generation dialog.
2. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see [Checking an OOM \[page 251\]](#)).
3. Select the *Options* tab and set the **Generate Ant build.xml file** option to true.
4. Select the *Tasks* tab, and select the **Hibernate: Run the generated unit tests** task.
5. Click **OK** to begin generation.

After you close the generation files list window, the JUnit task runs. You can see the result in output window:

```
completetest:
[mkdir] Created dir: C:\Documents and Settings\yayu\My Documents\PD\Code\build\testclasses
[javac] Compiling 8 source files to C:\Documents and Settings\yayu\My Documents\PD\Code\build\testclasses

unit:
[mkdir] Created dir: C:\Documents and Settings\yayu\My Documents\PD\Code\testout
[junit] Running general.SingleClass_InnerClass_InnerInnerClassTest
[junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 4.842 sec
[junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 0.031 sec
[junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 0.061 sec
[junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 0.139 sec

BUILD SUCCESSFUL
Total time: 9 seconds
```

#### Note

You run unit tests with Ant from the command line at any time by navigating to the directory where you have generated the code, and running the JUnit test task **Ant junit**.

## 2.9.2 Generating EJB 3 Persistent Objects

EJB 3.0 introduces a standard O/R mapping specification and moves to POJO based persistence. PowerDesigner provides support for EJB 3 through an extension file.

To enable the EJB 3 extensions in your model, select Model > Extensions, click the *Attach an Extension* tool, select the EJB 3.0 file (on the *O/R Mapping* tab), and click *OK* to attach it.

EJB 3.0 persistence provides a lightweight persistence solution for Java applications. It supports powerful, high performance and transparent object/relational persistence, which can be used both in container and out of container.

EJB 3.0 persistence lets you develop persistent objects using POJO (Plain Old Java Object). All the common Java idioms, including association, inheritance, polymorphism, composition, and the Java collections framework are supported. EJB 3.0 persistence allows you to express queries in its own portable SQL extension (EJBQL), as well as in native SQL.

PowerDesigner supports the design of Java classes, database schema and Object/Relational mapping (O/R mapping). Using these metadata, PowerDesigner can generate codes for EJB 3 persistence, including:

- Persistent EJB Entities (domain specific objects)
- Configuration file
- O/R mapping files (Optional)
- DAO factory
- Data Access Objects (DAO)
- Unit test classes for automated test

### 2.9.2.1 Generating Entities for EJB 3.0

You generate entities for EJB 3.0 as follows:

#### Procedure

1. Create an OOM and a PDM, and then define your O/R mappings. For detailed information, see [Object/Relational \(O/R\) Mapping \[page 430\]](#).
2. Define the EJB 3 persistence settings.
3. Generate Java code.
4. Run unit tests.

## 2.9.2.2 Defining EJB 3 Basic O/R Mapping

There are three kinds of persistent classes in EJB 3:

- Entity classes
- Embeddable classes
- Mapped superclasses

The following requirements apply to persistent classes:

- They must be defined as persistent classes (see [Entity Class Transformation \[page 431\]](#)).
- They must be top level classes (and not inner classes).
- Entity classes and Mapped superclasses should carry the EJBEntity stereotype.
- Embeddable classes are Value type classes, i.e. persistent classes with a Value type persistent type.

Classes that do not meet these requirements will be ignored.

Tip: You can set the stereotype and persistence of all the classes in a model or package (and sub-packages) by right-clicking the model or package and selecting Make Persistent from the contextual menu.

### 2.9.2.2.1 Defining Entity Mappings

Set the stereotype of persistent classes to make them EJB 3 Entity classes.

The Entity annotation is generated to specify that the class is an entity.

```
@Entity  
@Table(name="EMPLOYEE")  
public class Employee { ... }
```

For more informations about defining entity class mappings, see [Object/Relational \(O/R\) Mapping \[page 430\]](#).

### EJB 3 Entity Mapping Options

The following EJB3-specific mapping options can be set on the EJB 3 Persistence tab of the class property sheet.

| Option          | Description  |
|-----------------|--|
| Entity Name     | Specifies that the class alias that can be used in EJB QL. |
| Access strategy | Specifies the default access type (FIELD or PROPERTY)      |
| Schema name     | Specifies the name of the database schema.                 |
| Catalog name    | Specifies the name of the database catalog.                |

| Option                  | Description  |
|-------------------------|--|
| Mapping definition type | Specifies what will be generated for mapping meta data, the mapping file, annotations or both. |
| Discriminator value     | Specifies the discriminator value to distinguish instances of the class                        |

## Mapping to Multiple Tables

In EJB 3, Entity classes can be mapped to multiple tables. For more information on how to map one Entity class to multiple tables, see [Object/Relational \(O/R\) Mapping \[page 430\]](#).

There is a check to guarantee that secondary tables have reference keys referring to primary tables.

The SecondaryTable annotation is generated to specify a secondary table for the annotated Entity class. The SecondaryTables annotation is used when there are multiple secondary tables for an Entity.

## Defining Primary Identifier Mapping

Three kinds of primary identifier mapping are supported in EJB 3.0:

- Simple identifier mapping - This kind of primary key can be generated automatically in EJB 3. You can define the generator class and parameters. There are four generator class types, Identity, Sequence, Table and Auto. Table generator and sequence generators require certain parameters. See the EJB 3.0 persistence specification for details.  
You can define the generator class and parameters in the EJB 3 persistence tab of primary identifiers' property sheet. The parameters take the form of param1=value1; param2=value2.  
The Id annotation generated specifies the primary key property or field of an entity. The GeneratedValue annotation provides for the specification of generation strategies for the values of primary keys:

```
@Id
@GeneratedValue(strategy=GenerationType.TABLE, generator="customer_generator")
@TableGenerator(
    name="customer_generator",
    table="Generator_Table",
    pkColumnName="id",
    valueColumnName="curr_value",
    initialValue=4
)
@Column(name="cid", nullable=false)
```

- Composite identifier mapping - The IdClass annotation will be generated for an entity class or a mapped superclass to specify a composite primary key class that is mapped to multiple fields or properties of the entity:

```
@IdClass(com.acme.EmployeePK.class)
@Entity
public class Employee {
    @Id String empName;
    @Id Date birthDay;
```

```
...  
}
```

- Embedded primary identifier mapping - corresponds to component primary identifier mapping. The EmbeddedId annotation is generated for a persistent field or property of an entity class or mapped superclass to denote a composite primary key that is an embeddable class:

```
@EmbeddedId  
protected EmployeePK empPK;
```

## Defining Attribute Mappings

Each persistent attribute with basic types can be mapped to one column. Follow instructions to define attribute mappings for this kind of persistent attributes.

The following EJB3-specific attribute mapping options are available on the EJB 3 Persistence tab of each attribute's property sheet:

| Option            | Description  |
|-------------------|--|
| Version attribute | Specifies if attribute is mapped as version attribute                              |
| Insertable        | Specifies that the mapped columns should be included in any SQL INSERT statements. |
| Updatable         | Specifies that the mapped columns should be included in any SQL UPDATE statements. |
| Fetch             | Specify if attribute should be fetched lazily.                                     |
| Generate finder   | Generates a finder function for the attribute.                                     |

The Basic annotation is generated to specify fetch mode for the attribute or property and whether the attribute or property is mandatory. The Column annotation is generated to specify a mapped column for a persistent property or field.

```
@Basic  
@Column(name="DESC", nullable=false, length=512)  
public String getDescription() { return description; }
```

Other Annotations can also be generated to specify the persistence type of an attribute or property. A Temporal annotation specifies that a persistent property or attribute should be persisted as a temporal type. There is also the enumerated annotation for enumerated types and Lob for large object types.

## Defining Versioning Mapping

EJB 3.0 uses managed versioning to perform optimistic locking. If you want to use this kind of feature, you need to set one mapped persistent attribute as the Version attribute, by selecting the Version attribute option on the EJB 3 Persistence tab. The following types are supported for Version attribute: int, Integer, short, Short, long, Long, Timestamp.

The Version attribute should be mapped to the primary table for the entity class. Applications that map the Version property to a table other than the primary table will not be portable. Only one Version attribute should be defined for each Entity class.

The Version annotation is generated to specify the version attribute or property of an entity class that serves as its optimistic lock value.

```
@Version  
@Column(name="OPTLOCK")  
protected int getVersionNum() { return versionNum; }
```

## 2.9.2.2.2 Defining Embeddable Class Mapping

Embeddable classes are simple Value type classes. Follow the instructions for defining Value type class mappings to define Embeddable class mapping for EJB 3.

In EJB 3, Embeddable classes can contain only attribute mappings, and these persistent attributes can have only basic types, i.e. Embeddable classes cannot contain nested Embeddable classes.

Note: The Embeddable class must implement the java.io.Serializable interface and overrides the equals() and hashCode() methods.

The Embeddable annotation is generated to specify a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity.

```
@Embeddable  
public class Address implements java.io.Serializable {  
    @Basic(optional=true)  
    @Column(name="address_country")  
    public String getCountry() {}  
    ....  
}
```

## 2.9.2.3 Defining EJB 3 Association Mappings

EJB 3 persistence provides support for most of the association mapping strategies. We will just address the differences here.

One association must be defined between two Entity classes or one Entity class and one Mapped superclass before it can be mapped. Association mapping with a Mapped superclass as the target will be ignored.

Embeddable classes can be either the source or the target of associations.

Mapping for associations with association class is not currently supported. You must separate each kind of associations into two equivalent associations.

For more informations about mapping, see [Object/Relational \(O/R\) Mapping \[page 430\]](#).

### 2.9.2.3.1 Mapping One-to-one Associations

EJB 3 persistence supports both bi-directional one-to-one association mapping and unidirectional one-to-one association mapping.

The `ToOneOne` annotation is generated to define a single-valued association to another entity that has one-to-one multiplicity. For bi-directional one-to-one associations, the generated annotations will resemble:

```
@OneToOne(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumns({
    @JoinColumn(name="aid", referencedColumnName="aid")
})
public Account getAccount() { ... }
@OneToOne(cascade=CascadeType.PERSIST, mappedBy="account")
public Person getPerson() { ... }
```

Generated annotations for unidirectional one-to-one associations are similar. A model check is available to verify that mappings are correctly defined for unidirectional one-to-one associations. One unidirectional association can only be mapped to a reference that has the same direction as the association.

For more informations about mapping, see [Object/Relational \(O/R\) Mapping \[page 430\]](#).

### 2.9.2.3.2 Mapping One-to-many Associations

EJB 3 persistence supports bi-directional one-to-many association mapping, unidirectional one-to-one association mapping and unidirectional one-to-many association mapping.

A `ToMany` annotation is generated to define a many-valued association with one-to-many multiplicity. A `ManyToOne` annotation is generated to define a single-valued association to another entity class that has many-to-one multiplicity. The `JoinColumn` annotation is generated to specify a join column for the reference associating the tables. For bi-directional one-to-many associations, generated annotations will resemble:

```
@OneToMany(fetch=FetchType.EAGER, mappedBy="customer")
public java.util.Collection<Order> getOrder() { ... }

@ManyToOne
@JoinColumns({
    @JoinColumn(name="cid", referencedColumnName="cid")
})
public Customer getCustomer() { ... }
```

Generated annotations for unidirectional many-to-one associations are similar. A model check is available to verify that mappings for bi-directional one-to-many associations and unidirectional many-to-one associations are correctly defined. The references can only navigate from primary tables of classes on the multiple-valued side to primary tables of classes on the single-valued side.

For unidirectional one-to-many association, the `JoinTable` annotation is generated to define middle table and join columns for the two reference keys.

```
@OneToMany(fetch=FetchType.EAGER)
@JoinTable(
    name="Customer_Order",
    joinColumns={
        @JoinColumn(name="cid", referencedColumnName="cid")
```

```

},
inverseJoinColumns={
    @JoinColumn(name="oid",
        referencedColumnName="orderId")
}
public java.util.Collection<Order> getOrder() { ... }

```

A model check is available to verify that mappings for unidirectional one-to-many associations are correctly defined. Middle tables are needed for this kind of one-to-many association mapping.

One-to-many associations where the primary key is migrated are not supported in EJB 3.

For more informations about mapping, see [Object/Relational \(O/R\) Mapping \[page 430\]](#).

### 2.9.2.3.3 Mapping Many-to-many Associations

EJB 3 persistence supports both bi-directional many-to-many association mapping and unidirectional many-to-many association mapping.

A ManyToMany annotation is generated to define a many-valued association with many-to-many multiplicity.

```

@ManyToMany(fetch=FetchType.EAGER)
@JoinTable(
    name="Assignment",
    joinColumns={
        @JoinColumn(name="eid", referencedColumnName="eid")
    },
    inverseJoinColumns={
        @JoinColumn(name="tid", referencedColumnName="tid")
    }
)
public java.util.Collection<Title> getTitle() { ... }

```

A model check is available to verify that mappings are correctly defined for many-to-many associations. Middle tables are needed for many-to-many association mapping.

For more informations about mapping, see [Object/Relational \(O/R\) Mapping \[page 430\]](#).

### 2.9.2.3.4 Defining EJB 3 Association Mapping Options

The following EJB 3-specific options for association mappings are available on the EJB 3 Persistence tab of an association's property sheet:

| Field          | Description  |
|----------------|--|
| Inverse side   | Specifies which side is the inverse side.                          |
| Role A cascade | Specifies which cascade operation can be performed on role A side. |
| Role B cascade | Specifies which cascade operation can be performed on role B side. |

| Field           | Description   |
|-----------------|---|
| Role A fetch    | Specifies if role A side should be fetched eagerly. |
| Role B fetch    | Specifies if role B side should be fetched eagerly. |
| Role A order by | Specifies the order clause for role A side.         |
| Role B order by | Specifies the order clause for role B side.         |

## 2.9.2.4 Defining EJB 3 Inheritance Mappings

EJB 3 persistence supports all three popular inheritance mapping strategies and also mixed strategies.

You can use the following strategies:

- Table per class hierarchy (SINGLE\_TABLE) - The whole class hierarchy is mapped to one table. You can optionally define discriminator values for each Entity class in the hierarchy on the *EJB 3 Persistence* tab of the class property sheet to specify the value that distinguishes individual this class from other classes. The Inheritance annotation with SINGLE\_TABLE strategy is generated. The DiscriminatorColumn annotation is generated to define the discriminator column. The DiscriminatorValue annotation is generated to specify the value of the discriminator column for entities of the given type if you specify it for the class.

```
@Entity(name="Shape")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="shapeType",
discriminatorType=DiscriminatorType.STRING, length=100)
@Table(name="Shape")
public class Shape { ... }
@Entity(name="Rectangle")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorValue("Rectangle")
@Table(name="Shape")
public class Rectangle extends Shape { ... }
```

A model check is available to verify that discriminator columns are correctly defined.

- Joined subclass (JOINED) - Each class is mapped to its own primary table. Primary tables of child classes have reference keys referring to the primary tables of the parent classes. An Inheritance annotation with JOINED strategy is generated. The PrimaryKeyJoinColumn annotation is generated to define a join column that joins the primary table of an Entity subclass to the primary table of its superclass.

```
@Entity(name="Shape")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="shapeType")
@Table(name="Shape")
public class Shape { ... }
@Entity(name="Rectangle")
@Inheritance(strategy=InheritanceType.JOINED)
@PrimaryKeyJoinColumn({
    @PrimaryKeyJoinColumn(name="sid", referencedColumnName="sid")
})
@Table(name="Rectangle")
public class Rectangle extends Shape { ... }
```

A model check is available to verify that primary tables of child classes have reference keys referring to the primary tables of their parent classes.

- Table per concrete class (TABLE\_PER\_CLASS) - each class is mapped to a separate table. When transforming an OOM to a PDM, PowerDesigner only generates tables for leaf classes, and assumes that all other classes are not mapped to a table, even if you manually define additional mappings. The MappedSuperclass annotations are generated for those classes, and the Inheritance annotation will not be generated for all the classes. You need to customize the generated annotations and create additional tables if you want to map classes other than leaf classes to tables.

```
@MappedSuperclass  
public class Shape { .. }  
@Entity(name="Rectangle")  
@Table(name="Rectangle")  
public class Rectangle extends Shape { ... }
```

### i Note

PowerDesigner does not currently support defining mapped superclasses.

All classes in the class hierarchy should be either Entity classes or Mapped superclasses. For each class hierarchy, the primary identifier must be defined on the Entity class that is the root of the hierarchy or on a mapped superclass of the hierarchy.

You can optionally define a Version attribute on the entity that is the root of the entity hierarchy or on a Mapped superclass of the entity hierarchy.

For more informations about mapping, see [Object/Relational \(O/R\) Mapping \[page 430\]](#).

## 2.9.2.5 Defining EJB 3 Persistence Default Options

The following default persistent options can be set at the model, package or class level:

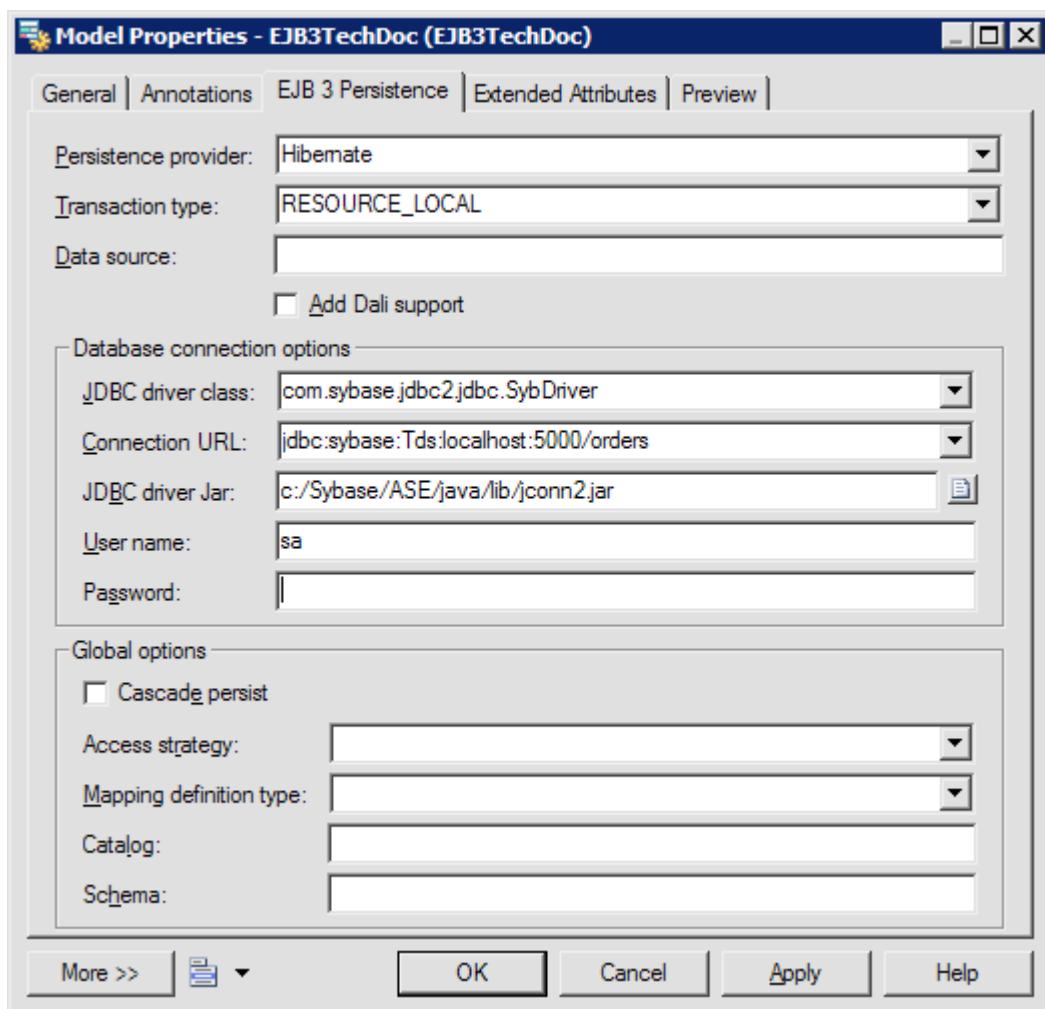
| Option                  | Description  |
|-------------------------|--|
| Default access          | Specifies an access strategy.                            |
| Mapping definition type | Specifies the level of mapping metadata to be generated. |
| Catalog name            | Specifies the catalog name for persistent classes.       |
| Schema name             | Specifies the schema name for persistent classes.        |

## 2.9.2.6 Defining EJB 3 Persistence Configuration

There are some persistence properties which are used for database connection. You need to set them before run the generated application.

### Procedure

1. Open the *EJB 3 Persistence* tab of the model's property sheet.



2. Select persistence provider you use. You should refer to compliance issues for some constraints with these persistence providers.
3. Define JDBC driver class, connection URL, JDBC driver jar file path, user name and password.

