



Object-Oriented Modeling

PowerDesigner® 16.0

Windows

DOCUMENT ID: DC38086-01-1600-01

LAST REVISED: July 2011

Copyright © 2011 by Sybase, Inc. All rights reserved.

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

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor. Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

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

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

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

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

All other company and product names used herein may be trademarks or registered trademarks of the respective companies with which they are associated.

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

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

Contents

PART I: Building OOMs	1
CHAPTER 1: Getting Started with Object-Oriented Modeling	3
Creating an OOM	5
OOM Properties	7
Previewing Object Code	8
Customizing Object Creation Scripts	10
Customizing your Modeling Environment	11
Setting OOM Model Options	11
Setting OOM Display Preferences	13
Viewing and Editing the Object Language Definition	
File	13
Changing the Object Language	13
Extending your Modeling Environment	14
Linking Objects with Traceability Links	15
CHAPTER 2: Use Case Diagrams	17
Use Case Diagram Objects	18
Use Cases (OOM)	18
Creating a Use Case	19
Use Case Properties	19
Actors (OOM)	20
Creating an Actor	22
Actor Properties	22
Reusing Actors	24
Use Case Associations (OOM)	24
Creating a Use Case Association	25
Use Case Association Properties	25

CHAPTER 3: Structural Diagrams	27
Class Diagrams	27
Class Diagram Objects	28
Composite Structure Diagrams	29
Composite Structure Diagram Objects	30
Object Diagrams	31
Object Diagram Objects	32
Package Diagrams	33
Package Diagram Objects	34
Classes (OOM)	34
Creating a Class	35
Class Properties	35
Creating Java BeanInfo Classes	39
Creating a Java BeanInfo Class from the Language Menu	41
Creating a Java BeanInfo Class from the Class Contextual Menu	42
Generic Types and Methods	42
Creating Generic Types	42
Creating Generic Methods	43
Creating a Specialized Classifier	43
Creating a Bound Classifier	45
Generic Type Example	45
Composite and Inner Classifiers	46
Creating Inner Classifiers	46
Creating a Composite Classifier Diagram	47
Attaching a Classifier to a Data Type or a Return Type	47
Attaching a Classifier	48
Viewing the Migrated Attributes of a Class	49
Packages (OOM)	50
OOM Package Properties	51
Defining the Diagram Type of a New Package	52

Interfaces (OOM)	53
Creating an Interface	53
Interface Properties	53
Objects (OOM)	56
Creating an Object	57
Object Properties	58
Linking a Classifier to an Object	59
Parts (OOM)	60
Creating a Part	60
Part Properties	61
Ports (OOM)	62
Creating a Port	63
Port Properties	63
Redefining Parent Ports	64
Attributes (OOM)	65
Creating an Attribute	66
Attribute Properties	66
Setting Data Profiling Constraints	70
Creating Data Formats For Reuse	70
Specifying Advanced Constraints	71
Adding Getter and Setter Operations to a Classifier	71
Copying an Attribute to a Class, Interface, or Identifier	72
Overriding an Attribute in PowerBuilder	74
Adding an Inherited Attribute to a Class	74
Identifiers (OOM)	75
Creating an Identifier	75
Creating a primary identifier when you create the class attributes	76
Defining the Primary Identifier from the List of Identifiers	76
Identifier Properties	77
Adding Attributes to an Identifier	77
Operations (OOM)	78
Creating an Operation	79

Creating a User-Defined Operation	79
Creating a Standard Operation	80
Inheriting and Overriding Operations from Parent Classifiers	82
Creating an Implementation Operation	82
Copying an Operation to Another Class	83
Operation Properties	84
Associations (OOM)	88
Creating an Association	89
Association Properties	90
Association Implementation	92
Understanding the Generated Code	94
Creating an Association Class	95
Migrating Association Roles in a Class Diagram	96
Migrating Navigable Roles	97
Rebuilding Data Type Links	97
Linking an Association to an Instance Link	98
Generalizations (OOM)	98
Creating a Generalization	100
Generalization Properties	100
Dependencies (OOM)	101
Creating a Dependency	103
Dependency Properties	103
Realizations (OOM)	105
Creating a Realization	105
Realization Properties	105
Require Links (OOM)	106
Creating a Require Link	107
Require Link Properties	107
Assembly Connectors (OOM)	107
Creating an Assembly Connector	108
Assembly Connector Properties	108
Delegation Connectors (OOM)	109
Creating a Delegation Connector	110
Delegation Connector Properties	110

Annotations (OOM)	111
Attaching an Annotation to a Model Object	111
Creating a New Annotation Type	113
Using the Annotation Editor	116
Instance Links (OOM)	116
Creating an Instance Link	119
Instance Link Properties	120
Domains (OOM)	120
Creating a Domain	121
Domain Properties	121
Updating Attributes Using a Domain in an OOM	124
CHAPTER 4: Dynamic Diagrams	125
Communication Diagrams	125
Communication Diagram Objects	127
Sequence Diagrams	127
Sequence Diagram Objects	130
Activity Diagrams	131
Activity Diagram Objects	133
Statechart Diagrams	133
Defining a Default Classifier in a Statechart Diagram	135
Statechart Diagram Objects	135
Interaction Overview Diagrams	136
Interaction Overview Diagram Objects	137
Messages (OOM)	137
Creating a Message	139
Message Properties	139
Creating Create and Destroy Messages in a Sequence Diagram	143
Creating Create Messages	143
Creating Destroy Messages	144
Creating a Recursive Message in a Sequence Diagram	145

Creating a Recursive Message Without Activation	146
Creating a Recursive Message with Activation	147
Messages and Gates	147
Sequence Numbers	149
Moving Sequence Numbers	149
Inserting Sequence Numbers	151
Increasing Sequence Numbers in a Communication Diagram	151
Decreasing Sequence Numbers in a Communication Diagram	151
Activations (OOM)	151
Creating an Activation	152
Creating Activations with Procedure Call Messages	152
Creating an Activation from a Diagram	152
Attaching a Message to an Activation	152
Detaching a Message from an Activation	153
Overlapping Activations	153
Moving an Activation	154
Resizing an Activation	154
Interaction References and Interaction Activities (OOM)	155
Creating an Interaction Reference	155
Creating an Interaction Activity	156
Interaction Reference and Interaction Activity Properties	156
Manipulating Interaction References	156
Interaction Fragments (OOM)	157
Creating an Interaction Fragment	157
Interaction Fragment Properties	157
Manipulating Interaction Fragments	159
Activities (OOM)	160
Creating an Activity	161

Activity Properties	162
Specifying Activity Parameters	164
Specifying Action Types	165
Example: Using the Call Action Type	166
Example: Reading and Writing Variables	169
Decomposed Activities and Sub-Activities	170
Converting an Activity Diagram to a Decomposed Activity	172
Organization Units (OOM)	173
Creating an Organization Unit	174
Creating Organization Units with the Swimlane Tool	174
Organization Unit Properties	174
Attaching Activities to Organization Units	175
Displaying a Committee Activity	176
Managing Swimlanes and Pools	176
Moving, Copying and Pasting Swimlanes	177
Grouping and Ungrouping Swimlanes	178
Creating Links Between Pools of Swimlanes	180
Changing the Orientation of Swimlanes	181
Resizing Swimlanes	182
Changing the Format of a Swimlane	182
Starts (OOM)	182
Creating a Start	183
Start Properties	183
Ends (OOM)	183
Creating an End	184
End Properties	184
Decisions (OOM)	185
Creating a Decision	187
Decision Properties	187
Synchronizations (OOM)	188
Creating a Synchronization	189
Synchronization Properties	189
Flows (OOM)	190

Creating a Flow	191
Flow Properties	191
Object Nodes (OOM)	192
Creating an Object Node	193
Object Node Properties	193
States (OOM)	194
Creating a State	195
State Properties	195
Decomposed States and Sub-states	197
Converting a Statechart Diagram to a Decomposed State	199
Transitions (OOM)	200
Creating a Transition	200
Transition Properties	201
Events (OOM)	202
Creating an Event	203
Event Properties	203
Defining Event Arguments	205
Actions (OOM)	205
Creating an Action	207
Action Properties	207
Junction Points (OOM)	209
Creating a Junction Point	209
Junction Point Properties	209
CHAPTER 5: Implementation Diagrams	211
Component Diagrams	211
Component Diagram Objects	212
Deployment Diagrams	213
Deployment Diagram Objects	214
Components (OOM)	214
Creating a Component	215
Using the Standard Component Wizard	215
Component Properties	216

Creating a Class Diagram for a Component	219
Deploying a Component to a Node	220
Nodes (OOM)	220
Creating a Node	221
Node Properties	221
Node Diagrams	222
Component Instances (OOM)	222
Creating a Component Instance	223
Component Instance Properties	224
Files (OOM)	224
Creating a File Object	225
File Object Properties	226
Node Associations (OOM)	227
Creating a Node Association	227
Node Association Properties	227
CHAPTER 6: Web Services	229
Defining Web Services Tools	229
Defining Web Services Targets	232
Defining Web Service Components	232
Web Service Component Properties	232
Creating a Web Service with the Wizard	235
Creating a Web Service from the Component Diagram	237
Defining Data Types for WSDL	238
WSDL Data Type Mappings	238
Selecting WSDL Data Types	238
Declaring Data Types in the WSDL	238
Web Service Implementation Class Properties	238
Managing Web Service Methods	239
Creating a Web Service Method	239
Web Service Method Properties	241
Implementing a Web Service Method in Java	241
Defining the Return Type of an Operation	241

- Defining the Parameters of an Operation242
- Implementing the Operation 244
- Implementing a Web Service Method in .NET246
- Defining Web Service Method Extended Attributes ...246
- Defining SOAP Data Types of the WSDL Schema 247
- Defining Web Service Component Instances248**
 - Web Service Tab of the Component Instance 249
 - WSDL Tab of the Component Instance 250
 - Using Node Properties 250
- Generating Web Services for Java250**
 - Generating JAXM Web Services 251
 - Generating JAX-RPC Web Services 252
 - Generating Stateless Session Bean Web Services ...253
 - Generating AXIS RPC Web Services 254
 - Generating AXIS EJB Web Services 255
 - Generating Java Web Services (JWS) 256
 - Testing Web Services for Java 256
- Generating Web Services for .NET256**
 - Defining Web Services Generation Options in .NET .. 256
 - Defining Web Service Generation Tasks in .NET 257
 - Generating Web Services in .NET 257
 - Generating a .NET Proxy Class for a Web Service ... 258
 - Define the WSDL Variable 258
 - Generate the Client Proxy Classes 258
 - Deploying Web Services in .NET 259
 - Testing Web Services for .NET 259
- Generating Web Services for Sybase WorkSpace260**
 - Creating a Java or EJB Web Service for Sybase WorkSpace 260
 - Defining the Java Class Package 261
 - Generating the Java or EJB Web Service for Sybase WorkSpace 262
 - Understanding the .svc_java or .svc_ejb File 262
- Importing WSDL Files263**
 - Browsing WSDL Files from UDDI 265

PART II: Working with OOMs	267
CHAPTER 7: Generating and Reverse Engineering OO Source Files	269
Generating OO Source Files from an OOM	269
Working with Generation Targets	272
Defining the Source Code Package	272
Reverse Engineering OO Source Files into an OOM	273
Reverse Engineering OO Files into a New OOM	273
Reverse Engineering Encoding Format	274
Reverse Engineering into an Existing OOM	275
Synchronizing a Model with Generated Files	276
CHAPTER 8: Generating Other Models from an OOM	279
Mapping OOM Objects to other Model Objects	280
Managing Object Persistence During OOM to CDM	
Generation	281
Defining Object Persistence in the OOM	281
Managing Complex Data Type Persistence	282
Managing Multiplicity for Complex Persistent Data Types	283
Managing Object Persistence During OOM to PDM	
Generation	286
Defining Object Persistence in the OOM	286
Managing Complex Data Type Persistence	288
Managing Multiplicity for Complex Persistent Data Types	291
CHAPTER 9: Checking an OOM	295
Domain Checks	295
Data Source Checks	296

Package Checks	297
Actor/Use Case Checks	298
Class Checks	298
Identifier Checks	305
Interface Checks	305
Class/Interface Attribute Checks	308
Class/Interface Operation Checks	309
Realization Checks	311
Generalization Checks	311
Object Checks	312
Instance Link Checks	313
Message Checks	313
State Checks	314
State Action Checks	315
Event Checks	316
Junction Point Checks	317
Activity Checks	317
Decision Checks	318
Object Node Checks	319
Organization Unit Checks	320
Start/End Checks	321
Synchronization Checks	321
Transition and Flow Checks	322
Component Checks	323
Node Checks	324
Data Format Checks	325
Component Instance Checks	325
Interaction Reference Checks	326
Class Part Checks	327
Class/Component Port Checks	328
Class/component Assembly Connector Checks	329
Association Checks	330
Activity Input and Output Parameter Checks	330

CHAPTER 10: Importing a Rational Rose Model into an OOM	331
Importing Rational Rose Use Case Diagrams	332
Importing Rational Rose Class Diagrams	333
Importing Rational Rose Collaboration Diagrams	334
Importing Rational Rose Sequence Diagrams	335
Importing Rational Rose Statechart Diagrams	335
Importing Rational Rose Activity Diagrams	336
Importing Rational Rose Component Diagrams	337
Importing Rational Rose Deployment Diagrams	338
CHAPTER 11: Importing and Exporting an OOM in XMI Format	339
Importing XMI Files	339
Exporting XMI Files	339
PART III: Object Language Definition Reference	341
CHAPTER 12: Working with Java	343
Java Public Classes	343
Java Enumerated Types (Enums)	343
JavaDoc Comments	348
Defining Values for Javadoc Tags	351
Javadoc Comments Generation and Reverse Engineering	353
Java 5.0 Annotations	353
Java Strictfp Keyword	354
Enterprise Java Beans (EJBs) V2	355
Using EJB Types	356
EJB Properties	357
Creating an EJB with the Wizard	358
Defining Interfaces and Classes for EJBs	361

Defining Operations for EJBs	363
Adding an Operation to the Bean Class	364
Adding an Operation to an EJB Interface	364
Understanding Operation Synchronization	365
Understanding EJB Support in an OOM	366
Previewing the EJB Deployment Descriptor	369
Generating EJBs	370
What Kind of Generation to Use?	371
Understanding EJB Source and Persistence	373
Generating EJB Source Code and the Deployment Descriptor	374
Generating JARs	375
Reverse Engineering EJB Components	376
Enterprise Java Beans (EJBs) V3	377
Creating an EJB 3.0 with the Enterprise JavaBean Wizard	377
EJB 3.0 BeanClass Properties	381
EJB 3.0 Component Properties	382
Adding Further Interfaces and Classes to the EJB	382
EJB 3.0 Operation Properties	383
Java Servlets	384
Servlet Page of the Component	384
Defining Servlet Classes	385
Creating a Servlet with the Wizard	385
Understanding Servlet Initialization and Synchronization	386
Generating Servlets	387
Generating Servlet Web Deployment Descriptor	390
Generating WARs	390
Reverse Engineering Servlets	391
Java Server Pages (JSPs)	393
JSP Page of the Component	393
Defining File Objects for JSPs	393

Creating a JSP with the Wizard	394
Generating JSPs	395
Generating JSP Web Deployment Descriptor	395
Reverse Engineering JSPs	398
Generating Java Files	399
Reverse Engineering Java Code	403
Reverse Engineer Java Options Tab	405
Reverse Engineering Java Code Comments	407
CHAPTER 13: Working with the Eclipse Modeling Framework (EMF)	409
EMF Objects	409
EPackages	409
Eclasses, EEnums, and EDataTypes	409
EAnnotations	410
Eattributes and EEnumLiterals	410
EReferences	410
EOperations and EParameters	411
Generating EMF Files	411
Reverse Engineering EMF Files	412
CHAPTER 14: Working with IDL	413
IDL Objects	413
Generating for IDL	422
Reverse Engineering IDL Files	423
CHAPTER 15: Working with PowerBuilder	425
PowerBuilder Objects	425
Generating PowerBuilder Objects	428
Reverse Engineering PowerBuilder	429
Reverse Engineered Objects	429
Operation Reversed Header	430
Overriding Attributes	431

PowerBuilder Reverse Engineering Process432
 Reverse Engineering PowerBuilder Objects432
 Loading a PowerBuilder Library Model in the
 Workspace434

CHAPTER 16: Working with VB .NET437

Inheritance & Implementation437
Namespace437
Project437
Accessibility438
Classes, Interfaces, Structs, and Enumerations439
Module440
Custom Attributes441
Shadows441
Variables442
Property443
Method444
Constructor & Destructor446
Delegate446
Event447
Event Handler448
External Method448
Generating VB.NET Files449
Reverse Engineering VB .NET451
 Selecting VB .NET Reverse Engineering Options451
 Defining VB .NET Reverse Engineering Options
452
 VB .NET Reverse Engineering Preprocessing453
 VB .NET Supported Preprocessing Directives
454
 Defining a VB .NET Preprocessing Symbol454
 VB .NET Reverse Engineering with
 Preprocessing455
 Reverse Engineering VB .NET Files456

Working with ASP.NET	457
ASP Tab of the Component	457
Defining File Objects for ASP.NET	458
Creating an ASP.NET with the Wizard	458
Generating ASP.NET	460
CHAPTER 17: Working with Visual Basic 2005	463
Visual Basic 2005 Assemblies	463
Visual Basic 2005 Compilation Units	465
Partial Types	467
Visual Basic 2005 Namespaces	467
Visual Basic 2005 Classes	468
Visual Basic 2005 Interfaces	469
Visual Basic 2005 Structs	469
Visual Basic 2005 Delegates	470
Visual Basic 2005 Enums	471
Visual Basic 2005 Fields, Events, and Properties	472
Visual Basic 2005 Methods	475
Visual Basic 2005 Inheritance and Implementation	476
Visual Basic 2005 Custom Attributes	476
Generating Visual Basic 2005 Files	476
Reverse Engineering Visual Basic 2005 Code	478
Visual Basic Reverse Engineer Dialog Options Tab ...	479
Visual Basic Reverse Engineering Preprocessing	
Directives	480
Visual Basic Supported Preprocessing	
Directives	481
Defining a Visual Basic Preprocessing Symbol	
.....	481
CHAPTER 18: Working with C#	483
Inheritance & Implementation	483
Namespace	483
Project	483

Accessibility	484
Classes, Interfaces, Structs, and Enumerations	485
Custom Attributes	487
Fields	487
Property	488
Indexer	489
Method	491
Constructor & Destructor	493
Delegate	493
Event	494
Operator Method	494
Conversion Operator Method	494
Documentation Tags	495
Generating C# Files	496
Reverse Engineering C#	498
Selecting C# Reverse Engineering Options	499
Defining C# Reverse Engineering Options	500
C# Reverse Engineering Preprocessing	500
C# Supported Preprocessing Directives	501
Defining a C# Preprocessing Symbol	502
C# Reverse Engineering with Preprocessing	503
Reverse Engineering C# Files	503
 CHAPTER 19: Working with C# 2.0	 505
C# 2.0 Assemblies	505
C# 2.0 Compilation Units	507
Partial Types	508
C# 2.0 Namespaces	509
C# 2.0 Classes	510
C# 2.0 Interfaces	511
C# 2.0 Structs	511
C# 2.0 Delegates	512
C# 2.0 Enums	513
C# 2.0 Fields	514

C# 2.0 Methods	514
C# 2.0 Events, Indexers, and Properties	517
C# 2.0 Inheritance and Implementation	520
C# 2.0 Custom Attributes	520
Generating C# 2.0 Files	520
Reverse Engineering C# 2.0 Code	522
C# Reverse Engineer Dialog Options Tab	523
C# Reverse Engineering Preprocessing Directives ...	524
C# Supported Preprocessing Directives	524
Defining a C# Preprocessing Symbol	525
 CHAPTER 20: Working with XML	 527
Designing for XML	527
Generating for XML	531
Reverse-Engineering XML	532
 CHAPTER 21: Working with C++	 535
Designing for C++	535
Generating for C++	536
 CHAPTER 22: Object/Relational (O/R) Mapping	 539
Top-Down: Mapping Classes to Tables	539
Entity Class Transformation	541
Attribute Transformation	542
Value Type Transformation	543
Association Transformation	544
Association Class Transformation	546
Inheritance Transformation	547
Bottom-Up: Mapping Tables to Classes	550
Meet in the Middle: Manually Mapping Classes to Tables	551
Entity Class Mapping	552
Attribute Mapping	555

Primary Identifier Mapping	556
Association Mapping	559
One-to-One Association Mapping Strategy	560
One-to-Many Association Mapping Strategy	561
Many-to-Many Association Mapping Strategy	563
Defining Inheritance Mapping	564
Table Per Class Hierarchy Inheritance Mapping Strategy	564
Joined Subclass Inheritance Mapping Strategy	567
Table Per Class Inheritance Mapping Strategy	568

CHAPTER 23: Generating Persistent Objects for Java and JSF Pages571

Generating Hibernate Persistent Objects	571
Defining the Hibernate Default Options	571
Defining the Hibernate Database Configuration Parameters	572
Defining Hibernate Basic O/R Mappings	574
Defining Hibernate Class Mapping Options	574
Defining Primary Identifier Mappings	577
Defining Attribute Mappings	581
Hibernate Association Mappings	582
Defining Hibernate Association Mapping Options	582
Collection Management Options	584
Persistence Options	585
Mapping Collections of Value Types	586
Defining Hibernate Inheritance Mappings	587
Generating Code for Hibernate	587
Checking the Model	588
Defining Generation Options	588
Generating Code for Hibernate	589

Using the Generated Hibernate Code	590
Importing the Generated Project into Eclipse ...	590
Performing the Unit Tests	590
Running Unit Tests in Eclipse	591
Running Unit Tests with Ant	593
Generating EJB 3 Persistent Objects	594
Generating Entities for EJB 3.0	595
Defining EJB 3 Basic O/R Mapping	595
Defining Entity Mappings	595
Defining Embeddable Class Mapping	598
Defining EJB 3 Association Mappings	599
Mapping One-to-one Associations	599
Mapping One-to-many Associations	599
Mapping Many-to-many Associations	600
Defining EJB 3 Association Mapping Options	
.....	601
Defining EJB 3 Inheritance Mappings	601
Mapped Superclasses	602
Table Per Class Hierarchy Strategy	602
Joined Subclass Strategy	602
Applying Table Per Class Strategy	603
Defining EJB 3 Persistence Default Options	603
Defining EJB 3 Persistence Configuration	603
Checking the Model	605
Generating Code for EJB 3 Persistence	606
Defining the Environment Variable	606
Generate Code	607
Authoring in Dali Tools	609
Run Unit Tests	609
Generated File List	612
Generating JavaServer Faces (JSF) for Hibernate	613
Defining Global Options	613
Defining Attribute Options	616
Derived Attributes	618

Attribute Validation Rules and Default Values	618
Defining Master-Detail Pages	619
Generating PageFlow Diagrams	621
Generating a class level PageFlow diagram	621
Generating a Package Level PageFlow Diagram:	622
Generating a Model Level PageFlow Diagram	622
Modifying Default High Level PageFlow Diagram	623
Installing JSF Runtime Jar Files	627
Installing the JSF Reference Implementation	627
Installing Apache My Faces	627
Configuring for JSF Generation	628
Generating JSF Pages	628
Testing JSF Pages	629
Testing JSF Pages with Eclipse WTP	629
Testing JSF Pages with Apache Tomcat	629

CHAPTER 24: Generating .NET 2.0 Persistent Objects and Windows Applications631

Generating ADO.NET and ADO.NET CF Persistent Objects	633
ADO.NET and ADO.NET CF Options	633
Class Mappings	634
Primary Identifier Mappings	636
Attribute Mappings	637
Defining Association Mappings	638
Defining Inheritance Mappings	640
Generating Code for ADO.NET or ADO.NET CF	640
Generating NHibernate Persistent Objects	641
NHibernate Options	642

Defining Class Mappings	643
Primary Identifier Mappings	646
Attribute Mappings	649
Defining Association Mappings	651
Defining NHibernate Collection Options	652
Defining NHibernate Persistence Options	653
Defining NHibernate Collection Container Type	654
Defining Inheritance Mappings	654
Generating Code for NHibernate	655
Configuring Connection Strings	656
Configuring a Connection String from the ADO.NET or ADO.NET CF Tab	657
Configuring a Connection String from the NHibernate Tab	657
OLEDB Connection String Parameters	657
ODBC Connection String Parameters	658
Microsoft SQL Server and Microsoft SQL Server Mobile Edition Connection String Parameters	658
Oracle Connection String Parameters	658
Generating Code for Unit Testing	659
Running NUnit Unit Tests	661
Running Visual Studio Test System Unit Tests	662
Running Tests in Visual Studio.NET 2005 IDE	663
Running Tests from the Command Line	663
Generating Windows or Smart Device Applications	664
Specifying an Image Library	664
Controlling the Data Grid View	664
Defining Attributes Display Options	664
Defining Attribute Validation Rules and Default Values	665
Generating Code for a Windows Application	665
Generating Code for a Smart Device Application	666
Deploying Code to a Smart Device	667

Contents

	Testing the Application on the Device	667
Index		669

PART I

Building OOMs

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

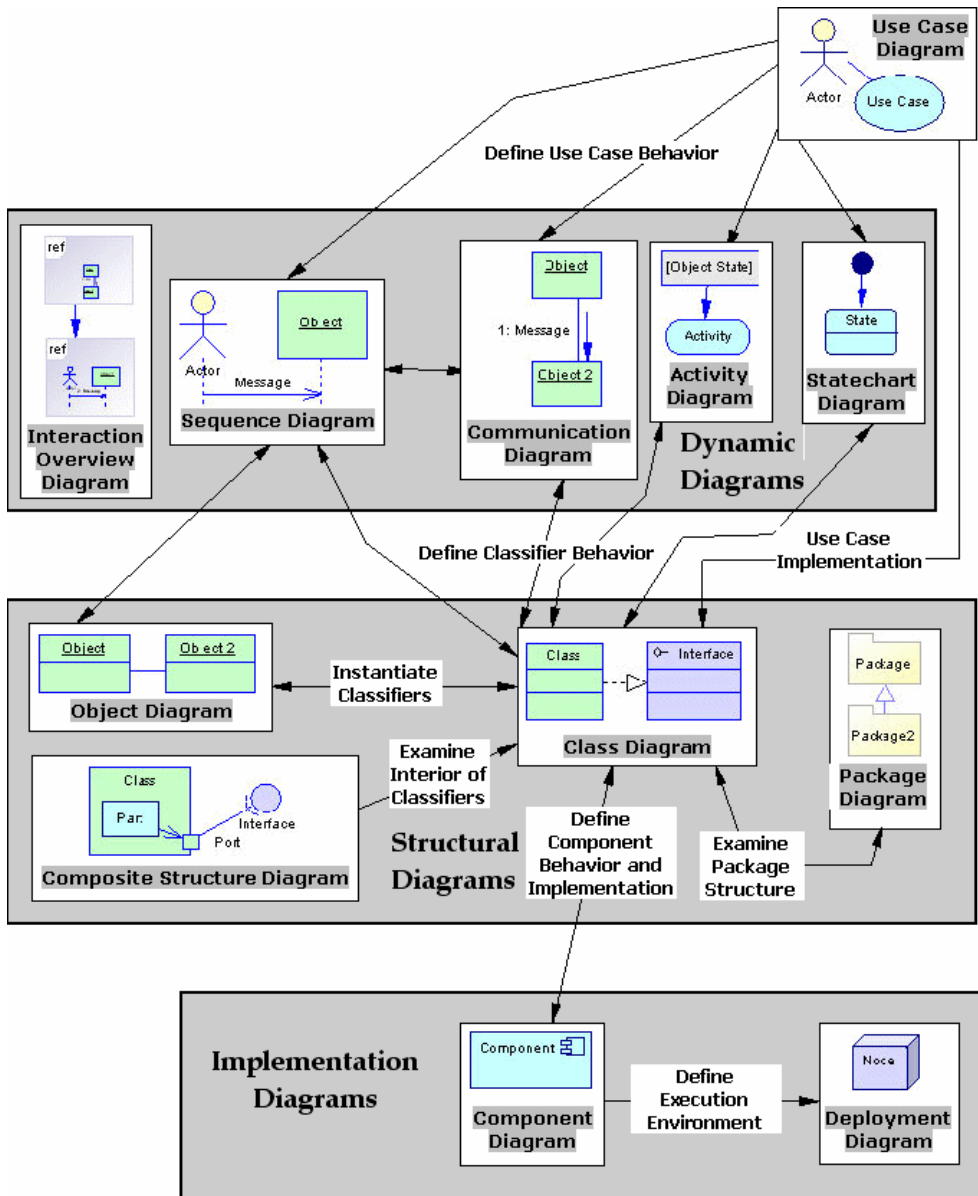
Getting Started with Object-Oriented Modeling

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.

PowerDesigner® supports the following UML diagrams:

- Use case diagram (📄) - see *Chapter 2, Use Case Diagrams* on page 17
- Structural Diagrams:
 - Class diagram (📄) - see *Class Diagrams* on page 27
 - Composite structure diagram (📄) - see *Composite Structure Diagrams* on page 29
 - Object diagram (📄) - see *Object Diagrams* on page 31
 - Package diagram (📄) - see *Package Diagrams* on page 33
- Dynamic Diagrams:
 - Communication diagram (📄) - see *Communication Diagrams* on page 125
 - Sequence diagram (📄) - see *Sequence Diagrams* on page 127
 - Activity diagram (📄) - see *Activity Diagrams* on page 131
 - Statechart diagram (📄) - see *Statechart Diagrams* on page 133
 - Interaction overview diagram (📄) - see *Interaction Overview Diagrams* on page 136
- Implementation Diagrams:
 - Component diagram (📄) - see *Component Diagrams* on page 211
 - Deployment diagram (📄) - see *Deployment Diagrams* on page 213

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

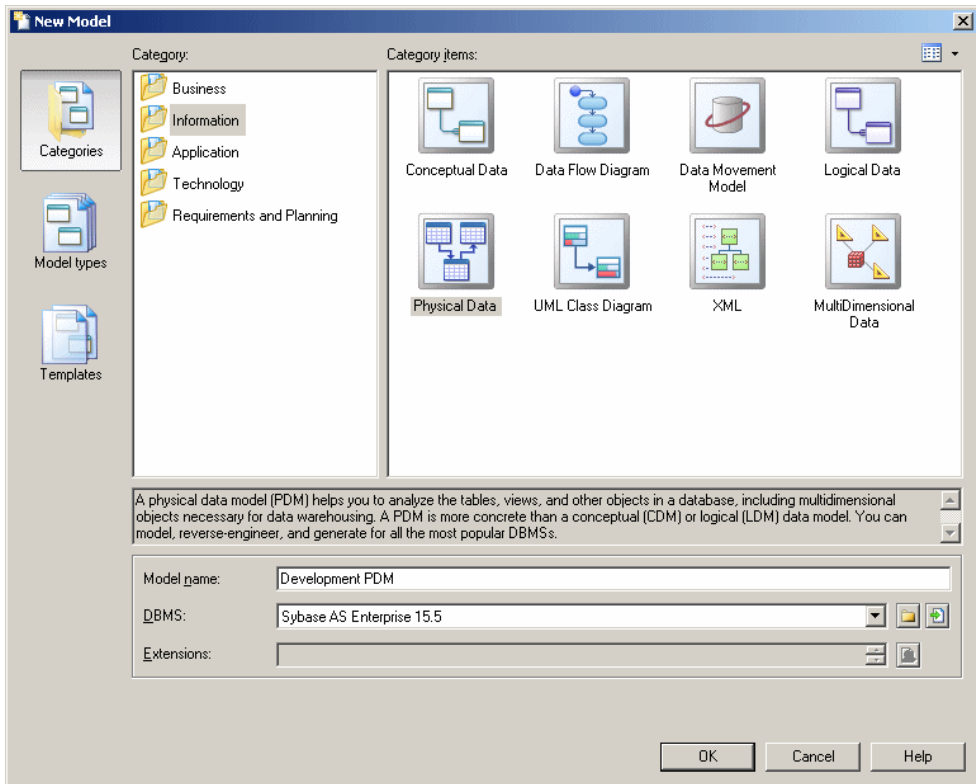
Creating an OOM

You create a new object-oriented model by selecting **File > New Model**.

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* on page 273).

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.



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 .

Note: Sample OOMs are available in the Example Directory.

OOM Properties

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

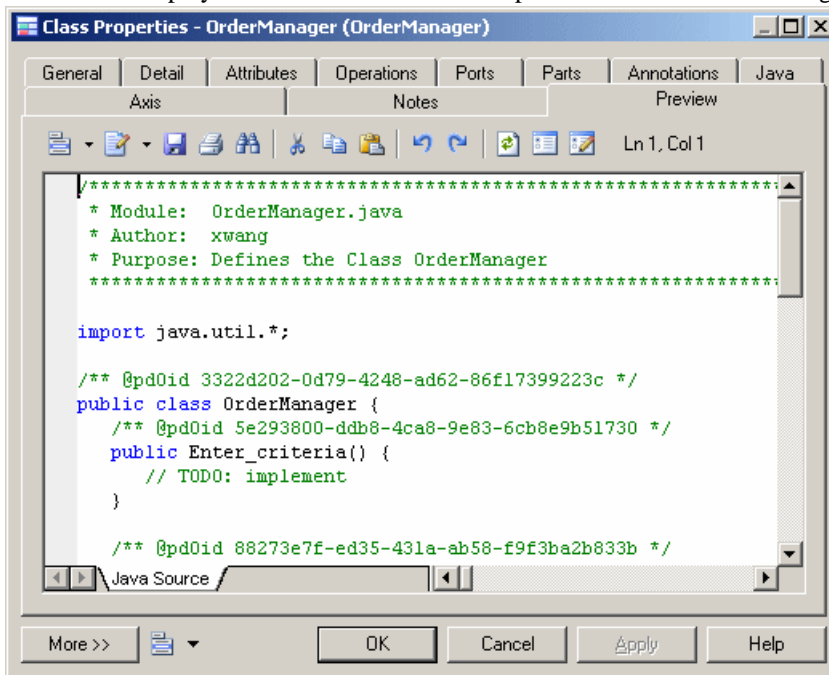
Each object-oriented model has the following model properties:

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 Code 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	Specifies the version of the model. You can use this box to display the repository version or a user defined version of the model. This parameter is defined in the display preferences of the Title node.
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.

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 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>Editor Menu [Shift+F11] - Contains the following commands:</p> <ul style="list-style-type: none"> • New [Ctrl+N] - Reinitializes the field by removing all the existing content. • Open... [Ctrl+O] - Replaces the content of the field with the content of the selected file. • Insert... [Ctrl+I] - Inserts the content of the selected file at the cursor. • Save [Ctrl+S] - Saves the content of the field to the specified file. • Save As... - Saves the content of the field to a new file. • Select All [Ctrl+A] - Selects all the content of the field. • Find... [Ctrl+F] - Opens a dialog to search for text in the field. • Find Next... [F3] - Finds the next occurrence of the searched for text. • Find Previous... [Shift+F3] - Finds the previous occurrence of the searched for text. • Replace... [Ctrl+H] - Opens a dialog to replace text in the field. • Go To Line... [Ctrl+G] - Opens a dialog to go to the specified line. • Toggle Bookmark [Ctrl+F2] Inserts or removes a bookmark (a blue box) at the cursor position. Note that bookmarks are not printable and are lost if you refresh the tab, or use the Show Generation Options tool • Next Bookmark [F2] - Jumps to the next bookmark. • Previous Bookmark [Shift+F2] - Jumps to the previous bookmark.
	<p>Edit With [Ctrl+E] - Opens the previewed code in an external editor. Click the down arrow to select a particular editor or Choose Program to specify a new editor. Editors specified here are added to the list of editors available at Tools > General Options > Editors.</p>
	<p>Save [Ctrl+S] - Saves the content of the field to the specified file.</p>
	<p>Print [Ctrl+P] - Prints the content of the field.</p>
	<p>Find [Ctrl+F] - Opens a dialog to search for text.</p>
	<p>Cut [Ctrl+X], Copy [Ctrl+C], and Paste [Ctrl+V] - Perform the standard clipboard actions.</p>
	<p>Undo [Ctrl+Z] and Redo [Ctrl+Y] - Move backward or forward through edits.</p>
	<p>Refresh [F5] - 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 Enable Trace Mode option, and click OK to return to your model. You may need to click the Refresh tool to display the templates.</p>

Tools	Description
	Select Generation Targets [Ctrl+F6] - 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 > Extension Files > Extending Generation and Creating Separate Generation Targets</i> .
	Show Generation Options [Ctrl+W] - 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.

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
	[Shift+F11] Open Editor Contextual menu
	[Ctrl+E] Edit With - Opens your default editor.
	Import Folder - [Imports sub-tab] Opens a selection window to select packages to import to the cursor position, prefixed by the keyword 'import'.
	Import Classifier - [Imports 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

Format code	Format of variable value in script
.U	Uppercase characters
.c	Upper-case first letter and lower-case next letters
.n	Maximum length where n is the number of characters
.nJ	Justifies to fixed length where n is the number of characters

You embed formatting options in variable syntax as follows:

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

Customizing your Modeling Environment

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

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 Modeling</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, All properties appear, but you can select to display only Name/Code to reduce the size of your model.</p> <hr/> <p>Note: 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 "Settings category" in the Resource Files and the Public Metamodel chapter of the <i>Customizing and Extending PowerDesigner</i> manual.</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"> • Data type – data type, length, and precision • Check – check parameters, such as minimum and maximum values • Rules – business rules
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 <<implement>> stereotype is applied to the methods.</p>
Interface/Class: Class attribute default visibility	<p>Specifies the default visibility of class attributes.</p>

Note: For information about specifying naming conventions for your model objects, see *Core Features Guide > The PowerDesigner Interface > Objects > Object Properties > Naming Conventions*.

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** from the contextual menu.

For detailed information about customizing and controlling the attributes and collections displayed on object symbols, see *Core Features Guide > The PowerDesigner Interface > Diagrams, Matrices, and Symbols > Display Preferences*.

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.

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

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*.

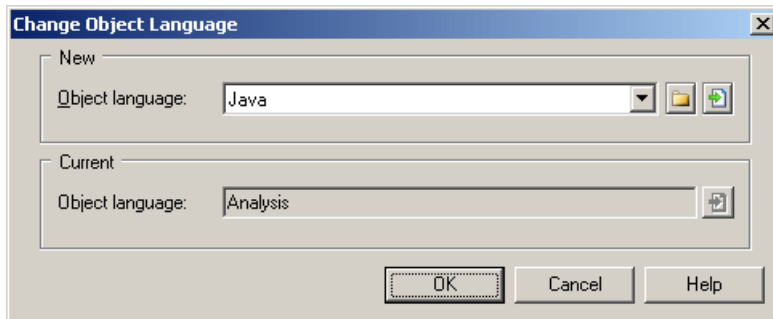
Note: Some resource files are delivered with "Not Certified" in their names. Sybase® will perform all possible validation checks, however Sybase does not maintain specific environments to fully certify these resource files. Sybase will support the definition by accepting bug reports and will provide fixes as per standard policy, with the exception that there will be no final environmental validation of the fix. Users are invited to assist Sybase by testing fixes of the definition provided by Sybase and report any continuing inconsistencies.

Changing the Object Language

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

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.

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.

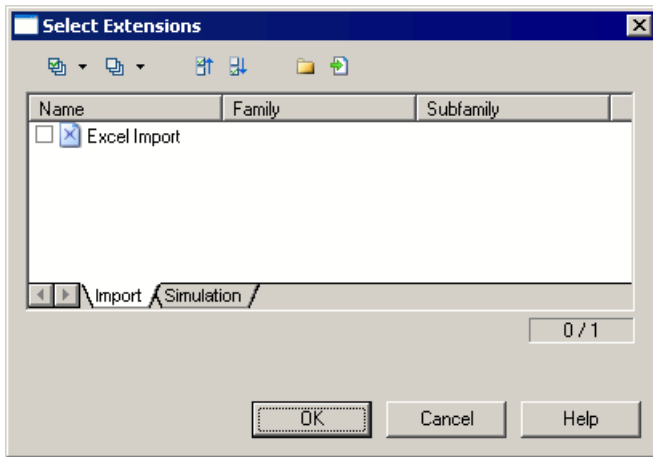
4. Click **OK** to return to the model.

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 extension 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 **Import 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 get started extending objects, see *Core Features Guide > The PowerDesigner Interface > Objects > Extending Objects*. For detailed information about working with extensions, see *Customizing and Extending PowerDesigner > Extension Files*.

Linking Objects with Traceability Links

You can create traceability links to show any kind of relationship between two model objects (including between objects in different models) via the **Traceability Links** tab of the object's property sheet. These links are used for documentation purposes only, and are not interpreted or checked by PowerDesigner.

For more information about traceability links, see *Core Features Guide > Linking and Synchronizing Models > Getting Started with Linking and Syncing > Creating Traceability Links*.

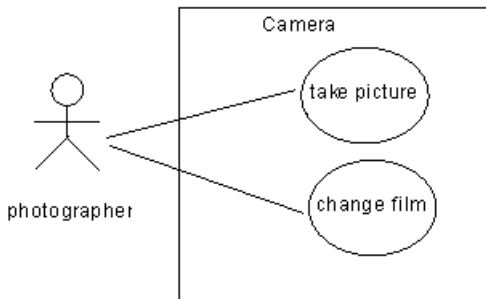
A *use case diagram* is a UML diagram that provides a graphical view of the requirements of your system, and helps you identify how users interact with it.

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.



Use Case Diagram Objects

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

Object	Tool	Symbol	Description
Actor		 Client	Used to represent an external person, process or something interacting with a system, sub-system or class. See <i>Actors (OOM)</i> on page 20.
Use case		 Purchase	Defines a piece of coherent behavior in a system, without revealing its internal structure. See <i>Use Cases (OOM)</i> on page 18.
Association			Communication path between an actor and a use case that it participates in. See <i>Use Case Associations (OOM)</i> on page 24.
Generalization			A link between a general use case and a more specific use case that inherits from it and add features to it. See <i>Generalizations (OOM)</i> on page 98.
Dependency			Relationship between two modeling elements, in which a change to one element will affect the other element. See <i>Dependencies (OOM)</i> on page 101.

Use Cases (OOM)

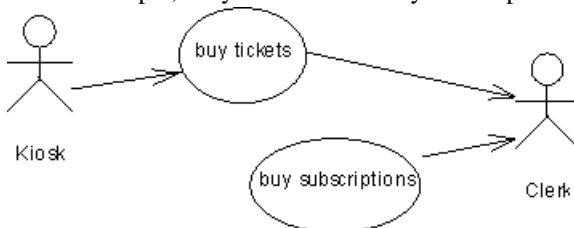
A *use case* is an interaction between a user and a system (or part of a system). It defines a discrete goal that a user wants to achieve with the system, without revealing the system's internal structure.

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.



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 > The PowerDesigner Interface > Objects*.

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 Code 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. 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".

Property	Description
Extension Points	Specifies a textual description of actions that extend the normal sequence of actions. Extensions are usually introduced with an "ifthen" statement. 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 > The PowerDesigner Interface > Diagrams, Matrices, and Symbols > Diagrams > Specifying Diagrams as Related Diagrams*.

Actors (OOM)

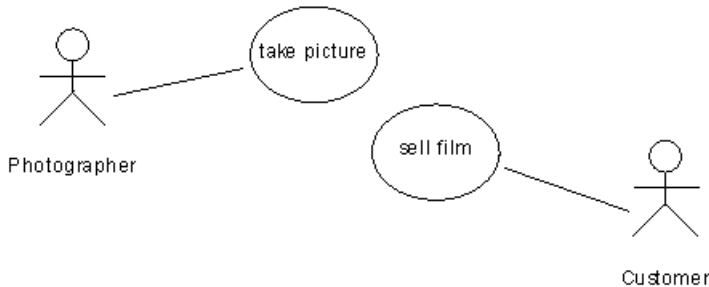
An *actor* is an outside user or set of users that interact with a system. Actors can be humans or other external systems. For example, actors in a computer network system may include a system administrator, a database administrator and users. Actors are typically those entities whose behavior you cannot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

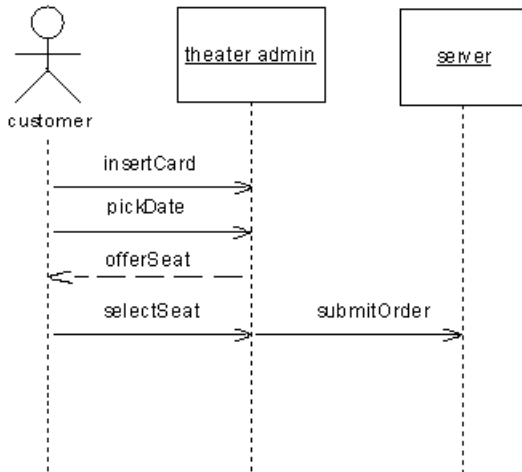
In a communication diagram, an actor may be connected to an object by an instance link, or may send or receive messages.



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.



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 > The PowerDesigner Interface > Objects*.

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 Code 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

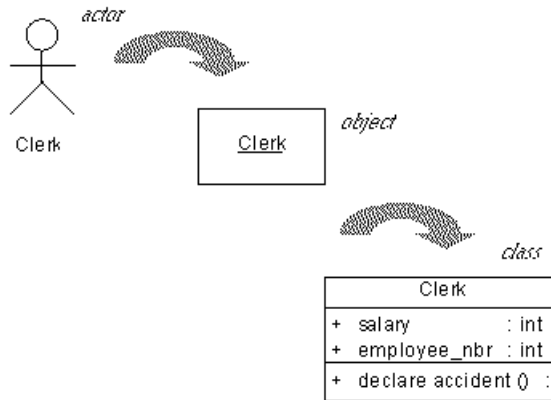
An actor can be a human being (person, partner) or a machine, or process (automated system). When analyzing what an actor must do, you can identify the classes and interfaces that need to be created for the actor to perform his task, and attach them to the actor. The Implementation Classes tab lists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

Tool	Action
	Add Objects – Opens a dialog box to select any class or interface in the model to implement the actor.
	Create a New Class – Creates a new class to implement the actor.
	Create a New Interface - 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 > The PowerDesigner Interface > Diagrams, Matrices, and Symbols > Diagrams > Specifying Diagrams as Related Diagrams*.

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 **Symbols > Show Symbols** in the new diagram to open the Show Symbols dialog box, select the actor to display, and click OK.

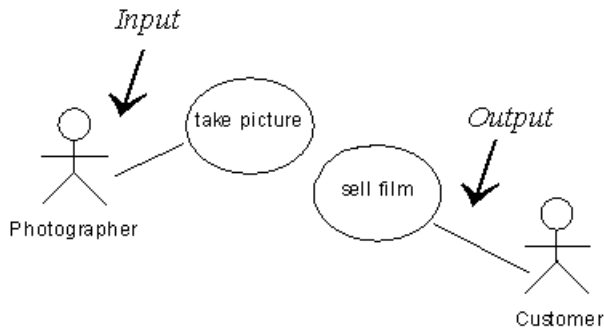
Use Case Associations (OOM)

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 > The PowerDesigner Interface > Diagrams, Matrices, and Symbols > Display Preferences*.

Example**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 > The PowerDesigner Interface > Objects*.

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 Code 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
Orientation	Defines the direction of the association. You can choose between: <ul style="list-style-type: none"> • Primary Actor – the association leads from the actor to the use case • Secondary Actor – the association leads from the use case to the actor
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.

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* on page 27.
- 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* on page 29.
- 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* on page 31.
- 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* on page 33.

Class Diagrams

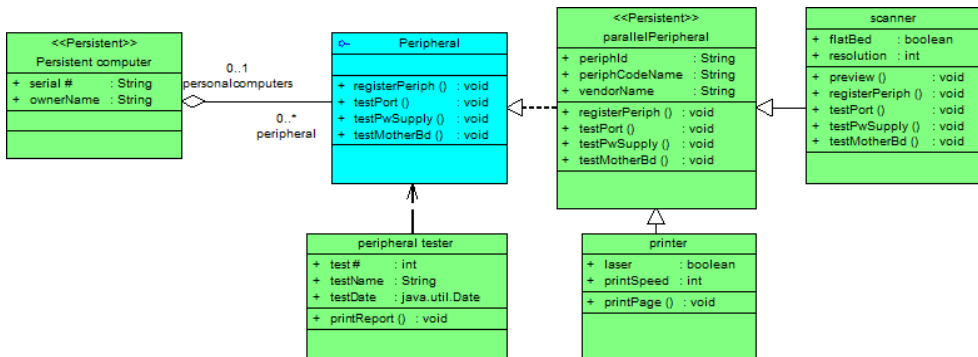
A *class diagram* is a UML diagram that provides a graphical view of the classes, interfaces, and packages that compose a system, and the relationships between them.

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.




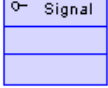










A class diagram does not express anything specific about the links of a given object, but it describes, in an abstract way, the potential link from an object to other objects.









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



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 <i>Classes (OOM)</i> on page 34.
Interface			Descriptor for the externally visible operations of a class, object, or other entity without specification of internal structure. See <i>Interfaces (OOM)</i> on page 53.
Port			Interaction point between a classifier and its environment. See <i>Ports (OOM)</i> on page 62.
Generalization			Link between classes showing that the sub-class shares the structure or behavior defined in one or more superclasses. See <i>Generalizations (OOM)</i> on page 98.
Require Link			Connects a class, component, or port to an interface. See <i>Require Links (OOM)</i> on page 106.
Association			Structural relationship between objects of different classes. See <i>Associations (OOM)</i> on page 88.
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 <i>Associations (OOM)</i> on page 88.

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 <i>Associations (OOM)</i> on page 88.
Dependency			Relationship between two modeling elements, in which a change to one element will affect the other element. See <i>Dependencies (OOM)</i> on page 101.
Realization			Semantic relationship between classifiers, in which one classifier specifies a contract that another classifier guarantees to carry out. See <i>Realizations (OOM)</i> on page 105.
Inner link			Exists when a class is declared within another class or interface. See <i>Composite and inner classifiers</i> on page 46.
Attribute	N/A	N/A	Named property of a class. See <i>Associations (OOM)</i> on page 88.
Operation	N/A	N/A	Service that can be requested from a class. See <i>Operations (OOM)</i> on page 78.

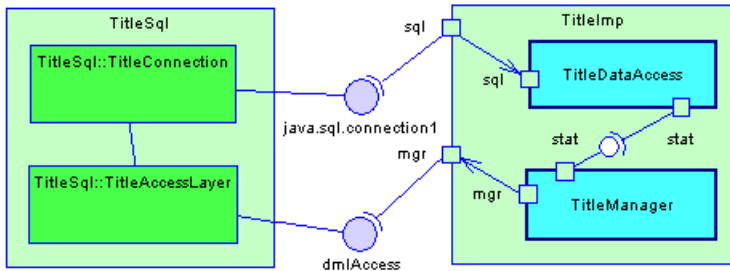
Composite Structure Diagrams

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:



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 <i>Classes (OOM)</i> on page 34.
Interface			Descriptor for the externally visible operations of a class, object, or other entity without specification of internal structure. See <i>Interfaces (OOM)</i> on page 53.
Port			Interaction point between a classifier and its environment. See <i>Ports (OOM)</i> on page 62.
Part			Classifier instance playing a particular role within the context of another classifier. See <i>Parts (OOM)</i> on page 60.
Generalization			Link between classes showing that the sub-class shares the structure or behavior defined in one or more superclasses. See <i>Generalizations (OOM)</i> on page 98.
Require Link			Connects classifiers to interfaces. See <i>Require Links (OOM)</i> on page 106.
Assembly Connector			Connects parts to each other. See <i>Assembly Connectors (OOM)</i> on page 107.
Delegation Connector			Connects parts to ports on the outside of classifiers. See <i>Delegation Connectors (OOM)</i> on page 109.
Association			Structural relationship between objects of different classes. See <i>Associations (OOM)</i> on page 88.

Object	Tool	Symbol	Description
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 <i>Associations (OOM)</i> on page 88.
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 <i>Associations (OOM)</i> on page 88.
Dependency			Relationship between two modeling elements, in which a change to one element will affect the other element. See <i>Dependencies (OOM)</i> on page 101.
Realization			Semantic relationship between classifiers, in which one classifier specifies a contract that another classifier guarantees to carry out. See <i>Realizations (OOM)</i> on page 105.
Attribute	N/A	N/A	Named property of a class. See <i>Associations (OOM)</i> on page 88.
Operation	N/A	N/A	Service that can be requested from a class. See <i>Operations (OOM)</i> on page 78.

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.

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.

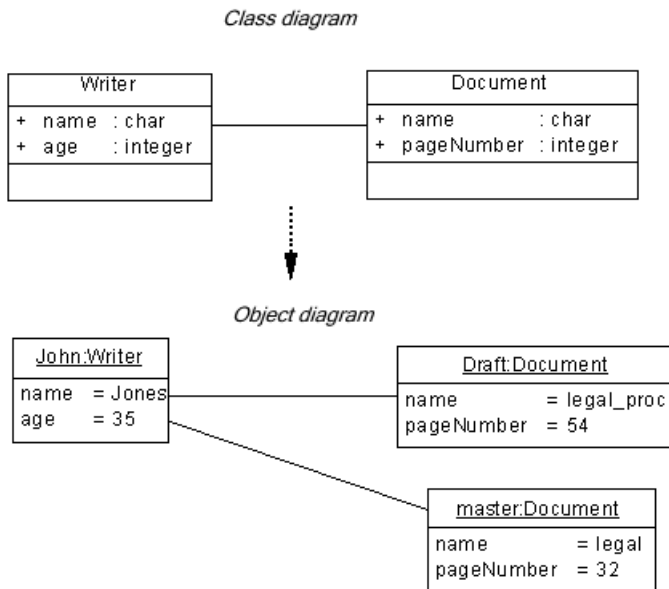
The object diagram can be used for analysis purposes: constraints between classes that are not classically represented in a class diagram can typically be represented in an object diagram.

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`.


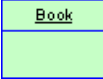
The object diagram, deduced from this class diagram, highlights some of the following details: the object named `John`, instance of the class `Writer` is linked to two different objects `Draft` and `Master` that are both instances of the class `Document`.







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.

Object Diagram Objects

PowerDesigner supports all the objects necessary to build object diagrams.

Object	Tool	Symbol	Description
Object			Instance of a class. See <i>Objects (OOM)</i> on page 56.
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 <i>Object Properties</i> on page 58.

Object	Tool	Symbol	Description
Instance link			Communication link between two objects. See <i>Instance Links (OOM)</i> on page 116.
Dependency			Relationship between two modeling elements, in which a change to one element will affect the other element. See <i>Dependencies (OOM)</i> on page 101.

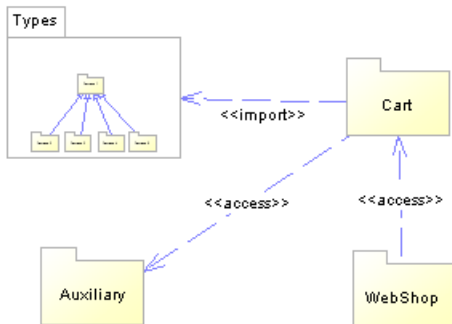
Package Diagrams

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.

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**.


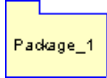




You can control the level of detail shown for each package, by toggling between the standard and composite package views via the Edit or contextual menus.

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:



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 <i>Packages (OOM)</i> on page 50.
Generalization			Link between packages showing that the sub-package shares the structure or behavior defined in one or more super-packages. See <i>Generalizations (OOM)</i> on page 98.
Dependency			Relationship between two packages, in which a change to one package will affect the other. See <i>Dependencies (OOM)</i> on page 101.

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

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 > The PowerDesigner Interface > Objects*.

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/ 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 Code field.
Extends	Specifies the parent class (to which the present class is linked by a generalization). Click the Select Classifier tool to the right to specify a parent class and click the Properties 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"> • <<actor>> - Coherent set of roles that users play • <<enumeration>> - List of named values used as the range of an attribute type • <<exception>> - Exception class, mainly used in relation to error messages • <<implementationClass>> - Class whose instances are statically typed. Defines the physical data structure and methods of a class as implemented in traditional programming languages • <<process>> - Heavyweight flow that executes concurrently with other processes • <<signal>> - Specification of asynchronous stimulus between instances • <<metaclass>> - a metaclass of some other class • <<powerType>> - a metaclass whose instances are sub-classes of another class • <<thread>> - Lightweight flow that executes concurrently with other threads within the same process. Usually executes inside the address space of an enclosing process • <<type>> - Abstract class used to specify the structure and behavior of a set of objects but not the implementation • <<utility>> - Class that has no instances <p>Other language-specific stereotypes may be available if they are specified in the object language file (see <i>Customizing and Extending PowerDesigner > Extension Files > 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"> • Private – only to the object itself • Protected – only to the object and its inherited objects • Package – to all objects contained within the same package • Public – to all objects (option by default)

Property	Description
Cardinality	<p>Specifies the number of instances a class can have. You can choose between:</p> <ul style="list-style-type: none"> • 0..1 – None to one • 0..* – None to an unlimited number • 1..1 – One to one • 1..* – One to an unlimited number • * – Unlimited number
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"> • Class • Generic • Bound – 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. <p>If you specify either Generic or Bound, then the Generic tab is displayed, allowing you to control the associated type variables. For more information on generic types and binding classes to them, see <i>Generic Types and Methods</i> on page 42.</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"> • Generate table - the class is generated as an entity or table. • Migrate columns – [PDM only] the class is not generated, and its attributes and associations are migrated to the generated parent or child table. • Generate ADT – [PDM only] the class is generated as an abstract data type (see <i>Data Modeling > Building Data Models > Physical Diagrams > Abstract Data Types</i>). • Value Type – the class is not generated, and its attributes are generated in their referencing types. <p>For more information, see <i>Managing Object Persistence During OOM to PDM Generation</i> on page 286.</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 <i>PURCH</i> in the Code box.</p>
Inner to	<p>Specifies the name of the class or interface to which the current class belongs as an inner classifier</p>
Association class	<p>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.</p>

For information about additional properties available if the class is a Web service implementation class, see *Web Service Implementation Class Properties* on page 238

The following tabs are also available:

- Attributes - lists and lets you add or create attributes (including accessors) associated with the class (see *Attributes (OOM)* on page 65). 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)* on page 75).
- Operations - lists and lets you add or create operations associated with the class (see *Operations (OOM)* on page 78).
- 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 *Generic Types and Methods* on page 42)

- Ports - lists and lets you create ports associated with the class (see *Ports (OOM)* on page 62).
- Parts - lists and lets you create parts associated with the class (see *Parts (OOM)* on page 60).
- Associations - lists and lets you create associations associated with the class (see *Associations (OOM)* on page 88).
- Inner Classifiers - lists and lets you create inner classes and interfaces associated with the class (see *Composite and Inner Classifiers* on page 46).
- Related Diagrams - lists and lets you add model diagrams that are related to the class (see *Core Features Guide > The PowerDesigner Interface > Diagrams, Matrices, and Symbols > Diagrams > Specifying Diagrams as Related Diagrams*).
- Script - lets you customize the class creation script (see *Customizing Object Creation Scripts* on page 10)
- Preview - lets you view the code to be generated for the class (see *Previewing OOM Code* on page 8)

Creating Java BeanInfo Classes

If you are using the Java object language, you can create Java BeanInfo classes from any class with a type of "JavaBean".

A JavaBean is a reusable software component written in Java that can be manipulated visually in a builder tool. A Java BeanInfo class is used as a standard view of a Bean. Each JavaBean can implement a BeanInfo class. Bean implementors may want to provide explicit information about the methods, properties, and events of a Bean by providing a Java BeanInfo class.

The BeanInfo class is generated with an attribute, and the following operations:

- constructor
- getPropertyDescriptors();
- getMethodDescriptors();

You can view the complete code by clicking the Preview tab in the BeanInfo class property sheet.

Attribute Created

The attribute has the following code:

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

Operations Created

The constructor has the following code:

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

The getPropertyDescriptors() operation has the following code:

```

public PropertyDescriptor[] getPropertyDescriptors ()
{
    // 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;
}

```

The `getMethodDescriptors()` operation has the following code:

```

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;
}

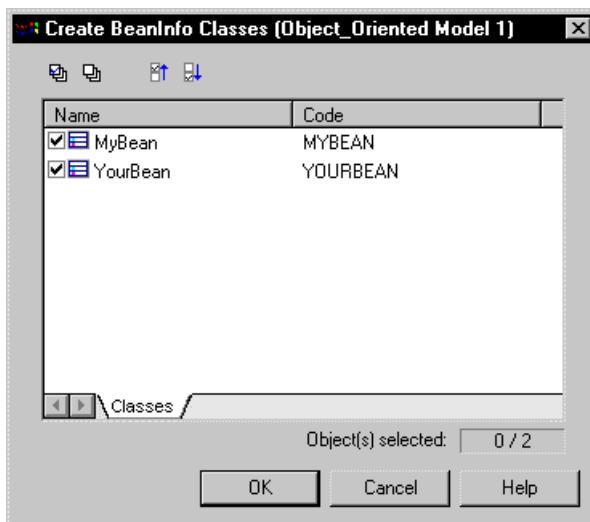
```

When you create a Java BeanInfo class, a dependency link is automatically created between both classes and the stereotype of the Java BeanInfo class is set to <<BeanInfo>>.

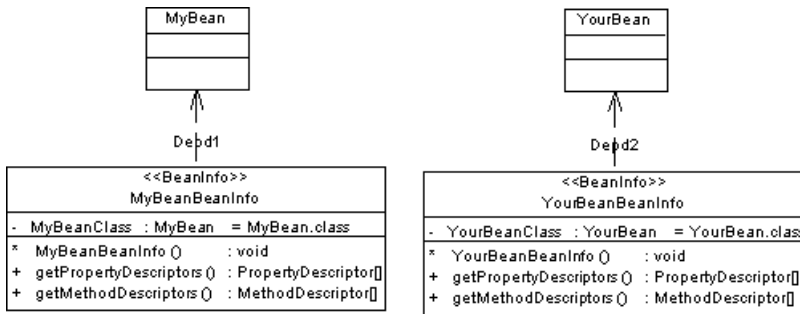
Creating a Java BeanInfo Class from the Language Menu

You can create a Java BeanInfo class from the Language menu.

1. Select **Language > Create BeanInfo Classes** to display the Create BeanInfo Classes selection window. This window contains a list of all the classes of type JavaBean in the model.



2. Select the classes for which you want to generate Java BeanInfo classes and click OK. A BeanInfo class is created in the model for each selected class.



Creating a Java BeanInfo Class from the Class Contextual Menu

You can create a Java BeanInfo class from the class contextual menu.

Right-click a class in the diagram, and select Create BeanInfo Class from the contextual menu.

Generic Types and Methods

Generic types and methods are a new feature of Java 5.0. A generic type is a class or interface that has one or more type variables and one or more methods that use a type variable as a placeholder for an argument or return type.

Using generic types and methods allows you to take advantage of stronger compile-time type checking. When a generic type is used, an actual type is specified for each type variable. This additional type information is used by the compiler to automatically cast the associated return values.

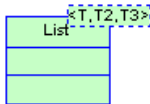
Creating Generic Types

PowerDesigner allows you to designate classes and interfaces as generic types.

You define a list of type variables that will be used as datatypes for attributes, method parameters, or return types. PowerDesigner requires the existence of 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.

1. Open the property sheet of the class or interface, and select Generic from the Type list on the General tab. The Generic tab will be automatically displayed, and a type variable created in the list in the tab.
2. Click the Generic tab, and add any additional type variables that you require with the Add a Row tool. 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 will now display 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.

Creating Generic Methods

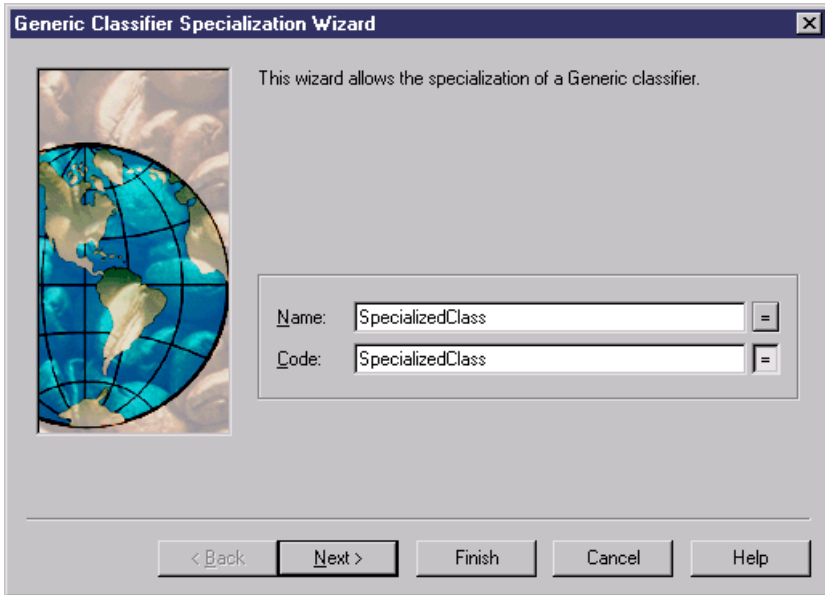
PowerDesigner allows you to designate operations as generic methods. Generic methods are methods that have their own list of type variables.

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.

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.

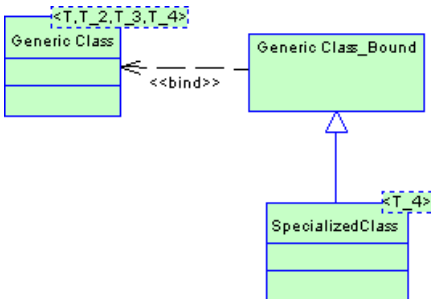
1. Right-click a generic class or interface, and select Create Specialized Class (or Interface) from the contextual menu to open the Generic Classifier Specialization Wizard:



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.

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.

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>>.

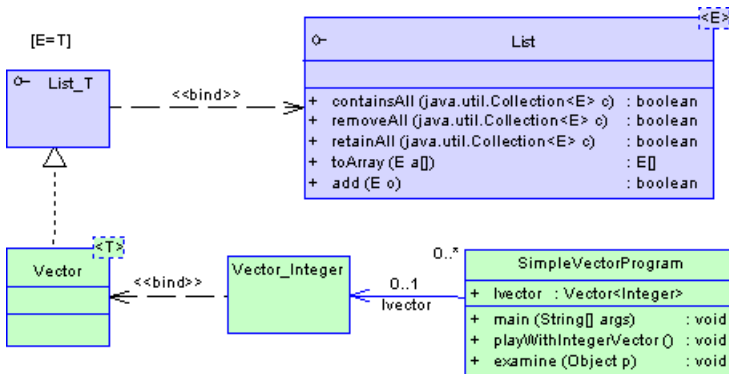
Generic Type Example

In the example below, 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.

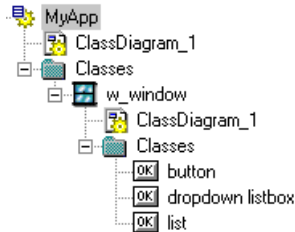


You must create a bound class for a generalization or a realization. However, we could have specified a parameter value for the generic type <T> directly (without creating a bound class) as an attribute data type, parameter data type, or return data type, by simply typing the following expression in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

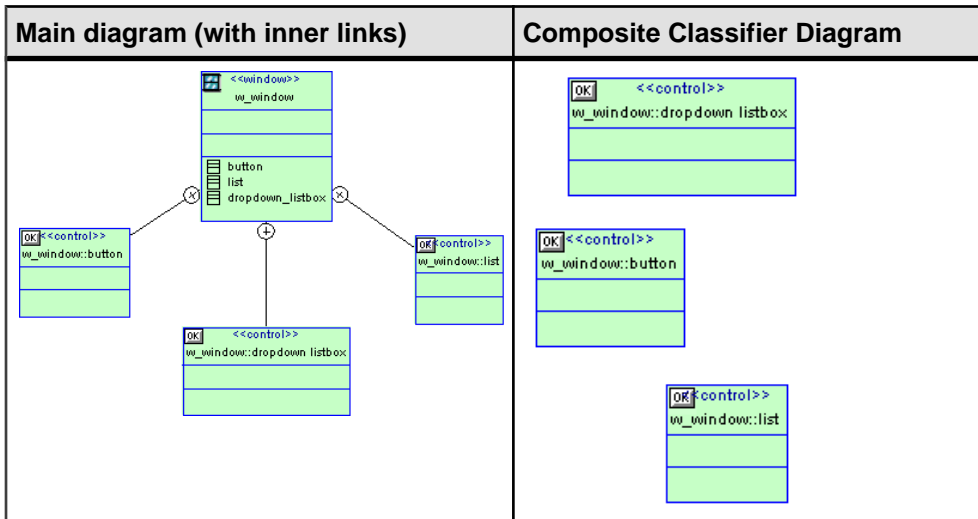
```
Vector<integer>
```

Composite and Inner Classifiers

A composite classifier is a class or an interface that contains other classes or interfaces (called inner classifiers). Inner classifiers are listed in the Browser as children of their composite classifier.



Inner classifiers can be displayed directly in the main class diagram, or in an additional class diagram specific to the composite classifier:



Note: You can display multiple composite classifiers in a composite structure diagram (see *Composite Structure Diagrams* on page 29).

Note: In previous versions of PowerDesigner, there was no composite classifier and inner classifiers used to appear at the same hierarchical level as their parent in the Browser tree view and in the list of classifiers.

Creating Inner Classifiers

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 **New > Class** or **New > Interface**.
- Select the **Inner Link** tool in the Toolbox and click and use it to connect two classes in the diagram. The first class will become a composite class and the second, an inner classifier
- Create a composite classifier diagram dedicated to the class (see following section), and create classes or interfaces there.

All inner classifiers are listed in the bottom of the class symbol.

Creating a Composite Classifier Diagram

You may want to create a diagram to show the internal structure of a composite classifier. You can create a composite classifier diagram in any of the following ways:

- Right-click a class or interface in the Browser, and select **New > Class Diagram**
- Double-click a class or interface in the diagram while holding down the CTRL key

The composite classifier diagram will be empty by default, even if the composite classifier already includes some inner classifiers. You can create symbols for the internal classifiers by selecting **Symbol > Show Symbols**, or by dragging and dropping them from the Browser to the diagram.

If you have created a composite classifier diagram, you can display it in the main class diagram by right-clicking the class symbol and selecting **Composite View > Read-only (Sub-Diagram)**.

Attaching a Classifier to a Data Type or a Return Type

A data type or a return type can be a classifier. You can choose to attach a class, an interface, or a classifier from a JDK library for example, to the following objects:

- Attribute data type
- Operation return type
- Parameter data type

The classifier can belong to the current model or to another model. If it belongs to the current model or package, it is displayed by default together with the other classifiers of the current package. If it belongs to another model or package, a shortcut of the classifier is automatically created in the current package.

Fully Qualified Name

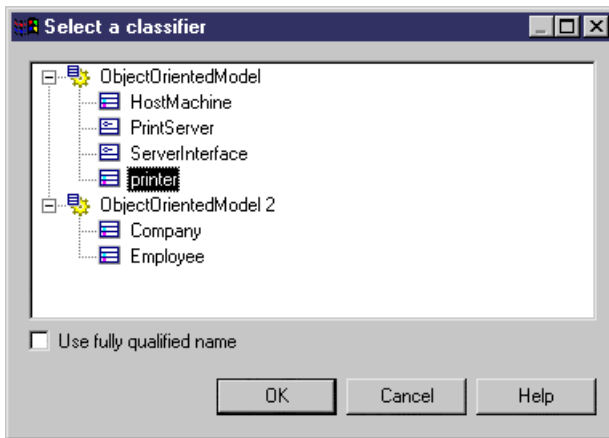
If the classifier belongs to another package of the same model, you can manually attach the classifier to a data type or return type by typing the fully qualified name of the classifier directly in the data type or return type box.

For example, if you need to reference `Class_2` as a classifier data type of an attribute named `Attr_1` in `Class_1`, open the `Attr_1` property sheet and type `PACKAGE_2 . CLASS_2` in the Data type list.

- Package_1
- |__ Class_1
- |__ Attr_1
- Package_2
- |__ Class_2

After you type the fully qualified name (path and classifier name), it is displayed in the data type or return type box.

When a data type or a return type is created, there is no classifier attached to it, that is why the Use fully qualified name check box is not selected in the Select a Classifier dialog box.



However if you type a relative name, like 'Lang.Integer' instead of 'Java.Lang.Integer', the Use fully qualified name check box is automatically selected because a fully qualified name corresponds to this name.

Attaching a Classifier

To attach a classifier:

1. Double-click a class in the diagram to display its property sheet.
 - To attach a classifier to an attribute data type, click the Attributes tab, select an attribute in the list, and click the Properties tool to open the attribute property sheet.
 - To attach a classifier to an operation return type, click the Operations tab, select an operation in the list, and click the Properties tool to open the operation property sheet.
 - To attach a classifier to an operation parameter data type, click the Operations tab, select an operation in the list, and click the Properties tool to open the operation property sheet. Then click the Parameters tab, select a parameter in the list, and click the Properties tool to open the parameter property sheet
2. Click the Select a Classifier button beside the Data type listbox to open the Selection window. This window contains a list of all the classifiers in the model.

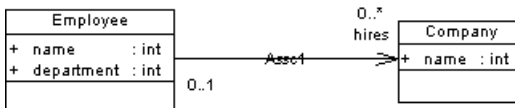
3. Select the classifier to be attached and click OK to return to the property sheet. The classifier name is now displayed in the Data type list.
4. Click OK.

Note: When you select a data type that matches the code of a classifier, the data type is automatically synchronized if the code of the classifier changes.