## Results

| Option               | Description   |
|----------------------|---|
| Persistence provider | Specifies the persistence provider to be used.  |
| Transaction type     | Specifies the transaction type to be used.  |
| Data source          | Specifies the data source name (if data source is used).  |
| Add Dali support     | Specifies that the generated project can be authored in Dali. A special Eclipse project builder and nature will be generated. |
| JDBC driver class    | Specifies the JDBC driver class.  |
| Connection URL       | Specifies the JDBC connection URL string.   |
| JDBC driver jar      | Specifies the JDBC driver jar file path.  |
| User name            | Specifies the database user name.   |
| Password             | Specifies the database user password.   |
| Cascade persist      | Specifies whether to set the cascade style to PERSIST for all relationships in the persistent unit.                           |

You can verify the configuration parameters in the *Preview* tab. The generated persistence configuration file looks like:

```
<persistence xmlns="http://java.oracle.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.oracle.com/xml/ns/persistence
    http://java.oracle.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="EJB3_0Model" transaction-type="RESOURCE_LOCAL">
    <description>
      This is auto generated configuration for persistent unit EJB3_0Model
    </description>
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <!-- mapped files -->
    <!--jar-file-->
    <!-- mapped classes -->
    <class>com.company.orders.Customer</class>
    <class>com.company.orders.Order</class>
    <properties>
      <property name="hibernate.dialect">org.hibernate.dialect.SybaseDialect</property>
      <property
        name="hibernate.connection.driver_class">com.sybase.jdbc2.jdbc.SybDriver</property>
        <property name="hibernate.connection.url">jdbc:sybase:Tds:localhost:5000/
      Production</property>
      <property name="hibernate.connection.username">sa</property>
      <property name="hibernate.connection.password"></property>
    </properties>
  </persistence-unit>
</persistence>
```

## 2.9.2.7 Generating Code for EJB 3 Persistence

PowerDesigner can generate code for EJB 3 persistence. You must have installed an EJB 3 persistence provider such as Hibernate Entity Manager, Kodo, TopLink and GlassFish.

### Context

#### i Note

To support editing in Eclipse using the Dali JPA Tools, which simplify mapping definition and editing through automated mapping wizards, intelligent mapping assistance, and dynamic problem identification, open the model property sheet, and select the [Add Dali support](#) property.

### Procedure

1. Select **Tools** **General Options**, click the **Variables** node, and add a variable **EJB3PERSISTENCE\_LIB** with your persistence provider libraries home directory path. For example, `D:\EJB 3.0\Hibernate Entity Manager\lib`.
2. Select **Tools** (or press F4) to verify if there are errors or warnings in the model. If there are errors, you must fix them before generating code.
3. Select **Language** to open the Generation dialog.  
On the **Targets** tab, ensure that the O/R Mapping target is selected.
4. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see [Checking an OOM \[page 251\]](#)).
5. [optional] Click the **Selection** tab and specify the objects that you want to generate from. By default, all objects are generated.
6. [optional] Click the **Options** tab and set any appropriate generation options:

| Option                             | Description  |
|------------------------------------|--|
| Generate DAO sources               | Specifies whether DAO sources should be generated.                             |
| Generate Eclipse project artifacts | Specifies whether Eclipse project file and classpath file should be generated. |
| Generate unit test sources         | Specifies whether unit test sources should be generated.                       |
| Java source directory              | Specifies directory for Java sources.  |
| Test source directory              | Specifies directory for unit test sources.                                     |
| Generate schema validation files   | Specifies whether schema file and validation script should be generated.       |

| Option                      | Description   |
|-----------------------------|---|
| Generate Ant build.xml file | Specifies whether Ant build.xml file should be generated. |

7. [optional] Click the *Generated Files* tab and specify which files will be generated. By default, all files are generated.

For information about customizing the files that will be generated, see *Customizing and Extending PowerDesigner > Extension Files > Generated Files (Profile)*.

8. [optional] Click the *Tasks* tab and specify any appropriate generation tasks to perform:

Run generated unit tests - PowerDesigner will run unit tests by Ant script after generation. You can also run unit tests later in Eclipse (see [Running Unit Tests in Eclipse \[page 479\]](#)) or [Ant Running Unit Tests with Ant \[page 481\]](#)).

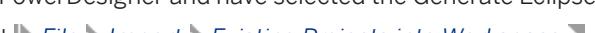
9. Click *OK* to begin generation.

The following files are generated:

- Eclipse Project Files - If you have selected the Generate Eclipse project artifacts generation option, .project file and .classpath file are generated by PowerDesigner. But if you are regenerating codes, these two files will not be generated again.
- Persistent Java Classes - If mapping definition type specified includes Annotation, Default, Annotation or Mapping File & Annotation, annotations will be generated in Java sources.
- Primary Key Classes - To ease find-by-primary-key operation, or mandatory for composite primary key.
- EJB 3 Configuration File (persistence.xml) - In the META-INFO sub directory of Java source directory.
- Log4J Configuration File (log4j.properties) - In the Java source directory.
- Utility Class (Util.java) - Contains utility functions used by unit tests, such as compare date by precision.
- EJB 3 O/R Mapping Files - If mapping definition type specified includes Mapping File & Annotation or Mapping file, EJB 3 O/R mapping files will be generated. These mapping files are generated in the same directory with Java source.
- Factory and Data Access Objects - To help simplify the development of your application, PowerDesigner generates DAO Factory and Data Access Objects (DAO), using Factory and DAO design pattern.
- Unit Test Classes and test helper classes.
- AllTest Class - a test suite that runs all the unit test cases.
- Ant Build File - to help you to compile and run unit tests

#### i Note

If you are working in the PowerDesigner Eclipse plugin then, after the code generation, the project is automatically imported or refreshed in Eclipse.

10. [optional - if you are working in the desktop PowerDesigner and have selected the Generate Eclipse project artifacts generation option] In Eclipse, select .

Eclipse will automatically compile all the Java classes. If there are errors, you should check that all the required Jar files are included in the .classpath file, and that the JDK version is the right one. If you use Java as the language in OOM, you need to use the JDK 5.0 to compile the code.

## 2.9.3 Generating JavaServer Faces (JSF) for Hibernate

JavaServer Faces (JSF) is a UI framework for Java Web applications. It is designed to significantly ease the burden of writing and maintaining applications that run on a Java application server and render their UIs back to a target client.

PowerDesigner supports JSF through an extension file that provides JSF extended attributes, model checks, JSP templates, invoker-managed bean templates, and face-configure templates for your Java OOM.

You can quickly build Web applications without writing repetitive code, by using PowerDesigner to automatically generate persistent classes, DAO, managed beans, page navigation and JSF pages according to your Hibernate or EJB3.0 persistent framework.

To enable the JSF extensions in your model, select Model > Extensions, click the *Attach an Extension* tool, select the JavaServer Faces (JSF) file on the *User Interface* tab), and click *OK* to attach it.

### Note

Since JSF uses Data Access Objects (DAO) to access data from the database, you will need also to add a persistence management extension such as Hibernate in order to generate JSF.

JSF generation can help you to test persistent objects using Web pages with your own data and can also help you to generate default JSF Web application. You can use an IDE to improve the generated JSF pages or change the layout.

### 2.9.3.1 Defining Global Options

Each page could use a style sheet, a header file and a footer file to define its standard presentation.

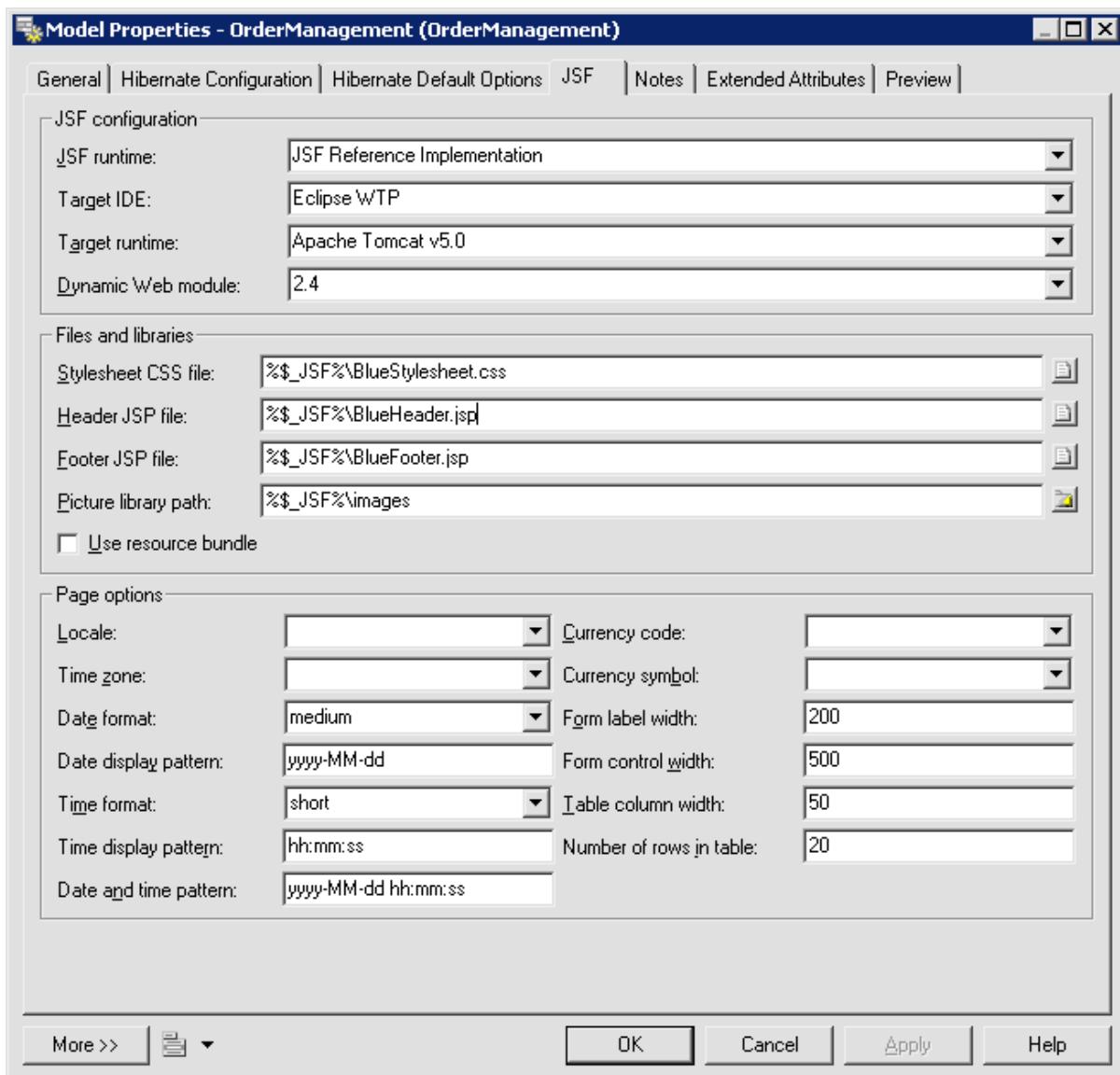
#### Context

PowerDesigner provides default style sheet, header and footer files. Alternatively, you can specify your own files.

You can also define global default options like data format, time format, etc.

#### Procedure

1. Open the model property sheet, and click the *JSF* tab:



2. Define style sheet, header and footer files.
3. Define the directory where the images used by style sheet, header and footer.
4. Define the JSF library Jar files directory.
5. Define default options.

## Results

The following options are available:

| Option               | Description  |
|----------------------|--|
| JSF runtime          | Specifies the JSF runtime.<br>It can be JSF Reference Implementation or Apache My Faces.             |
| Target IDE           | Specifies the target IDE.<br>List/Default Values: Eclipse WTP, Sybase WorkSpace                      |
| Target runtime       | Specifies the target runtime.<br>List/Default Values: Apache Tomcat v5.0, Sybase EAServer v5.x, etc. |
| Dynamic Web Module   | Specifies the dynamic web module's version<br>List/Default Values: 2.2, 2.3, and 2.4.                |
| Stylesheet CSS File  | Specifies the stylesheet file for JSF pages.<br>List/Default Values: %\$_JSF%\stylesheet.css         |
| Header JSP file      | Specifies the header file for JSF pages.<br>List/Default Values: %\$_JSF%\header.jsp                 |
| Footer JSP file      | Specifies the footer file for JSF pages.<br>List/Default Values: %\$_JSF%\footer.jsp                 |
| Picture library path | Specifies the path that contains pictures for JSF pages.<br>List/Default Values: %\$_JSF%\images     |
| Use resource bundle  | Specifies to use a resource bundle.  |
| Locale               | Specifies the locale   |
| Time zone            | Specifies the time zone  |
| Date format          | Specifies the date format could be default, short, medium, long, full<br>Default: short              |
| Date display pattern | Specifies the date pattern   |
| Time format          | Specifies the date format could be default, short, medium, long, full<br>Default: short              |
| Time display pattern | Specifies the time pattern   |

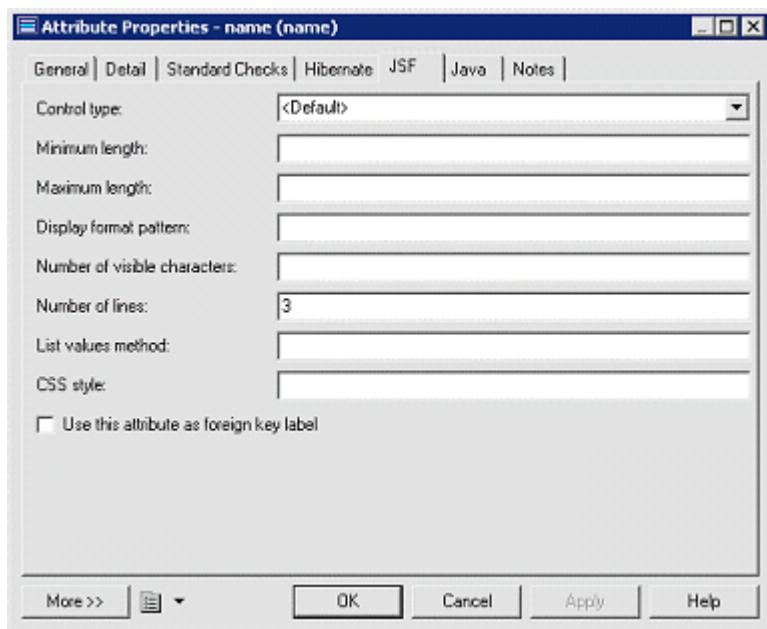
| Option                | Description  |
|-----------------------|--|
| Date and time pattern | Specifies the date and time pattern  |
| Currency code         | Specifies the currency code  |
| Currency symbol       | Specifies the currency symbol  |
| Form label width      | Specifies the width of the control label in pixel in a form<br>Default: 200  |
| Form control width    | Specifies the width of the control in pixel in a form<br>Default: 500        |
| Table column width    | Specifies the width of the control in pixel in a table<br>Default: 50        |
| Table number rows     | Specifies the number of rows that can be displayed in a table<br>Default: 20 |

### 2.9.3.2 Defining Attribute Options

You can define attribute-level options for validation or presentation style.

#### Procedure

1. Open the attribute property sheet, and click the *JSF* tab.
2. Define the attribute options.



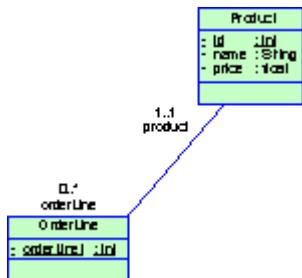
## Results

| Option                       | Description  |
|------------------------------|--|
| Control type                 | <p>Specifies the type of control.</p> <p>Note: You should select the type that can support the Attribute Java type.</p> <ul style="list-style-type: none"> <li>• String - EditBox, MultilineEdit</li> <li>• Boolean - CheckBox</li> <li>• Date - Date, Time, DateTime, Year, Month, Day</li> <li>• &lt;Contains List of Value&gt; - ListBox, ComboBox, RadioButtons</li> </ul> |
| Minimum length               | Specifies the minimum number of characters   |
| Maximum length               | Specifies the maximum number of characters   |
| Display format pattern       | Specifies the display format pattern for the attribute   |
| Number of visible characters | Specifies the number of visible characters per line  |
| Number of lines              | <p>Specifies the number of lines for multiline edit control</p> <p>Default: 3</p>  |
| List values method           | Specifies the method that provides the list of values for ListBox, ComboBox or radioButtons.   |
| CSS style                    | Specifies the CSS formatting style   |

| Option                                 | Description   |
|--|---|
| Use the attribute as foreign key label | <p>Specifies that the column associated to the attribute will be used as the foreign key label for the foreign key selection.</p> <p>If no FK label column is defined, PowerDesigner will choose the first not-PK and non FK column for the default label column.</p> <p>Default: False</p> |

Note: If the *Use the attribute as foreign key label* checkbox is not selected and if there is a foreign key in the current table, PowerDesigner generates a combo box by default to display the foreign key id. If you want to display the value of another column (for example, the product name instead of the product id), you can select the *Use the attribute as foreign key label* option for product name attribute to indicate that it will be used as foreign key label.

Remember that if some attributes specify the choice to be true, we will generate the foreign key label only according to the first attribute of them.



### Create New OrderLine

OrderLineId:

Quantity:

Orders:

Product:

## 2.9.3.2.1 Derived Attributes

To support derived attributes in PowerDesigner, you can:

1. Define an attribute, and indicate that it is not persistent, and is derived.
2. Generate and implement a getter.

When generating pages, PowerDesigner will automatically include the derived attributes.

## 2.9.3.2.2 Attribute Validation Rules and Default Values

PowerDesigner can generate validation and default values for the edit boxes in the *Create* and *Edit* pages.

### Procedure

1. Open the attribute property sheet, and click the *Standard Checks* tab.
2. You can define minimum value and maximum values to control the value range.
3. You can define a default value. A string, must be enclosed in quotes. You can also define the Initial value in the *Details* tab.
4. You can define a list of values that will be used in a listbox, combo box or radio buttons.

### Results

Note: You can set a "list of values" on the *Standard Checks* tab of an attribute property sheet, and PowerDesigner will generate a combo box that includes the values. For validation rules, you can define the customized domain as well, and then select the domain you want to apply in the specified attribute.

You can also select which control style to use:

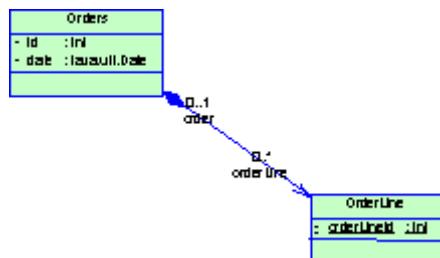
- Combobox
- Listbox
- Radio buttons (if the number of values is low). For example: Mr. Ms.

## 2.9.3.3 Defining Master-Detail Pages

If two objects have a master-detail relationship, PowerDesigner renders them (create, update, delete and find methods) in the same page. When you click the detail link button column in the master table view, the detail page view in the same page will change dynamically.