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;
}
  
```

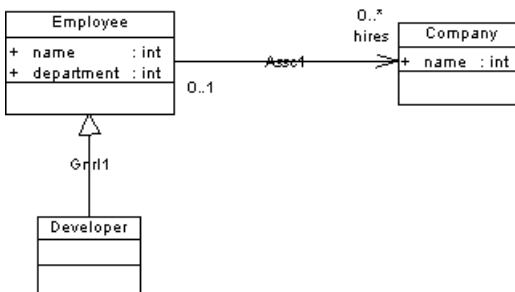
The association between `Employee` and `Company` is migrated as the attribute `public COMPANY hires []`.

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

Inherited Associations

When a class inherits from a parent class through a generalization link, you can use the list of Inherited Associations to view the list of migrated attributes from the parent of the current class.

For example, the class `Developer` inherits from `Employee` through a generalization.



If you double-click the class Developer, click the Associations tab in the class property sheet and click the Inherited button, the list of Inherited Associations displays the migrated attribute of the class Employee.

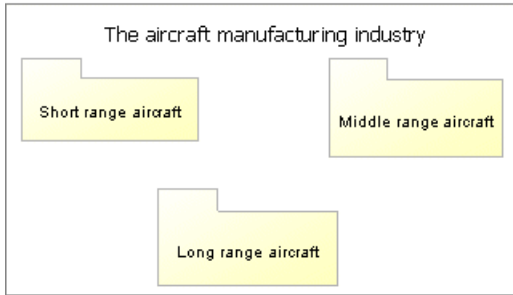
Name	Code	Role A	Multiplicity A	Role B
Association_1	Association_1		0..1	hires

For more information on associations, see *Associations (OOM)* on page 88.

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.

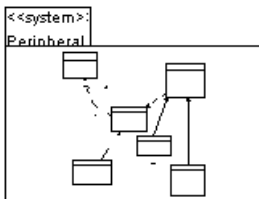
When you work with large models, you can split them into smaller subdivisions to avoid manipulating the entire set of data of the model. Packages can be useful to assign portions of a model, representing different tasks and subject areas to different development teams.



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**.

OOM Package Properties

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

Property	Description
Name	Name that clearly identifies the package
Code	Codes are references for packages
Comment	Optional label that describes a package and provides additional information

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"> • <<archive>> Jar or War archive (Java only) • <<assembly>> – Specifies that a package produces a portable executable (PE), (C# and VB.NET only) • <<CORBAModule>> – UML Package identified as IDL module (IDL-CORBA only) • <<facade>> – Package is a view of another package • <<framework>> – Package consists mostly of patterns • <<metamodel>> – Package is an abstraction of another package • <<model>> – Specifies a semantically closed abstraction of a system • <<stub>> – Package serves as a proxy for the public contents of another package • <<subsystem>> – Grouping of elements, some of which constitute a specification of the behavior offered by the other contained elements • <<system>> – Package represents the entire system being modeled • <<systemModel>> – Package that contains other packages with the same physical system. It also contains all relationships and constraints between model elements contained in different models • <<topLevel>> – Indicates the top-most package in a containment hierarchy
Use parent namespace	Defines the package as being the area in which the name of an object must be unique in order to be used
Default diagram	Diagram displayed by default when opening the package

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.

Interfaces (OOM)

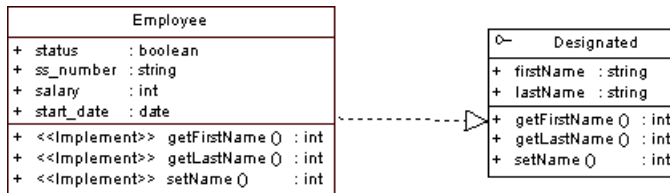
An interface is similar to a class but it is used to define the specification of a behavior. It is a collection of operations specifying the externally visible behavior of a class. It has no implementation of its own.

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).



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 > The PowerDesigner Interface > Objects*.

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 Code field.
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"> • <<metaclass>> - interface that will interact with a model that contains classes with metaclass stereotypes • <<powertype>> - a metaclass whose instances are sub-classes of another class • <<process>> - heavyweight flow that can execute concurrently with other processes • <<thread>> - lightweight flow that can execute concurrently with other threads within the same process. Usually executes inside the address space of an enclosing process • <<utility>> - a class that has no instances
Visibility	<p>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:</p> <ul style="list-style-type: none"> • Private – only to the object itself • Protected – only to the object and its inherited objects • Package – to all objects contained within the same package • Public – to all objects (option by default)
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"> • Interface • Generic • Bound – 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. <p>If you specify either Generic or Bound, then the Generic tab is displayed, allowing you to control the associated type variables (see <i>Generic Types and Methods</i> on page 42).</p>
Generate	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)* on page 65.
- **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)* on page 78.
- **Generic** - 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 *Generic Types and Methods* on page 42)
- **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 *Composite and Inner Classifiers* on page 46.
- **Related Diagrams** - lists and lets you add model diagrams that are related to the interface (see *Core Features Guide > The PowerDesigner Interface > Diagrams, Matrices, and Symbols > Diagrams > Specifying Diagrams as Related Diagrams*).

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.



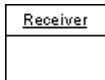
Objects in an Object Diagram

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.



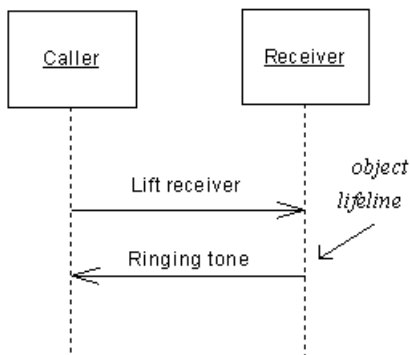
Objects in a Sequence Diagram

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.



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 > The PowerDesigner Interface > Objects*.

Object Properties

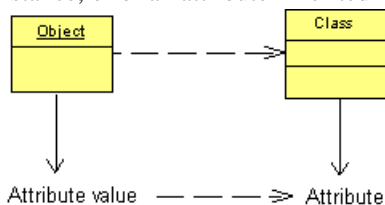
To view or edit an object's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

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

Property	Description
Name/Code/ Comment	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 Code field. 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 Classifier . Names must be unique per classifier.
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 <i>Linking a Classifier to an Object</i> on page 59).
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

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.

From the object property sheet, you can:

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

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.

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.

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.

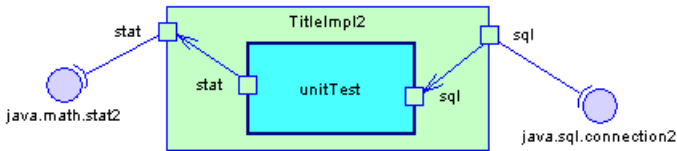
A part can be created in the following diagrams:

- Composite Structure Diagram (inside a class)
- Component Diagram (inside a component)

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.

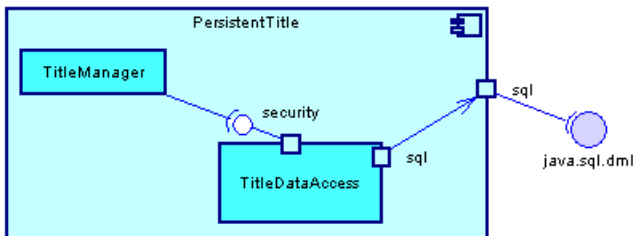
Parts in a Composite Structure Diagram

In the example below, the class TitleImpl2 contains a part called unitTest



Parts in a Component Diagram

In the example below, the component PersistentTitle contains two parts, TitleManager and TitleDataAccess



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 > The PowerDesigner Interface > Objects*.

Part Properties

To view or edit a part'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 object.
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 Code 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 name-space. You can choose between: <ul style="list-style-type: none"> • Private – only to the object itself • Protected – only to the object and its inherited objects • Package – to all objects contained within the same package • Public – to all objects (option by default)
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. <p>You can choose between:</p> <ul style="list-style-type: none"> • * – none to unlimited • 0..* – zero to unlimited • 0..1 – zero or one • 1..* – one to unlimited • 1..1 – exactly one
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.

Property	Description
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. You can create ports directly in this tab. For more information, see *Ports (OOM)* on page 62.

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:

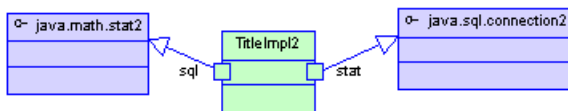
- a part via a delegation connector, through which requests can be made to invoke the behavioral features of a classifier
- an interface via a require link, through which the port may specify the services a classifier provides (offers) to its environment as well as the services that a classifier expects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

A port can be created in the following diagrams:

- Class Diagram (on a class)
- Composite Structure Diagram (on a class, a part, or an interface)
- Component Diagram (on a component or a part)

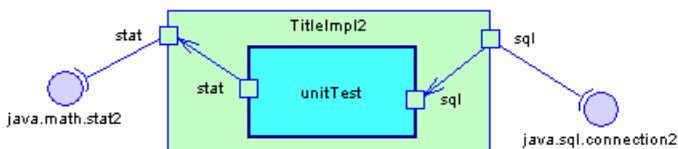
Ports in a Class Diagram

In the example below, 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:



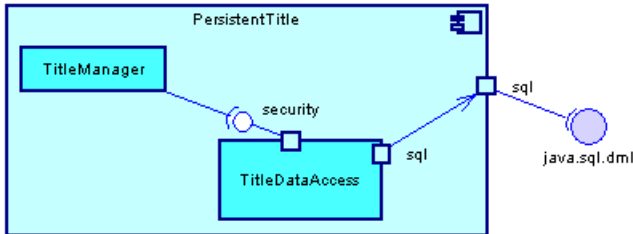
Ports in a Composite Structure Diagram

In the example below, 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:



Ports in a Component Diagram

In the example below, the use of ports to connect parts with an enclosing component is demonstrated:



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.

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

Port Properties

To view or edit a port'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.
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 Code 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.

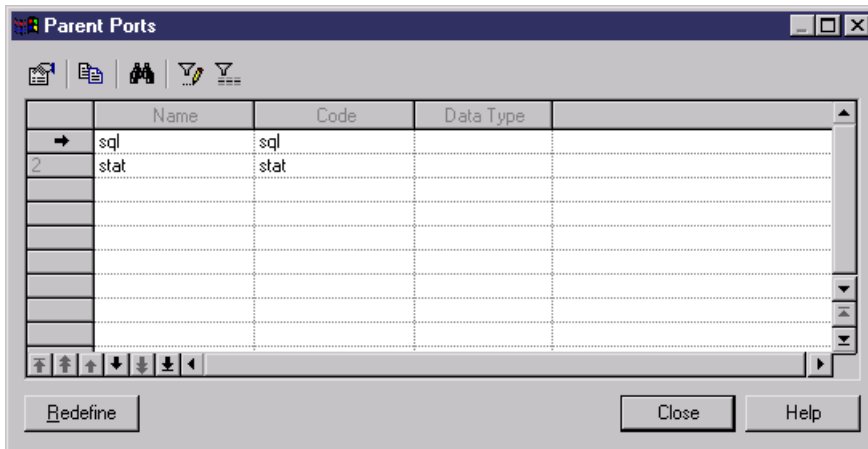
Property	Description
Visibility	<p>Specifies the visibility of the object, how it is seen outside its enclosing namespace. You can choose between:</p> <ul style="list-style-type: none"> • Private – only to the object itself • Protected – only to the object and its inherited objects • Package – to all objects contained within the same package • Public – (default) to all objects
Data type	Specifies a classifier as a data type.
Multiplicity	<p>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.</p> <p>You can choose between:</p> <ul style="list-style-type: none"> • * – none to unlimited • 0..* – zero to unlimited • 0..1 – zero or one • 1..* – one to unlimited • 1..1 – exactly one
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	<p>Specifies that this port is used to provide the published functionality of a classifier (default).</p> <p>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.</p>
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.

Redefining Parent Ports

A classifier that is connected to a parent by way of a generalization can redefine the ports of the parent.

1. Open the property sheet of a class, interface, or component, and click the Ports tab.

- Click the Redefine button at the bottom of the tab to open the Parent Ports window, which will display a list of ports belonging to the parent classifier.



- Select a port and then click Redefine to have it redefined by the child classifier.
- Click Close to return to the child's property sheet. The redefined port will now appear in the list on the Ports tab.

Attributes (OOM)

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

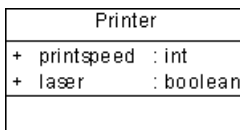
An attribute can be created for a class or interface in the following diagrams:

- 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`:



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.

Creating an Attribute

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

- Open the Attributes tab in the property sheet of a class, identifier, or interface, and click the Add a Row tool

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

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.

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 Code 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	<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, or similarly, the other object can affect the properties of the class. You can choose between:</p> <ul style="list-style-type: none"> • Private – only to the class to which it belongs • Protected – only to the class and its derived objects • Package – to all objects contained within the same package • Public – to all objects (option by default)
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"> • 0..1 – zero or one • 0.* – zero to unlimited • 1..1 – exactly one • 1.* – one to unlimited • * – none to unlimited <p>You can change the default format of multiplicity from the registry.</p> <pre>HKEY_CURRENT_USER\Software\Sybase\PowerDesigner 16\ModelOptions\Cld\ MultiplicityNotation = 1 (0..1) or 2 (0,1)</pre>

Property	Description
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 <code>int n[4,6,8]</code> or set array size to [,] to get the c# syntax <code>int[,]</code> n;</p> <p>Depending on the model language, the following will be generated:</p> <ul style="list-style-type: none"> • Java, C# and C++ – [2][4][6] • PowerBuilder – [2,4,6] • VB .NET – (2,4,6)
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 <i>Adding Getter and Setter Operations to a Classifier</i> on page 71.
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.

Property	Description
Changeability	<p>Specifies if the value of the attribute can be modified once the object has been initialized. You can choose between:</p> <ul style="list-style-type: none"> • Changeable – The value can be changed • Read-only – Prevents the creation of a setter operation (a setter is created in the method inside the class) • Frozen – Constant • Add-only – Allows you to add a new value only
Domain	<p>Specifies a domain (see <i>Domains (OOM)</i> on page 120), which will define the data type and related data characteristics for the attribute and may also indicate check parameters, and business rules.</p> <p>Select a domain from the list, or click the Ellipsis button to the right to create a new domain in the List of Domains.</p>
Primary Identifier	<p>[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</p>
Migrated from	<p>Contains the association name that is at the origin of creation of the attribute. Click the Properties tool to the right of the field to open the association property sheet.</p> <p>For more information on migrating attributes, see <i>Migrating Association Roles in a Class Diagram</i> on page 96.</p>
Persistent	<p>[class attributes] Specifies that the attribute will be persisted and stored in a database (see <i>Managing Object Persistence During OOM to PDM Generation</i> on page 286).</p>
Code	<p>Specifies the code of the table or entity that will be generated in a persistent CDM or PDM model.</p>
Data type	<p>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.</p>
Length	<p>Specifies the maximum number of characters of the persistent data type.</p>
Precision	<p>Specifies the number of places after the decimal point, for persistent data type values that can take a decimal point</p>
Overrides	<p>[PowerBuilder only] Indicates which parent attribute the current attribute is overriding through a generalization link</p>

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	Specifies the range of acceptable values. You can set a: <ul style="list-style-type: none">• Minimum - The lowest acceptable numeric value• Maximum - The highest acceptable numeric value• Default - The value assigned in the absence of an expressly entered value.
Characteristics	Specifies the shape of acceptable data. You can choose a: <ul style="list-style-type: none">• Format - A number of standard formats are available in the list and you can create your own format for reuse elsewhere or simply enter a format in the field.• Unit - A standard measure. This field is informational only and is not generated.• No space - Space characters are not allowed.• Cannot modify - The value cannot be updated after initialization.
Character case	Specifies the acceptable case for the data. You can choose between: <ul style="list-style-type: none">• Mixed case [default]• Uppercase• Lowercase• Sentence case• Title case
List of values	Specifies the various values that are acceptable. Select the Complete check box beneath the list to exclude all other values not appearing in the list.

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.

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/ 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 Code 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"> • Date/Time • String • Regular Expression
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.

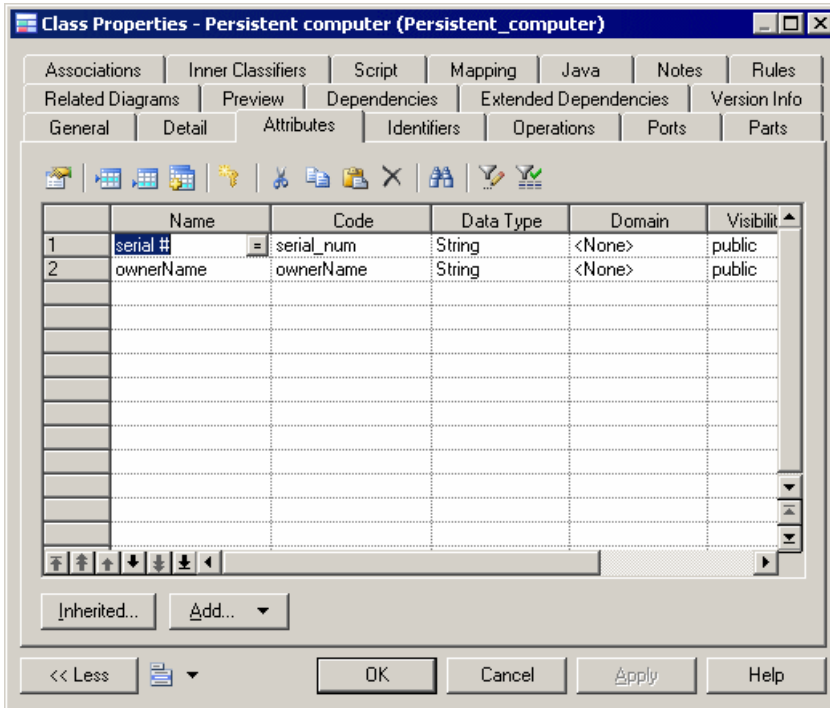
Specifying Advanced Constraints

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.

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.

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



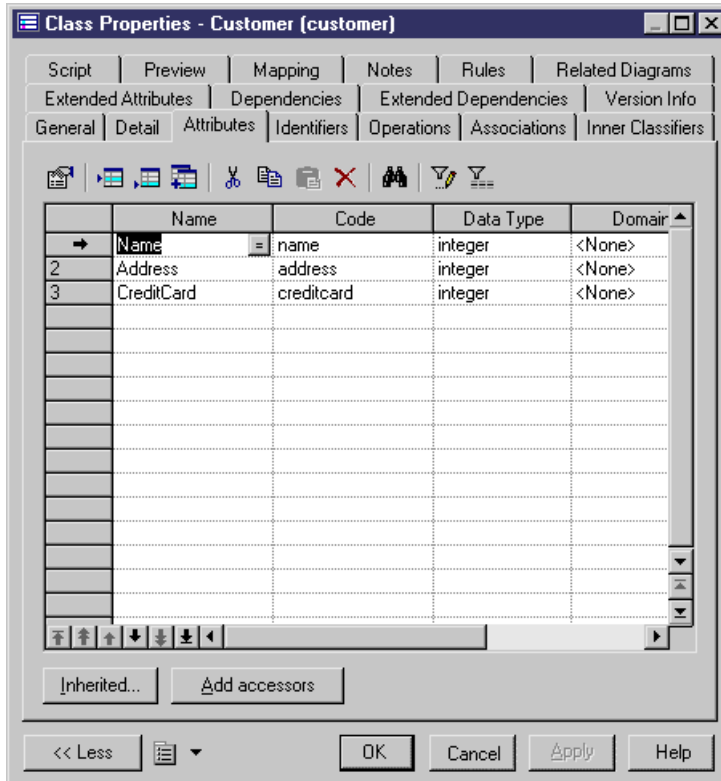
2. Select one or more attributes and then click the **Add** button at the bottom of the attributes tab and select the action you want to perform. Depending on the target language, some of the following actions will be available:
 - Get/Set Operations - Creates get and set operations on the **Operations 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.

Copying an Attribute to a Class, Interface, or Identifier

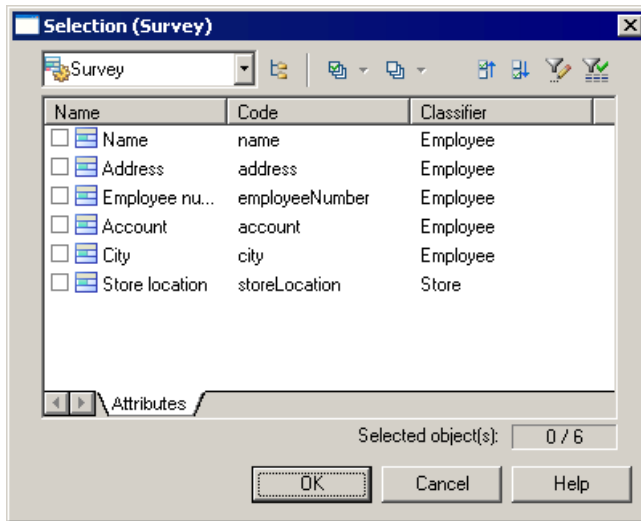
You can duplicate an attribute from one classifier to another. If the classifier already contains an attribute with the same name or code as the copied attribute, the copied attribute is renamed.

For example the attribute PERIPHLD is renamed into PERIPHLD2 when it is copied to a class which already contains an attribute PERIPHLD.

1. Open the property sheet of a class, interface, or identifier, and click the Attributes tab.



2. Click the Add Attributes tool to open the Selection window. This window contains a list of all the attributes in the model, except those that already belong to the object.



3. Select the attributes you want to add to the object.

or

Use the Select All tool to add all the attributes in the list to the object.

4. Click OK to add the selected attributes to the current object.

Overriding an Attribute in PowerBuilder

The Inherited Attributes dialog box allows you to:

- View the attributes inherited from the parent classes or interfaces
- Override an attribute, this feature is only available in PowerBuilder

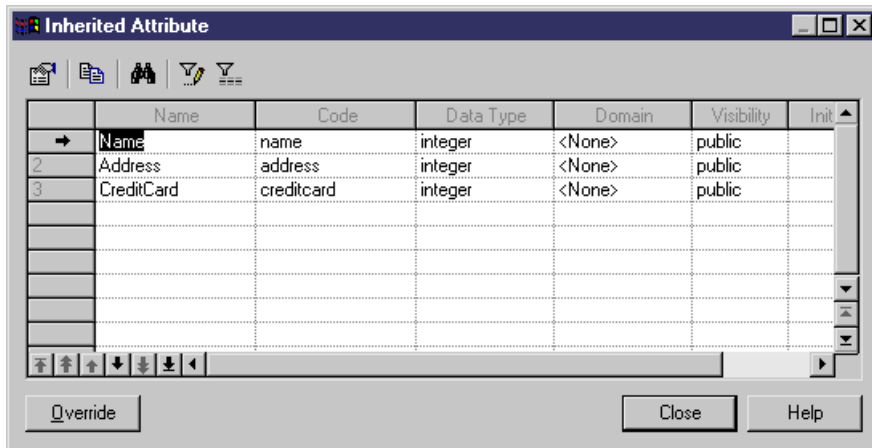
Within the context of an inheritance, you can add an attribute that belongs to a parent class to a child class. This is called overriding. The attribute of the parent class is said to be overridden when this attribute is redefined in a child class.

Once you add an attribute to a class, you can only modify the initial value of the attribute. You cannot modify other attribute properties.

Adding an Inherited Attribute to a Class

To add an inherited attribute to a class:

1. Double-click a class that is linked to a parent class in the diagram to open its property sheet, and then click the Attributes tab.
2. Click the Inherited button to open the Inherited Attribute window. This window lists all the attributes that belong to all the parent classes of the class.



3. Select an attribute and click Override and then Close. A copy of the attribute is added to the list of attributes of the child class. It is grayed to indicate that its properties cannot be modified, and its stereotype is set to <<Override>>.
4. Click OK.

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.

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.

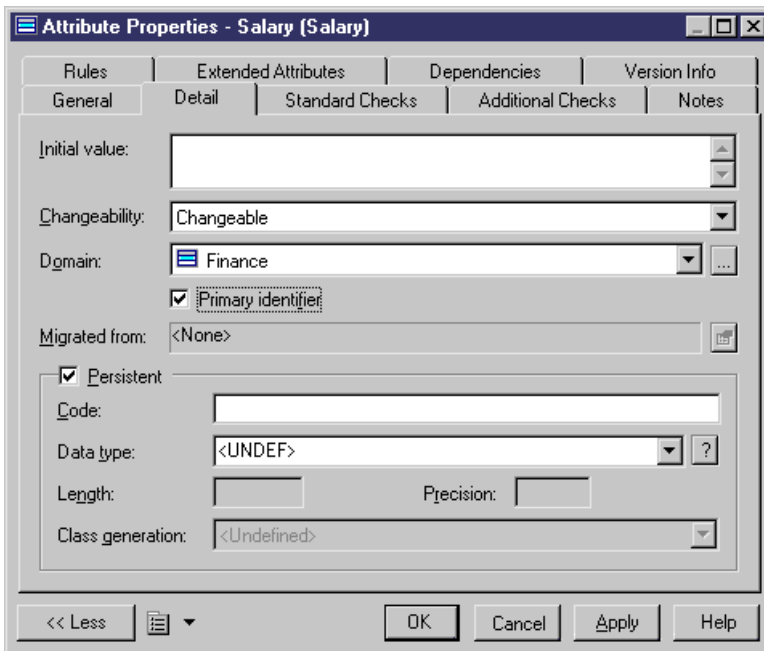
- Open the Attributes tab in the property sheet of a class or interface, and select the Primary Identifier checkbox when you create an attribute.

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

Creating a primary identifier when you create the class attributes

You can create a primary identifier when you create attributes for a class.

1. Double-click a class in the diagram to display its property sheet, and then click the Attributes tab.
2. Double-click an attribute in the list to display its property sheet, and then click the Detail tab.
3. Select the Primary Identifier check box and click OK to return to the Attributes tab of the class property sheet.



4. Click the Identifiers tab to view the new identifier in the list.
5. Click OK.

Defining the Primary Identifier from the List of Identifiers

You can define the primary identifier from the List of Identifiers.

1. Select **Model > Identifiers** to display the list of identifiers.
2. Double-click an identifier in the list to display its property sheet.

3. Select the Primary Identifier check box.
4. Click OK in each of the dialog boxes.

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.

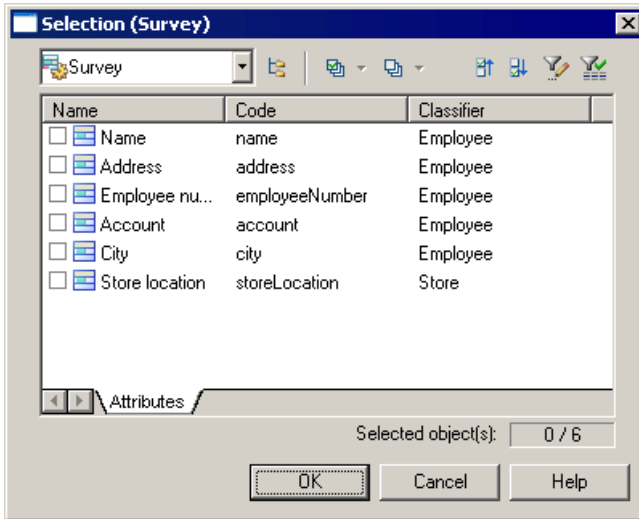
The **General** tab contains the following properties:

Property	Description
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 Code field.
Class	Specifies the class to which the identifier belongs.
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.

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.

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.

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
+ start engine () : 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* on page 363.

Creating an Operation

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

- Open the Operations tab in the property sheet of a class or interface, and click the Add a Row tool
- Right-click a class or interface in the Browser, and select **New > Operation**

You can create the following types of operation in a class or interface:

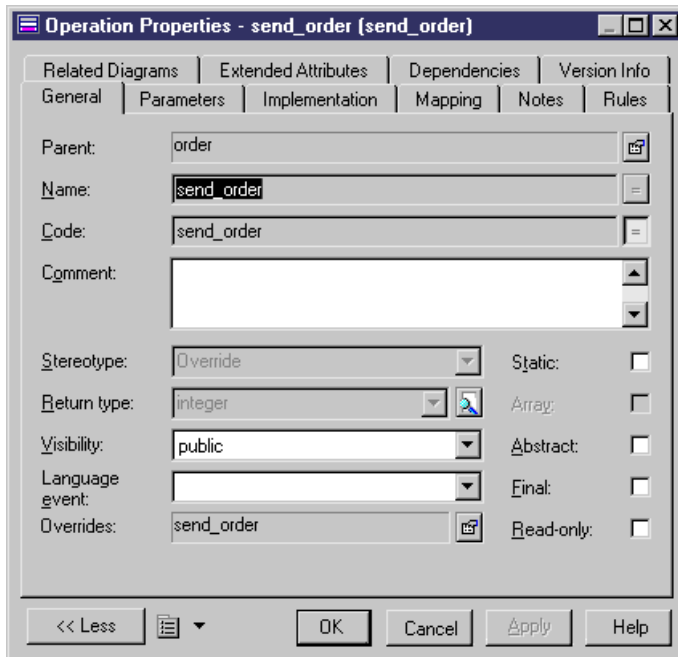
- Your own operations
- Operations inherited from a parent of the current class
- Standard operations such as constructors/destructors or initializers
- Operations to be implemented from an interface by which the current class is linked by a realization link

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

Creating a User-Defined Operation

You can create your own operations from the **Operations** tab of a classifier.

1. Double-click a class or interface to display its property sheet, and then click the Operations tab.
2. Click the Add a Row tool to create a new operation.
3. Click the Properties tool and click Yes in the confirmation box to open the property sheet of the new operation.

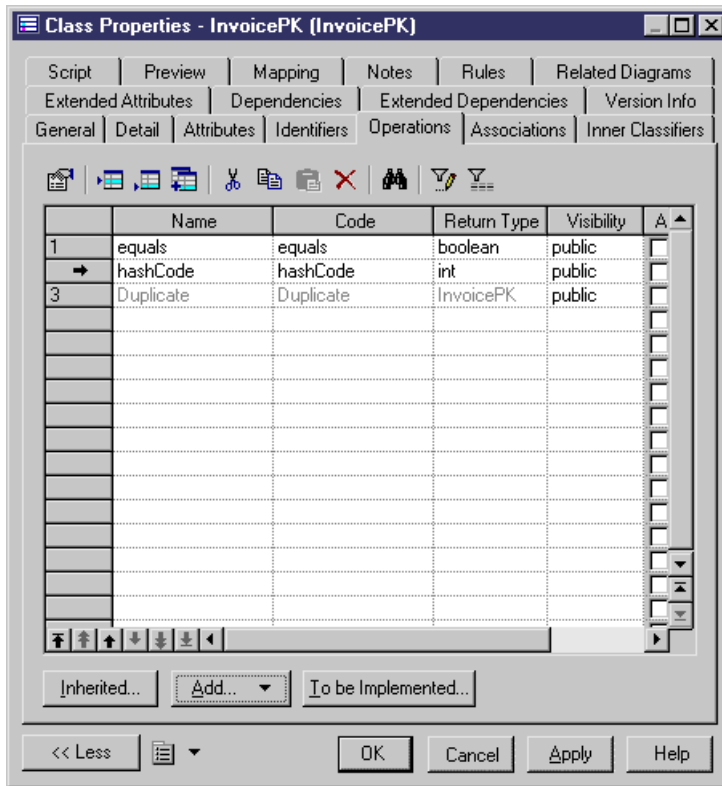


4. Enter any necessary properties and then click OK to complete the creation of the operation.

Creating a Standard Operation

PowerDesigner can create standard operations for your classifiers using tools on the classifier property sheet **Operations** tab.

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



2. Click the **Add** button at the bottom of the tab, and select the type of standard operation you want to add. Depending on the target language, some of the following types of standard operations will be available:
 - *Default Constructor/Destructor* - to perform initialization/cleanup for classifiers. You can add parameters afterwards.
 - *Copy Constructor Operations* - to copy the attributes of a class instance to initialize another instance.
 - *Initializer/Static Initializer Operations* - [Java only] to initialize a class before any constructor.
 - *Duplicate Operations* - to create and initialize an instance of a class within the class.
 - *Activate/Deactivate Operations* - [PowerBuilder only]

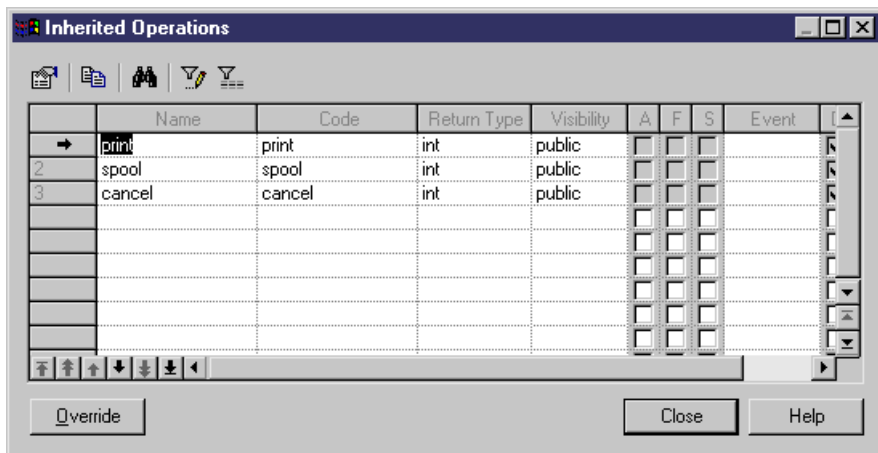
The operation is added to the list. Some or all of its properties will be dimmed to indicate that they are uneditable.
3. [optional] Select the operation and then click the **Properties** tool to add parameters to it or otherwise complete its definition.
4. Add other operations as necessary, or click **OK** to close the property sheet and return to your model.

Inheriting and Overriding Operations from Parent Classifiers

PowerDesigner lets you view and override operations inherited from parent classes from the child class property sheet **Operations** tab.

In order to inherit and be able to override inherited operations, your class must be linked to one or more parent classifiers that have operations specified.

1. Open the property sheet of a class that is linked by a generalization link to one or more parents and click the **Operations** tab.
2. Click the **Inherited** button to open the Inherited Operations window, which lists the operations that the classifier inherits from its parents:



3. Select one or more operations, and then click the **Override** button to copy them to the list of operations of the child class with their stereotype set to <<Override>>. Each overridden operation has the same signature (name and parameters) as the original operation, and you can only modify the code on its **Implementation** tab.
4. Click **Close** to return to the **Operations** tab, and then click **OK** to return to your model.

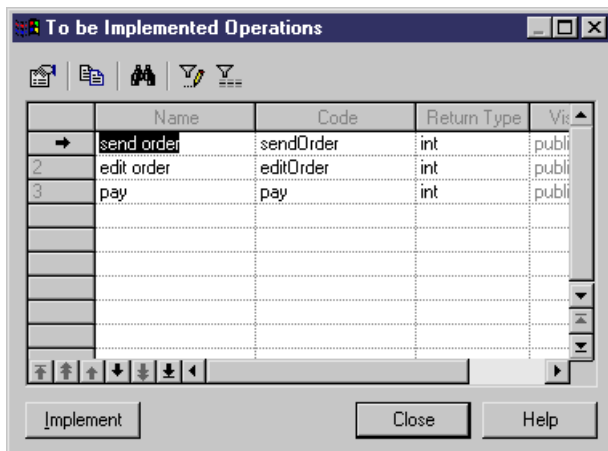
Creating an Implementation Operation

When you create a realization link between a class and an interface, the class must implement all the operations of the interface.

Note: To automatically create the necessary operations in your class, click **Tools > Model Options** to open the Model Options dialog, and select the **Auto-Implement Realized Interfaces** option. If this option is not selected, you can implement the operations manually.

1. Open the property sheet of a class that is linked to one or more interfaces by realization links and click the **Operations** tab.

2. Click the **To be Implemented** button to open the To Be Implemented Operations window, which lists all the operations waiting to be implemented in the class. You can use the Properties button to display the property sheet of an operation.

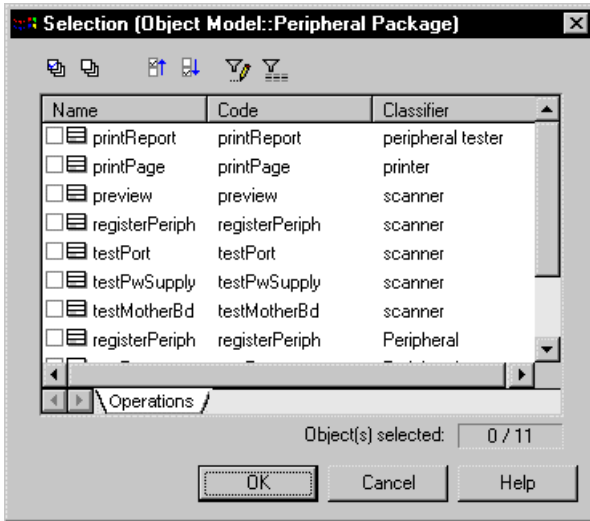


3. Select one or more operations, and then click the **Implement** button to copy them to the list of operations of the class with their stereotype set to <<Implement>>. Each implemented operation has the same signature (name and parameters) as the original operation, and you can only modify the code on its **Implementation** tab.
4. Click **Close** to return to the **Operations** tab, and then click **OK** to return to your model.

Copying an Operation to Another Class

You can copy an operation from one class and add it to another. If the class already contains an operation with the same name or code as the copied operation, the copied operation is renamed. For example the operation testPort is renamed testPort2 when it is copied to a class which already contains an operation testPort.

1. Double-click a class in the diagram to display its property sheet, and then click the Operations tab.
2. Click the Add Operations tool to open a Selection window. This window lists all the operations attached to all the classes in the model.



3. Select one or more operations in the list and then click OK to add the copied operations to the list of operations.
4. Click OK.

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.
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 Code 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 available by default:</p> <ul style="list-style-type: none"> • <<constructor>> - Operation called during the instantiation of an object that creates an instance of a class • <<create>> - Operation used by a class when instantiating an object • <<destroy>> - Operation used by a class that destroys an instance of a class • <<storedProcedure>> - Operation will become a stored procedure in the generated PDM • <<storedFunction>> - Operation will become a stored function in the generated PDM • <<EJBCreateMethod>> - EJB specific CreateMethod • <<EJBFinderMethod>> - EJB specific FinderMethod • <<EJBSelectMethod>> - EJB specific SelectMethod <p>For more information on EJB specific methods, see <i>Defining Operations for EJBs</i> on page 363.</p>
Return Type	A list of values returned by a call of the operation. If none are returned, the return type value is null
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"> • Private - Only to the class to which it belongs • Protected - Only to the class and its derived objects • Package - To all objects contained within the same package • Public - To all objects
Language event	When classes represent elements of interfaces, this box allows you to show an operation as triggered by a significant occurrence of an event
Static	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
Array	Flag defining the return type of the operation. It is true if the value returned is a table
Abstract	The operation cannot be instantiated and thus has no direct instances
Final	The operation cannot be redefined

Property	Description
Read-only	Operation whose execution does not change the class instance
Web service method	If displayed and selected, implies that the operation is used as a web service method
Influent object	Operation that influences the current operation. The most common influence links are "overrides" or "implements" as explained below
Overrides	Indicates which parent operation the current operation is overriding through a generalization link
Implements	Name of the interface operation the current operation is implementing though a realization link
Generic	Specifies that the operation is a generic method (see <i>Generic Types and Methods</i> on page 42).
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	Specifies the name of the item, which should be clear and meaningful, and should convey the item's purpose to non-technical users.
Code	Specifies the technical name of the object, which is used for generating code or scripts.
Comment	Descriptive comment for the object.
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"> • In - Input parameter passed by value. The final value may not be modified and information is not available to the caller • In/Out - Input parameter that may be modified. The final value may be modified to communicate information to the caller • Out - Output parameter. The final value may be modified to communicate information to the caller
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)

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 Add Exception tool to select an exception classifier to add at the cursor position.
Pre-condition	A constraint that must be true when an operation is invoked.
Post-condition	A constraint that must be true at the completion of an operation.
Specification	Similar to the pseudo code, it is a description of the normal sequence of actions.

The following tabs are also available:

- Generic - lets you specify the type parameters of a generic method (see *Generic Types and Methods* on page 42)

- Related Diagrams - lists and lets you add model diagrams that are related to the operation (see *Core Features Guide > The PowerDesigner Interface > Diagrams, Matrices, and Symbols > Diagrams > Specifying Diagrams as Related Diagrams*).

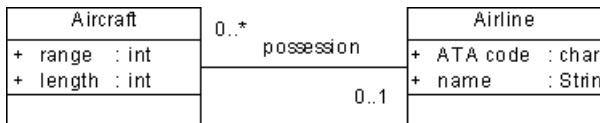
Associations (OOM)

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

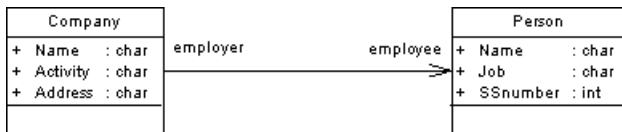
An association can be created in the following diagrams:

- Class Diagram
- Composite Structure Diagram

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



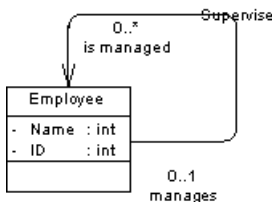
In addition to naming the association itself, you can specify a role name for each end in order to describe the function of a class as viewed by the opposite class. For example, a person considers the company where he works as an employer, and the company considers this person as an employee.



Reflexive Association

A reflexive association is an association between a class and itself.

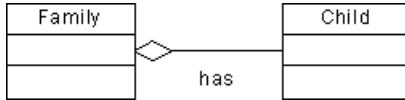
In the following example, the association Supervise expresses the fact that an employee can, at the same time, be a manager and someone to manage.



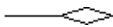
In the **Dependencies** tab of the class, you can see two identical occurrences of the association, this is to indicate that the association is reflexive and serves as origin and destination for the link.

Aggregation

An *aggregation* is a special type of 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 the following example, the family is the whole that can contain children.

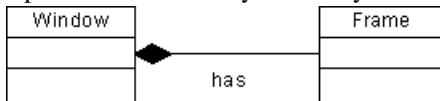


You can create an aggregation directly using the **Aggregation** tool in the Toolbox. The aggregation symbol in a diagram is the following:



Composition

A *composition* is a special type of 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 the following example, the frame is a part of a window. If you destroy the window object, the frame part also disappears.



You can create a composition directly using the **Composition** tool in the Toolbox. The composition symbol in a diagram is the following:



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.

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 > The PowerDesigner Interface > Objects*.

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 Code 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 class.
Type	Specifies the type of association. You can choose between: <ul style="list-style-type: none">• Association• Aggregation – a part-whole relationship between a class and an aggregate class• Composition – a form of aggregation but with strong ownership and coincident lifetime of parts by the whole
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
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"> • Private – only to the object itself • Protected – only to the object and its inherited objects • Package – to all objects contained within the same package • Public – to all objects (option by default)
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"> • 0..1 – zero or one • 0..* – zero to unlimited • 1..1 – exactly one • 1..* – one to unlimited • * – none to unlimited <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 Profile\Association\ExtendedAttributes category, under the roleAContainer and roleBContainer 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 > 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"> • Changeable – Associations may be added, removed, and changed freely • Read-only – You are not allowed to modify the association • Frozen – Constant association • Add-only – New associations may be added from a class on the opposite end of the association

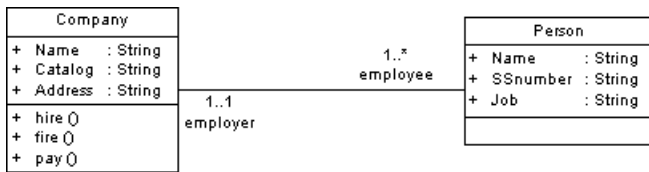
Property	Description
Ordering	The association is included in the ordering which sorts the list of associations by their order of creation. You can choose between: <ul style="list-style-type: none"> Sorted – The set of objects at the end of an association is arranged according to the way they are defined in the model Ordered – The set of objects at the end of an association is arranged in a specific order Unordered – The end of an association is neither sorted nor ordered
Initial value	Specifies an instruction for initializing migrated attributes, for example 'new client ()'.
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 <i>Association Implementation</i> on page 92).
Migrated attribute	Specifies the name of the migrated association role.

Association Implementation

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

When an association end is navigable, it means you want to be able to retrieve the instance of the class it is linked to, this instance is displayed as a *migrated attribute* in the current instance of a class. The rolename of this end can be used to clarify the structure used to represent the link.

For example, let us consider an 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.

Default Implementation

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;
}
  
```

Depending on the language you are working with, you may have other ways to implement migrated attributes. PowerDesigner lets you choose an implementation in the Detail tab of the association property sheet.

Container Type and Implementation Class

A *container* is a collection type of objects that stores elements. The container structure is more complex than the array type and 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. This type will be used by migrated attributes. The code of a migrated attribute that uses a container type contains getter and setter functions used to define the implementation of the association. These functions are visible in the Code Preview tab, but do not appear in the list of operations.

When you use the role migration feature to visualize migrated attributes and select a container type, 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.

More on Implementation Class

The default implementation mechanism is defined in the object language resource file under the Profile\Association category. This mechanism 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 can use the resource editor interface to modify implementation details.

For more information on template definition, see *Customizing and Extending PowerDesigner > Customizing Generation with GTL*

Understanding the Generated Code

When you define an implementation class, association implementation details are always generated in the origin and/or destination classes of the navigable association.

Generation uses specific documentation comment tags to store association information. These documentation comment tags gather all the required details to be able to recreate the association upon reverse engineering. The documentation comment tags are processed during reverse engineering in order to make round-trip engineering possible.

The following documentation tags are used:

- *pdRoleInfo* is used to retrieve the classifier name, container type, implementation class, multiplicity and type of the association
- *pdGenerated* is used to flag automatically generated functions linked to association implementation. These functions should not be reverse engineered otherwise generated and reverse engineered models will be different

Warning! Make sure you do not modify these tags in order to preserve round-trip engineering.

In Java

The javadoc tag syntax is used */**@tag value*/*.

In the following example, the tag @pdRoleInfo is used to store association implementation details, and @pdGenerated is used to indicate that the getter method is automatically generated and should not be reverse engineered.

```
/**@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;
}
...

```

In C#

The documentation tag `///<tag value />` is used.

In the following example, the tag `<pdRoleInfo>` is used to store association implementation details, and `<pdGenerated>` is used to indicate that the getter method is automatically generated and should not be reverse engineered.

```

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

```

In VB .NET

The documentation tag `"<tag value />"` is used.

In the following example, the tag `<pdRoleInfo>` is used to store association implementation details, and `<pdGenerated>` is used to indicate that the getter method is automatically generated and should not be reverse engineered.

```

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

```

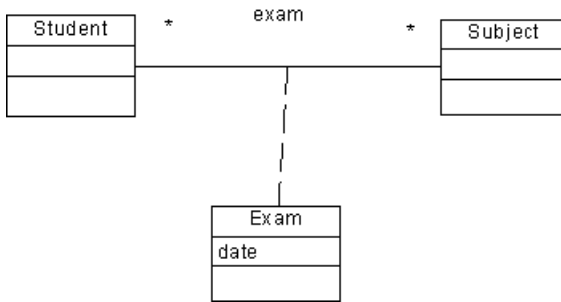
Creating an Association Class

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.

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.



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.

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

Migrating Association Roles in a Class Diagram

You can migrate association roles and create attributes before generation. This is very convenient for many reasons including data type customization and the ability to change the attribute order in the list of attributes. The last feature is especially important in XML.

Regardless of the navigability, the migration creates an attribute and sets its properties as follows:

- Name and code of the attribute: association role if already set, if not the association name
- Data type: code of the classifier linked by the association
- Multiplicity: role multiplicity
- Visibility: role visibility

Migration Rules

The following rules apply when migrating association roles:

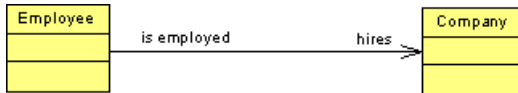
- If the migrated attribute name is the same as the role name, then modifying the role name synchronizes the migrated attribute name
- If the migrated attribute data type is the same as the role classifier, then modifying the role multiplicity synchronizes the migrated attribute multiplicity
- If the code of the classifier, linked by the association, changes, then the migrated attribute data type is automatically synchronized
- If you manually change the migrated attribute, the synchronization does not work, the association role is not synchronized
- The migrated attribute is automatically deleted if the association is deleted

After migration, the property sheet of the new attribute displays the name of the association in the Migrated from box in the Detail tab.

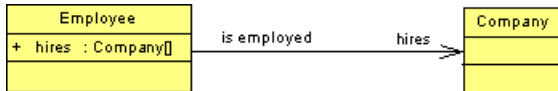
Migrating Navigable Roles

You can migrate the navigable role of an association and turn it into an attribute:

1. Right-click the association in the diagram.



2. Select **Migrate > Migrate Navigable Roles** from the association contextual menu.



An attribute is created and named after the navigable role of the association followed by the code of the classifier.

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.

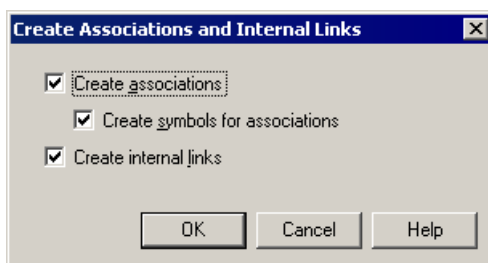
The Rebuild Data Type Links scans the following data types:

- 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:

1. Select **Tools > Rebuild Data Type Links** 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
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.

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.

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.

A generalization can be created in the following diagrams:

- Class Diagram
- Composite Structure Diagram
- Component Diagram
- Use Case Diagram

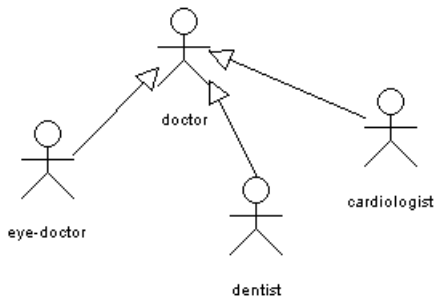
You create a generalization relationship when several objects have common behaviors. You can also create a generalization between a shortcut of an object and an object but, if the link is oriented, only the parent object can be the shortcut.

Generalizations in a Use Case Diagram

In a use case diagram, you can create a generalization between:

- Two actors
- Two use cases

For example two or more actors may have similarities, and communicate with the same set of use cases in the same way. This similarity is expressed with generalization to another actor. In this case, the child actors inherit the roles, and relationships to use cases held by the parent actor. A child actor includes the attributes and operations of its parent.



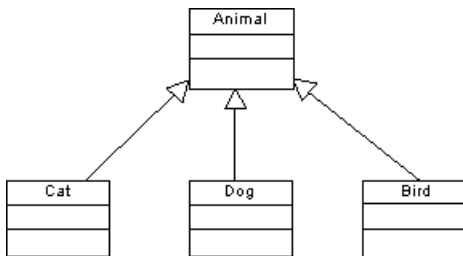
Generalizations in a Class or Composite Structure Diagram

In a class diagram, you can create a generalization between:

- Two classes
- Two interfaces

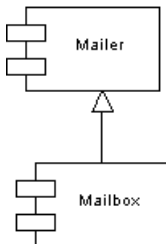
For example, an animal is a more general concept than a cat, a dog or a bird. Inversely, a cat is a more specific concept than an animal.

Animal is a super class. Cat, Dog and Bird are sub-classes of the super class.



Generalizations in a Component Diagram

In a component diagram, you can create a generalization between two components, as shown below:



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 > The PowerDesigner Interface > Objects*.

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 Code field.
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.
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">• Private – only to the generalization itself• Protected – only to the generalization and its inherited objects• Package – to all objects contained within the same package• Public – to all objects (option by default)

Property	Description
Generate parent class as table	Selects the "Generate table" persistence option in the Detail tab of the parent class property sheet. If this option is not selected, the "Migrate columns" persistence option of the parent class is selected.
Generate child class as table	Selects the "Generate table" persistence option in the Detail tab of the child class property sheet. If this option is not selected, the "Migrate columns" 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 New 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.

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).

A dependency can be created in the following diagrams:

- Class Diagram
- Composite Structure Diagram
- Object Diagram
- Use Case Diagram
- Component Diagram
- Deployment Diagram

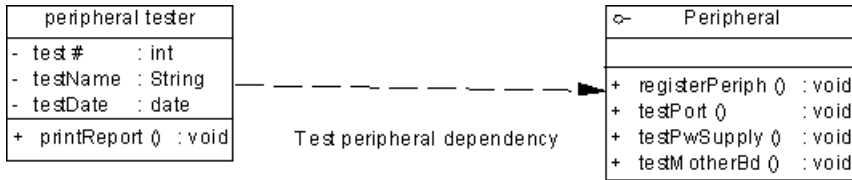
The dependency relationship indicates that one object in a diagram uses the services or facilities of another object. You can also define dependencies between a package and a modeling element.

Dependencies in a Class or Composite Structure Diagram

In a class diagram, you can create a dependency between:

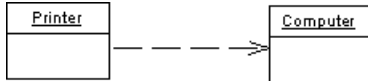
- A class and an interface (and vice versa)
- Two classes
- Two interfaces

For example:



Dependencies in an Object Diagram

In an object diagram, you can create a dependency between two objects as follows:

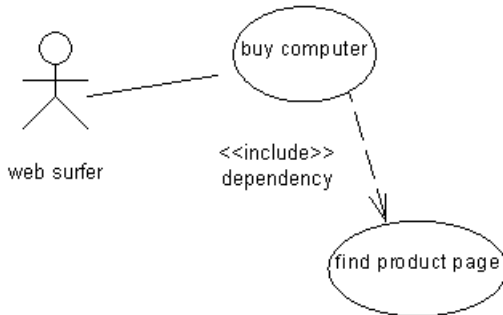


Dependencies in a Use Case Diagram

In a use case diagram, you can create a dependency between:

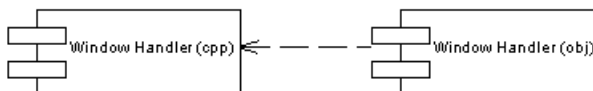
- An actor and a use case (and vice versa)
- Two actors
- Two use cases

Buying a computer from a web site involves the activity of finding the product page within the seller's web site:



Dependencies in a Component Diagram

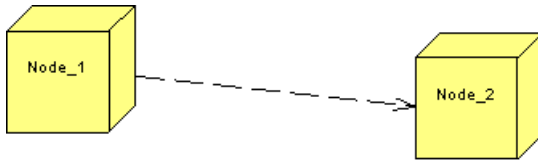
In a component diagram, you can create a dependency between two components as shown below. You cannot create a dependency between a component and an interface.



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

Dependencies in a Deployment Diagram

In a deployment diagram, a dependency can be created between nodes, and component instances as follows:



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 > The PowerDesigner Interface > Objects*.

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 Code field.
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.

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"> • << Access >> - Public contents of the target package that can be accessed by the source package • << Bind >> - Source object that instantiates the target template using the given actual parameters • << Call >> - Source operation that invokes the target operation • << Derive >> - Source object that can be computed from the target • << Extend >> - (Use Case/Class) Target object extends the behavior of the source object at the given extension point • << Friend >> - Source object that has special visibility towards the target • << Import >> - Everything declared public in the target object becomes visible to the source object, as if it were part of the source object definition • << Include >> - (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 • << Instantiate >> - Operations on the source class create instances of the target class • << Refine >> - The target object has a greater level of detail than the source object • << Trace >> - Historical link between the source object and the target object • << Use >> - Semantics of the source object are dependent on the semantics of the public part of the target object • << ejb-ref >> - (Java only) Used in Java Generation to create references to EJBs (entity beans and session beans) for generating the deployment descriptor • << sameFile >> - (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
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.

Realizations (OOM)

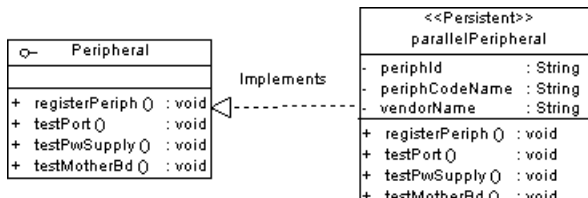
A realization is a relationship between a class or component and an interface.

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. You can also create a realization between a shortcut of an interface and a class. Whenever the link is oriented, only the parent object can be the shortcut.

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



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 > The PowerDesigner Interface > Objects*.

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 Code 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 interface.
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.

Require Links (OOM)

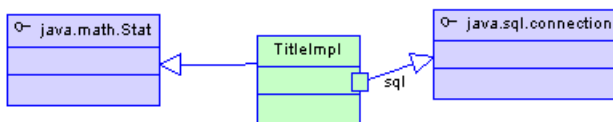
Require links connect classifiers and interfaces. A *require link* can connect a class, a component, or a port on the outside of one of these classifiers to an interface.

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



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 > The PowerDesigner Interface > Objects*.

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 Code 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 interface.
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.

Assembly Connectors (OOM)

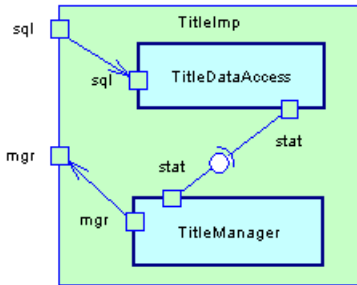
Assembly connectors represent the paths of communication by which parts in your classifiers request and provide services to each other.

An assembly connector can be created in the following diagrams:

- Composite Structure Diagram
- Component Diagram

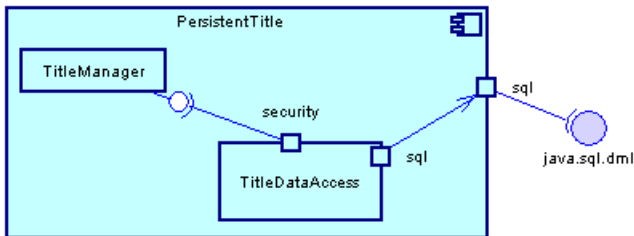
Assembly Connectors in a Composite Structure Diagram

In the example below an assembly connector connects the supplier part "TitleDataAccess" to the client part "TitleManager".



Assembly Connectors in a Component Diagram

In the example below an assembly connector connects the supplier part "TitleDataAccess" to the client part "TitleManager".



Creating an Assembly Connector

You can create assembly connector using the **Require Link/Connector** tool in the Toolbox.

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

Assembly Connector Properties

To view or edit an assembly connector'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 Code field.
Client	Specifies the part requesting the service. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected part.
Supplier	Specifies the part providing the service. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected part.
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.
Interface	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 interface.
Keywords	Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.

Delegation Connectors (OOM)

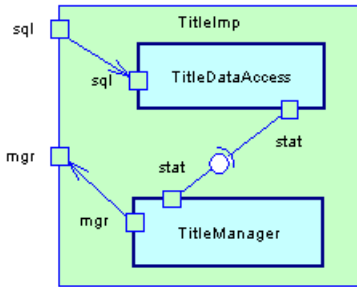
Delegation connectors represent the paths of communication by which parts inside a classifier connect to ports on the outside of that classifier and request and provide services to each other.

A delegation connector can be created in the following diagrams:

- Composite Structure Diagram
- Component Diagram

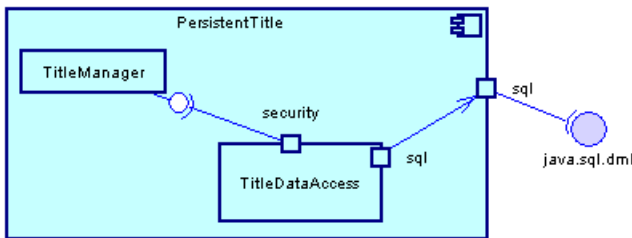
Delegation Connectors in a Composite Structure Diagram

In the example below 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".



Delegation Connectors in a Component Diagram

In the example below 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 a Delegation Connector

You can create a delegation connector using the **Require Link/Connector** tool in the Toolbox.

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

Delegation Connector Properties

To view or edit a delegation connector'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 Code field.

Property	Description
Client	Specifies 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.
Supplier	Specifies the part or port providing the service. Use the tools to the right of the list to create, browse for, or view the properties of the currently selected 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.
Keywords	Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.

Annotations (OOM)

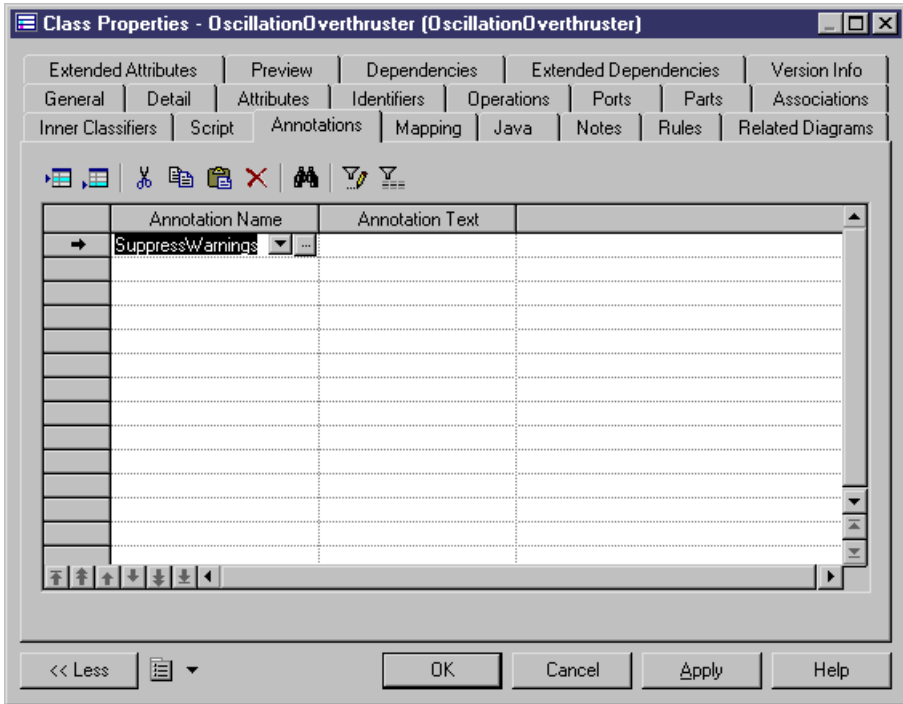
Java 5.0 and the .NET languages (C# 2.0 and VB 2005) provide methods for adding metadata to code. PowerDesigner provides full support for Java annotations and .NET custom attributes. For language-specific information, see the relevant chapter in Part Two of this book.

This metadata can be accessed by post-processing tools or at run-time to vary to behavior of the system.

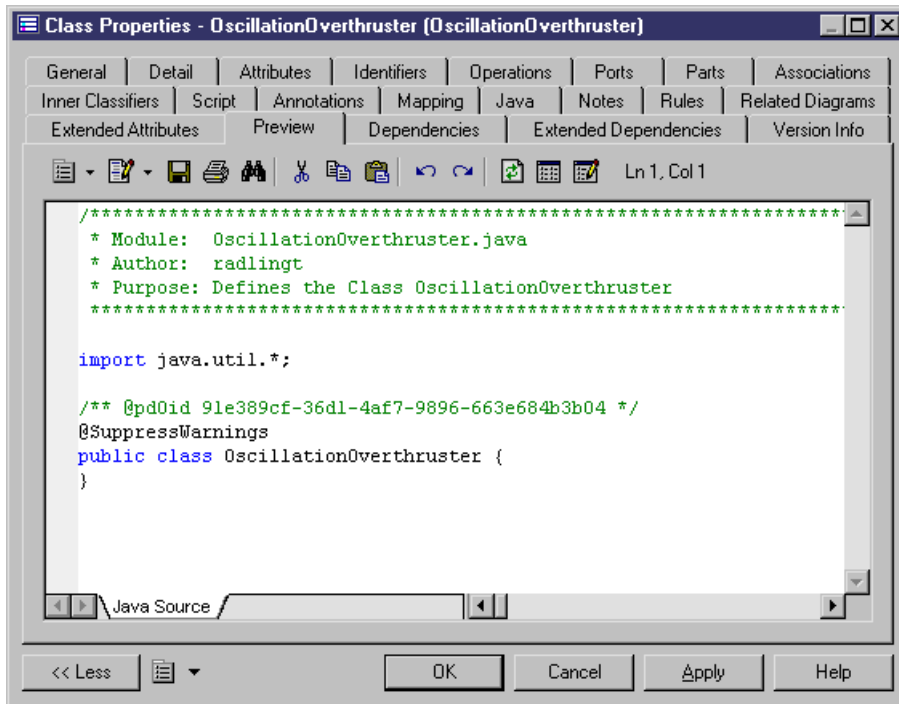
Attaching an Annotation to a Model Object

PowerDesigner supports the Java 5.0 built-in annotations, the .NET 2.0 built-in custom attributes and for both Java 5.0 and .NET 2.0, also allows you to create your own. You can attach annotations to types and other model objects:

1. Double-click a class or other object to open its property sheet, and then click the Annotations tab.
2. Click in the Annotation Name column, and select an annotation from the list.



3. 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.
4. [optional] Click the Preview tab to see the code that will be generated for the class, with its declaration preceded by the annotation:

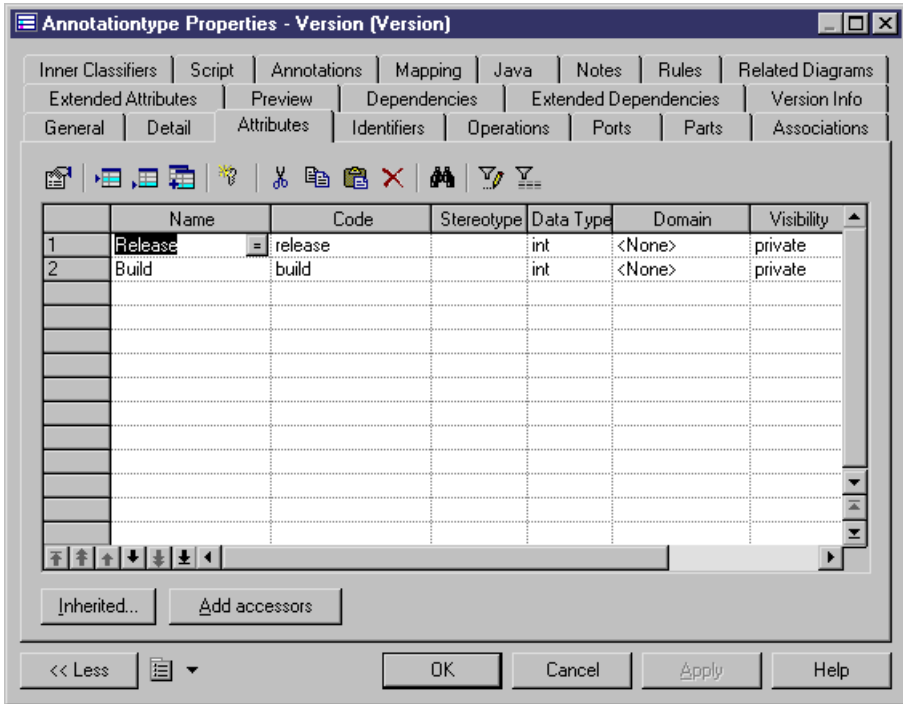


5. Click OK to return to the diagram.

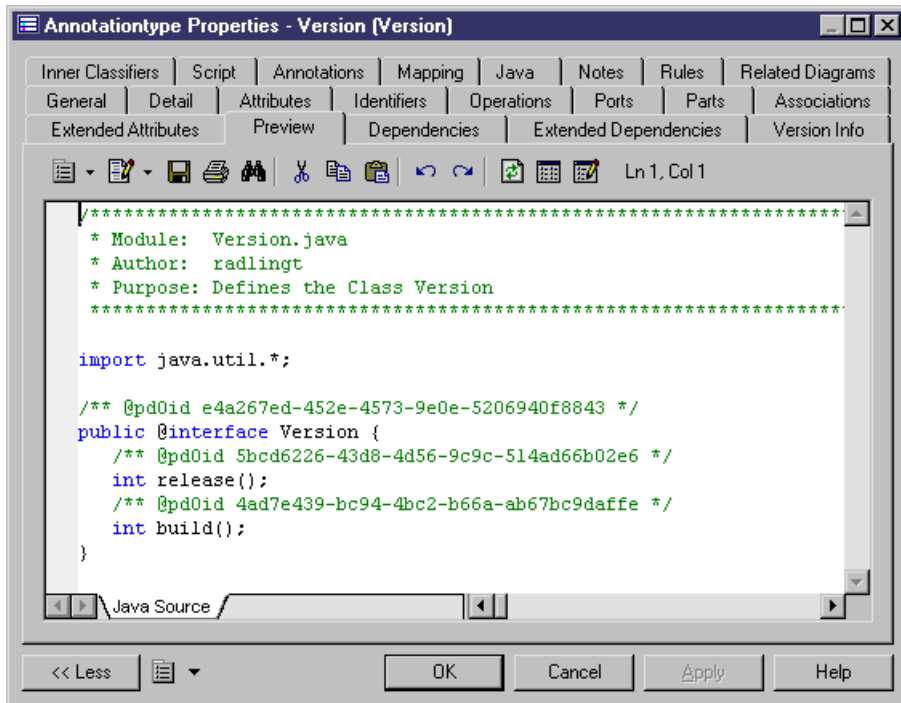
Creating a New Annotation Type

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

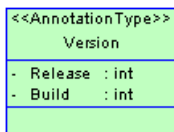
1. Create a class and then double-click it to access its property sheet.
2. On the General tab, in the Stereotype list:
 - For Java 5.0 - select AnnotationType
 - For .NET 2.0 - select AttributeType
3. Click the Attributes tab, and add an attribute for each parameter accepted by the annotation type.



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

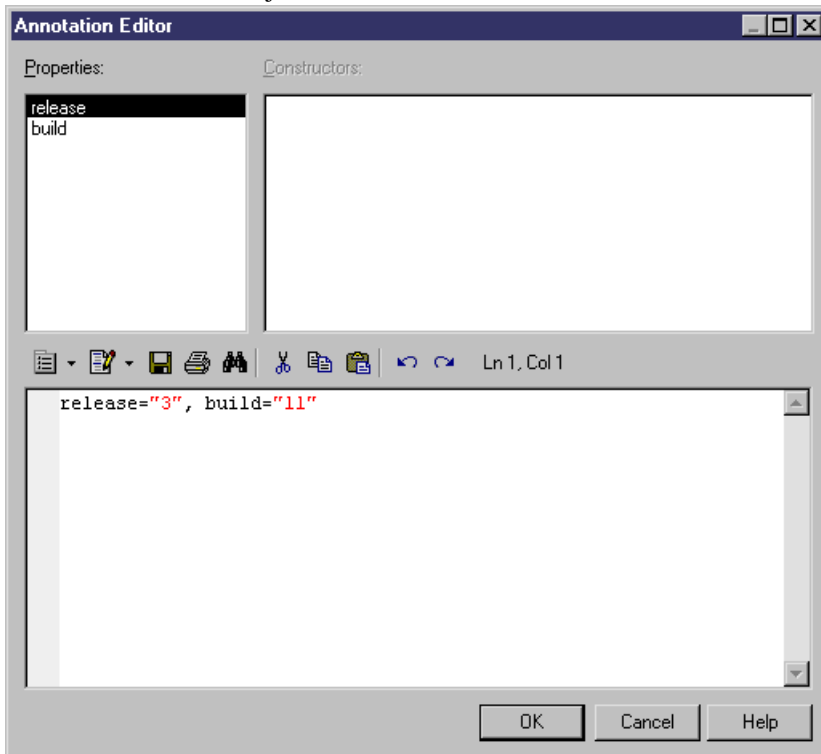


5. Click OK to return to the diagram. The annotation type will be represented as follows:



Using the Annotation Editor

The Annotation Editor assists you in specifying the values for annotation parameters. It is accessible by clicking the ellipsis button in the Annotation Text Column on the Annotations tab of a class or other object:



The top panes provide lists of available properties and constructors, which you can double-click to add them to the bottom, editing pane.

Instance Links (OOM)

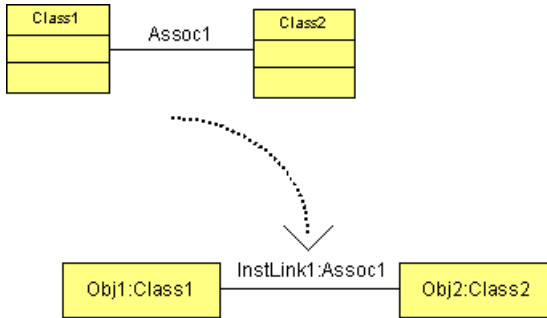
An *instance link* represents a connection between two objects. It is drawn as a solid line between two objects.

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.

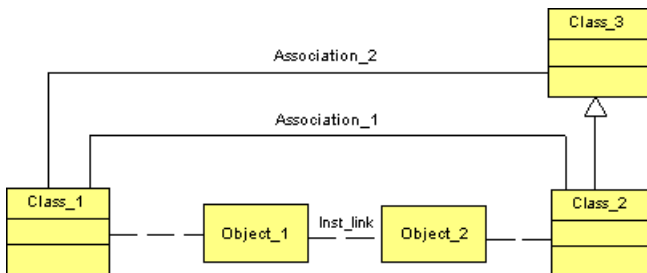


The roles of the instance link are duplicated from the roles of the association. An instance link can therefore be an aggregation or a composition, exactly like an association of the class diagram. If it is the case, the composition or aggregation symbol is displayed on the instance link symbol. The 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:

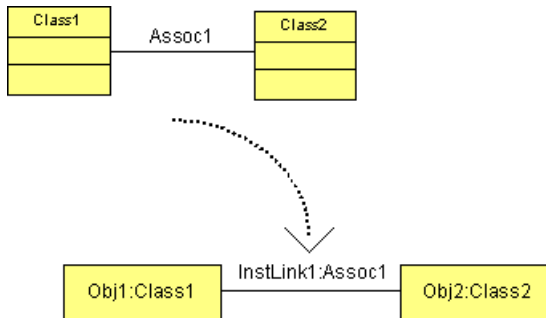
- When an association between classes becomes an instance link, both classes linked by the association, and both classes of the objects linked by the instance link must match (or the class of the object must inherit from the parent classes linked by the association). This is also valid for an association 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:

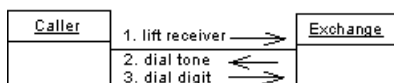
- Two objects
- An object and an actor (and vice versa)

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)* on page 137.

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

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 > The PowerDesigner Interface > 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.

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

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

Creating a Domain

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

- Select **Model > Domains** 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 > Domains**

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

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 Code 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"> • 0..1 – zero or one • 0..* – zero to unlimited • 1..1 – exactly one • 1..* – one to unlimited • * – none to unlimited

Property	Description
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 <i>Managing Object Persistence During OOM to PDM Generation</i> on page 286).
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* on page 70)
- 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 > The PowerDesigner Interface > 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	—	—

Data Type	Content	Length	Mandatory Precision
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	—	—
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	—

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:

- Data type
- Check parameters
- Business rules

1. Select **Model > Domains** 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.

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* on page 125.
- 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* on page 127.
- 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* on page 131.
- 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* on page 133.
- An *interaction overview diagram* provides a high level view of the interactions that occur in your system. For more information, see *Interaction Overview Diagrams* on page 136.

Communication Diagrams

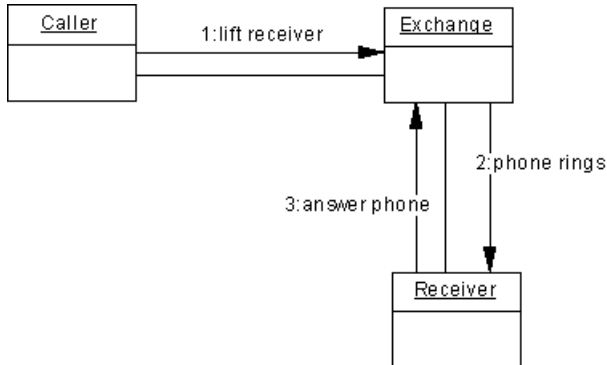
A *communication diagram* is a UML diagram that provides a graphical view of the interactions between objects for a use case scenario, the execution of an operation, or an interaction between classes, with an emphasis on the system structure.

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




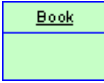




Building a communication diagram can also be the opportunity to test a static model at the conception level; it may represent a scenario in which classes from the class diagram are instantiated to create the objects necessary to run the scenario.

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 an empty communication diagram. Then you have simply to add the necessary messages.

Communication Diagram Objects

PowerDesigner supports all the objects necessary to build communication diagrams.

Object	Tool	Symbol	Description
Actor		 Client	An external person, process or something interacting with a system, sub-system or class. See <i>Actors (OOM)</i> on page 20.
Object		 Book	Instance of a class. See <i>Objects (OOM)</i> on page 56.
Instance link			Communication link between two objects. See <i>Instance Links (OOM)</i> on page 116.
Message			Interaction that conveys information with the expectation that action will ensue. It creates an instance link by default when no one exists. See <i>Messages (OOM)</i> on page 137.

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.

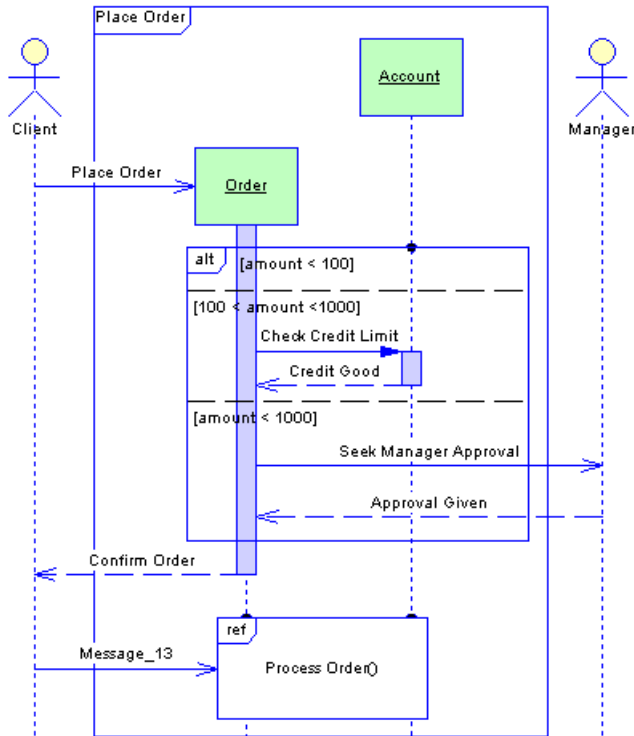
You can use one or more sequence diagrams to enact a use case or to identify all the possibilities of a complex behavior.