### Context

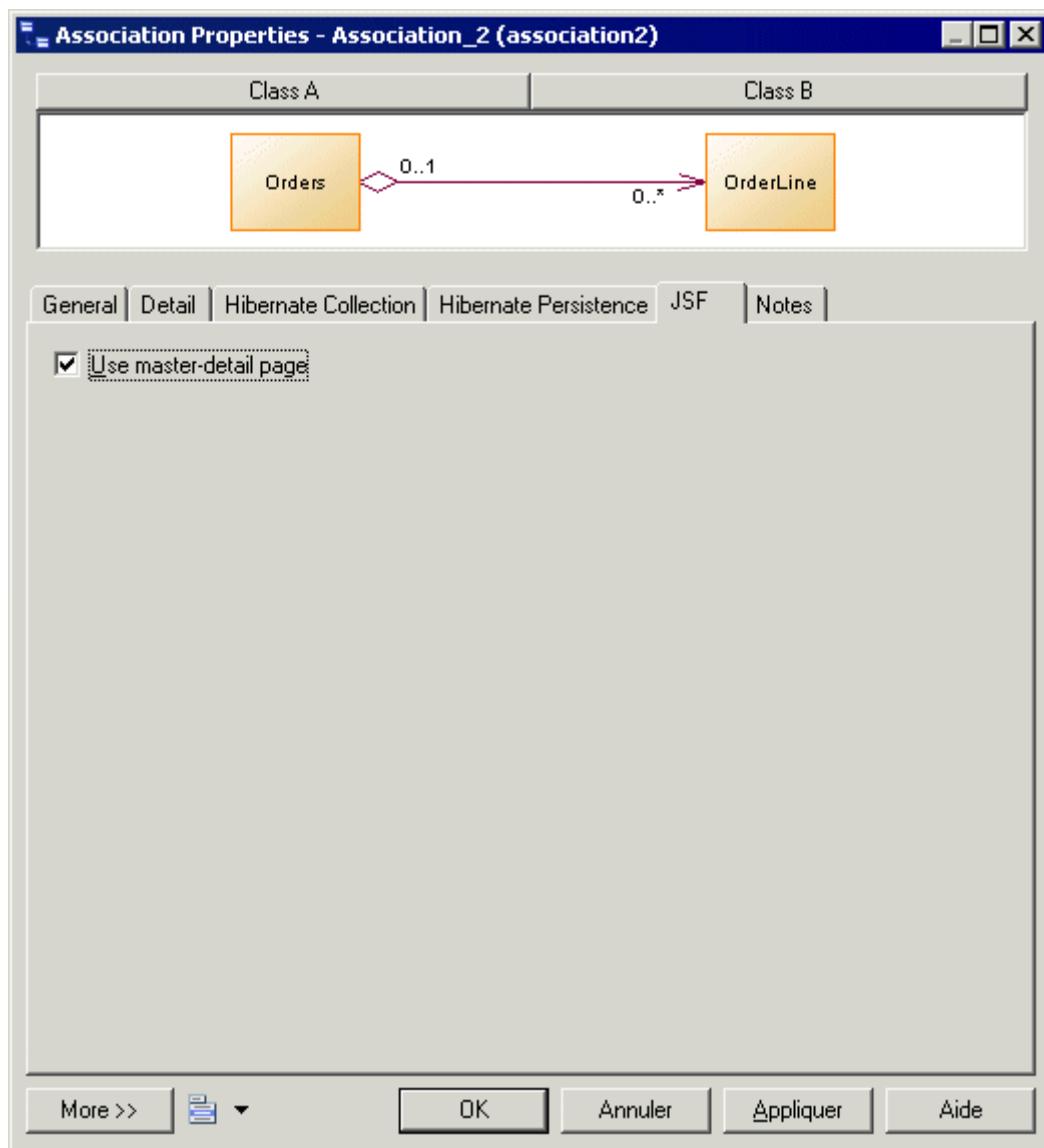
For example, there is a table Orders (Master table) and a table Orderline (Detail table). The association is a composition. If you delete an order, the order lines should be deleted. They will be shown on the same page:





## Procedure

1. Create a one-to-many association, where the one-to-many direction is navigable.
2. Open the association property sheet, and click the *JSF* tab.
3. Select the *Use master-detail page* checkbox:



The association type must be set to Composition or Aggregation, which means that one side of association is a weak-reference to the master class.

The generated master-detail JSF page will resemble the following:

**Orders List**

| <b>Id</b> | <b>Date</b> | <b>Total</b> | <b>Customer</b>          | <b>OrderLines</b>          | <b>Edit</b>          | <b>Delete</b>          |
|-----------|-------------|--------------|--------------------------|----------------------------|----------------------|------------------------|
| 16        | Jan 1, 2005 | 3000.0       | <a href="#">Customer</a> | <a href="#">OrderLines</a> | <a href="#">Edit</a> | <a href="#">Delete</a> |

[Create](#) [Cancel](#)

**OrderLine List**

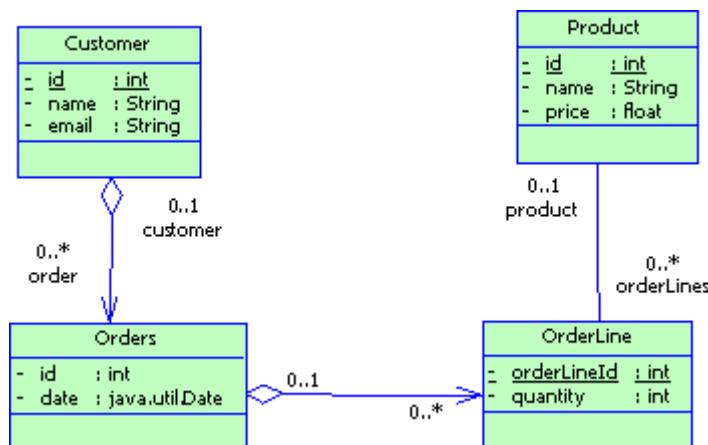
| <b>OrderLineId</b> | <b>Quantity</b> | <b>Order</b>          | <b>Product</b>          | <b>Edit</b>          | <b>Delete</b>          |
|--------------------|-----------------|-----------------------|-------------------------|----------------------|------------------------|
| 1                  | 10              | <a href="#">Order</a> | <a href="#">Product</a> | <a href="#">Edit</a> | <a href="#">Delete</a> |
| 2                  | 5               | <a href="#">Order</a> | <a href="#">Product</a> | <a href="#">Edit</a> | <a href="#">Delete</a> |
| 4                  | 1               | <a href="#">Order</a> | <a href="#">Product</a> | <a href="#">Edit</a> | <a href="#">Delete</a> |
| 5                  | 2               | <a href="#">Order</a> | <a href="#">Product</a> | <a href="#">Edit</a> | <a href="#">Delete</a> |

[Create](#) [Cancel](#)

### 2.9.3.4 Generating PageFlow Diagrams

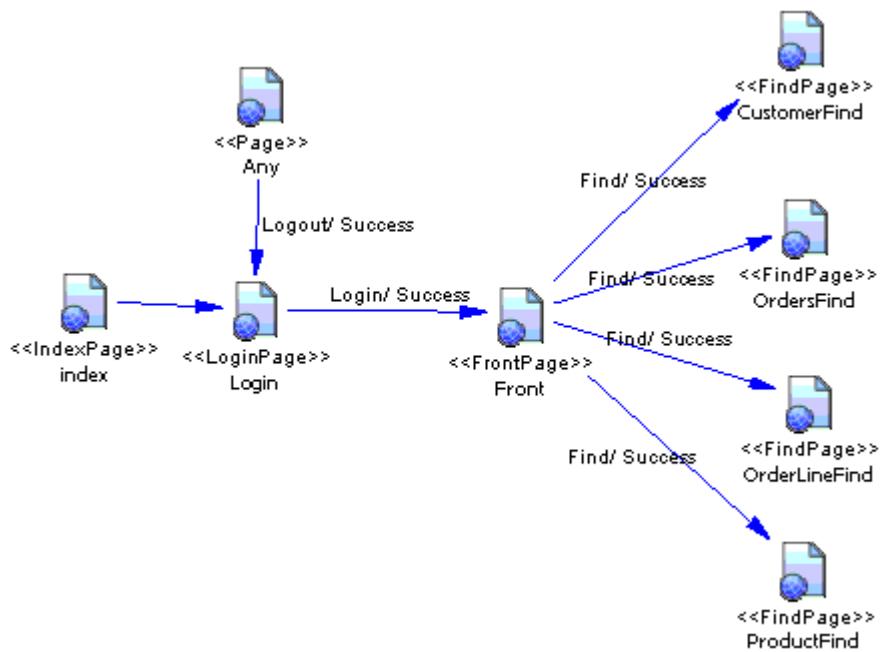
In Java Server Faces, a configuration (xml) file is used to define navigation rules between different web pages, which is called PageFlow. Power Designer will provide a high level PageFlow diagram to abstract different kinds of definition, and can generate navigation rules for JSF web application and JSF page bean based on PageFlow diagram.

You can generate PageFlow diagrams in three levels, Model, Package and Class. For example, working from the following class diagram:

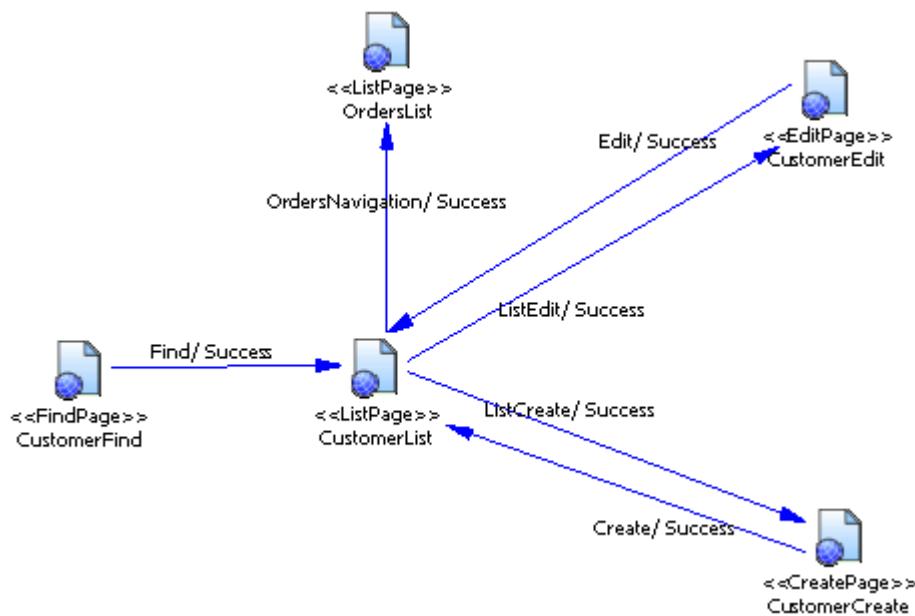


To generate a model-level or package-level PageFlow diagram, right-click the model or a package in the Browser, and select [Generate PageFlow Diagram](#). In addition, a PageFlow will be generated for each class under this package

and its sub packages recursively, e.g., CustomerPageFlow, OrdersPageFlow, ProductPageFlow, OrderLinePageFlow:



To generate a class-level PageFlow diagram, right-click a class in the Browser or a diagram, and select *Generate PageFlow Diagram*:



## 2.9.3.4.1 Modifying Default High Level PageFlow Diagram

After generating the default High Level PageFlow, you can define customized pages and pageflows in the class level PageFlow.

All the pages in the default High Level PageFlow diagram have their pre-defined stereotype, e.g., The stereotype for CustomerFind is "FindPage", CustomerEdit is "EditPage", etc. You can add your customized Page.

You can also add new PageFlows to link the pages in the PageFlow diagram, which is similar to adding a transition in a statechart diagram.

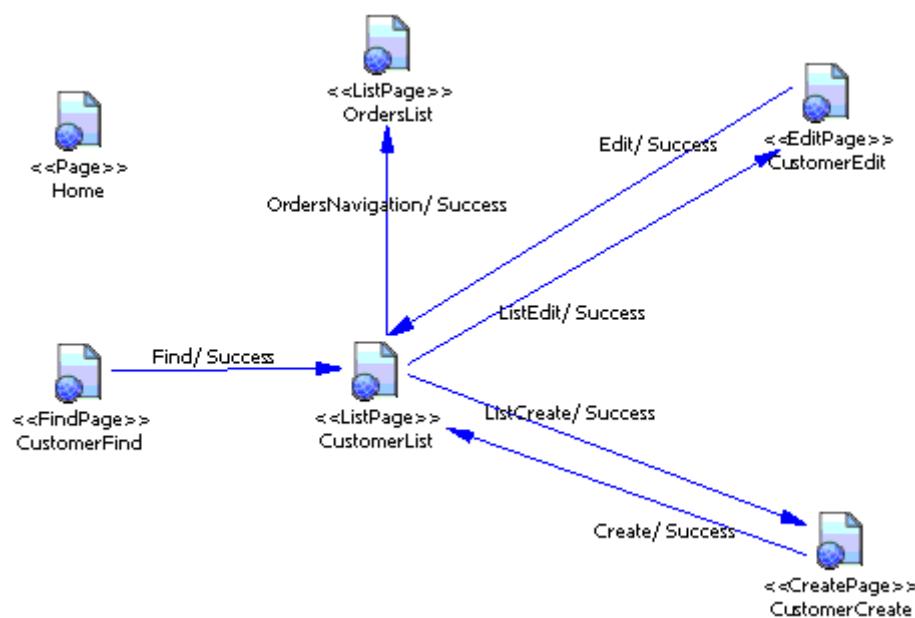
### 2.9.3.4.1.1 Adding a New Page

You add a new page from the Toolbox.

#### Procedure

1. Select the *State* tool in the Toolbox, and drag it to the PageFlow diagram, it will create a new Page with the default name.
2. You can change its name and change its stereotype to Page in its property dialog, e.g., change the name to "Home". This dialog is same with general State.

#### Results



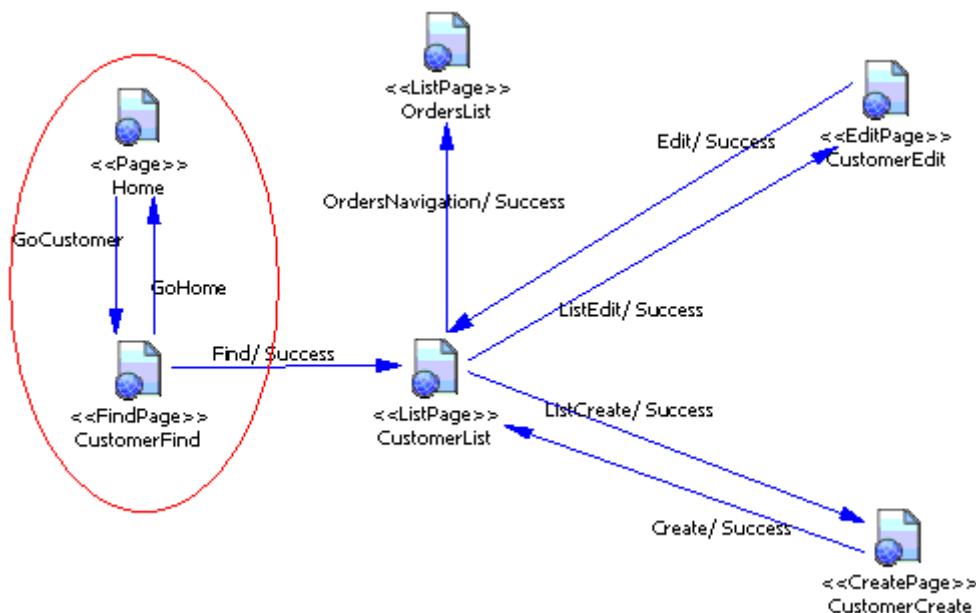
After create a new Page, when the code generation, a default JSF page and its page bean will be generated.

### 2.9.3.4.1.2 Adding a New PageFlow

You add a new PageFlow from the Toolbox.

#### Procedure

1. Select *Transition* from the Toolbox.
2. Draw a transition from the source state, e.g. Home, to the target state, e.g. CustomerFind.
3. Open the property sheet of the transition and click the *Trigger* tab.
4. Click the *Create* tool to the right of the *Trigger Event* field to create a new event, enter an appropriate name for the event and click *OK* to return to the transition property sheet.
5. Click *OK* to create the new PageFlow:



After you modify the default pageflow, and generate JSF codes, the corresponding default JSF pages, page beans and faces-config file will be updated.

## 2.9.3.5 Generating JSF Pages

Before generation, make sure that you have attached the Hibernate Extension file to the model, and checked the model for errors.

### Context

You can edit, deploy and test JSF pages in the Eclipse Web Tools Platform IDE. If the IDE does not include JSF runtime Jar files, you should download the appropriate release of the Apache MyFaces JSF implementation from the : [Apache MyFaces Project website](#). You need the following jar files:

- commons-beanutils-1.6.1.jar
- commons-codec-1.2.jar
- commons-collections-3.0.jar
- commons-digester-1.5.jar
- commons-el.jar
- commons-logging.jar
- commons-validator.jar
- log4j-1.2.8.jar

### Procedure

1. Select **Tools** **General Options**, click the **Variables** node, and add a variable **JSF\_LIB** with your JSF Jar file library folder. Then, click the **Named Paths** node, and add a named path **\_JSF** with the folder containing your JSF style sheets, headers, footers, and images.
2. Select **Language** **Generate Java Code** to open the Generation dialog.
3. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see [Checking an OOM \[page 251\]](#)).
4. [optional] Click the **Options** tab and specify the Web root directory.
5. Click **OK** to begin generation.

The following files are generated:

- Persistent files (persistent classes, DAO, ...)
- Eclipse and Eclipse WTP project artifacts
- A home page
- A find, list, create, and edit JSF page for persistent classes:
- Managed beans
- Page flows (face configuration files)

6. You can deploy a JSF Web application in a Web server or an application server that supports JSP. For example, Apache Tomcat, JBoss.

### **2.9.3.5.1 Testing JSF Pages with Eclipse WTP**

You test JSF Pages with Eclipse WTP.

#### **Procedure**

1. Install a Web server such as Tomcat or an application server such as JBoss.
2. Generate Java code and import the JSF project into Eclipse. The project is built.
3. Configure your Web server or application server, and start the database.
4. Start the Web server or application server using the WTP Servers view.
5. Right-click the index.jsp under the webroot folder, select  *Run As*  *Run on Server*, and select the server you want to use.

### **2.9.3.5.2 Testing JSF Pages with Apache Tomcat**

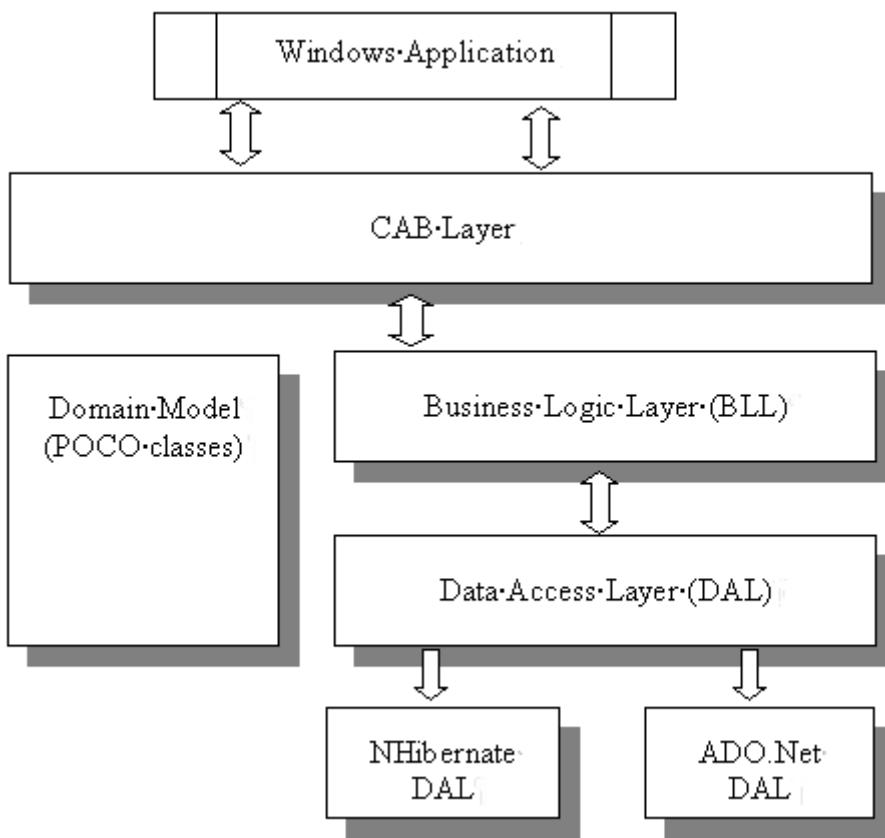
You test JSF Pages with Apache Tomcat.

#### **Procedure**

1. Install the Tomcat Web server.
2. Generate Java code, and import the JSF project into Eclipse. The project is built.
3. Copy the <ProjectName> folder under the .deployables folder into the Apache Tomcat webapps folder. Where <ProjectName> is the Eclipse project name.
4. Start the database and start the Tomcat server.
5. Run the Web application using the URL: `http://<hostname>:8080/<ProjectName>/index.jsp`. If Apache Tomcat is installed locally, <hostname> is localhost.

## **2.10 Generating .NET 2.0 Persistent Objects and Windows Applications**

PowerDesigner follows the best practices and design patterns to produce n-tier architecture enterprise applications for the .NET framework, as shown in the following figure:



PowerDesigner can be used to generate all these layers:

- Domain Model - contains persistent POCOs (Plain Old CLR Objects), which are similar to Java's POJOs. These act as information holders for the application and do not contain any business logic. A primary key class is generated for each persistent class in order to help the "find-by-primary-key" function, especially when the class has a composite primary identifier.
- Data Access Layer - follows the standard DAO pattern, and provides typical CRUD methods for each class. This layer is divided into two parts, one of which contains interfaces for DAL, while the other contains the implementation for these interfaces, using ADO.NET technology to access databases.