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* on page 147). 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 > The PowerDesigner Interface > 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.




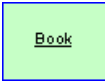



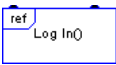

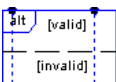






A sequence diagram allows you to analyze class operations more closely than a communication diagram. You can create an operation in the class of an object that receives a message through the property sheet of the message. This can also be done in a communication diagram, but there is more space in a 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.

Sequence Diagram Objects

PowerDesigner supports all the objects necessary to build sequence diagrams.

Object	Tool	Symbol	Description
Actor			An external person, process or something interacting with a system, sub-system or class. See <i>Actors (OOM)</i> on page 20.
Object			Instance of a class. See <i>Objects (OOM)</i> on page 56.
Activation			Execution of a procedure, including the time it waits for nested procedures to execute. See <i>Activations (OOM)</i> on page 151.
Interaction Reference			Reference to another sequence diagram. See <i>Interaction References and Interaction Activities (OOM)</i> on page 155.
Interaction Fragment			Collection of associated messages. See <i>Interaction Fragments (OOM)</i> on page 157.
Message			Communication that conveys information with the expectation that action will ensue. See <i>Messages (OOM)</i> on page 137.
Self Message			Recursive message: the sender and the receiver are the same object. See <i>Messages (OOM)</i> on page 137.
Procedure Call Message			Procedure call message with a default activation. See <i>Messages (OOM)</i> on page 137.

Object	Tool	Symbol	Description
Self Call Message			Procedure call recursive message with a default activation. See <i>Messages (OOM)</i> on page 137.
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 <i>Messages (OOM)</i> on page 137.
Self Return Message			Recursive message with a Return control flow type. See <i>Messages (OOM)</i> on page 137.

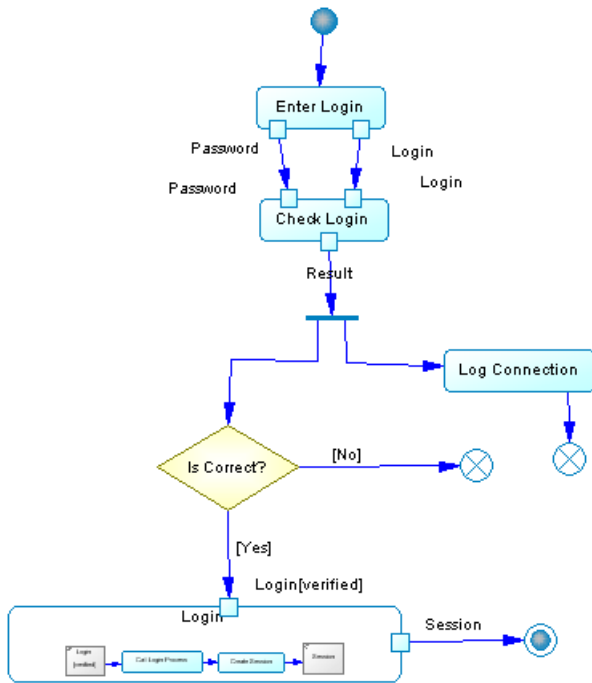
Activity Diagrams

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.

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.




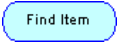


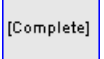










Analyzing a Business Process

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 Modeling*.

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 <i>Starts (OOM)</i> on page 182.
Activity			Invocation of an action. See <i>Activities (OOM)</i> on page 160.
Composite activity	N/A		Complex activity decomposed to be further detailed. See <i>Activities (OOM)</i> on page 160.
Object node			A specific state of an activity. See <i>Object Nodes (OOM)</i> on page 192.
Organization unit			A company, a system, a service, an organization, a user or a role. See <i>Organization Units (OOM)</i> on page 173.
Flow			Path of the control flow between activities. See <i>Flows (OOM)</i> on page 190.
Decision			Decision the control flow has to take when several flow paths are possible. See <i>Decisions (OOM)</i> on page 185.
Synchronization			Enables the splitting or synchronization of control between two or more concurrent actions. See <i>Synchronizations (OOM)</i> on page 188.
End			Termination point of the activities described in the activity diagram. See <i>Ends (OOM)</i> on page 183.

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.

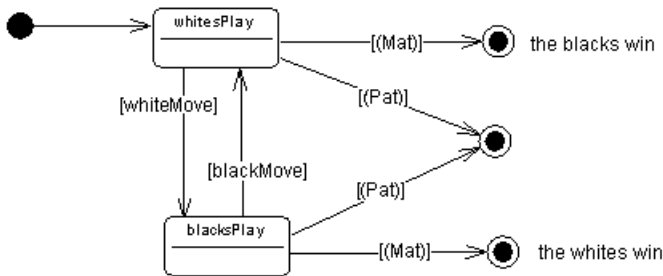
Note: To create a statechart diagram in an existing OOM, right-click the model in the Browser and select **New > Statechart Diagram**. To create a new model, select **File > New Model**, choose Object Oriented Model as the model type and **Statechart Diagram** as the first diagram, and then click **OK**.

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.

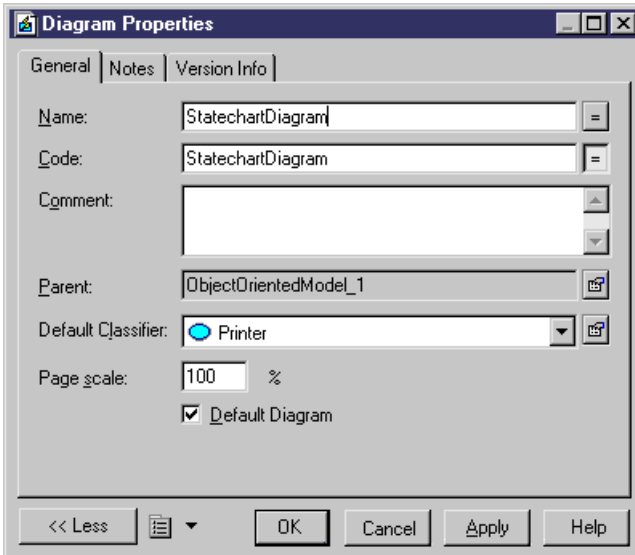
You can also use the transitions identified to establish the order in which operations can be invoked. This type of diagram is called a "protocol state machine".

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.

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.

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 <i>Starts (OOM)</i> on page 182.
State			The situation of a model element waiting for events. See <i>States (OOM)</i> on page 194.
Action	N/A	N/A	Specification of a computable statement. See <i>Actions (OOM)</i> on page 205.

Object	Tool	Symbol	Description
Event	N/A	N/A	Occurrence of something observable, it conveys information specified by parameters. See <i>Events (OOM)</i> on page 202.
Transition			Path on which the control flow moves between states. See <i>Transitions (OOM)</i> on page 200.
Junction point			Divides a transition between states. Used particularly when specifying mutually exclusive conditions. See <i>Junction Points (OOM)</i> on page 209.
Synchronization			Enables the splitting or synchronization of control between two or more concurrent states. See <i>Synchronizations (OOM)</i> on page 188.
End			Termination point of the states described in the state-chart diagram. See <i>Ends (OOM)</i> on page 183.

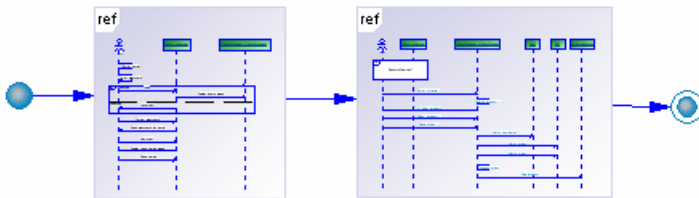
Interaction Overview Diagrams

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.

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**.




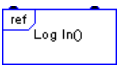








You can include references to sequence diagrams, communication diagrams, and other interaction diagrams.

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



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 <i>Starts (OOM)</i> on page 182.
Interaction activity			Reference to a sequence diagram, communication diagram, or interaction overview diagram. See <i>Interaction References and Interaction Activities (OOM)</i> on page 155.
Flow			Flow of control between two interactions. See <i>Flows (OOM)</i> on page 190.
Decision			Decision the flow has to take when several paths are possible. See <i>Decisions (OOM)</i> on page 185.
Synchronization			Enables the splitting or synchronization of control between two or more flows. See <i>Synchronizations (OOM)</i> on page 188.
End			Termination point of the interactions described in the diagram. See <i>Ends (OOM)</i> on page 183.

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 *Sequence numbers* on page 149)
- 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.

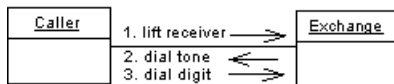
The sequence number attached to a message is identical in all diagrams if the message is reused.

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.

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.

Creating a Message

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

- Use the **Message** tool (or, in a sequence diagram, the **Self Message**, **Procedure Call Message**, or **Self Call Message** tool) in the Toolbox.
- 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 > The PowerDesigner Interface > Objects*.

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 Code field.

Property	Description
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
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, or to reverse the direction of the message. Note: You can right-click a message in the diagram and select Reverse to reverse its direction. You cannot reverse the direction of a Create or Destroy message.
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	Specifies the type of message action. You can choose between: <ul style="list-style-type: none"> • 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. • 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. • 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.

Property	Description
Control flow	<p>Specifies the mode in which messages are sent. You can choose between:</p> <ul style="list-style-type: none"> Asynchronous – the sending object does not wait for a result, it can do something else in parallel. <i>No-wait semantics</i> 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. <i>Wait semantics</i> Return – Generally associated with a Procedure Call. The Return arrow may be omitted as it is implicit at the end of an activation Undefined – No control flow defined
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
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]

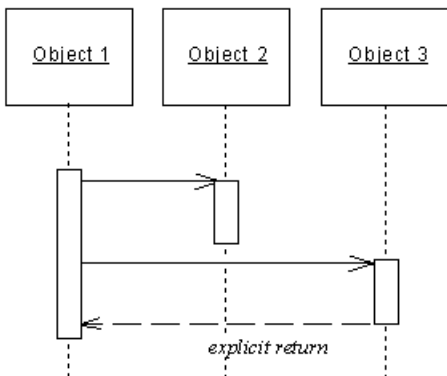
Property	Description
Begin time	User-defined time alias, used for defining constraints. Example: Begin time = t1, End time = t2, constraint = {t2 - t1 < 30 sec}
End time	User-defined time alias, used for defining constraints.
Support delay	<p>Specifies that the message may have duration. The message 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

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.

Creating Create and Destroy Messages in a Sequence Diagram

Create and Destroy messages are specified via the Action property on the Detail tab of their property sheet.

Creating Create Messages

A message can create an object if it is the first message received by the object and you set its Action property to "Create".

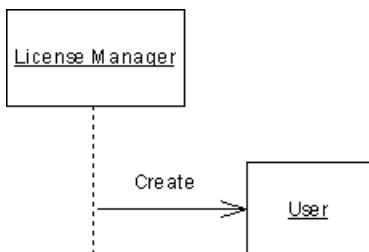
You cannot create an actor or use the create action with a recursive message.

When a message creates an object in a sequence diagram, the message is drawn with its arrowhead on the object; both object and message are at the same level.

In a sequence diagram, you can also create a create message automatically as follows:

1. Click the **Message** tool in the Toolbox.
2. Click the lifeline of the sender object or actor and, while holding down the mouse button, drag the cursor towards the receiver object.
3. Release the mouse button on the object symbol of the receiver (and not on its lifeline).

If the message is the first message to be received by the receiver object, the receiver object symbol moves down to line up with the create message.



Creating Destroy Messages

A message can destroy an object if it is the last message received by the object and you set its Action property to "Destroy".

You cannot destroy an actor or use the destroy action with a recursive message.

A destroyed object lifeline is marked by an X at the intersection point between the object lifeline and the message. The Destroy action ends both activation and object lifeline at this same point.

Note that the Destroy action does not destroy the object, but only represents this destruction in the diagram. The object lifeline ends at a precise point in time; it is not possible to graphically pull the lifeline downwards any more.

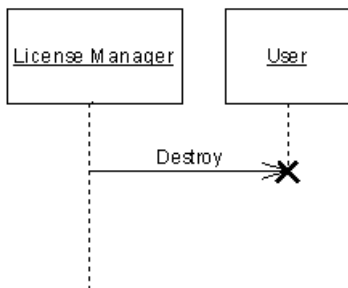
There are two possible forms of destroy messages:

- Destroy message
- Self-Destroy message

Creating a Destroy Message

You can create a destroy message from the Toolbox.

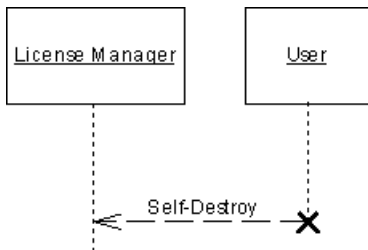
1. Click the **Message** tool in the Toolbox.
2. Click the lifeline of the sender object or actor and, while holding down the mouse button, drag the cursor towards the lifeline of the receiver object.
3. Release the mouse button on the lifeline of the receiver, and then double-click the newly created message symbol to display its property sheet.
4. Select Destroy from the Action list in the **Detail** tab (this action is not available if the message is not the last message on the receiver lifeline).
5. Click **OK**. An X is placed at the intersection point between the Destroy message arrow and the receiver object lifeline.



Creating a Self-Destroy Message

You create a self-destroy message from the Toolbox.

1. Click the **Message** tool in the Toolbox.
2. Click the lifeline of the sender object and, while holding down the mouse button, drag the cursor towards the lifeline of the receiver object.
3. Release the mouse button on the object symbol of the receiver, and then double-click the newly created message symbol to display its property sheet.
4. Select Self-Destroy from the Action list in the **Detail** tab.
5. Select Return from the Control flow list.
6. Click OK. The lifeline of the self-destroyed object is marked by an X.



Creating a Recursive Message in a Sequence Diagram

A message is recursive when the object sends the message to itself. In this case, the arrow starts and finishes on the lifeline of the same object.

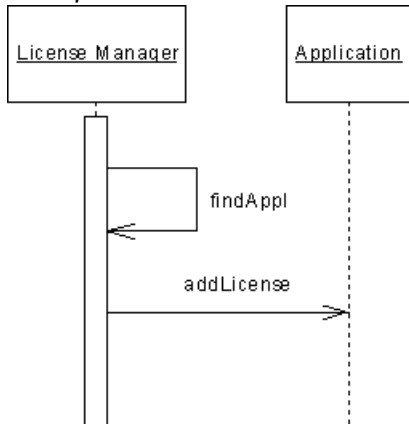
The Create and Self-Destroy actions, and the Support delay option are not available with a recursive message.

When you create Undefined or Return recursive messages from the Toolbox, the control flow value is already selected:

Message type	Symbol
Undefined recursive message	↻
Return recursive message	↺

You can also create an Undefined recursive message and change the control flow value afterwards.

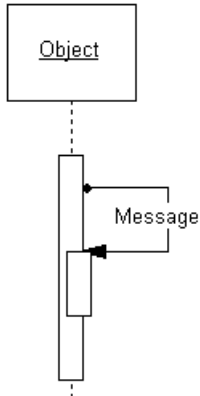
Example



You can choose to create a recursive message with or without activation from the Toolbox.

When you create a recursive message with activation, the recursive message is automatically attached to an activation and its control flow value is a Procedure Call which, by default, starts the activation.

Activation symbols are automatically created on the object lifeline as shown below:



Creating a Recursive Message Without Activation

You can create a recursive message without activation from the Toolbox.

1. Click the **Self Message** tool in the Toolbox.
2. Click the object lifeline to create a recursive message.

Creating a Recursive Message with Activation

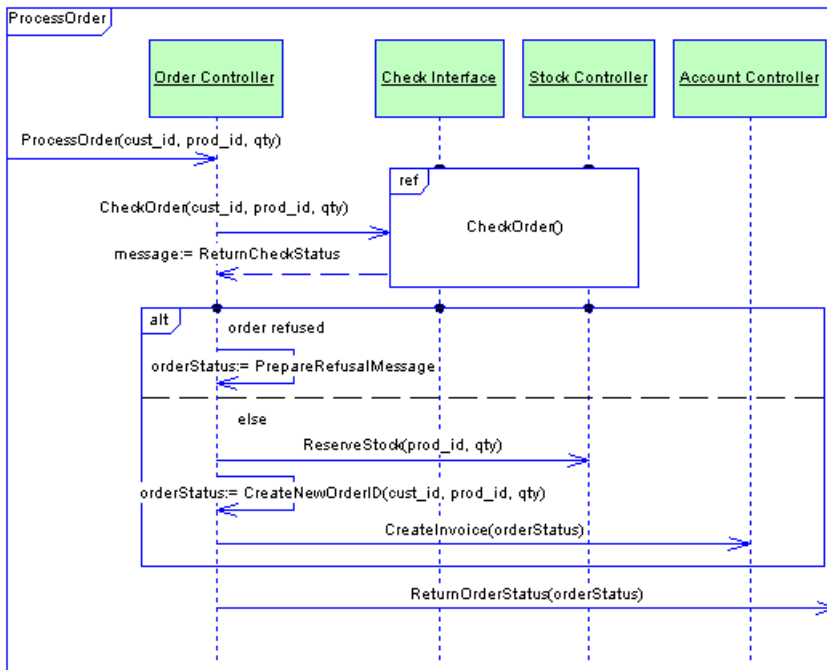
You can create a recursive message with activation from the Toolbox.

1. Click the **Self Call Message** tool in the Toolbox.
2. Click the object lifeline to create a recursive message with activation.

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*.

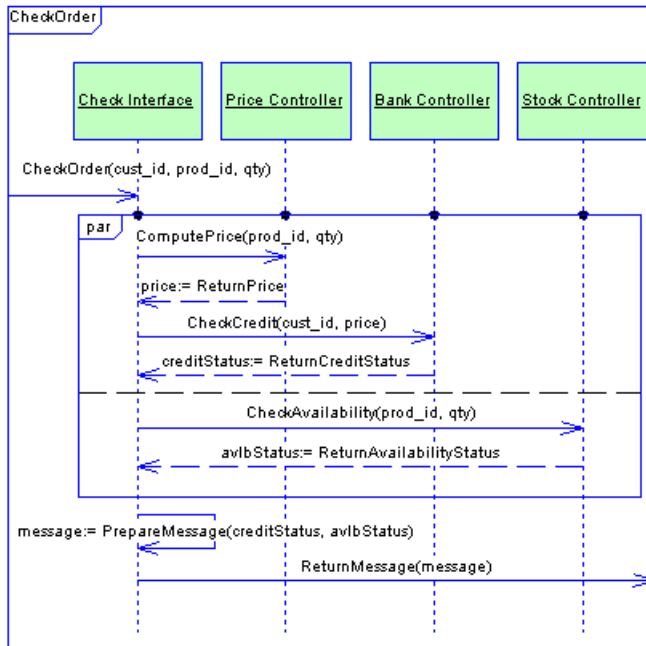
In the example below, a high-level sequence diagram, ProcessOrder, shows a series of communications between a user and an sales system:



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.

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.

Sequence Numbers

Sequence numbers can be assigned to messages in both communication and sequence diagrams, but they are most important in communication diagrams.

When you create a message in a communication diagram, the default value of the sequence number is calculated with regards to the most recently created or modified message. The first sequence number created is 1.

A succession of sequence numbers is built from the most recent sequence number plus 1. For example, $3 + 1 \Rightarrow 4$, or $2.1 + 1 \Rightarrow 2.2$

The creation of sequence numbers respects the syntax of numbers already used in the diagram (1, 2, 3, etc... or 1.1, 1.2, etc.).

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 need to change sequence numbers manually, you can move or insert messages in the diagram or increase and decrease the sequence numbers.

Moving Sequence Numbers

You can move and insert message numbers within the communication diagram.

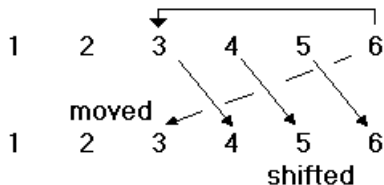
When you move an existing number and attach it to another message, the sequence numbers are recalculated with respect to the following rules:

- For a number "x", all numbers equal to or greater than number "x" are modified
- Any gap is filled with the sequence number that is immediately available after the move

Example 1

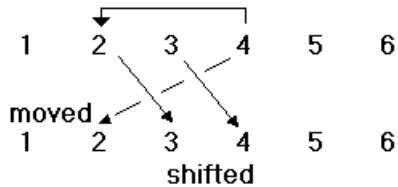
The sequence numbers in a communication diagram are 1, 2, 3, 4, 5, and 6.

When you change sequence number 6 and place it in third position, sequence number 6 becomes sequence number 3: all numbers between 3 and 6 are modified as follows:



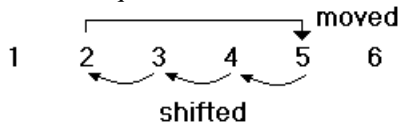
Example 2

When you change sequence number 4 and place it in second position, sequence number 4 becomes sequence number 2: all numbers between 2 and 4 are modified, 5 and 6 remain:



Example 3

When you change sequence number 2 and place it in fifth position, sequence number 2 becomes sequence number 5: all numbers between 2 and 5 are modified as follows:

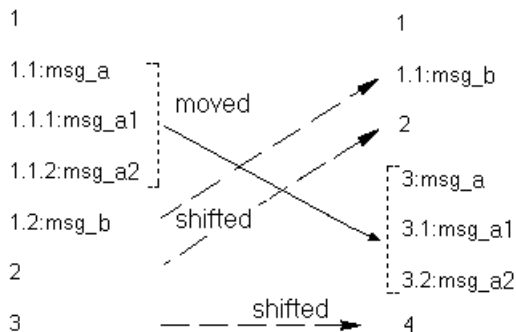


Example 4

The sequence numbers in a communication diagram are:

- 1
- 1.1:msg_a
- 1.1.1:msg_a1
- 1.1.2:msg_a2
- 1.2:msg_b
- 2
- 3

When you change sequence number 1.1:msg_a to sequence number 3, the following changes occur:



Note: You can use the Undo feature whenever needed while manipulating these elements within communication diagrams.

Inserting Sequence Numbers

When you insert a new message with a new sequence number in an existing communication diagram, the sequence numbers are recalculated with respect to the following rule: for each number after the insertion, all numbers are incremented by 1.

In the same manner, a parent changes its children. For example, the number is incremented by 1 for numbers like 1.1, 1.2, 1.3 as follows: $1.1 + 1 = 1.2$.

The syntax of sequence numbers currently used in the diagram is respected: that means that the number is incremented by 1 regardless of the syntax (1, 2, 3... or 1.1, 1.2, 1.3...).

Increasing Sequence Numbers in a Communication Diagram

You can increment a sequence number using the following methods:

- Right-click the message in the diagram and select Increase Number from the contextual menu.
or
Select the message in the diagram and press **Ctrl** and the numpad + sign to increment the number by 1.

Decreasing Sequence Numbers in a Communication Diagram

You can decrease a sequence number using the following methods:

- Right-click the message in the diagram and select Decrease Number from the contextual menu.
or
Select the sequence number in the diagram and press **Ctrl** and + sign to decrease the number by 1.

Activations (OOM)

Activations are optional symbols that represent the time required for an action to be performed. They are created on the lifeline of an object. They are purely symbols and do not have property sheets.

An activation can be created in the following diagrams:

- Sequence Diagram

In a communication diagram, messages that are passed during the period represented by an activation are given sub-numbers. Thus an activation created by message 1, may give rise to messages 1.1 and 1.2.

You can attach or detach a message to an activation. You can also move, resize, and cause the activation to overlap other activations.

Creating an Activation

You can create an activation when you create a Procedure Call message or afterwards for any type of message. A Procedure Call generally starts an activation, that is why the message is automatically attached to an activation.

Creating Activations with Procedure Call Messages

Activations are automatically created when you create procedure call messages in the diagram.

1. Select the **Procedure Call Message** tool in the Toolbox.
2. Click the lifeline of the sender object or actor and, while holding down the mouse button, drag the cursor towards the receiver object.
3. Release the mouse button on the object symbol of the receiver.

The message is created along with activations on the lifelines of both the sender and receiver (note that activations are not created on the lifelines of actors).

Creating an Activation from a Diagram

You can create an activation from a diagram using the Activation tool.

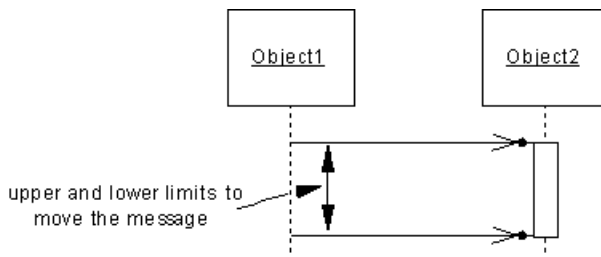
1. Select the **Activation** tool in the Toolbox.
2. Click the lifeline of an object to create an activation symbol at the click position. If you create the activation on top of an existing message, it is automatically attached to the activation.

Attaching a Message to an Activation

A message is attached to an activation when its begin or endpoint is on the activation symbol and not on the object lifeline. Attachment symbols appear at the endpoints of the message arrow (if attachment symbols are not displayed, select **Tools > Display Preferences**, and select the Activation Attachment display preference in the Message category).

You can attach an existing message that touches an activation but which does not display an activation symbol, by dragging the message inside the activation, while holding the **Ctrl** key down.

When a message is attached to an activation, you cannot move it outside the limits of the activation symbol as shown below:



If you delete an activation with a message attached, the message will be detached from the activation but will not be deleted.

Message Control Flow and Activation

When a message is attached to an activation, the control flow value of the message influences the position of the activation towards the message:

- Procedure Call - A Procedure Call message attached to an activation starts the activation on the receiver lifeline, that is to say the arrival point of the message is located at the top of the activation.
- Return - A Return message attached to an activation finishes the activation on the sender lifeline, that is to say the starting point of the message is located at the bottom of the activation.

Procedure Call and Return messages are the only messages defined on a definite location in the activation: a Procedure Call message is at the top of the activation, a Return message is at the bottom of the activation. Other messages attached to an activation can be moved without any constraint inside the activation.

Detaching a Message from an Activation

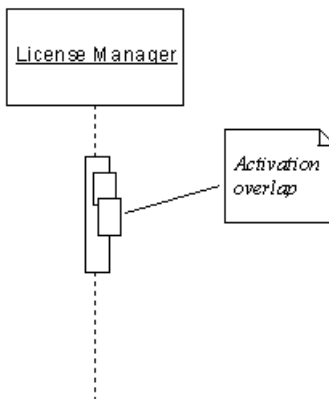
You can detach a message from an activation by dragging the message outside the activation, while holding the **Ctrl** key down.

Overlapping Activations

An activation can overlap other existing activations.

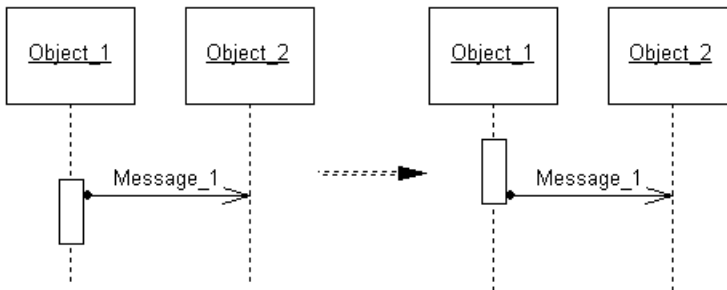
For example, you may want an activation to overlap another one to represent an action in a loop. The action is done repeatedly until it reaches its goal, this loop can start and finish at the same time another activation is representing another action.

You can make activations overlap this way to show concurrent activities.

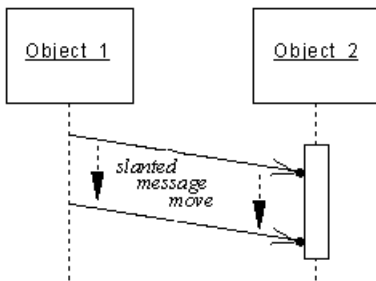


Moving an Activation

If you move an activation with a message attached to it, the message does not move. Thus, it is possible to move the activation up and down until its top or bottom reaches the message level.:



If you move the endpoint of a message with Support Delay, the inclination angle of the message is preserved as shown below:



Resizing an Activation

You may want to resize an activation by pulling the top or bottom of the activation symbol. This will resize the activation vertically.

When you resize an activation, the following rules apply:

- A Procedure Call message is always attached to the top of the activation on the receiver lifeline. The Procedure Call message stays at the top of the activation if the activation is moved up, or resized upwards.
- A Return message is always attached to the bottom of the activation on the sender lifeline. The Return message stays at the bottom of the activation if the activation is moved down.
- Messages that are covered by the activation after resizing are not automatically attached to the activation.

To change the activation of a message, press **Ctrl** and click to select the begin or endpoint of the message and drag it onto another activation.

Interaction References and Interaction Activities (OOM)

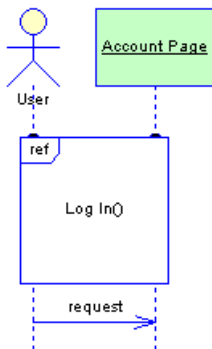
An interaction reference is used to represent one sequence diagram in the body of another. This feature allows you to modularize and reuse commonly-used interactions across a range of sequence diagrams. Interaction activities are similar, but are able to represent a sequence, communication, or interaction overview diagram.

Interaction references and interaction activities can be created in sequence diagrams.

For an interaction activity, right-click the activity and select **Composite View > Read-only (Sub-Diagram)** to see the referenced diagram displayed in the symbol. Select **Adjust to read-only view** from the contextual menu to automatically resize the symbol to optimize the display of the referenced diagram.

Example: Interaction Reference in Activity Diagram

In the example below, 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.



Creating an Interaction Reference

You can create an interaction reference from the Toolbox or Browser.

- Use the **Interaction Reference** tool in the Toolbox. You can either click near the lifeline of an object to create an interaction reference attached to that lifeline, or click and hold while drawing a box that will overlap and attach to several lifelines.

The **Select a sequence diagram** dialog box opens to allow you to specify the sequence diagram to which the reference refers. Select an existing or new diagram and click OK.

- Drag another sequence diagram from the browser and drop it into the present sequence diagram.

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

Creating an Interaction Activity

You can create an interaction activity from the Toolbox or Browser.

- Use the **Interaction Activity** tool in the Toolbox.
The **Select a diagram** dialog box opens to allow you to specify the diagram to which the activity refers. Select an existing or new diagram and click **OK**.
- Drag a sequence, communication or interaction overview diagram from the browser and drop it into an interaction overview diagram.

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

Interaction Reference and Interaction Activity Properties

You can modify an object's properties from its property sheet. To open an interaction reference 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 sequence diagram that will be represented in the current diagram by the interaction reference. You can click on the Create tool to the right of the box to create a new sequence 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.

Manipulating Interaction References

You can move interaction references, resize them, and generally manipulate them freely. When you cause the symbol to overlap an object lifeline, it attaches to it automatically, and this attachment is represented by a small bump on the top edge of the symbol where it meets the lifeline. If you drag or resize the symbol away from a lifeline, it detaches automatically.

If you move an object that is attached to an interaction, the symbol resizes itself automatically to remain attached to the object lifeline.

You can manually control whether object lifelines are attached to an interaction reference that passes over them by clicking the attachment point.

Note that an interaction reference cannot be copied or re-used in another diagram. However, multiple references to the same diagram can be created.

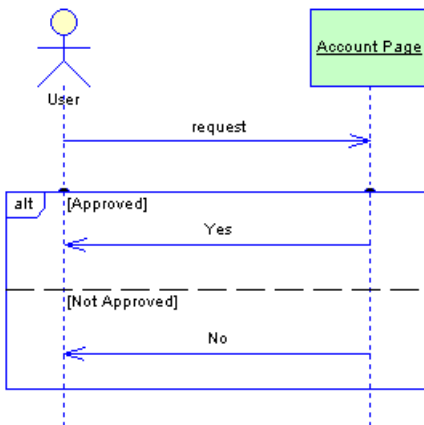
Interaction Fragments (OOM)

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.

An interaction fragment can be created in the following diagrams:

- Sequence Diagram

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.



Creating an Interaction Fragment

You can create an interaction fragment from the Toolbox.

- Use the **Interaction Fragment** tool in the Toolbox. You can either 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.

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

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 following sections detail the property sheet tabs that contain the properties most commonly entered for interaction fragments.

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"> • 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. • Assertion (assert) – the interaction must occur exactly as indicated or it will be invalid. • Break (break) – if the associated condition is true, the parent interaction terminates at the end of the fragment. • Consider (consider) – only the messages shown are significant. • Critical Region (critical) – no other messages can intervene until these messages are completed. • Ignore (ignore) – some insignificant messages are not shown. • Loop (loop) – the interaction fragment will be repeated a number of times. • Negative (neg) – the interaction is invalid and cannot happen. • Option (opt) – the interaction only occurs if the guard condition is satisfied. • Parallel (par) – the fragment is split into two or more regions, all of which will be executed in parallel at runtime. • Strict Sequencing (strict) – the ordering of messages is strictly enforced. • Weak Sequencing (seq) – the ordering of messages is enforced on each lifeline, but not between lifelines. <p>The operator type is shown in the top left corner of the interaction fragment symbol.</p>
Stereotype	<p>Extends the semantics of the object beyond the core UML definition.</p>
Condition	<p>Specifies any condition associated with the fragment. This may be the evaluation of a variable, such as:</p> <p>$X > 3$</p> <p>Or, for a loop fragment, the specification of the minimum and (optionally) maximum number of times that the loop will run. For example:</p> <p>1,10</p> <p>For the Consider or Ignore operators, this field lists the associated messages.</p> <p>This field is not available if the fragment does not support conditions.</p>

Sub-Regions Tab

The Interaction Sub-Regions Tab lists the regions contained within the fragment. It is only displayed if you select an operator that requires more than one region. You can add or delete regions and (if appropriate) specify conditions for them.

Manipulating Interaction Fragments

You can manipulate interaction fragments with some limitations.

Moving and Resizing Fragments

You can move interaction fragments, resize them, and generally manipulate them freely. When you cause the symbol to overlap an object lifeline, it attaches to it automatically, and this attachment is represented by a small bump on the top edge of the symbol where it meets the lifeline. If you drag or resize the symbol away from a lifeline, it detaches automatically.

If you move an object that is attached to an interaction fragment, the symbol resizes itself automatically to remain attached to the object lifeline.

You can manually control whether object lifelines are attached to an interaction fragment that passes over them by clicking the attachment point.

Moving Messages

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. However, if you move the fragment away from either of the lifelines to which it is attached, the message will be detached from the fragment and not moved.

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. If you move a message so either of its ends is outside the fragment, then it is detached from the fragment.

Messages and Regions

When a fragment is split into two or more regions, you can move messages freely between regions. However, 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, you must 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.

Note that an interaction fragment cannot be copied as a shortcut in another diagram.

Activities (OOM)

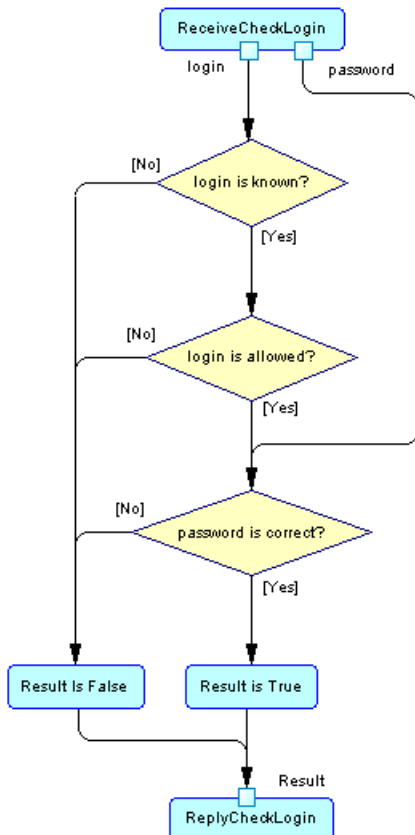
An *activity* is the invocation of a manual or automated action, such as "send a mail", or "increment a counter". When the activity gains control, it performs its action and then, depending on the result of the action, the transition (control flow) is passed to another activity.

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* on page 164)
- action type of the activity and associate it with other model objects (see *Specifying action types* on page 165)



In the example above, the ReceiveCheckLogin activity has an action type of "Accept call" (see *Specifying action types* on page 165), and passes the two output parameters "login" and "password" (see *Specifying activity parameters* on page 164) 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* on page 170.

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.

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 > The PowerDesigner Interface > Objects*.

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 Code 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.
Organization unit	Specifies the organization unit (see <i>Organization Units (OOM)</i> on page 173) linked to the activity and also allows you to assign the Committee Activity value (see <i>Displaying a Committee Activity</i> on page 176) 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. 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.
Composite status	Specifies whether the activity is decomposed into sub-activities. You can choose between: <ul style="list-style-type: none"> • Atomic Activity – (default) The activity does not contain sub-activities • 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. <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 <i>Specifying action types</i> on page 165.
[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

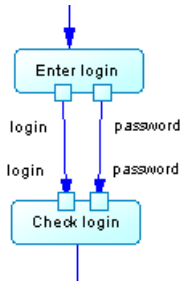
These tabs list the input and output parameters required by the activity (see *Specifying Activity Parameters* on page 164).

Sub-Activities Tab

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

Specifying Activity Parameters

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



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.

Note: You can also create parameters as a part of specifying an activity action type. See [Specifying action types](#) on page 165.

Activity parameters can have the following properties:

Property	Description
Parent	Specifies the parent activity.
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 Code 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.

Property	Description
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.

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.

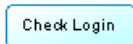
1. Open the property sheet of an activity and click the Action tab.
2. Select an action type. The following list details the available action types, and specifies where appropriate, the required action object:
 - *<Undefined>* [no object] - default. No action defined
 - *Reusable Activity* [no object] – a top-level container.
 - *Call*[operation or activity] – calls an operation or activity. See *Example: Using the Call action type* on page 166
 - *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 *Example: Reading and writing variables* on page 169.
 - *Write Variable* [variable] - reads a value from a local variable. See *Example: Reading and writing variables* on page 169.
 - *Evaluate Expression* [expression text] – evaluates an expression and returns the value as an output pin.
 - *Unmarshall*[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.

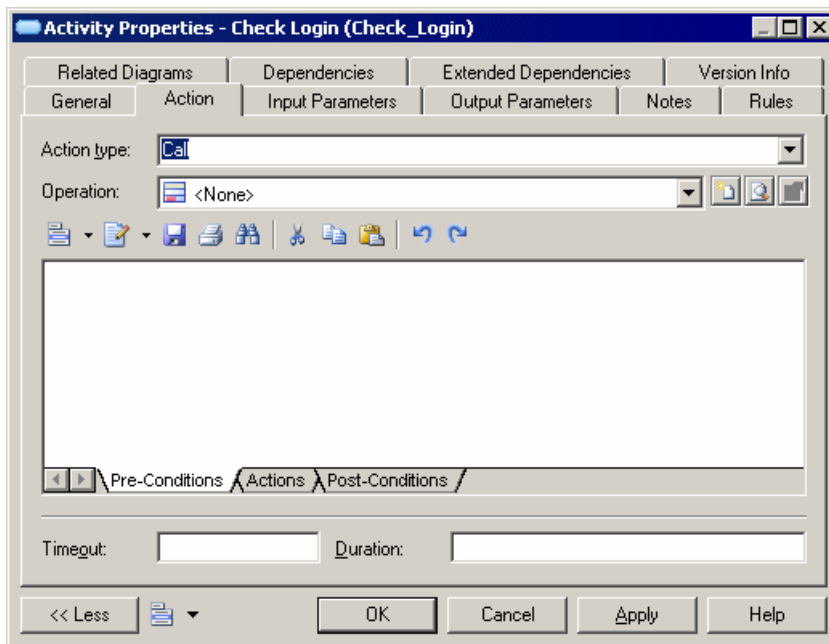
Example: Using the Call Action Type

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

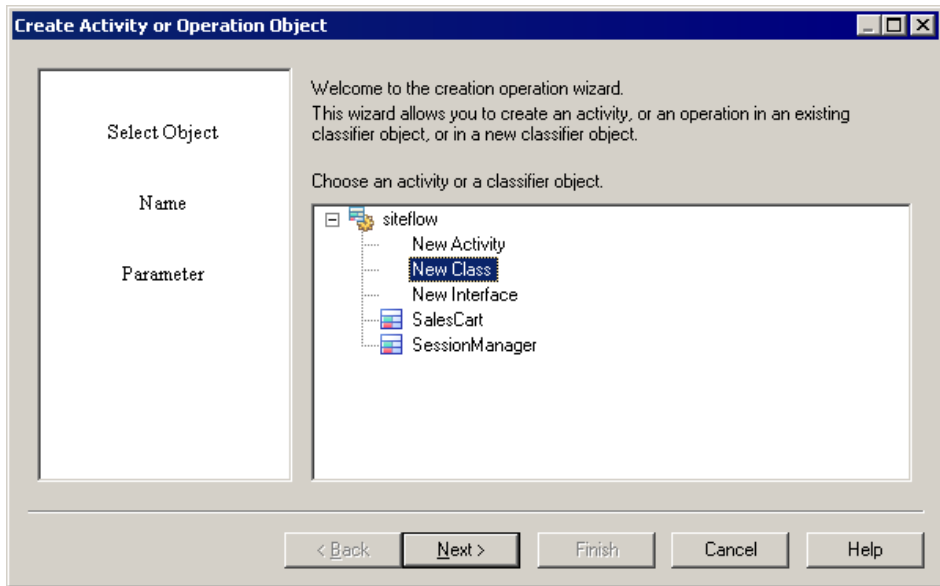
1. Create an activity and call it Check Login.



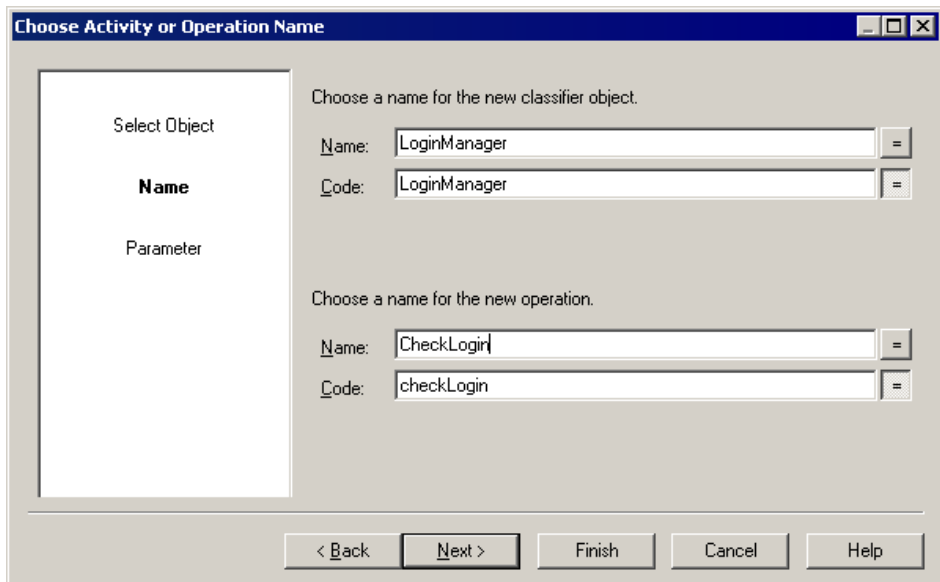
2. Open its property sheet, click the Action tab, and select Call from the Action type list. The Operation field appears:



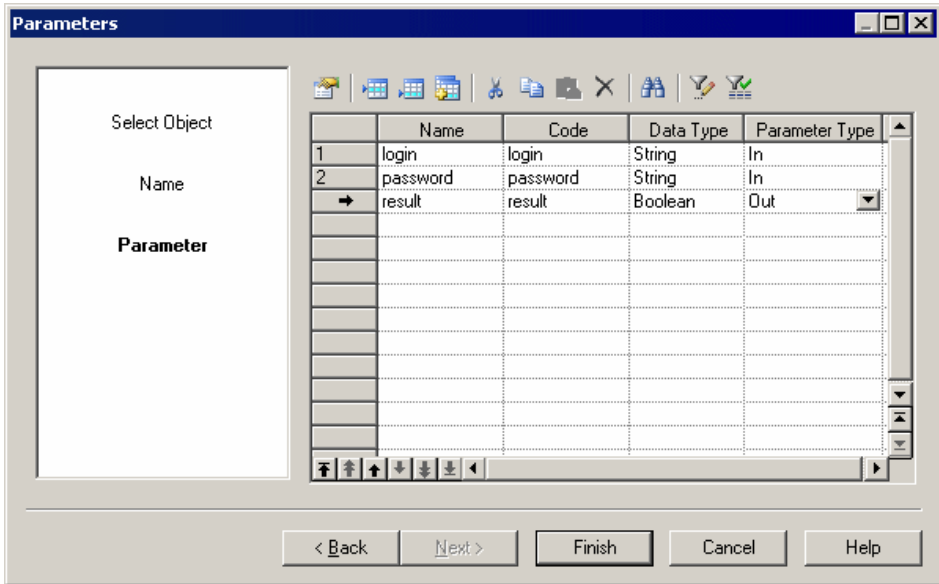
3. Click the Create tool to the right of the new field to open a wizard to choose an operation:



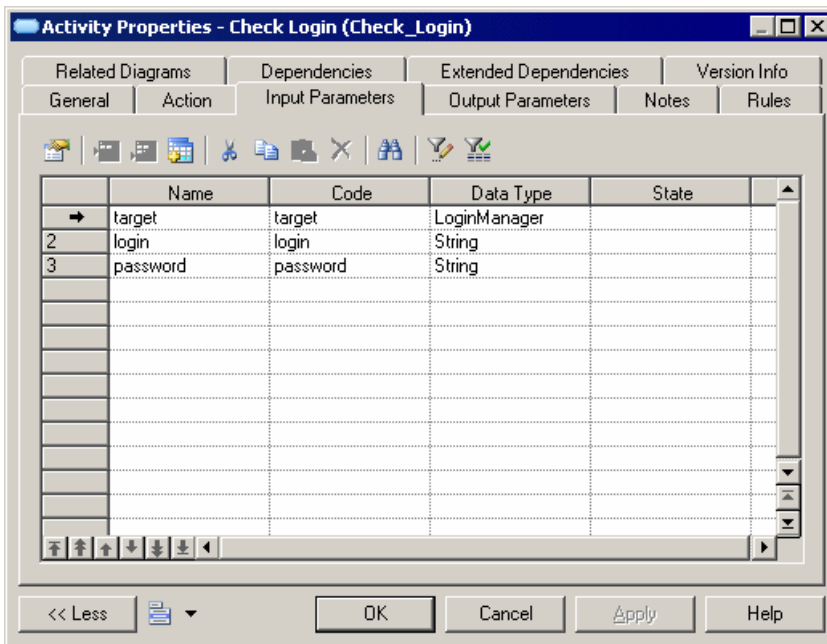
4. You can choose an existing classifier or activity, or select to create one. Select New Class, and then click Next:



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

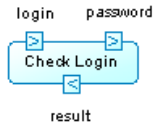


6. Create two input and one output parameter for the operation, and then click Finish. The property sheet of the new operation opens to allow you to further specify the operation. When you are finished, click OK to return to the Activity property sheet and click the Input Parameters tab to view the parameters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

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

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/ 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 Code 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"> • * – none to unlimited • 0..* – zero to unlimited • 0..1 – zero or one • 1..* – one to unlimited • 1..1 – exactly one
Keywords	<p>Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.</p>

Creating a Variable

To create a variable:

1. Open the property sheet of an activity and click the Action tab.
2. Select an action type of Write Variable, and then click the Create tool to the right of the variable field to create a new variable and open its property sheet.
3. Specify the name and other properties of the variable and click OK to return to the activity property sheet.

Writing to an Existing Variable

To write to an existing variable:

1. Open the property sheet of an activity and click the Action tab.
2. Select an action type of Write Variable, and then click the Select Object tool to the right of the variable field to choose the variable.
3. Select the appropriate variable in the list and click OK to return to the activity property sheet.

Reading from a Variable:

To read from a variable:

1. Open the property sheet of an activity and click the Action tab.
2. Select an action type of Read Variable, and then click the Create or Select Object tool to the right of the variable field to create a new variable or select an existing one.

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.

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.

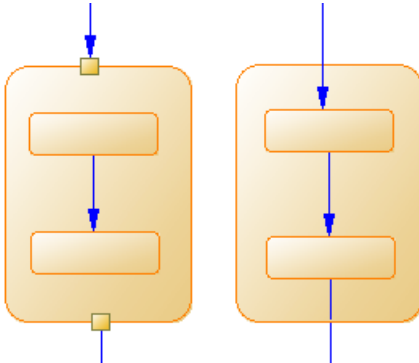
You can choose the mode for viewing composite activities on a per-object basis, by right-clicking the symbol and selecting the desired mode from the **Composite View** menu.

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:

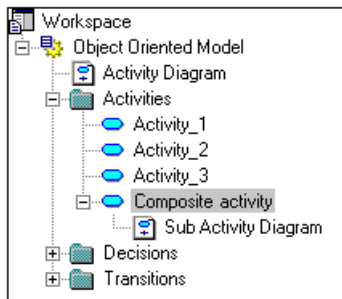


Converting an Atomic Activity to a Decomposed Activity

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.

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**.

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.

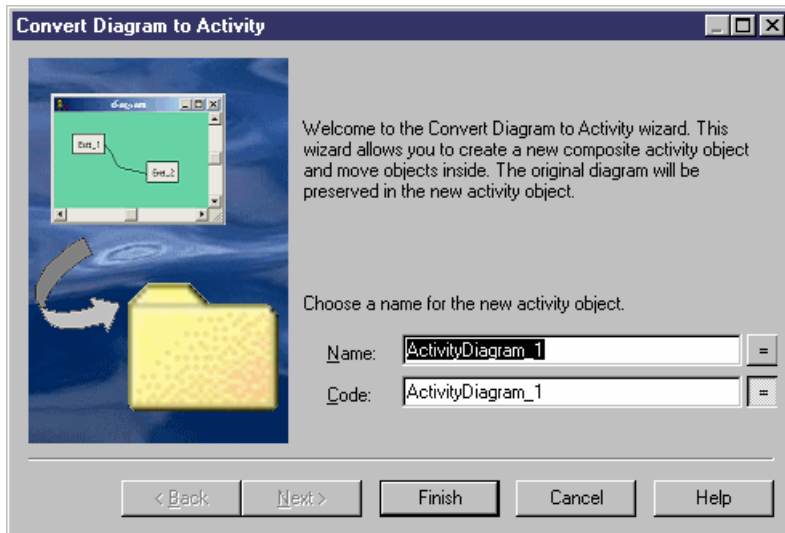
1. Right-click the diagram node in the Browser and select Convert to Composite Activity from the contextual menu.

or

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

or

Select **Tools > Convert to Composite Activity**.



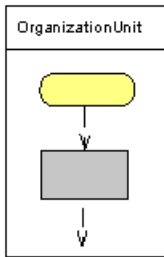
2. Specify a name and a code in the Convert Diagram to Activity page, and then click Next to open the Selecting Objects to Move page.
3. Select the activities that you want to move to the new decomposed activity diagram. Activities that you select will be moved in the Browser to under the new decomposed 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

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:



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.
- Select **Model > Organization Units** 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 **New > Organization Unit**.

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

Creating Organization Units with the Swimlane Tool

Use the Organization Unit Swimlane tool in the Toolbox to quickly create organization unit swimlanes.

Click in or next to an existing swimlane or pool of swimlanes to add a swimlane to the pool.

Click away from existing swimlanes to create a new pool.

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 Code field.
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>An organization unit has the following predefined stereotypes:</p> <ul style="list-style-type: none"> • Role – specifies a role a user plays • User • Group – specifies a group of users • Company • Organization – specifies an organization as a whole • Division – specifies a division in a global structure • Service – specifies a service in a global structure
Parent organization	<p>Specifies another organization unit as the parent to this one.</p> <p>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.</p> <p>The relationship between parent and child organization units can be used to group swimlanes having the same parent. For more information, see <i>Grouping and Ungrouping Swimlanes</i> on page 178.</p>
Keywords	Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.

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.

Displaying a Committee Activity

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

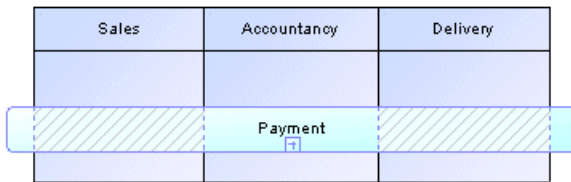
1. Open the property sheet of a decomposed activity.
2. Select **Committee Activity** from the Organization Unit list and click **OK**.

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.

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:

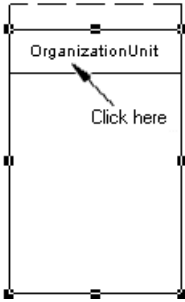
- Manage any sub-activities
- Have any symbol in the sub-activity diagram

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

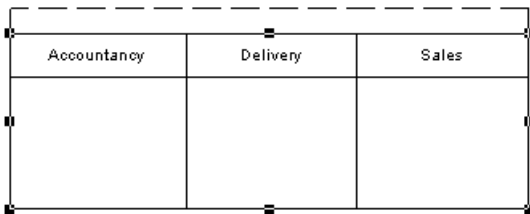
Managing Swimlanes and Pools

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:



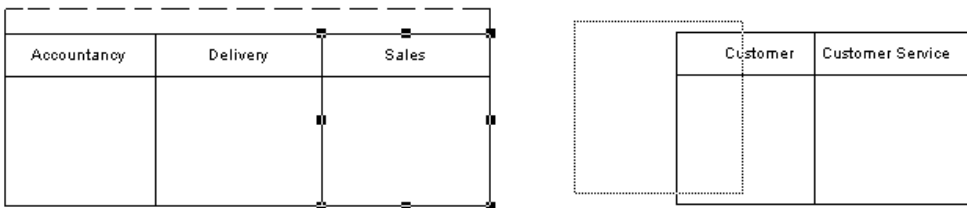
Moving, Copying and Pasting Swimlanes

You can move, copy, and paste swimlanes and pools in the same or in another diagram.

Diagram	What happens...
Same	When 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), so as to preserve the layout of the diagram.
Different	When you move or copy a swimlane or pool to another folder or 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, and is moved to another pool containing Customer and Customer Service:



After the move, Sales has moved from its original pool, and joined the pool containing Customer and Customer Service:

Accountancy	Delivery

Sales	Customer	Customer Service

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

Accountancy	Delivery

Sales

Customer	Customer Service

If you move linked objects inside a swimlane, the width or height of the swimlane varies with them.

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

Grouping and Ungrouping Swimlanes

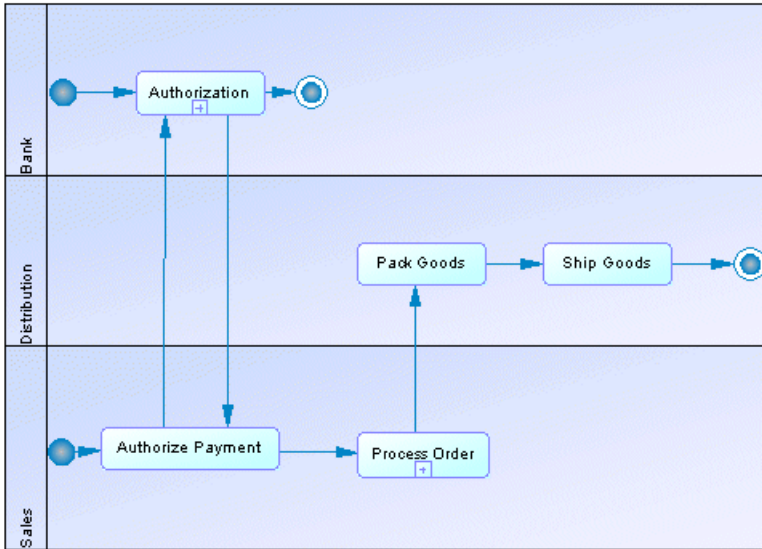
Group organization unit swimlanes within a pool to organize them under a common parent or user-defined name.

To group swimlanes within a pool, right-click the pool, select **Swimlane Group Type**, and then select one of the following options:

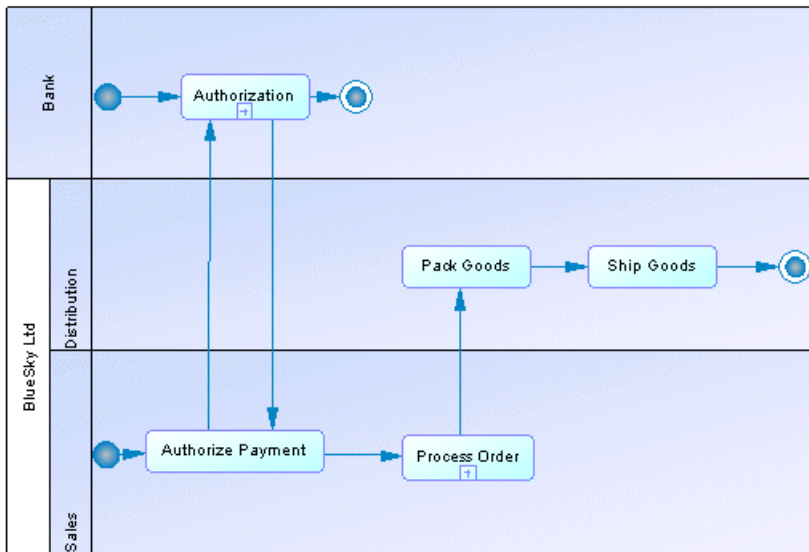
- **By Parent** - to assign the name of the last common parent for the group. This name comes from the Parent Organization Unit field in the swimlanes property sheet.
- **User-Defined** - to assign a name of your choice for the group. Then, you must select at least two attached swimlanes, and select **Symbol > Group Symbols** from the menu bar to display a default name that you can modify.

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

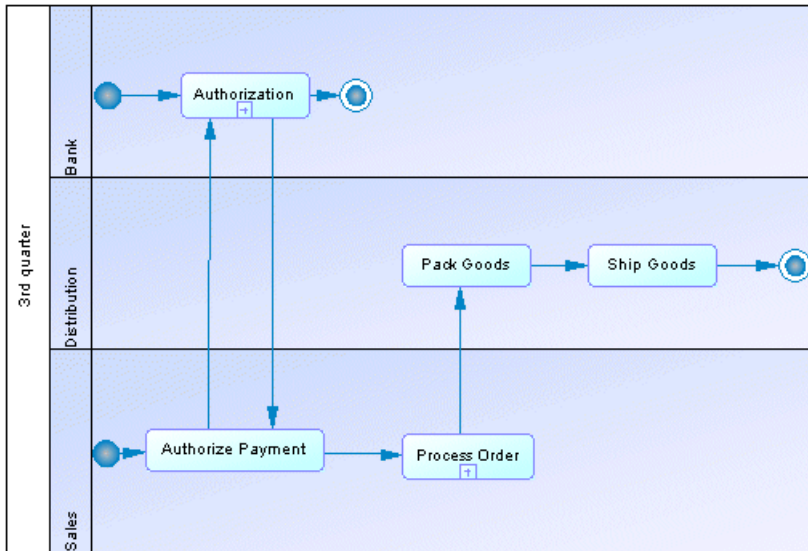
The following example shows a pool without any group:



In the following example, Sales and Distribution are grouped by their BlueSky Ltd common parent:



In the following example, the pool is assigned a user-defined group named 3rd quarter:

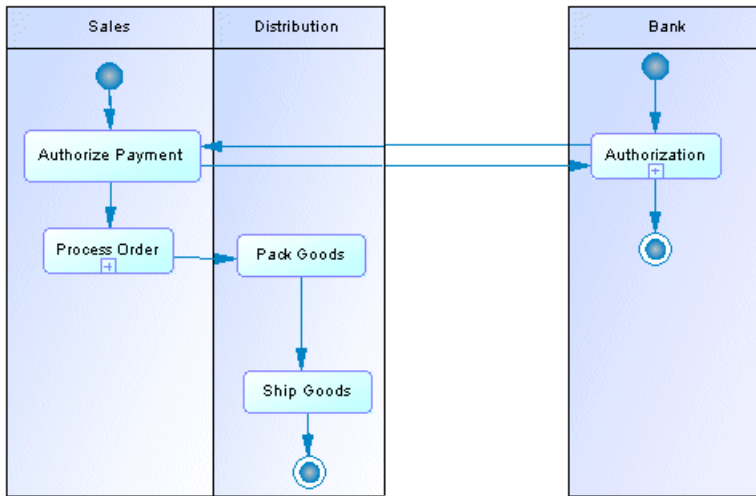


Creating Links Between Pools of Swimlanes

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:



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

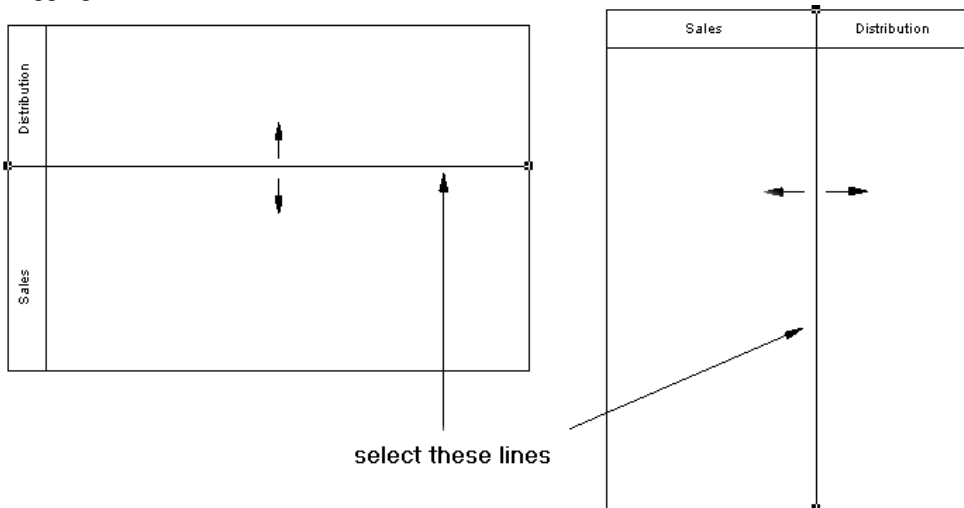
Changing the Orientation of Swimlanes

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.

1. Select **Tools > Display Preferences** to open the Display Preferences dialog box.
2. Select the appropriate radio button in the Organization unit swimlane groupbox, and click **OK**.

Resizing Swimlanes

You can resize swimlanes within a pool by clicking the dividing line between them and dragging it.



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

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.

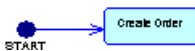
Changing the Format of a Swimlane

You can change the format of swimlanes or a pool using the Symbol Format dialog box. Format changes apply to all swimlanes in the pool.

1. Right-click the swimlane or a pool, and select **Format** to display the Symbol Format dialog box.
2. Enter or select changes in the different tabs, and click **OK** to return to the diagram.

Starts (OOM)

A *start* is a starting point of the flow represented in the diagram.



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

Note: The start is compared and merged in the Merge Model feature, which checks that there is no additional start in decomposed activities.

Creating a Start

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

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

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

Start Properties

To view or edit a start'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 Code 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.

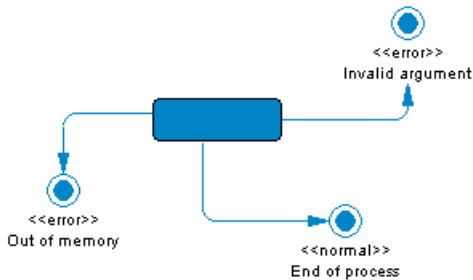
Ends (OOM)

An *end* is a termination point of the flow represented in the diagram.

An end can be created in the following diagrams:

- Activity Diagram
- Statechart Diagram
- Activity Overview Diagram

You can create several ends within the same diagram, if you want to show divergent end cases, like errors scenarios:



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

You should not use the same end in more than one diagram. You are not allowed to use shortcuts of ends, but graphical synonyms are permitted.

Creating an End

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

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

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

End Properties

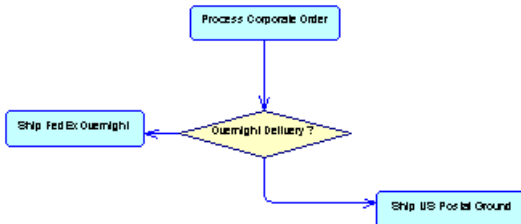
To view or edit an end's properties, double-click its diagram symbol or Browser or list entry. The property sheet tabs and fields listed here are those available by default, before any customization of the interface by you or an administrator.

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

Property	Description
Name/Code/ 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 Code 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	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.

Decisions (OOM)

A *decision* specifies which path to take, when several paths are possible.



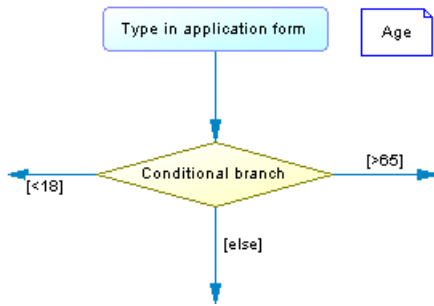
A decision can be created in the following diagrams:

- Activity Diagram
- Interaction Overview Diagram

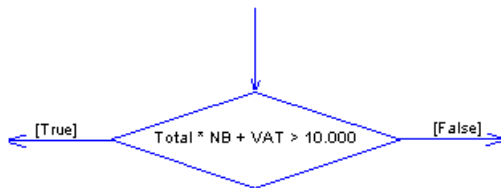
A decision can have one or more input flows and one or more output flows, each labeled with a distinct *guard condition*, a condition that must be satisfied for an associated flow to execute some action. Your guard conditions should avoid ambiguity by not overlapping, yet should also cover all possibilities in order to avoid process freeze.

A decision can represent:

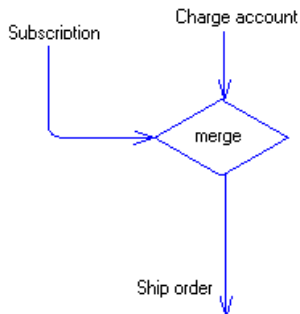
- A conditional branch: one input flow and several output flows. In the following example, the control flow passes to the left if the age given in the application form is <18, to the right if the age is >65, and takes the another route if the age is not mentioned:



You can display a condition on the decision symbol in order to factorize the conditions attached to the flows. In the following example, the condition $\text{Total} * \text{NB} + \text{VAT} > 10.000$ is entered in the Condition tab in the decision property sheet, and True and False are entered in the Condition tabs of the flows:



- 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: It is not possible to attach two flows of opposite directions to the same corner on a decision symbol.

Creating a Decision

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

- Use the **Decision** tool in the Toolbox
- Select **Model > Decisions** 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 **New > Decision**.

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

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/ 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 Code 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 that can have the following values: <ul style="list-style-type: none"> • Incomplete - No input or no output transition has been defined or exactly one input and one output transitions have been defined • Conditional branch – One input and several outputs have been defined • Merge - Several inputs and one output have been defined
Keywords	Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.

Decision Property Sheet Condition Tab

The **Condition** tab 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.

Synchronizations (OOM)

A *synchronization* enables the splitting or synchronization of control between two or more concurrent actions.

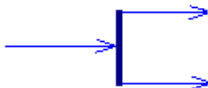
A synchronization can be created in the following diagrams:

- Activity Diagram
- Statechart Diagram
- Interaction Overview Diagram

Synchronizations are represented as horizontal or vertical lines. You can change the orientation of the symbol by right-clicking it and selecting Change to Vertical or Change to Horizontal from the contextual menu.

A synchronization can be either a:

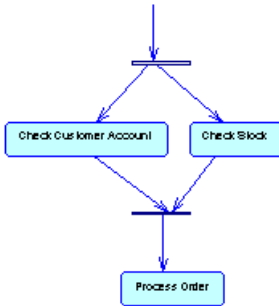
- Fork - Splits a single input flow into several independent output flows executed in parallel:



- Join – Merges multiple input flows into a single output flow. All input flows must reach the join before the single output flow continues:



In the following example, the flow coming into the first synchronization is split into two separate 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:



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**.

By default, the synchronization symbol is created horizontally. To toggle between horizontal and vertical display, right-click the symbol and select **Change to Vertical** or **Change to Horizontal** in the contextual menu.

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

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/ 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 Code field.

Property	Description
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.
Type	[read-only] Calculates the form of the synchronization: <ul style="list-style-type: none"> • Incomplete - No or exactly one input or no output transition has been defined • Conditional branch – One input and several outputs have been defined • Merge - Several inputs and one output have been defined
Keywords	Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.

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*.

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.

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 > The PowerDesigner Interface > Objects*.

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.
Source	Specifies the object that the flow leads from. 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 object by clicking the Source button located at the top part of the current object property sheet.
Destination	Specifies the object that the flow leads 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 destination object by clicking the Destination button located at the top part of the current object property sheet.
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
Flow type	<p>Specifies the type of the flow. You can create your own type of flow in the list, or choose one of the following values:</p> <ul style="list-style-type: none"> • Success - defines a successful flow • Timeout - defines the occurrence of a timeout limit • Technical error • Business error • Compensation - defines a compensation flow <p>The flow type is unavailable if you associate an event with the flow on the Condition tab.</p>
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.

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.

Flow Property Sheet Condition Tab

Parameter	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 how the flow 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 flow symbol, and selecting Condition in the contextual menu.

Flow Property Sheet Parameters Tab

The **Parameters** tab lists the parameters that are passed along the flow. The list is automatically completed if you draw the flow between two activity parameters .

Flow Property Sheet Transformation Tab

The **Transformations** tab specifies a data transformation to apply to input tokens. For example, it could extract a single attribute value from an input object.

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:



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 **Model > Object Nodes** 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 **New > Object Node**.
- 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 > The PowerDesigner Interface > Objects*.

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 Code 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.

States (OOM)

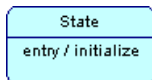
A *state* represents a situation during the life of a classifier that is usually specified by conditions. It can also be defined as the situation of a classifier waiting for events. Stability and duration are two characteristics of a state.

A state can be created in the following diagrams:

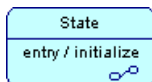
- Statechart Diagram

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* on page 197.

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 exit 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.

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 > The PowerDesigner Interface > Objects*.

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 Code 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.

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)* on page 205.

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.

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.

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.

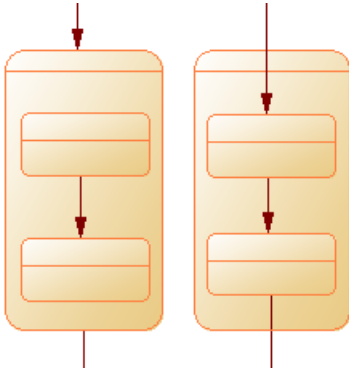
You can choose how to view composite states on a per-object basis, by right-clicking the symbol and selecting the desired mode from the **Composite View** menu.

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 substates in the Editable Composite View mode:

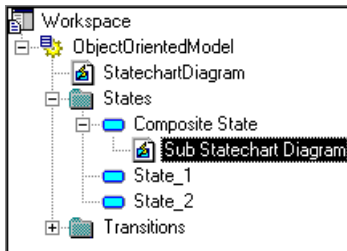


Working with Sub-State Diagrams

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.

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**.

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.

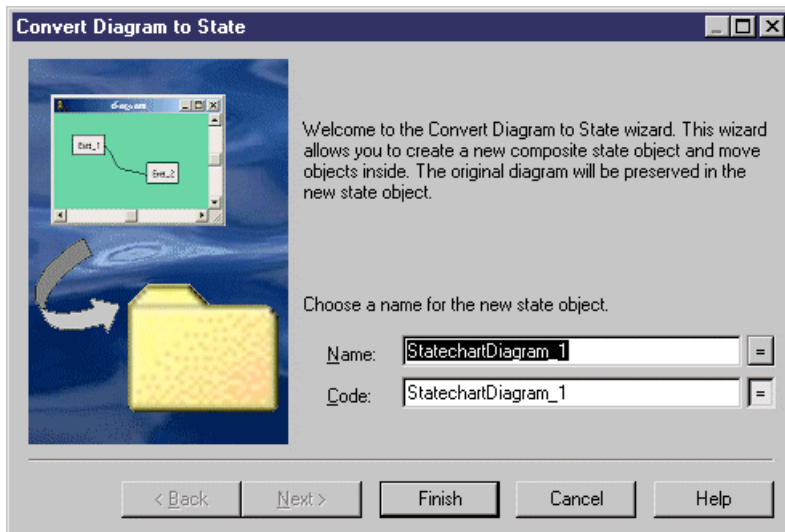
1. Right-click the diagram node in the Browser and select Convert to Decomposed State from the contextual menu.

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

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:

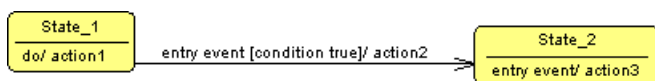
- Statechart Diagram

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.

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.

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 > The PowerDesigner Interface > Objects*.

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/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.
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.
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.
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"> • Success – Defines a successful flow • Timeout – Defines a timeout limit • Exception – Represents an exception case
Keywords	Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.

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.

Trigger Tab

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

Property	Description
Trigger event	Specifies the event (see <i>Events (OOM)</i> on page 202) 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

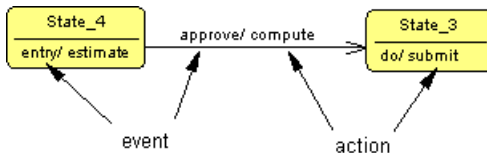
Events (OOM)

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:

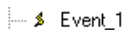
- Statechart Diagram

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* on page 205.

Creating an Event

You can create an event from the Browser, from the **Model** menu or from a transition property sheet.

- Select **Model > Events** to access the List of Events, and click the Add a Row tool
- Right-click the model or package in the Browser, and select **New > Event**
- 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 > The PowerDesigner Interface > Objects*.

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 Code 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.

Dependencies Tab

The **Dependencies** tab contains a Triggered Objects tabbed tab that displays the actions on states and on transitions that are triggered by this event.

Parameters Tab

The **Parameters** tab lists all the parameters associated with the event. It allows you to define event parameters that correspond to the event signature. Add parameters to the list with the **Add a Row** tool, and view the properties of a parameter with the **Properties** tool.

A parameter has the following properties:

Property	Description
Parent	[Read-only] 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 Code field.
Data type	Set of instances sharing the same semantics
Array	When selected, indicates that the data type is a table format

Property	Description
Parameter Type	<p>Direction of information flow for the parameter. Indicates what is returned when the parameter is called by the event during the execution process. You can choose from the following:</p> <ul style="list-style-type: none"> • In – Input parameter passed by value. The final value can not be modified and information is not available to the caller • In/Out – Input parameter that may be modified. The final value may be modified to communicate information to the caller • Out – Output parameter. The final value may be modified to communicate information to the caller
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>

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".

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)* on page 202.

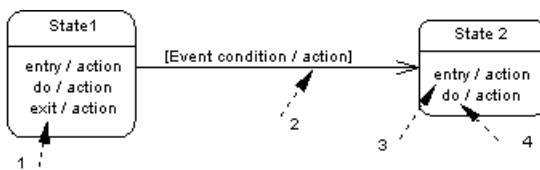
Action on State and on Transition

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.



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 > The PowerDesigner Interface > Objects*.

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	<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 Code field.</p> <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 Code field.</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.

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"> • Add an event to an action by choosing it from the list. • Add multiple events by clicking the ellipsis tool next to the list. • Create a new event by clicking the Create tool. • Select an event created in the current model or other models by clicking the Select Trigger Event tool. <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 <i>Events (OOM)</i> on page 202).</p>
Event arguments	<p>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</p>
Operation	<p>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</p>
Operation arguments	<p>Arguments of an event defined on an operation</p>
Keywords	<p>Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.</p>

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 →

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.