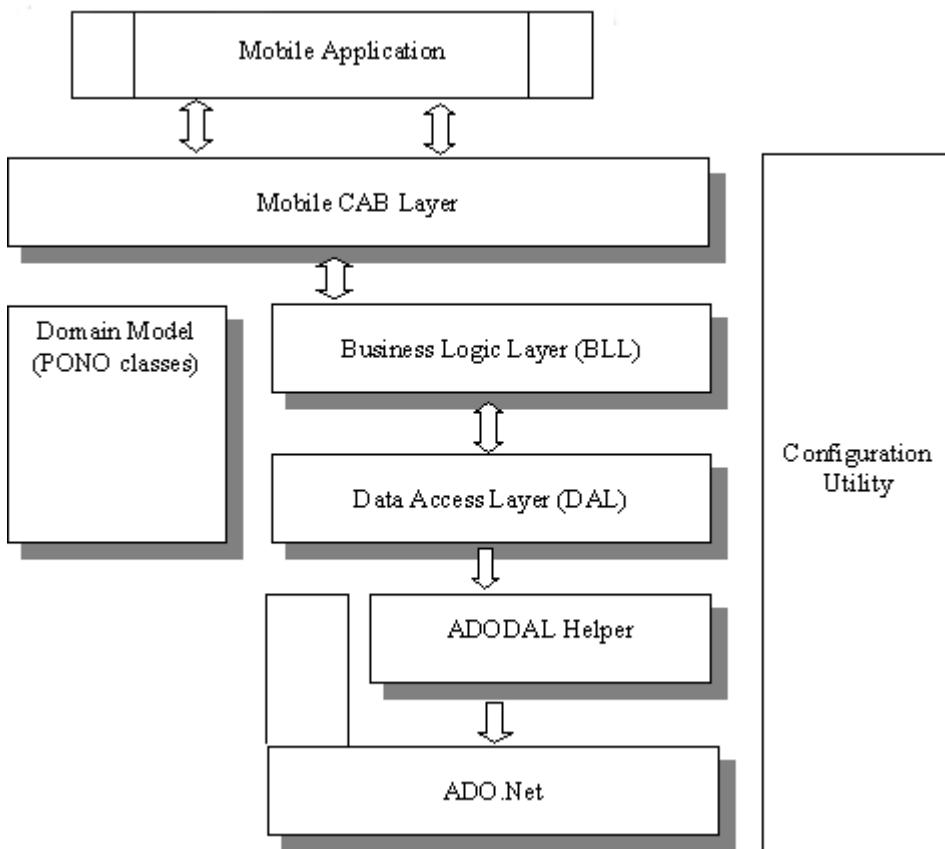
The DAL Helper provides common features used by all the DAL implementations, such as connection and transaction management, and the supply of SQL command parameters. Some common classes, such as Session, Criteria, and Exception, are also defined.

PowerDesigner supports two kinds of DAL implementation:

- ADO.NET (see [Generating ADO.NET and ADO.NET CF Persistent Objects \[page 510\]](#))
- Nhibernate (see [Generating NHibernate Persistent Objects \[page 519\]](#))
- Business Logic Layer - contains the typical user-defined business logic. This is a wrapper for the DAL, exposing CRUD functionalities provided by the DAL underneath. You can customize this layer according to your needs.
- Windows Application - the Composite UI Application Block, or CAB layer helps you build complex user interface applications that run in Windows. It provides an architecture and implementation that assists with building applications by using the common patterns found in line-of-business client applications.

PowerDesigner can generate data-centric windows applications based on the CAB (see [Generating Windows or Smart Device Applications \[page 541\]](#)).

The .NET CF (Compact Framework) has a similar organization, but with a configuration utility class that provides the capability to load and parse configuration used in different layers, e.g., data source configuration, log and exception configuration, etc:



PowerDesigner supports the ADO.NET DAL implementation for the .NET CF (see [Generating ADO.NET and ADO.NET CF Persistent Objects \[page 510\]](#))

## 2.10.1 Generating ADO.NET and ADO.NET CF Persistent Objects

Microsoft ADO.NET and ADO.NET CF are database-neutral APIs.

PowerDesigner supports ADO.NET and ADO.NET CF through extension files that provide enhancements to support:

- Cascade Association: one-to-one, one-to-many, many-to-one, and complex many-to-many associations.
- Inheritance: table per class, table per subclass, and table per concrete class hierarchies.
- Value Types
- SQL statements: including complex SQL statement like Join according to OOM model and PDM model.

To enable the ADO.NET or ADO.NET CF extensions in your model, select  **Model**  **Extensions**, click the *Attach an Extension* tool, select the ADO.NET or ADO.NET Compact Framework file (on the *O/R Mapping* tab), and click **OK** to attach it.

PowerDesigner also supports the design of .NET classes, database schema and Object/Relational mapping (O/R mapping), and can use these metadata, to generate ADO.NET and ADO.NET CF persistent objects including:

- Persistent .NET classes (POCOs)
- ADO.NET O/R mapping classes (ADODALHelper)
- DAL factory
- Data Access Objects (DAL)

### 2.10.1.1 ADO.NET and ADO.NET CF Options

To set the database connection parameters and other ADO.NET or ADO.NET CF options, double-click the model name in the browser to open its property sheet, and click the ADO.NET or ADO.NET CF tab.

| Option               | Description   |
|----------------------|---|
| Target Device        | [ADO.NET CF only] Specifies the operating system on which the application will be deployed.   |
| Output file folder   | [ADO.NET CF only] Specifies the location on the device to which the application will be deployed. Click the ellipsis button to the right of this field to edit the root location and add any appropriate sub-directories.   |
| Data Provider        | Specifies which data provider you want to use. For ADO.NET, you can choose between: <ul style="list-style-type: none"><li>• OleDb</li><li>• SqlCommand</li><li>• ODBC</li><li>• Oracle</li></ul> For ADO.NET CF, you can choose between: <ul style="list-style-type: none"><li>• Microsoft SQL Server 2005 Mobile Edition</li><li>• Sybase ASA Mobile Edition</li></ul> |
| Connection String    | Specifies the connection string. You can enter this by hand or click the ellipsis tool to the right of the field to use a custom dialog. For information about the provider-specific parameters used to build the connection string, see <a href="#">Configuring Connection Strings [page 535]</a> .  |
| Default access       | Specifies the default class attribute access type. This and the other package options, are valid for the whole model. You can fine-tune these options for an individual package through its property sheet.   |
| Default cascade      | Specifies the default cascade type.   |
| Default command type | Specifies the default command type, which can be overridden by concrete class.  |

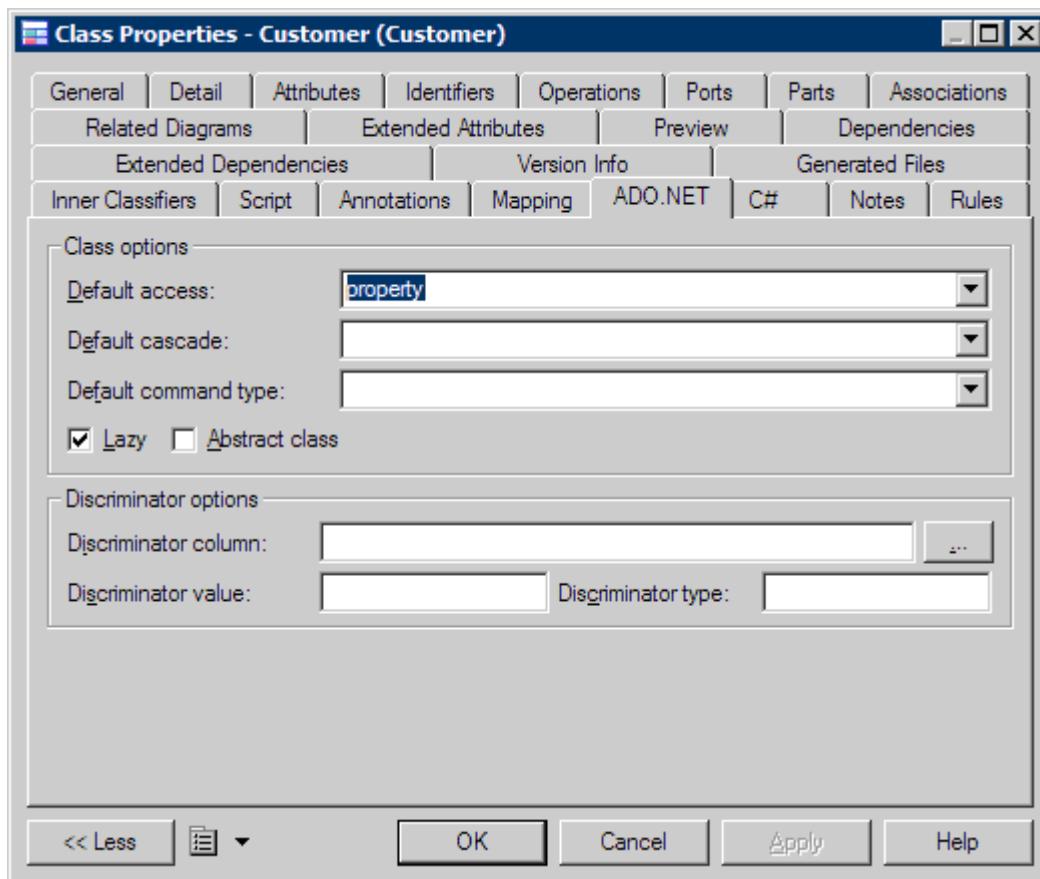
| Option       | Description   |
|--------------|---|
| DALContainer | Specifies the collection type returned from database. You can choose between Generic List Collection and System.Collections.ArrayList.  |
| Logging Type | [ADO.NET only] The common logging component is Log4Net, but you can reuse it as well if you have your own logging framework. By default, the value of logging type is Console type, but PowerDesigner also supports "None" or Log4Net |

## 2.10.1.2 Class Mappings

There are two kinds of classes in ADO.NET and ADO.NET CF:

- Entity classes - have their own database identities, mapping files and life cycles
- Value type classes - depend on entity classes. Also known as component classes

Framework-specific class mapping options are defined on the ADO.NET or ADO.NET CF tab of the class property sheet:



| Option               | Description   |
|----------------------|---|
| Default cascade      | Specifies the default cascade style.  |
| Default access       | Specifies the default access type (field or property)   |
| Default command type | Specifies command type, currently we supply Text and StoreProcedure to users.   |
| Lazy                 | Specifies that the class should be lazy fetching.   |
| Abstract class       | Specifies that the class is abstract.   |
| Discriminator column | Specifies the discriminator column or formula for polymorphic behavior in a one table per hierarchy mapping strategy. |
| Discriminator value  | Specifies a value that distinguishes individual subclasses, which are used for polymorphic behavior.                  |
| Discriminator type   | Specifies the discriminator type.   |

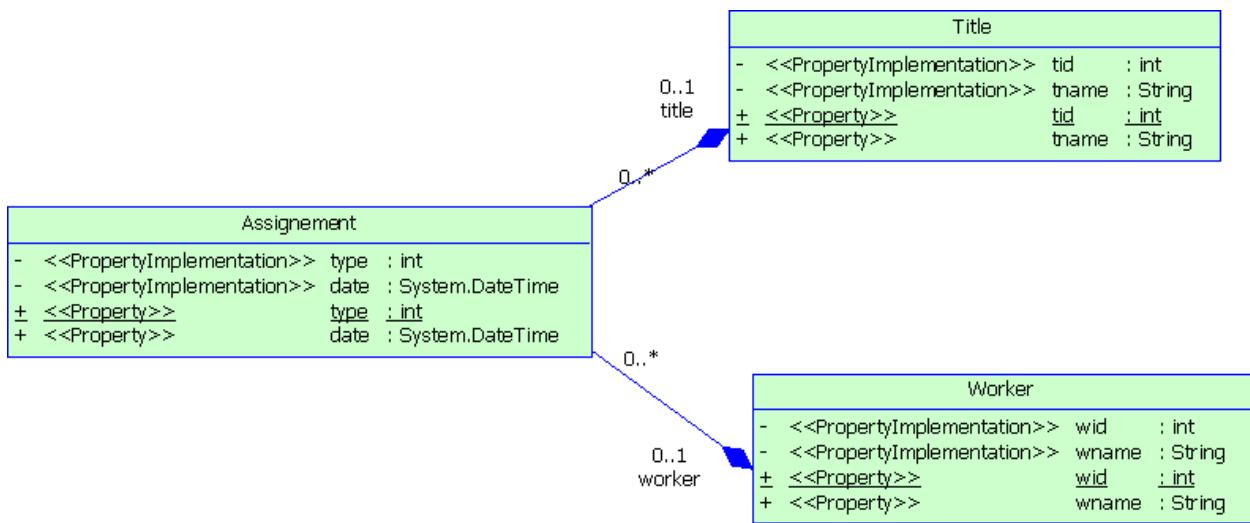
### 2.10.1.2.1 Primary Identifier Mappings

Primary identifier mapping is mandatory in ADO.NET and ADO.NET CF. Primary identifiers of entity classes are mapped to primary keys of master tables in data sources. If not defined, a default primary identifier mapping will be generated, but this may not work properly.

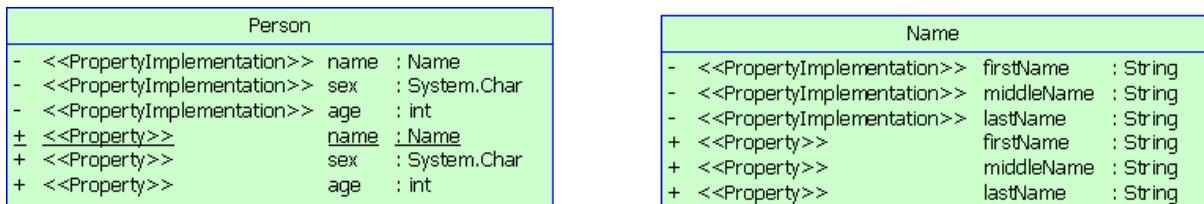
Mapped classes must declare the primary key column of the database table. Most classes will also have a property holding the unique identifier of an instance.

There are three kinds of primary identifier mapping in ADO.NET and ADO.NET CF:

- Simple identifier mapping - When a primary key is attached to a single column, only one attribute in the primary identifier can be mapped. This kind of primary key can be generated automatically. You can define increment, identity, sequence, etc., on the corresponding column in PDM.
- Composite identifier mapping - If a primary key comprises more than one column, the primary identifier can have multiple attributes mapped to these columns. In some cases, the primary key column could also be the foreign key column. In the following example, the Assignment class has a primary identifier with three attributes: one basic type attribute and two migrated attributes:



- Component identifier mapping - For more convenience, a composite identifier can be implemented as a separate value type class. The primary identifier has just one attribute with the class type. The separate class should be defined as a value type class. Component class mapping will be generated then. In the example below, three name attributes are grouped into one separate class **Name**, which is mapped to the same table as the **Person** class.



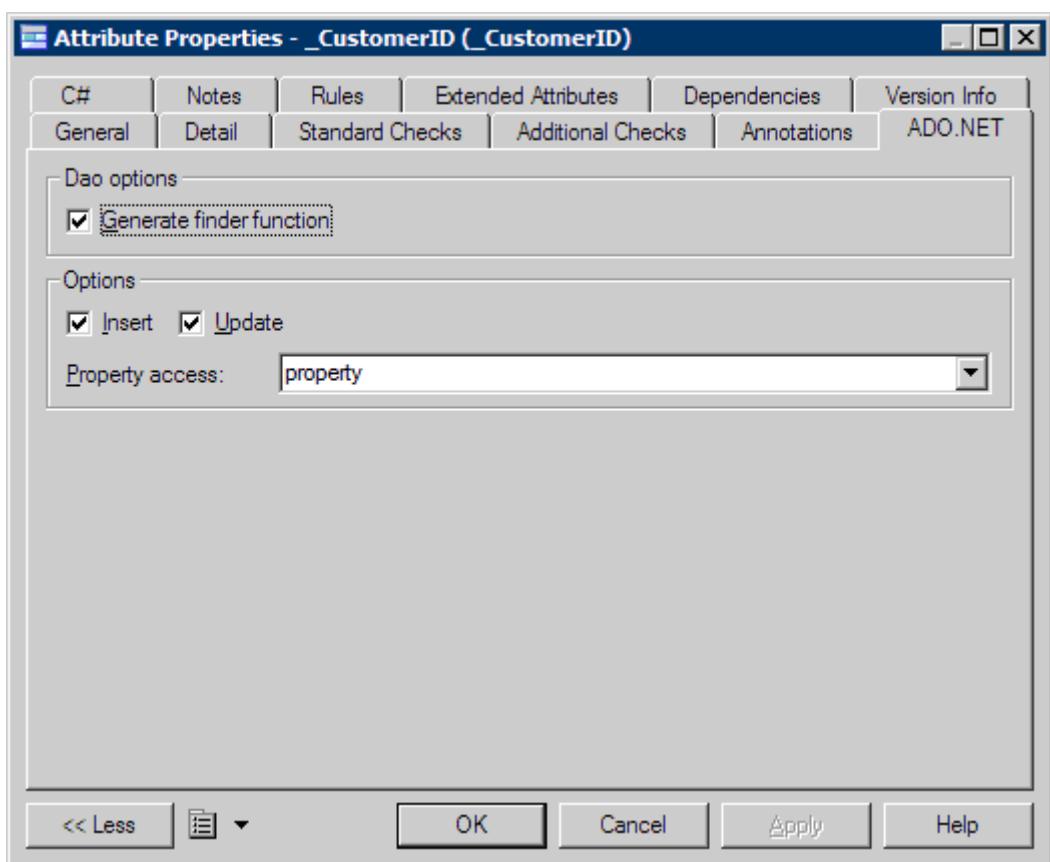
## 2.10.1.2.2 Attribute Mappings

Attributes can be migrated attributes or ordinary attributes. Ordinary attributes can be mapped to columns or formulas. Migrated attributes do not require attribute mapping.

The following types of mapping are possible:

- Component attribute mapping - A component class can define the attribute mapping as for other classes, except that there is no primary identifier.
- Discriminator mapping - In inheritance mapping with a one-table-per-hierarchy strategy, the discriminator needs to be specified in the root class. You can define the discriminator in the ADO.NET or ADO.NET CF tab of the class property sheet.

Framework-specific attribute mapping options are defined in the ADO.NET or ADO.NET CF tab of the Attribute property sheet.



| Option                   | Description   |
|--------------------------|---|
| Generate finder function | Generates a finder function for the attribute.  |
| Insert                   | Specifies that the mapped columns should be included in any SQL INSERT statements.  |
| Update                   | Specifies that the mapped columns should be included in any SQL UPDATE statements.  |
| Lazy                     | Specifies that this property should be fetched lazily when the instance variable is first accessed (requires build-time byte code instrumentation). |
| Property access          | Specifies the strategy used for accessing the property value.   |

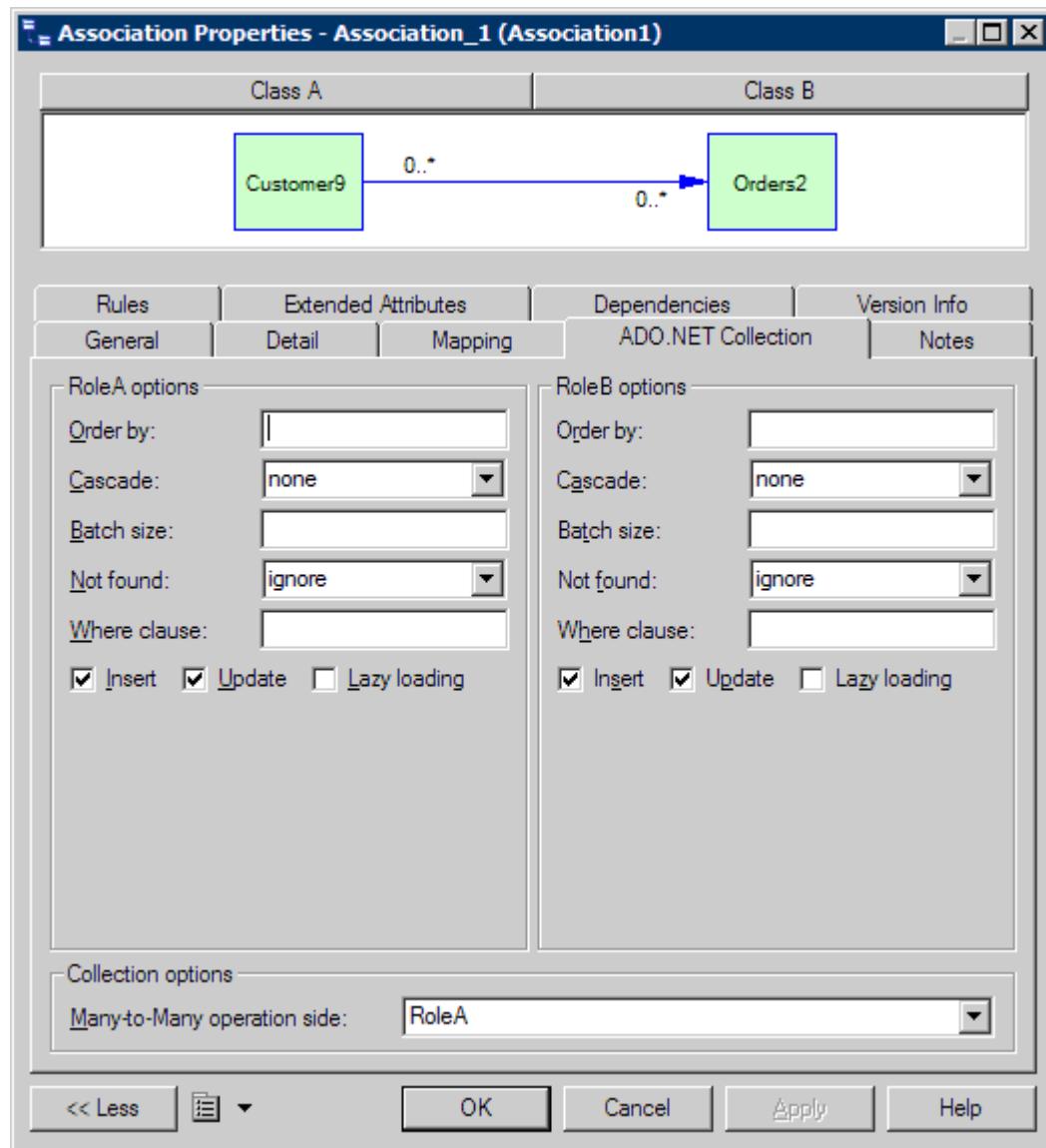
## 2.10.1.3 Defining Association Mappings

ADO.NET and ADO.NET CF support one-to-one, one-to-many/many-to-one, and many-to-many association mappings. PowerDesigner allows you to define standard association attributes like Container Type class, role navigability, array size and specific extended attributes for association mappings.

### Procedure

Open the Association property sheet and click the *ADO.NET* or *ADO.NET CF Collection* tab.

### Results



- 
1. Define the appropriate options and then click *OK*.

The following options are available on this tab:

| Option                      | Description   |
|-----------------------------|---|
| Order by                    | Specifies a table column (or columns) that define the iteration order of the Set or bag, together with an optional asc or desc.                             |
| Cascade                     | Specifies which operations should be cascaded from the parent object to the associated object.  |
| Batch size                  | Specifies the batch load size.  |
| Not found                   | Specifies how foreign keys that reference missing rows will be handled: ignore will treat a missing row as a null association.                              |
| Insert                      | Specifies that the mapped columns should be included in any SQL INSERT statements.  |
| Update                      | Specifies that the mapped columns should be included in any SQL UPDATE statements.  |
| Lazy                        | Specifies that this property should be fetched lazily when the instance variable is first accessed.   |
| Many-to-Many operation side | Specifies the entry point when operating data in bi-directional many-to-many association. No matter the choice is RoleA or RoleB, the results are the same. |

#### 2.10.1.4 Defining Inheritance Mappings

ADO.NET and ADO.NET CF support the three basic inheritance mapping strategies:

- Table per class hierarchy
- Table per subclass
- Table per concrete class
- These strategies all follow the standard inheritance mapping definitions.

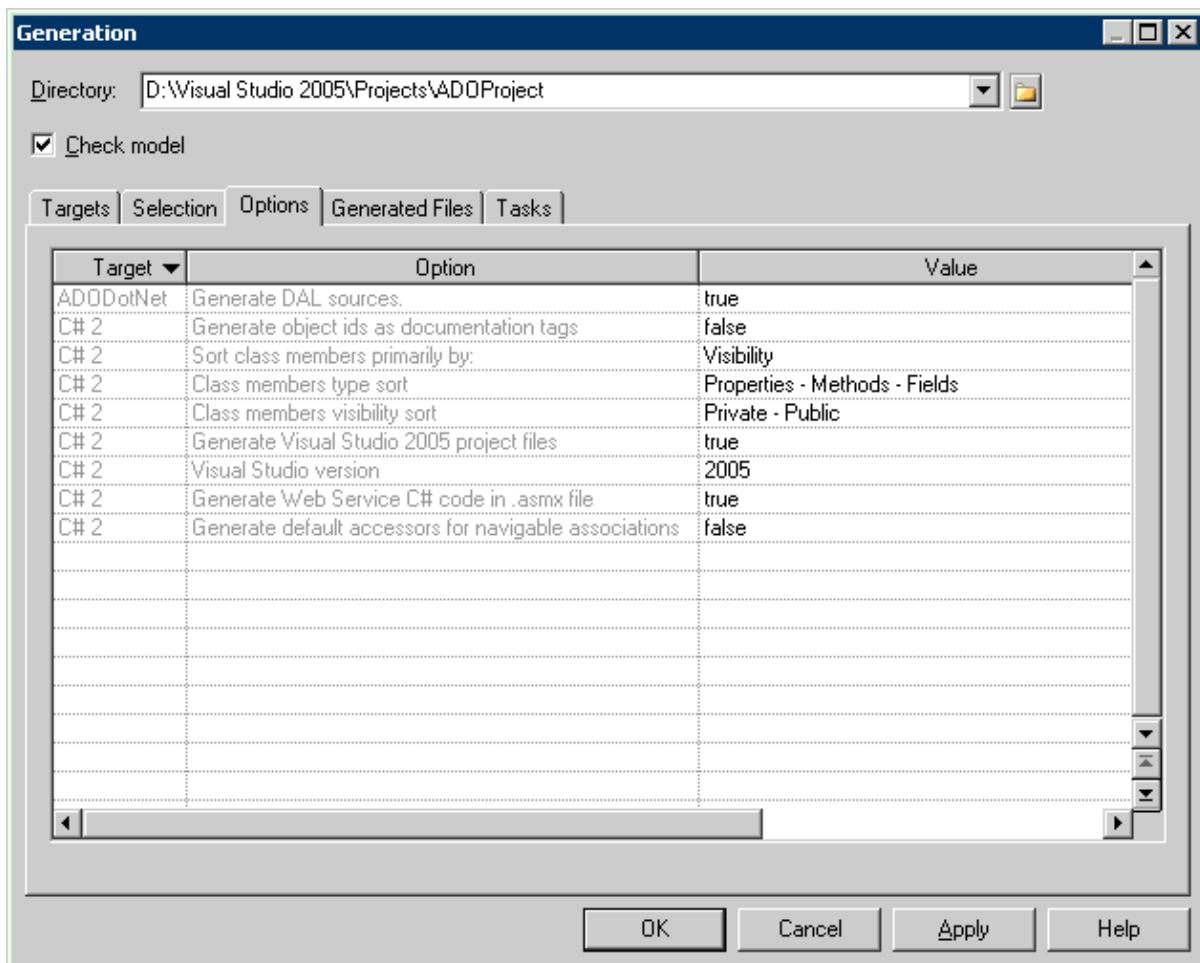
#### 2.10.1.5 Generating Code for ADO.NET or ADO.NET CF

In order to generate code for ADO.NET or ADO.NET CF, you must have the .NET Framework 2.0 Visual Studio.NET 2005 or above installed:

#### Procedure

1. Select  *Tools*  *Check Model* to verify if there are errors or warnings in the model. If there are errors, fix them before continuing with code generation.

2. Select **Language** > **Generate C# 2 Code or Generate Visual Basic 2005 Code** to open the Generation dialog box:
3. Specify the root directory where you want to generate the code and then click the Options tab:



4. [optional] To use DAL, set the Generate DAL sources option to true. For information about the standard C# and VB.NET generation options, see [Generating VB.NET Files \[page 388\]](#) or [Generating C# 2.0 Files \[page 420\]](#).
5. Click OK to generate code immediately or Apply and then Cancel to save your changes for later.

## Results

Once generation is complete, you can use an IDE such as Visual Studio.NET 2005 to modify the code, compile, and develop your application.

## 2.10.2 Generating NHibernate Persistent Objects

NHibernate is a port of the Hibernate Core for Java to the .NET Framework. It handles persistent POCOs (Plain Old CLR Objects) to and from an underlying relational database.

PowerDesigner supports NHibernate through an extension file that provides enhancements to support all the common .NET idioms, including association, inheritance, polymorphism, composition, and the collections framework are supported. NHibernate allows you to express queries in its own portable SQL extension (HQL), as well as in native SQL, or with Criteria and Example objects.

To enable the NHibernate extensions in your model, select **Model > Extensions**, click the *Attach an Extension* tool, select the **NHibernate** file (on the *O/R Mapping* tab), and click **OK** to attach it.

PowerDesigner supports the design of .NET classes, database schema, and Object/Relational mapping (O/R mapping). Using these metadata, PowerDesigner can generate NHibernate persistent objects including:

- Persistent .NET classes (domain specific objects)
- NHibernate Configuration file
- NHibernate O/R mapping files
- DAL factory
- Data Access Objects (DAL)

### 2.10.2.1 NHibernate Options

To set the database connection parameters and other NHibernate options, double-click the model name in the browser to open its property sheet, and click the **NHibernate** tab.

| Option            | Description   |
|-------------------|---|
| Dialect           | Specifies the dialect, and hence the type of database.<br>Scripting name: <code>dialect</code>  |
| Driver class      | Specifies the driver class.<br>Scripting name: <code>connection.driver_class</code>   |
| Connection String | Specifies the connection string. You can enter this by hand or click the ellipsis tool to the right of the field to use a custom dialog. For information about the provider-specific parameters used to build the connection string, see <a href="#">Configuring Connection Strings [page 535]</a> .<br>Scripting name: <code>connection.url</code> |
| Auto import       | Specifies that users may use an unqualified class name in queries.<br>Scripting name: <code>AutoImport</code>   |

| Option                        | Description  |
|-------------------------------|--|
| Default access                | Specifies the default class attribute access type. This and the other package options, are valid for the whole model. You can fine-tune these options for an individual package through its property sheet.<br><br>Scripting name: DefaultAccess |
| Specifies the default cascade | Specifies the default cascade type.<br><br>Scripting name: DefaultCascade  |
| Schema name                   | Specifies the default database schema name.<br><br>Scripting name: SchemaName  |
| Catalog name                  | Specifies the default database catalog name.<br><br>Scripting name: CatalogName  |
| Show SQL                      | Specifies that SQL statements should be shown in the log.<br><br>Scripting name: show_sql  |
| Auto schema export            | Specifies the mode of creation from tables.<br><br>Scripting name: hbm2ddl.auto  |

## 2.10.2.2 Defining Class Mappings

There are two kinds of classes in NHibernate:

- Entity classes - have their own database identities, mapping files and life cycles
- Value type classes - depend on entity classes. Also known as component classes

NHibernate uses mapping XML files to define the mapping metadata. Each mapping file can contain metadata for one or many classes. PowerDesigner uses the following grouping strategy:

- A separate mapping file is generated for each single entity class that is not in an inheritance hierarchy.
- A separate mapping file is generated for each inheritance hierarchy that has a unique mapping strategy. All mappings of subclasses are defined in the mapping file. The mapping file is generated for the root class of the hierarchy.
- No mapping file is generated for a single value type class that is not in an inheritance hierarchy. Its mapping is defined in its owner's mapping file.

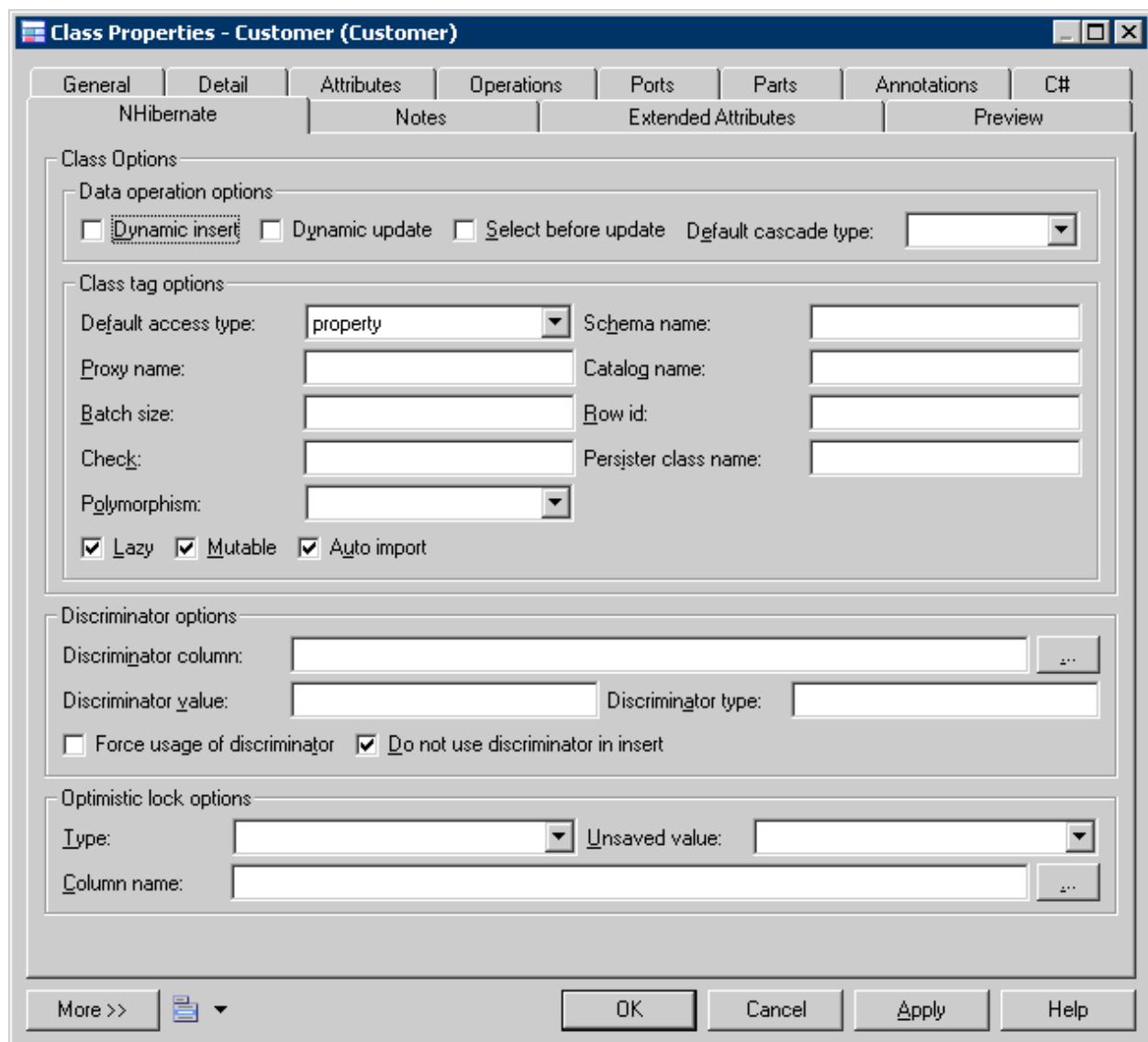
Classes can be mapped to tables or views. Since views have many constraints and limited functionality (for example they do not have primary keys and reference keys), some views cannot be updated, and the mappings may not work properly in some cases.

There are some conditions that need to be met in order to generate mapping for a specific class:

- The source can be generated. This may not be possible if, for example, the visibility of the class is private.

- The class is persistent.
- The generation mode is not set to *Generate ADT* (abstract data type) and *Value Type*.
- If the class is an inner class, it must be static, and have public visibility. NHibernate should then be able to create instances of the class.

NHibernate specific class mapping options are defined in the *NHibernate* tab of the class property sheet:



| Option         | Description   |
|----------------|---|
| Dynamic insert | Specifies that INSERT SQL should be generated at runtime and will contain only the columns whose values are not null.<br>Scripting name: dynamic-insert |

| Option               | Description   |
|----------------------|---|
| Dynamic update       | <p>Specifies that UPDATE SQL should be generated at runtime and will contain only the columns whose values have changed.</p> <p>Scripting name: <code>dynamic-update</code></p>       |
| Select before update | <p>Specifies that NHibernate should never perform a SQL UPDATE unless it is certain that an object is actually modified.</p> <p>Scripting name: <code>select-before-update</code></p> |
| Default cascade      | <p>Specifies the default cascade style.</p> <p>Scripting name: <code>default-cascade</code></p>   |
| Default access       | <p>Specifies the default access type (field or property)</p> <p>Scripting name: <code>default-access</code></p>   |
| Proxy name           | <p>Specifies an interface to use for lazy initializing proxies.</p> <p>Scripting name: <code>proxy</code></p>   |
| Batch size           | <p>Specifies a "batch size" for fetching instances of this class by identifier.</p> <p>Scripting name: <code>batch-size</code></p>  |
| Check                | <p>Specifies a SQL expression used to generate a multi-row check constraint for automatic schema generation.</p> <p>Scripting name: <code>check</code></p>                            |
| Polymorphism         | <p>Specifies whether implicit or explicit query polymorphism is used.</p> <p>Scripting name: <code>polymorphism</code></p>  |
| Schema name          | <p>Specifies the name of the database schema.</p> <p>Scripting name: <code>schema</code></p>  |
| Catalog name         | <p>Specifies the name of the database catalog.</p> <p>Scripting name: <code>catalog</code></p>  |
| Row id               | <p>Specifies that NHibernate can use the ROWID column on databases which support it (for example, Oracle).</p> <p>Scripting name: <code>rowed</code></p>                              |
| Persister class name | <p>Specifies a custom persistence class.</p> <p>Scripting name: <code>persister</code></p>  |
| Lazy                 | <p>Specifies that the class should be lazy fetching.</p> <p>Scripting name: <code>lazy</code></p>   |

| Option                             | Description  |
|------------------------------------|--|
| Mutable                            | <p>Specifies that instances of the class are mutable.</p> <p>Scripting name: <code>mutable</code></p>  |
| Abstract class                     | <p>Specifies that the class is abstract.</p> <p>Scripting name: <code>abstract</code></p>  |
| Auto import                        | <p>Specifies that an unqualified class name can be used in a query.</p> <p>Scripting name: <code>Auto-import</code></p>  |
| Discriminator column               | <p>Specifies the discriminator column or formula for polymorphic behavior in a one table per hierarchy mapping strategy.</p> <p>Scripting name: <code>discriminator</code></p> |
| Discriminator value                | <p>Specifies a value that distinguishes individual subclasses, which are used for polymorphic behavior.</p> <p>Scripting name: <code>discriminator-value</code></p>            |
| Discriminator type                 | <p>Specifies the discriminator type.</p> <p>Scripting name: <code>type</code></p>  |
| Force usage of discriminator       | <p>Forces NHibernate to specify allowed discriminator values even when retrieving all instances of the root class.</p> <p>Scripting name: <code>force</code></p>               |
| Do not use discriminator in insert | <p>Forces NHibernate to not include the column in SQL INSERTs.</p> <p>Scripting name: <code>insert</code></p>  |
| Optimistic lock type               | <p>Specifies an optimistic locking strategy.</p> <p>Scripting name: <code>optimistic-lock</code></p>   |
| Optimistic lock column name        | <p>Specifies the column used for optimistic locking. A field is also generated if this option is set.</p> <p>Scripting name: <code>version/ timestamp</code></p>               |
| Optimistic lock unsaved value      | <p>Specifies whether an unsaved value is null or undefined.</p> <p>Scripting name: <code>unsaved-value</code></p>  |

## 2.10.2.2.1 Primary Identifier Mappings

Primary identifier mapping is mandatory in NHibernate. Primary identifiers of entity classes are mapped to primary keys of master tables in data sources. If not defined, a default primary identifier mapping will be generated, but this may not work properly.