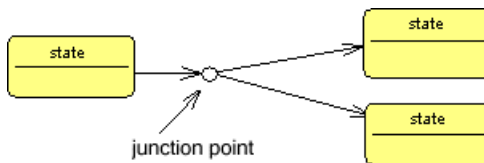
A junction point can be created in the following diagrams:

- 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:



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 > The PowerDesigner Interface > Objects*.

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 Code 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.

CHAPTER 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* on page 211.
- 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* on page 213.

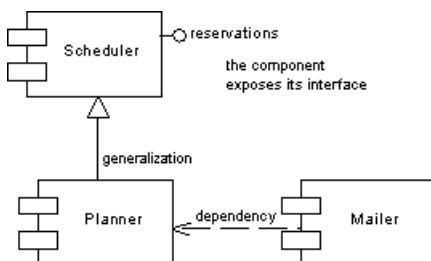
Component Diagrams

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.

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 *Chapter 12, Working with Java* on page 343 and *Chapter 16, Working with VB .NET* on page 437.

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.

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 <i>Components (OOM)</i> on page 214.
Interface			Descriptor for the externally visible operations of a class, object, or other entity without specification of internal structure. See <i>Interfaces (OOM)</i> on page 53.
Port			Interaction point between a classifier and its environment. See <i>Ports (OOM)</i> on page 62.
Part			Classifier instance playing a particular role within the context of another classifier. See <i>Parts (OOM)</i> on page 60.
Generalization			A link between a general component and a more specific component that inherits from it and add features to it. See <i>Generalizations (OOM)</i> on page 98.
Realization			Semantic relationship between classifiers, in which one classifier specifies a contract that another classifier guarantees to carry out. See <i>Realizations (OOM)</i> on page 105.
Require Link			Connects components to interfaces. See <i>Require Links (OOM)</i> on page 106.
Assembly Connector			Connects parts to each other. See <i>Assembly Connectors (OOM)</i> on page 107.

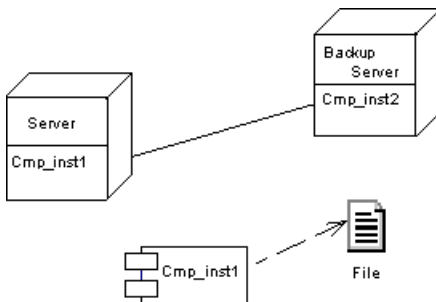
Object	Tool	Symbol	Description
Delegation Connector			Connects parts to ports on the outside of components. See <i>Delegation Connectors (OOM)</i> on page 109.
Dependency			Relationship between two modeling elements, in which a change to one element will affect the other element. See <i>Dependencies (OOM)</i> on page 101.

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.

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 component 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.


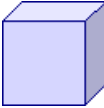






You can build a deployment diagram to show the following views, from a high level architecture that describes the material resources and the distribution of the software in these resources, to final complete deployment into a server:

- 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

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 <i>Nodes (OOM)</i> on page 220.
Component instance			Instance of a deployable component that can run or execute on a node. See <i>Component Instances (OOM)</i> on page 222.
Node association			An association between two nodes means that the nodes communicate to each other. See <i>Node Associations (OOM)</i> on page 227.
Dependency			Relationship between two modeling elements, in which a change to one element will affect the other element. See <i>Dependencies (OOM)</i> on page 101.

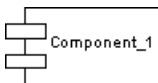
Components (OOM)

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 *Chapter 12, Working with Java* on page 343 and *Chapter 16, Working with VB .NET* on page 437.

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 *Chapter 22, Object/Relational (O/R) Mapping* on page 539.

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.

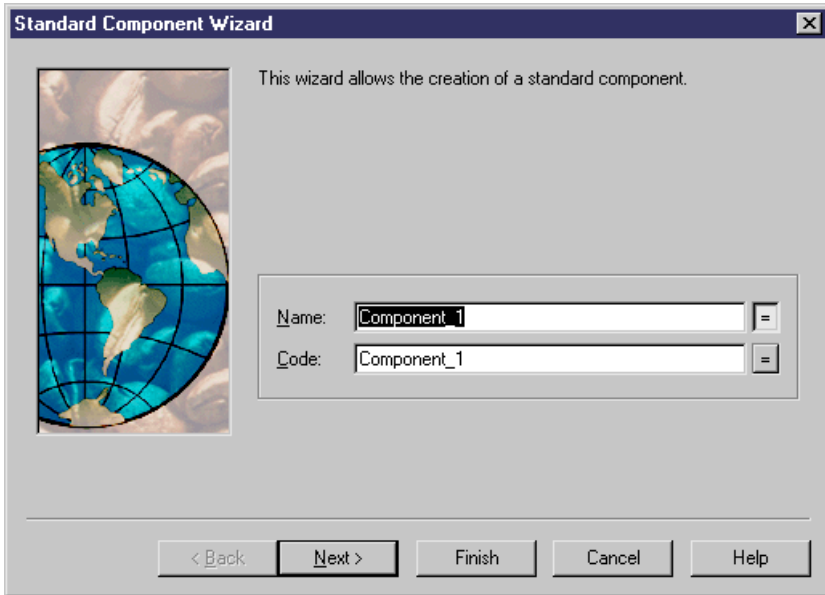
For general information about creating objects, see *Core Features Guide > The PowerDesigner Interface > Objects*.

For more information on other types of components: EJB, servlets, JSP and ASP.NET, see *Chapter 12, Working with Java* on page 343 and *Chapter 16, Working with VB .NET* on page 437.

Using the Standard Component Wizard

PowerDesigner provides a wizard to help you in creating components from classes.

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.



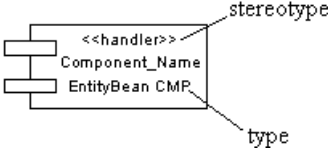
3. Type a name and a code for the component and click Next.
4. If you want the component to have a symbol and appear in a diagram, then select the Create Symbol In checkbox and specify the diagram in which you want it to appear (you can choose to create a new diagram). If you do not select this checkbox, 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.

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	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 Code 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 standard stereotypes are available by default:</p> <ul style="list-style-type: none"> • <<Document>> - Generic file that is not a source file or an executable • <<Executable>> - Program file that can be executed on a computer system • <<File>> - Physical file in the context of the system developed • <<Library>> - Static or dynamic library file • <<Table>> - Database table <p>You can modify an existing stereotype or create a new one in an object language or extension file.</p>
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 <i>Chapter 6, Web Services</i> on page 229).</p> <p>To display the type of a component, select Tools > Display Preferences 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 <i>Creating a Class Diagram for a Component</i> on page 219).
Web service	Indicates that the component is a Web service.

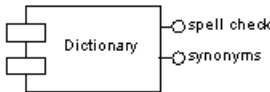
Property	Description
Keywords	Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.

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* on page 361.

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:

- Ports - lists the ports associated with the component. You can create ports directly in this tab (see *Ports (OOM)* on page 62).

- Parts - lists the parts associated with the component. You can create parts directly in this tab (see *Parts (OOM)* on page 60).
- 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)* on page 224).
- Operations - lists the operations contained within the interfaces associated with the component. Use the filter in the toolbar to filter by a specific interfaces.
- Related Diagrams - lists and lets you add model diagrams that are related to the component (see *Core Features Guide > The PowerDesigner Interface > Diagrams, Matrices, and Symbols > Diagrams > Specifying Diagrams as Related Diagrams*).

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.

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.

Right-click the component in the component diagram and select Create/Update Class Diagram from the contextual menu.

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.

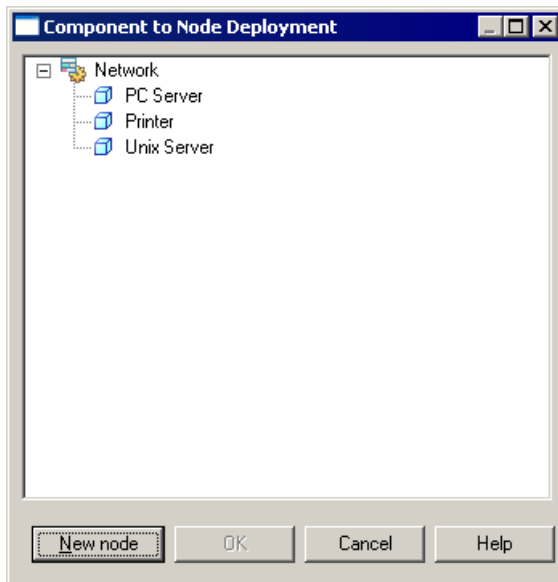
Note: To open the class diagram for a component, right-click the component in the diagram and select Open Class Diagram from the contextual menu or press **Ctrl** and double-click the component.

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.

You can only select one component to deploy at a time.

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.

Nodes (OOM)

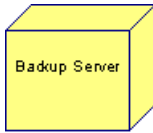
A node is the main element of the deployment diagram. It is a physical element that represents a processing resource, a real physical unit, or physical location of a deployment (computer, printer, or other hardware units).

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:

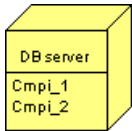
- Deployment Diagram

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**.

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 > The PowerDesigner Interface > Objects*.

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/ 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 Code 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)* on page 222). You can specify component instances directly on this tab and they will be displayed within the node symbol.

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**.

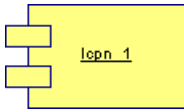
Component Instances (OOM)

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* on page 220.

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 > Component Instances** 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 > Component Instance**.

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

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/ 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 Code 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.
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.

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

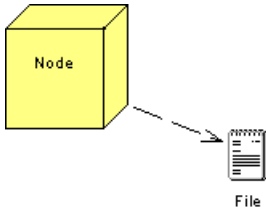
The symbol of a file object is as follows:



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.

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 > Files** 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 > File**.
- 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 > The PowerDesigner Interface > Objects*.

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/ 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 Code 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"> • Embedded – 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. • External – the file is stored in the Windows file system, and you must enter its path in the Location field. If you subsequently change the type to embedded, you will be prompted to import the contents of the file into the model. • URL – the file is on the web and you must enter its URL in the Location field
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.
Artifact	Specifies that the file object is not a piece of documentation, but rather forms an integral part of the application. If an artifact has an extension that is defined in the Editors page in the General Options dialog linked to the <i><internal></i> editor, a Contents tab is displayed in the artifact property sheet, which allows you to edit the artifact file in the PowerDesigner text editor.
Keywords	Provide a way of loosely grouping objects through tagging. To enter multiple keywords, separate them with commas.

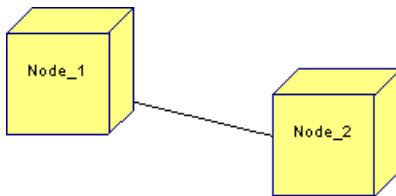
Node Associations (OOM)

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:



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 > The PowerDesigner Interface > Objects*.

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 Code 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 node.
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"> • 0..1 – zero or one • 0..* – zero to unlimited • 1..1 – exactly one • 1..* – one to unlimited • * – none to unlimited <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> 
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.
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.

A Web service is a service offered via the web. The principle on which a Web service works is the following: a business application sends a request to a service at a given URL address. The request can use the SOAP protocol over HTTP. The service receives the request, processes it, and returns a response. An example that is most commonly used for a Web service is a stock quote service in which the request asks for the price of a specific stock and the response gives the stock price.

In an OOM, you design a Web service as a *component* (EJB, servlet, or standard component) that includes a Web service implementation class.

When you work with Web services in an OOM, you use the class, component and deployment diagrams, which allow you to:

- Create new Web Service component
- Reverse engineer WSDL to create Web Service component
- Browse UDDI to search WSDL
- Generate WSDL from Web Service component definition
- Generate server side Web Services code for Java (AXIS, JAXM, JAX-RPC, Web Services for J2EE) or .NET (C# and VB .NET)
- Generate client proxy for Java or .NET
- Reverse engineer for Java and .NET

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. The WSDL-to-Java and Java-to-WSDL tools are used by the WSDL for Java extension file'. For example, the WSDP (Web Service Developer Pack) provides a XRPC tool, Apache AXIS provides a `wsdl2java` and a `java2wsdl` tool (which can be downloaded from the Sun Java Development Site: <http://java.sun.com/index.jsp>). Apache AXIS can be downloaded from: <http://ws.apache.org/axis>.

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.

Defining Web Services Tools

A Web Service is an interface that describes a collection of operations that are accessible on the network through SOAP messages.

Web services operate over the Internet or a corporate intranet in the exact same way as you locate a web site: either you type in the URL address or you use a search engine to locate the

site. You may know the address of the Web service you want to invoke, the address of its interface (WSDL for example), or you must search for the service by querying a Web service registry. The UDDI specification defines a standard mechanism for publishing and locating the existence of businesses and the services they provide.

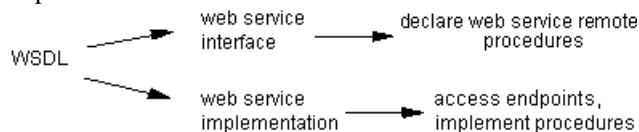
WSDL is a language that describes what a Web service is capable of and how a client can locate and invoke that service. The Web Services Description Language (WSDL) 1.1 document, available at <http://www.w3.org/TR/wSDL>, describes WSDL documents as follows:

"A WSDL document defines services as collections of network endpoints, also called ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment. This allows the reuse of abstract definitions: *messages* are abstract descriptions of the data being exchanged, and *port types* are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports defines a service. Hence, a WSDL document uses the following elements in the definition of network services:

- *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"

Interface and Implementation

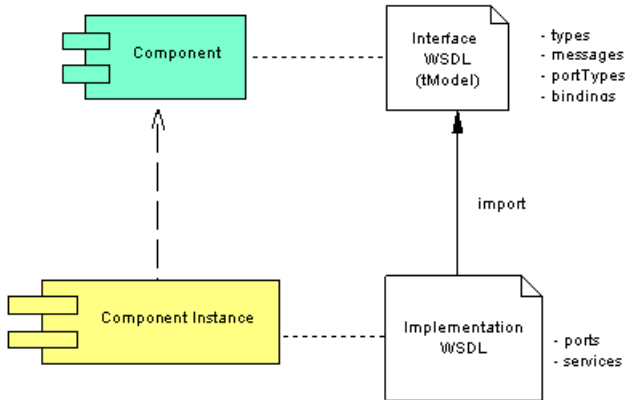
WSDL is used to define the Web service interface, the Web service implementation, or both. As a result, it is possible to use two WSDL files, one for the interface and one for the implementation.



In an *interface WSDL*, you declare the procedures that allow you to create a Web service.

In an *implementation WSDL*, you define how to implement these procedures through services and ports (access endpoints URLs).

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.



For detailed information about WSDL, see <http://www.w3.org/2002/ws/desc>.

Simple Object Access Protocol (SOAP)

SOAP is a protocol based upon XML for exchange of information in a distributed environment. It represents the invocation mechanism within a Web service architecture. WSDL allows a user to understand which format of the SOAP message should be sent to invoke the service and what is the expected return message format.

Universal Description Discovery and Integration (UDDI)

UDDI is an XML-based registry for businesses worldwide. This registry lists all Web services on the Internet and handles their addresses.

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 in the UDDI the following items that describe how to invoke the Web service:

- The company, called *businessEntity*
- The service, called *businessService*
- The access endpoints, called the *bindingTemplate*
- The specification, called the *tModel*

Supported Web Services

For Web services implemented using .NET, PowerDesigner generates .asmx files for C# or VB .NET, WSDL files and client proxies in C# or VB .NET.

For Web services implemented using Java, PowerDesigner allows you to use one of the following models: AXIS, JAXM, JAX-RPC and Web Services for J2EE (Stateless Session Bean).

Defining Web Services Targets

The Java and .NET (C# and Visual Basic .NET) OOM languages support Web services.

In general, a WSDL can be generated by the server where the Web service is deployed. As a result, the WSDL generated by the server contains both the interface and implementation definition.

When you work with Java, C# or VB.NET in an OOM, an extension file is automatically attached to the model to complement the definition of these languages within the context of Web services.

Defining Web Service Components

A Web service is represented as a component that you can display in a component diagram. From a component diagram, you can display the Web service interface and implementation code, eventually, you can also deploy Web service components to nodes if you want to describe deployment of components into servers.

For more information on how to deploy components, see *Deploying a Component to a Node* on page 220.

A component can be a Web service Interface or a Web service Implementation type. You have to check the Web Service check box in the component property sheet to declare your component as a Web service.

The following Web service types are supported for the Java language:

- *Java Web Service*: exposes a Java class with the .jws extension as a Web service using Apache Axis.
- *Axis RPC*: exposes a Java class as a Web service using the Apache Axis RPC model.
- *Axis EJB*: exposes a Stateless Session Bean as a Web service using the Apache Axis EJB model.
- *JAX-RPC*: uses a Java class and an interface to implement the JAX-RPC model.
- *JAXM*: exposes a Servlet as a Web service using JAXM.
- *Web Service for J2EE*: exposes a Stateless Session Bean as a Web service using the Web Service for J2EE (JSR109) model.

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. 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 (soap:xxx) or XML-Schema (xsd:xxx) 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* on page 247).

Component UDDI/Extended Attributes Tab

These tabs include the following properties:

Name	Description
SOAP binding style	Defines the SOAP binding style. It could be a document or rpc Scripting name: SoapBindingStyle
SOAP binding transport	Defines the SOAP binding transport URI Scripting name: SoapBindingTransport
SOAP body namespace	Defines the namespace of the XML schema data types in the WSDL Scripting name: SoapBodyNamespace
Business name	Stores the name of the business found in a UDDI registry Scripting name: BusinessName
Business description	Stores the business description of the Web Service found in a UDDI registry Scripting name: BusinessDescription
Business key	Stores the key of the business found in a UDDI registry Scripting name: BusinessKey
Namespaces	Stores additional namespace that are not automatically identified by PowerDesigner Scripting name: Namespaces
Service name	Stores the name of the service found in a UDDI registry Scripting name: ServiceName
Service description	Stores the service description of the Web Service found in a UDDI registry Scripting name: ServiceDescription
Service key	Stores the key of the service found in a UDDI registry Scripting name: ServiceKey
tModel name	Stores the name of the tModel found in a UDDI registry Scripting name: tModelName

Name	Description
tModel key	Stores the key of the tModel found in a UDDI registry Scripting name: tModelKey
tModel URL	Stores the URL of the tModel found in a UDDI registry. It allows you to retrieve the WSDL Scripting name: tModelURL
UDDI operator URL	Stores the URL of the UDDI registry operator URL used to find the WSDL 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.

Creating a Web Service with the Wizard

You can create a Web service with the *wizard* that will guide you through the creation of the component. The wizard is invoked from a *class diagram* and is only available if you use the Java or the .NET family.

You can either create a Web service without selecting any class, or use the standard approach that consists in selecting an existing class first and then start the wizard from the contextual menu of the class.

You can also create several Web services of the same type by selecting several classes at the same time. The wizard will automatically create one Web service per class. The classes you have selected in the class diagram become Web service classes, they are renamed to match the naming conventions standard, and they are linked to the new Web service component.

Note: You must create the Web service component within a package so that the package acts as a namespace.

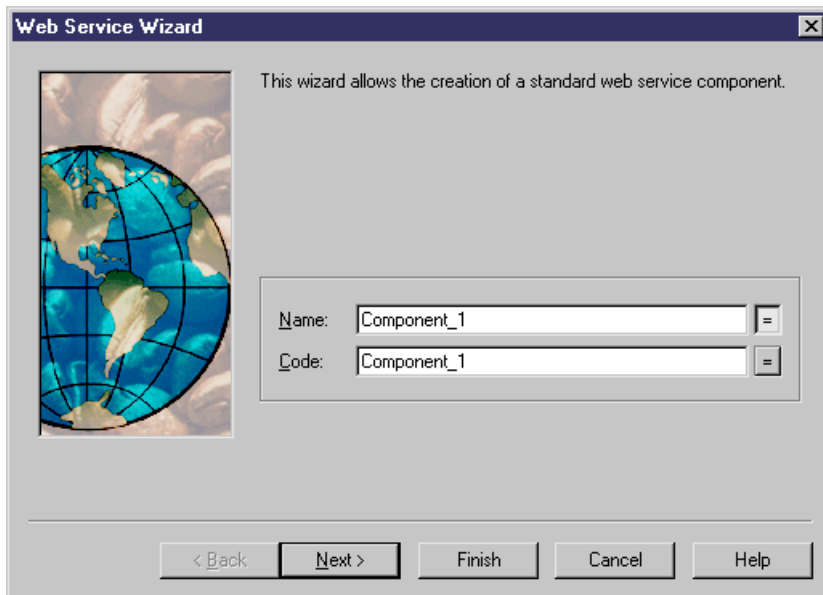
The wizard for creation of Web services lets you define the following parameters:

Property	Description
Name	Name of the Web service component
Code	Code of the Web service component
Web service type	Interface or Implementation. Interface refers to a component that defines the service interface only. Implementation refers to a component that implements a service interface
Component type	The component type depends on the web service type. You can select among a list of web service interface or implementation protocols

Property	Description
Web service implementation class	Defines the class that is used to implement the Web service
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
Create Class Diagram for component classifiers	Available only for stateless session beans and servlets. It 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

1. Select **Tools > Create Web Service Component** from a class diagram.

The Web Service Wizard dialog box is displayed.



2. Type a name and code for the Web service component and click Next.

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.

When you create a Web service with the wizard, you have to select the type of the Web service and the type of the component that is used as a source. The following table maps available component types to Web service types in Java:

Component type	Java service interface	Java service implementation	Use
Standard		—	Java class
Axis RPC	—		Java class
Axis EJB	—		EJB Stateless Session Bean
Java Web Service (JWS)	—		Java class with .jws extension
JAXM	—		Java Servlet
JAX-RPC	—		Java class
Web Service for J2EE	—		EJB Stateless Session Bean

= allowed

— = not allowed

3. Select a Web service type and a component type and click Next.
4. Select a Web service implementation class and click Next.
5. At the end of the wizard, you have to define the creation of symbols.

When you have finished using the wizard, the following actions are executed:

- A Web service component flagged as 'Web service' is created
- A Web service implementation 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
- A default operation with the "Web method" flag is created
- Depending on the component type, required interfaces associated with the component are added

Creating a Web Service from the Component Diagram

You can also create a web service from the component diagram.

1. Click the **Component** tool in the Toolbox and click in the diagram to create a component.
2. Click the **Pointer** tool or right-click to release the **Component** tool.
3. Double-click the component symbol to display the property sheet.
4. Select the Web Service check box in the **General** tab.
5. Click OK.

Defining Data Types for WSDL

WSDL uses XML Schema to define data types for message structures.

WSDL Data Type Mappings

To generate WSDL, it is necessary to map Java or .NET types to XML types.

In an OOM, there are three data type maps defined in the WSDL extension file.

- WSDL2Local - converts WSDL types to Java or .NET types
- Local2SOAP - converts Java or .NET types to SOAP types for SOAP encoding
- Local2XSD map - converts Java or .NET types to XML Schema types for XML Schema encoding

Selecting WSDL Data Types

If you need to use a specific data type, you can select the WSDL data type for the operation return type or the parameter type.

You can select a WSDL data type from the list in the operation and parameter property sheets. This box includes basic data types for XML Schema encoding or SOAP encoding.

You can also click the Properties button beside the WSDL data type box to open the WSDL Schema tab in the component property sheet. This tab shows the contents of the WSDL schema.

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 Preview tab in the property sheet of the class to verify the code at any time.

1. From the Web Method tab in the operation property sheet, select or type a new data type in the WSDL data type box.

or

From the parameter property sheet, select or type a new data type from the WSDL data type box.

2. Click Apply

Declaring Data Types in the WSDL

Classes used as data types are declared as Complex Types inside the <types> section of WSDL.

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)

Managing Web Service Methods

You can define one or several methods as part of a Web Service. In PowerDesigner, you use operations to create Web service methods.

Creating a Web Service Method

A Web service method is an operation with the Web Service Method property selected.

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.

Web service methods can belong to the component implementation class or to component interfaces.

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.

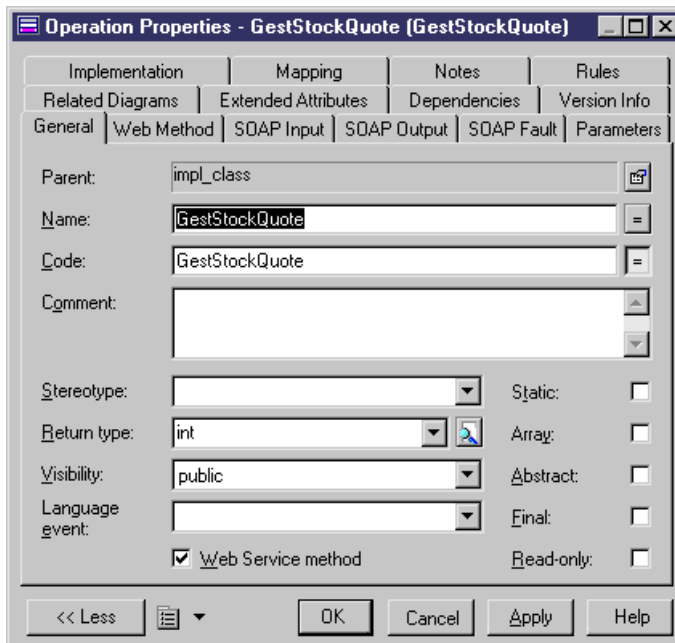
Interface methods use the same extended attributes as classes methods for WSDL customization, as explained in *Defining Web service method extended attributes* on page 246.

Three extended attributes are used to decide which Port Type should be generated: SOAPPortType, HttpGetPortType and HttpPostPortType. 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.

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* on page 241 and *Implementing a Web service method in .NET* on page 246.

1. Open the property sheet of the Web Service class or interface.
2. Click the Operation tab, then click the Insert a row tool to create a new operation.
3. Click Apply and click the Properties tool to display the operation property sheet.
4. Select the Web Service method check box in the General tab.



5. Click OK.

Web Service Method Properties

When the Web Service method check box is selected in the operation property sheet, a number of additional tabs are displayed.

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 class.
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 - defines the name and schema of the SOAP input message element.
- SOAP Output - defines the name and schema of the SOAP output message element.
- SOAP Fault - defines the default name and schema of the SOAP fault message element.

Implementing a Web Service Method in Java

To implement a Web service method, you have to define the following:

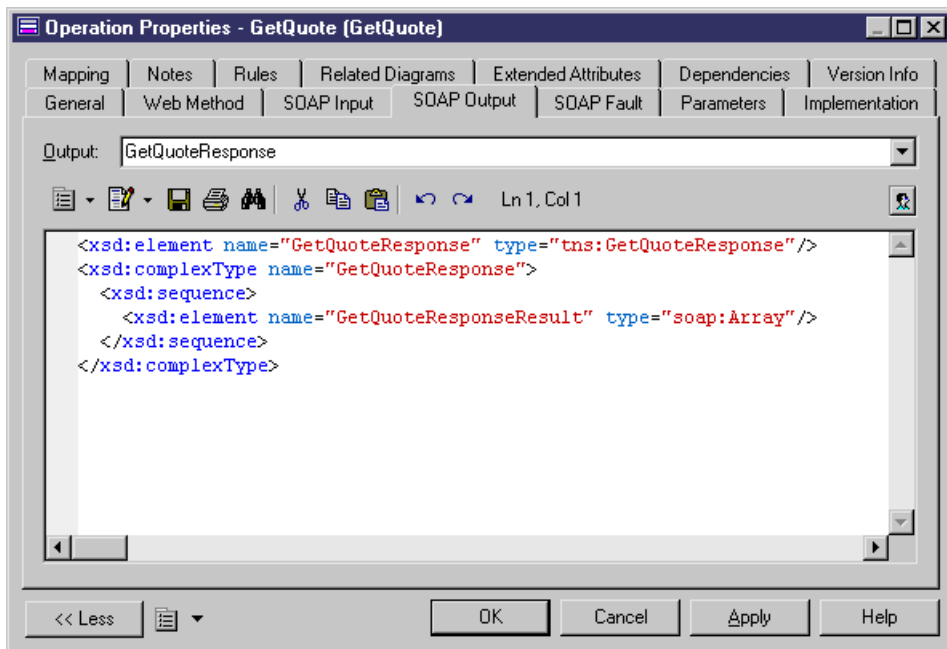
- The operation return type
- The operation parameters
- The operation implementation

Defining the Return Type of an Operation

To define the return type of an operation you have to specify:

- *The return type for the Java method:* in the General tab of the operation property sheet. If the return value in Java 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 value, the output message type for WSDL can be defined in the Web Method tab of the operation property sheet

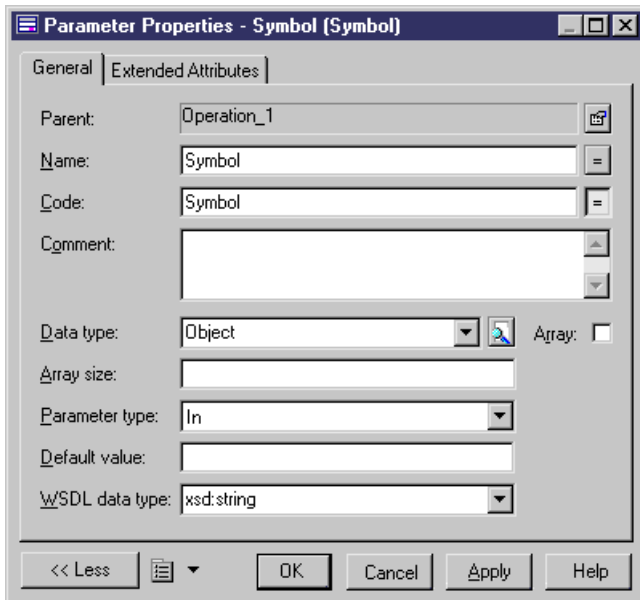
For more complex return types, you can manually define the output message schema using the SOAP Output tab of the operation property sheet.



Defining the Parameters of an Operation

To define the parameters of an operation, you have to specify:

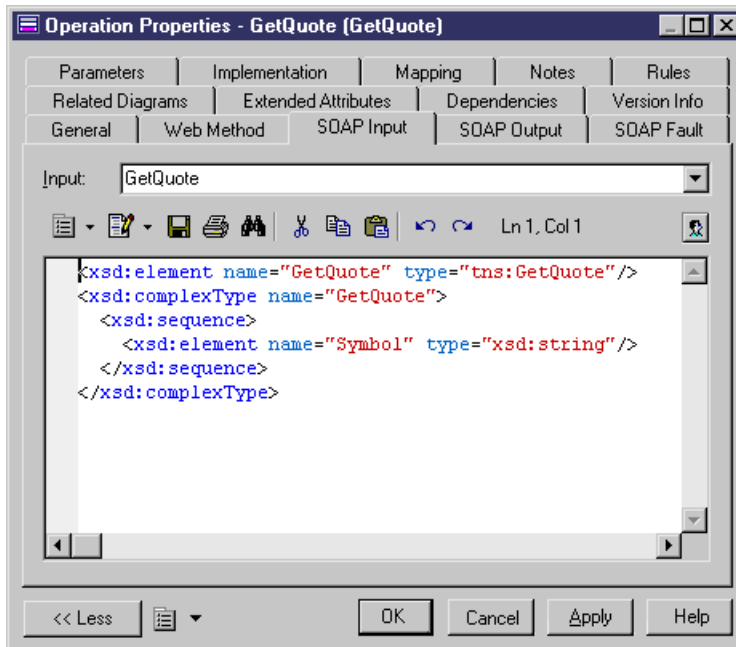
- The *parameters type* for the Java method using the Data Type list in the Parameter property sheet
- The *input message type* for WSDL using the WSDL Data Type list in the Parameter property sheet. This list displays basic data types from the object language and complex data types from the WSDL Schema. 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



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 detailed information about SOAP, see <http://www.w3.org/2000/xml/Group>.

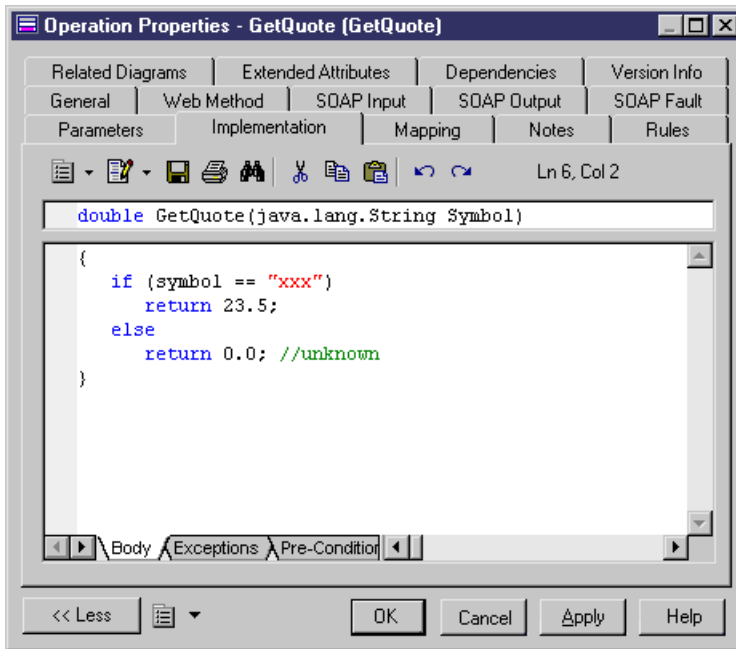
For more complex parameter types and input message types, you can manually define the input message schema using the SOAP Input tab in the operation property sheet.



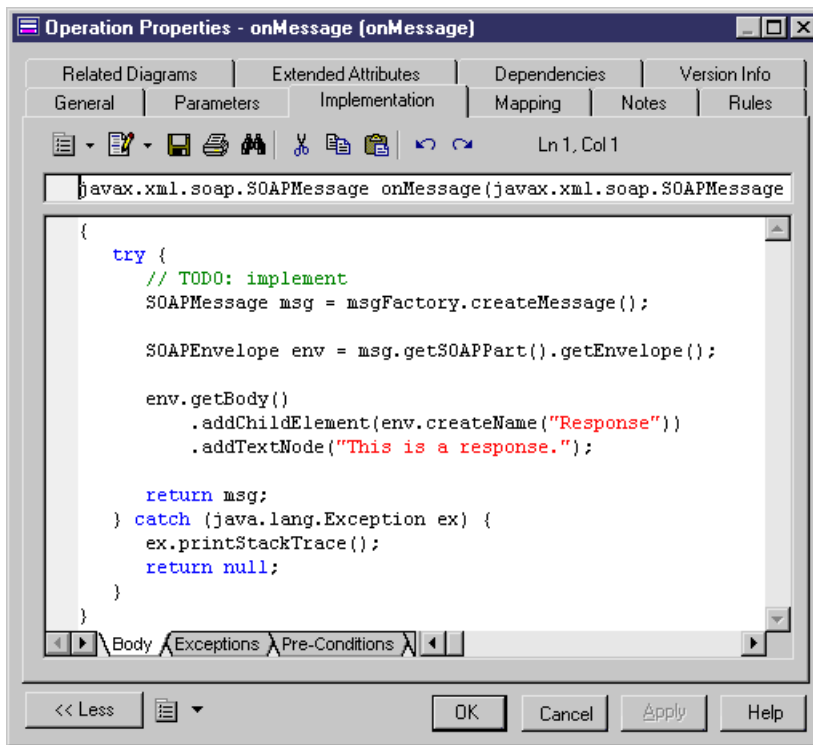
Implementing the Operation

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.



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* on page 241.

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.

Defining Web Service Method Extended Attributes

You can customize the WSDL generation using extended attributes. They allow you to modify input message name, output message name, SOAP operation style, port types to generate, SOAP action, Web method type, and so on.

1. Open the operation property sheet.

2. Click the WSDL attributes tab and modify extended attributes.

Operation Properties - GetQuote (getQuote)

General | Web Method | SOAP Input | SOAP Output | SOAP Fault | Parameters
 Implementation | Mapping | **WSDL attributes** | JavaDoc | Notes | Extended Attributes

HTTP get port type
 HTTP post port type

Input HTTP get message name: %Code%HttpGetIn
 Input HTTP post message name: %Code%HttpPostIn
 Input SOAP body parts:
 Input SOAP body use: literal
 Input SOAP message name: %Code%SoapIn
 Output HTTP get message name: %Code%HttpGetOut
 Output HTTP post message name: %Code%HttpPostOut
 Output SOAP body parts:
 Output SOAP body use: literal
 Output SOAP message name: %Code%SoapOut
 Port type: SOAP and HTTP
 Return MIME type: text/xml
 Return name: %Code%Result
 SOAP action:
 SOAP body namespace:
 SOAP operation style: document
 SOAP port type
 Web method type: Request-Response

More >> | [List Icon] | OK | Cancel | Apply | Help

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 *Chapter 9, Checking an OOM* on page 295).

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.

Defining Web Service Component Instances

The deployment diagram is used with Web services to model the Web services deployment. This diagram is useful if you want to deploy a Web service into one or several servers, and if you want to know which Web service is deployed where, as deployment diagrams can display network addresses, access endpoints and access types.

A component instance defines the ports, the access type, and the access endpoint that is the full URL to invoke the Web service.

When the Web Service check box is selected in the property sheet of the component instance, it means that the component instance is an instance of a Web service component. A component instance that inherits from a component inherits its type: the type of the component is displayed in the property sheet of the component instance.

When the Web Service check box is selected, a Web Service tab and a WSDL tab automatically appear in the property sheet of the component instance.

For more information on the deployment diagram, see *Chapter 5, Implementation Diagrams* on page 211.

Web Service Tab of the Component Instance

The Web Service tab in the property sheet of the component instance includes the following properties:

Property	Description
Access point URL	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 User-Defined tool to the right of the box. Once clicked, the URL can be overridden
WSDL URL	Indicates where the WSDL should be published on the web. You can type your own URL by using the User-Defined tool to the right of the box. Once clicked, the URL can be overridden
Use external WSDL	Indicates that the WSDL is published at a specific URL

When a WSDL is not user-defined, it is regenerated each time and can be displayed in the Preview tab of the component instance property sheet. An external WSDL has a user-defined WSDL text.

Access Point URL Examples

Here are some examples of the syntax used in .NET and Java:

For .NET, the default syntax is:

```
accesstype://machine_networkaddress:port/application_namespace/
webservice_code.asmx
```

For example: <http://doc.sybase.com:8080/WebService1/StockQuote.asmx>.

For Java, the default syntax is:

```
accesstype://machine_networkaddress:port/application_namespace/
webservice_code
```

For example: <http://doc.sybase.com/WebService1/StockQuote>.

Computed attributes `AccessType` and `PortNumber` for code generator & VBScript are computed from the access point URL. For example: `http`, `https`.

WSDL URL Examples

For .NET, the default syntax is:

```
accesstype://machine_networkaddress:port/application_namespace/
Webservice_code.asmx?WSDL
```

For Java, the default syntax is:

```
accesstype://machine_networkaddress:port/application_namespace/
wsdl_file
```

WSDL Tab of the Component Instance

The WSDL tab in the property sheet of the component instance includes 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
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 User-Defined tool among the available tools, you make the contents user-defined. Once clicked, the contents can be overridden

Note: You can display some of the tabs of the property sheet of a component instance by right-clicking the component instance symbol in the diagram and selecting the appropriate tab from the contextual menu.

Using Node Properties

The node property sheet includes the following property, specific to Web services:

Property	Description
Network address	Address or machine name. For example: <code>doc.sybase.com</code> , or <code>123.456.78.9</code>

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.

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.

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.

PowerDesigner supports the JAXM, JAX-RPC, Web Service for J2EE (JSR 109), AXIS RPC, EJB, and Java Web Service (JWS) types of Java Web services.

Generating JAXM Web Services

JAXM is a Java API for XML messaging that provides a standard way to send XML documents over the Internet from the Java platform.

If the Web service implementation type is JAXM, PowerDesigner uses the JAXM model for implementation. JAXM Web Service components provide the flexibility for handling complex message formats.

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.

To Use JAXM, you can use the Java Web Services Developer Pack (JWS DP) 1.1 or higher from Sun or a Servlet container or J2EE server that supports JAXM. You can download JWS DP from <http://java.sun.com/webservices/>.

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. You can also define a JAVACLASSPATH variable in PowerDesigner to define the classpath specific to PowerDesigner, in this case the JAVACLASSPATH variable replaces the CLASSPATH environment variable

You need an application server or a Servlet container that supports JAXM. JWS DP ships with Apache Tomcat that supports JAXM.

1. Select **Language > Generate Java Code**.
2. Select a directory where you want to generate the code.
3. In the Tasks tab, select the command Java: Package J2EE application in an EAR file. This command will create a .WAR file and a .EAR file.

4. Click OK.

Generating JAX-RPC Web Services

JAX-RPC is a Java API for XML-based RPC (Remote Procedure Calling protocol). It facilitates RPC over the Internet allowing XML formatted parameters to be passed to remote services and allowing XML formatted values to be returned.

If the Web service implementation type is JAX-RPC, PowerDesigner uses the JAX-RPC model for implementation. JAX-RPC defines RPC type invocation for Web Services but it is limited to simple message formats. You can use very complex Objects/XML mapping.

Using the JAX-RPC model implies to:

- Generate the Web Service Java class and interface code
- Compile the Web Service Java class and interface
- 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
- Deploy the .WAR file in a server that supports JAX-RPC

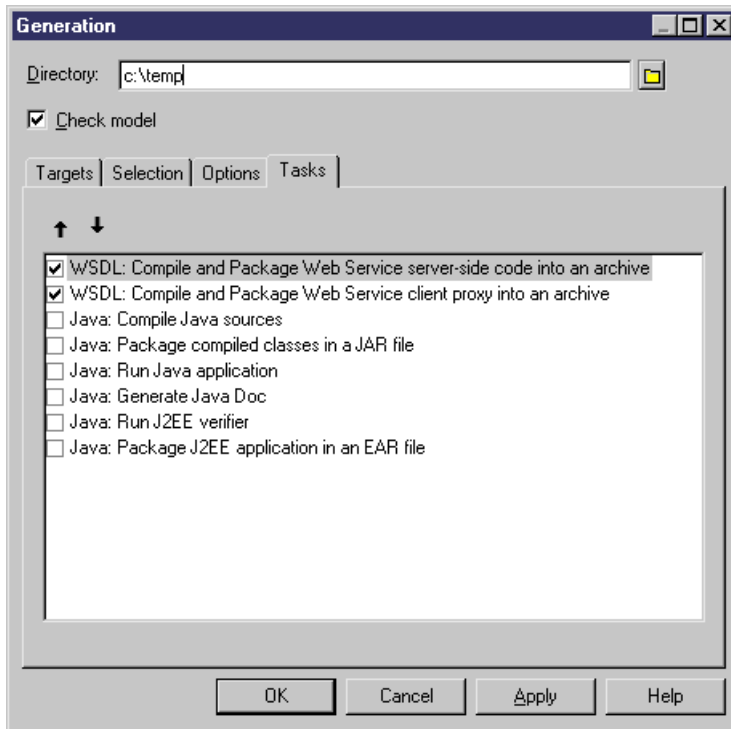
To Use JAX-RPC, you can use the Java Web Services Developer Pack (JWSDP) 1.1 or higher from Sun or other application servers that support the JAX-RPC model. You can download JWSDP from <http://java.sun.com/webservices/>.

To generate server side code and client proxy for JAX-RPC, if you use JWSDP, you can use the wscompile.bat tool. For other JAX-RPC compatible implementations, please refer to the documentation. To invoke the wscompile.bat tool from PowerDesigner, you have to define an environment variable WSCOMPILER in the Variables category located in the General Options windows (**Tools > General Options**). The WSCOMPILER variable should indicate the full path of the wscompile.bat file. To run wscompile.bat, the jaxrpc-api.jar file must be in your CLASSPATH environment variable. You can also define a JAVACLASSPATH variable in PowerDesigner to define the classpath specific to PowerDesigner, in this case the JAVACLASSPATH variable replaces the CLASSPATH environment variable

To deploy JAX-RPC Web service components, you need an application server or a Servlet container that supports JAX-RPC. JWSDP ships with Apache Tomcat that supports JAX-RPC

1. Select **Language > Generate Java Code**.
2. Select a directory where you want to generate the code.
3. In the **Tasks** tab, 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.

These commands will compile the Java classes generated by PowerDesigner, run the WSCOMPILER tool and create a .WAR file.



4. Click **OK**.

Generating Stateless Session Bean Web Services

PowerDesigner supports the Web Services for J2EE specification that defines the programming model and runtime architecture for implementing Web services.

In Java, Web services may be implemented either through JAX-RPC endpoints or EJB stateless session bean components. Both of these implementations expose their Web methods through a service endpoint interface (SEI).

JAX-RPC endpoints are considered as Web components, they are represented as *servlets* in the OOM and are packaged into a WAR, while EJB *stateless session beans* are packaged into an EJB JAR. 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.

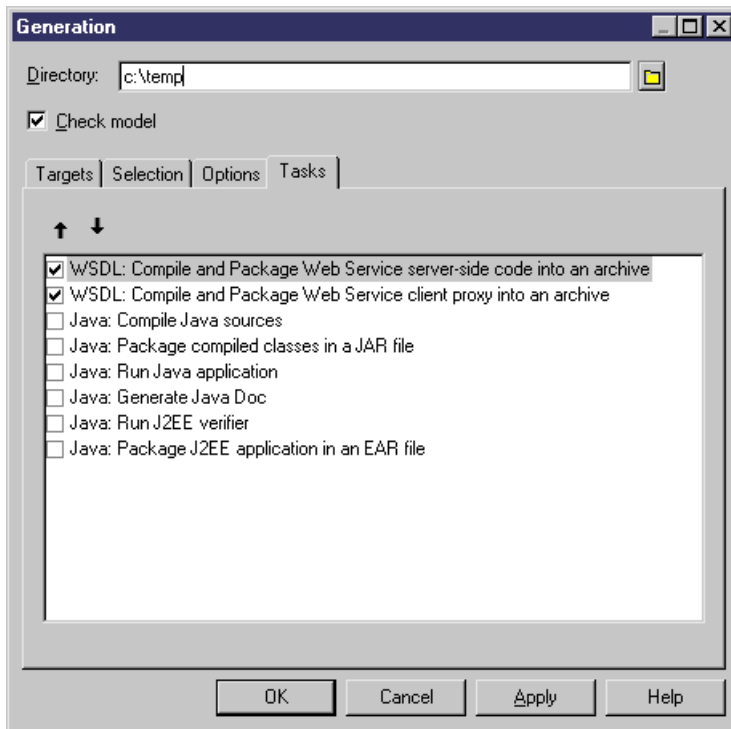
If the Web service implementation type is Stateless Session Bean, PowerDesigner uses Web Services for J2EE specification for implementation.

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.

For more information on JAX-RPC, see *Generating JAX-RPC Web services* on page 252.

1. Select **Language > Generate Java Code**.
2. Select a directory where you want to generate the code.
3. In the Tasks tab, to generate server side code, select the command WSDL: Compile and Package Web Service Server-Side Code into an archive. To generate client proxy, select the command WSDL: Compile and Package Web Service Client Proxy into an archive.

These commands will compile the Java classes generated by PowerDesigner, run the WSCOMPILER tool and create a .WAR file.



4. Click OK.

Generating AXIS RPC Web Services

If the Web service implementation type is 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. If the provider style is <Default>, PowerDesigner will automatically select the best provider style. To select the provider style, you can change the AxisProviderStyle extended attribute of the Web service component.

To customize Axis deployment descriptor generation, you can change several Axis specific extended attributes in the Web service component property sheet.

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.

1. Select **Language > Generate Java Code**.
2. Select a directory where you want to generate the code.
3. [optional] Click the **Selection** tab and select the objects that you want to generate on the various sub-tabs. By default, all objects are generated.
4. [optional] Click the **Options** tab and select any appropriate generation options.
5. [optional] Click the **Tasks** tab and select any appropriate tasks to perform during generation
6. Click **OK** to begin generation.

A progress box is displayed during generation. When the generation is complete PowerDesigner displays a list of the generated files that you can edit. The results are also displayed on the **Generation** tab of the Output window.

Generating AXIS EJB Web Services

If the Web service implementation type is 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 customize the generation of the Axis deployment descriptor, you can change several Axis specific extended attributes in the Web service component properties.

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.

To expose a Stateless Session Bean as Web service using Axis, you need to:

- Generate the EJB code
 - Compile and package the EJB
 - Deploy the EJB in a J2EE server
 - Expose the EJB as a Web Service using Axis
1. Select **Language > Generate Java Code**.
 2. Select a directory where you want to generate the code.
 3. In the options tab, you can modify generation options and deployment options.
 4. In the Tasks tab, you can select the commands in the following order: Package J2EE application in an EAR file, Deploy J2EE application, Expose EJB as Web Services.

Generating Java Web Services (JWS)

If the Web service implementation type is Java Web Service (JWS), PowerDesigner uses a Java class for implementation. The Java class will have the .jws extension.

To deploy the Java class, you can simply copy the .jws file to the server that supports Java Web Service format. For example, for Apache Axis, you can copy the .jws file to the directory webapps\axis.

Testing Web Services for Java

To test a Java Web Service, there are several methods:

- Send SOAP message. You can write a Java program to send SOAP message to a Web Service and process the returned output SOAP message using the SAAJ API
- Use Dynamic Invocation. You can use the Dynamic Invocation method defined by JAX-RPC
- Use Dynamic Proxy. You can use the Dynamic Proxy method defined by JAX-RPC
- Use Client Proxy. You can use a client proxy to invoke a Web Service easily. 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

Generating Web Services for .NET

The following items are generated in .NET languages (C# and VB.NET):

- 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
- A .ASMX file

Note

It is not necessary to define the super class (also known as WebService) for the Web service classes; if the WebService super class is not defined, the code generator adds it to the code.

When generating the server side code, you can use the following default options and tasks that help you automatically start generating with pre-defined characteristics.

Defining Web Services Generation Options in .NET

You can select Web services generation options available for .NET languages by starting the Generate object language command in the Language menu.

C# Generation Options

The following options are available from the Options tab in the Generation dialog box for C#:

Option	Description
Generate C# web Service code in .ASMX	Generates the C# code in the .ASMX file

VB.NET Generation Options

The following options are available from the Options tab in the Generation dialog box for VB.NET:

Option	Description
Generate VB.NET Web service code in .ASMX	Generates the Visual Basic.NET code in the .ASMX file

Defining Web Service Generation Tasks in .NET

You can select Web services generation tasks available for .NET languages by starting selecting **Language > Generate object language**.

The following tasks are available from the Tasks tab in the Generation dialog box for C# and VB.NET:

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. You need to define a component instance for the Web Service deployment URL
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

Generating Web Services in .NET

The deployment of a Web service consists in copying the generated implementation class and .ASMX file into the directory of the web server virtual folder.

The .ASMX is an ASP.NET file, it contains the code of the C# or VB.NET Web service class.

1. Select **Language > Generate C# Code** or **Generate VB .NET Code**.
2. Select the generation directory. You can select a Microsoft Internet Information Server (IIS) directory for generation, for example, C:\Inetpub\wwwroot\StockQuote. If you have defined you Web Services inside a package, you can generate the Web Services code in the C:\Inetpub\wwwroot directory. Each package will create a subdirectory.

3. Set the Generate Web Service C# code in .asmx file option to false in the Options tab if you want to generate the C# or VB .NET class outside a .asmx file.
4. Select the Compile C# code or VB .NET code command in the Tasks tab if you generate the C# or VB .NET Web Service class outside a .asmx file.
5. Click OK.

The code generation process creates a subdirectory under wwwroot using the package name, and creates a <WebServiceName>.ASMX file within the subdirectory.

Generating a .NET Proxy Class for a Web Service

PowerDesigner can also generate a client proxy class to simplify the invocation of the Web Service. To generate the client proxy class, PowerDesigner uses the wsdl.exe program that comes with Visual Studio .NET. You have to define a WSDL variable to indicate where the wsdl.exe program is located.

Define the WSDL Variable

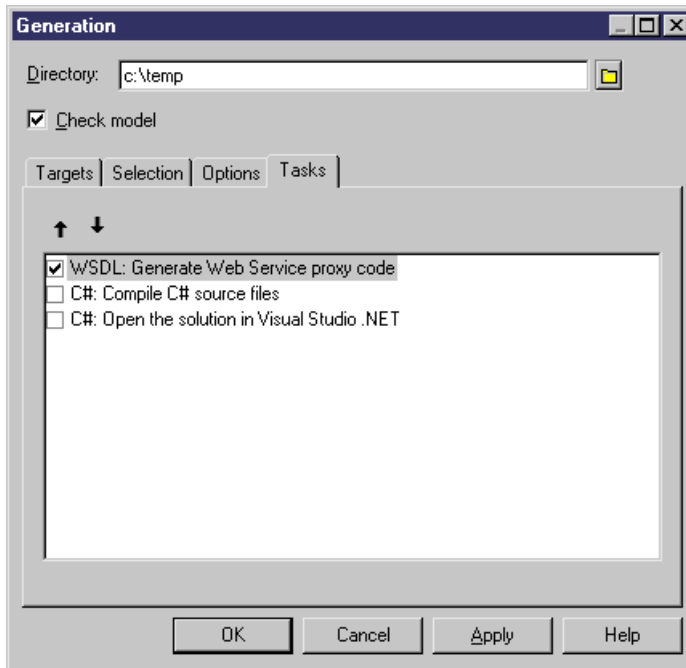
To define a WSDL variable:

1. Select **Tools > General Options**.
2. Select the Variables category.
3. Add a WSDL variable in the Name column.
4. Browse for the wsdl.exe file in the Value column.
5. Click OK.

Generate the Client Proxy Classes

To generate the client proxy classes:

1. Select **Language > Generate C# Code** or **Generate VB .NET Code**.
2. Open the Tasks tab.
3. Select the command WSDL: Generate Web Service proxy code.



4. Click OK.

Deploying Web Services in .NET

To deploy and test a Web Service for .NET, you have to install Microsoft Internet Information Server (IIS). In the machine where you install IIS, you also have to install the .NET Framework. You can download .NET Framework or .NET Framework SDK from the Microsoft web site.

To deploy the generated Web Service code, you simply copy the .asmx file and the C# or VB .NET class files under the IIS directory `C:\Inetpub\wwwroot<PackageName>`, where `<PackageName>` is the name of the package. For example: `C:\Inetpub\wwwroot\StockQuote`.

Testing Web Services for .NET

To test the Web Service, you have to enter the URL of the Web Service in the browser: `http://[HostName]/[PackageName]/[ServiceName].asmx`. For example: `http://localhost/StockQuote/StockQuote.asmx`.

The IIS Web server will generate a testing tab to let you test the deployed Web Service if the input parameters and the return value use simple data types.

To test Web Services with complex data types, you have to create a testing program using Web Service proxy or use a tool to send a SOAP message to the Web Service.

Generating Web Services for Sybase WorkSpace

Sybase WorkSpace provides an integrated development environment to develop, test and deploy Web services. PowerDesigner allows you to generate Java and EJB Web services and use the Sybase WorkSpace utilities to fine-tune their implementation.

You can design your Web services using the standard PowerDesigner environment or using the PowerDesigner Eclipse plugin that runs in the WorkSpace environment (see *Core Features Guide > The PowerDesigner Interface > The PowerDesigner Plugin for Eclipse*).

PowerDesigner helps you to quickly:

- Create a Web service with the correct implementation class
- Define the Java class package
- Generate the Web service and open it in the Workspace Java Service Editor

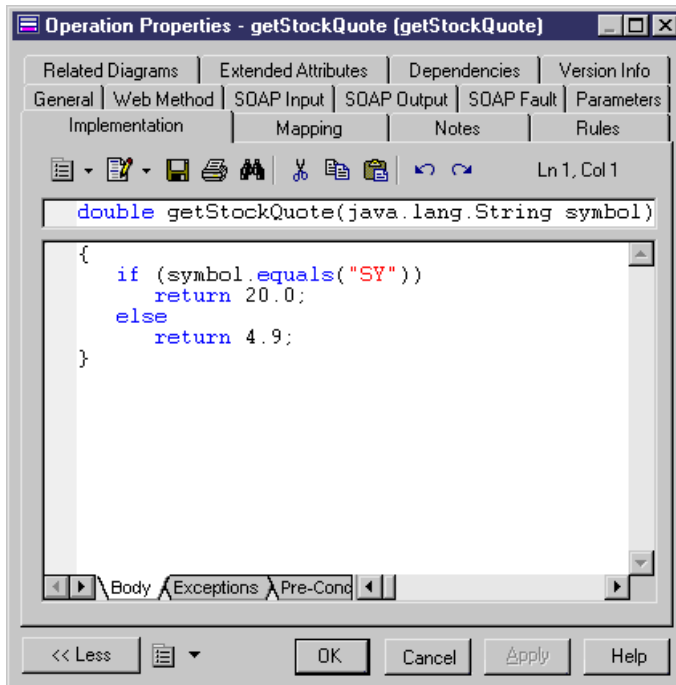
Creating a Java or EJB Web Service for Sybase WorkSpace

PowerDesigner provides full support for creating a Java Web service for Sybase WorkSpace.

1. Select **File > New Model** to open the New Model dialog, and select Object Oriented Model in the **Model Type** list.
2. Select Java in the **Object language** list and select Class diagram in the **Diagram** list.
3. Click the **Select Extensions** button, click the **IDE** sub-tab, select the Sybase WorkSpace xem, and click **OK** to return to the New Model dialog.
4. Click **OK** to create the OOM.
5. Select **Tools > Create Web Service Component** to open the Web Service Wizard.
6. Type a name and a code for the component and click **Next**.
7. Select Implementation in the **Web Service Type** list and select one of the following in the **Component Type** list:
 - Java Web Service (JWS)
 - Axis EJB (Stateless Session Bean)
8. Click **Next**, select a Web service implementation class, and then click **Finish**.

The Web service component is created together with the implementation class and the corresponding Web method.

9. Double-click the implementation class in the diagram and open the **Operations** tab in the class property sheet.
10. Double-click the WebMethod created by default. The **Web Service Method** check box is selected. You can rename the operation if necessary.
11. Click the **Implementation** tab and enter the implementation of the Web service:



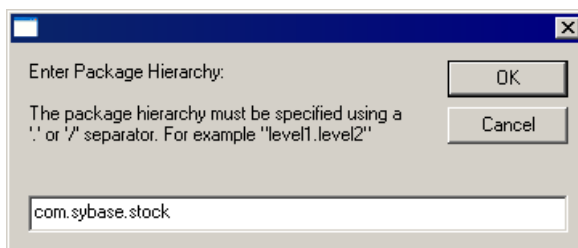
12. Click **OK** in each of the dialog boxes.

You can preview the code that will be generated for the Web service by clicking the **Preview** tab of the associated class or component. You can customize the Web service by editing the properties of the component.

Defining the Java Class Package

When you create a Web service using the Web Service Wizard, the Java class and the Web service component are both created at the model level. You can, if necessary, define a package name for your Java class and component.

1. Right-click the model in the Browser, and select **Add Package Hierarchy** from the contextual menu.



2. Enter the appropriate package hierarchy and click **OK** to create it in the model.

Generating the Java or EJB Web Service for Sybase WorkSpace

PowerDesigner can generate all the files that you need to work with to define the Web service in the Workspace Java Service Editor.

1. Select **Language > Generate Java code** to open the Generation dialog.
2. Enter a destination directory to which to generate the files.
3. Ensure that the Sybase WorkSpace IDE is selected on the **Targets** tab.
4. Click **OK** to generate the following files:
 - A `svc_java` or `svc_ejb` file that defines the Web service
 - A `.java` file for the implementation of the Web service
 - A `.project` file for creating a Java project if it does not exist
 - A `.classpath` file for defining Java libraries in a Java project

When you generate using the PowerDesigner Eclipse plugin an Eclipse project will be created or, if you are generating to an existing project, the project will be refreshed to show the new files.

5. In Workspace, right-click the `.svc_java` file in the Navigator window and select **Open with > Java Service Editor**.
6. Click the **Interface** sub-tab and continue with the implementation of the service.

For more information on the Java Service Editor, see your Sybase Workspace documentation.

Understanding the .svc_java or .svc_ejb File

PowerDesigner generates a `svc_java` or `svc_ejb` file for each Web service.

Each file contains the following fields:

Field	Component property
<code>serviceName</code>	Web service component code
<code>serviceComment</code>	Web service component comment
<code>projectName</code>	[<code>svc_java</code> only] Last directory name of the generation full path
<code>projectPath</code>	[<code>svc_ejb</code> only] The project directory and folder name
<code>serviceFilename</code>	% <code>serviceName</code> %. <code>svc_java</code>
<code>authorName</code>	Web service component modifier name
<code>dateCreated</code>	Generation date and time in the format: Mmm dd, yyyy hh:mm:ss AM/PM
<code>ejbFullName</code>	[<code>svc_ejb</code> only] <PackageCode>.<ComponentCode>

Field	Component property
operationName	Web method code
operationComment	Web method comment
static	[svc_java only] Use "METHOD_TYPE_STATIC" if the web method is static
inputMessage	Input message name
outputMessage	Output message name
returnType	Web operation return type
parameterName	Operation parameter code
parameterComment	Operation parameter comment
dataType	Operation parameter data type
javaServiceParamType	[svc_java only] Web operation parameter type
classFullPath	%projectName %/ %qualifiedClassName%
qualifiedClassName	Fully qualified Java class file name
endpointName	[svc_ejb only] End point name
connectionName	[svc_ejb only] Application server connection profile name.
EJBComponentURI	[svc_ejb only] EJB component URI
jndiProviderURL	[svc_ejb only] JNDI provider URL
initialContextFactory	[svc_ejb only] Initial context factory class name
jndiName	[svc_ejb only] EJB JNDI name
clientJAR	[svc_ejb only] Client JAR file path
ejbRemoteInterface	[svc_ejb only] EJB remote interface fully qualified name
ejbHomeInterface	[svc_ejb only] EJB home interface fully qualified name

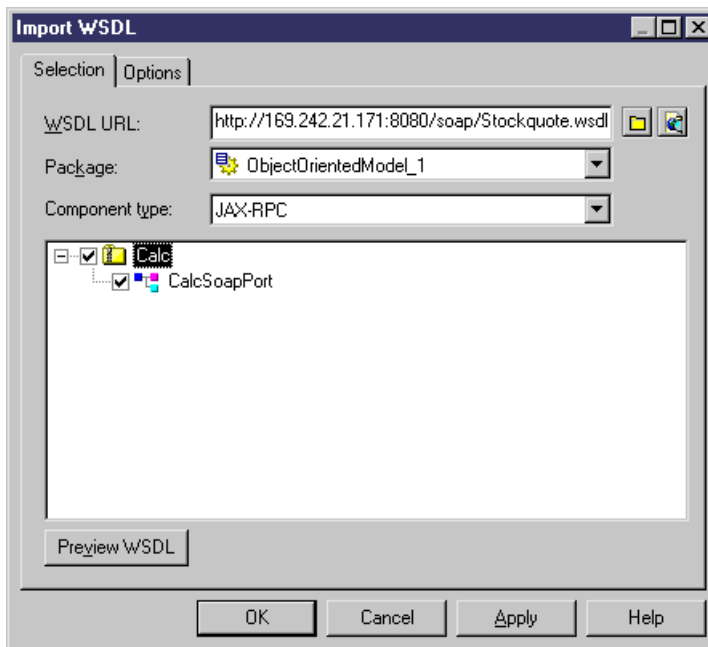
Importing WSDL Files

PowerDesigner can import WSDL files for .NET and Java.

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"> • Entering the location directly in the field • Clicking the Browse File tool to browse on your local file system • Clicking the Browse UDDI tool to search on a UDDI server (see <i>Browsing WSDL Files from UDDI</i> on page 265)
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 > The PowerDesigner Interface > Comparing and Merging Models*.

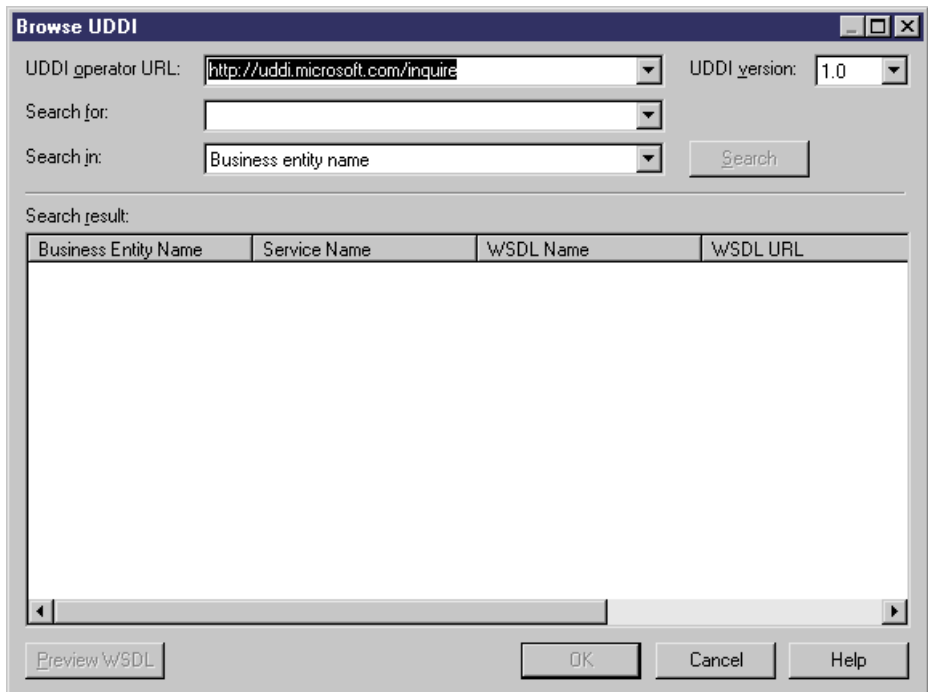
Each Web service selected will be imported as a component and an implementation class. Each port type selected in a selected Web service generates an interface.

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.

Browsing WSDL Files from UDDI

PowerDesigner provides an interface for browsing for a WSDL directly on a UDDI server.

1. Click the **Browse UDDI** tool to the right of the WSDL URL field on the Selection tab of the Import WSDL dialog.



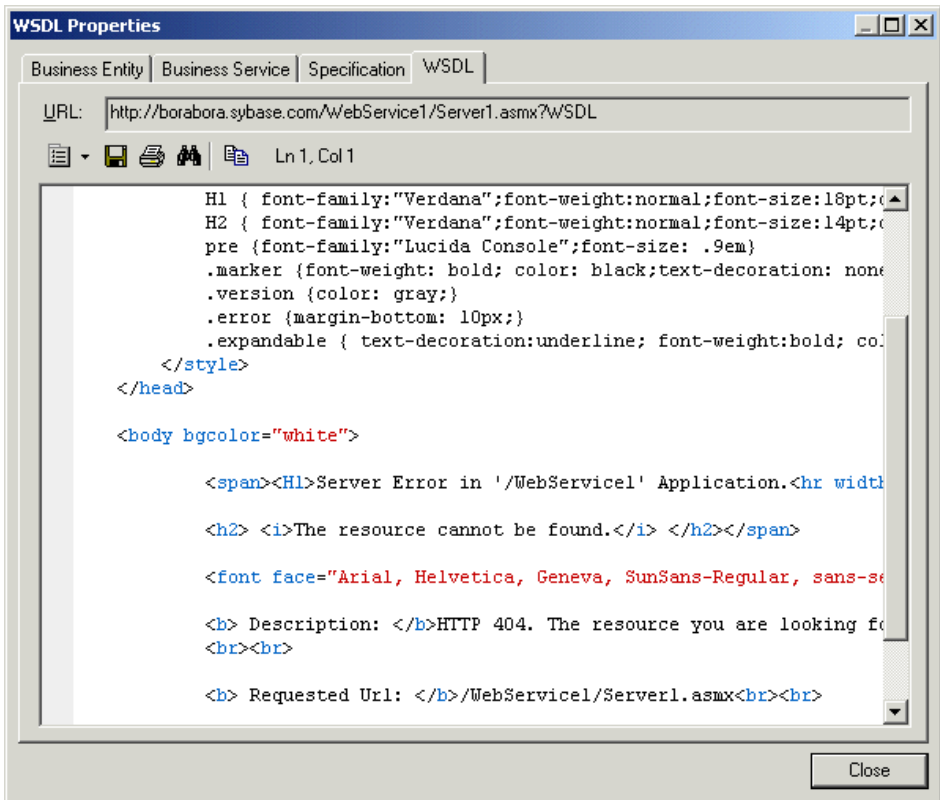
2. Complete the following fields to specify your search criteria:

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.

PART II

Working with OOMs

The chapters in this part provide information about PowerDesigner features that allow you to check, generate from, and reverse-engineer to your object-oriented models .

Generating and Reverse Engineering OO Source Files

PowerDesigner can generate and reverse engineer source files from and to an OOM.

Generating OO Source Files from an OOM

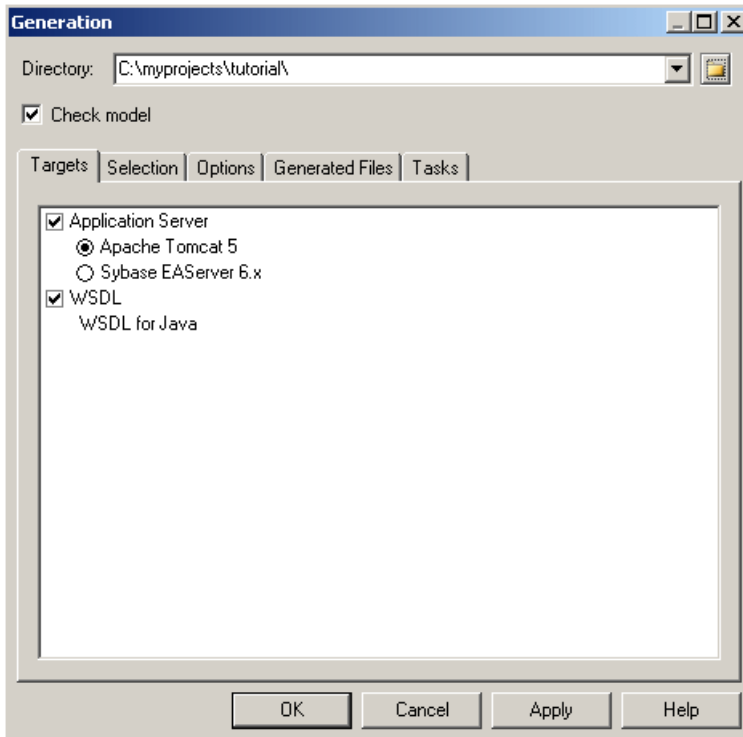
PowerDesigner provides a standard interface for generating source files for all the supported OO languages. For details of language-specific options and generation tasks, see the appropriate language chapter.

By default, PowerDesigner supports the generation of the following types of objects for the languages supported by the OOM:

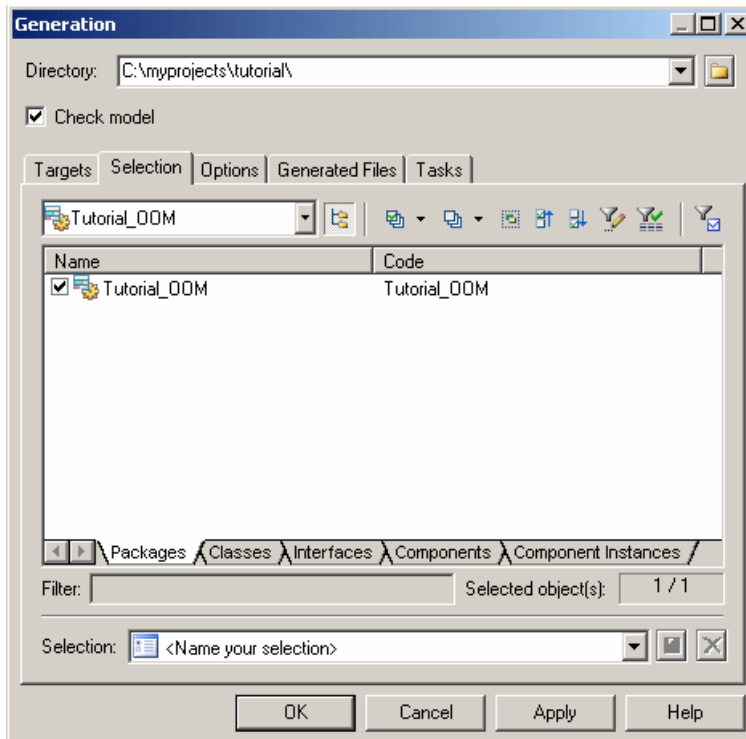
Object language	What is generated
Analysis	No files generated as this language is mainly used for modeling purpose
C#	.CS definition files
C++	C++ definition files (.h and .cpp)
IDL-CORBA	IDL-CORBA definition files
Java	Java files from classes and interfaces of the model. Includes support of EJB and 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

Note: The PowerDesigner generation system is extremely customizable through the use of extensions (see *Extending your Modeling Environment* on page 14). For detailed information about customizing generation, including adding generation targets, options, and tasks, see *Customizing and Extending PowerDesigner > Extension Files*.

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 *Chapter 9, Checking an OOM* on page 295).
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* on page 272).
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.



5. [optional] Click the **Options** tab and set any necessary generation options. For more information about these options, see the appropriate language chapter.

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 > Templates and Generated Files (Profile)*.

7. [optional] Click the **Tasks** tab and specify any additional language-specific generation tasks to perform.
8. Click **OK** to begin generation.

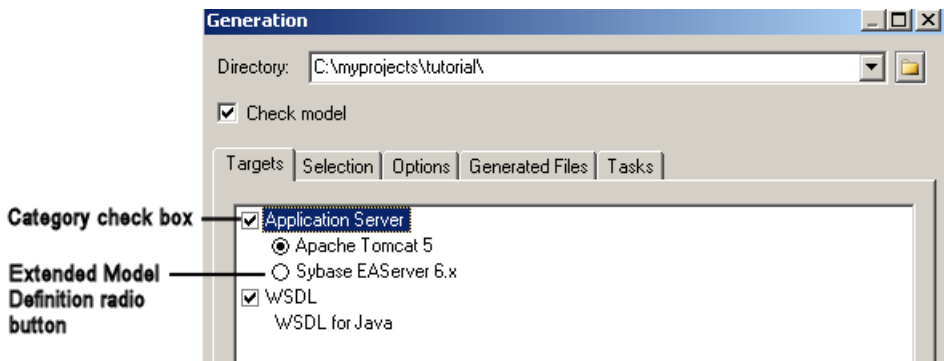
When generation is complete, the Generated Files dialog opens, listing the files that have been generated to the specified directory. Select a file in the list and click **Edit** to open it in your associated editor, or click **Close** to exit the dialog.

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 you own. For information about attaching extensions to your model, see *Extending your Modeling Environment* on page 14.

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*.

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.

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:

```
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.

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 *Composite and Inner Classifiers* on page 46) 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.

Reverse Engineering OO Files into a New OOM

You can reverse engineer object language files to create a new OOM.

1. Select **File > Reverse Engineer > Object Language** 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

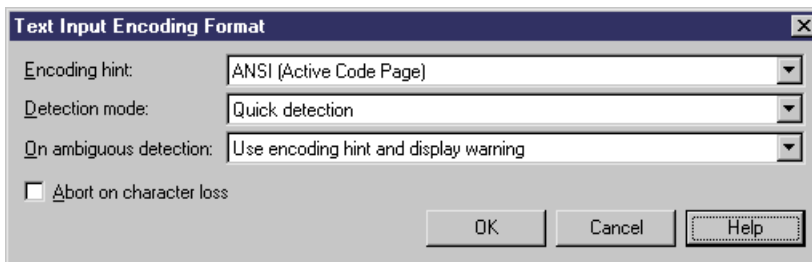
Note: This product includes XML4C 3.0.1 software developed by the Apache Software Foundation (<http://www.apache.org>)

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.

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

Option	Description
Detection mode	<p>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:</p> <ul style="list-style-type: none"> • No detection - Turns off the detection feature. Select this option when you know what the encoding format is • Quick detection - Analyzes a small buffer to perform detection. Select this option when you think that the encoding format will be easy to detect • 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
On ambiguous detection	<p>Specifies what action should be taken in case of ambiguity. You can select from the following options:</p> <ul style="list-style-type: none"> • 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 • Use encoding hint - uses the encoding format selected in the Encoding Hint box, if possible. No warning message is displayed • Use detected encoding - Uses the encoding format detected by PowerDesigner
Abort on character loss	<p>Allows you to stop reverse engineering if characters cannot be identified and are to be lost in current encoding</p>

Here is an example on how to read encoding formats from the list:

```

ASCII
OEM
UTF-8 → No Byte-Order-Mark in the header
UTF-8 (with signature)
Unicode
Unicode (with signature) → There must be a Byte-Order-Mark
Unicode big endian          in the header for the file to be valid
Unicode big endian (with signature)
ANSI (Active Code Page)

```

Reverse Engineering into an Existing OOM

You can reverse engineer source files to add objects to an existing OOM.

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.

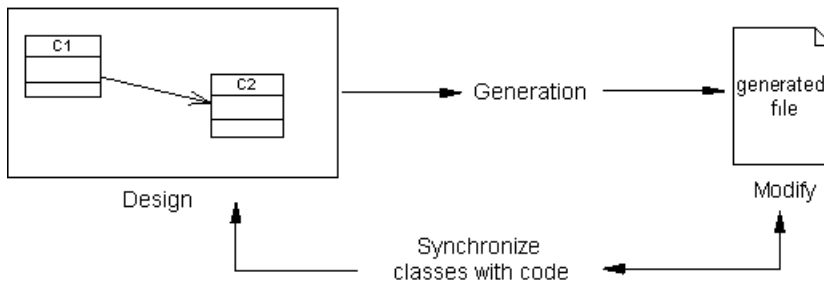
6. Review the objects that you will be importing, and the changes that they will make to the model.

For more information on merging models, see *Core Features Guide > The PowerDesigner Interface > Comparing and Merging Models*.

7. Click OK to merge the selected changes into your model.

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.

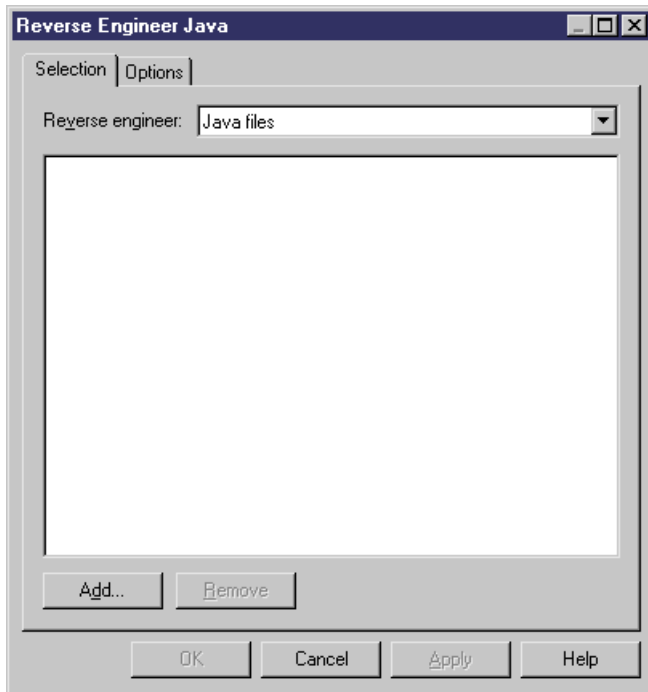


The synchronization launches a reverse engineering dialog box, pre-selects option, and fills the list of classifiers with the classifiers selected in the class diagram.

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.

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.

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.

Generating Other Models from an OOM

You can generate the following types of models from an OOM:

- CDM - 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.
- PDM – 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.
- OOM – 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.
- XSM – to generate a message format from your class structure.

1. Select Tools, and then one of the following to open the appropriate Model Generation Options Window:

- Generate Conceptual Data Model... Ctrl+Shift+C
- Generate Physical Data Model... Ctrl+Shift+P
- Generate Object-Oriented Model... Ctrl+Shift+O
- Generate XML Model... Ctrl+Shift+M

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.

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*.

Mapping OOM Objects to other Model Objects

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	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	Reference or table	KeyRef constraints
Association class	Entity/relationship notation: entity with two associations. Merise notation: association, and two association links	A table and two associations between the end points of the association class	-
Dependency	-	-	-
Realization	-	-	-

OOM	CDM	PDM	XSM
Generalization	Inheritance	Reference	Complex type derivation (XSD) or attribute migration to child element (DTD)

Generating from Classes

For a class to become a table you need to select the Persistent check box and define a type of generation in the Generation list in the Detail tab of the class property sheet.

The cardinality of a class becomes the number of records of a table.

Generating from Associations

If the association has a many-to-many multiplicity: both roles of the association have the * sign selected in their multiplicity lists, the association is translated into a table in the generated PDM. If it has any other multiplicity, the association becomes a reference.

A role name becomes a migrated foreign key after PDM generation.

Managing Object Persistence During OOM to CDM Generation

Developers tend to use object-oriented programming languages like Java, to develop business objects and components. Eventually, these objects can be represented in a conceptual data model prior to being stored in a database. A problem arises because object codes in object-oriented programming languages are often different than codes used in a CDM.

You can define persistent codes to bypass this impedance-mismatch.

Defining Object Persistence in the OOM

Persistent codes are codes defined for OOM classes and attributes that are used during OOM to CDM generation. They also facilitate round-trip engineering by allowing you to recover object codes from the conceptual data model.

All persistent classes are generated as entities. If you set the class generation mode (in the Persistent group box in the Detail tab of the class property sheet) to "Migrate columns", the Generated check box in the generated entity property sheet is cleared by default.

You can also define persistent data types for class attributes and domains. For data type persistence management, you have to take into account the following parameters:

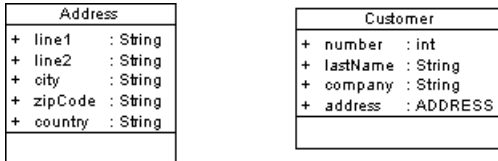
- The data type is *simple*, based on standard OOM data types. If the data type is persistent, generation keeps persistent data type in the target model. Otherwise data type is converted to standard data type in the target model

- The data type is *complex*, based on a classifier. If the data type is persistent, you can define different generation options according to what you want to have in the target model and to the attribute multiplicity

Managing Complex Data Type Persistence

In the following example, both classes are defined as persistent.

Class Customer contains an attribute address with a complex data type ADDRESS. This data type is based on the code of class Address.



For more information on how to use a class as a data type, see *Attaching a Classifier to a Data Type or a Return Type* on page 47.

When you apply a persistent complex data type to a class attribute or a domain, you can define the following different generation options:

- Standard data type generation
- Embedded data type generation

Generating as Persistent Class

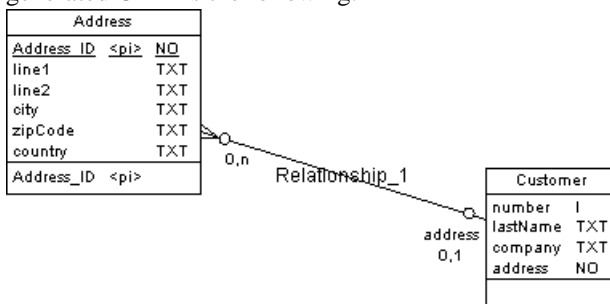
In the Detail page of the property sheet of an attribute using a complex data type, you have to select:

- The Persistent check box
- Persistent in the Class Generation list

After OOM to CDM generation, both classes become entities and a relationship is created between them. The class used as data type contains a new primary identifier attribute.

Example

If you define the standard data type generation options in the Address/Customer example, the generated CDM is the following:



The new column `Address_ID` is the primary identifier in entity `Address`.

Generating as Embedded Class

In the Detail page of the property sheet of an attribute using a complex data type, you have to select:

- The Persistent check box
- Embedded in the Class Generation list

After OOM to CDM generation, all the attributes of the class used as data type are embedded into the entity containing the attribute with complex data type. These attributes are prefixed by the name of the attribute using the complex data type.

Example

If you define the embedded data type generation options in the `Address/``Customer` example, the generated CDM is the following:

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

All the attributes of class `Address` are embedded into entity `Customer` and prefixed by "address" which is the name of the attribute using `Address` as complex data type.

Managing Multiplicity for Complex Persistent Data Types

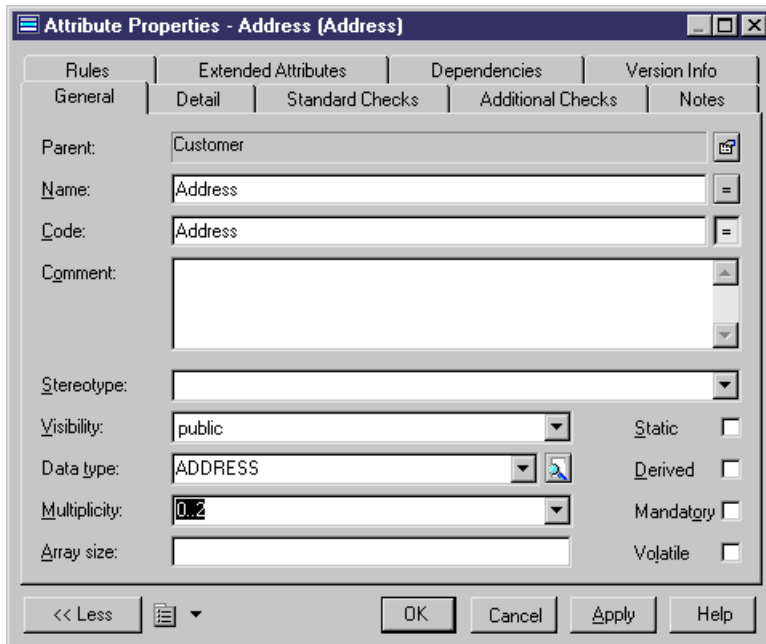
If you define a multiplicity on a class attribute with a complex data type, the generation will reflect this multiplicity.

In the following example, attribute `address` in class `Customer` has a complex data type based on class `Address`.

Address	
+ line1	: String
+ line2	: String
+ city	: String
+ zipCode	: String
+ country	: String

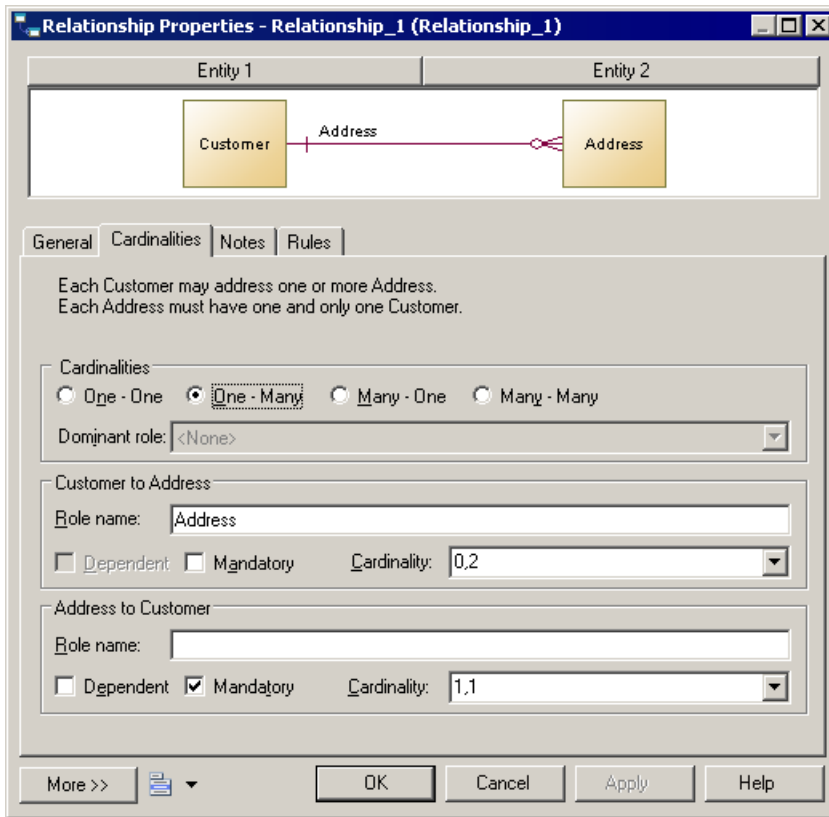
Customer	
+ number	: int
+ lastName	: String
+ company	: String
+ address	: ADDRESS

You also define the multiplicity 0..2 for attribute `address`:



Generating as Persistent Class

When the Persistent check box is selected for the attribute and the class generation is also set to Persistent, then the attribute multiplicity becomes a cardinality on the relationship between entities after OOM to CDM generation.



Generating as Embedded Class

When the Persistent check box is selected for the attribute and the class generation is set to Embedded, all the attributes of the class used as data type are embedded into the entity containing the attribute with complex data type. Attributes are generated as many times as required by multiplicity, provided multiplicity indicates a fixed value.

In the following example, attribute multiplicity is 0..2, so attributes will be embedded twice:

Address	
line1	TXT
line2	TXT
city	TXT
zipCode	TXT
country	TXT

Customer	
number	I
lastName	TXT
company	TXT
address1_line1	TXT
address1_line2	TXT
address1_city	TXT
address1_zipCode	TXT
address1_country	TXT
address2_line1	<UNDEF>
address2_line2	<UNDEF>
address2_city	<UNDEF>
address2_zipCode	<UNDEF>
address2_country	<UNDEF>

Embedded generation of complex data types is not possible when multiplicity is undefined (0..n).

Managing Object Persistence During OOM to PDM Generation

Developers tend to use object-oriented programming languages like Java, to develop business objects and components. Eventually, these objects need to be stored in a database, then a problem arises when the user tries to store objects in a relational database because object codes in object-oriented programming languages are often different than codes used in a relational database.

You have to modify object codes after generating an OOM into a PDM to be compliant with the DBMS.

You can define persistent codes to bypass this impedance-mismatch.

You can also create object to relational mappings between OOM and PDM objects to further define the links between objects from different models (see *Core Features Guide > Linking and Synchronizing Models > Object Mappings*).

Defining Object Persistence in the OOM

Persistent codes are defined for OOM classes and attributes. They are used during OOM to PDM generation in order to fine tune data storage in a relational database. They also facilitate round-trip engineering by allowing to recover object codes from the database.

You can define the type of persistence you want to implement in the Persistent group box in the Detail tab of a class property sheet:

- Generate table means the class is generated in a table.
- Migrate columns means the class is not generated, its attributes and associations are migrated to the generated parent or child table.

- Generate ADT means the class is generated as an abstract data type, a user-defined data type that can encapsulate a range of data values and functions.

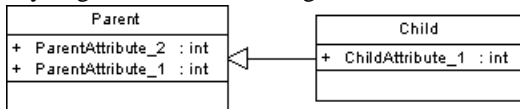
Attribute and associations of persistent classes with a persistence generation mode set to "Migrate columns" are migrated to parent or child tables.

You can also define persistent data types for class attributes and domains. For data type persistence management, you have to take into account the following parameters:

- The data type is *simple*, based on standard OOM data types. If the data type is persistent, generation keeps persistent data type in the target model. Otherwise data type is converted to standard data type in the target model
- The data type is *complex*, based on a classifier. If the data type is persistent, you can define different generation options according to what you want to have in the target model and to the attribute multiplicity

Attribute Migration

If you generate the following OOM into a PDM:

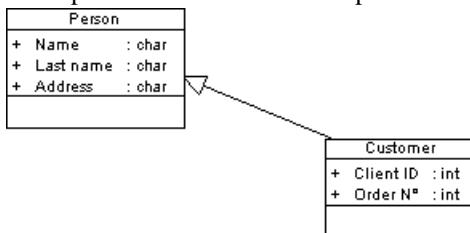


The migration of attributes in the generated PDM depends on the persistence option of the classes in the source OOM:

Persistence	Result
Parent and child / Generate table	Two tables are created for each class
Parent / Migrate columns Child / Generate table	Table Child is generated with parent attributes
Parent / Generate table Child / Migrate columns	Table Parent is generated, with child attributes
Parent and child / Migrate columns	Nothing is generated

Inherited Attributes

In the following OOM, class Customer inherits from Person via a generalization link. Person is not persistent but Customer is persistent.



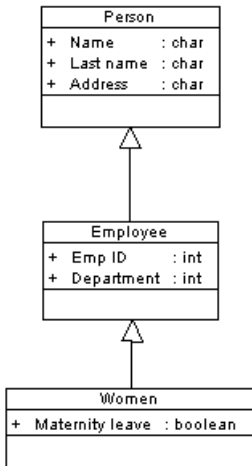
Customer inherits the attributes of the parent class in the generated PDM:

Customer	
Client ID	integer
Order N°	integer
Name	char
Last name	char
Address	char

Derived Class

A derived class is created for conceptual reasons to improve the readability of a model but it adds no semantic information. There is no point of generating a derived class in a PDM, and attributes of this class should be migrated to parent.

In the following example, class Women is created to further analyze class Employee and highlight a specific attribute "Maternity leave". This class is derived, thus not generated but persistent:



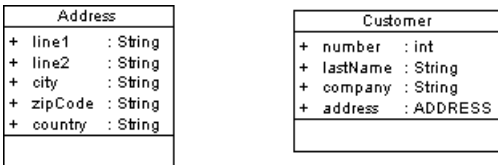
In the generated PDM, Employee inherits from parent class and also from derived class:

Employee	
Emp ID	integer
Department	integer
Name	char
Last name	char
Address	char
Maternity leave	smallint

Managing Complex Data Type Persistence

In the following example, both classes are defined as persistent.

Class Customer contains an attribute address with a complex data type ADDRESS. This data type is based on the code of class Address.



For more information on how to use a class as a data type, see *Attaching a Classifier to a Data Type or a Return Type* on page 47.

When you apply a persistent complex data type to a class attribute or a domain, you can define the following different generation options:

- Standard data type generation
- Embedded data type generation
- Abstract data type generation

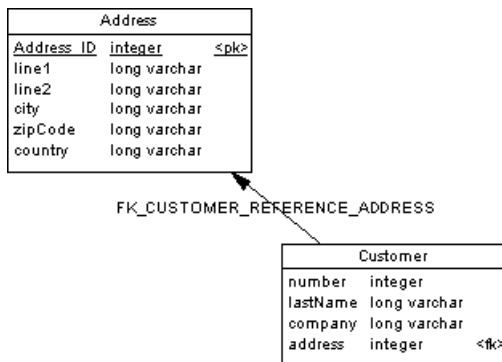
Generating as Persistent Class

In the Detail page of the property sheet of an attribute using a complex data type, you have to select:

- The Persistent check box
- Persistent in the Class Generation list

After OOM to PDM generation, both classes become tables and a reference is created between them. This reference contains a join linking the primary key column in the parent table Address (this column is new, it was created during generation) and the foreign key column in the child table Customer. The foreign key column stands for the attribute using a complex data type in the OOM.

For example, if you define the standard data type generation options in the Address/Customer example, the generated PDM is the following:



The reference contains a join between Address_ID (new column in table Address) and address in child table Customer. Address_ID contains the primary key migrated to column address.

Generating as Embedded Class

In the Detail page of the property sheet of an attribute using a complex data type, you have to select:

- The Persistent check box
- Embedded in the Class Generation list

After OOM to PDM generation, all the attributes of the class used as data type are embedded into the table containing the attribute with complex data type. These attributes are prefixed by the name of the attribute using the complex data type.

For example, if you define the embedded data type generation options in the Address/ Customer example, the generated PDM is the following:

Address	
line1	long varchar
line2	long varchar
city	long varchar
zipCode	long varchar
country	long varchar

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

All the attributes of class Address are embedded into table Customer and prefixed by "address" which is the name of the attribute using Address as complex data type.

Generating as Abstract Data Type Class

In the Detail page of the property sheet of an attribute using a complex data type, you have to select:

- The Persistent check box
- Embedded in the Class Generation list

You also have to select the Abstract Data Type check box in the Detail page of the property sheet of the class used as data type.

After OOM to PDM generation, the class used as data type becomes an abstract data type.

For example, if you define the abstract data type generation options in the Address/Customer example, the generated PDM is the following:

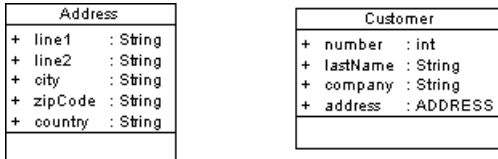
Customer	
number	INTEGER
lastName	LONG
company	LONG
address	Address

Class Address is generated as an abstract data type attached to column address in table Customer.

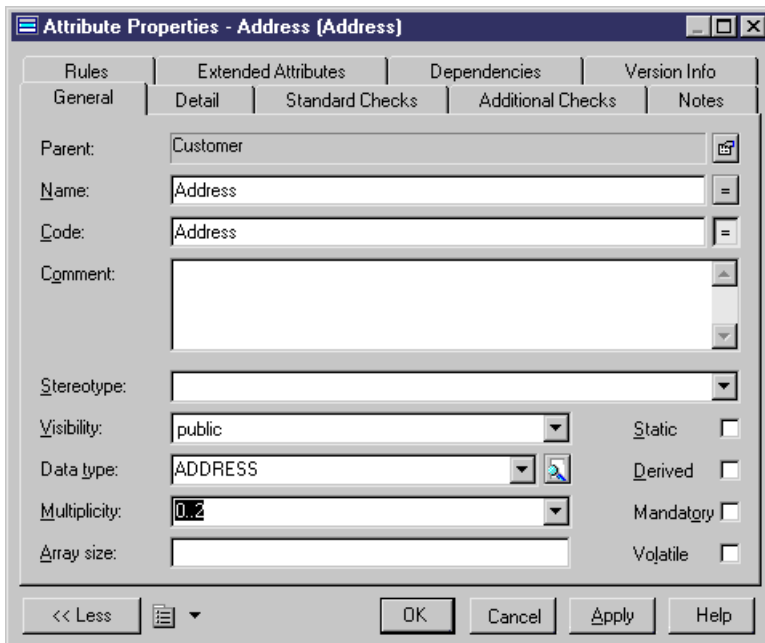
Managing Multiplicity for Complex Persistent Data Types

If you define a multiplicity on a class attribute with a complex data type, the generation will reflect this multiplicity.

In the following example, attribute address in class Customer has a complex data type based on class Address.

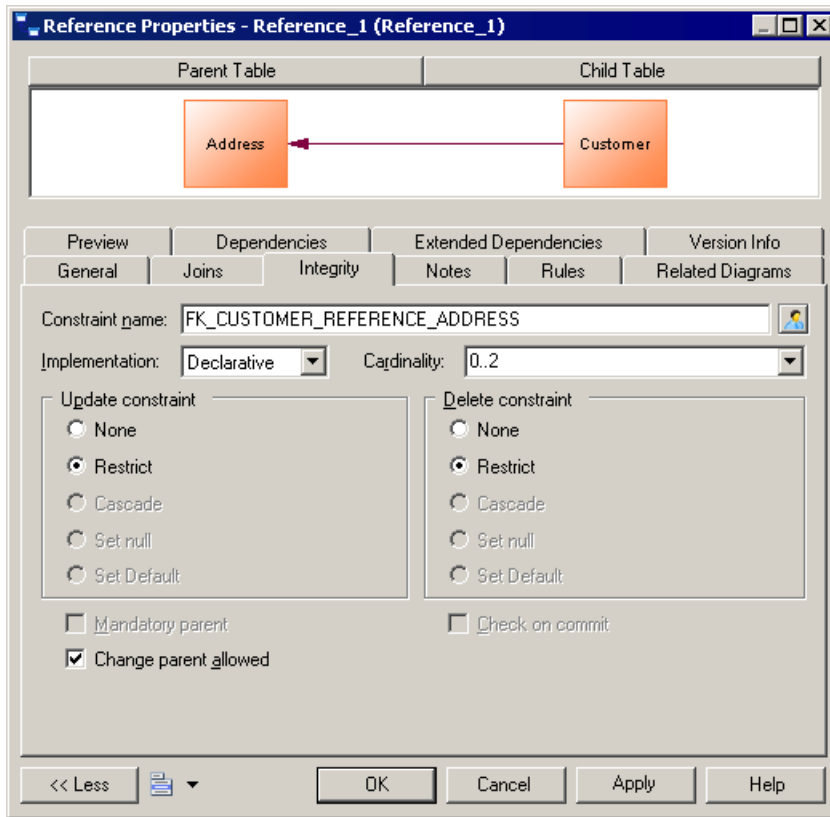


You also define the multiplicity 0..2 for attribute address:



Generating as Persistent Class

When the Persistent check box is selected for the attribute and the class generation is also set to Persistent, then the attribute multiplicity becomes a cardinality on the reference between tables after OOM to PDM generation.



Generating as Embedded Class

When the Persistent check box is selected for the attribute and the class generation is set to Embedded, all the attributes of the class used as data type are embedded into the table containing the column created for the attribute with complex data type. Attributes are generated as many times as required by multiplicity, provided multiplicity indicates 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

Embedded generation of complex data types is not possible when multiplicity is undefined (0..n).

Generating as Abstract Data Type Class

When both attribute and class generation are set to Persistent, and when the class used as data type is generated as an abstract data type, the class is generated as an abstract data type without taking into account the multiplicity of the generated attribute.

However, for those DBMS that support ARRAY and LIST for abstract data types, the generation of a class into an abstract data type can reflect the attribute multiplicity in the following way:

Multiplicity	Generated as
0..n or 1..n	Abstract data type with the LIST type (example: TABLE for Oracle)
1..2 or 0..10	Abstract data type with the ARRAY type (example: VARRAY for Oracle)
0..1 or 1..1	Abstract data type with the OBJECT type

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 > The PowerDesigner Interface > Objects > Checking Models*.

Domain Checks

PowerDesigner provides default model checks to verify the validity of domains.

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"> • Manual correction - Modify the name or code to contain only glossary terms. • Automatic correction - None.

Check	Description and Correction
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
Inconsistency between default values and 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"> Manual correction: Modify default, minimum, maximum or list of values in the check parameters tab Automatic correction: None

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	<p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.

Check	Description and Correction
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"> Manual correction: Add a physical data model from the Models tab of the property sheet of the data source Automatic correction: Deletes data source without physical data model
Data source contain 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"> Manual correction: Delete models with different DBMS or modify the DBMS of models in the data source Automatic correction: None

Package Checks

PowerDesigner provides default model checks to verify the validity of packages.

Check	Description and Correction
Circular inheritance	<p>Objects cannot be dependent on each other. Circular links must be removed.</p> <ul style="list-style-type: none"> Manual correction: Remove circular generalization links Automatic correction: None
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"> Manual correction: Remove circular links Automatic correction: None
Shortcut code uniqueness	<p>Two shortcuts with the same code cannot be in the same namespace.</p> <ul style="list-style-type: none"> Manual correction: Change the code of one of the shortcuts Automatic correction: None
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"> Manual correction: Modify the design of your model in order to create the association in the package where the child class is defined Automatic correction: None

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	[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary. <ul style="list-style-type: none">• Manual correction - Modify the name or code to contain only glossary terms.• Automatic correction - None.
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">• Manual correction - Modify the name or code to contain only glossary terms.• Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	Object names must be unique in the namespace. <ul style="list-style-type: none">• Manual correction - Modify the duplicate name or code.• Automatic correction - Appends a number to the duplicate name or code.
Actor/use case not linked to any object	Actors and use cases should be linked to at least one object in the model. <ul style="list-style-type: none">• Manual correction: Create a link between the actor and a use case, or an object• Automatic correction: None

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">• Manual correction - Modify the name or code to contain only glossary terms.• Automatic correction - None.

Check	Description and Correction
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"> • Manual correction - Modify the name or code to contain only glossary terms. • Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> • Manual correction - Modify the duplicate name or code. • Automatic correction - Appends a number to the duplicate name or code.
Empty classifier	<p>Attributes and operations are missing for this classifier.</p> <ul style="list-style-type: none"> • Manual correction: Add attributes or operations to the classifier • Automatic correction: None
Persistent class without persistent attributes	<p>All attributes of a persistent class cannot be non-persistent.</p> <ul style="list-style-type: none"> • Manual correction: Define at least one attribute as persistent • Automatic correction: None
Association class with identifier(s)	<p>An associated class should not have identifiers.</p> <ul style="list-style-type: none"> • Manual correction: Remove the identifier • Automatic correction: None
Classifier visibility	<p>A private or protected classifier should be inner to another classifier.</p> <ul style="list-style-type: none"> • Manual correction: Change classifier visibility to public or package • Automatic correction: Changes the visibility to public or package
Class constructor return type	<p>A constructor cannot have a return type.</p> <ul style="list-style-type: none"> • Manual correction: Remove current return type of the constructor • Automatic correction: Removes current return type of the constructor
Class constructor modifiers	<p>A constructor cannot be static, abstract, or final.</p> <ul style="list-style-type: none"> • Manual correction: Remove the static, abstract, or final property of the constructor • Automatic correction: Removes the static, abstract or final property of the constructor

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 Operations tab in the class property sheet and select the To be Implemented button at the bottom of the tab to implement the missing operations.</p> <ul style="list-style-type: none"> • Manual correction: Implement the operations of the interface within the class • Automatic correction: None
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"> • Manual correction: Assign a role name for the association role • Automatic correction: None
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"> • Manual correction: Change the name of the duplicate role • Automatic correction: None
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"> • Manual correction: Create a BeanInfo class • Automatic correction: None
BeanInfo without a JavaBean class	<p>A BeanInfo class must depend on a JavaBean class.</p> <ul style="list-style-type: none"> • Manual correction: Create a JavaBean class and recreate its BeanInfo, or delete the BeanInfo • Automatic correction: None
Emuneration type parent	<p>A enum may not have children.</p> <ul style="list-style-type: none"> • Manual correction: Remove links to child classes. • Automatic correction: None
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"> • Manual correction: Change the class visibility to public, define a public constructor with no arguments, do not define the finalize() method • 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

Check	Description and Correction
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"> • Manual correction: Add a method with the same name, number, return type and types of arguments in the Bean class • Automatic correction: Adds a method with the appropriate values in the Bean class
Bean class Home interface methods implementation	<p>For each create<METHOD> method of the bean Home Interface(s), there must be a matching ejbCreate<METHOD> 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<METHOD> 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 <i>Entity Beans only</i>.</p> <p>For each ejbCreate<METHOD> method of the Bean class, there must be a matching ejbPostCreate<METHOD> method in the Bean class with the same number and types of arguments.</p> <ul style="list-style-type: none"> • Manual correction: Add a method with the same name and types of arguments in the Bean class • Automatic correction: Adds a method with the appropriate values in the Bean class <p>The following check applies to <i>BMP Entity Beans only</i>.</p> <p>For each find<METHOD> finder method defined in the bean Home Interface(s), there must be a corresponding ejbFind<METHOD> method with the same number, return type, and types of arguments.</p> <ul style="list-style-type: none"> • Manual correction: Add a method with the same number, return type and types of arguments in the bean Home Interface(s) • Automatic correction: Adds a method with the appropriate values in the bean Home Interface(s)

Check	Description and Correction
Bean class ejb-Create methods	<p>ejbCreate<METHOD> methods must be defined as public, and cannot be final nor static.</p> <p>The following check applies to <i>Entity Beans only</i>.</p> <p>The return type of an ejbCreate() method must be the primary key type.</p> <ul style="list-style-type: none"> • Manual correction: Select the primary key in the Return type list of the Operation property sheet • Automatic correction: Selects the primary key as return type <p>The following check applies to <i>Session Beans and Message Driven Beans, and Message Driven Beans</i>.</p> <p>The return type of an ejbCreate() method must be void.</p> <ul style="list-style-type: none"> • Manual correction: Select void in the Return type list of the Operation property sheet • Automatic correction: Changes the return type to void <p>The following check applies to <i>Message Driven Beans only</i>.</p> <p>The Bean class must define an ejbCreate() method that takes no arguments.</p> <ul style="list-style-type: none"> • Manual correction: Add a method with no argument in the Bean class • Automatic correction: Adds a method with no argument in the Bean class
Bean class ejb-PostCreate methods	<p>The following check applies to <i>Entity Beans only</i>.</p> <p>ejbPostCreate<METHOD> 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"> • 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 • Automatic correction: Changes the method visibility to public, changes the final and static check boxes and changes the return type to void
Bean class ejb-Find methods	<p>BMP Entity Bean specific.</p> <p>ejbFind<METHOD> 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"> • Manual correction: Change the method visibility to public and deselect the static check box • Automatic correction: Changes the method visibility to public and deselects the static and final check boxes. Forces the return type of ejbFind<METHOD> to the primary key type

Check	Description and Correction
Bean class ejb-Home methods	<p>ejbHome<METHOD> methods must be defined as public and cannot be static.</p> <ul style="list-style-type: none"> Manual correction: Change the method visibility to public and deselect the static check box Automatic correction: Changes the method visibility to public and deselects the static check box
Bean class ejbSelect methods	<p>The following check applies to <i>CMP Entity Beans only</i>.</p> <p>EjbSelect <METHOD> methods must be defined as public and abstract. Their throws clause must include the javax.ejb.FinderException.</p> <ul style="list-style-type: none"> Manual correction: Change the method visibility to public, select the abstract check box, and include the javax.ejb.FinderException Automatic correction: Changes the method visibility to public, selects the abstract check box, and includes the javax.ejb.FinderException
Primary key class definition	<p>The following check applies to <i>Entity Beans only</i>.</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"> Manual correction: Change the method visibility to public and add a default constructor in the primary key class Automatic correction: Changes the method visibility to public and adds a default constructor in the primary key class
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"> 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 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
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"> 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 Automatic correction: Creates a primary key class, or selects the appropriate object/classifier data type

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"> • Manual correction: Define the mapping from the Mapping tab of the class property sheet. (Class Sources tab), or remove the data source • Automatic correction: Removes the data source from the Mapping For list in the class Mapping tab <p>For more information about O/R mapping, see <i>Core Features Guide > Linking and Synchronizing Models > Object Mappings > Object to Relational (O/R) Mapping.</i></p>
Attribute mapping not defined	<p>The attribute must be mapped to columns in the data source.</p> <ul style="list-style-type: none"> • Manual correction: Define the mapping from the Mapping tab of the Class property sheet. (Attributes Mapping tab), or remove the data source • Automatic correction: Removes the data source from the Mapping For list in the class Mapping tab <p>For more information about O/R mapping, see <i>Core Features Guide > Linking and Synchronizing Models > Object Mappings > Object to Relational (O/R) Mapping.</i></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"> • Manual correction: Specify a generic classifier in the field to the right of the type list on the General tab of the bound classifier's property sheet. You can also connect it to the generic classifier by way of a dependency with stereotype <<bind>>. • Automatic correction: None
Invalid generation mode	<p>If a class has its persistence mode set to Migrate Columns, it must have a persistent parent or child to which to migrate the columns</p> <ul style="list-style-type: none"> • Manual correction: Link the class to a persistent parent or child, or change its persistence mode on the Detail tab of its property sheet. • Automatic correction: None

Identifier Checks

PowerDesigner provides default model checks to verify the validity of identifiers.

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	Object names must be unique in the namespace. <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
Existence of attribute	Identifiers must have at least one attribute. <ul style="list-style-type: none"> Manual correction: Add an attribute to the identifier, or delete the identifier Automatic correction: None
Identifier inclusion	Two identifiers should not use the same attributes. <ul style="list-style-type: none"> Manual correction: Remove the unnecessary identifier Automatic correction: None

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.

Check	Description and Correction
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
Empty classifier	<p>Attributes and operations are missing for this classifier.</p> <ul style="list-style-type: none"> Manual correction: Add attributes or operations to the classifier Automatic correction: None
Classifier visibility	<p>A private or protected classifier should be inner to another classifier.</p> <ul style="list-style-type: none"> Manual correction: Change classifier visibility to public or package Automatic correction: Changes the visibility to public or package
Interface constructor	<p>An interface cannot be instantiated. Thus a constructor cannot be defined for an interface.</p> <ul style="list-style-type: none"> Manual correction: Remove the constructor Automatic correction: None
Interface navigability	<p>Navigation is not allowed from an interface.</p> <ul style="list-style-type: none"> Manual correction: Deselect navigability on the class side of the association Automatic correction: Deselects navigability on the class side of the association
Home interface create methods	<p>The return type for create<METHOD> 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<METHOD> and ejbPostCreate<METHOD> methods of the Bean class.</p> <ul style="list-style-type: none"> Manual correction: Include the javax.ejb.CreateException and all exceptions defined in the throws clause of the matching ejbCreate<METHOD> and ejbPostCreate<METHOD> methods of the Bean class, or remove exceptions from the ejbPostCreate<METHOD> method Automatic correction: Includes the javax.ejb.CreateException and all exceptions defined in the throws clause of the matching ejbCreate<METHOD> and ejbPostCreate<METHOD> methods of the Bean class

Check	Description and Correction
Home interface finder methods	<p>The return type for find<METHOD> 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"> • Manual correction: Include the javax.ejb.FinderException in the throws clause • Automatic correction: Includes the javax.ejb.FinderException in the throws clause, and sets Return Type to be the component interface type <p>The following check applies to <i>BPM Entity Beans only</i>.</p> <p>The throws clause must include all exceptions defined in the throws clause of the matching ejbFind<METHOD> methods of the Bean class.</p> <ul style="list-style-type: none"> • Manual correction: Include all exceptions defined in the throws clause of the matching ejbFind<METHOD> methods of the Bean class, or remove exceptions from the ejbFind<METHOD> method • Automatic correction: Includes all exceptions defined in the throws clause of the matching ejbFind<METHOD> methods of the Bean class
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"> • Manual correction: Include the java.rmi.RemoteException • Automatic correction: Includes the java.rmi.RemoteException
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"> • Manual correction: Include the java.rmi.RemoteException • Automatic correction: Includes the java.rmi.RemoteException
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"> • Manual correction: Specify a generic classifier in the field to the right of the type list on the General tab of the bound classifier's property sheet. You can also connect it to the generic classifier by way of a dependency with stereotype <<bind>>. • Automatic correction: None

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
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"> Manual correction: Modify default, minimum, maximum or list of values in the check parameters tab Automatic correction: None
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"> Manual correction: Assign a valid data type to the attribute Automatic correction: None
Initial value for final attribute	<p>The final attribute of a classifier must be initialized.</p> <ul style="list-style-type: none"> Manual correction: Give a default value to the final attribute Automatic correction: None

Check	Description and Correction
Domain divergence	<p>The definition of the attribute definition is diverging from the definition of the domain.</p> <ul style="list-style-type: none"> Manual correction: Modify attribute type to respect domain properties Automatic correction: Corrects attribute type to prevent divergence from domain <p>For more information about domain divergence, see <i>Setting OOM Model Options</i> on page 11.</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"> Manual correction: set the data type to an appropriate delegate Automatic correction: None

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
Return type assignment	<p>The return type of an operation should be defined.</p> <ul style="list-style-type: none"> Manual correction: Assign a return type to the operation Automatic correction: Assigns a void return type to the operation

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"> • Manual correction: Choose a valid data type for the parameter • Automatic correction: None
Abstract operation body	<p>[classes] Abstract operations in a class cannot be implemented.</p> <ul style="list-style-type: none"> • Manual correction: Remove the body or the abstract property of the operation • Automatic correction: None
Abstract operation in a instantiable class	<p>[classes] Abstract operations must be declared in abstract classes only.</p> <ul style="list-style-type: none"> • Manual correction: Set the class to abstract, or remove the abstract property of the operation • Automatic correction: Removes the abstract property in the operation property sheet
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"> • Manual correction: Change the operation name, parameter data type, or return type • Automatic correction: None
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 overridden.</p> <ul style="list-style-type: none"> • Manual correction: Keep the same modifiers for child operation • Automatic correction: None
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"> • Manual correction: Make sure each constant is associated with a concrete method that overloads the abstract method. • Automatic correction: None

Realization Checks

PowerDesigner provides default model checks to verify the validity of realizations.

Check	Description and Correction
Redundant realizations	<p>Only one realization is allowed between two given objects.</p> <ul style="list-style-type: none"> Manual correction: Remove redundant realizations Automatic correction: None
Realization generic missing child type parameters	<p>A child of a generic classifier must resolve all of the type parameters defined by its parent.</p> <ul style="list-style-type: none"> Manual correction: Resolve the missing type parameters. Automatic correction: None.
Realization generic child cannot be bound	<p>A bound classifier cannot be the child of any classifier other than its generic parent.</p> <ul style="list-style-type: none"> Manual correction: Remove the additional links. Automatic correction: None.

Generalization Checks

PowerDesigner provides default model checks to verify the validity of generalizations.

Check	Description and Correction
Redundant generalizations	<p>Only one generalization is allowed between two classes or two interfaces.</p> <ul style="list-style-type: none"> Manual correction: Remove redundant generalizations Automatic correction: None
Class multiple inheritance	<p>The following check applies only to <i>Java</i> and <i>PowerBuilder</i>.</p> <p>Multiple inheritance is accepted in UML but not in this language.</p> <ul style="list-style-type: none"> Manual correction: Keep single inheritance Automatic correction: None
Extend final class	<p>A final class cannot be extended.</p> <ul style="list-style-type: none"> Manual correction: Remove the generalization link, or remove the final property in the parent class Automatic correction: None

Check	Description and Correction
Non-persistent specifying attribute	<p>If a generalization has a specifying attribute, the attribute must be marked as persistent.</p> <ul style="list-style-type: none"> Manual correction: Select the Persistent checkbox on the Detail tab of the specifying attribute property sheet. Automatic correction: None
Generic: Child type parameters	<p>A child of a generic classifier must resolve all of the type parameters defined by its parent.</p> <ul style="list-style-type: none"> Manual correction: Resolve the missing type parameters. Automatic correction: None.
Generic: Child cannot be bound	<p>A bound classifier cannot be the child of any classifier other than its generic parent.</p> <ul style="list-style-type: none"> Manual correction: Remove the additional links. Automatic correction: None.

Object Checks

PowerDesigner provides default model checks to verify the validity of objects.

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.

Check	Description and Correction
Isolated object	<p>An object should not be isolated in the model.</p> <ul style="list-style-type: none"> Manual correction: Create a relationship to or from the object. The relationship can be a message, an instance link, or a dependency <i>or</i> Link the object to an object node in the activity diagram Automatic correction: None <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>

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"> Manual correction: Remove one of the redundant instance links Automatic correction: None

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"> Manual correction: Enter a sequence number on the message Automatic correction: None
Message used by several instance links	<p>A message should not be attached to several instance links.</p> <ul style="list-style-type: none"> Manual correction: Detach the message from the instance link Automatic correction: None

Check	Description and Correction
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"> Manual correction: Create a message between two objects or between an actor and an object Automatic correction: None

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
Input transition missing	<p>Each state must have at least one input transition. A state without an input transition cannot be reached.</p> <ul style="list-style-type: none"> Manual correction: Add a transition linked to the state Automatic correction: None
Composite state does not have any start	<p>A composite state details the state behavior in a sub-statechart diagram. To be complete, this sub-statechart diagram requires a start.</p> <ul style="list-style-type: none"> Manual correction: Add a start in the sub-statechart diagram, or deselect the Composite check box in the state property sheet Automatic correction: None

Check	Description and Correction
Incorrect action order	<p>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.</p> <ul style="list-style-type: none"> • Manual correction: Move all entry at the top of the list and all exit at the bottom • Automatic correction: Moves all entry at the top of the list and all exit at the bottom

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	<p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> • Manual correction - Modify the name or code to contain only glossary terms. • Automatic correction - None.
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"> • Manual correction - Modify the name or code to contain only glossary terms. • Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> • Manual correction - Modify the duplicate name or code. • Automatic correction - Appends a number to the duplicate name or code.
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"> • Manual correction: Specify a trigger event in the action property sheet • Automatic correction: None

Check	Description and Correction
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"> • Manual correction: Change the trigger event or the condition of the action • Automatic correction: None

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"> • Manual correction - Modify the name or code to contain only glossary terms. • Automatic correction - None.
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"> • Manual correction - Modify the name or code to contain only glossary terms. • Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> • Manual correction - Modify the duplicate name or code. • Automatic correction - Appends a number to the duplicate name or code.
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"> • Manual correction: Delete the event or use it within an action on a state or on a transition • Automatic correction: Deletes the event

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	Object names must be unique in the namespace. <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
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"> Manual correction: Add any missing transitions on the junction point Automatic correction: None

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.

Check	Description and Correction
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
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"> Manual correction: Add a transition linked to the activity Automatic correction: None
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"> Manual correction: Add a start in the sub-activity diagram, or deselect the Composite check box in the activity property sheet Automatic correction: None
Non-Reusable Activity Calls	<p>Only activities with an action type of <undefined> 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"> Manual correction: Change the action type of the referenced activity, or remove any references to it. Automatic correction: None

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.

Check	Description and Correction
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
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"> Manual correction: Add any missing transitions on the decision Automatic correction: None

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.

Check	Description and Correction
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"> Manual correction: In the property sheet of the object node, select or create an object from the Object list Automatic correction: None
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"> Manual correction: Select a Data type in the object node property sheet. Automatic correction: None

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	<p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
Circular dependency	<p>The organization unit cannot be parent of itself or parent of another child organization unit.</p> <ul style="list-style-type: none"> Manual correction: Change the organization unit in the Parent box in the organization unit property sheet Automatic correction: None

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	Object names must be unique in the namespace. <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
Missing transition	Starts and ends must be linked to an object of the statechart or activity diagram. <ul style="list-style-type: none"> Manual correction: Create a transition from the start and/or to the end Automatic correction: None

Synchronization Checks

PowerDesigner provides default model checks to verify the validity of synchronizations.

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.

Check	Description and Correction
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
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"> Manual correction: Add any missing transitions to the synchronization Automatic correction: None

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	<p>The transition or flow has no source or destination.</p> <ul style="list-style-type: none"> Manual correction: Select or create an object as source or destination. Automatic correction: None
Useless condition	<p>If there is only one output transition/flow, there is no reason to have a condition or type on the transition/flow.</p> <ul style="list-style-type: none"> Manual correction: Remove the unnecessary condition or type, or create another transition/flow with another condition or type. Automatic correction: None

Check	Description and Correction
Missing condition	<p>If an object has several output transitions/flows, or if the transition/flow is reflexive, each transition/flow must contain a condition.</p> <p>In a statechart diagram, a transition must contain an event or condition.</p> <ul style="list-style-type: none"> Manual correction: Define a condition, or create a synchronization to specify a parallel execution Automatic correction: None
Duplicated transition between states/ Duplicated flow between activities	<p>Two parallel transitions (with the same extremities) must not occur simultaneously, but must rather be governed by conditions (and, for transitions, trigger events).</p> <ul style="list-style-type: none"> Manual correction: Change one of the trigger events or conditions Automatic correction: None

Component Checks

PowerDesigner provides default model checks to verify the validity of components.

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.

Check	Description and Correction
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"> • Manual correction: Attach some classes or interfaces to the component • Automatic correction: None
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"> • Manual correction: Complete existing view(s), or create a remote view if no interface has been exposed • Automatic correction: Completes existing view(s), or creates a remote view if no interface has been exposed
SOAP message re-definition	<p>You cannot have the same SOAP input and SOAP output data type inside the same component.</p> <ul style="list-style-type: none"> • 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 • Automatic correction: None <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>

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"> • Manual correction - Modify the name or code to contain only glossary terms. • Automatic correction - None.
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"> • Manual correction - Modify the name or code to contain only glossary terms. • Automatic correction - Replaces synonyms with their associated glossary terms.

Check	Description and Correction
Name/Code uniqueness	Object names must be unique in the namespace. <ul style="list-style-type: none"> • Manual correction - Modify the duplicate name or code. • Automatic correction - Appends a number to the duplicate name or code.
Empty node	A node is said to be empty when it does not contain any component instance. <ul style="list-style-type: none"> • Manual correction: Add at least one component instance to the node • Automatic correction: None

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 Expression field. <ul style="list-style-type: none"> • Manual correction: Specify an expression for the data format. • Automatic correction: None

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"> • Manual correction - Modify the name or code to contain only glossary terms. • Automatic correction - None.
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"> • Manual correction - Modify the name or code to contain only glossary terms. • Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	Object names must be unique in the namespace. <ul style="list-style-type: none"> • Manual correction - Modify the duplicate name or code. • Automatic correction - Appends a number to the duplicate name or code.

Check	Description and Correction
Component instance without component	<p>An instance of a component has been created but no component is attached. You must attach it to a component.</p> <ul style="list-style-type: none"> 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 Automatic correction: None
Duplicate component instances	<p>Several instances of the same component exist in the same node.</p> <ul style="list-style-type: none"> Manual correction: Delete the duplicate component instance or attach it to the right component Automatic correction: None
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"> Manual correction: Attach it to a node Automatic correction: None

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"> Manual correction: Open the interaction reference property sheet and specify the sequence diagram to reference. Automatic correction: None

Check	Description and Correction
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"> • 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. • Automatic correction: None
Too many input messages for reference	<p>The interaction reference has more incoming than outgoing messages.</p> <ul style="list-style-type: none"> • Manual correction: Delete incoming messages or add outgoing messages, until the numbers are equal. • Automatic correction: None
Too many output messages for reference	<p>The interaction reference has more outgoing than incoming messages.</p> <ul style="list-style-type: none"> • Manual correction: Delete outgoing messages or add incoming messages, until the numbers are equal. • Automatic correction: None

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"> • Manual correction - Modify the name or code to contain only glossary terms. • Automatic correction - None.

Check	Description and Correction
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
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"> Manual correction: Specify a data type for the part and connect the relevant classifier to its owner classifier. Automatic correction: None
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"> Manual correction: Enable or disable the Composition property. Automatic correction: The Composition property is enabled or disabled.

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	<p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.

Check	Description and Correction
Name/Code uniqueness	Object names must be unique in the namespace. <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
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"> Manual correction: Specify a data type or interface Automatic correction: None

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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - None.
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"> Manual correction - Modify the name or code to contain only glossary terms. Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	Object names must be unique in the namespace. <ul style="list-style-type: none"> Manual correction - Modify the duplicate name or code. Automatic correction - Appends a number to the duplicate name or code.
Component assembly connector null interface connector	An interface must be defined for each assembly connector. <ul style="list-style-type: none"> Manual correction: Define an interface for the assembly connector. Automatic correction: None.
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"> Manual correction: Ensure that the supplier and client are correctly defined. Automatic correction: None.

Association Checks

PowerDesigner provides default model checks to verify the validity of associations.

Check	Description and Correction
Generic: Child type parameters	<p>In a navigable association, if the parent is generic, the child must redefine all the type parameters of the parent.</p> <p>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.</p> <ul style="list-style-type: none"> • Manual correction: Resolve the missing type parameters. • Automatic correction: None.
Generic: Child cannot be bound	<p>A bound classifier cannot be the child of any navigable association other than its generic parent.</p> <ul style="list-style-type: none"> • Manual correction: Remove the additional links. • Automatic correction: None.

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	<p>[if glossary enabled] Names and codes must contain only approved terms drawn from the glossary.</p> <ul style="list-style-type: none"> • Manual correction - Modify the name or code to contain only glossary terms. • Automatic correction - None.
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"> • Manual correction - Modify the name or code to contain only glossary terms. • Automatic correction - Replaces synonyms with their associated glossary terms.
Name/Code uniqueness	<p>Object names must be unique in the namespace.</p> <ul style="list-style-type: none"> • Manual correction - Modify the duplicate name or code. • Automatic correction - Appends a number to the duplicate name or code.

CHAPTER 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.

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

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
	Note: The Global property is not imported.
Diagram	Diagram

Rose Object	OOM Object
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

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"> Diagrams 	Use case: <ul style="list-style-type: none"> Related diagrams
Association: <ul style="list-style-type: none"> Navigable 	Association: <ul style="list-style-type: none"> Orientation

Note: Dependencies between a use case and an actor are not imported.

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"> • 'Class utility' Type • Export Control • Implementation • Cardinality: 0..n, 1..n • Nested class • Persistence • Abstract 	Class: <ul style="list-style-type: none"> • 'Class' Type • Visibility • Package • Cardinality: 0..*, 1..* • Inner classifier • Persistence • Abstract
Interface: <ul style="list-style-type: none"> • Export Control • Implementation • Nested class 	Interface: <ul style="list-style-type: none"> • Visibility • Package • Inner classifier
Attribute: <ul style="list-style-type: none"> • Export Control • Implementation • Initial value • Static • Derived 	Attribute: <ul style="list-style-type: none"> • Visibility • Package • Initial value • Static • Derived
Operation: <ul style="list-style-type: none"> • Export Control • Implementation 	Operation: <ul style="list-style-type: none"> • Visibility • Package
Generalization: <ul style="list-style-type: none"> • Export Control • Implementation • Virtual inheritance • Multi inheritance 	Generalization: <ul style="list-style-type: none"> • Visibility • Package • Extended attribute • Multi inheritance

Rose Object	OOM Object
Association: <ul style="list-style-type: none"> • Role name • Export Control • Navigable • Cardinality • Aggregate (class A or B) • Aggregate (by reference) • Aggregate (by value) 	Association: <ul style="list-style-type: none"> • Role name • Visibility • Navigable • Multiplicity • Container • Aggregation • Composition
Dependency	Dependency

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 or class instance: <ul style="list-style-type: none"> • Class • Multiple instances 	Object: <ul style="list-style-type: none"> • Class or interface • Multiple
Link or object link: <ul style="list-style-type: none"> • Assoc • Messages list 	Instance link: <ul style="list-style-type: none"> • Association • Messages
Actor	Actor
Message: <ul style="list-style-type: none"> • Simple stereotype • Synchronous stereotype • Asynchronous stereotype • Balking 	Message: <ul style="list-style-type: none"> • Undefined • Procedure call • Asynchronous • Condition

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 or class instance: <ul style="list-style-type: none"> • Persistence • Multiple instances 	Object: <ul style="list-style-type: none"> • Persistence • Multiple
Actor	Actor
Message: <ul style="list-style-type: none"> • Simple stereotype • Synchronous stereotype • Asynchronous stereotype • Balking • Return message • Destruction marker 	Message: <ul style="list-style-type: none"> • Undefined • Procedure call • Asynchronous • Condition • Return • Recursive message with Destroy action

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
State: <ul style="list-style-type: none"> • When action • OnEntry action • OnExit action • Do action • OnEvent action • Event action • Event arguments 	State or object node: <ul style="list-style-type: none"> • Trigger event • Entry • Exit • Do • Event • Trigger event • Event arguments
State transition: <ul style="list-style-type: none"> • <No name> • <No code> • Event • Arguments • Guard condition • Action 	Transition: <ul style="list-style-type: none"> • Calculated name • Calculated Code • Trigger Event • Event arguments • Condition • Trigger action

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
Activity: <ul style="list-style-type: none"> • Actions 	Activity: <ul style="list-style-type: none"> • Action
Object (associated with a state)	Object node
State	State (no symbol in activity diagram)
Start state	Start
Self Transition or Object Flow	Transition
Synchronization	Synchronization
Decision	Decision
End state	End

Rose Object	OOM Object
Swimlane	Organization unit/swimlane

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.

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"> • Interface of realize • Class of realize • File • URL file • Declaration 	Component: <ul style="list-style-type: none"> • Interface • Class • External file in Ext. Dependencies • URL file in Ext. Dependencies • Description in Notes tab

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

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"> • Device • Processor • Device characteristics • Processor characteristics 	Node: <ul style="list-style-type: none"> • Node • Node • Description • Description
File objects: <ul style="list-style-type: none"> • File • URL definition 	File objects: <ul style="list-style-type: none"> • File object linked to the package • File object
Node association: <ul style="list-style-type: none"> • Connection • Characteristics 	Node association: <ul style="list-style-type: none"> • Node association • Description

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.

Importing XMI Files

PowerDesigner supports the import of an OOM from an XMI file. Since XMI only supports the transfer of objects, PowerDesigner assigns default symbols to imported objects and assigns them to default diagrams.

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.

Exporting XMI Files

PowerDesigner supports the export of an OOM to an XMI file. Since XMI only supports the transfer of objects, any associated PowerDesigner symbols and diagrams are not preserved during the export.

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, Communication Diagram, or Interaction Overview Diagram

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++.

PART III

Object Language Definition Reference

The chapters in this part provide information specific to the object languages supported by PowerDesigner.

CHAPTER 12 **Working with Java**

PowerDesigner supports the modeling of Java programs including round-trip engineering.

Note: To model for Java v5 and higher, select Java as your target language. For earlier versions, select Java 1.x.

For information specific to modeling for Java in the Eclipse environment, see *Core Features Guide > The PowerDesigner Interface > The PowerDesigner Plugin for Eclipse*.

Java Public Classes

Java allows the creation of multiple classes in a single file but one class, and only one, has to be public.

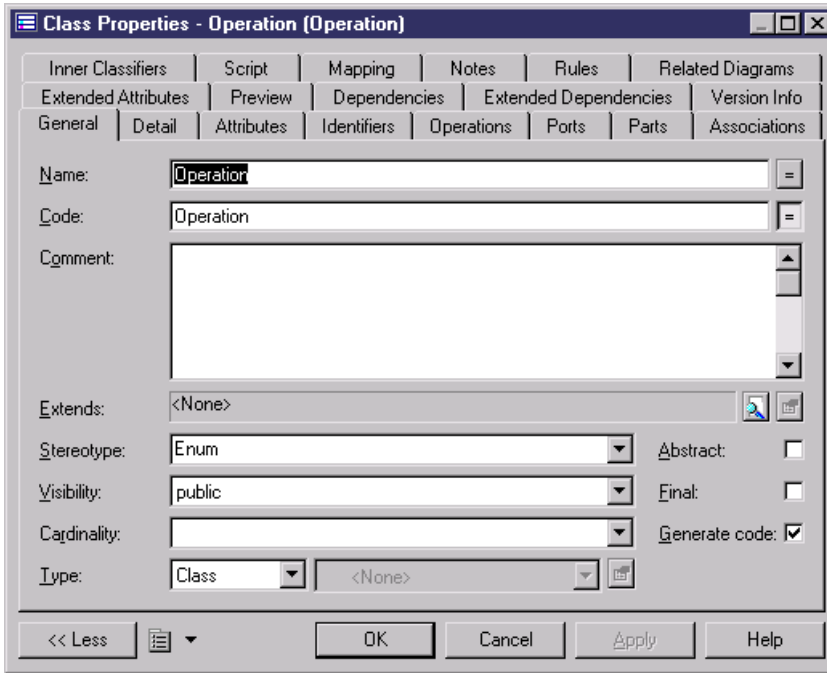
In PowerDesigner, you should create one public class and several dependent classes and draw dependency links with stereotype <<sameFile>> between them. This type of link is handled during generation and reverse engineering.

Java Enumerated Types (Enums)

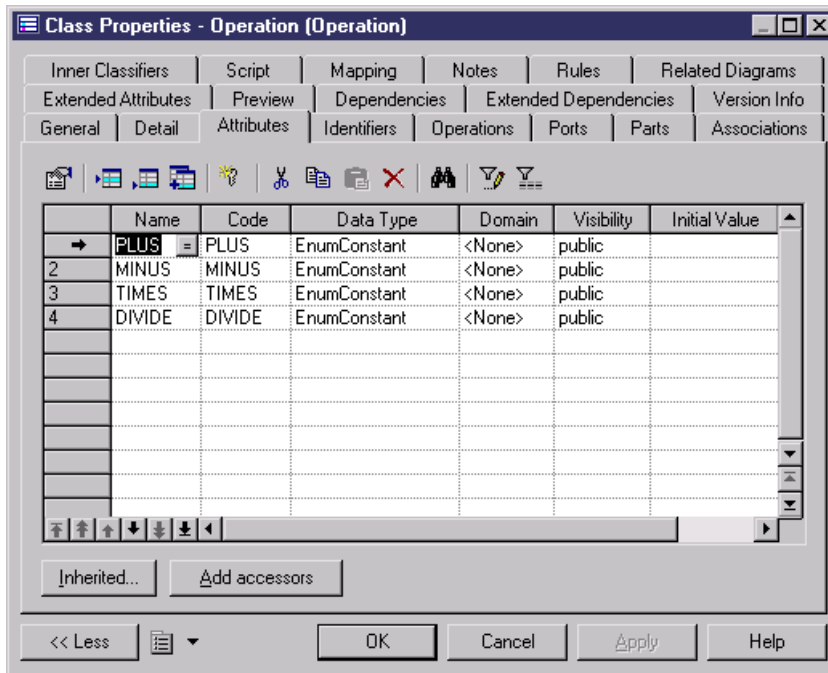
Java 5 supports enumerated types. These replace the old typesafe enum pattern, and are much more compact and easy to maintain. They can be used to list such collections of values as the days of the week or the suits of a deck of cards, or any fixed set of constants, such as the elements in a menu.

An enum is represented by a class with an <<enum>> stereotype.

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.

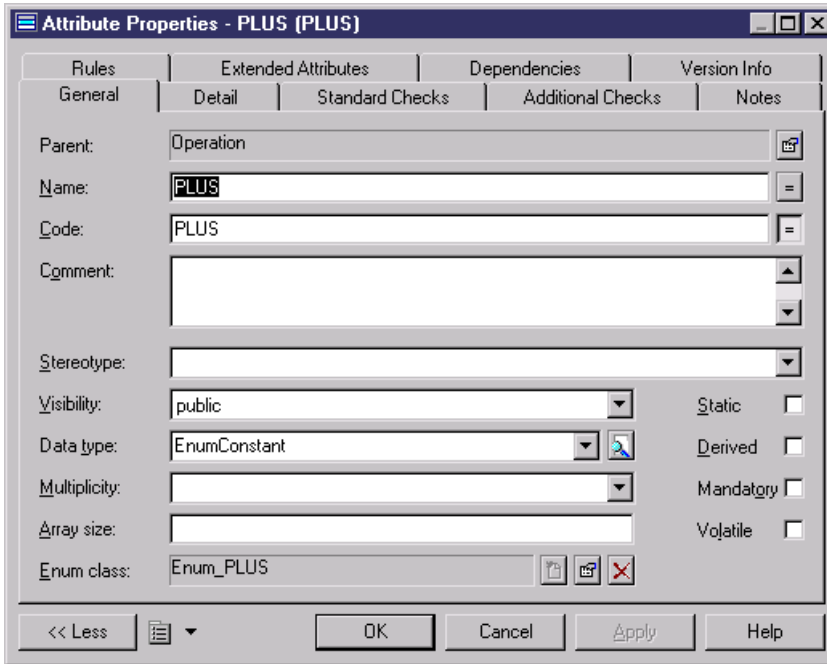


3. Click the Attributes tab, and add as many attributes as necessary. These attributes have, by default, a data type of EnumConstant. For example, to create an enum type that contained standard mathematical operations, you would create four EnumConstants with the names "PLUS", "MINUS", "TIMES", and "DIVIDE".

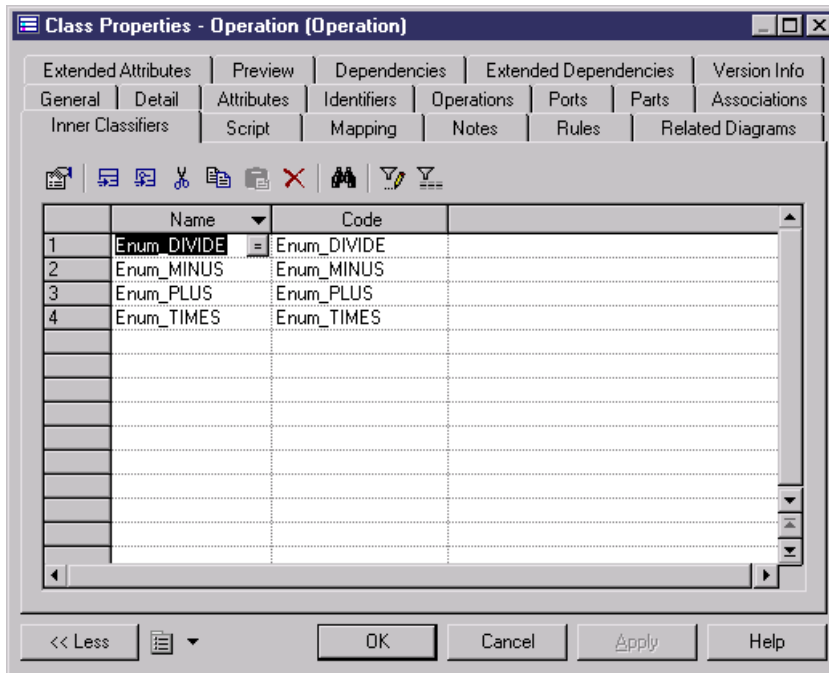


Note that, since a Java enum is a full featured class, you can also add other kinds of attributes to it by clicking in the Data Type column and selecting another type from the list.

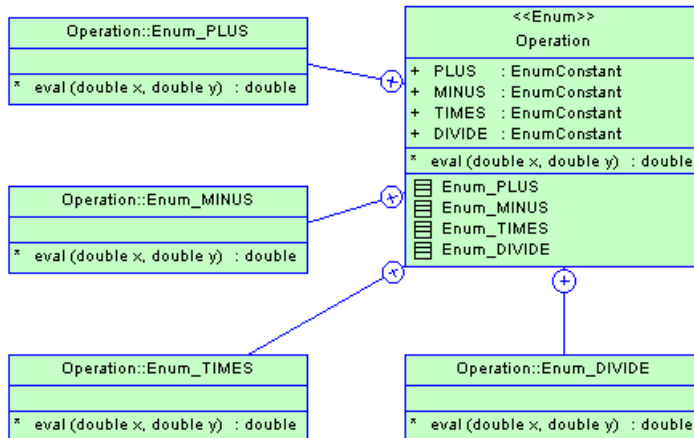
4. [optional] You can create an anonymous class for an EnumConstant by selecting its row on the Attributes tab and clicking the Properties tool to open its property sheet. On the General tab, click the Create tool next to the Enum class box to create the internal class.



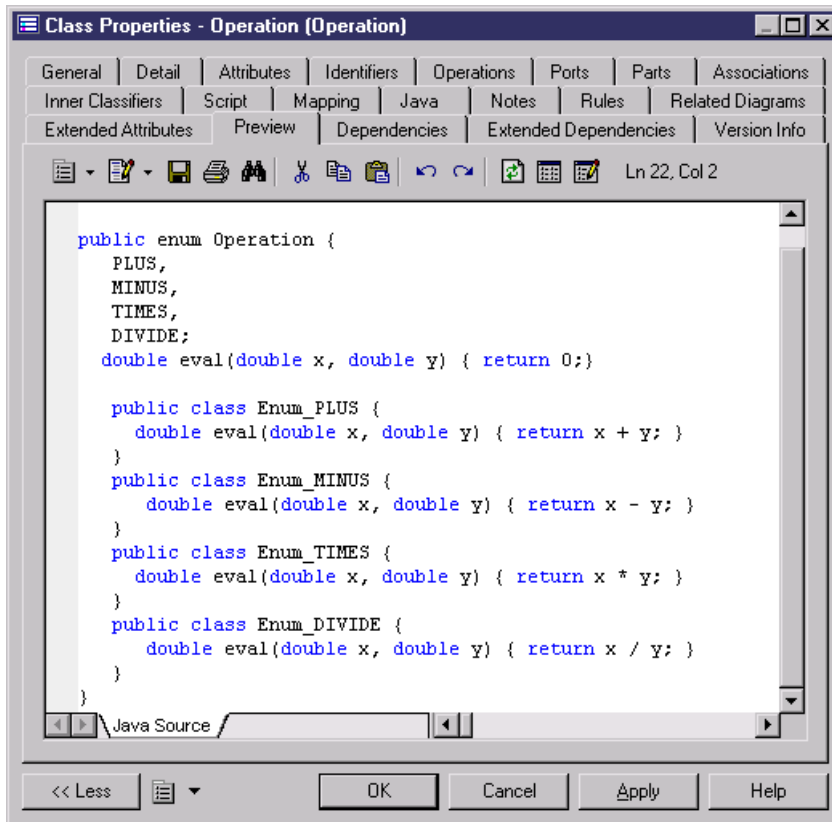
These anonymous classes will be displayed on the Inner Classifiers tab of the Enum class property sheet:



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:



JavaDoc Comments

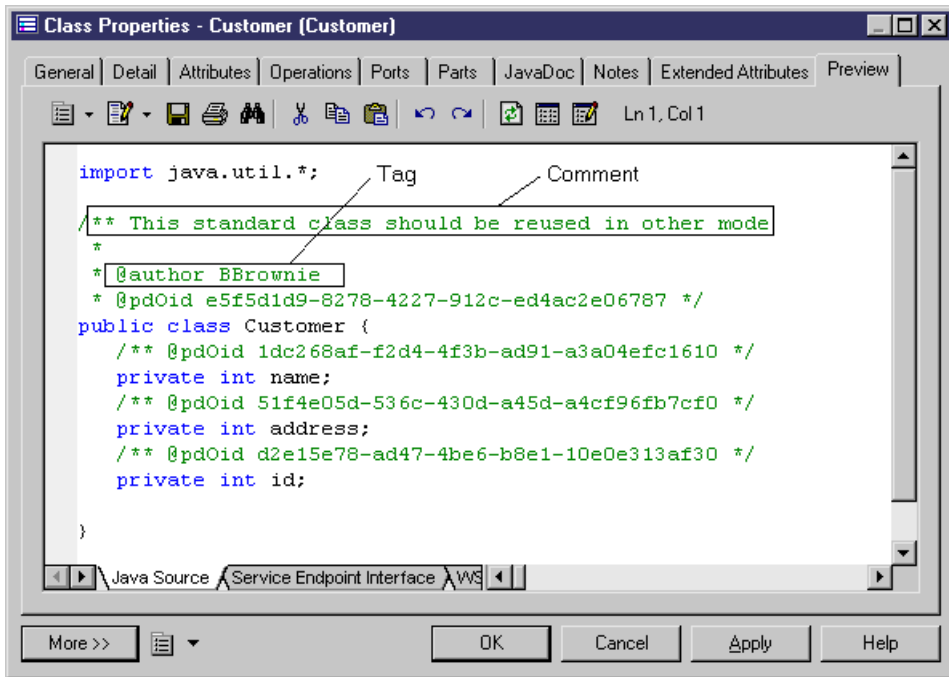
Javadoc is a tool delivered in the JDK that parses the declarations and documentation comments in a set of Java source files and produces a corresponding set of HTML pages describing model objects.

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. 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 ObjectOutputStreamField component of a Serializable class' serialPersistentFields member	Attribute	Javadoc misc
@serial	Used in the doc comment for a default serializable field	Attribute	Javadoc misc

Javadoc	Description	Applies to	Corresponding extended attribute
@param	Adds a parameter to the Parameters section	Attribute	Javadoc misc

Defining Values for Javadoc Tags

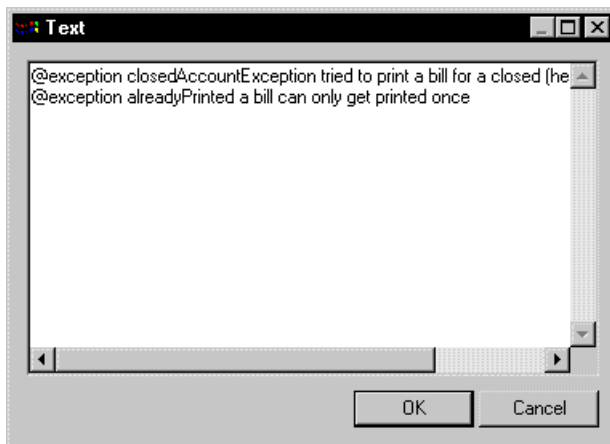
You can define values for Javadoc tags from the Javadoc tab of an object property sheet.

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

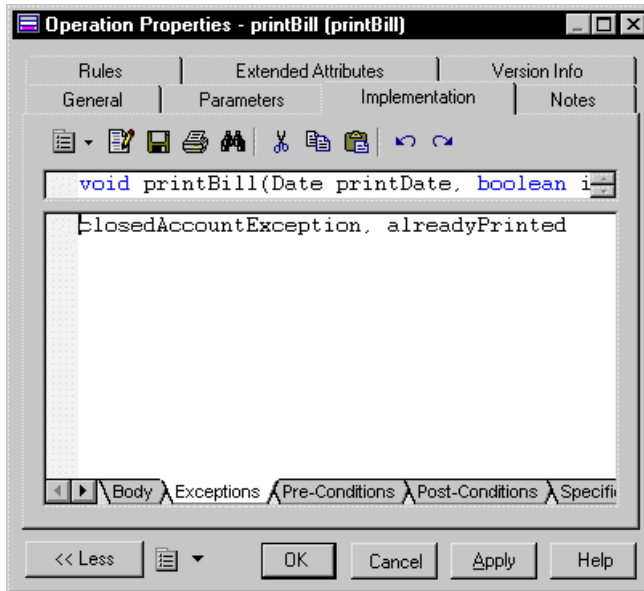
@author is not generated if the extended attribute has no value.

@exceptions and @throws

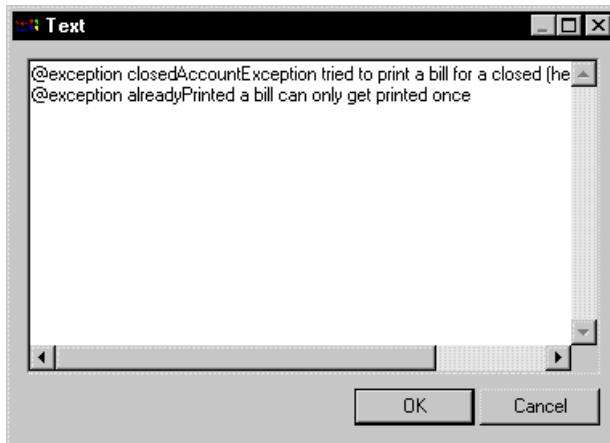
@exceptions and @throws are synonymous Javadoc tags used to define the exceptions that may be thrown by an operation.

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.

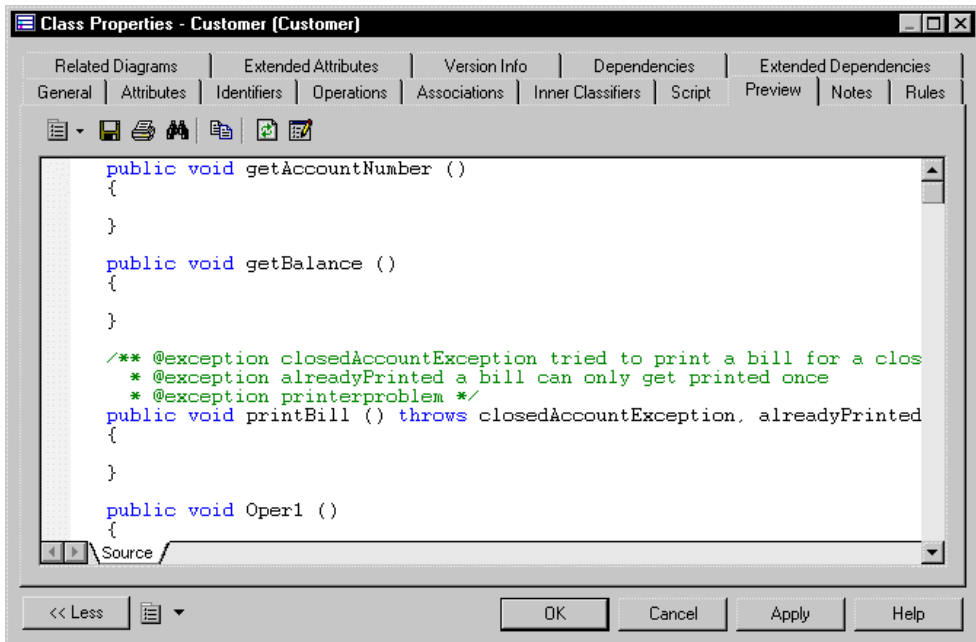


- In the same property sheet, click the Extended Attributes tab, select the Javadoc@exception line in the list and click the ellipsis button in the Value column. Type values for each declared exception, do not forget to repeat @exception or @throws before each exception.



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:



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* on page 353.

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)* on page 65.

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.

To enable the Strictfp keyword:

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

Enterprise Java Beans (EJBs) V2

The Java TM 2 Platform, Enterprise Edition (*J2EE*™) 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.

Java 2 Enterprise Edition supports the following technologies that you can use in PowerDesigner:

- Enterprise JavaBean (EJB)
- Java Servlets (Servlets)
- Java Server Page (JSP)

The following sections develop how to use these technologies in PowerDesigner.

EJB

Sun Microsystems definition of Enterprise JavaBeans™ is:

"The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional and multi-user secure. These applications may be written once and then deployed on any server platform that supports the Enterprise JavaBeans specification".

EJBs are non-visual, remotely executable components that can be deployed on a server, they provide a component architecture framework for creating distributed n-tier middleware. They communicate with clients to make some services available to them.

Enterprise JavaBeans enable rapid and simplified development of distributed, transactional, secure, and portable Java applications. EJB servers reduce the complexity of developing middleware by providing automatic support for middleware services such as transactions, security and database connectivity.

For more information on EJB, a specification is available from Sun Web site, at <http://java.sun.com>.

PowerDesigner supports the EJB 2.0 specification, with special emphasis on entity beans (both CMP and BMP). PowerDesigner implementation of EJB support can take full advantage of the tight integration between PDM and OOM.

What Do You Need to Work with EJBs?

You need 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. You also need a J2EE application server supporting EJB 2.0.

It is also recommended that you set up 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
```

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: <i>CMP (Container Managed Persistence)</i> With CMP Entity Beans, persistence is handled by the component server (also known as Container) <i>BMP (Bean Managed Persistence)</i> With BMP Entity Beans, persistence management is left to the bean developer
<i>Session Beans</i> (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)
<i>Message Driven Beans</i>	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.