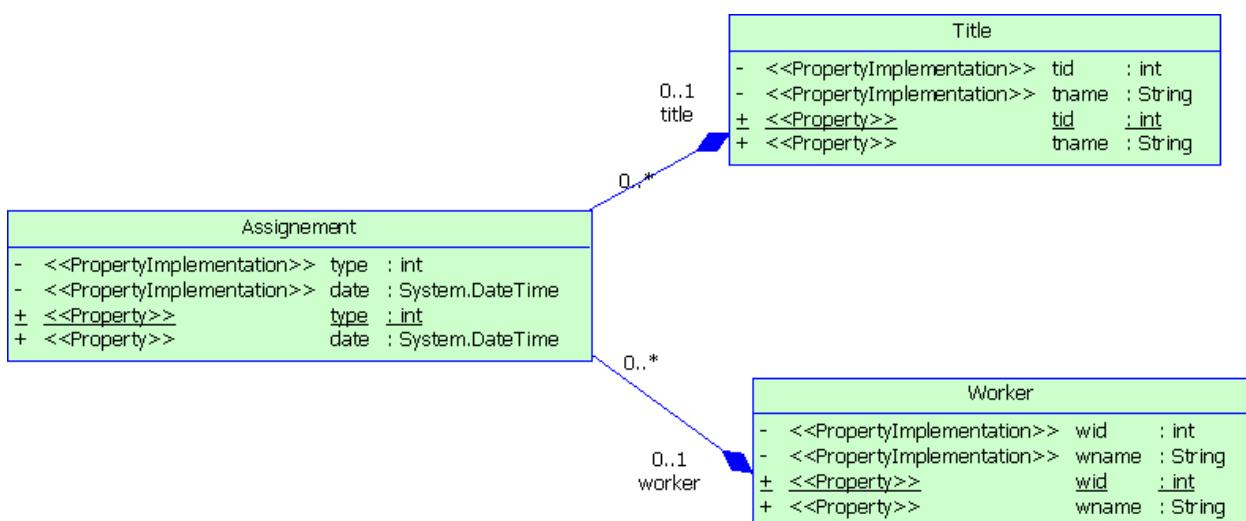
There are three kinds of primary identifier mapping in NHibernate:

- Simple identifier mapping -
- Composite identifier mapping
- Component identifier mapping

Mapped classes must declare the primary key column of the database table. Most classes will also have a property holding the unique identifier of an instance.

## Composite Identifier Mapping

If a primary key comprises more than one column, the primary identifier can have multiple attributes mapped to these columns. In some cases, the primary key column could also be the foreign key column.



In the above example, the Assignment class has a primary identifier with three attributes: one basic type attribute and two migrated attributes. The primary identifier mapping is as follows:

```
<composite-id>
  < key-property name="Type" access="property">
    < column name="Type" sql-type="integer"
      not-null="true"/>
  </key-property>
  <key-many-to-one name="title" access="property">
  </key-many-to-one>
  <key-many-to-one name="worker" access="property">
  </key-many-to-one>
</composite-id>
```

## Component Primary Identifier Mapping

For more convenience, a composite identifier can be implemented as a separate value type class. The primary identifier has just one attribute with the class type. The separate class should be defined as a value type class. Component class mapping will be generated then.

| Person   | Name   |
|--|--|
| - <<PropertyImplementation>> name : Name       | - <<PropertyImplementation>> firstName : String  |
| - <<PropertyImplementation>> sex : System.Char | - <<PropertyImplementation>> middleName : String |
| - <<PropertyImplementation>> age : int         | - <<PropertyImplementation>> lastName : String   |
| + <<Property>> name : Name                     | + <<Property>> firstName : String                |
| + <<Property>> sex : System.Char               | + <<Property>> middleName : String               |
| + <<Property>> age : int                       | + <<Property>> lastName : String                 |

In the example above, three name attributes are grouped into one separate class Name. It is mapped to the same table as Person class. The generated primary identifier is as follows:

```
<composite-id name="name" class="identifier.Name">
    <key-property name="firstName">
        <column name="name(firstName)">
            <sql-type>text</sql-type>
    </key-property>
    <key-property name="middleName">
        <column name="name(middleName)">
            <sql-type>text</sql-type>
    </key-property>
    <key-property name="lastName">
        <column name="name(lastName)">
            <sql-type>text</sql-type>
    </key-property>
</composite-id>
```

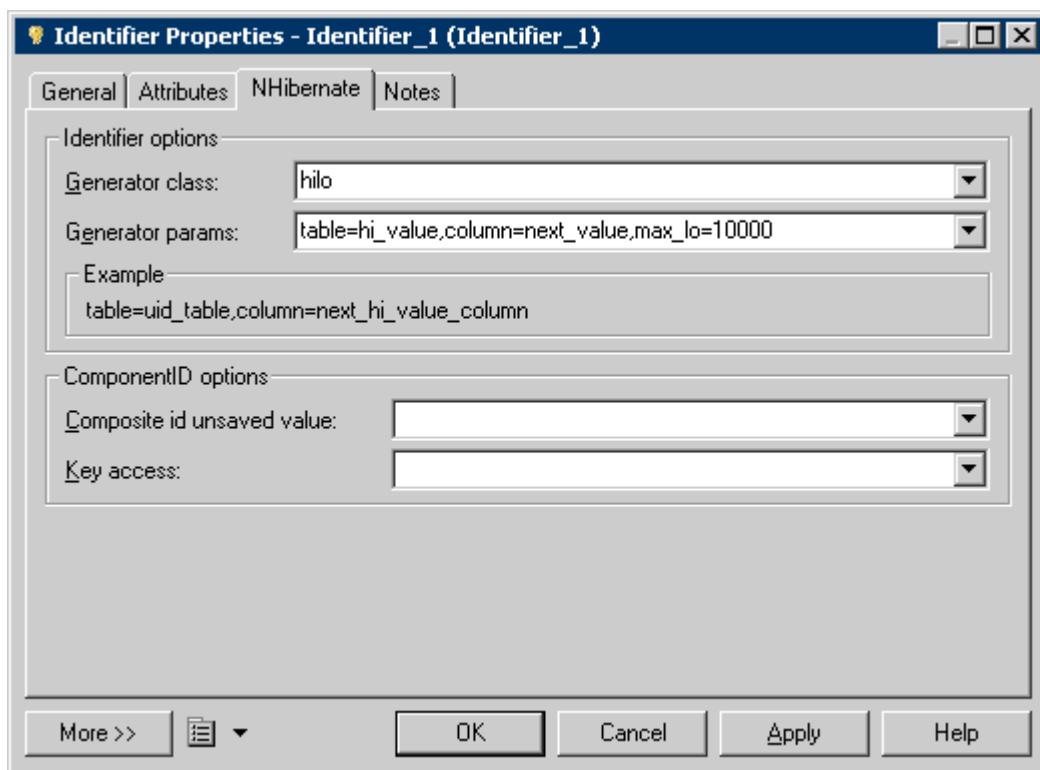
Note: The value type class must implement the java.io.Serializable interface, which implements the equals() and hashCode() methods.

### 2.10.2.2.1.1 Simple Identifier Mapping

When a primary key is attached to a single column, only one attribute in the primary identifier can be mapped. This kind of primary key can be generated automatically. You can define the generator class and parameters. There are many generator class types, such as increment, identity, sequence, etc. Each type of generator class may have parameters that are meaningful to it. See your NHibernate documentation for detailed information.

### Context

You can define the generator class and parameters in the NHibernate tab of the primary identifier property sheet. The parameters take the form of param1=value1; param2=value2.



## Procedure

1. Open the class property sheet and click the Identifiers tab. Double-click the entry to open its property sheet.
2. Click the NHibernate tab, select a generator class and define its parameters.

Example parameters:

- Select hilo in the Generator class list
- Enter "table=hi\_value,column=next\_value,max\_lo=10000" in the Generator params box. You should use commas to separate the parameters.

3. You can check the code in the Preview tab:

## Results

**Class Properties - Customer (Customer)**

| General | Detail              | Attributes | Operations | Ports | Parts | Annotations | C#      | NHibernate |
|---------|---------------------|------------|------------|-------|-------|-------------|---------|------------|
| Notes   | Extended Attributes |            |            |       |       |             | Preview |            |