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.EJBLocal-Home 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

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* on page 365.

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* on page 8). 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.

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.

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* on page 361.

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 <i>Using EJB types</i> on page 356
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

Property	Description
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.EJBLocal-Home interface
Primary key class	Class providing a pointer into the database. It is only applicable to entity beans
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 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
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 from Sun Microsystems, Inc.

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

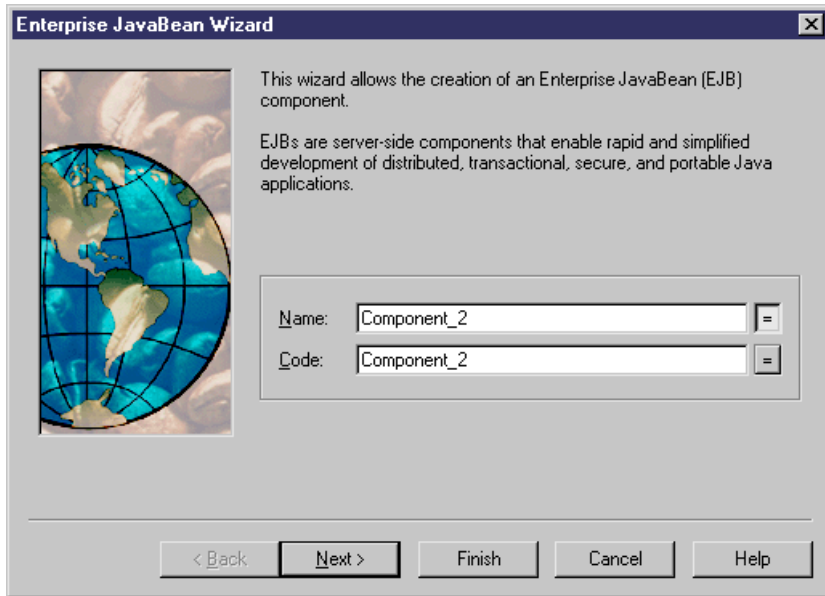
The EJB deployment descriptor supports transaction type for each method: you can specify a transaction type for each method of EJB remote and local interface.

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.

1. Select **Tools > Create Enterprise JavaBean** from a class diagram.

The Enterprise JavaBean 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. Type 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.

When you have finished using the wizard, the following actions are executed:

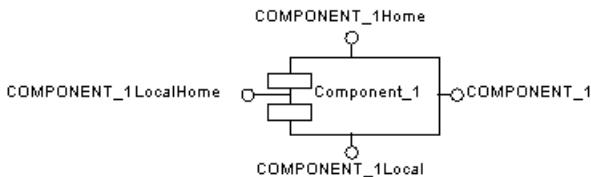
- An EJB component is created
- Classes and interfaces are associated with the component, and any missing classes or interfaces associated with the component are added

- 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

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.

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:

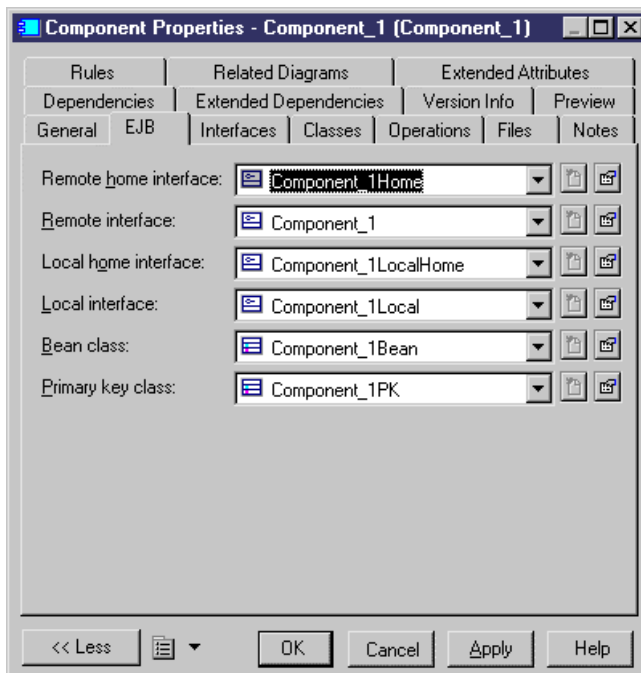
Stereotype	Describes
<<EJBRemoteHome>>	The remote home interface
<<EJBRemote>>	The remote interface
<<EJBLocalHome>>	The local home interface
<<EJBLocal>>	The local interface
<<EJBEntity>>	The bean class of an entity bean
<<EJBSession>>	The bean class of a session bean

Stereotype	Describes
<<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.

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.

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

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.

Note: If you need to modify the methods of an interface, you can do so from the Operations page of the interface property sheet.

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.

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 to display the Implementation page.

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 to display the Preview page.

You can now validate the to-be-generated Java code for the Bean class.

10. Click OK.

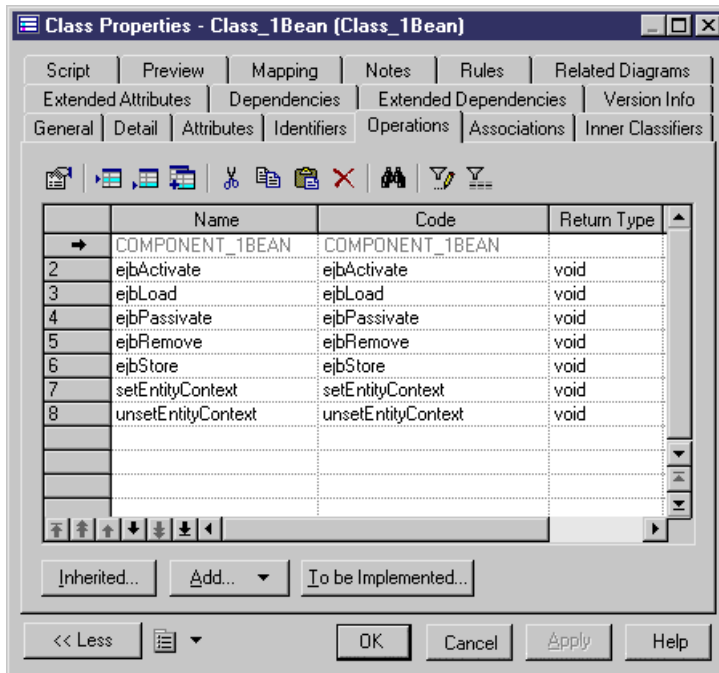
Adding an Operation to an EJB Interface

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.

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.

For more information on synchronization, see *Understanding operation synchronization* on page 365

1. Double-click the Bean class to display the class property sheet.
2. Click the Operations tab to display the Operations page.
3. Click the Add button at the bottom of the page.
4. Select the required EJB operation from the list.



5. Click Apply.

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.

6. Click OK.

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

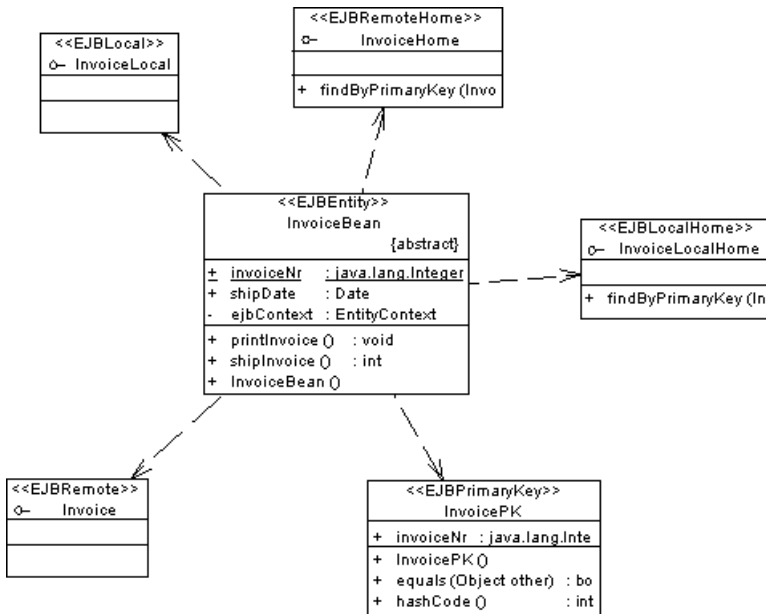
Exceptions are synchronized from the Bean class to interfaces. The exception list of the home interface create method is a superset of the union of the exception lists of the matching `ejbCreate` and `ejbPostCreate` implementation operations in the bean class.

The interface exception attributes are thus updated whenever the exception list of the bean class implementation method is modified.

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 on page 365*).
 - Exceptions - Synchronization occurs from Bean class to interface. (*Understanding Operation Synchronization on page 365*).

- 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 *Chapter 9, Checking an OOM* on page 295).
- 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`

```

/*****
 * Module: INVOICE.java
 * Author: lpelletta
 * 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.Remote
    public Date getSHIPDATE() throws java.rmi.RemoteException;
    public void setSHIPDATE(Date SHIPDATE) throws java.rmi.Remote
}

```

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: lpelletta
* 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);
  
```

<< 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:

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

```

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
    }
  
```

non persistent attribute (pointing to `INVOICEBEAN`)

persistent attribute (pointing to `PRINTINVOICE()`)

<< Less | OK | Cancel | Apply | Help

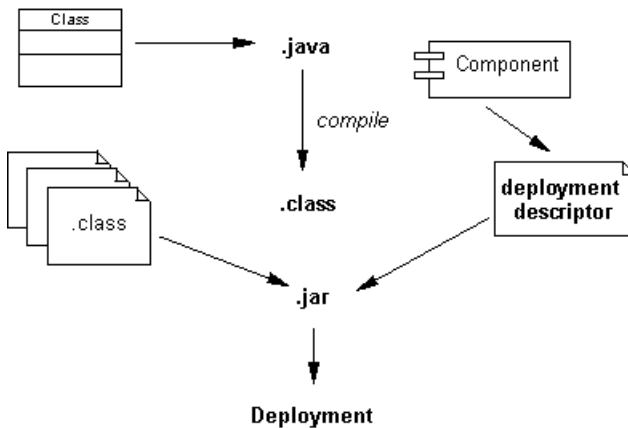
At a higher level, PowerDesigner supports different approaches to assist you in the component development process including:

- 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

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.

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.

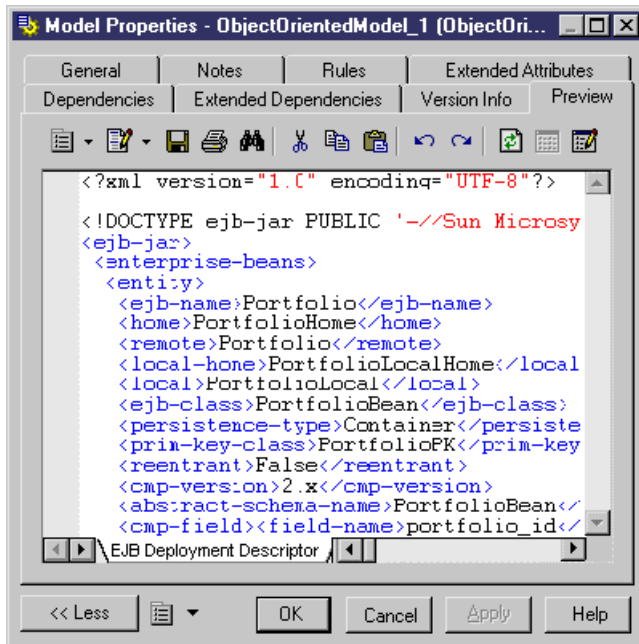
The EJB deployment tool of the application server imports the Java classes from the JAR file and configures the EJBs in the application server based on the description of EJBs contained in the EJB deployment descriptor.

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*.

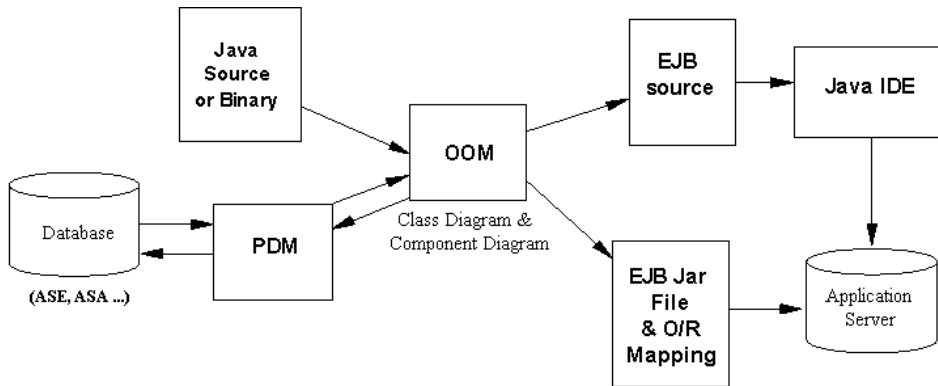
1. Right-click the model in the Browser and select **Properties** to open the model property sheet.
2. Click the **Preview** tab.



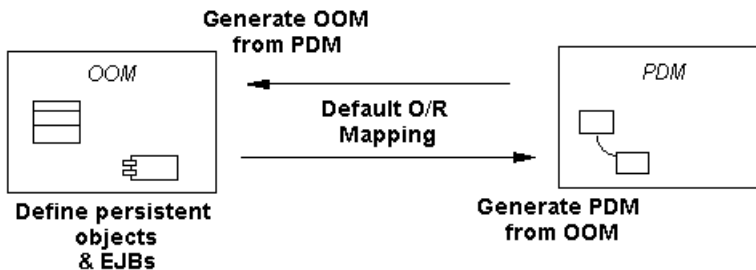
Generating EJBs

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 *Chapter 22, Object/Relational (O/R) Mapping* on page 539.

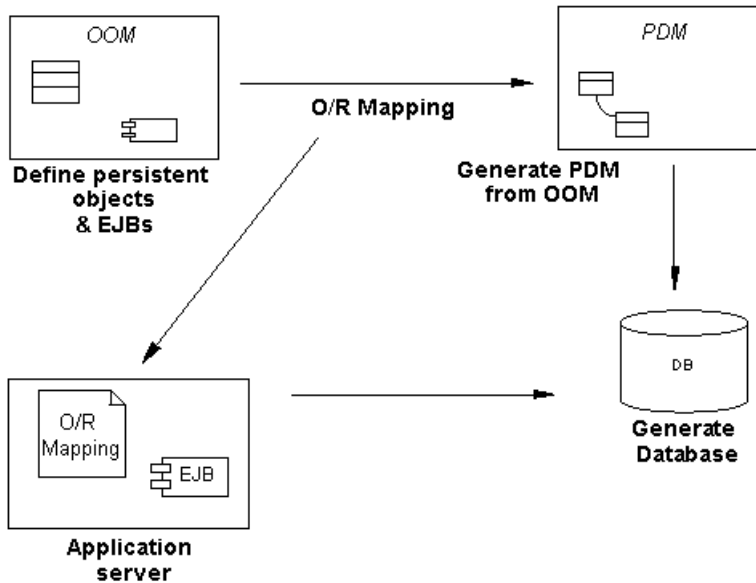
What Kind of Generation to Use?

You may face two situations when generating EJB source code:

- You have no database
- You already have a database

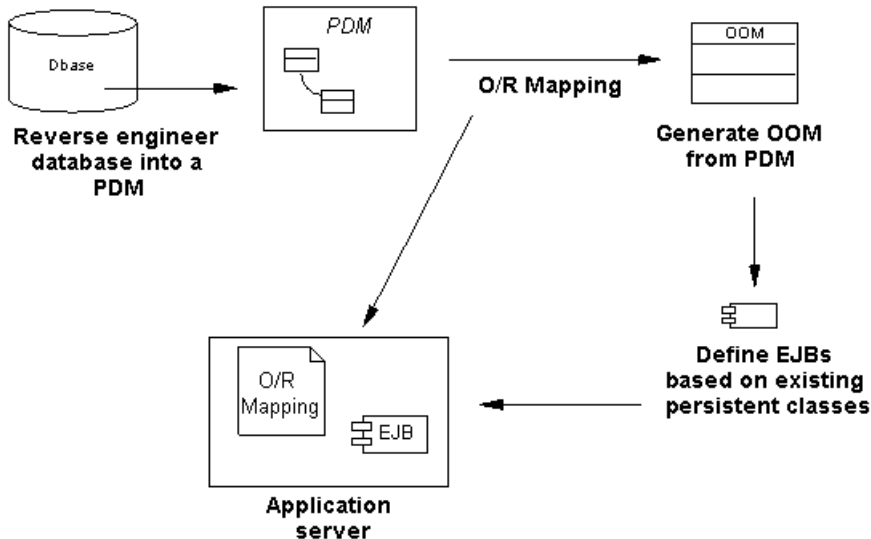
You want to generate EJB source code when creating a database. If there is no database, the development process is the following:

- 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 *Chapter 22, Object/Relational (O/R) Mapping* on page 539.

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:

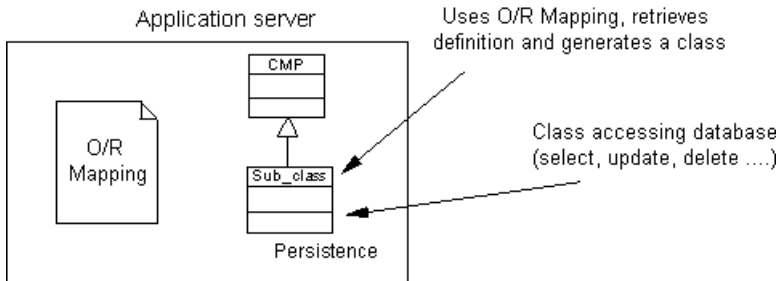
- A CMP involves the application server. It includes the EJB and the O/R mapping descriptor (.XML). The server retrieves both EJB and O/R mapping descriptor to generate the code

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 *Chapter 22, Object/Relational (O/R) Mapping* on page 539.

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.

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.

1. Select **Language > Generate Java Code** to open the Generation dialog.
2. Enter the directory in which you want to generate the Java files.
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* on page 399).
5. [optional] Click the **Tasks** tab and select any appropriate task to perform during generation (see *Generating Java Files* on page 399).
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**.

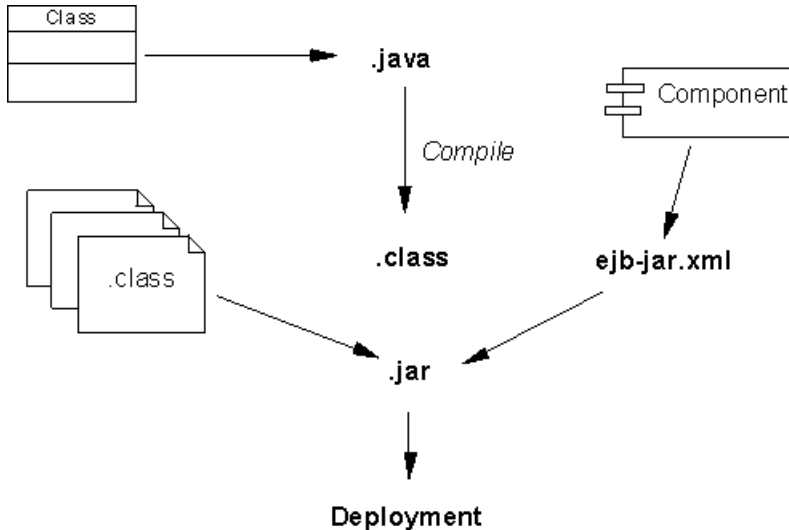
The `ejb-jar.xml` deployment descriptor is created in the META-INF directory and all files are generated in the generation directory.

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.



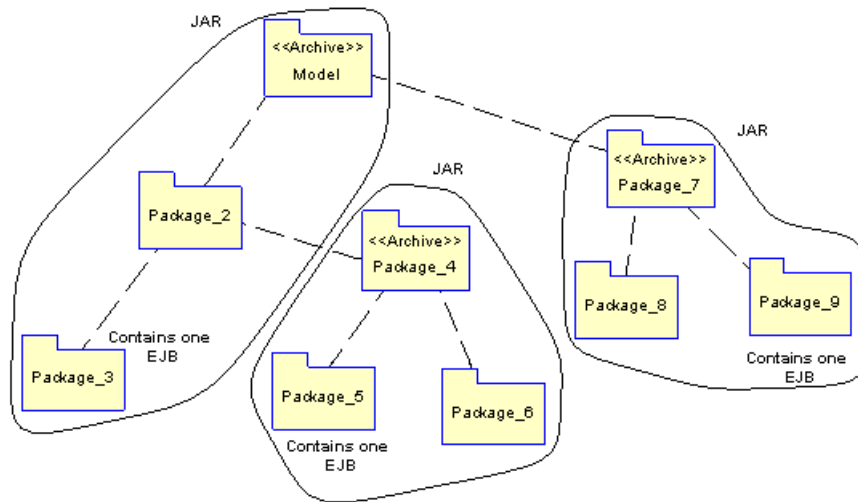
Warning! You must set the values of the commands used during generation from the Variables section of the General Options dialog box in order to enable them. For example, you can set the `javac.exe` and `jar.exe` executables at this location.

For more information on how to set these variables, see *Core Features Guide > The PowerDesigner Interface > Customizing Your Modeling Environment > General Options > Defining 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.



Reverse Engineering EJB Components

PowerDesigner can reverse engineer EJB components located in .JAR files.

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 > The PowerDesigner Interface > 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.

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 *Chapter 23, Generating Persistent Objects for Java and JSF Pages* on page 571).

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.

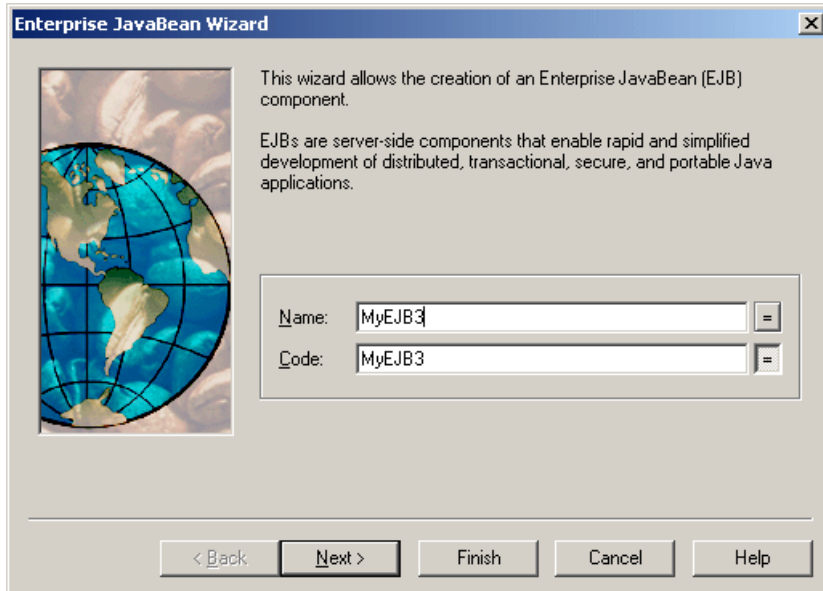
Note: The EJB 2.1 programming model (with deployment descriptors and home/remote interfaces) is still valid and supported by PowerDesigner. The new simplified model, which is only available with Java 5.0, does not entirely replace the EJB 2.1 model.

Creating an EJB 3.0 with the Enterprise JavaBean Wizard

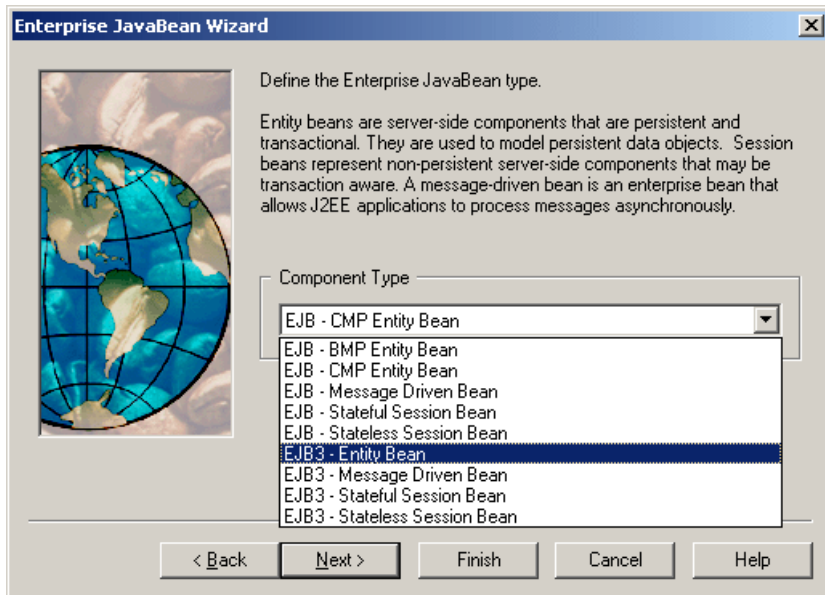
To create an EJB3, launch the Enterprise JavaBean Wizard from a class diagram.

The following types of EJB3 beans are available:

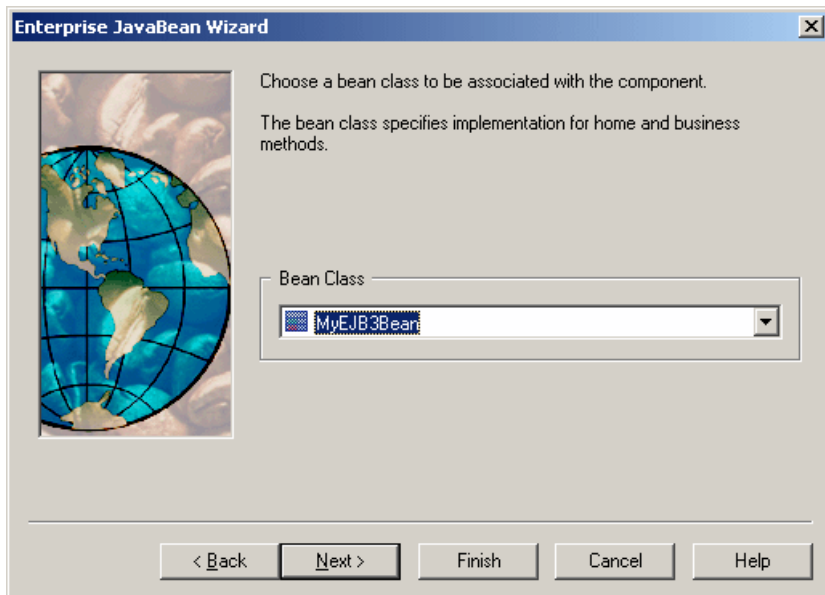
- Entity Bean – generated with an @Entity annotation
 - Message Driven Bean – generated with a @MessageDriven annotation
 - Stateful Session Bean – generated with an @Stateful annotation
 - Stateless Session Bean – generated with an @Stateless annotation
1. If you have already created a class to serve 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:



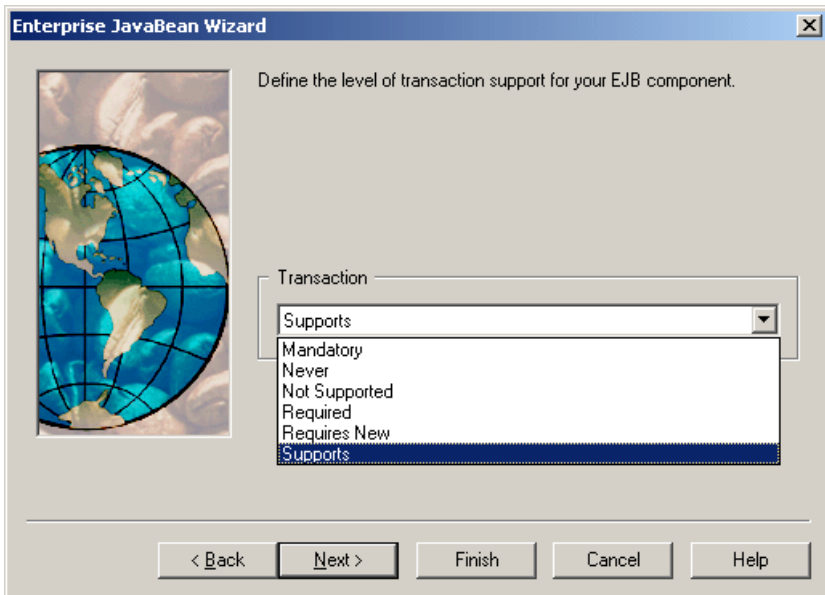
2. Specify a name for the EJB, and then click Next to go to the next screen:



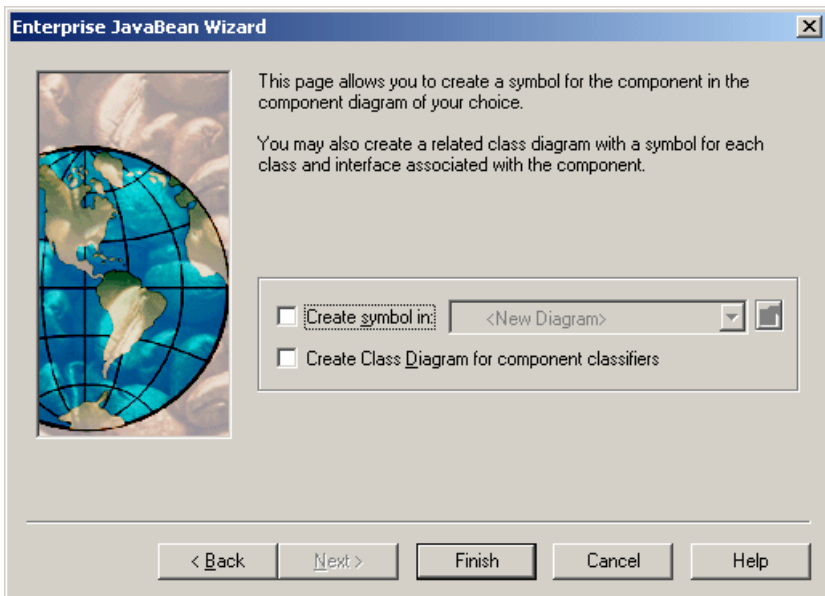
3. Choose a type of EJB3, and then click Next to go to the next screen:



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 screen:



5. Choose the desired level of transaction support, and then click Next to go to the next screen:



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.

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"> • Bean • Container <p>Generated as a <code>@TransactionManagement</code> annotation.</p>
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"> • Not Supported • Supports • Required • Requires New • Mandatory • Never <p>Generated as a <code>@TransactionAttribute</code> annotation.</p>
Exclude Default Interceptors	<p>Specifies that the invocation of default interceptor methods is excluded.</p> <p>Generated as a <code>@ExcludeDefaultInterceptors</code> annotation.</p>
Exclude Superclass Listeners	<p>Specifies that the invocation of superclass listener methods is excluded.</p> <p>Generated as a <code>@ExcludeSuperclassListeners</code> annotation.</p>
Mapped Name	<p>Specifies a product specific name.</p> <p>Generated as a <code>@MappedName</code> annotation.</p>
Run-As	<p>Specifies the bean's run-as property (security role).</p> <p>Generated as a <code>@RunAs</code> annotation.</p>
Declare Roles	<p>Specifies references to security roles.</p> <p>Generated as a <code>@DeclareRoles</code> annotation.</p>

Property	Description
Roles Allowed	Specifies the roles allowed for all bean methods. Generated as a @RolesAllowed annotation.
Permit All	Specifies that all roles are allowed for all bean business methods. Generated as a @PermitAll annotation.

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).

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.

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:

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.

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. Generated as a @Init annotation.
Remove Method	Specifies an remove method. Generated as a @Remove annotation.
Post-Construct	Specifies a post construct method. Generated as a @PostConstruct annotation.
Post-Activate	Specifies a post activate method. Generated as a @PostActivate annotation.
Pre-Passivate	Specifies a pre passivate method. Generated as a @PrePassivate annotation.
Pre-Destroy	Specifies a pre destroy method. Generated as a @PreDestroy annotation.
Interceptor Method	Specifies an interceptor method. Generated as a @AroundInvoke annotation.
Timeout Method	Specifies a timeout method. Generated as a @Timeout annotation.
Exclude Default Interceptors	Excludes invocation of default interceptor for the method. Generated as a @ExcludeDefaultInterceptors annotation.
Exclude Class Interceptors	Excludes invocation of class-level interceptors for the method. Generated as a @ExcludeClassInterceptors annotation.
Transaction Attribute Type	Specifies a Transaction Attribute Type for the method. Generated as a @TransactionAttribute annotation.
Permit All Roles	Specifies that all roles are permitted for the method. Generated as a @PermitAll annotation.

Property	Description
Deny All Roles	Specifies that method may not be invoked by any security role. Generated as a @DenyAll annotation.
Roles Allowed	Specifies the roles allowed for the method. Generated as a @RolesAllowed annotation.

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.

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 Properties tool beside this box to display the property sheet of the class, or click the Create 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

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.

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.

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.

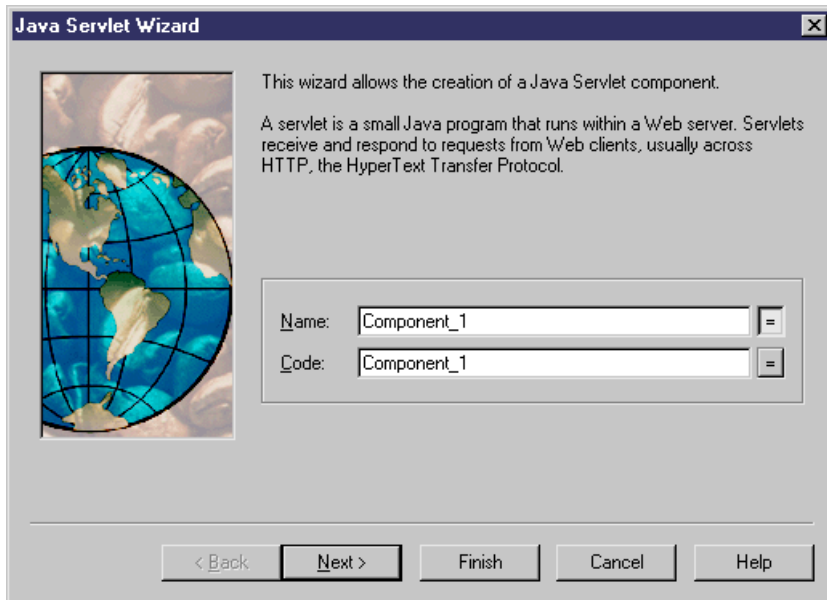
You can also create several servlets of the same type by selecting several classes at the same time. The wizard will automatically create one servlet per class. The classes you have selected in the class diagram become servlet classes. They are renamed to fit the naming conventions standard, and they 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 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
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

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.

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

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:

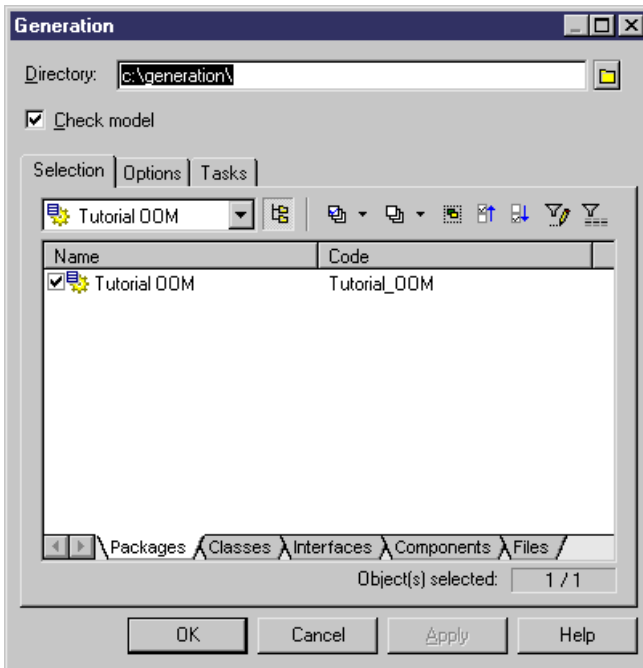
- When the servlet class is attached to a servlet component, implementation methods for operations defined in the `javax.servlet.Servlet` interface are added by default. This interface is the base interface for all servlets, it may be included in the code depending on the servlet type selected. For `HttpServlet` 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**.

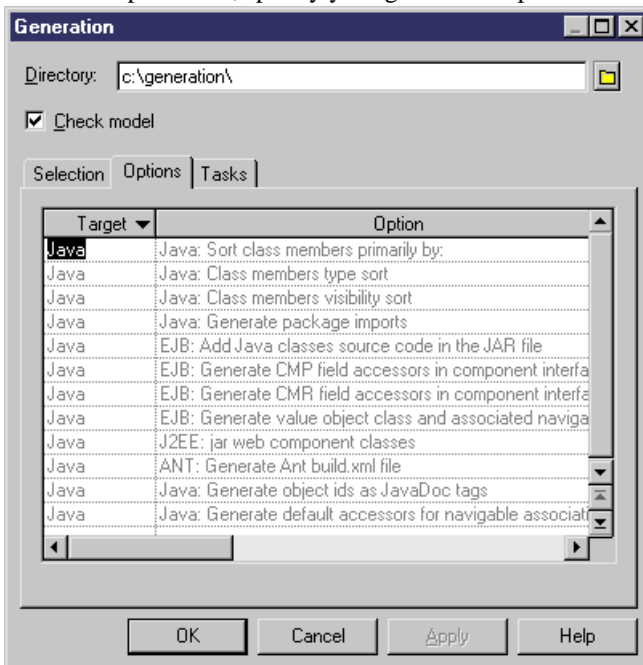
Generating Servlets

The generation process retrieves all classes used by the servlet components to generate the servlets.

1. Select **Language > Generate Java 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 tabbed pages.

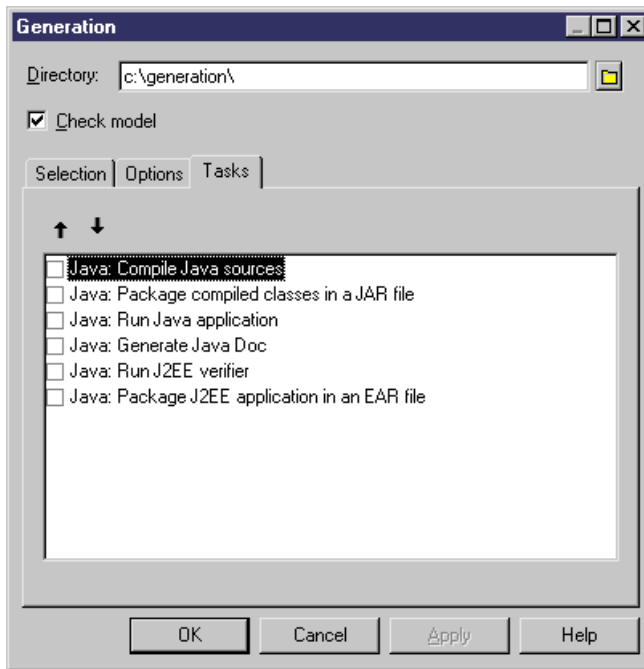


4. Click Apply.
5. Click the Options tab, specify your generation options.



For more information on the generation options, see *Generating Java Files* on page 399.

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* on page 399.

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 > The PowerDesigner Interface > Customizing Your Modeling Environment > General Options > Defining Environment Variables*.

8. 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.

9. Click Close.

The web.XML file is created in the WEB-INF directory and all files are generated in the generation directory.

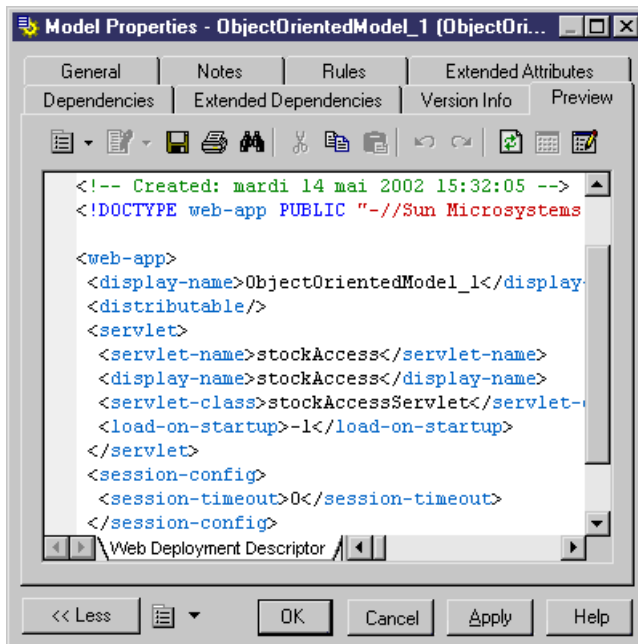
Generating Servlet Web Deployment Descriptor

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* on page 387.

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.



Generating WARs

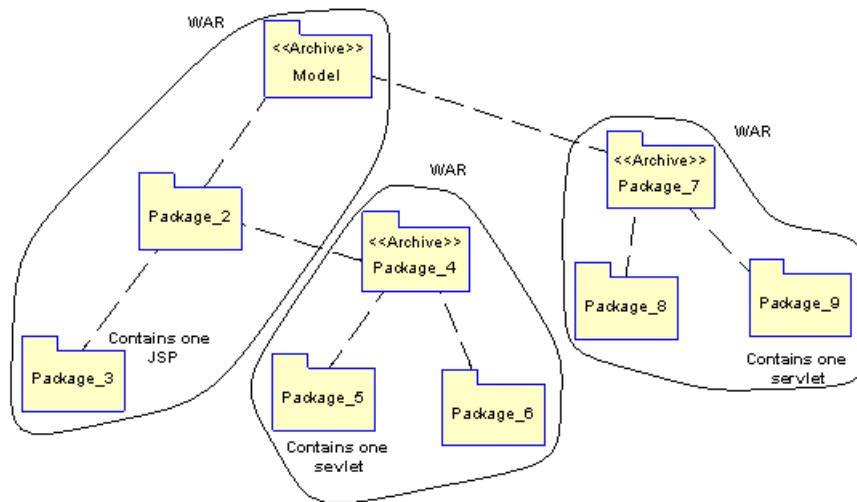
You can package servlets and JSPs into a .WAR file.

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* on page 399.

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.

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* on page 403.

1. Select **Language > Reverse Engineer Java**.

The Reverse Java dialog box is displayed.

2. Select Archive from the Reverse list in the Selection page.

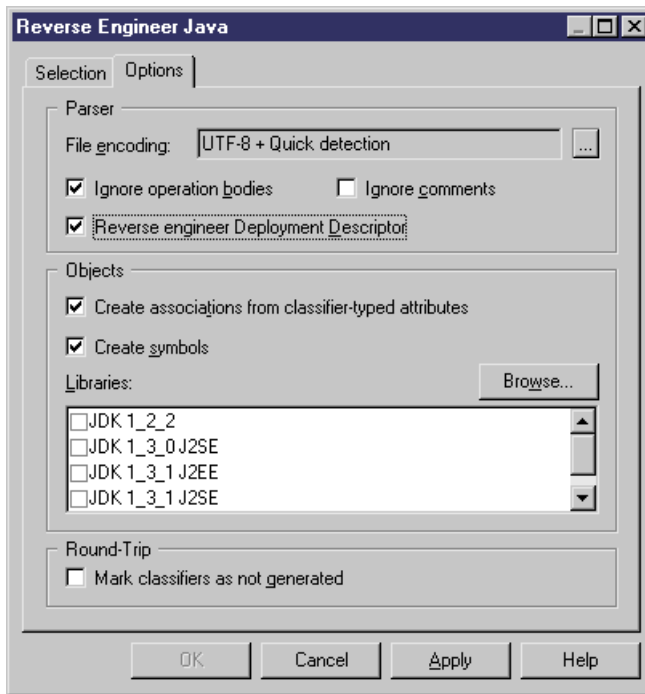
3. Click the Add button.

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 > The PowerDesigner Interface > 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.

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.

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* on page 394.

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 > text editor** from the contextual menu to display the content of the file object.

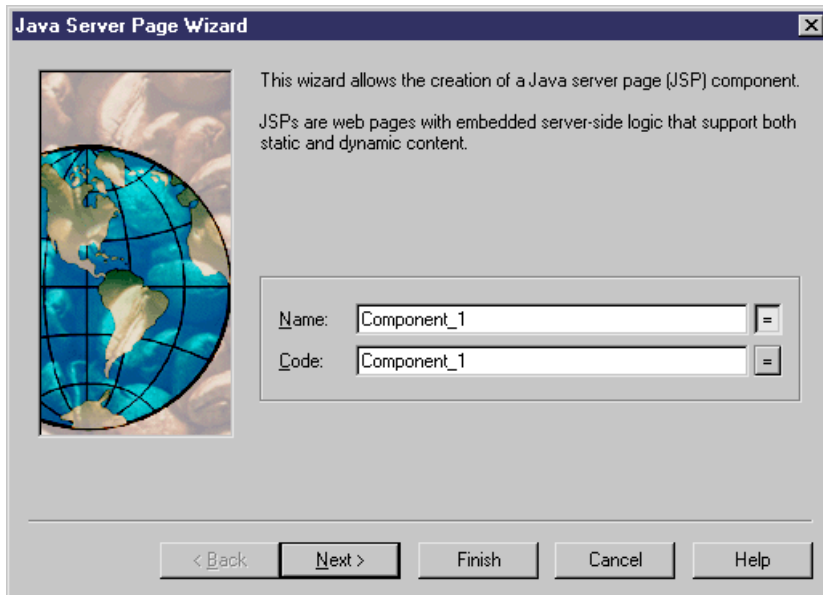
Creating a JSP with the Wizard

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.

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.

1. Select **Tools > Create JSP** 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.

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* on page 226.
- 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

Generating JSPs

The generation process generates only file objects with the `Artifact` property selected.

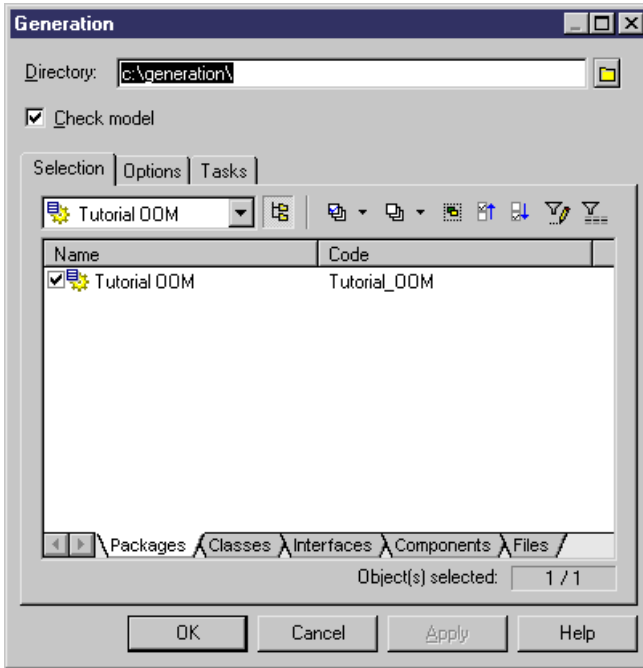
Generating JSP Web Deployment Descriptor

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.

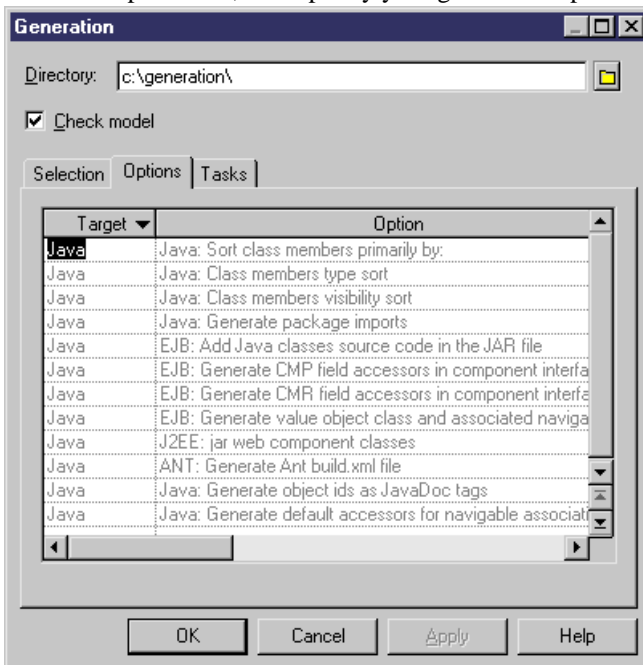
The Web application deployment descriptor is generated per package. A `WAR` command available in the `Tasks` page of the `Generation` dialog box allows you to build a `Web Archive` that contains the `Web deployment descriptor`, in addition to all classes and file objects referenced by JSP components. 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.

1. Select **Language > Generate Java Code** to display the `Generation` dialog box.
2. Select or browse to a directory that will contain the generated files.
3. Click the `Selection`, then select the objects you need in the different tabbed pages.

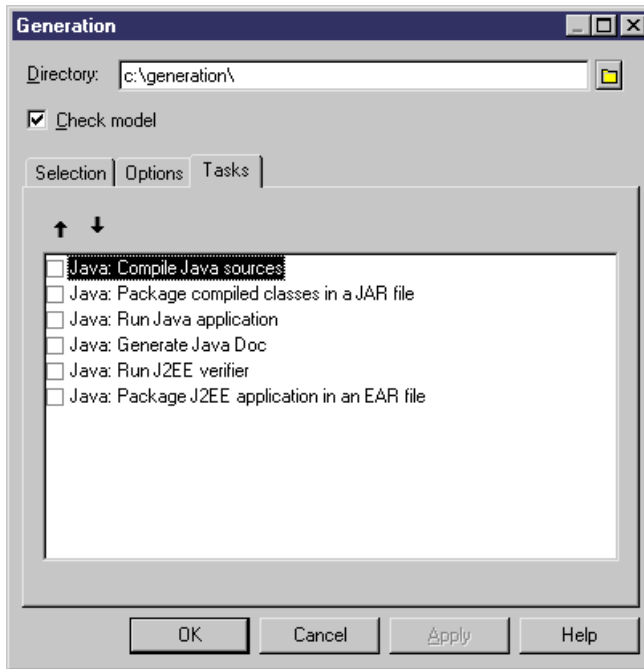


4. Click Apply.
5. Click the Options tab, then specify your generation options in the Options page.



For more information on the generation options, see *Generating Java Files* on page 399.

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* on page 399.

You must beforehand set the environment variables from the Variables tab of the General Options dialog box in order to activate them in this page.

For more information on how to set these variables, see *Core Features Guide > The PowerDesigner Interface > Customizing Your Modeling Environment > General Options > Defining Environment Variables*.

8. 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.

9. Click Close.

The web.XML file is created in the WEB-INF directory and all files are generated in the generation directory.

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.

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* on page 403.

1. Select **Language > Reverse Engineer Java.**

The Reverse Java dialog box is displayed.

2. Select Archive from the Reverse list in the Selection page.

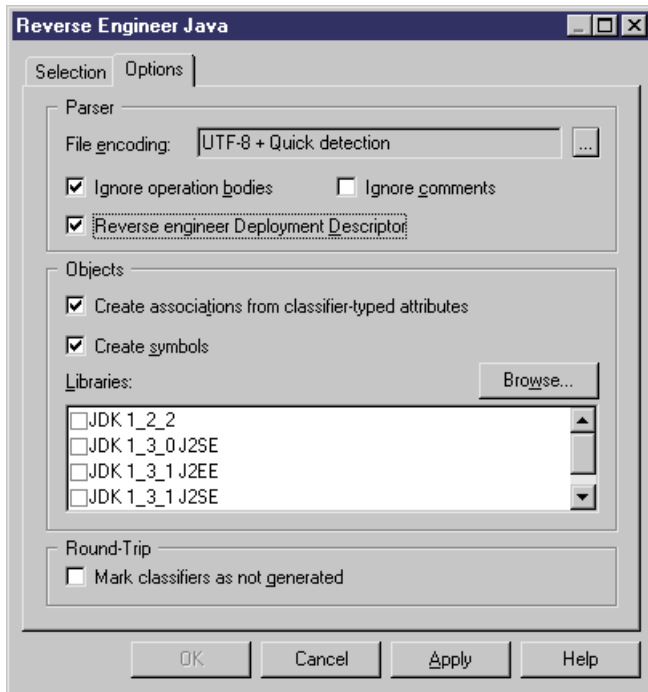
3. Click the Add button.

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 > The PowerDesigner Interface > 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.

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.

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.

1. Select **Language > Generate Java Code** to open the Java Generation dialog.
2. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see *Chapter 9, Checking an OOM* on page 295).
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* on page 272).
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	Sorts attributes and operations by: <ul style="list-style-type: none"> • Visibility- [default] Public attributes and operations are generated before private ones • Type - Attributes and operations are sorted by type whatever their visibility
Java: Class members type sort	Sorts attributes and operations in the order: <ul style="list-style-type: none"> • Attributes – Operations - Attributes are generated before the operations • Operations – Attributes - Operations are generated before the attributes
Java: Class members visibility sort	Sorts attributes and operations in the order: <ul style="list-style-type: none"> • Public – Private - Public attributes and operations are generated before private ones • Private – Public - Private attributes and operations are generated before public attributes and operations • None - Attributes and operations order remains unchanged

Option	Description
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 JavaDoc 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
Java: Generate default accessors for navigable associations	Generates the getter and setter methods for navigable associations
Ant: Generate Ant build.xml file	Generates the build.xml file. You can use this file if you have installed Ant
EJB: Generate CMP field accessors in component interfaces	Generates CMP fields getter and setter operations to EJB interfaces
EJB: Generate CMR field accessors in component interfaces	Generates CMR fields getter and setter declarations in EJB interfaces
EJB: Add Java classes source code in the JAR file	Includes Java classes code in the JAR
EJB: Generate value object class and associated navigation methods for CMP Entity Beans	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
J2EE: Jar Web component classes	Archives Web component classes in a Jar

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*.

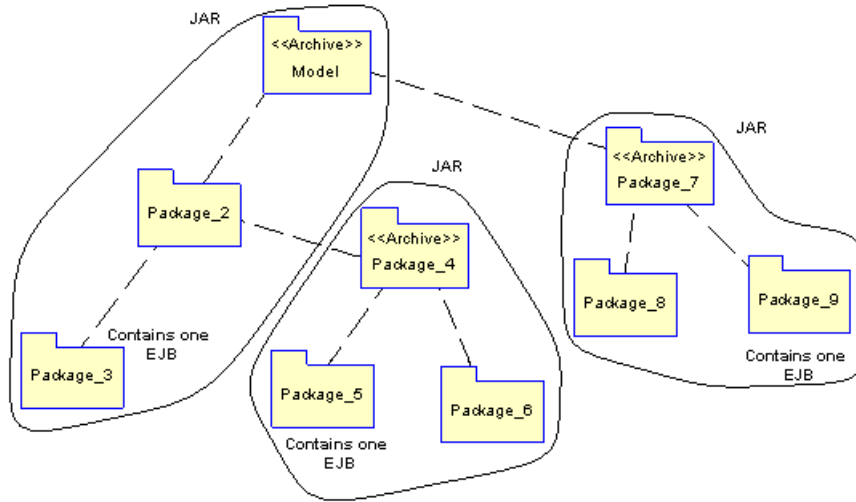
- [optional] Click the **Generated Files** tab and specify which files will be generated. By default, all files are generated.

For information about customizing the files that will be generated, see *Customizing and Extending PowerDesigner > Extension Files > Templates and Generated Files (Profile)*.

7. [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.

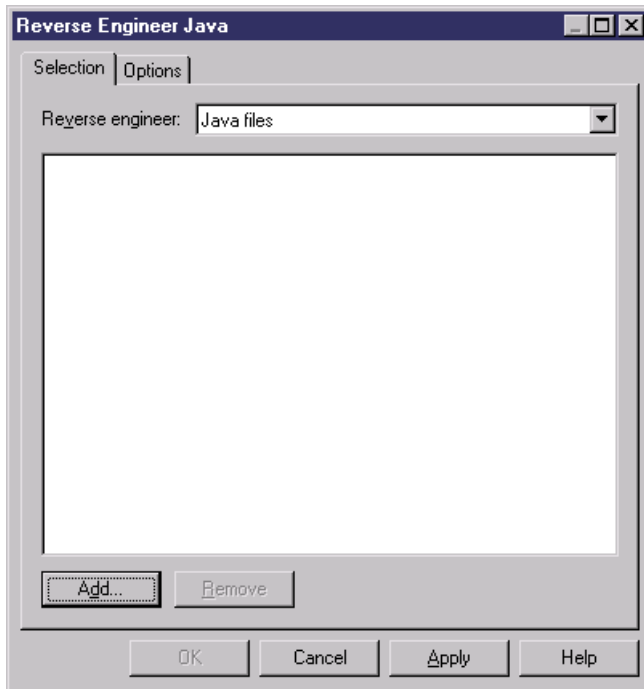
When generation is complete, the Generated Files dialog opens, listing the files that have been generated to the specified directory. Select a file in the list and click **Edit** to open it in your associated editor, or click **Close** to exit the dialog.

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.

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.

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* on page 405

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.

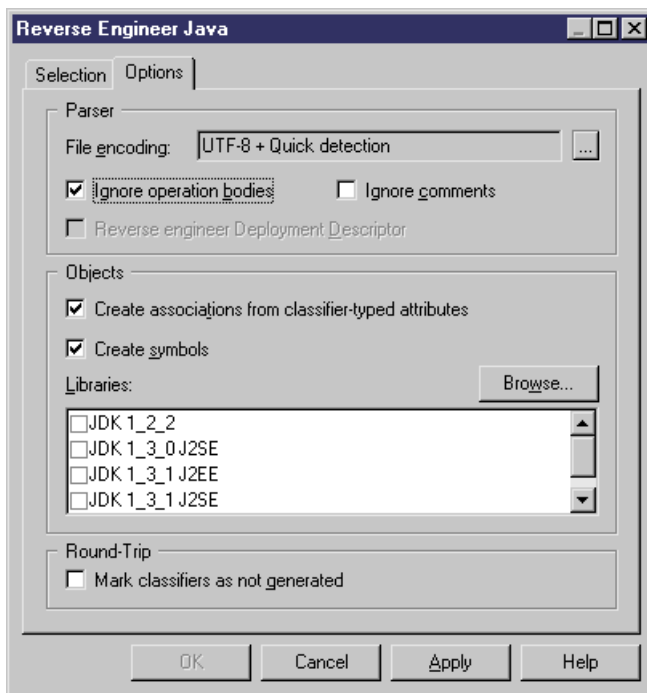
5. Click OK to begin the reverse engineering process. If the model in which you are reverse engineering already contains data, the Merge Models dialog box will open to allow you to specify whether to control whether existing objects will be overwritten.

For more information on merging models, see *Core Features Guide > The PowerDesigner Interface > 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

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 <i>Reverse Engineering EJB Components</i> on page 376, <i>Reverse Engineering Servlets</i> on page 391, and <i>Reverse Engineering JSPs</i> on page 398.
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.
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 > Linking and Synchronizing Models > Shortcuts and Replicas > 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.

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

Working with the Eclipse Modeling Framework (EMF)

PowerDesigner supports modeling in the EMF language including round-trip engineering.

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 lots of Eclipse based tools, including Sybase DI and Sybase WorkSpace.

For more information on EMF, see the EMF documentation and tutorials at <http://www.eclipse.org/emf>.

EMF Objects

The following objects are available in an OOM targeted for EMF.

EPackages

PowerDesigner models EMF EPackages as standard UML packages, but with additional properties.

EPackage property sheets contains all the standard package tabs along with the EMF tab, the properties of which are listed below:

Property	Description
Namespace prefix	Used when references to instances of the classes in this package are serialized.
Namespace URI	Appears in the xmlns tag to identify this package in an XMI document.
Base package name	Contains the generated code for the model.

For information about creating and working with packages, see *Packages (OOM)* on page 50.

EClasses, EEnums, and EDataTypes

PowerDesigner models EMF EClasses as standard UML classes, and EEnums and EDataTypes as standard UML classes with an Enum and an EDataType stereotype, respectively.

For information about creating and working with classes, see *Classes (OOM)* on page 34.

EClass, Eenum, and EDataType property sheets contain all the standard class tabs along with the EMF tab, the properties of which are listed below:

Property	Description
Instance Class Name	Specifies the data type instance class name.

EAnnotations

PowerDesigner models EMF EAnnotations as standard UML classes, with an AnnotationType stereotype.

For information about creating and working with classes, see *Classes (OOM)* on page 34.

EAnnotation property sheets contain all the standard class tabs along with the EMF tab, the properties of which are listed below:

Property	Description
References	Specifies the EMF annotation references.
URI	Specifies the EMF annotation source.

EAttributes and EEnumLiterals

PowerDesigner models EMF EAttributes and EEnumLiterals (EAttributes belonging to EEnums) as standard UML attributes, but with additional properties.

For information about creating and working with attributes, see *Attributes (OOM)* on page 65.

EAttribute and EEnumLiteral property sheets contains all the standard attribute tabs along with the EMF specific tabs listed in the sections below.

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.

EReferences

PowerDesigner models EMF EReferences as standard UML associations, but with additional properties.

For information about creating and working with associations, see *Associations (OOM)* on page 88.

EReference property sheets contains all the standard association tabs along with the EMF tab, the properties of which are listed below:

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.

EOperations and EParameters

PowerDesigner models EMF EOperations and EParameters as standard UML operations and operation parameters.

For information about creating and working with operations, see *Operations (OOM)* on page 78.

Generating EMF Files

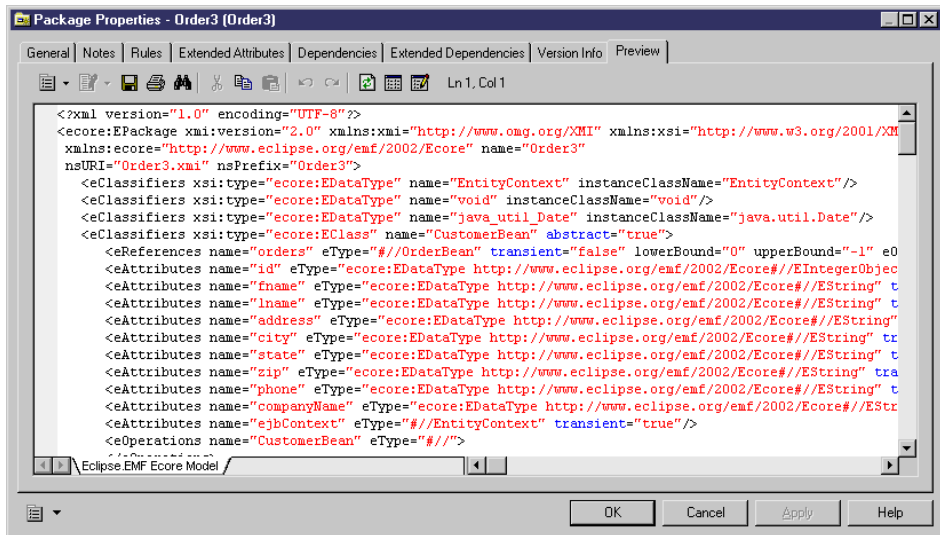
PowerDesigner can generate EMF .ecore and .genmodel files.

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.



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.

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 > The PowerDesigner Interface > Comparing and Merging Models*.

The packages are added to your model.

CHAPTER 14 Working with IDL

PowerDesigner supports modeling in the Interface Definition Language.

IDL stands for Interface Definition Language. It was defined for the Common Object Request Broker Architecture (CORBA) to enable client/server object interaction.

The IDL language is independent from the platform you are working on. In this manual, we refer to version 2.4 of the IDL-CORBA specification, and we support the concepts defined in the UML Profile for CORBA Specification from the Object Management Group (OMG).

IDL Objects

PowerDesigner supports modeling for all IDL objects.

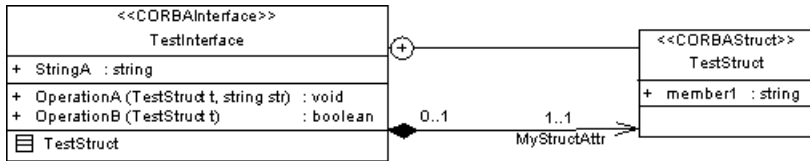
Interfaces

PowerDesigner represents CORBA interfaces as standard classes with the <<CORBAInterface>> stereotype or as standard interfaces (to which a <<CORBAInterface>> stereotype is automatically applied). IDL interface properties are represented as follows:

IDL concept	PowerDesigner implementation
Inheritance between interfaces	Generalization link between classes with <<CORBAInterface>> stereotype
Readonly attribute	Readonly stereotype for attribute
Local interface	isLocal extended attribute set to True
Abstract interface	Abstract property selected in class property sheet
Interface declaration	Use inner link between <<CORBAInterface>> class and other items

Note: You can draw associations between interfaces. Use an inner link to declare an item inner to another.

In the following example, the composition link indicates that an attribute in TestInterface uses the interface TestStruct as data type. The declaration inside the interface is performed with an inner link.



```

interface TestInterface {
    struct TestStruct {
        string member1;
    };

    attribute string StringA;
    attribute TestInterface::TestStruct MyStructAttr;
    void OperationA(inout TestStruct t, in string str);
    boolean OperationB(inout TestStruct t);
};
  
```

Modules

PowerDesigner represents CORBA modules as packages bearing the `<<CORBAModule>>` stereotype.

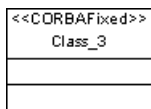
One IDL file is generated per package. The directory hierarchy follows the module hierarchy (and not the package hierarchy), so that only `<<CORBAModule>>` stereotyped packages are mapped onto file directories. `#include` directives are generated based on `<<include>>` stereotyped dependencies between packages and based on class shortcuts defined in a given package

Data Types

PowerDesigner represents CORBA data type as classes bearing the `<<CORBAPrimitive>>` stereotype, and fixed data types as classes bearing the `<<CORBAFixed>>` stereotype.

When you apply the `<<CORBAFixed>>`stereotype, the following extended attributes are automatically added to the class, for which you must define values:

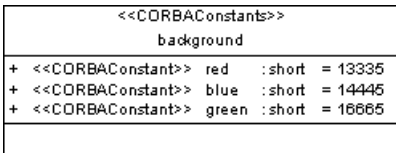
- *digits*: indicates the number of digits of the fixed-point decimal number
- *scale*: indicates the scale of the fixed-point decimal number



```
typedef fixed<8, 3> Class_1;
```

General Constants

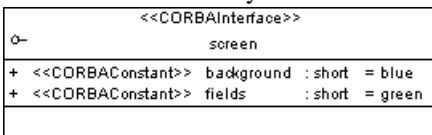
PowerDesigner represents CORBA general constants (values defined independently of any object and likely to be reused) as attributes created in a class, where both the class and the attribute bear the `<<CORBAConstants>>` stereotype.



```
const char red = 13325;
const char blue = 14445;
const long green = 26664;
```

Interface Constants

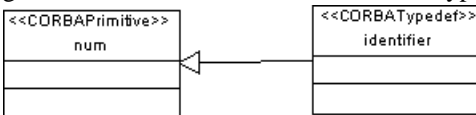
PowerDesigner represents CORBA interface constants (values defined for a specific interface) as attributes bearing the <<CORBAConstants>> stereotype, which are created in the interface where they will be used.



```
interface screen {
  const short background = blue;
  const short fields = green;
}
```

Typedefs

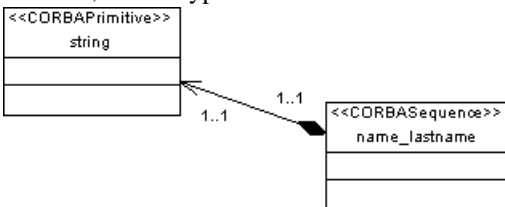
PowerDesigner represents CORBA simple typedefs as classes bearing the <<CORBATypedef>> stereotype. The typedef class should be linked to a class with the <<CORBAPrimitive>>, <<CORBAStruct>> or <<CORBAUnion>> stereotype through a generalization link in order to define the type of data.



```
typedef num identifier;
```

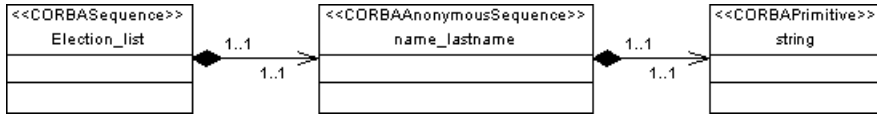
Sequences, Anonymous Sequences, and Sequences in Structs

PowerDesigner represents CORBA sequences as classes bearing the <<CORBASequence>> stereotype. The upper bound of the sequence is specified by its *UpperBound* extended attribute, and its type is defined in another class linked to it by an association.



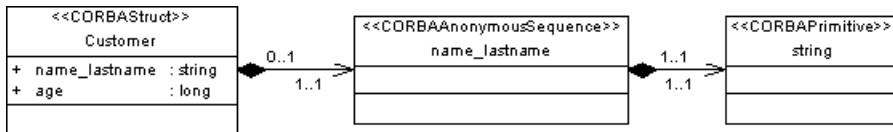
```
typedef sequence< string > name_lastname;
```

To avoid defining directly a type for a sequence to design sequence imbrication, use the `<<CORBAAnonymousSequence>>` stereotype.



```
typedef sequence< sequence< string > > Election_list;
```

To create a sequence in a struct, apply the `<<CORBAAnonymousSequence>>` stereotype to the sequence class.

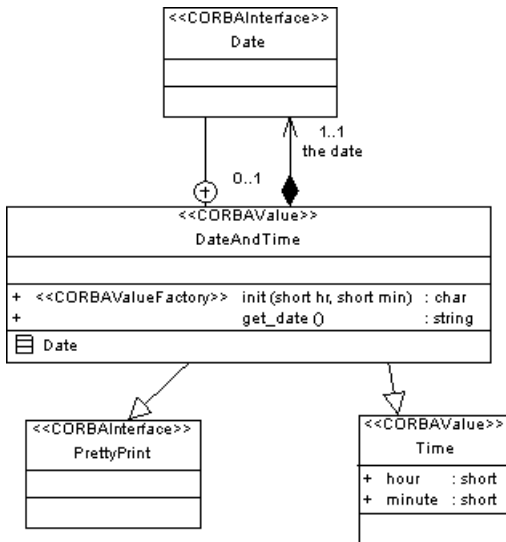


```
struct Customer {
    string name_lastname;
    long age;
    sequence< string > w;
};
```

Valuetypes and Custom Valuetypes

PowerDesigner represents CORBA valuetypes as classes bearing the `<<CORBAValue>>` stereotype and custom valuetypes as classes bearing the `<<CORBACustomValue>>` stereotype. You can further define the valuetype as follows:

- Inheritance between valuetypes has to be designed as a generalization between two valuetype classes
- The interface supported by a value type is the one linked to the valuetype class with a generalization
- Members of a valuetype are linked with a composition to the valuetype
- You can declare an interface inside a valuetype using an inner link
- Boolean extended attribute *Istruncatable* allows you to specify if the valuetype is truncatable or not
- A value type factory operation is represented using an operation with the `<<CORBAValueFactory>>` stereotype



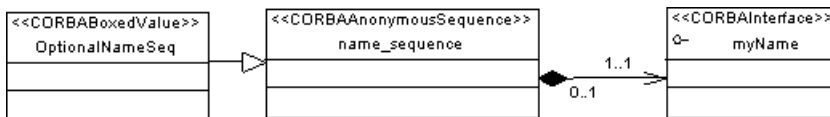
- Code for DateAndTime:

```

valuetype DateAndTime : Time supports PrettyPrint {
public DateAndTime::Date the date;
factory init(in short hr, in short min);
string get_date();
};
  
```

Boxed Values

PowerDesigner represents CORBA boxed values as classes bearing the `<<CORBABoxedValue>>` stereotype. Since the boxed value does not support inheritance, or operations, you should use a class with the `<<CORBAAnonymousSequence>>` stereotype to associate a boxed value with an interface.



```

valuetype OptionalNameSeq sequence< myName >;
  
```

Enums

PowerDesigner represents CORBA enums as classes bearing the `<<CORBAEnum>>` stereotype, and enum elements as attributes bearing the `<<CORBAEnum>>` stereotype.

Structs

PowerDesigner represents CORBA struct types as classes bearing the `<<CORBAStruct>>` stereotype. You can mix attribute types in the struct class.

```

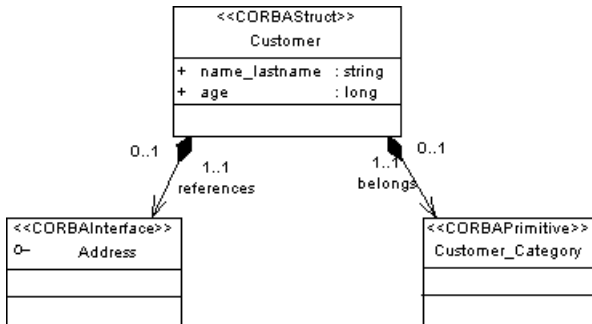
struct Customer {
string name_lastname;
  
```

```

long age;
};

```

You use composition links to define complex struct classes as defined in the following example:

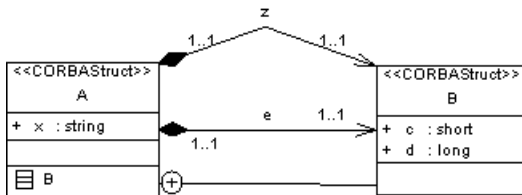


```

struct Customer {
string name_lastname;
long age;
Address references;
Customer_Category belongs;
};

```

To define a struct inside another struct, create two classes with the <<CORBAStruct>> stereotype, add a composition between classes and use the *inner link* feature to declare one class as inner to the other.



```

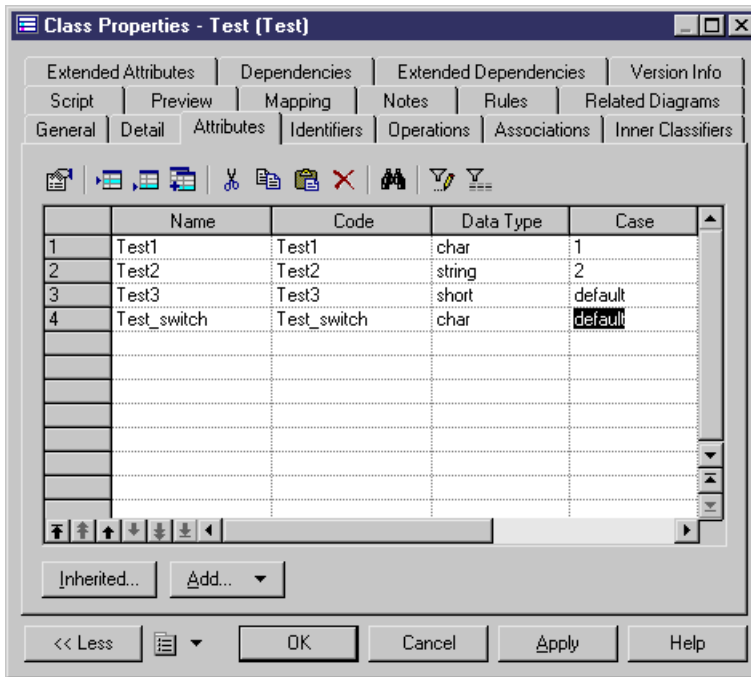
struct A {
struct B {
short c;
long d;
} e, z;
string x;
};

```

Unions

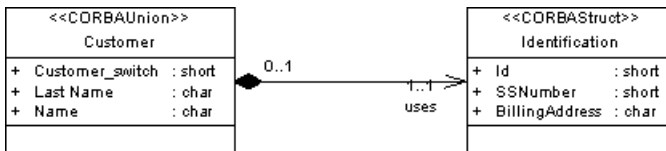
PowerDesigner represents CORBA unions as classes bearing the <<CORBAUnion>> stereotype. Each attribute in a union represents a case, the *Case* extended attribute (in Profile \Attribute\Criteria\IDL union member\Extended Attributes) contains the case default value.

To define the *switch* data type, the discriminant data type for union cases, you have to add an attribute named `<class name>_switch` to the list of union attributes.



```
union Test switch(char) {
  case 1:
    char Test1;
  case 2:
    string Test2;
  default:
    short Test3;
};
```

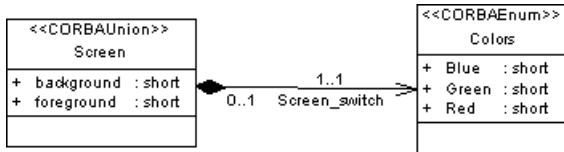
You can use an enum or a struct as a case in the union using a composition association.



```
union Customer switch(short) {
  case XYZ:
    char Last_Name;
  case ZYX:
    char Name;
  default:
    Identification uses;
};
```

The name of the attribute is on the association role and the case is on the association.

You can use an enum or a struct as a switch data type using a composition association. In this situation, the composition role is used as switch attribute for the union.



```

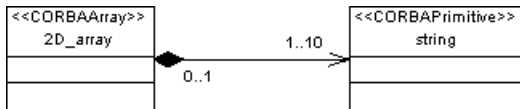
union Screen switch(Colors) {
  case red:
    short background;
  case blue:
    short foreground;
};
    
```

If you do not use the composition role as switch attribute, you still have to define a switch attribute in the Union.

Arrays

PowerDesigner represents CORBA arrays as classes bearing the <<CORBAArray>> stereotype. You must link the array class with another class representing the array type (use the <<CORBAPrimitive>> stereotype to define the type), and define the array dimension in the Dims extended attribute of the array class, a comma-separated list of the dimensions (integer values) of the array. It is used in place of the index<i> qualifiers on the associationEnd specified in the CORBA profile.

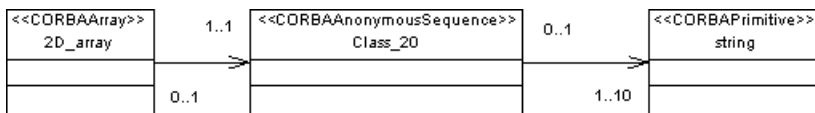
In the following example, array 2D_array uses string data type set to 10:



```

typedef string 2D_array[10];
    
```

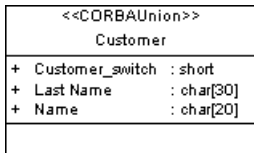
You can use a class with the <<CORBAAnonymousSequence>> stereotype to avoid directly defining a type for an array for a sequence.



```

typedef sequence< string > 2D_array[10];
    
```

To define arrays for a union or a struct attribute, you have to use the multiplicity properties of the attribute.



```
union Customer switch(short) {
  case XYZ:
    char Last_Name[30];
  case ZYX:
    char Name[20];
};
```

You can also use the association multiplicity when the union or struct is linked with another class. The association role and multiplicity will become array attributes of the union or struct classes:

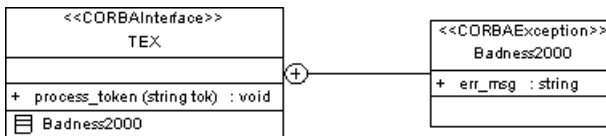


```
struct Client {
  char Name[30];
  short Age[3];
  European_Client European attributes[5];
};
```

Exceptions

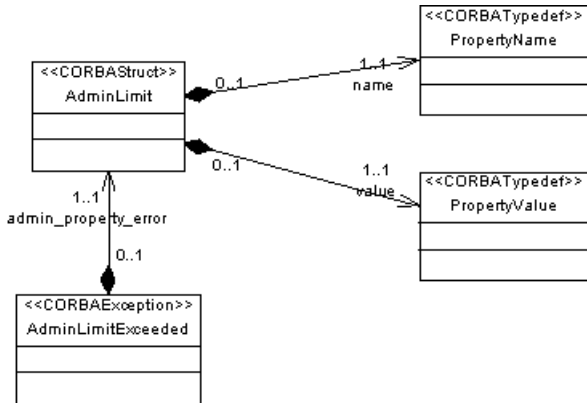
PowerDesigner represents CORBA exceptions as classes bearing the <<CORBAException>> stereotype, where the class is declared as inner to the interface likely to raise the exception.

The list of exceptions is defined in the Exceptions tabbed page in the Implementation page of an operation, with *no enclosing parentheses*.



```
interface TEX {
  exception Badness2000 {
    string err_msg;
  };

  void process_token(in string tok) raises (Badness2000);
};
```



```

struct AdminLimit {
    PropertyName name;
    PropertyValue value;
};

exception AdminLimitExceeded {
    AdminLimit admin_property_error;
};
  
```

Generating for IDL

IDL generation produces files with the .idl extension. One IDL file is generated per package.

A generation log file is also created after generation.

You can select the Check model option in the Options page of the Generation dialog box before starting generation.

For more information on how to design objects for IDL generation, see *Chapter 14, Working with IDL* on page 413.

1. Select **Language > Generate IDL-CORBA 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 items to include in the generation from the Selection page.
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 IDL files are generated in the destination directory.

Reverse Engineering IDL Files

You can reverse engineer IDL files to create classes and interfaces with their attributes and operations in a class diagram.

Reverse-engineering IDL files is subject to the following limitations:

- Pre-processing - When a # symbol is displayed at the beginning of a line, the reverse feature processes both "if" and "else" values during reverse.
- The following tags are not reversed:
 - fixed <8, 4>
 - fixed <8, 2>
 - sequence <short, 4>
 - sequence
 - [wire_marshall(wireVARIANT)]

An example is shown below:

```
struct bar {
    fixed <8, 4> high_scale;
    fixed <8, 2> low_scale;
};

struct bar {
    long val;
    sequence <short, 4> my_shorts;
};

typedef sequence<LinkKind>LinkKinds;
typedef [wire_marshall( wireVARIANT )] struct tagVARIANT VARIANT;
```

Note: 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.

1. Select **Language > Reverse Engineer IDL** to display the Reverse IDL dialog box.
2. Select to reverse engineer files or directories from the Reverse Engineering dropdown listbox.
3. Click the Add button in the Selection page.

A standard Open dialog box is displayed.

4. Select the items or directory you want to reverse engineer, then click the Open button.

Note: You select several files simultaneously using the **Ctrl** or **Shift** keys. You cannot select several directories.

The Reverse IDL dialog box displays the items you selected.

5. Select reverse engineering options in the Options page.

Option	Result of selection
Create symbols	Creates a symbol for each reversed IDL object in the diagram. Otherwise, reversed objects are visible only in the browser
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 Generate check box in its property sheet
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 > Linking and Synchronizing Models > Shortcuts and Replicas > Working with Target Models</i>.</p>

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 > The PowerDesigner Interface > Comparing and Merging Models*.

The classes and interfaces are added to your model. They are visible in the diagram and in the Browser. They are also listed in the Reverse page of the Output window, located in the lower part of the main window.

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.

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.

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 <code><<global>></code> stereotype
Property	Attribute with <code><<property>></code> stereotype
External function	Operation with <code><<external>></code> stereotype
Function	Operation
Event	Operation with <code><<event>></code> stereotype or operation with non-null event name
Structure in object	Inner class with <code><<structure>></code> 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

PowerBuilder	PowerDesigner
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
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.

PowerBuild-er	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* on page 429.

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.

1. Select **Language > Generate PowerBuilder** 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 <i>Chapter 9, Checking an OOM</i> on page 295).

Option	Description
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"> • Package with specified library path (defined in an extended attribute during reverse engineering) is generated in corresponding library from target/application library list • Package at the root of the model without library path is generated in a new library at the same level as target/application library • Child package without library path is generated in parent package • Object at the root of the model is generated in the target/application library
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"> • Classes Defined at the Model Level are generated as source files in the specified directory. • Classes Defined in Packages are generated as source files in sub-directories. <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.

Reverse Engineering PowerBuilder

This section explains how PowerBuilder objects are reverse engineered and how to define reverse engineering options for PowerBuilder.

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.

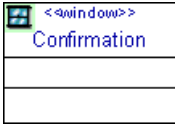
Libraries

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* on page 425.

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* on page 425.

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.

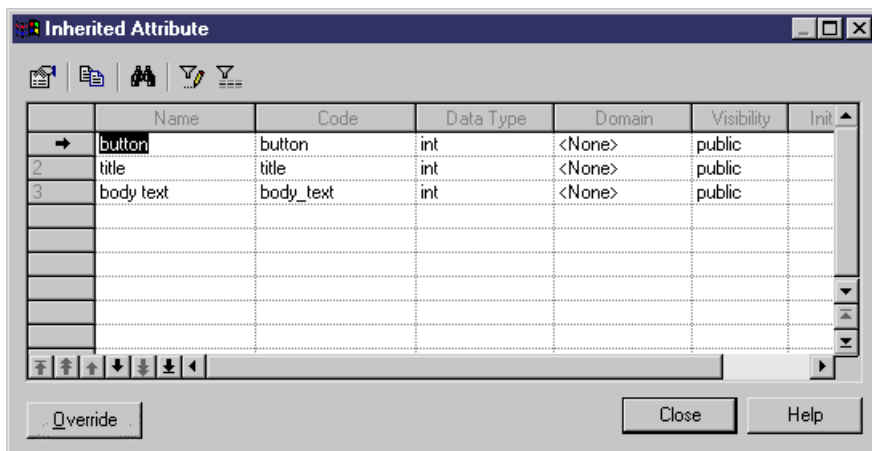
Overriding Attributes

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.

1. Open the property sheet of a child class, and display the attribute tab.
2. Click the Inherited button.

The Inherited Attributes dialog box is displayed. It contains the attributes of the parent class.

3. Select one or several attributes in the list.
4. Click the Override button.



5. Click Close.

The attributes appear in the child class list of attributes. You can modify their initial value in the corresponding column.

6. Click OK.

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.

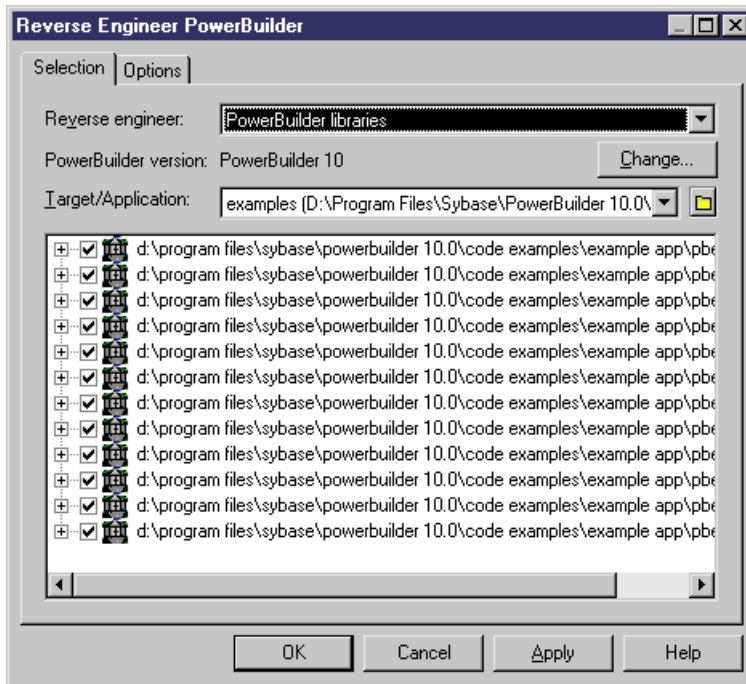
Reverse Engineering Directories

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.

Reverse Engineering PowerBuilder Objects

You can reverse engineer PowerBuilder objects by selecting **Language > Reverse Engineer PowerBuilder**.

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 Generate check box in its property sheet
Create Associations	Creates associations between classes and/or interfaces

Option	Description
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 > Linking and Synchronizing Models > Shortcuts and Replicas > 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 > The PowerDesigner Interface > 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.

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.

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.

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.

PowerDesigner supports the modeling of VB .NET programs including round-trip engineering.

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.

Namespace

You define a VB .NET *namespace* using a package.

Only packages with the Use Parent Namespace check box deselected become VB .NET namespaces.

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.

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

VB .NET assembly properties	PowerDesigner equivalent
Company	AssemblyCompany extended attribute
Copyright	AssemblyCopyright extended attribute
Product	AssemblyProduct extended attribute
Trademark	AssemblyTrademark extended attribute
Version	AssemblyVersion extended attribute
AssemblyInformationalVersion AssemblyFileVersion AssemblyDefaultAlias	Stored in CustomAttributes extended attribute

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.

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
<i>Public</i> (no restriction)	<i>Public</i>
<i>Protected</i> (accessible by derived classes)	<i>Protected</i>
<i>Friend</i> (accessible within the program that contains the declaration of the class)	<i>Friend</i>
<i>Protected Friend</i> (accessible by derived classes and within the program that contains the declaration of the class)	<i>Protected Friend</i>
<i>Private</i> (only accessible by the class)	<i>Private</i>

In the following example, the visibility of class Customer is friend:

```
Friend Class Customer
```


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.

<table border="1"> <tr> <td>Customer {abstract}</td> </tr> <tr> <td>- Name : Char</td> </tr> <tr> <td>- ID : Integer</td> </tr> </table>	Customer {abstract}	- Name : Char	- ID : Integer	<pre>Public MustInherit Class Customer Private Name As Char Private ID As Integer End Class</pre>
Customer {abstract}				
- Name : Char				
- ID : Integer				

- *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.

<table border="1"> <tr> <td>FinalClass</td> </tr> <tr> <td>- At1 : Object</td> </tr> <tr> <td>- At2 : Object</td> </tr> </table>	FinalClass	- At1 : Object	- At2 : Object	<pre>Public NotInheritable Class FinalClass Private At1 As Object Private At2 As Object End Class</pre>
FinalClass				
- At1 : Object				
- At2 : Object				

Note: You design a VB .NET *nested type* using an inner class or interface.

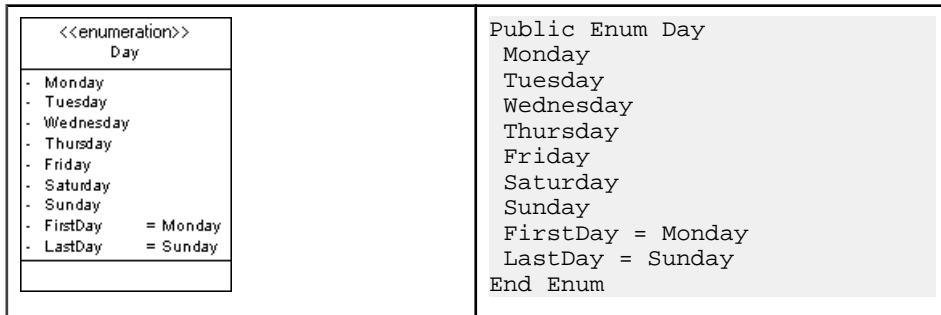
- VB .NET *interfaces* are modeled as standard interfaces. They can contain events, properties, and methods; they do not support variables, constants, and constructors.
- Structures can implement interfaces but do not support inheritance; they can contain events, variables, constants, methods, constructors, and properties. The following structure contains two attributes and a constructor operation:

<table border="1"> <tr> <td>Point</td> </tr> <tr> <td># Y : Integer</td> </tr> <tr> <td># X : Integer</td> </tr> <tr> <td>+ <<Constructor>> New()</td> </tr> </table>	Point	# Y : Integer	# X : Integer	+ <<Constructor>> New()	<pre>... Public Class Point Protected Y As Integer Protected X As Integer Public Sub New() End Sub End Class ...</pre>
Point					
# Y : Integer					
# X : Integer					
+ <<Constructor>> New()					

- 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

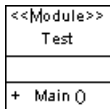
For example:



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.



```

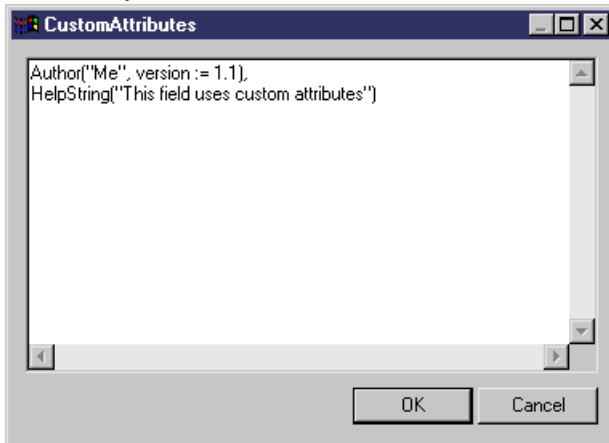
...
Public Module Test
  Public Sub Main()
    Dim val1 As Integer = 0
    Dim val1 As Integer = val1
    val2 = 123
    Dim ref1 As New Class1 ()
    Dim ref1 As Class1 = ref1
    ref2.Value = 123
    Console.WriteLine ("Value: "& val1, "& val2)
    Console.WriteLine ("Refs: "&ref1.Value &", "& ref2.Value)

  End Sub
End Module
...

```

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.



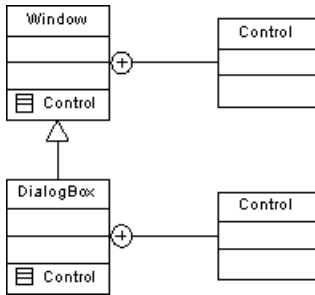
Custom Attributes for Return Types

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.

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
...

```

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
<i>ReadOnly</i> variable	Attribute with <i>Read-only</i> changeability
<i>Const</i> variable	Attribute with <i>Frozen</i> changeability
<i>Shared</i> variable	Attribute with <i>Static</i> property selected
<i>Shadowing</i> variable	Extended attribute <i>Shadowing</i> set to <i>Shadows</i> or <i>Overloads</i>
<i>WithEvents</i> variable	Attribute with <i>withEvents</i> stereotype
<i>Overrides</i> variable	Extended attribute <i>Overrides</i> set to True
<i>New</i> variable	Extended attribute <i>AsNew</i> 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* on page 444 for more details on shadowing

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
+ << <i>Property</i> >>	Caption : String
+ << <i>Setter</i> >>	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
```

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.

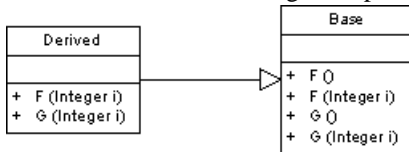
The following table summarizes the different methods supported in PowerDesigner:

VB .NET method	PowerDesigner equivalent
<i>Shadowing</i> or <i>Overloads</i> method	Select <i>Shadows</i> or <i>Overloads</i> from the <i>Shadowing</i> list on the VB.NET tab of the operation property sheet
<i>Shared</i> method	Select the <i>Static</i> check box on the General tab of the operation property sheet

VB .NET method	PowerDesigner equivalent
<i>NotOverridable</i> method	Select the <i>Final</i> check box on the General tab of the operation property sheet
<i>Overridable</i> method	Select the <i>Overridable</i> check box on the VB.NET tab of the operation property sheet
<i>MustOverride</i> method	Select the <i>Abstract</i> check box on the General tab of the operation property sheet
<i>Overrides</i> method	Select the <i>Overrides</i> check box on the VB.NET 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
<i>ByVal</i>	Select <i>In</i> in the Parameter Type box on the parameter property sheet General tab
<i>ByRef</i>	Select <i>In/Out</i> or <i>Out</i> in the Parameter Type box on the parameter property sheet General tab
<i>Optional</i>	Set the <i>Optional</i> extended attribute on the Extended Attributes tab to True

VB .NET modifier	PowerDesigner equivalent
<i>ParamArray</i>	Select the <i>Variable Argument</i> checkbox on the parameter property sheet General 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.

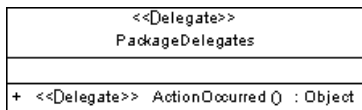
Constructor & Destructor

You design VB .NET *constructors* and *destructors* by clicking the **Add > Default Constructor/Destructor** 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.

Delegate

You can design the following types of VB .NET *delegates*:

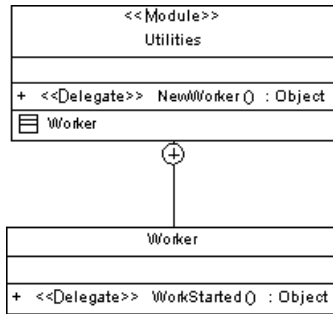
- To create a delegate at the namespace level, create a class with the <<*Delegate*>> stereotype, and add an operation with the <<*Delegate*>> stereotype to this class and define a visibility for this operation. This visibility becomes the visibility of the delegate



```

...
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



```

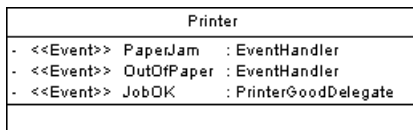
...
Public Module Utilities
  Public Delegate Function NewWorker () As Object
  Public Class Worker
    Public Delegate Function WorkStarted () As Object
  End Class
End Module
...

```

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.

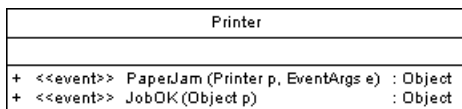


```

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.



```

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.

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
+	Operation_2()	: Object

```
...
Public Function Operation_2() As Object Handles Print
End Function
...
```

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

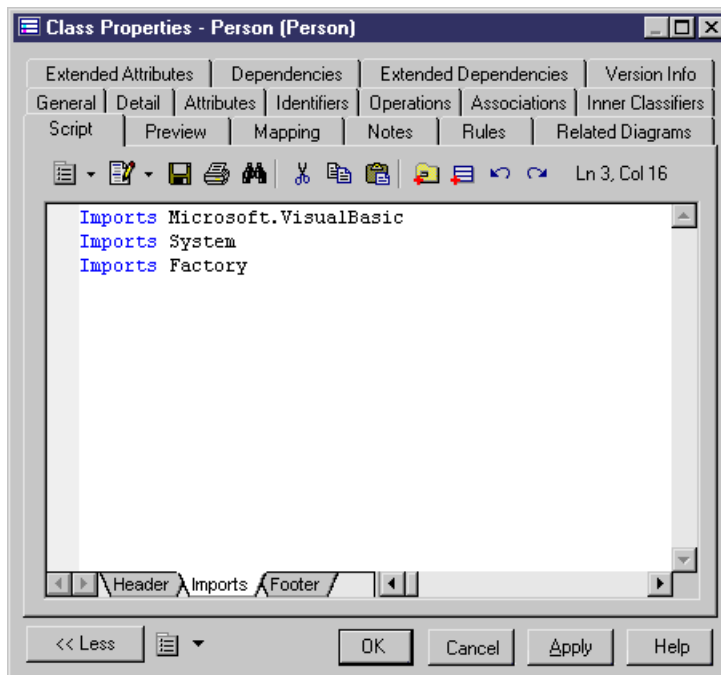
Generating VB.NET Files

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.

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.

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 *Chapter 9, Checking an OOM* on page 295).
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* on page 272).
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 > Templates and 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
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

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.

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.

Selecting VB .NET Reverse Engineering Options

You define the following VB .NET reverse engineering option from the Reverse Engineer VB .NET dialog box:

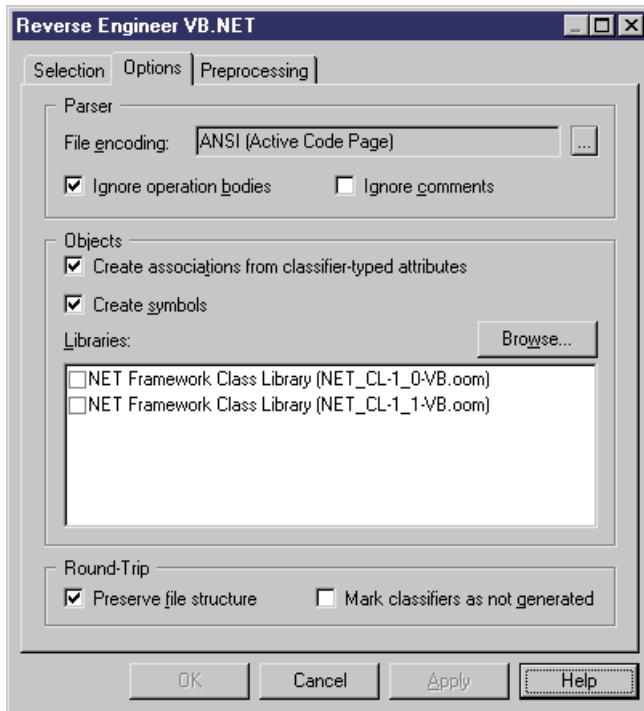
Option	Result of selection
File encoding	Allows you to modify the default file encoding of the files to reverse engineer

Option	Result of selection
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
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 > Linking and Synchronizing Models > Shortcuts and Replicas > 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 Generate check box in its property sheet

Defining VB .NET Reverse Engineering Options

To define VB .NET reverse engineering options:

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.

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.

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.

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.

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* on page 276.

You can use the Set As Default button to save the list of symbols in the registry.

1. Select **Language > Reverse engineering VB .NET.**

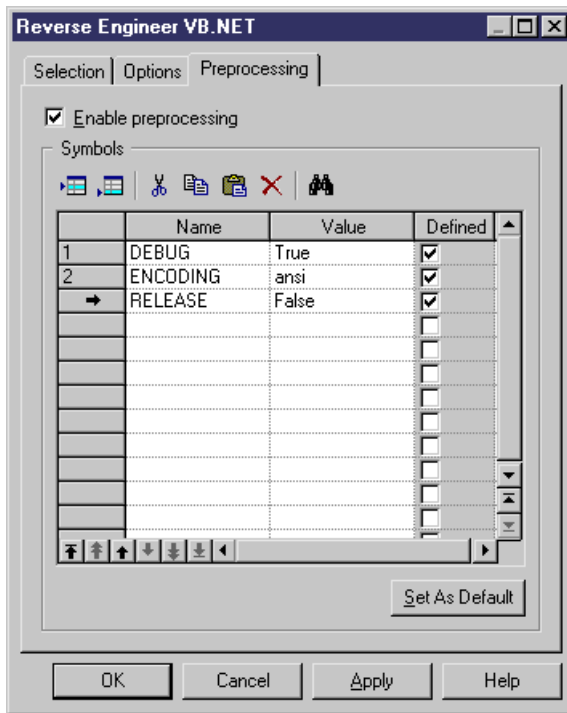
The Reverse Engineering VB .NET dialog box is displayed.

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.

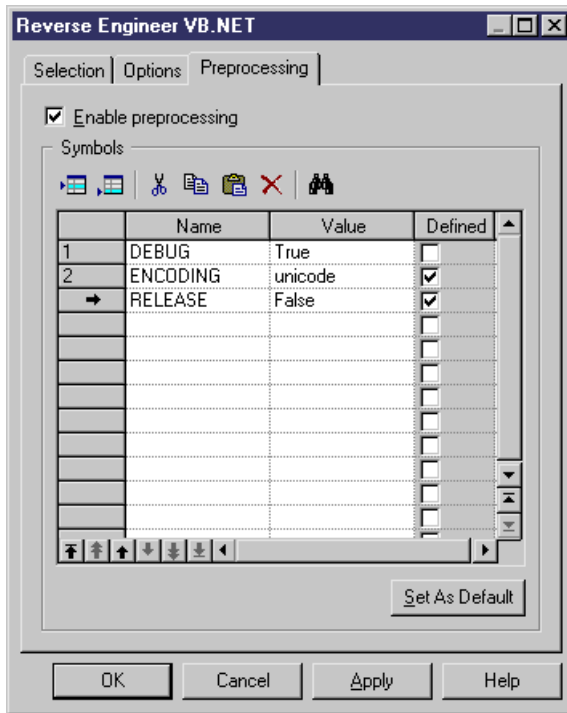
VB .NET Reverse Engineering with Preprocessing

Preprocessing is a reverse engineering option you can enable or disable.

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.

Reverse Engineering VB .NET Files

You can reverse engineer VB .NET files.

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 Engineering 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 > The PowerDesigner Interface > 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.

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.

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 Properties tool beside this box to display the property sheet of the file object, or click the Create tool to create a file object

Property	Description
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* on page 458.

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.

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.

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.

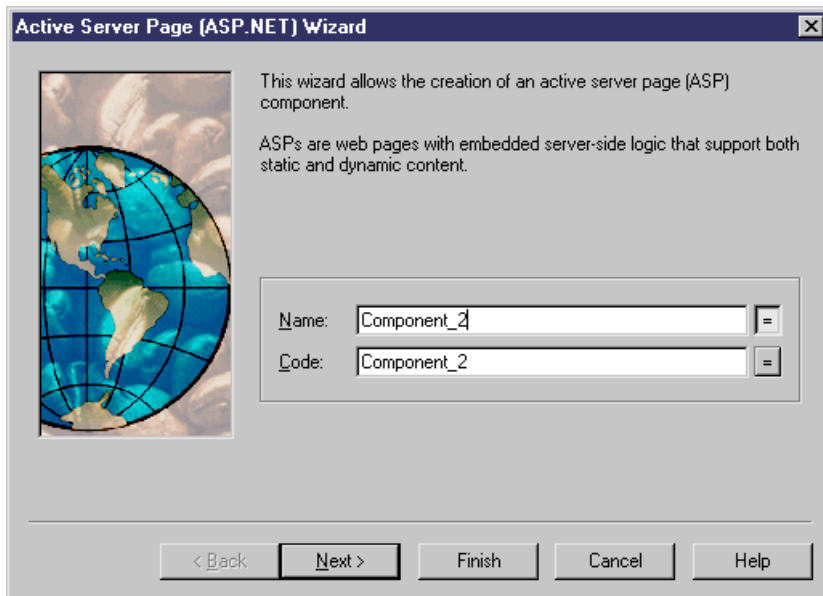
You can also create several ASP.NET of the same type by selecting several file objects at the same time. The wizard will automatically create one ASP.NET per file object: the file objects you have selected in the class diagram become ASP.NET files.

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

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.

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

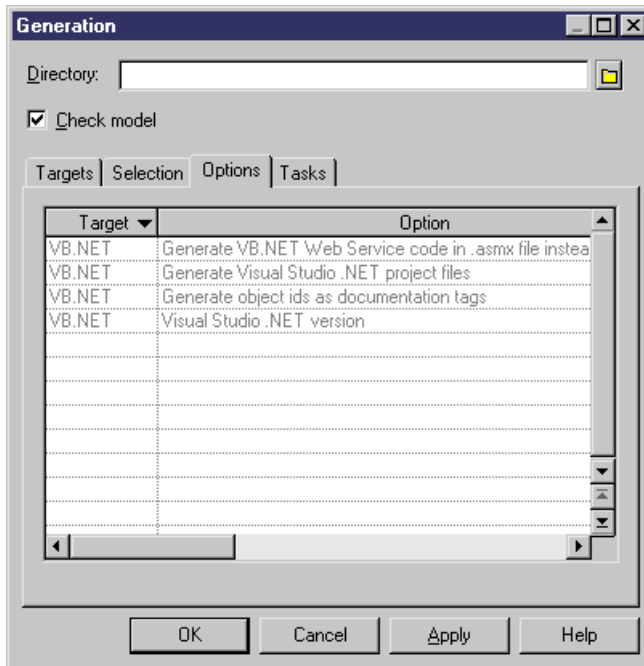
For more information on artifact file objects, see *File Object Properties* on page 226.

- 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

Generating ASP.NET

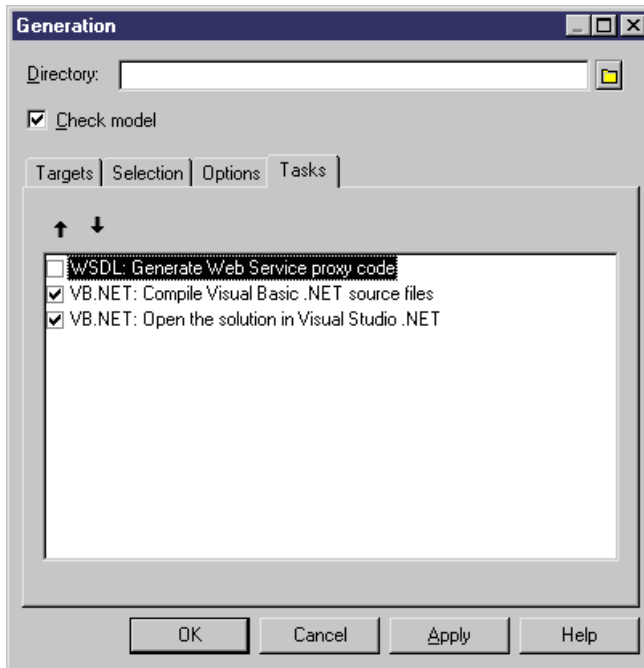
The generation process generates only file objects having the Artifact property selected.

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 in the Options tab.



For more information on the generation options, see *Generating VB.NET Files* on page 449.

6. Click Apply.
7. Click the Tasks tab to display 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* on page 449.

You must beforehand set the environment variables from General Options dialog box (Variables section) in order to activate them in this tab.

For more information on how to set these variables, see *Core Features Guide > The PowerDesigner Interface > Customizing Your Modeling Environment > General Options > Defining 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.

CHAPTER 17 Working with Visual Basic 2005






PowerDesigner provides full support for modeling all aspects of Visual Basic 2005 including round-trip engineering.

Visual Basic 2005 is a high-level programming language for the Microsoft .NET framework.

PowerDesigner can be used as a standalone product and as a plug-in for Visual Studio, allowing you to integrate its enterprise-level modeling capabilities in your standard .NET workflow. For more information, see *Core Features Guide > The PowerDesigner Interface > The PowerDesigner Add-In for Visual Studio*.

This chapter outlines the specifics of PowerDesigner's support for modeling the Visual Basic 2005 language, and should be read in conjunction with the language-neutral chapters in Part One of this book.

In addition to PowerDesigner's standard palettes, the following custom tools are available to help you rapidly develop your class and composite structure diagrams:

Icon	Tool
	Assembly – a collection of Visual Basic 2005 files (see <i>Visual Basic 2005 Assemblies</i> on page 463).
	Custom Attribute – for adding metadata (see <i>Visual Basic 2005 Custom Attributes</i> on page 476).
	Delegate – type-safe reference classes (see <i>Visual Basic 2005 Delegates</i> on page 470).
	Enum – sets of named constants (see <i>Visual Basic 2005 Enums</i> on page 471).
	Struct – lightweight types (see <i>Visual Basic 2005 Structs</i> on page 469).

Visual Basic 2005 Assemblies

An assembly is a collection of Visual Basic 2005 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 VB 2005 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 Visual Basic 2005 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 Visual Basic 2005-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.

Property	Description
Output Type	Specifies the type of application being designed. You can choose between: <ul style="list-style-type: none"> • Class library • Windows Application • Console Application
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 General tab.
Description	Specifies an optional description for the assembly manifest.
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.

Visual Basic 2005 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.

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.

Visual Basic 2005 Namespaces

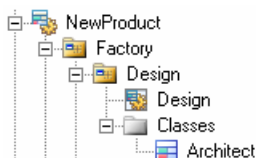
Namespaces restrict the scope of an object's name. Each class or other type must have a unique name within the namespace.

PowerDesigner models namespaces as standard packages with the Use Parent Namespace property set to false. For information about creating and working with packages, see *Packages (OOM)* on page 50.

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
```

This structure, part of the `NewProduct` model, appears in the PowerDesigner Browser as follows:



Classifiers defined directly at the model level fall into the Visual Basic 2005 global namespace.

Visual Basic 2005 Classes

PowerDesigner models Visual Basic 2005 classes as standard UML classes, but with additional properties.

For information about creating and working with classes, see *Classes (OOM)* on page 34.

The following specific classes are also supported in PowerDesigner:

- *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.

<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="text-align: center;">Customer</td></tr> <tr><td style="text-align: center;">{abstract}</td></tr> <tr><td>- Name : Char</td></tr> <tr><td>- ID : Integer</td></tr> </table>	Customer	{abstract}	- Name : Char	- ID : Integer	<pre>Public MustInherit Class Customer Private Name As Char Private ID As Integer End Class</pre>
Customer					
{abstract}					
- Name : Char					
- ID : Integer					

- *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.

<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="text-align: center;">FinalClass</td></tr> <tr><td>- At1 : Object</td></tr> <tr><td>- At2 : Object</td></tr> </table>	FinalClass	- At1 : Object	- At2 : Object	<pre>Public NotInheritable Class FinalClass Private At1 As Object Private At2 As Object End Class</pre>
FinalClass				
- At1 : Object				
- At2 : Object				

Visual Basic 2005 Class Properties

Visual Basic 2005 class property sheets contain all the standard class tabs along with the following properties, located on the VB tab:

Property	Description
Explicit	Specifies the Explicit option directive for the class declaration.
Shadows	Specifies that the class redefines a class defined in a parent class.
Strict	Specifies the Strict option directive for the class declaration.
Compare	Specifies the Compare option directive for the class declaration.

Visual Basic 2005 Interfaces

PowerDesigner models Visual Basic 2005 interfaces as standard UML interfaces, with additional properties.

For information about creating and working with interfaces, see *Interfaces (OOM)* on page 53.

Visual Basic 2005 interfaces can contain events, properties, indexers and methods; they do not support variables, constants, and constructors.

Visual Basic 2005 Interface Properties

Visual Basic 2005 interface property sheets contain all the standard interface tabs along with the following properties, located on the VB tab:

Property	Description
Explicit	Specifies the Explicit option directive for the interface declaration.
Shadows	Specifies that the interface redefines a interface defined in a parent interface .
Strict	Specifies the Strict option directive for the interface declaration.
Compare	Specifies the Compare option directive for the interface declaration.

Visual Basic 2005 Structs

Structs are lightweight types that make fewer demands on the operating system and on memory than conventional classes. PowerDesigner models Visual Basic 2005 structs as classes with a stereotype of <<Structure>>.

For information about creating and working with classes, see *Classes (OOM)* on page 34.

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 and a constructor operation:

<pre> classDiagram class Point { # Y : Integer # X : Integer + <<Constructor>> New() } </pre>	<pre> ... Public Class Point Protected Y As Integer Protected X As Integer Public Sub New() End Sub End Class ... </pre>
---	---

Creating a Struct

You can create a struct in any of the following ways:

- Use the **Struct** tool in the Visual Basic 2005 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

Visual Basic 2005 struct property sheets contain all the standard struct tabs along with the following properties, located on the **VB** tab:

Property	Description
Explicit	Specifies the Explicit option directive for the struct declaration.
Shadows	Specifies that the struct redefines a struct defined in a parent struct.
Strict	Specifies the Strict option directive for the struct declaration.
Compare	Specifies the Compare option directive for the struct declaration.

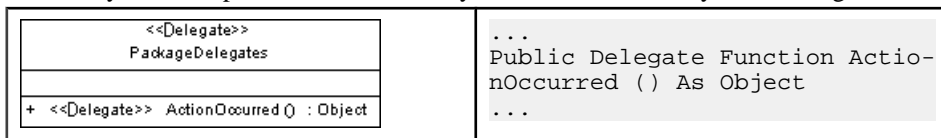
Visual Basic 2005 Delegates

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.

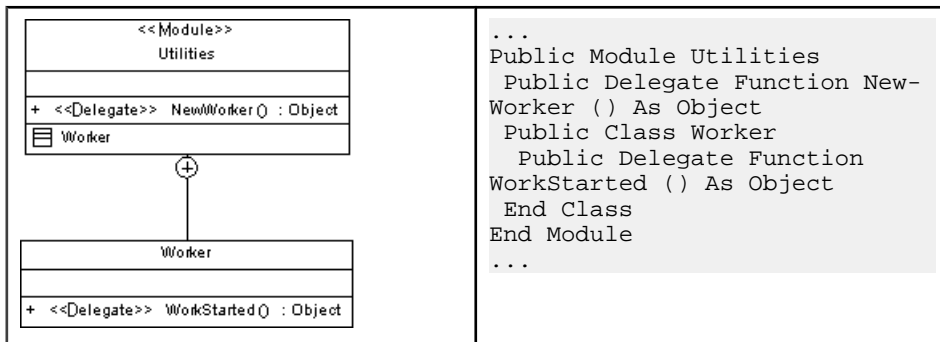
For information about creating and working with classes, see *Classes (OOM)* on page 34.

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



- 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



Creating a Delegate

You can create a delegate in any of the following ways:

- Use the **Delegate** tool in the Visual Basic 2005 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

Visual Basic 2005 delegate property sheets contain all the standard delegate tabs along with the following properties, located on the **VB** tab:

Property	Description
Explicit	Specifies the Explicit option directive for the delegate declaration.
Shadows	Specifies that the delegate redefines a delegate defined in a parent delegate.
Strict	Specifies the Strict option directive for the delegate declaration.
Compare	Specifies the Compare option directive for the delegate declaration.

Visual Basic 2005 Enums

Enums are sets of named constants. PowerDesigner models enums as classes with a stereotype of <<Enum>>.

For information about creating and working with classes, see *Classes (OOM)* on page 34.

<pre> <<enumeration>> 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>
--	---

Creating an Enum

You can create an enum in any of the following ways:

- Use the **Enum** tool in the Visual Basic 2005 Toolbox.
- Select **Model > Enum Objects** 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 **New > Enum**.

Enum Properties

Visual Basic 2005 enum property sheets contain all the standard enum tabs along with the following properties, located on the **VB** tab:

Property	Description
Base Integral Type	Specifies the base integral type for the enum.
Compare	Specifies the Compare option directive for the enum declaration.
Explicit	Specifies the Explicit option directive for the enum declaration.
Shadows	Specifies that the enum redefines a enum defined in a parent enum .
Strict	Specifies the Strict option directive for the enum declaration.

Visual Basic 2005 Fields, Events, and Properties

PowerDesigner models Visual Basic 2005 fields, events, and properties as standard UML attributes with additional properties.

Creating a Field, Event, or Property

To create a field, event, 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.

For general information about creating and working with attributes, see *Attributes (OOM)* on page 65.

Working with Events

Events are modeled as attributes with a stereotype of <<Event>>, and with one or two linked operations representing the add and/or remove handlers. You declare events within classes, structures, modules, and interfaces using the Event keyword, as in the following example:

```
Event AnEvent (ByVal EventNumber As Integer)
```

An event is like a message announcing that something important has occurred. The act of broadcasting the message is called raising the event.

Events must be raised within the scope where they are declared. For example, a derived class cannot raise events inherited from a base class.

Any object capable of raising an event is an *event sender*, also known as an *event source*. Forms, controls, and user-defined objects are examples of event senders.

Event handlers are procedures that are called when a corresponding event occurs. You can use any valid subroutine as an event handler. You cannot use a function as an event handler, however, because it cannot return a value to the event source.

Visual Basic uses a standard naming convention for event handlers that combines the name of the event sender, an underscore, and the name of the event. For example, the click event of a button named `button1` would be named `Sub button1_Click`. It is recommended that you use this naming convention when defining event handlers for your own events, but it is not required; you can use any valid subroutine name.

Before an event handler becomes usable, you must first associate it with an event by using either the `Handles` or `AddHandler` statement.

The `WithEvents` statement and `Handles` clause provide a declarative way of specifying event handlers. Events raised by an object declared with `WithEvents` can be handled by any subroutine with a `Handles` clause that names this event. Although the `Handles` clause is the standard way of associating an event with an event handler, it is limited to associating events with event handlers at compile time.

The `AddHandler` and `RemoveHandler` statements are more flexible than the `Handles` clause. They allow you to dynamically connect and disconnect the events with one or more event handlers at run time, and they do not require you to declare object variables using `WithEvents`.

The following example shows the `Button` class, which contains three events:

Printer	
- <<Event>> PaperJam	: EventHandler
- <<Event>> OutOfPaper	: EventHandler
- <<Event>> JobOK	: PrinterGoodDelegate

```
Public Class Printer
    Public PaperJam As EventHandler
    Public OutOfPaper As EventHandler
    Public JobOK As PrinterGoodDe-
    delegate
End Class
```

The following example shows the `Printer` class, which contains an event handler:

Printer	
+ <<event>> Print() : Object	
+ Operation_2() : Object	


```

...
Public Function Operation_2() As
Object Handles Print
    End Function
...

```

Working with Properties

Properties are modeled as attributes with a stereotype of <<Property>>, and with one or two linked operations representing the get and/or set accessors.

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.

The Get and Set accessors in Visual Basic 2005 can now have different accessibility settings, as long as Set is more restrictive than Get.

It is possible to add a Property for an existing attribute to access it. The attribute will have the <<PropertyImplementation>> stereotype. The created Property will use the same code as the implemented attribute but starting with an underscore (_) character. By default, the Property will have a public visibility and will not be persistent.

Field, Event, and Property Properties

Visual Basic 2005 field, event, and property property sheets contain all the standard attribute tabs along with the VB, 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).
As New	Specifies that the attribute is created by new object instance.
Shadowing	Specifies the form of shadowing. You can choose between: <ul style="list-style-type: none"> Shadows Overloads Overrides
Default	[properties only] Specifies whether the property is a default.
Overriding	[properties only] Specifies the form of overriding available. You can choose between: <ul style="list-style-type: none"> Overridable NotOverridable MustOverride

Property	Description
Property Parameters	[properties only] Specifies the parameters of the property.

Visual Basic 2005 Methods

PowerDesigner models Visual Basic 2005 methods as operations.

For information about creating and working with operations, see *Operations (OOM)* on page 78.

Method Properties

Method property sheets contain all the standard operation tabs along with the VB 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).
Overridable	Specifies that the method can be overridden.
Overrides	Specifies that the method overrides another.
Handles	Specifies the name of the event that the method handles.
Shadows	Specifies the form of shadowing. You can choose between: <ul style="list-style-type: none"> • Shadows • Overloads
Library Name	Specifies the library DLL name.
Alias Name	Specifies the alias name.
Character Set	Specifies the character set of the external method.

Constructors and Destructors

You design VB .NET *constructors* and *destructors* by clicking the **Add Default Constructor/ Destructor** 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.

Visual Basic 2005 Inheritance and Implementation

PowerDesigner models Visual Basic 2005 inheritance links as standard UML generalizations or realizations.

PowerDesigner models Visual Basic 2005 inheritance links between types as standard UML generalizations (see *Generalizations (OOM)* on page 98).

PowerDesigner models Visual Basic 2005 implementation links between types and interfaces as standard UML realizations (see *Realizations (OOM)* on page 105).

Visual Basic 2005 Custom Attributes

PowerDesigner provides full support for Visual Basic 2005 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)* on page 111.

Generating Visual Basic 2005 Files

You generate Visual Basic 2005 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.

The following PowerDesigner variables are used in the generation of Visual Basic 2005 source files:

Variable	Description
CSC	Visual Basic 2005 compiler full path. For example, <code>C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705\csc.exe</code>
WSDL	Web Service proxy generator full path. For example, <code>C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Bin\wsdl.exe</code>

To review or edit these variables, select **Tools > General Options** and click the **Variables** category.

1. Select **Language > Generate Visual Basic 2005 Code** to open the Visual Basic 2005 Generation dialog.
2. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see *Chapter 9, Checking an OOM* on page 295).
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* on page 272).
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"> • Visibility • Type
Class members type sort	Specifies the order by which class members are sorted in terms of their type: <ul style="list-style-type: none"> • Methods – Properties - Fields • Properties – Methods - Fields • Fields – Properties - Methods
Class members visibility sort	Specifies the order by which class members are sorted in terms of their visibility: <ul style="list-style-type: none"> • Public - Private • Private – Public • None
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*.

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 > Templates and Generated Files (Profile)*.

7. [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.

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.

Reverse Engineering Visual Basic 2005 Code

You can reverse engineer Visual Basic 2005 files into an OOM.

1. Select **Language > Reverse Engineer Visual Basic** to open the Reverse Engineer Visual Basic dialog box.
2. Select what form of code you want to reverse engineer. You can choose between:
 - Visual Basic files (.vb)
 - Visual Basic directories
 - Visual Basic projects (.vbproj)
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. For more information, see *Visual Basic Reverse Engineer dialog Options tab* on page 479.
5. [optional] Click the Preprocessing tab and set any appropriate preprocessing symbols. For more information, see *Visual Basic reverse engineering preprocessing directives* on page 480.
6. 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 > The PowerDesigner Interface > 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.

Visual Basic Reverse Engineer Dialog Options Tab

The following options are available on this tab:

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.</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 > Linking and Synchronizing Models > Shortcuts and Replicas > 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	Specifies that reversed classifiers (classes and interfaces) will not be generated from the model. To generate the classifier, you must select the Generate check box in its property sheet

Visual Basic Reverse Engineering Preprocessing Directives

Visual Basic 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 Visual Basic 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.

Visual Basic 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.

Defining a Visual Basic Preprocessing Symbol

You can define Visual Basic preprocessing symbols and values in the preprocessing tab of the reverse engineering dialog box.

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* on page 276.

You can use the Set As Default button to save the list of symbols in the registry.

1. Select **Language > Reverse engineering Visual Basic.**

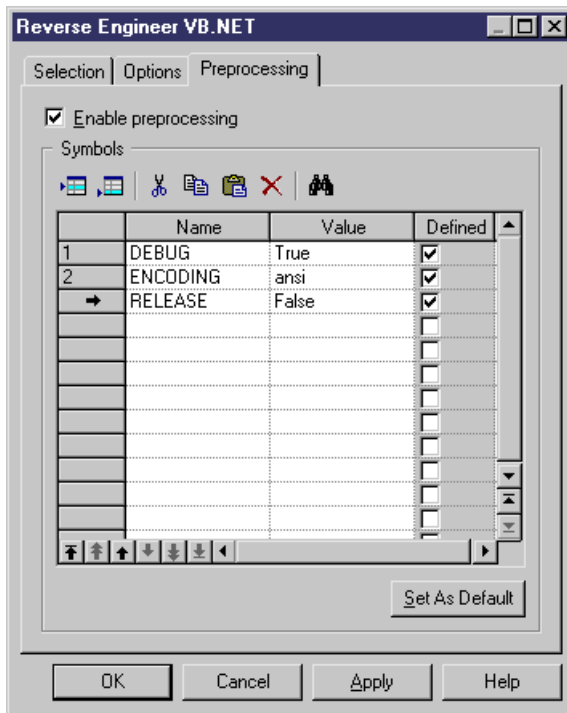
The Reverse Engineering Visual Basic dialog box is displayed.

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.

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.

CHAPTER 18 Working with C#

PowerDesigner supports the modeling of C# programs including round-trip engineering.

Inheritance & Implementation

You design C# *inheritance* using a *generalization* link between classes.

You design C# *implementation* using a *realization* link between a class and an interface.

Namespace

You define a C# *namespace* using a package.

Only packages with the Use Parent Namespace check box deselected become C# namespaces.

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
    {
    }
}
```

Classifiers defined directly at the model level fall into the C# global namespace.

Project

You can reverse engineer C# projects when you select C# projects from the Reverse Engineer list in the Reverse Engineer C# dialog box.

Make sure you reverse engineer each project into a separate model.

Assembly properties are reverse engineered as follow:

C# assembly properties	PowerDesigner equivalent
Title	Name of the model
Description	Description of the model
Configuration	AssemblyConfiguration extended attribute

C# assembly properties	PowerDesigner equivalent
Company	AssemblyCompany extended attribute
Copyright	AssemblyCopyright extended attribute
Product	AssemblyProduct extended attribute
Trademark	AssemblyTrademark extended attribute
Version	AssemblyVersion extended attribute
Culture	AssemblyCulture extended attribute
AssemblyVersionAttribute System.CLSCompliant AssemblyFlagsAttribute	Stored in CustomAttributes extended attribute

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.

Accessibility

To define *accessibility* for a class, an interface, or a method, you have to use the *visibility* property in PowerDesigner.

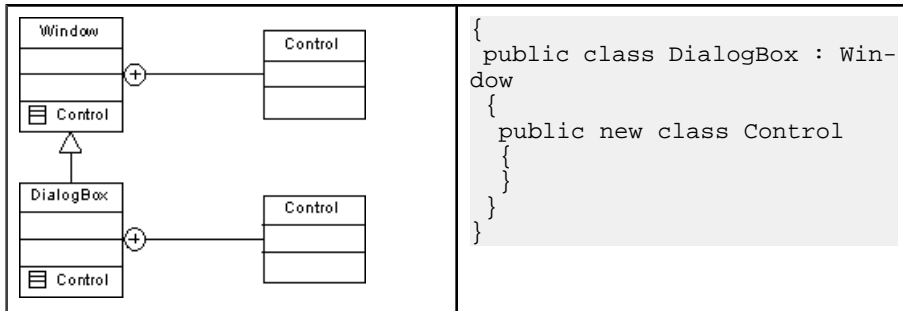
The following accessibility attributes are supported in PowerDesigner:

C# accessibility	PowerDesigner visibility
<i>Public</i> (no restriction)	<i>Public</i>
<i>Private</i> (access limited to the containing type)	<i>Private</i>
<i>Protected</i> (access limited to the containing class or types derived from the containing class)	<i>Protected</i>
<i>Internal</i> (access limited to current program)	<i>Internal</i>
<i>Protected internal</i> (access limited to this program or types derived from the containing class)	<i>Protected internal</i>

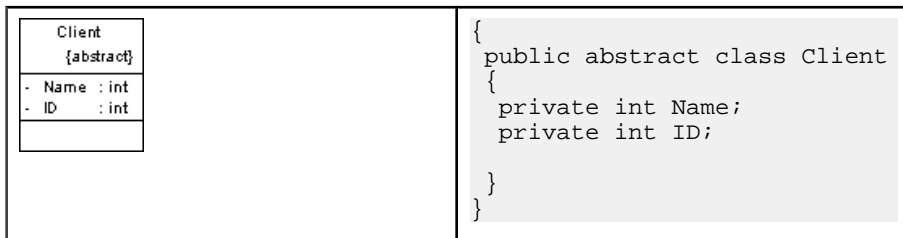
Classes, Interfaces, Structs, and Enumerations

You design a C# *class* using a class in PowerDesigner. *Structures* are classes with the <<structure>> stereotype, and *enumerations* are classes with the <<enumeration>> stereotype.

- C# classes can contain events, variables, constants, methods, constructors, properties, and indexers. The following specific classes are also supported in PowerDesigner:
 - *Newclass* is used to declare a member with the same name or signature as an inherited member. To design a new class, you have to set the class *newextended* 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 new extended attribute to True, in the Control class inner to DialogBox:



- *Abstract* 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.



- *Sealed* 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> SealedClass - A1 : int - A2 : int </pre>	<pre> { public sealed class Sealed- Class { private int A1; private int A2; } } </pre>
--	--

Note: You design a C# *nested type* using an inner class or interface.

- C# *interfaces* are modeled as standard interfaces. They can contain events, properties, indexers 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 struct contains two attributes:

<pre> Point # Y : Integer # X : Integer + <<Constructor>> New() </pre>	<pre> { public struct Point { public int New() { return 0; } private int x; private int y; } } </pre>
--	---

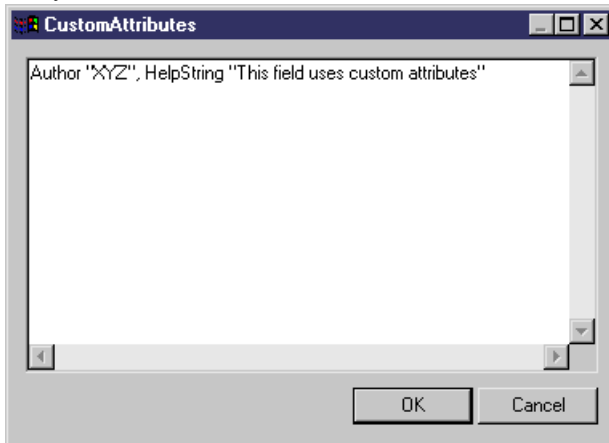
- Enumeration class attributes are used as enumeration values. The following items must be set:
 - *Data Type* - using the *EnumDataType* extended attribute of the enumeration (for example Byte, Short, or Long)
 - *Initial Expression* - using the *Initial Value* field of an enum attribute

For example:

<pre> <<enumeration>> Color - Red : Object - Blue : Object - Green : Object - Max : Object = Blue </pre>	<pre> { public enum Color : colors { Red, Blue, Green, Max = Blue } } </pre>
--	--

Custom Attributes

To define *custom attributes* for a class, an interface, a variable, a parameter or a method, you have to use the *CustomAttributes* extended attribute in PowerDesigner. You can use the CustomAttributes input box to type all the custom attributes you wish to add using the correct C# syntax.



Fields

You design a C# *field* using an attribute in PowerDesigner.

The following table summarizes the C# field modifiers supported in PowerDesigner:

C# field modifiers	PowerDesigner equivalent
<i>Static</i> field	<i>Static</i> property selected
<i>New</i> field	<i>New</i> extended attribute set to True
<i>Unsafe</i> field	<i>Unsafe</i> extended attribute set to True
<i>Public</i> field	<i>Public</i> visibility
<i>Protected</i> field	<i>Protected</i> visibility
<i>Internal</i> field	<i>Internal</i> visibility
<i>Private</i> field	<i>Private</i> visibility
<i>ReadOnly</i> field	<i>Read-only</i> changeability
<i>Volatile</i> field	<i>Volatile</i> property selected

- *Data Type*: You define the data type of a field using the attribute Data Type property
- *Initial Value*: You define the initial value of a field using the attribute Initial Value property

Property

To design a C# *property* you have to design an attribute with the <<Property>> stereotype.

When you do so, 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 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 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;
    }
}
}

```

The following table lists the different property modifiers supported in PowerDesigner:

C# property modifiers	PowerDesigner equivalent
<i>Abstract</i>	<i>Abstract extended attribute set to True</i>
<i>Extern</i>	<i>Extern extended attribute set to True</i>
<i>Override</i>	<i>Override extended attribute set to True</i>
<i>Sealed</i>	<i>Sealed extended attribute set to True</i>
<i>Unsafe</i>	<i>Unsafe extended attribute set to True</i>
<i>Virtual</i>	<i>Virtual extended attribute set to True</i>

Indexer

You design a C# *indexer* using an attribute with the <<*Indexer*>> stereotype. Another attribute with the <<*IndexerImplementation*>> 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 both getter and setter operations, the attribute no longer has the <<*Indexer*>> stereotype.

When you define a <<*Indexer*>> attribute, the attribute changeability and the getter/setter operations are tightly related as explained in the following table:

Operations	Indexer attribute changeability
If you keep both getter and setter operations	Indexer is Changeable
If you remove the setter operation of a changeable property	Indexer becomes Read-only
If you remove the getter operation of a changeable property	Indexer becomes Write-only

On the other hand, if you modify the indexer changeability, operations will reflect this change, for example, if you turn a changeable indexer into a read-only indexer, the setter operation is automatically removed.

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;
```

The following table lists the different indexer modifiers supported in PowerDesigner:

C# property modifier	PowerDesigner equivalent
<i>New</i>	<i>New extended attribute set to True</i>
<i>Unsafe</i>	<i>Unsafe extended attribute set to True</i>
<i>Virtual</i>	<i>Virtual extended attribute set to True</i>

C# property modifier	PowerDesigner equivalent
<i>Override</i>	<i>Override extended attribute set to True</i>
<i>Extern</i>	<i>Extern extended attribute set to True</i>
<i>Abstract</i>	<i>Abstract extended attribute set to True</i>
<i>Sealed</i>	<i>Sealed extended attribute set to True</i>

Parameters

Type a value in the value box of the *Indexer parameters* extended attribute to specify which value of the property attribute is to be used as parameter.

Method

You design a C# *method* using an operation.

The following table summarizes the different methods supported in PowerDesigner:

C# method	PowerDesigner equivalent
<i>Extern</i>	Select the <i>Extern</i> check box in the C# tab of the operation property sheet
<i>New</i>	Select the <i>New</i> check box in the C# tab of the operation property sheet
<i>Virtual</i>	Select the <i>Virtual</i> check box in the C# tab of the operation property sheet
<i>Override</i>	Select the <i>Override</i> check box in the C# tab of the operation property sheet
<i>Unsafe</i>	Select the <i>Unsafe</i> check box in the C# tab of the operation property sheet
<i>Explicit</i>	Select the <i>Explicit</i> check box in the C# tab of the operation property sheet
<i>Static</i>	Select the <i>Static</i> check box in the General tab of the operation property sheet
<i>Sealed</i>	Select the <i>Final</i> check box in the General tab of the operation property sheet
<i>Abstract</i>	Select the <i>Abstract</i> check box in the General tab of the operation property sheet

New

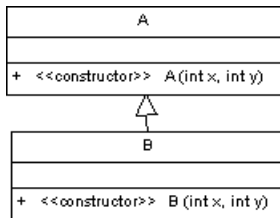
When a class inherits from another class and contains methods with identical signature as in the parent class, the *New* check box is automatically selected to make the child method prevail over the parent method.

BaseInitializer and ThisInitializer for Constructors

The *Base Initializer* property in the C# tab of the operation property sheet is used to create an instance constructor initializer of the form `base`. It causes an instance constructor from the base class to be invoked.

The *This Initializer* property in the C# tab of the operation property sheet is also used to create an instance constructor initializer, it causes an instance constructor from the class itself to be invoked.

In the following example, class B inherits from class A. You define a Base Initializer extended attribute in class B constructor, this extended attribute will be used to initialize class A constructor:



```
internal class B : A
{
    public B(int x, int y) : base(x + y, x - y)
    {}
}
```

Method Parameters

You define C# method parameters using operation parameters.

You can define the following parameter modifiers in PowerDesigner:

C# modifier	PowerDesigner equivalent
<i>[none]</i>	Select <i>In</i> in the Parameter Type box on the parameter property sheet General tab
<i>ref</i>	Select <i>In/Out</i> in the Parameter Type box on the parameter property sheet General tab
<i>out</i>	Select <i>Out</i> in the Parameter Type box on the parameter property sheet General tab

C# modifier	PowerDesigner equivalent
...	Select the <i>Variable Argument</i> checkbox on the parameter property sheet General 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.

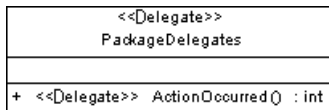
Constructor & Destructor

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, but you can still remove them from the list.

Delegate

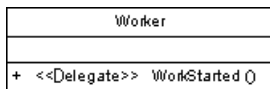
You can design the following types of C# *delegates*:

- 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 int ActionOccurred();
}
```

- To create a delegate in a class, or structure, you just have to create an operation with the <<Delegate>> stereotype. In the following example, Class worker contains an internal delegate designed as an operation with the <<Delegate>> stereotype



```
{
public class Worker
{
public delegate WorkStarted();
}
```

```
}  
}
```

Event

You design a C# *event* using an attribute with the <<Event>> stereotype.

Button		
-	<<Event>> Click	: Object
-	<<Event>> DoubleClick	: Object
-	<<Event>> RightClick	: Object

Operator Method

You design a C# *operator* using an operation with the <<Operator>> stereotype. Make sure the <<Operator>> operation has the 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.

IntVector	
+	<<Operator>> Oper1 (+, -) : int

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:

<<struct>> Digit		
-	value	: byte
+	<<constructor>> Digit (byte value)	
+	<<ConversionOperator>> byte (Digit d)	: byte
+	<<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;
}

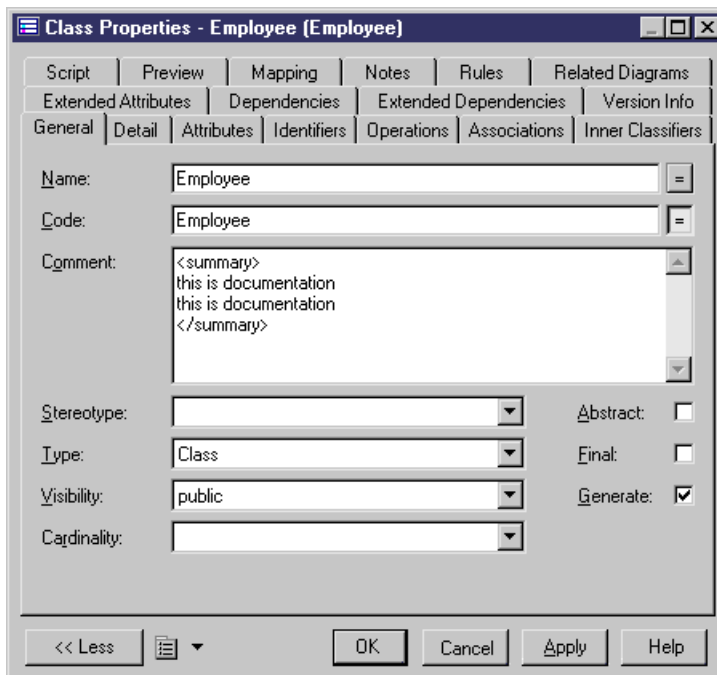
public static explicit operator Digit(byte b)
{
    return new Digit(b);
}
private byte value;
}
```

Documentation Tags

To generate documentation tags, you should use the Comment box in property sheets. You can use any documentation tag before and after documentation, for example you can type `<summary>` and `</summary>`.

If you add a tag, the documentation is preceded by `///` in the generated code. During reverse engineering, the `/` signs used to identify documentation are removed and the start and end documentation tags are automatically added.

In the following example, you add a comment to class `Employee`:



The following code is generated:

```

/// <summary>
/// this is documentation
/// this is documentation
/// </summary>

```

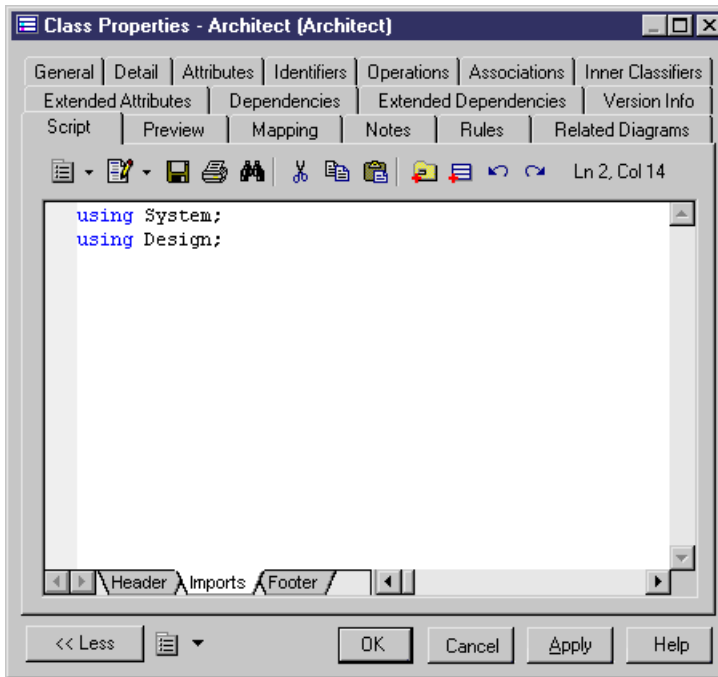
Generating C# Files

You generate C# 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.

During C# generation, each main object (class, interface, and so on) generates a source file with the `.cs` extension. Inner classifiers are generated in the source of the container classifier.

The `using` directive can appear at the beginning of the script of each generated file.

In PowerDesigner, you can define a `using` directive in the `Script\Imports` tab of the property sheet of a main object. You can type the `using` statement or use the `Import Folder` or `Import Classifier` tools in the `Imports` tab.



The following PowerDesigner variables are used in the generation of C# 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 > General Options** and click the **Variables** category.

1. Select **Language > Generate C# Code** to open the C# Generation dialog.
2. Enter a directory in which to generate the files, and specify whether you want to perform a model check (see *Chapter 9, Checking an OOM* on page 295).
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* on page 272).
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 Web Service C# code in .ASMX file	Generates the C# 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*.

- [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 > Templates and Generated Files (Profile)*.

- [optional] Click the **Tasks** tab and specify any appropriate generation tasks to perform:

Task	Description
Compile C# source files	Compiles the source files
Generate Web service proxy code (WSDL)	Generates the proxy class
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.

Reverse Engineering C#

You can reverse engineer C# 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.

Selecting C# Reverse Engineering Options

You define the following C# reverse engineering option from the Reverse Engineer C# dialog box:

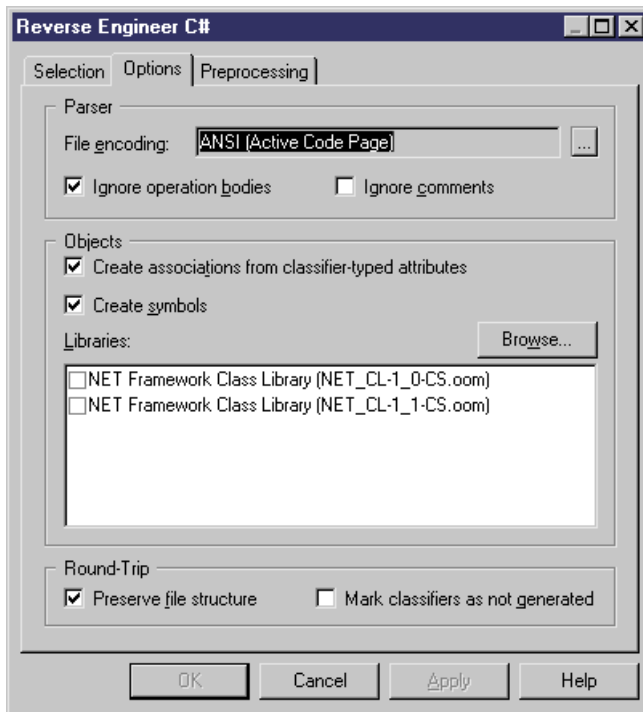
Option	Result of selection
File encoding	Allows you to modify the default file encoding of the files to reverse engineer
Ignore operation body	Reverses classes without including the body of the code
Ignore comments	Reverses classes without including code comments
Create Associations from classifier-typed 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
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 > Linking and Synchronizing Models > Shortcuts and Replicas > Working with Target Models</i>).</p>

Option	Result of selection
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 Generate check box in its property sheet

Defining C# Reverse Engineering Options

To define C# reverse engineering options:

1. Select **Language > Reverse Engineer C#**.
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.

C# Reverse Engineering Preprocessing

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 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 C#, symbols have no value, they can 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.

C# Supported Preprocessing Directives

The following directives are supported during preprocessing:

Directive	Description
<code>#define</code>	Defines a symbol
<code>#undef</code>	Removes a previous definition of the symbol
<code>#if</code>	Evaluates a condition, if the condition is true, the statement following the condition is kept otherwise it is ignored
<code>#elif</code>	Used with the <code>#if</code> directive, if the previous <code>#if</code> test fails, <code>#elif</code> includes or exclude source code, depending on the resulting value of its own expression or identifier
<code>#endif</code>	Closes the <code>#if</code> conditional block of code
<code>#warning</code>	Displays a warning message if the condition is true
<code>#error</code>	Displays an error message if the condition is true

Note: `#region`, `#endregion`, and `#line` directives are removed from source code.

Defining a C# Preprocessing Symbol

You can define C# preprocessing symbols in the preprocessing tab of the reverse engineering dialog box.

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* on page 276.

You can use the Set As Default button to save the list of symbols in the registry.

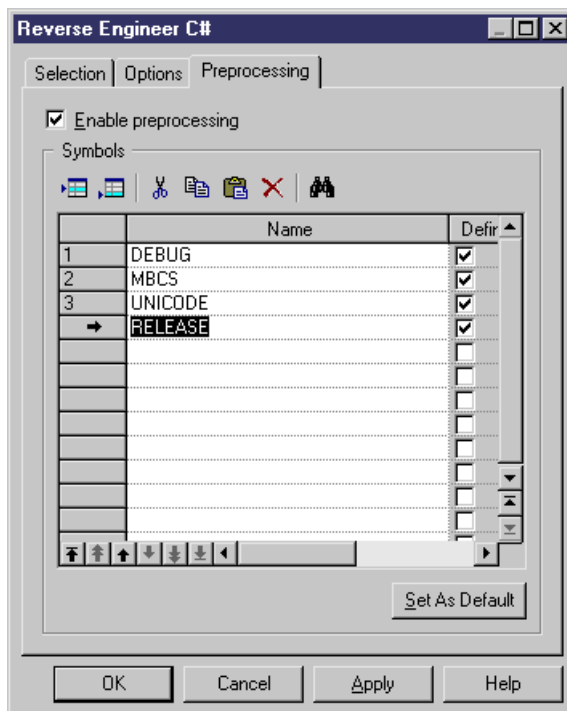
1. Select **Language > Reverse engineering C#.**

The Reverse Engineering C# 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.

The Defined check box is automatically selected for each symbol to indicate that the symbol will be taken into account during preprocessing.



- Click Apply.

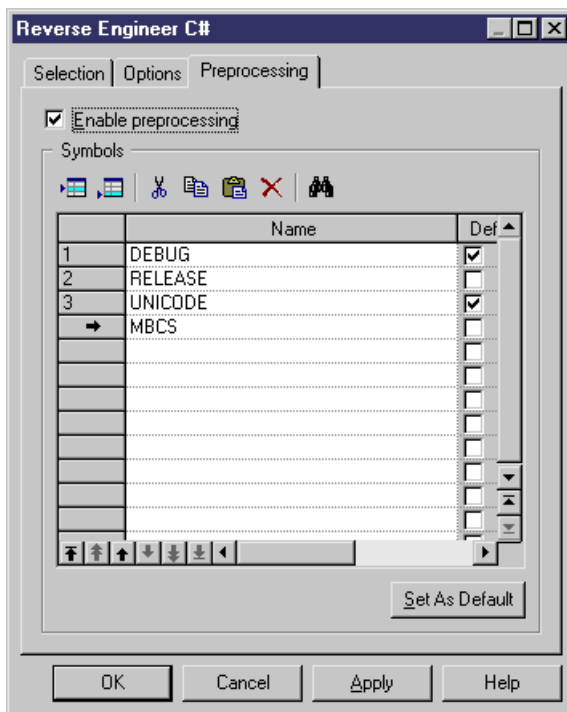
C# Reverse Engineering with Preprocessing

Preprocessing is an option you can enable or disable when you reverse engineer a model.

- Select **Language > Reverse engineering C#**.

The Reverse Engineering C# dialog box is displayed.

- Select files to reverse engineer in the Selection tab.
- Select reverse engineering options in the Options tab.
- Select the Enable preprocessing check box in the Preprocessing tab.
- Select symbols in the list of symbols.



- Click OK to start reverse engineering.

When preprocessing is over the code is passed to reverse engineering.