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- NHibernate XML Mapping File -->
<!-- Author: xgzheng -->
<!-- Modified: Friday, April 14, 2006 11:18:19 AM -->
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0" default-cascade="save-update" auto-import="true">
    <class name="Model.Orders.Customer" mutable="true">
        <id name="Cid" access="property">
            <column name="cid" sql-type="integer" not-null="true"/>
            <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">10000</param>
            </generator>
        </id>
        <property name="Cname" access="property" insert="true" update="true">
            <column name="cname" sql-type="varchar(254)"/>
        </property>
        <set name="Order" lazy="true">
            <key>
                <column name="cid" sql-type="integer" not-null="false"/>
            </key>
            <one-to-many class="Model.Orders.Order" />
        </set>
    </class>
</hibernate-mapping>
```

NHibernate.Mapping File NHibernate.Model.IDAL Implement

More >> OK Cancel Apply Help

Note that, if there are several Primary identifier attributes, the generator is not used.

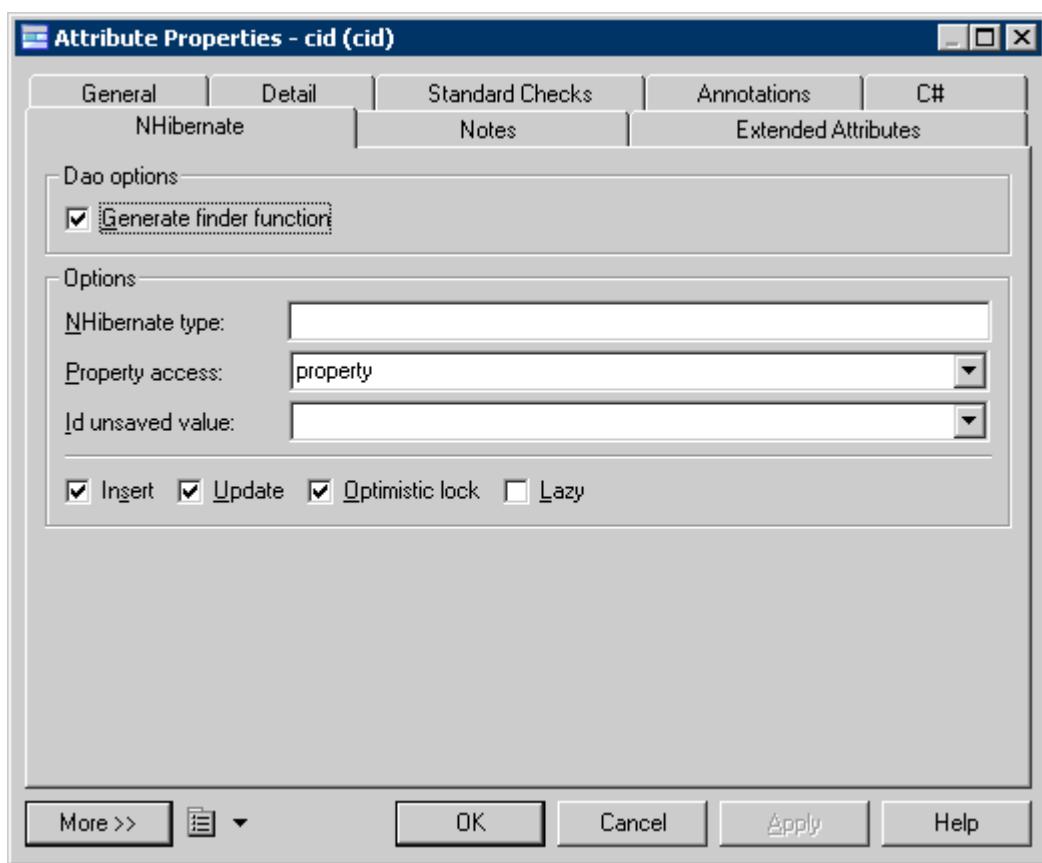
## 2.10.2.2 Attribute Mappings

Attributes can be migrated attributes or ordinary attributes. Ordinary attributes can be mapped to columns or formulas. Migrated attributes do not require attribute mapping.

The following types of mapping are possible:

- Map attribute to formula - When mapping an attribute to a formula, you should ensure that the syntax is correct. There is no column in the source table of the attribute mapping.
- Component attribute mapping - A component class can define the attribute mapping as for other classes, except that there is no primary identifier.
- Discriminator mapping - In inheritance mapping with a one-table-per-hierarchy strategy, the discriminator needs to be specified in the root class. You can define the discriminator in the NHibernate tab of the class property sheet.

NHibernate-specific attribute mapping options are defined in the NHibernate tab of the Attribute property sheet.



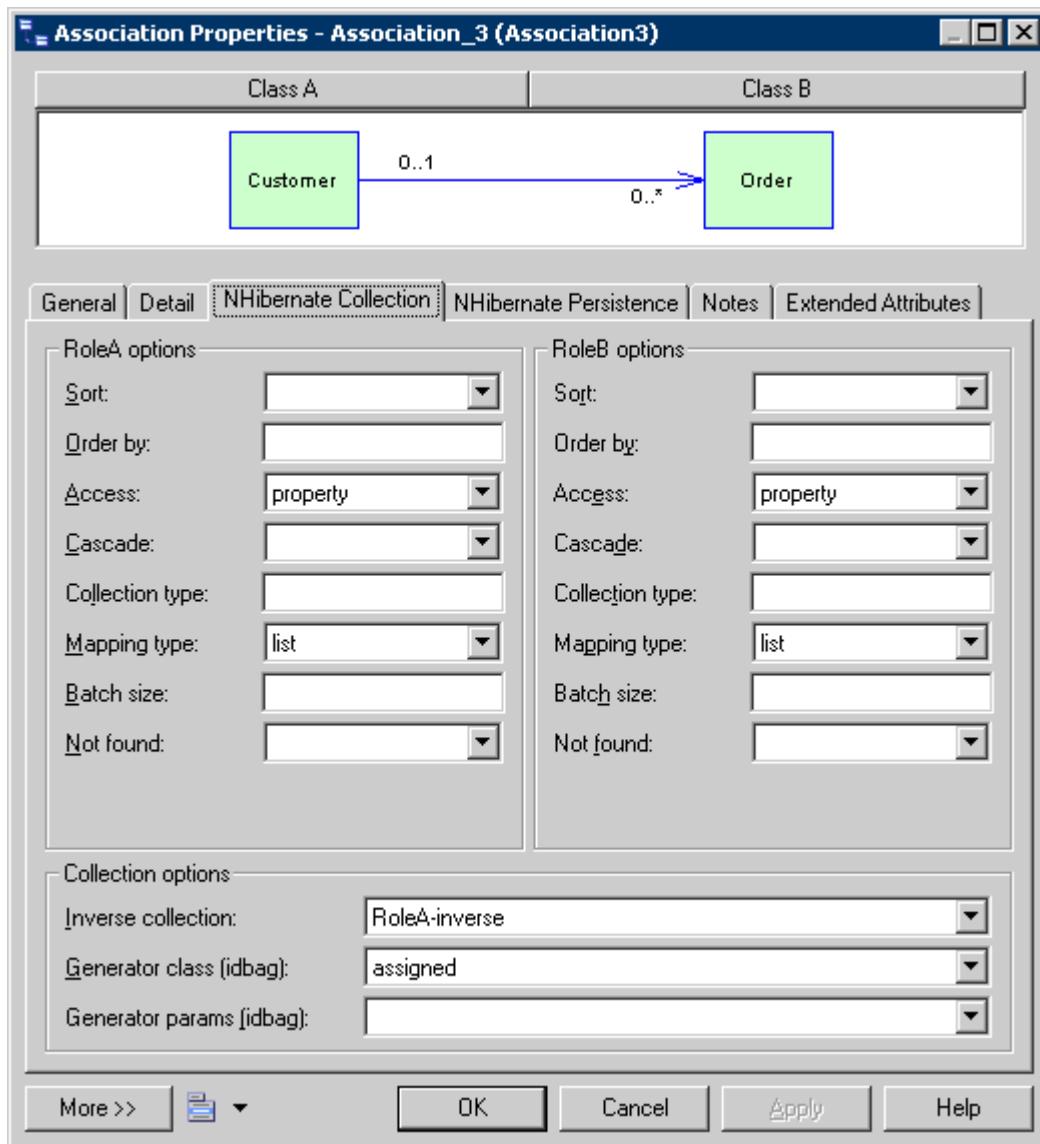
| Option                   | Description   |
|--------------------------|---|
| Generate finder function | Generates a finder function for the attribute.  |
| NHibernate type          | Specifies a name that indicates the NHibernate type.  |
| Property access          | Specifies the strategy that NHibernate should use for accessing the property value.   |
| Id unsaved value         | Specifies the value of an unsaved id.   |
| Insert                   | Specifies that the mapped columns should be included in any SQL INSERT statements.  |
| Update                   | Specifies that the mapped columns should be included in any SQL UPDATE statements.  |
| Optimistic lock          | Specifies that updates to this property require acquisition of the optimistic lock.   |
| Lazy                     | Specifies that this property should be fetched lazily when the instance variable is first accessed (requires build-time byte code instrumentation). |

## 2.10.2.3 Defining Association Mappings

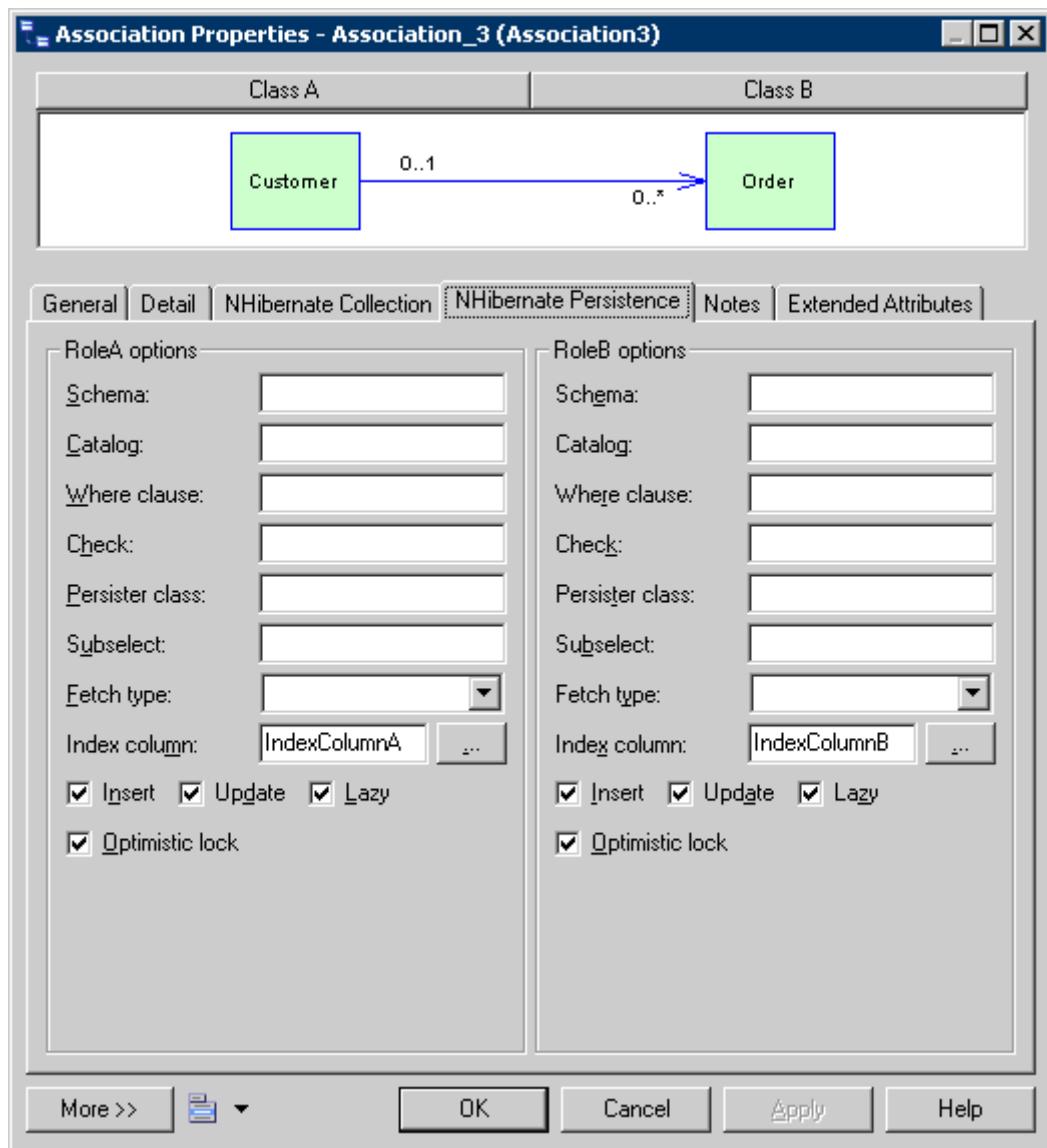
NHibernate supports one-one, one-to-many/many-to-one, and many-to-many association mappings. The mapping modeling is same with standard O/R Mapping Modeling. However, NHibernate provides special options to define its association mappings, which will be saved into <Class>.hbm.xml mapping file. PowerDesigner allows you to define standard association attributes like Container Type class, role navigability, array size and specific extended attributes for NHibernate association mappings.

### Procedure

1. Open the association property sheet and click the *NHibernate Collection* tab.
2. Define the collection management options (see [Defining NHibernate Collection Options \[page 531\]](#)).



3. Click the *NHibernate Persistence* tab, and define the persistence options (see [Defining NHibernate Persistence Options \[page 532\]](#)).



## 2.10.2.3.1 Defining NHibernate Collection Options

The following options are available:

| Option             | Description  |
|--------------------|--|
| Sort               | Specifies a sorted collection with natural sort order, or a given comparator class.<br>Scripting name: <code>sort</code>   |
| Order by           | Specifies a table column (or columns) that define the iteration order of the Set or bag, together with an optional asc or desc.<br>Scripting name: <code>order-by</code> |
| Access             | Specifies the strategy Nhibernate should use for accessing the property value.<br>Scripting name: <code>access</code>  |
| Cascade            | Specifies which operations should be cascaded from the parent object to the associated object.<br>Scripting name: <code>cascade</code>                                   |
| Collection type    | Specifies a name that indicates the NHibernate type.<br>Scripting name: <code>type</code>  |
| Batch size         | Specifies the batch load size.<br>Scripting name: <code>batch-size</code>  |
| Not found          | Specifies how foreign keys that reference missing rows will be handled: ignore will treat a missing row as a null association.<br>Scripting name: <code>not-found</code> |
| Inverse collection | Specifies that the role is the inverse relation of the opposite role.<br>Scripting name: <code>inverse</code>  |
| Mapping type       | Specifies the collection mapping type<br>Scripting name: <code>Set, Array, Map, or List.</code>  |

## 2.10.2.3.2 Defining NHibernate Persistence Options

The following options are available:

| Option          | Description   |
|-----------------|---|
| Schema          | Specifies the name of the schema.<br>Scripting name: <code>schema</code>  |
| Catalog         | Specifies the name of the catalog.<br>Scripting name: <code>catalog</code>  |
| Where clause    | Specifies an arbitrary SQL WHERE condition to be used when retrieving objects of this class.<br>Scripting name: <code>where</code>              |
| Check           | Specifies a SQL expression used to generate a multi-row check constraint for automatic schema generation.<br>Scripting name: <code>check</code> |
| Fetch type      | Specifies outer-join or sequential select fetching.<br>Scripting name: <code>fetch</code>   |
| Persister class | Specifies a custom persistence class.<br>Scripting name: <code>persister</code>   |
| Subselect       | Specifies an immutable and read-only entity to a database subselect.<br>Scripting name: <code>subselect</code>                                  |
| Index column    | Specifies the column name if users use list or array collection type.<br>Scripting name: <code>index</code>                                     |
| Insert          | Specifies that the mapped columns should be included in any SQL INSERT statements.<br>Scripting name: <code>insert</code>                       |
| Update          | Specifies that the mapped columns should be included in any SQL UPDATE statements.<br>Scripting name: <code>update</code>                       |
| Lazy            | Specifies that this property should be fetched lazily when the instance variable is first accessed.<br>Scripting name: <code>lazy</code>        |
| Optimistic lock | Specifies that a version increment should occur when this property is dirty.<br>Scripting name: <code>optimistic-lock</code>                    |

| Option     | Description  |
|------------|--|
| Outer join | Specifies to use an outer-join.<br>Scripting name: <code>outer-join</code> |

### 2.10.2.3.3 Defining NHibernate Collection Container Type

NHibernate supports Set, Bag, List, Array, and Map mapping type, it restricts the container type. PowerDesigner does not support Map mapping type.

| Collection Mapping Type | Collection Container Type |
|-------------------------|---------------------------|
| Set                     | lesi.Collections.ISet     |
| Bag                     | System.Collections.IList  |
| List                    | System.Collections.IList  |
| Array                   | <None>                    |

You can input the container type manually, or select the needed mapping type, and PowerDesigner will automatically select the correct container type.

### 2.10.2.4 Defining Inheritance Mappings

NHibernate supports the two basic inheritance mapping strategies:

- Table per class hierarchy
- Table per subclass

It does not support the "Table per concrete class" mapping strategy.

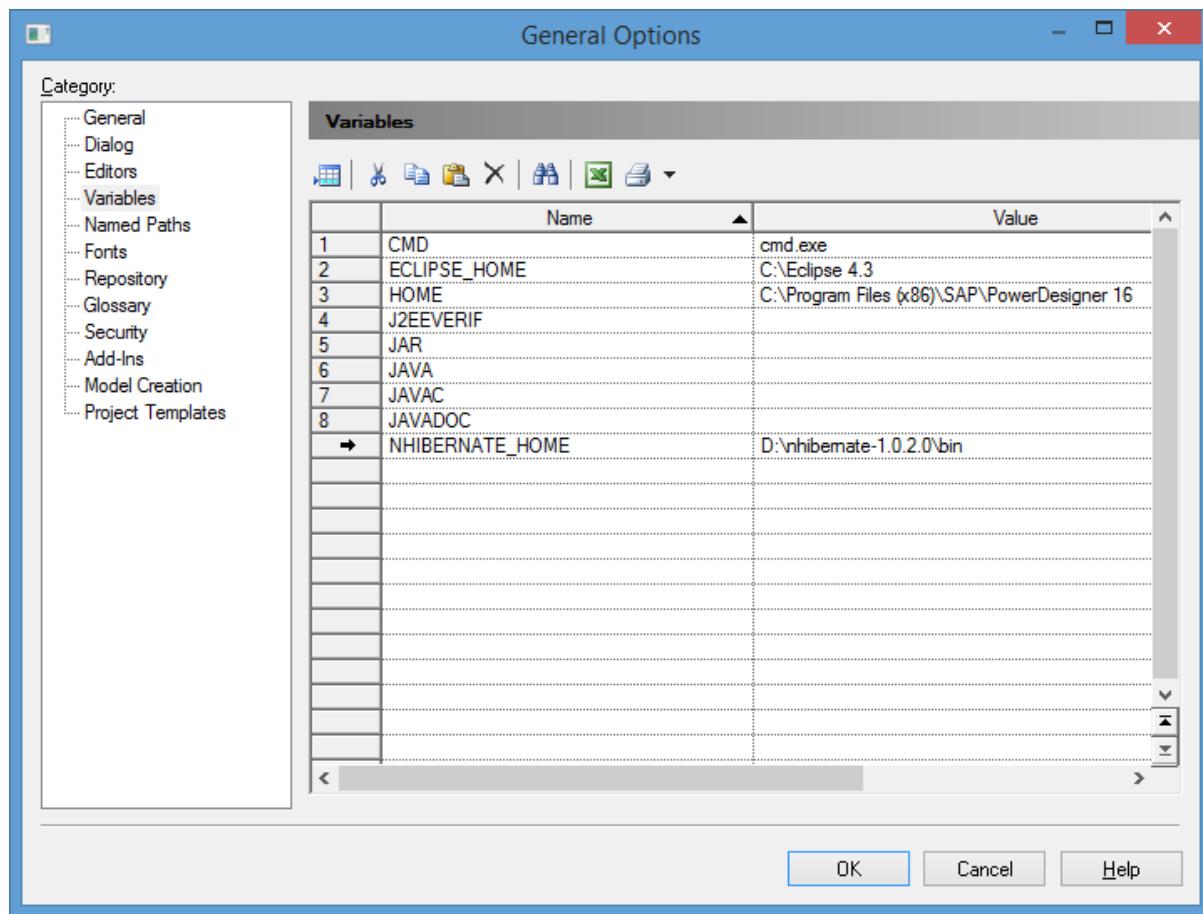
These strategies all follow the standard inheritance mapping definitions. However, a separate mapping file is generated for each inheritance hierarchy that has a unique mapping strategy. All mappings of subclasses are defined in the mapping file. The mapping file is generated for the root class of the hierarchy.

## 2.10.2.5 Generating Code for NHibernate

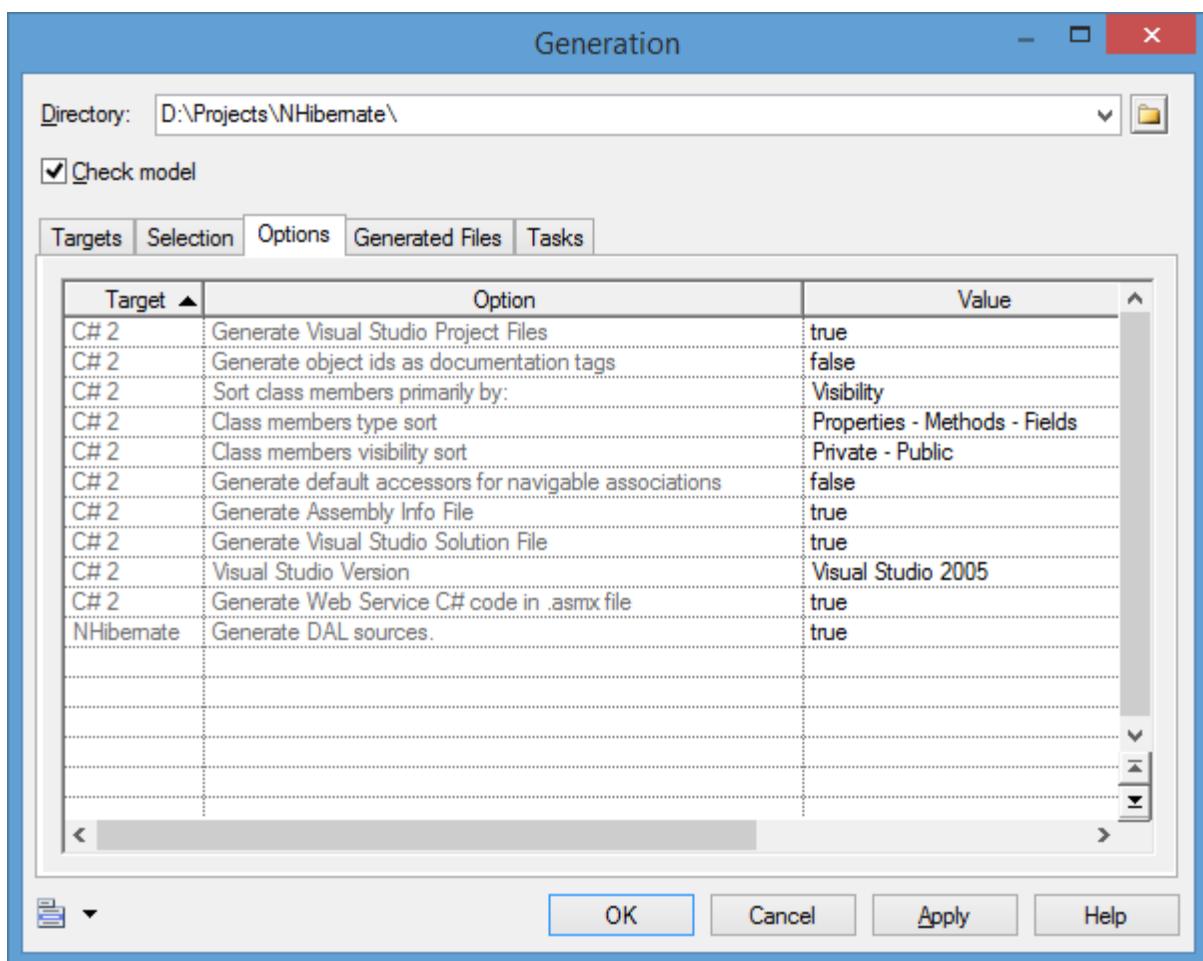
Before generating code for NHibernate, you must have NHibernate 1.0.2 or higher installed.

### Procedure

1. Select **Tools > Check Model** to verify if there are errors or warnings in the model. If there are errors, fix them before continuing with code generation.
2. Select **Tools > General Options**, and click the **Variables** node.



3. Add a variable **NHIBERNATE\_HOME** and, in the value field, enter the NHibernate home directory path. For example, `D:\nhibernate-1.0.2.0\bin`.
4. Select **Language > Generate C# 2** or **Generate Visual Basic 2005 Code** to open the Generation dialog box:
5. Specify the root directory where you want to generate the code, and then click the **Options** tab:



6. [optional] To use DAL, set the *Generate DAL sources* option to true. For information about the standard C# and VB.NET generation options, see the relevant chapters.
7. Click *OK* to generate code immediately or *Apply* and then *Cancel* to save your changes for later.

## Results

Once generation is complete, you can use an IDE such as Visual Studio.NET 2005 to modify the code, compile, and develop your application.

### 2.10.3 Configuring Connection Strings

PowerDesigner supports multiple types of database connection with each of the .NET frameworks.

To configure a connection string from the ADO.NET or ADO.NET CF tab, select a data provider, click the ellipsis button to open a provider-specific connection string dialog, enter the necessary parameters and click *Test Connection* to validate the connection and *Close*.

To configure a connection string from the NHibernate tab, select a dialect and driver class, click the ellipsis button to open a provider-specific connection string dialog, enter the necessary parameters and click *Test Connection* to validate the connection. Then click *Apply to Connection String* and *Close*.

Each connection requires a different set of parameters, which can be entered by hand in the Connection String field of the ADO.NET or NHibernate tab, or through custom dialogs accessible via the ellipsis tool to the right of this field.

The following parameters are required to configure an ODBC connection string:

| Option               | Description                                    |
|----------------------|--|
| ODBC source name     | Specifies the ODBC source name                 |
| User name / password | Specifies the database user name and password. |

The following parameters are required to configure an OLEDB connection string:

| Option                             | Description                                       |
|------------------------------------|---|
| Data provider                      | Specifies the data provider from the list.        |
| Server or file name                | Specifies the database server or file name.       |
| User name                          | Specifies the database user name and password.    |
| Use Windows NT integrated security | Specifies to use Windows NT integrated security.  |
| Allow saving password              | Specifies whether to allow saving password or not |
| Initial catalog                    | Specifies database's initial catalog.             |

The following parameters are required to configure a Microsoft SQL Server or Microsoft SQL Server Mobile Edition connection string:

| Option                                   | Description  |
|--|--|
| Server name                              | Specifies the server name.   |
| User name / Password                     | Specify the database user name and password.   |
| Authentication type                      | Specifies authentication type, Use SQL Server Authentication, or Use Windows Authentication. |
| Database name / file name / logical name | Specify the database name, file name, and the logical name for the database file.            |

The following parameters are required to configure an Oracle connection string:

| Option               | Description                                    |
|----------------------|--|
| Server name          | Specifies the server name.                     |
| User name / Password | Specifies the database user name and password. |

## 2.10.4 Generating Code for Unit Testing

You can run the tests using NUnit or Visual Studio Test System. PowerDesigner provides support for unit test code generation through an extension file.

### Context

If there are many persistent classes, it can be difficult to test them all to prove that:

- The mappings are correctly defined
- The CRUD (Create, Read, Update, Delete) options work
- The find methods work
- The navigations work

Usually, developers have to develop unit tests or user-interfaces in order to test these objects. PowerDesigner can automate this time-consuming process by using the NUnit or Visual Studio Test System (VSTS) to generate the unit test classes. Most code generated for these two UnitTest frameworks is very similar. The main differences are:

- Team Test use different attributes with NUnit in test class, such as [TestClass()] to [TestFixture] and [TestMethod()] to [Test] etc.
- AllTests file is not generated because all tests will be run in Visual Studio gui or command prompt.
- Log4net is replaced by test result .trx file that can be opened in Test Result window in Visual Studio.

Some conditions must be met to unit test for a class:

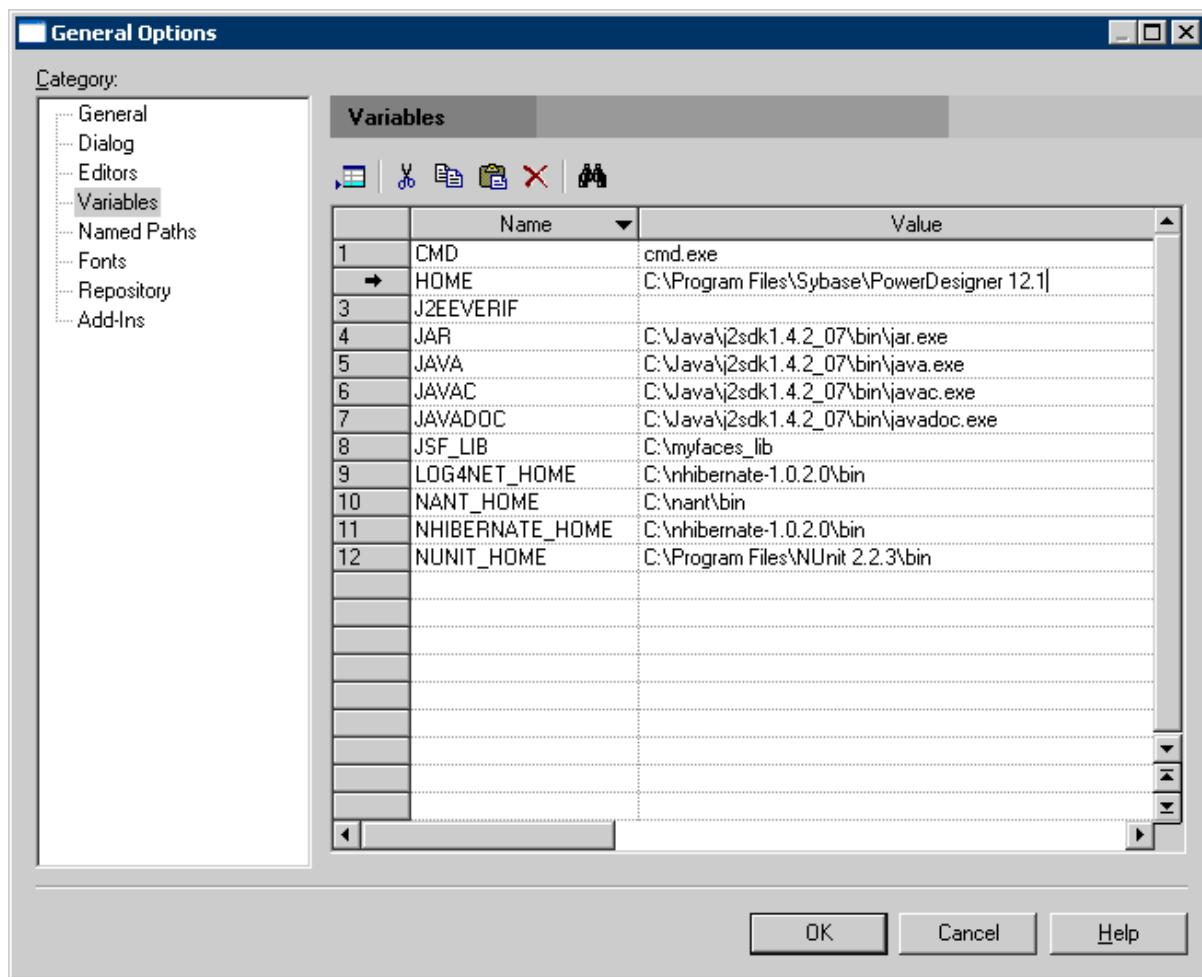
- Mapping of the class should be defined.
- The class can be instantiated. Unit test classes cannot be generated for abstract classes.
- The class is not a value type.
- The Mutable property is set to true. If Mutable is set to false, the class can not be updated or deleted.
- The class has no unfulfilled foreign key constraints. If any foreign key is mandatory, the parent class should be reachable (navigable on the parent class side) for testing.

To enable the unit test extensions in your model, select Model > Extensions, click the *Attach an Extension* tool, select the `UnitTest.NET` or `UnitTest.NET CF` file (on the *Unit Test* tab), and click *OK* to attach it.

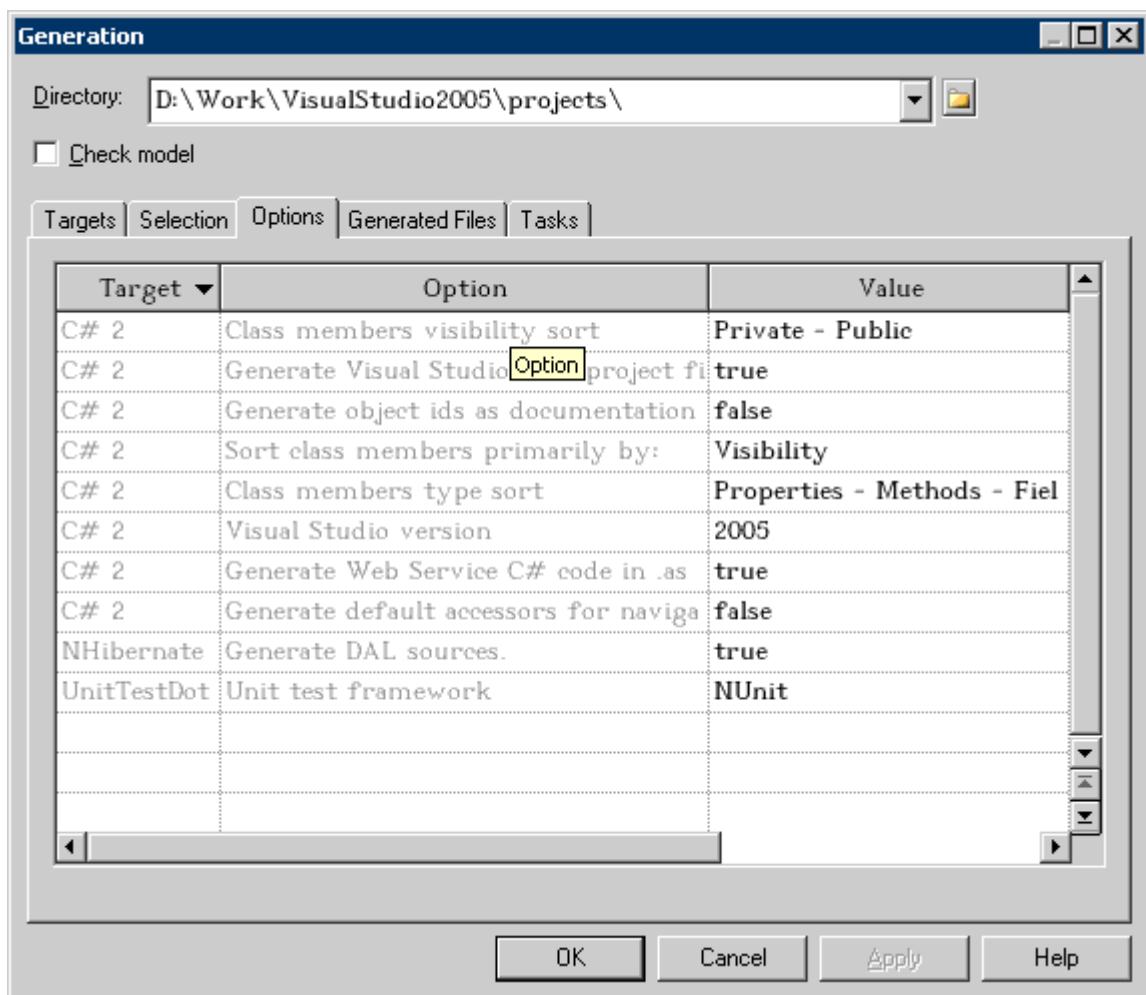
Before generating code for UnitTest, you must have NUnit 2.2.3 or higher installed (available at <http://www.nunit.org> ).

## Procedure

1. Select **Tools > Check Model** to verify if there are errors or warnings in the model. If there are errors, fix them before continuing with code generation.
2. Select **Tools > General Options**, and click the Variables node.
3. Add a variable NUNIT\_HOME and, in the value field, enter the NUnit home directory path. For example, D:\NUnit2.2.3\bin. Add a variable LOG4NET\_HOME in the same way if log4net is going to be used for logging.



4. Select **Language > Generate C# 2 Code** or **Generate Visual Basic 2005 Code** to open the Generation dialog box.
5. Specify the root directory where you want to generate the code, and then click the **Options** tab:

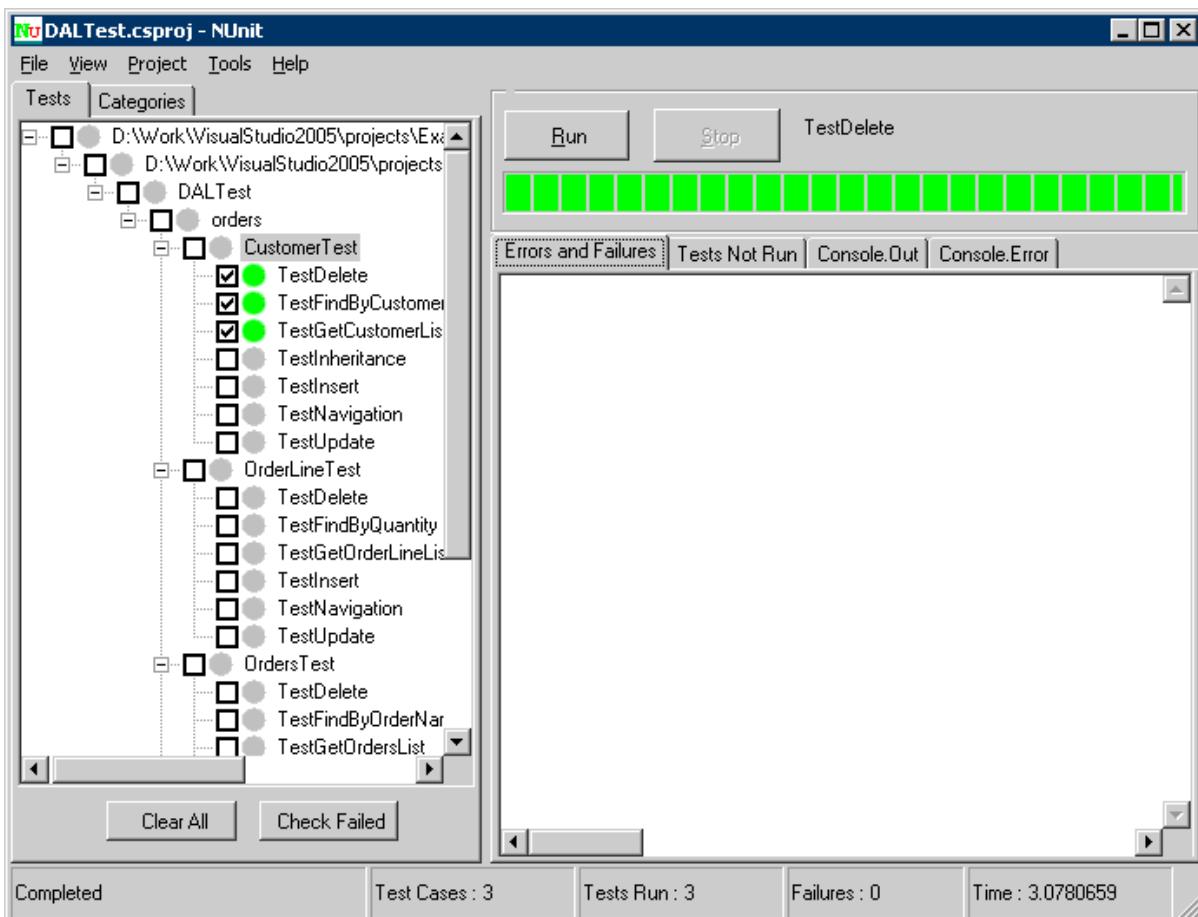


6. Select a UnitTest framework in *Unit test framework*. You can choose between Nunit or Visual Studio Team Test.
7. Click on *OK* to generate code immediately or *Apply*, and then *Cancel* to save your changes for later.

### 2.10.4.1 Running NUnit Unit Tests

After generating your test code, you can run it in one of three ways:

- Run in Nunit - NUnit has two different ways to run your test cases. The console runner, nunit-console.exe, is the fastest to launch, but is not interactive. The GUI runner, nunit-gui.exe, is a Windows Forms application that allows you to work selectively with your test cases and provides graphical feedback.  
NUnit also provide Category attribute, which provides an alternative to suites for dealing with groups of tests. Either individual test cases or fixtures may be identified as belonging to a particular category. Both the GUI and console test runners allow specifying a list of categories to be included in or excluded from the run. When categories are used, only the tests in the selected categories will be run. Those tests in categories that are not selected are not reported at all.
- NUnit GUI - The nunit-gui.exe program is a graphical runner. It shows the tests in an explorer-like browser window and provides a visual indication of the success or failure of the tests. It allows you to selectively run single tests or suites and reloads automatically as you modify and re-compile your code.



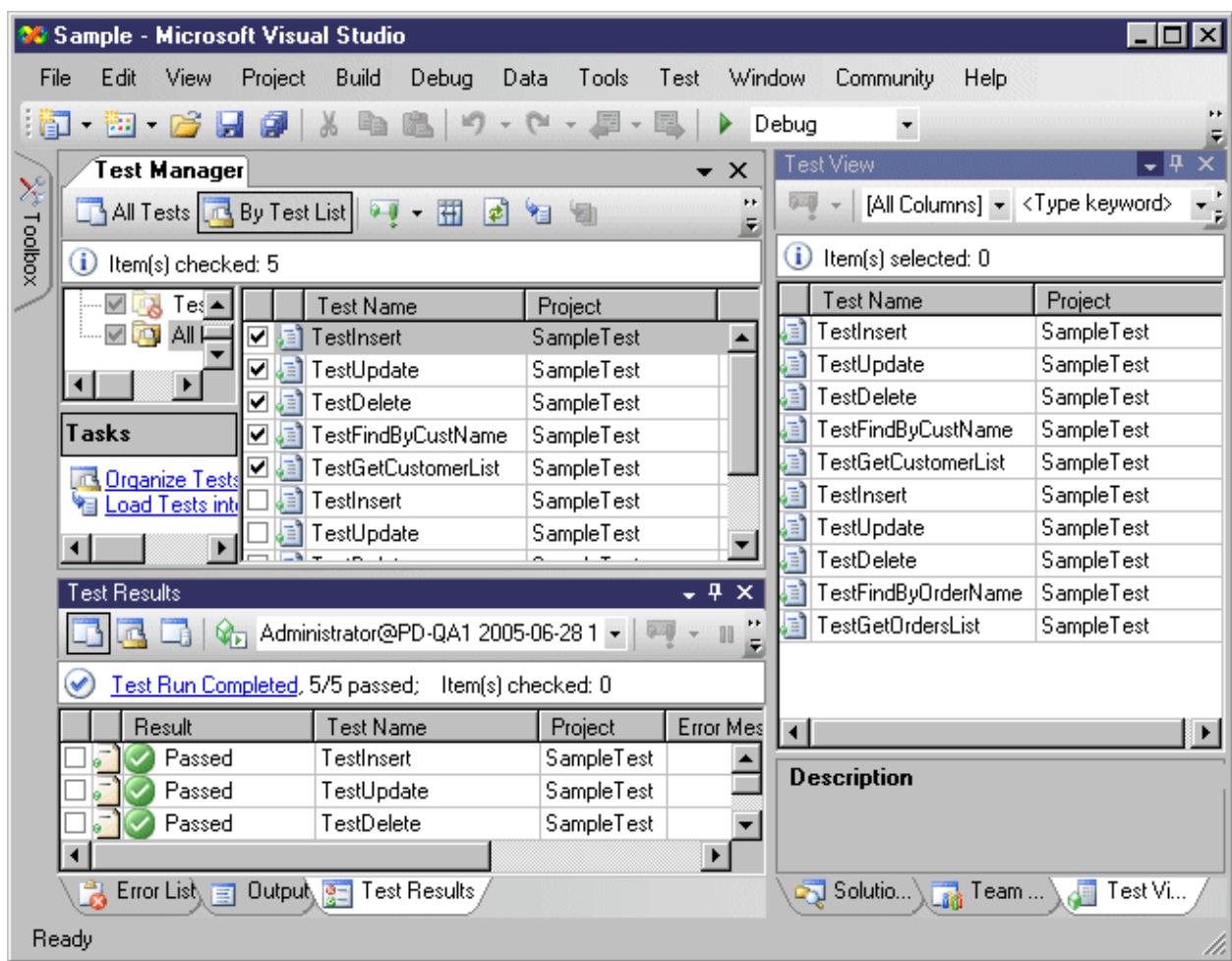
As you can see, the tests that were not run are marked with a grey circle, while those that were run successfully are colored green. If any tests had failed, they would be marked red.

- **Nunit Console** - The nunit-console.exe program is a text-based runner and can be used when you want to run all your tests and don't need a red/yellow/green indication of success or failure. It is useful for automation of tests and integration into other systems. It automatically saves its results in XML format, allowing you to produce reports or otherwise process the results.

## 2.10.4.2 Running Visual Studio Test System Unit Tests

The Visual Studio Team System Team Test tools offer a number of ways to run tests, both from the Visual Studio integrated development environment (IDE) and from a command prompt.

To run the tests in the IDE, use the Test Manager or Test View window. You can also rerun tests from the Test Result window. In the Test Manager or Test View window, select the tests you want to run, and then click *Run Tests* in the toolbar:



To run tests from the command line, open the Visual Studio command prompt, navigate to your solution folder directory, and run the MSTest.exe program.

## 2.10.5 Generating Windows or Smart Device Applications

The Composite UI Application Block (CAB) helps you build complex user interface applications that run in Windows. It provides an architecture and implementation that assists with building applications by using the common patterns found in line-of-business client applications. The current version of the Composite UI Application Block is aimed at Windows Forms applications that run with the Microsoft .NET Framework 2.0.

PowerDesigner supports development for Windows and Smart Device application development through extension files for your C# v2.0 or Visual Basic 2005 OOM.

You can quickly build Windows and Smart Device applications without writing repetitive code, by using PowerDesigner to generate persistent classes, DAL, BLL and UI files based on CAB according to your NHibernate or ADO.NET persistent framework.

To enable the Windows or Smart Device extensions in your model, select **Model > Extensions**, click the **Attach an Extension** tool, select the Windows Application or Smart Device Application file on the **User Interface** tab), and click **OK** to attach it.

### Note

You will also need to add the ADO.NET or Nhibernate persistence management extension in order to generate your application files.

Using this Windows application, you can test persistent objects with your own data. You can also improve the generated files and change the layout as you like in Visual Studio.

## 2.10.5.1 Specifying an Image Library

Your forms will probably use some images as icons. PowerDesigner provides a default image library, which it uses by default for Windows applications. You can also specify your own image library.

### Procedure

1. Open the model property sheet, and click the *Window Application* tab.
2. Specify a path to your image library, and then click *OK* to return to the model.

## 2.10.5.2 Controlling the Data Grid View

You can specify the length of your ADO.NET CF datagrid views.

### Procedure

1. Open the model property sheet, and click the *Smart Device Application* tab.
2. Specify the number of rows, and then click *OK* to return to the model.

## 2.10.5.3 Defining Attributes Display Options

You can define attribute-level options for presentation style.

### Procedure

1. Open the attribute property sheet, and click the *Windows Application* tab.

- 
- Set the appropriate options and then click **OK**

## Results

The following options are available:

| Option                       | Description  |
|------------------------------|--|
| Control Type                 | You can choose TextBox or ComboBox as input control  |
| Display as foreign key label | Specifies to display the current attribute as a foreign key label in combo boxes, instead of the foreign key. For example, select this option for the product name attribute to use it as foreign key label instead of the product id. |

### 2.10.5.4 Defining Attribute Validation Rules and Default Values

PowerDesigner provides validation and default values for the edit boxes in the Create, Find, ListView, and DetailView forms.

#### Procedure

- Open the attribute property sheet, and click the *Standard Checks* tab.
- [optional] Define minimum value and maximum values to control the value range.
- [optional] Define a default value. A string value must be enclosed in quotes. You can also define the Initial value in the *Details* tab.
- [optional] Define a list of values. PowerDesigner will automatically generate a combo box that includes these values.
- Click **OK** to return to the model.

### 2.10.5.5 Generating Code for a Windows Application

Before generating code for Windows Application, you need to install CAB (available from the Microsoft Web site).

#### Context

Check for the following required files in your installation directory:

- Microsoft.Practices.CompositeUI.dll
- Microsoft.Practices.CompositeUI.WinForms.dll
- Microsoft.Practices.ObjectBuilder.dll

## Procedure

1. Select **Tools > Check Model** to verify if there are errors or warnings in the model. If there are errors, fix them before continuing with code generation.
2. Select **Tools > General Options**, and click the Variables node.
3. Add a variable CAB\_PATH with the value of your installation directory
4. Select **Language > Generate C# 2.0 Code or Generate Visual Basic 2005 Code** to open the Generation dialog box.
5. Specify a target directory, and then click OK to begin generation.

The following files will be generated:

- Domain files (persistent classes, mapping files)
- DAL files (database connection file and data access files)
- A solution file and project files
- A login dialog
- Forms for persistent classes
- Controller files

## Results

For each persistent class, PowerDesigner generates:

- A find dialog for searching objects
- A list view form for displaying find results
- A detail view form for displaying object's detailed information
- A create view form for creating new objects

You can deploy or edit a Windows application in an IDE, such as Visual Studio .NET 2005.

## 2.10.5.6 Generating Code for a Smart Device Application

PowerDesigner can generate code for smart device applications.

### Context

Before generating a smart device application, you must:

- Ensure that you have attached the ADO.NET Compact Framework (and, if you want to generate unit tests, UnitTest.NET Compact Framework) extensions (see [Generating Windows or Smart Device Applications \[page 541\]](#)).
- Set any appropriate model properties on the ADO.NET CF and Application tabs, including a functioning connection string (see [ADO.NET and ADO.NET CF Options \[page 511\]](#)).
- Specify appropriate values for the following variables (select ► *Tools* ► *General Options* ▾, and click the Variables category):
  - CFUT\_HOME – if using Microsoft Mobile Client Software Factory CFUnitTester
  - ASANET\_HOME – if using Sybase ASA. Specifies the location of iAnywhere.Data.AsaClient.dll.
  - SQLSERVERMOBILENET\_HOME – if using Microsoft SQL Server Mobile Edition. Specifies the location of System.Data.SqlServerCe.dll
  - ULTRALITENETCE\_HOME – if using Sybase UltraLite. Specifies the location of ulnet10.dll
  - ULTRALITENET\_HOME – if using Sybase UltraLite. Specifies the location of iAnywhere.Data.UltraLite.dll and en\iAnywhere.Data.UltraLite.resources.dll

### Procedure

1. Select ► *Tools* ► *Check Model* ▾ to verify that there are no errors in the model. If there are errors, fix them before continuing with code generation.
2. Select ► *Language* ► *Generate C#2 Code or Generate Visual Basic 2005* ▾ to open the Generation dialog box.
3. Specify the root directory where you want to generate the code and then click the *Options* tab.
4. Specify any appropriate options and then click *OK* to generate code immediately or *Apply* and then *Cancel* to save your changes for later.
5. Compile your generated code in Visual Studio, and deploy the start up project, i.e. the <model>Test project or User Interface project. Then, deploy the SystemFramework project separately with the database file and required DLLs (such as ulnet10.dll for UltraLite support).

If you have generated and deployed the user interface projects to the device, you can run them and test the application by inputting some data. If you have generated for 'Microsoft Mobile Client Software Factory', you can run the unit tests by clicking GuiTestRunner.exe in the deployment folder in the device. The exe file and its references can be copied from the Microsoft Mobile Client Software Factory installation folder.

# Important Disclaimers and Legal Information

## Coding Samples

Any software coding and/or code lines / strings ("Code") included in this documentation are only examples and are not intended to be used in a productive system environment. The Code is only intended to better explain and visualize the syntax and phrasing rules of certain coding. SAP does not warrant the correctness and completeness of the Code given herein, and SAP shall not be liable for errors or damages caused by the usage of the Code, unless damages were caused by SAP intentionally or by SAP's gross negligence.

## Gender-Neutral Language

As far as possible, SAP documentation is gender neutral. Depending on the context, the reader is addressed directly with "you", or a gender-neutral noun (such as "sales person" or "working days") is used. If when referring to members of both sexes, however, the third-person singular cannot be avoided or a gender-neutral noun does not exist, SAP reserves the right to use the masculine form of the noun and pronoun. This is to ensure that the documentation remains comprehensible.

## Internet Hyperlinks

The SAP documentation may contain hyperlinks to the Internet. These hyperlinks are intended to serve as a hint about where to find related information. SAP does not warrant the availability and correctness of this related information or the ability of this information to serve a particular purpose. SAP shall not be liable for any damages caused by the use of related information unless damages have been caused by SAP's gross negligence or willful misconduct. All links are categorized for transparency (see: <https://help.sap.com/viewer/disclaimer>).





[go.sap.com/registration/  
contact.html](https://go.sap.com/registration/contact.html)



© 2018 SAP SE or an SAP affiliate company. All rights reserved.  
No part of this publication may be reproduced or transmitted in any  
form or for any purpose without the express permission of SAP SE  
or an SAP affiliate company. The information contained herein may  
be changed without prior notice.

Some software products marketed by SAP SE and its distributors  
contain proprietary software components of other software vendors.  
National product specifications may vary.

These materials are provided by SAP SE or an SAP affiliate company  
for informational purposes only, without representation or warranty  
of any kind, and SAP or its affiliated companies shall not be liable for  
errors or omissions with respect to the materials. The only  
warranties for SAP or SAP affiliate company products and services  
are those that are set forth in the express warranty statements  
accompanying such products and services, if any. Nothing herein  
should be construed as constituting an additional warranty.

SAP and other SAP products and services mentioned herein as well  
as their respective logos are trademarks or registered trademarks of  
SAP SE (or an SAP affiliate company) in Germany and other  
countries. All other product and service names mentioned are the  
trademarks of their respective companies.

Please see <https://www.sap.com/corporate/en/legal/copyright.html>  
for additional trademark information and notices.