Reverse Engineering C# Files

You can reverse engineer C# files

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.

1. Select **Language > Reverse Engineer C#** to display the Reverse Engineer C# dialog box.
2. Select to reverse engineer files or directories from the Reverse Engineering 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 C# 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 the *Core Features Guide > The PowerDesigner Interface > 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.






CHAPTER 19 Working with 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++.

PowerDesigner can be used as a standalone product and as a plug-in for Visual Studio, allowing you to integrate its enterprise-level modeling capabilities in your standard .NET workflow. For more information, see *Core Features Guide > The PowerDesigner Interface > The PowerDesigner Add-In for Visual Studio*.

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 <i>C# 2.0 Assemblies</i> on page 505).
	Custom Attribute – for adding metadata (see <i>C# 2.0 Custom Attributes</i> on page 520).
	Delegate – type-safe reference classes (see <i>C# 2.0 Delegates</i> on page 512).
	Enum – sets of named constants (see <i>C# 2.0 Enums</i> on page 513).
	Struct – lightweight types (see <i>C# 2.0 Structs</i> on page 511).

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">• Class library• Windows Application• Console Application

Property	Description
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 General tab.
Description	Specifies an optional description for the assembly manifest.
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.

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.

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.

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 set to false. For information about creating and working with packages, see *Packages (OOM)* on page 50.

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:



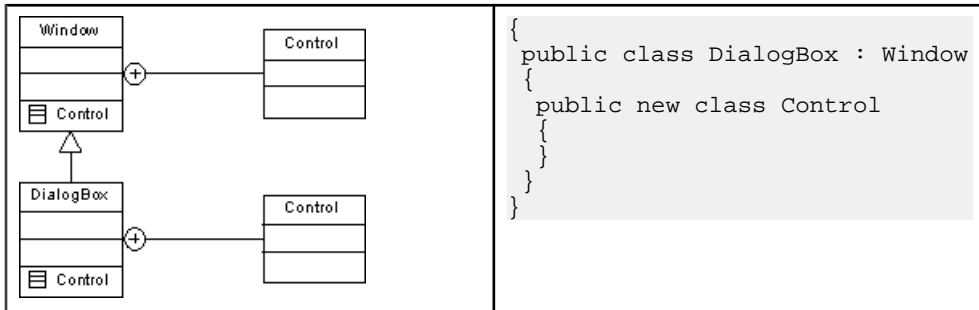
Classifiers defined directly at the model level fall into the **C#** global namespace.

C# 2.0 Classes

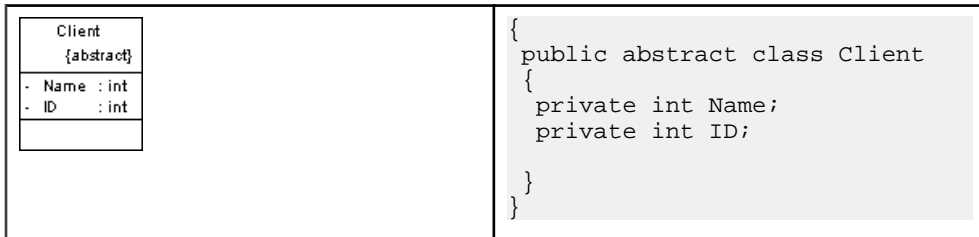
PowerDesigner models C# 2.0 classes as standard UML classes, but with additional properties.

For information about creating and working with classes, see *Classes (OOM)* on page 34.

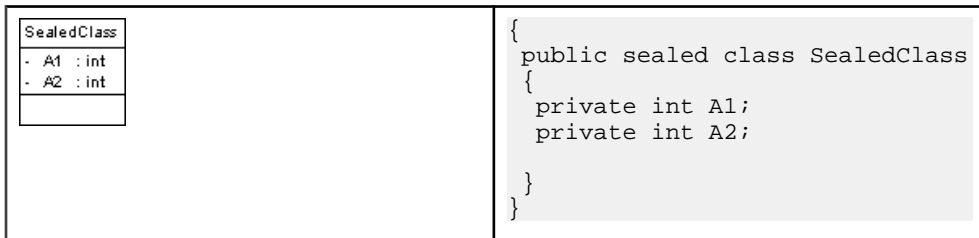
In the following example, class `DialogBox` inherits from class `Window`, which contains an inner classifier `Control`, as does class `DialogBox`:



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.

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)* on page 53.

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.

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)* on page 34.

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:

Point	<pre> { public struct Point { public int New() { return 0; } private int x; private int y; } } </pre>
# 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.

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)* on page 34.

<pre><<Delegate>> PackageDelegates + <<Delegate>> ActionOccurred() : int</pre>	<pre>{ public delegate int ActionOc- curred(); }</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.

C# 2.0 Enums

Enums are sets of named constants. PowerDesigner models enums as classes with a stereotype of <<Enum>>.

<pre><<enumeration>> 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)* on page 34.

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 **Model > Enum Objects** 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 **New > Enum**.

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.

C# 2.0 Fields

PowerDesigner models C# fields as standard UML attributes.

For information about creating and working with attributes, see *Attributes (OOM)* on page 65.

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.
ReadOnly	Specifies the readonly modifier for the field declaration.

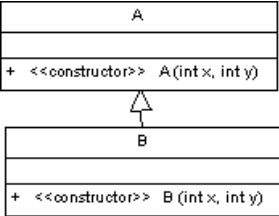
C# 2.0 Methods

PowerDesigner models C# methods as operations.

For information about creating and working with operations, see *Operations (OOM)* on page 78.

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.
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.

IntVector
+ <<Operator>> Oper1 (+, -) : int

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:

<pre><<struct>> Digit</pre>	<pre>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; }</pre>
<pre>- value : byte + <<constructor>> Digit(byte value) + <<ConversionOperator>> byte(Digit d) : byte + <<ConversionOperator>> Digit(byte b) : Digit</pre>	

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)* on page 65.

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).
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;

```

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)* on page 98.

PowerDesigner models C# implementation links between types and interfaces as standard UML realizations. For more information, see *Realizations (OOM)* on page 105.

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)* on page 111.

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.

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 > General Options** and click the **Variables** category.

1. Select **Language > Generate C# 2 Code** 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 *Chapter 9, Checking an OOM* on page 295).

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* on page 272).
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"> • Visibility • Type
Class members type sort	Specifies the order by which class members are sorted in terms of their type: <ul style="list-style-type: none"> • Methods – Properties - Fields • Properties – Methods - Fields • Fields – Properties - Methods
Class members visibility sort	Specifies the order by which class members are sorted in terms of their visibility: <ul style="list-style-type: none"> • Public - Private • Private – Public • None
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*.

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 > Templates and Generated Files (Profile)*.

7. [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.

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.

Reverse Engineering C# 2.0 Code

You can reverse engineer C# files into an OOM.

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. For more information, see *C# Reverse Engineer dialog Options tab* on page 523.
5. [optional] Click the Preprocessing tab and set any appropriate preprocessing symbols. For more information, see *C# reverse engineering preprocessing directives* on page 524.
6. 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 > The PowerDesigner Interface > 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.

C# Reverse Engineer Dialog Options Tab

The following options are available on this tab:

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
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 > Linking and Synchronizing Models > Shortcuts and Replicas > 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	Specifies that reversed classifiers (classes and interfaces) will not be generated from the model. To generate the classifier, you must select the Generate check box in its property sheet

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 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 C#, symbols have no value, they can 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.

C# Supported Preprocessing Directives

The following directives are supported during preprocessing:

Directive	Description
<code>#define</code>	Defines a symbol
<code>#undef</code>	Removes a previous definition of the symbol
<code>#if</code>	Evaluates a condition, if the condition is true, the statement following the condition is kept otherwise it is ignored
<code>#elif</code>	Used with the <code>#if</code> directive, if the previous <code>#if</code> test fails, <code>#elif</code> includes or exclude source code, depending on the resulting value of its own expression or identifier
<code>#endif</code>	Closes the <code>#if</code> conditional block of code

Directive	Description
#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.

Defining a C# Preprocessing Symbol

You can define C# preprocessing symbols in the preprocessing tab of the reverse engineering dialog box.

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* on page 276.

You can use the Set As Default button to save the list of symbols in the registry.

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.

CHAPTER 20 Working with XML

You can model XML schemas, DTDs and XDR files in an OOM.

PowerDesigner also provides the XML Schema Model (XSM), a dedicated XML modeling environment (see *XML Modeling*).

Designing for XML

This section explains how to design XML Schema objects in the PowerDesigner Object Oriented Model.

XML Object	Modeling in PowerDesigner
Schema	Any package or model can generate an XML Schema. You do not need to define specific attribute or stereotype for the package or model to generate an XML Schema.
Complex type	<p>In XML Schema, a complex type allows elements in its content and may carry attributes. Complex types can be:</p> <ul style="list-style-type: none">• Global, that is to say defined as a child of the schema element, in order to be reused among schema elements. You design a global complex type using a class with the <code><<complexType>></code> stereotype. In the following example, <code>UsAddress</code> is a complex type with a set of attributes that specify it:<pre><xsd:complexType name="UsAddress"> <xsd:element name="name" type="Name"/> <xsd:element name="street" type="string"/> <xsd:element name="town" type="string"/> <xsd:element name="zip" type="Integer"/> </xsd:complexType></pre>• Local to an element definition. In this case, you have to create a class with the <code><<element>></code> stereotype. You then need to set the <code>isComplexType</code> extended attribute of the class to <code>True</code>. This is to make sure that attributes defined in the <code><<element>></code> class are generated as a complex type:<pre><xsd:element name="customer"> <xsd:complexType> <xsd:element name="name" type="int"/> <xsd:element name="address" type="int"/> </xsd:complexType> </xsd:element></pre>

XML Object	Modeling in PowerDesigner
Simple type	<p>In XML Schema, a simple type can be a string or a decimal built into XML Schema, it can also be a type derived from those built-in the language. A simple type cannot contain elements or attributes. In PowerDesigner, you design a simple type using a class with the <<simpleType>> stereotype. You must add an attribute with the <<simpleType>> stereotype to this class.</p> <pre data-bbox="440 369 919 470"> <xsd:simpleType name="string"> <xsd:restriction base="string"> </xsd:restriction> </xsd:simpleType> </pre>
Deriving types	<p>XML Schema allows you to derive new types by extending or restricting an existing type. In PowerDesigner, you design the extension mechanism using a generalization between two classes. The contentDerivation extended attribute of the generalization allows you to set the type of derivation: extension or restriction.</p> <p>To design type derivation from a basic type (defined in Settings\DataTypes\BasicDataTypes in the object language editor) you cannot use classes and generalizations, you have to use the simpleContentBase and simpleContentDerivation extended attributes. For example, class A derives from basic data type xsd:string. You define this derivation setting the following values for class A:</p>
Union	<p>A <<union>> class generates a <<union>> attribute and is migrated to specify the attribute location. You can generate single line <union> tag as follows: <union memberTypes="{member types list}"/>. You can define a value for the extended attribute memberTypes used with simple Type attributes (either <<simpleType>> or <<simpleAttribute>>).</p>
Global element	<p>A global element is declared as a child of the schema element; it can be referenced in one or more declarations. You define an XML Schema global element using a class with the <<element>> stereotype in PowerDesigner. You define the type of a global element using the following methods:</p> <ul style="list-style-type: none"> • For a complex type, use attributes. In this case you have to set the isComplexType extended attribute to True for attributes defined in the <<element>> class to be generated as a complex type: <pre data-bbox="481 1248 1056 1399"> <xsd:element name="customer"> <xsd:complexType> <xsd:element name="name" type="int"/> <xsd:element name="address" type="int"/> </xsd:complexType> </xsd:element> </pre> • For a simple type, set a value for the type extended attribute <pre data-bbox="481 1442 1173 1491"> <xsd:element name="Customer" type="CustomerList-Type"/> </pre>

XML Object	Modeling in PowerDesigner
Element group	<p>An element group is declared as a child of the schema element; it can be referenced in one or more declarations using the <<ref>> stereotype on an association. You define an XML Schema element group using a class with the <<group>> stereotype in PowerDesigner.</p>
Attribute group	<p>An attribute group is a set of attributes. You define an XML Schema attribute group using a class with the <<attributeGroup>> stereotype in PowerDesigner. All the attributes of this class should have the <<attribute>> stereotype. For example, the following attribute group called item gathers information about an item in a purchase order:</p> <pre data-bbox="440 505 1150 631"> <xsd:attributeGroup name="item"> <xsd:attribute name="weight" type="Integer"/> <xsd:attribute name="shipping_method" type="Integer"/> </xsd:attributeGroup> </pre>
Reference	<p>A reference is a simplified declaration of an element or an attribute group referencing a global definition. In PowerDesigner, you design a reference using an association with the <<ref>> stereotype. In the following example, element Customer references complex type UsAddress.</p> <pre data-bbox="440 788 1103 991"> <xsd:element name="Customer"> <xsd:complexType> <xsd:element name="name" type="int"/> <xsd:element name="ID" type="int"/> <xsd:complexType ref="UsAddress" minOccurs="0" maxOccurs="unbounded"/> </xsd:complexType> </xsd:element> </pre> <p>Note that the referenced element is introduced by its stereotype. In the above example, UsAddress is of <<complexType>> and complexType is displayed in the reference line: <xsd:complexType ref="UsAddress" minOccurs="0" maxOccurs="unbounded"/></p>

XML Object	Modeling in PowerDesigner
Sequence	<p>XML Schema elements can be constrained to appear in the same order as they are declared, this is called a sequence. In PowerDesigner, depending on the type of sequence you need to design, you can use one of the following methods:</p> <ul style="list-style-type: none"> • If all the class attributes are defined in the sequence, you should create a class without stereotype and set the class extended attribute <code>isSequence</code> to true. In the following example, all attributes of class <code>item_sequence</code> are defined in the sequence: <pre data-bbox="481 447 1089 621"> <xsd:element name="item_sequence"> <xsd:sequence> <xsd:element name="prodName" type="int"/> <xsd:element name="prodID" type="int"/> <xsd:element name="prodPrice" type="int"/> </xsd:sequence> </xsd:element> </pre> • If some of the class attributes do not belong to the sequence, you have to design the following construct: create a class containing the attributes belonging to the sequence and assign the <code><<sequence>></code> stereotype to this class. Create another class containing the other attributes. Select the Inner link tool in the Toolbox and draw a link from the second class to the <code><<sequence>></code> class. The resulting code is the following: <pre data-bbox="481 812 1042 986"> <xsd:element name="PurchaseOrder"> <xsd:sequence> <xsd:element name="shipTo" type="int"/> <xsd:element name="billTo" type="int"/> </xsd:sequence> <xsd:element name="prodID" type="int"/> </xsd:element> </pre> <p>By default, inner classes are generated before attributes in a class (as defined in the Class\Template\body entry in the object language definition file). However, you can modify generation order among class attributes using the attribute migration feature. To do so, you should create an association from the parent class to the <code><<sequence>></code> class, right-click the association and select Migrate > Migrate Navigable Roles. The migrated attribute can then be moved in the list of class attributes in order to have the desired generation order.</p> <pre data-bbox="440 1225 1005 1399"> <xsd:element name="PurchaseOrder"> <xsd:element name="prodID" type="int"/> <xsd:sequence> <xsd:element name="shipTo" type="int"/> <xsd:element name="billTo" type="int"/> </xsd:sequence> </xsd:element> </pre>

XML Object	Modeling in PowerDesigner
Choice & All	<p>A choice allows you to display only one child in an instance of an element. All allows you to display all the elements in the group once or not at all. In PowerDesigner, you design a choice/all mostly as a sequence.</p> <ul style="list-style-type: none"> If all the class attributes are defined in the choice/all, you should create a class without stereotype and set the class extended attribute isChoice/isAll to true. In the following example, all attributes of class InternationalShipping are defined in the choice: <pre data-bbox="481 447 1159 621"> <xsd:element name="InternationalShipping"> <xsd:choice> <xsd:element name="EuropeShipping" type="int"/> <xsd:element name="AfricaShipping" type="int"/> <xsd:element name="AsiaShipping" type="int"/> </xsd:choice> </xsd:element> </pre> If some of the class attributes do not belong to the choice, you have to create a class containing the attributes belonging to the choice and assign the <<choice>>/<<all>> stereotype to this class. Create another class containing the other attributes. Select the Inner link tool in the Toolbox and draw a link from the second class to the <<choice>>>>/<<all>> class: You can also use the attribute migration feature to modify generation order among attributes (see Sequence, above).

Generating for XML

When you generate XML from an OOM, PowerDesigner creates one XML file per package. The file contains the definition of each of the classes you select to generate in the Generation dialog box. You can select any of the classes from the model, including those that are contained within packages or sub-packages.

- XML Schema - a .XSD file will be generated per package or model. You can generate an XML Schema definition in this specific format for validations and/or data exchange purposes
- XML DTD - a .DTD file will be generated per package or model. A DTD file provides an overall structure for an XML file. You can generate an XML DTD definition in this specific format for validations and/or data exchange purposes

Navigable associations are migrated and generated as attributes although they do have their own definition in the object language file. Interfaces, operations, and links (dependencies, realizations and generalizations) are not included in the generated file.

To change the XML format type, you must change the object language for the model (by selecting **Language > Change Current Object Language**). You can create a new XML object language based on an existing one if you want to generate in another type of XML format that is not available in PowerDesigner.

The files generated are generated with the .xsd, .dtd or .xdr extension depending on the XML format specified. A generation log file is also created after generation.

1. Select **Language > Generate XML-language Code** to open the Generation dialog box.
2. Enter a destination directory for the generated files and, optionally, select the Check Model checkbox to verify the validity of your model before generation.
3. Click the Selection tab and select the objects to include in the generation from the various sub-tabs.
4. Click OK to generate XML files in the specified directory.

Reverse-Engineering XML

You can reverse engineer *.DTD, *.XSD, and *.XML files into an OOM. 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.

When you reverse engineer a DTD file into an OOM, you get a more readable view of the DTD. This feature can be very helpful when you want to check and understand a new DTD that you have not generated. When you reverse engineer a DTD file into an OOM:

- Referenced DTD files, using DOCTYPE keyword, may not be reversed correctly, it is therefore better to choose a real DTD file rather than an XML document referencing a DTD file with a DOCTYPE document type declaration
- Elements of type #PCDATA are reversed as attributes
- An element that has both a parent and a child element is linked to its parent element by an aggregation link
- If an empty element has no child object but has attributes, it is reversed as a class and its attributes become attributes of the class
- Attributes of type ID and IDREF(S) are reversed as attributes with ID and IDREF(S) data types
- The attribute sequence order may not be preserved
- Attribute groups structure is not preserved
- DTD files may not be reversed properly if they contain some reference to an undefined parameter entity

When you reverse engineer an XML Schema file into an OOM, the sequence order of the contents of an element may not be preserved and the following main elements are not reverse engineered:

- Namespace
- Key/keyref
- Field/selector

- Unique
 1. Select **Language > Reverse Engineer XML** to open the Reverse XML dialog box.
 2. Click the Add button in the Selection page, select the .xsd, .xdr or .dtd files you want to reverse, and click Open. You can select multiple files with the CTRL or SHIFT keys. The Reverse XML dialog box displays the files you selected.
 3. Click OK to reverse the files. If the model to which you are reverse engineering already contains objects, the Merge Models dialog box is displayed. For more information on merging models, see *Core Features Guide > The PowerDesigner Interface > Comparing and Merging Models*.

The classes are added to your model and are visible in the diagram and in the Browser.

CHAPTER 21 Working with C++

PowerDesigner supports the modeling of C++ programs including round-trip engineering.

Designing for C++

This section explains how to design C++ objects in the PowerDesigner Object Oriented Model.

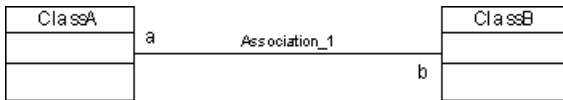
Namespace Declaration for Classifiers

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;

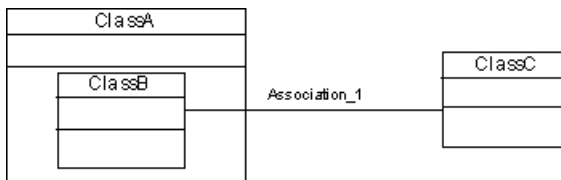
protected:
private:

};

#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

Generating for C++

When generating with C++, the files generated are generated for classes and interfaces.

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.

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 Select tools to the right of the Folder Selection 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.

CHAPTER 22 **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:

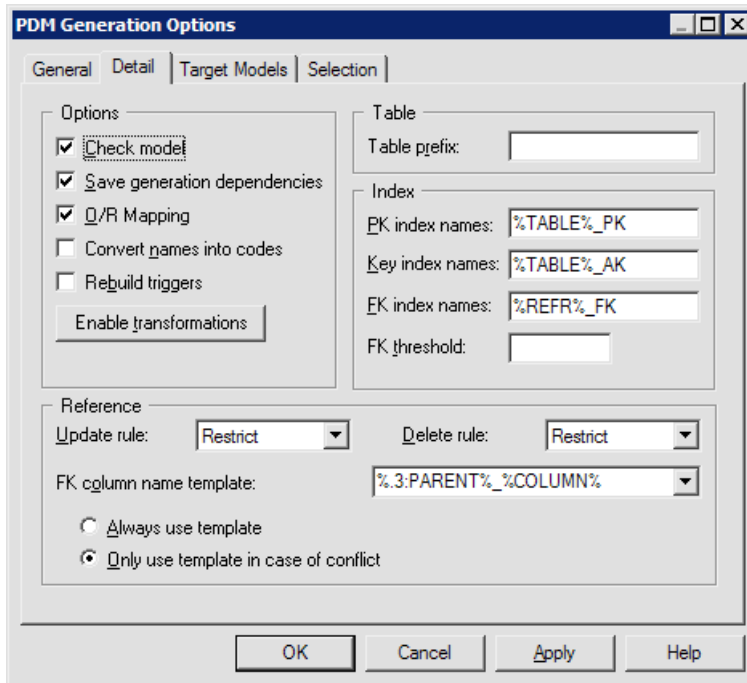
- Top-down – generate tables and other PDM objects from OOM classes
- Bottom-up – generate classes and other OOM objects from PDM tables
- Meet-in-the-middle – manually define mappings between classes and tables using the visual mapping editor

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.

1. Create your OOM, and populate it with persistent classes (see *Entity Class Transformation* on page 541), 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.

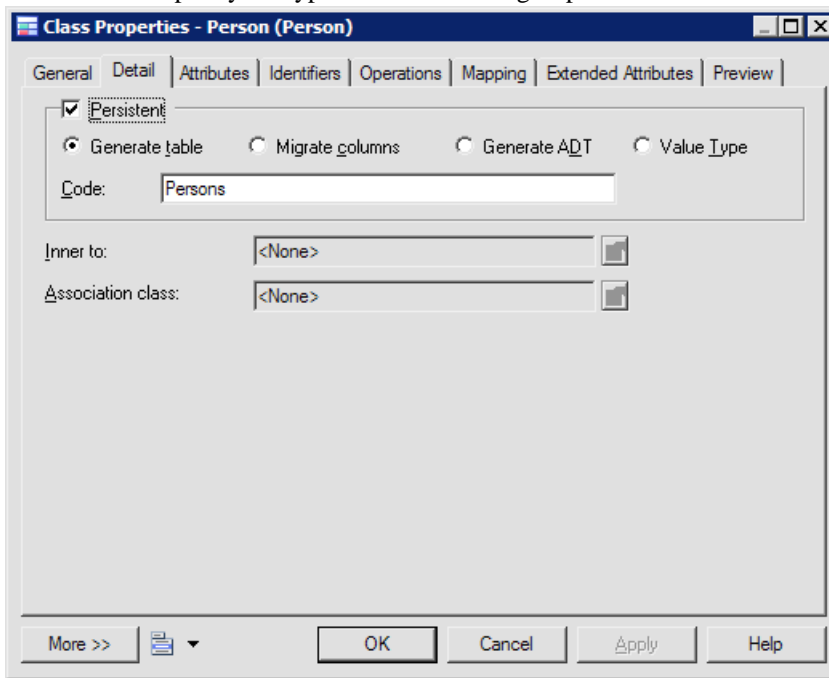
3. On the General tab, specify the DBMS type and the name and code of the PDM to generate (or select an existing PDM to update).
4. Click the Detail tab and select the O/R Mapping checkbox. You can optionally also specify a table prefix that will be applied to all generated tables.



5. Click the Selection tab and select the OOM objects that you want to transform into PDM objects.
6. Click OK to generate (or update) your PDM.

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:

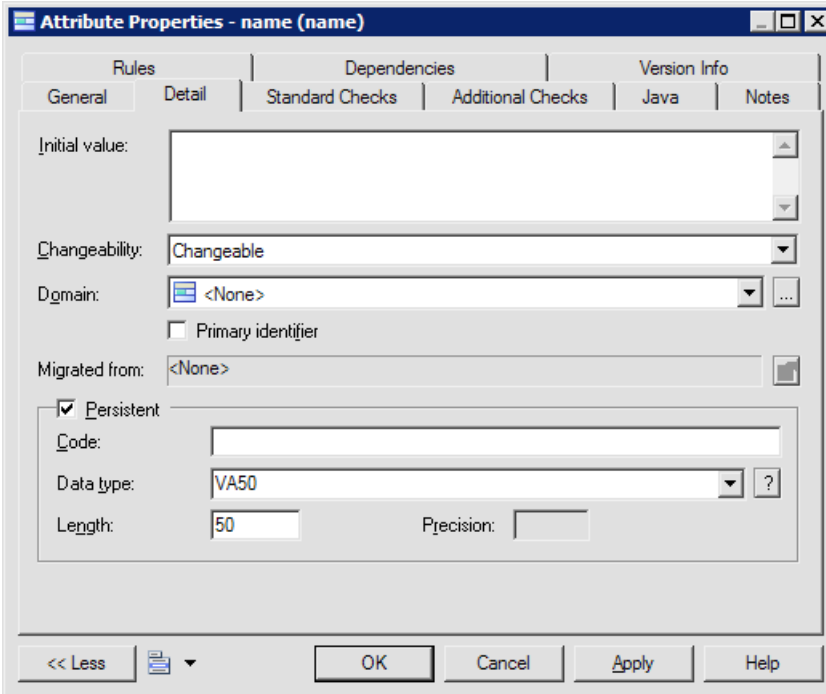
- **Generate table** - These classes are called Entity classes, and will be generated as separate tables. You can customize the code of the generated tables in the Code box in the Persistent groupbox. Only one table can be generated for each entity class with this type, but you can manually map an entity class to multiple tables (see *Defining entity class mapping* on page 552).
- **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)

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 *Defining primary identifier mapping* on page 556). Persistent attributes contained in primary identifiers are transformed into columns of primary keys.

Attribute Transformation

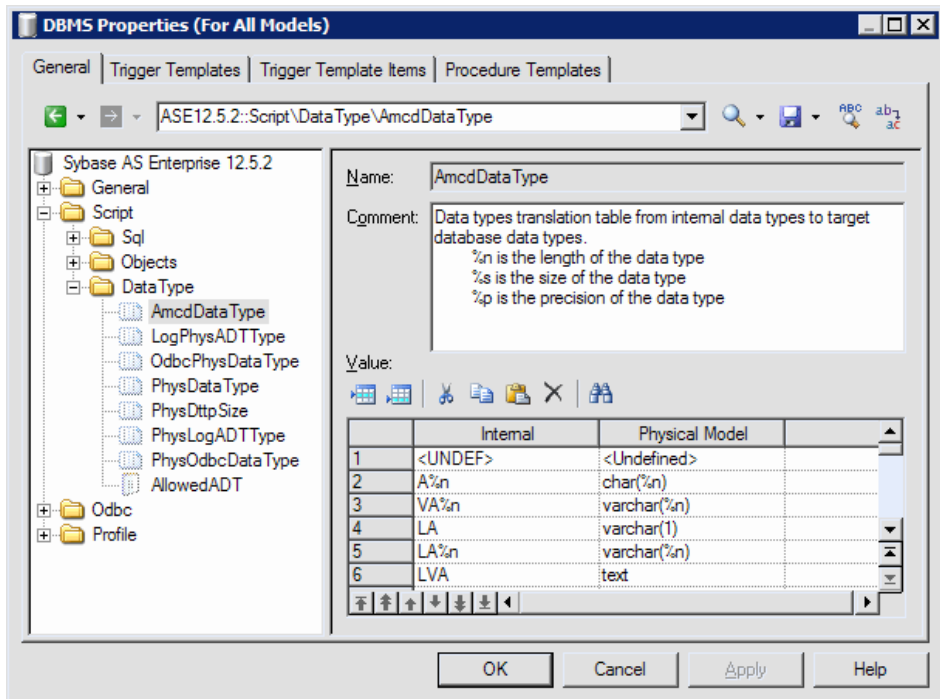
To transform an attribute into a column, select the Persistent option on the Detail tab of its property sheet.



The screenshot shows the 'Attribute Properties - name (name)' dialog box with the 'Detail' tab selected. The 'Persistent' checkbox is checked. The 'Data type' is set to 'VA50', 'Length' is 50, and 'Precision' is empty. Other fields include 'Initial value', 'Changeability' (Changeable), 'Domain' (<None>), 'Migrated from' (<None>), and 'Primary identifier' (unchecked). The bottom of the dialog has buttons for '<< Less', 'OK', 'Cancel', 'Apply', and 'Help'.

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:



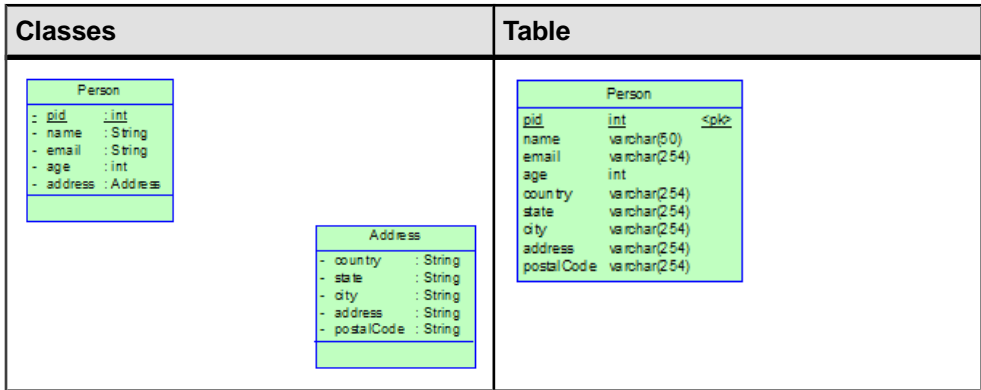
- Complex Data Type - based on a classifier. The transformation depends on the persistent settings of the classifier. The classifier is generally used as a value type class (see *Value type transformation* on page 543).

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.

Value Type Transformation

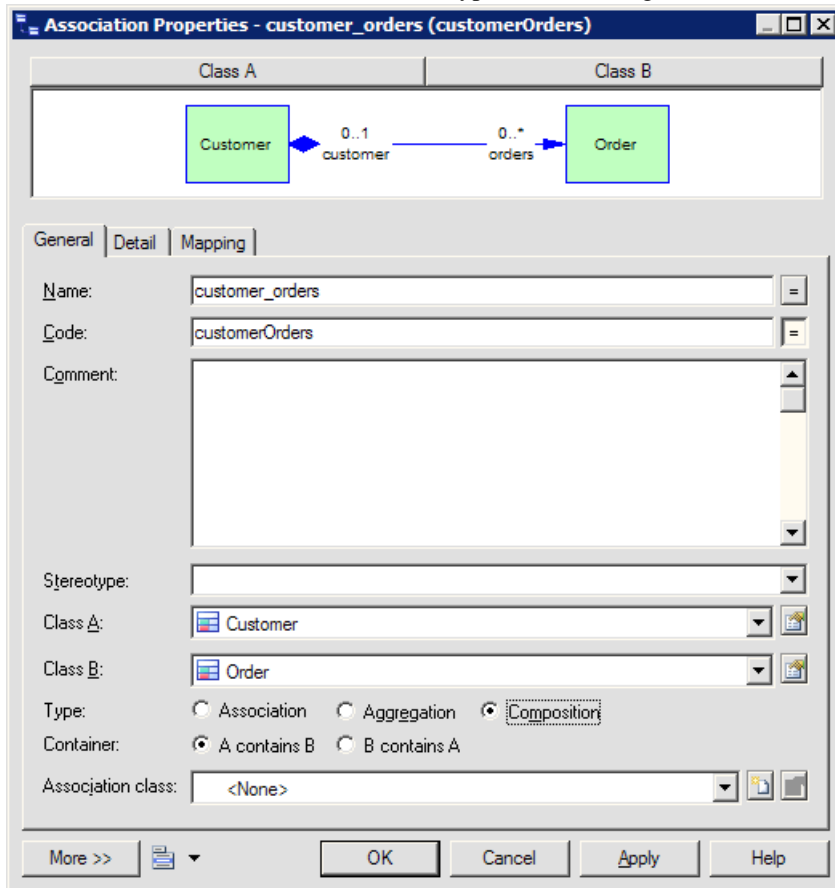
PowerDesigner supports fine-grained persistence model. Multiple classes can be transformed into single table.

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*.



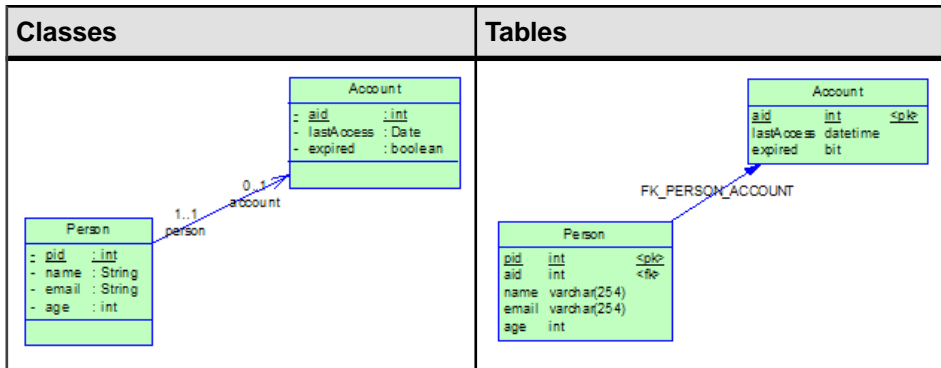
Association Transformation

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.



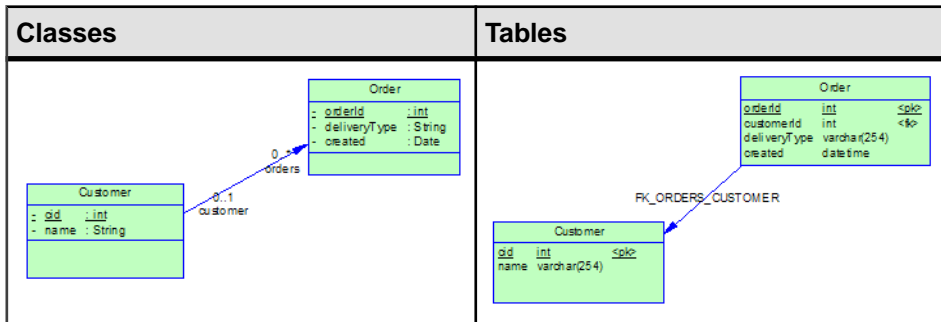
Transformation rules differ according to the type of the association:

- 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.

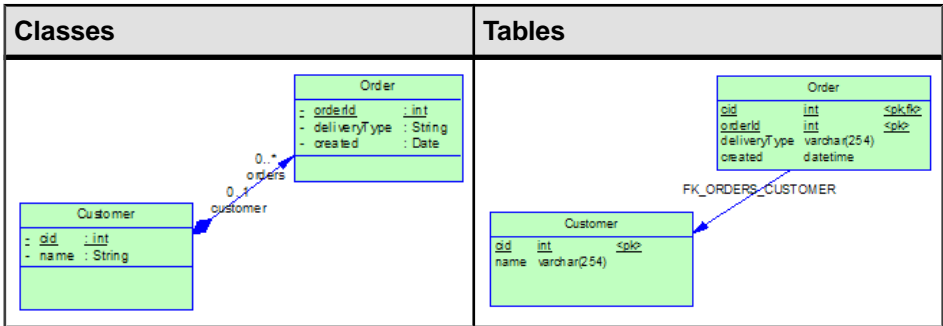


The generated foreign key has the same direction as the association direction. If the association is bidirectional (can navigate in two ways), foreign keys with both directions will be generated since PowerDesigner does not know which table is the parent table. You need to delete one manually.

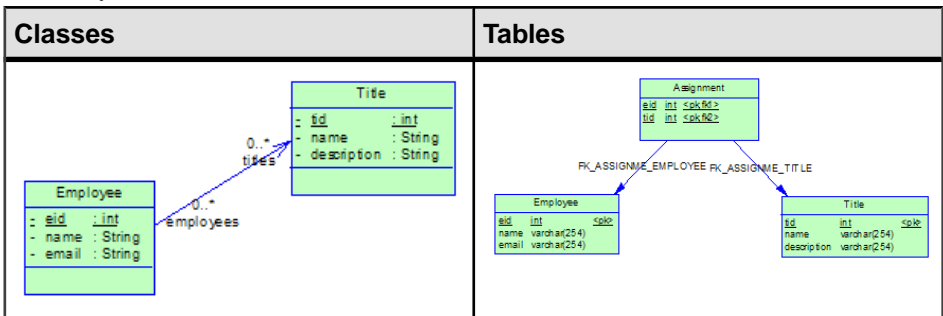
- One-to-many association - just one foreign key will be generated for each one-to-many association, whatever its direction (bidirectional or unidirectional). The reference key navigates from the table generated from the entity class on multiple-valued side to the table generated from the entity class 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 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.

For more information, see *One-to-Many Association Mapping* on page 561.

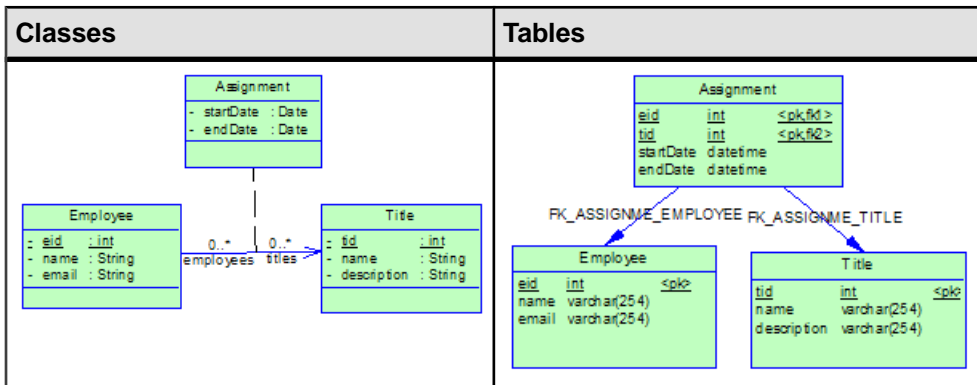
Note: The minimal multiplicity of association ends can affect the Mandatory property of the generated reference keys:

- 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.

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.

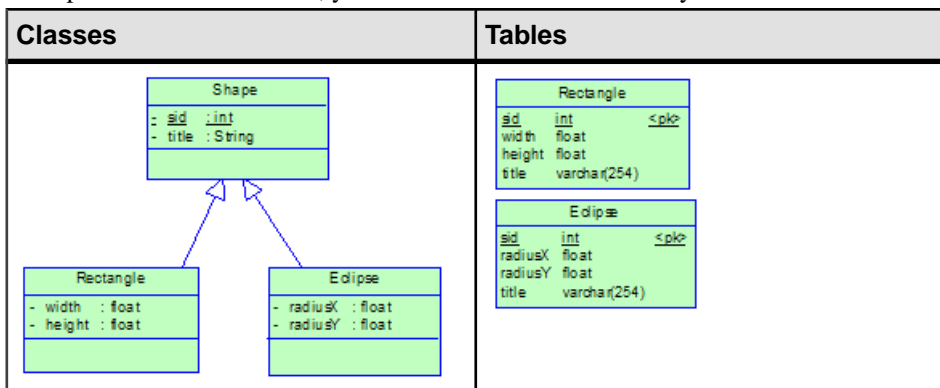
In the following example, we have defined an association class to hold extra information for the association:



Inheritance Transformation

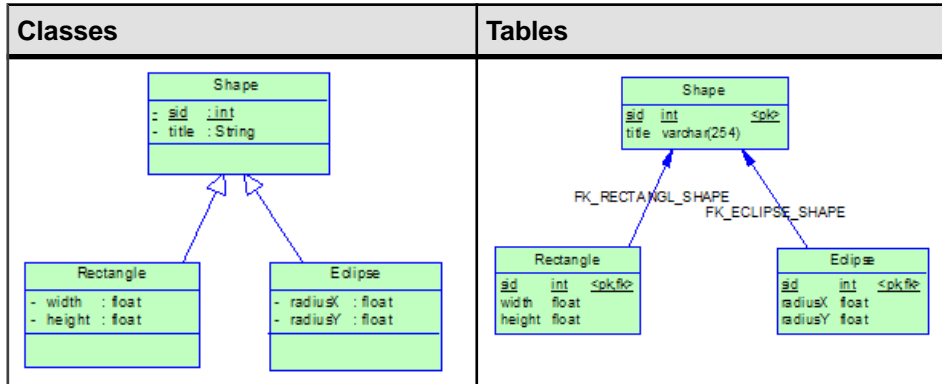
PowerDesigner supports various mapping strategies for inheritance persistence. Each strategy has its pros and cons, and you should select the most appropriate one for your needs. You can also apply mixed strategies, but this may not be well supported by your persistence framework.

- 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.

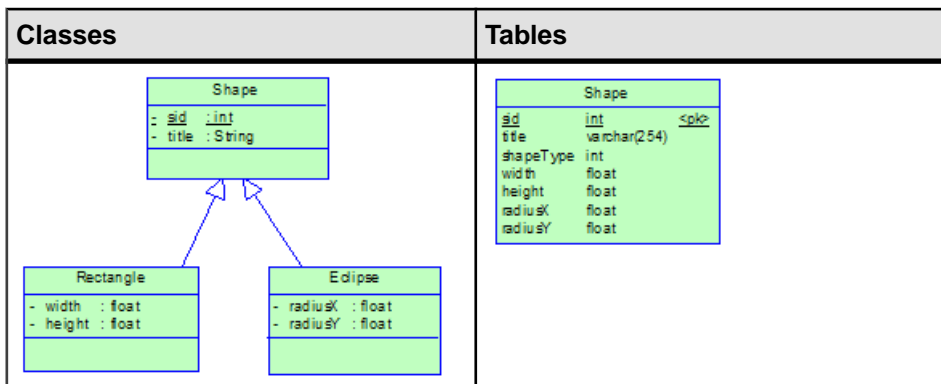
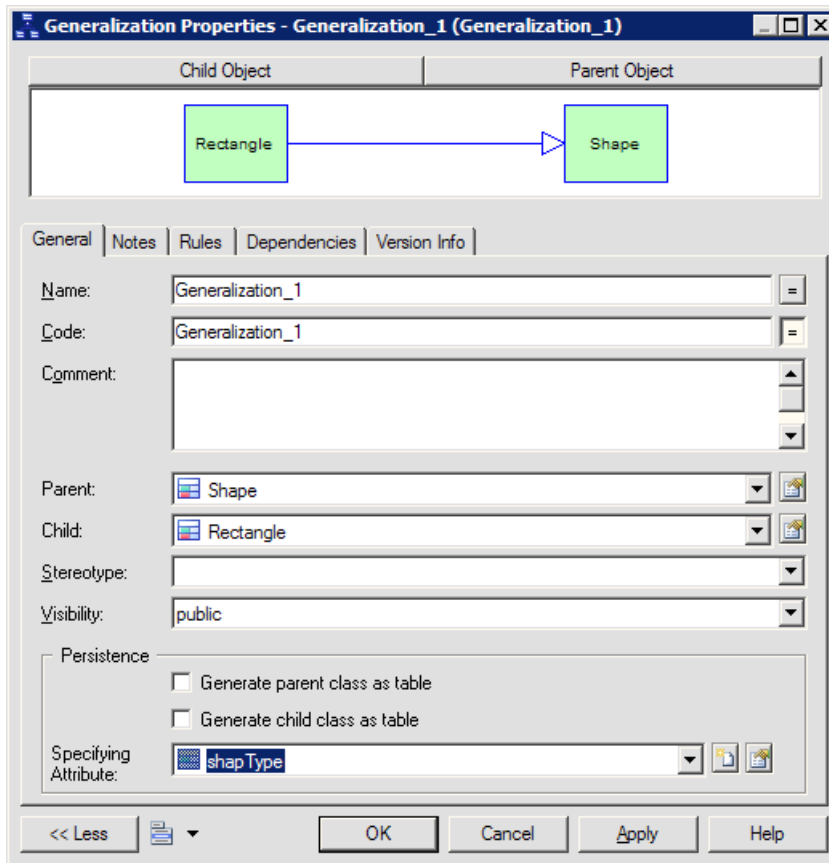
In order to apply this kind of strategy, you should set the transformation type of all the classes to *Generate table*. You can also, optionally, define a discriminator.



- 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

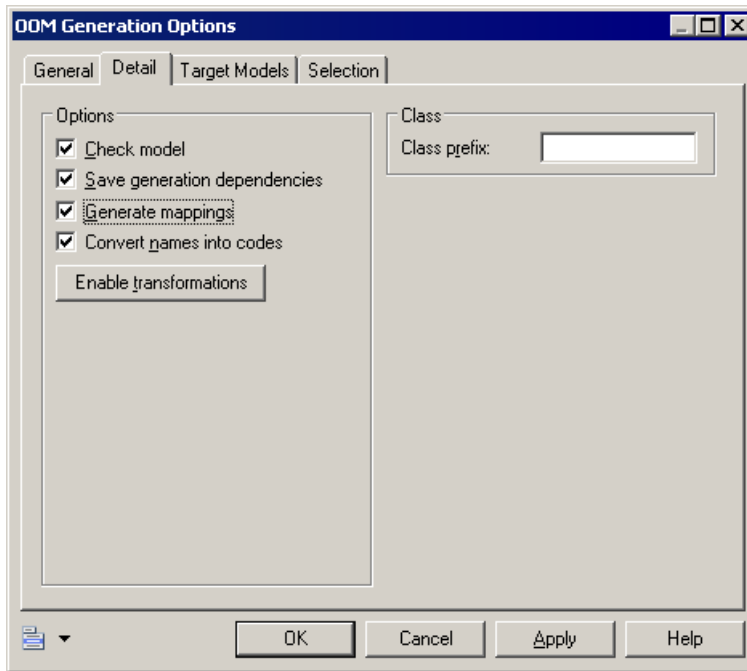
Generate table, the columns transformed from their persistent attributes will be migrated into tables transformed from sub-classes. The migration to sub-classes has higher priority.

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.

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.
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 select the tables that you want to transform into classes.
6. Click OK to generate (or update) your OOM.

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.

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.

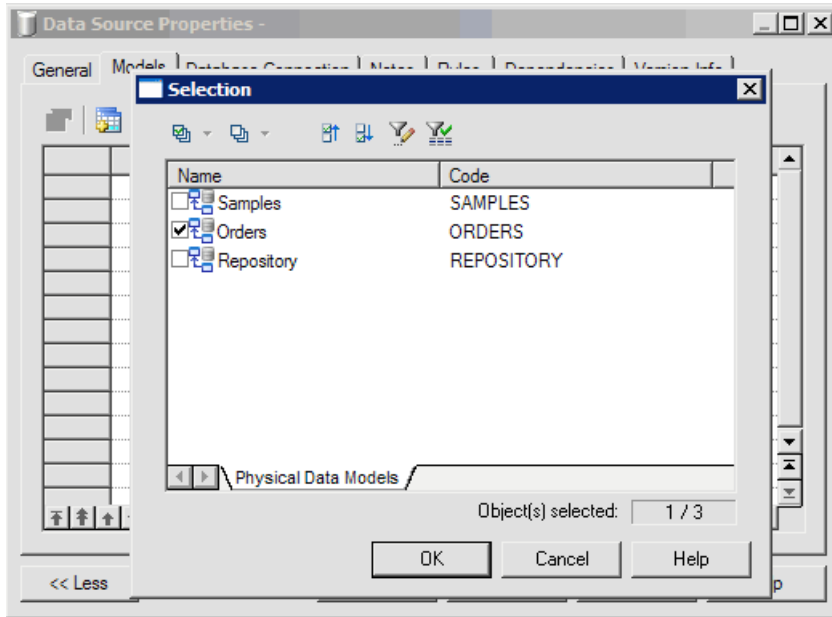
Note: when your O/R Mapping models are related with a specific technology, for example when you are modeling for EJB 3.0 persistence, there will be some constraints and we provide model checks to help you check the syntax of the mappings you have defined.

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.

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.

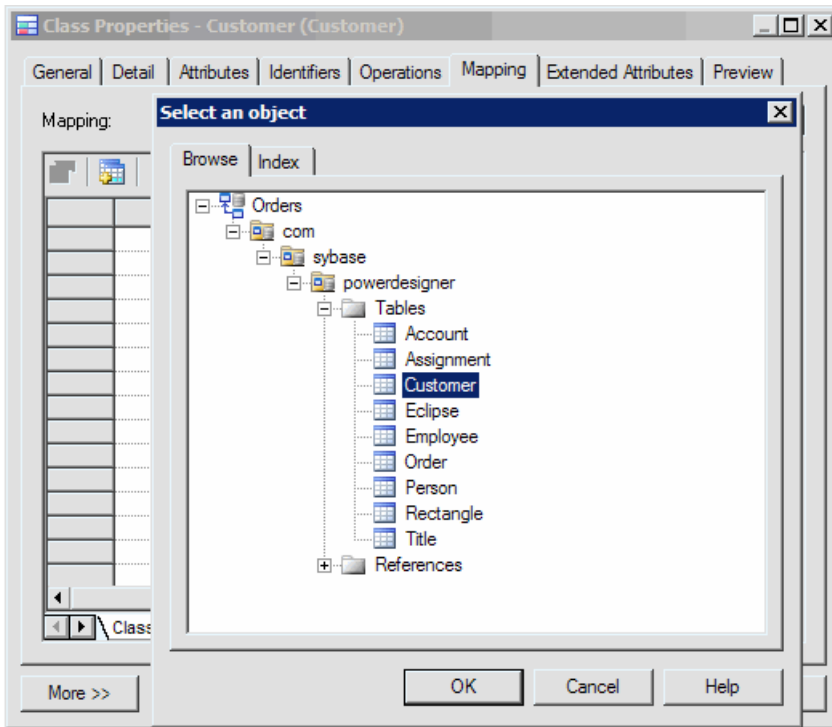
When you are familiar with O/R Mapping concepts, you can use the Mapping Editor.

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.



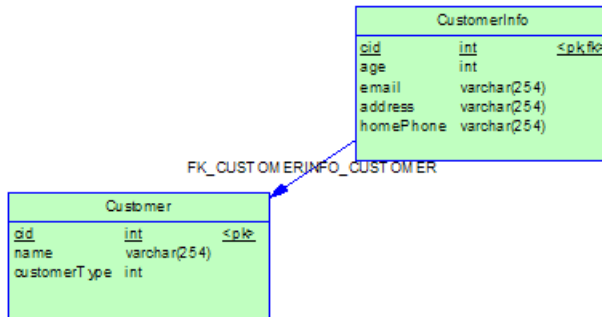
You can add tables, views and references as mapping sources. There are some constraints on views as mapping sources, as some views cannot be updated. When you add references as mapping sources, tables at the two ends will also be added.

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*:

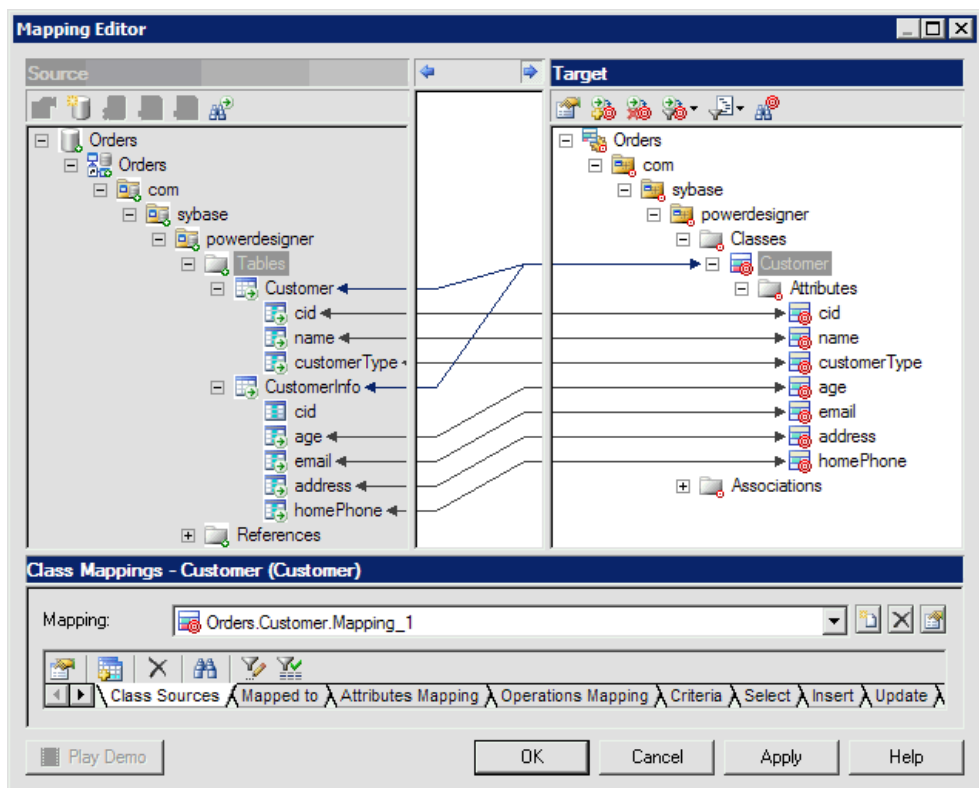
Customer	
- id	: int
- name	: String
- customerType	: int
- age	: int
- email	: String
- address	: String
- homePhone	: String

It can be mapped to two tables:



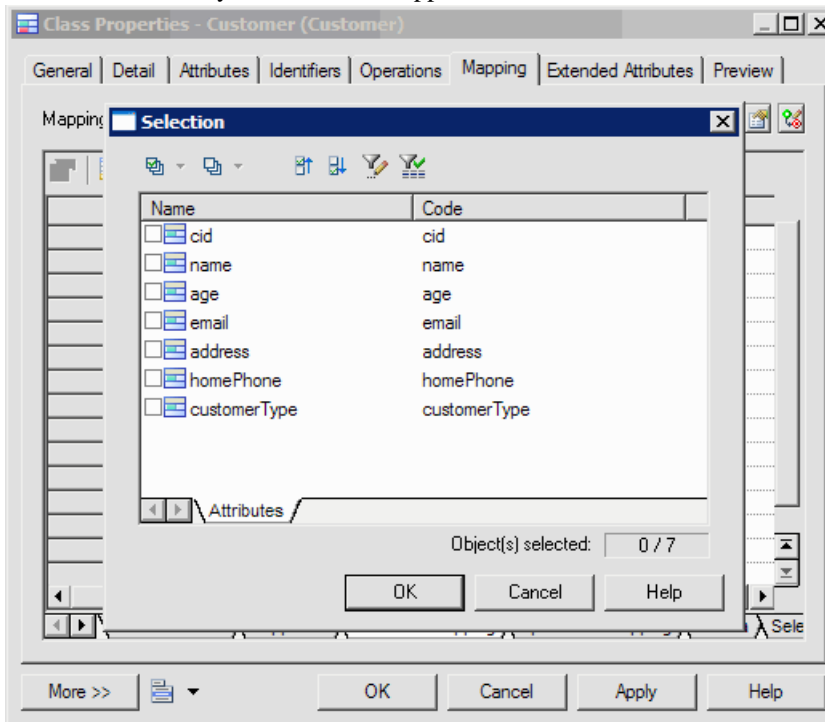
The *Customer* table is the primary table. The *CustomerInfo* table is the secondary table and it has one reference key referring to the primary table, which is joined on its primary key.

With the Mapping Editor, you just have to drag the two tables and drop them to class *Customer* to define class mappings.

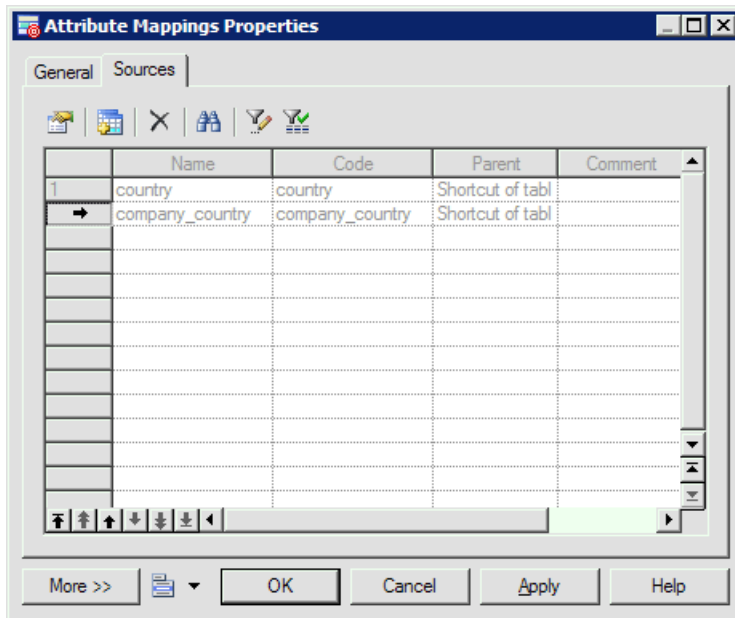


Attribute Mapping

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.

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.

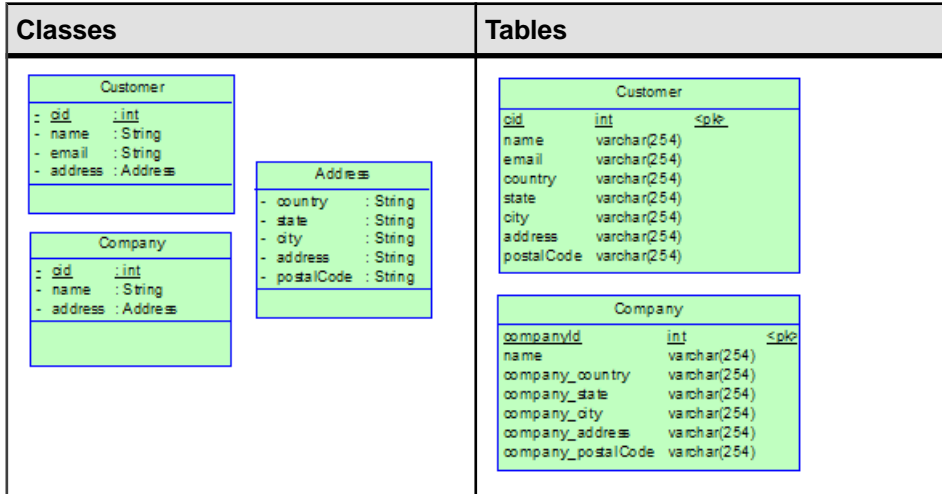
Column(s) of primary keys can be mapped to associations (see *Association Transformation* on page 544). They are migrated from primary keys of other tables.

- *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.

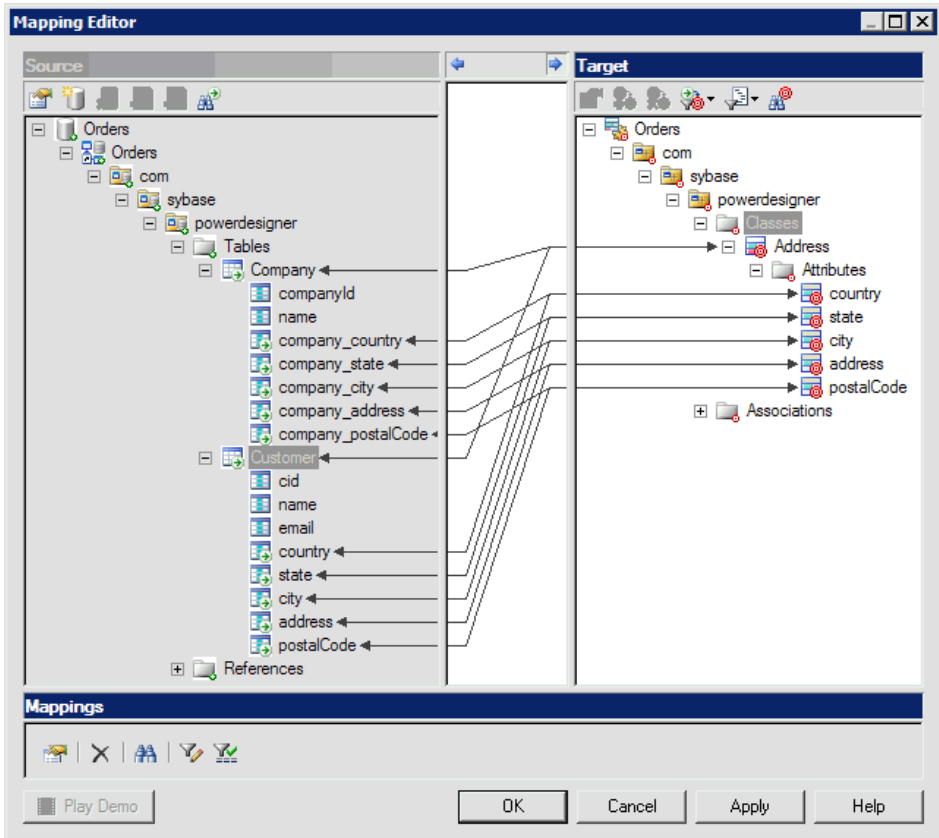
Attributes of value type classes are mapped to columns, which are embedded in primary tables mapped by other entity classes. So you have to add primary tables of the containing classes as value type classes' mapping sources. If the value type class is used in more than

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:



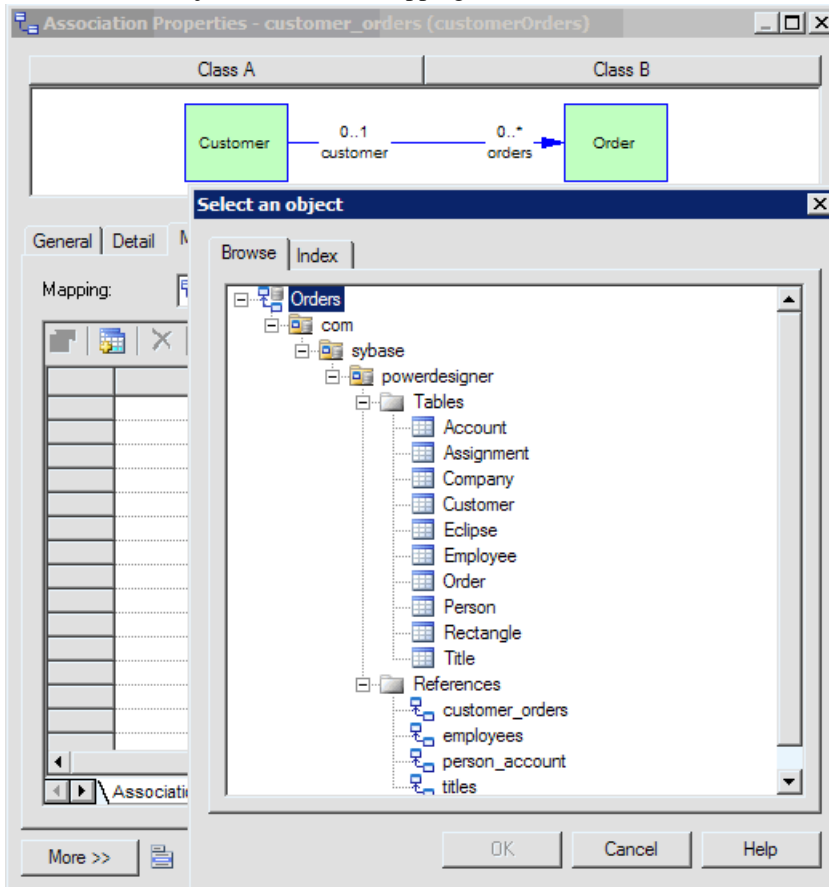
The mapping is easier to visualize in the Mapping Editor.



Primary identifier mapping is mandatory for entity classes.

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.



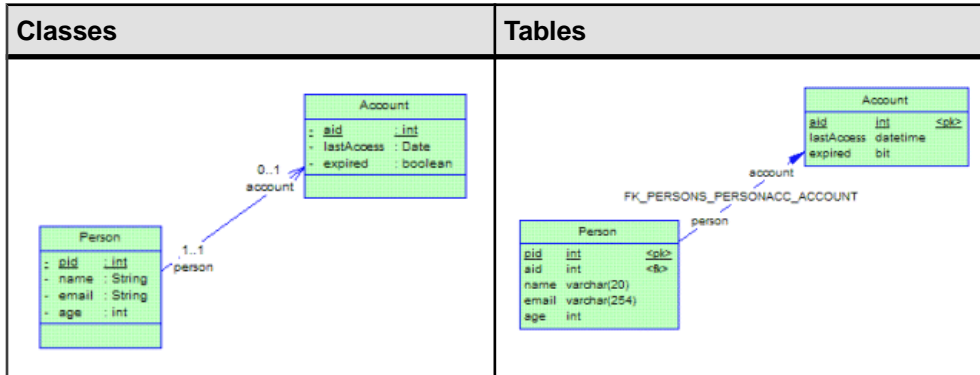
Associations defined between entity classes can be mapped to reference keys or tables. In order to define association mapping, you have to add the references keys or tables as mapping sources. When you add reference keys, the tables on their ends will also be added.

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.

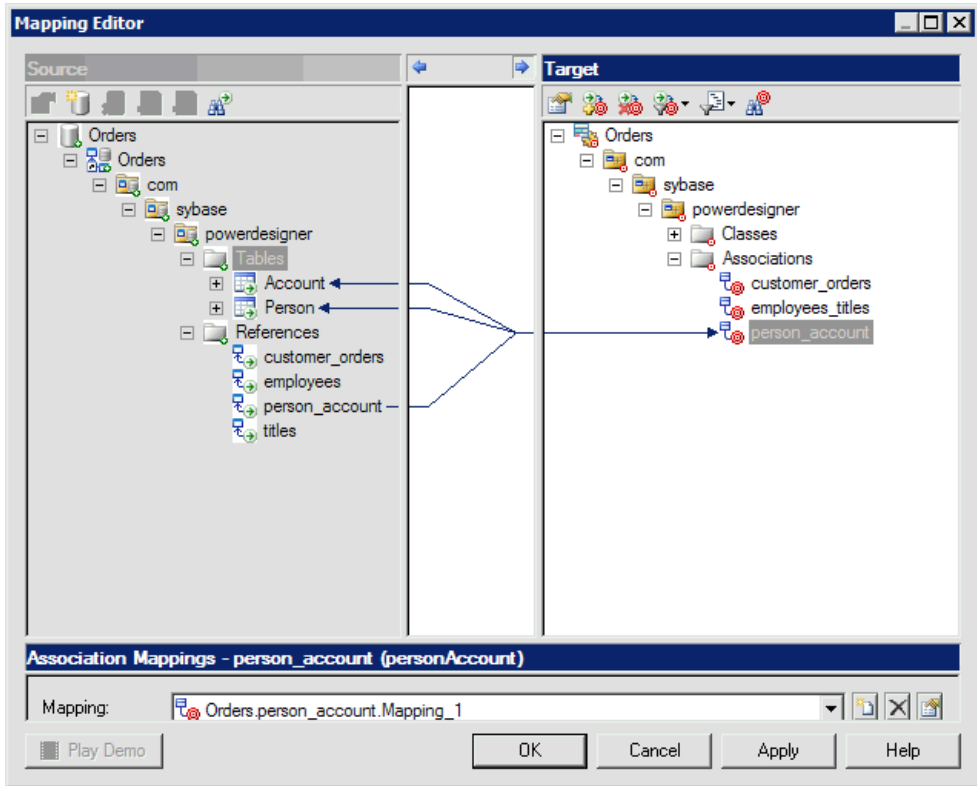
One-to-One Association Mapping Strategy

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.

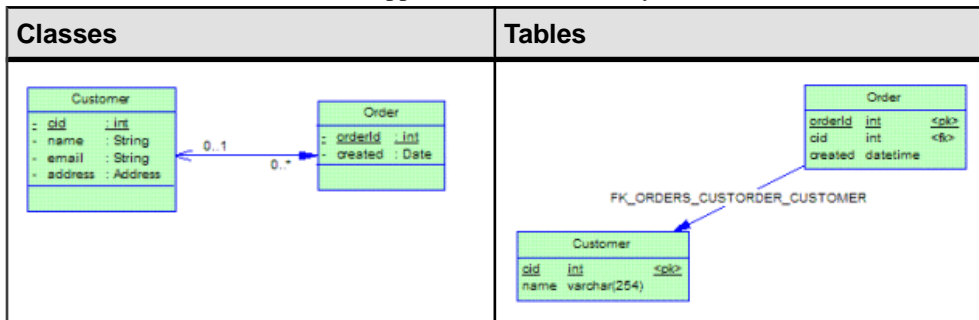


For a bi-directional one-to-one association, you also just can map it to one reference key. But the reference can navigate in either direction.

One-to-Many Association Mapping Strategy

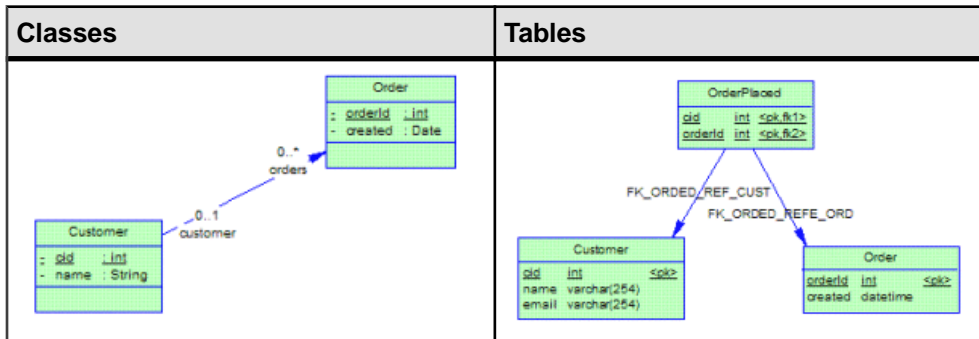
Each unidirectional many-to-one association is mapped to a reference that has the same direction as the association.

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:



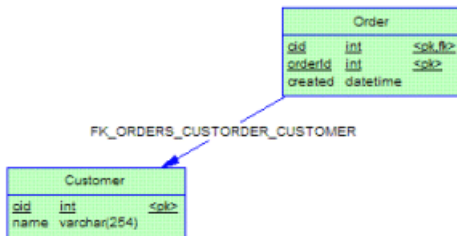
Each unidirectional one-to-many association should be mapped to a middle table and two references that refer to tables mapped by the entity classes on both ends.

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:

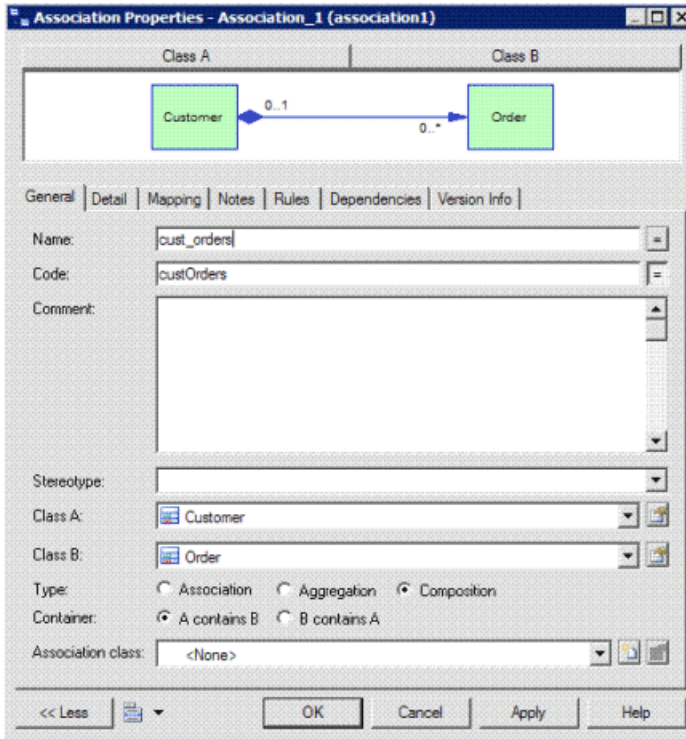


You can map a bi-directional one-to-many association as unidirectional many-to-one association. The reference just can navigate from primary table of class on multiple-valued side to primary table of class on single-valued side.

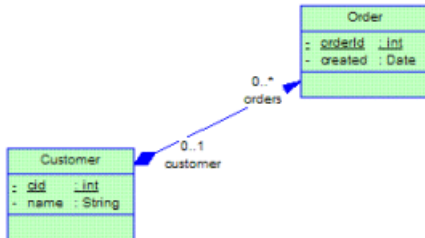
Sometimes we want to make the primary key of parent table be part of primary key of the child table and reference key join on the migrated column(s). For example we can map *Customer*, *Order* and bi-directional one-to-many association to tables and reference key as follows:



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:

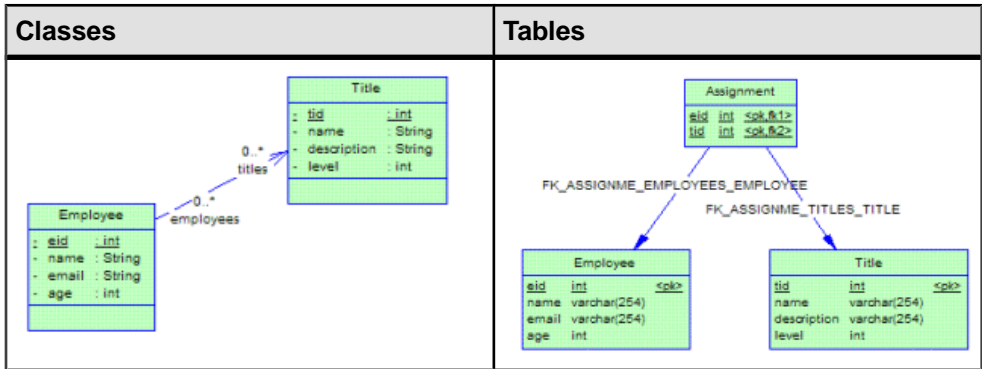


Then add the reference as mapping sources. You just can define the same way association mapping for bi-directional one-to-many association.

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:

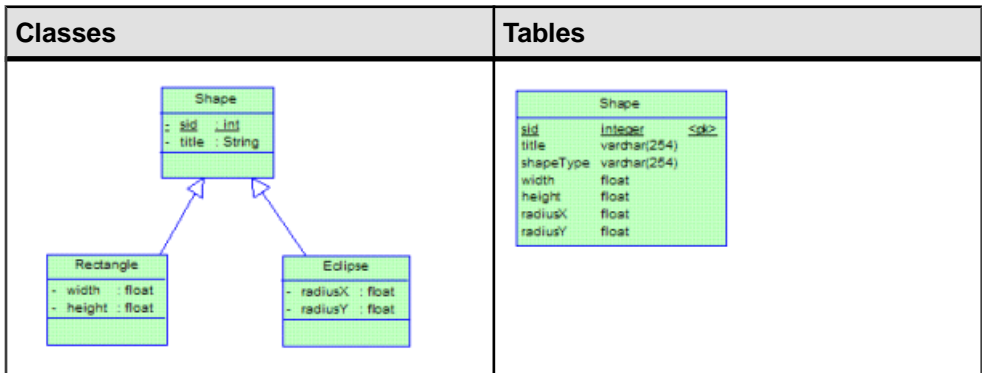


Defining Inheritance Mapping

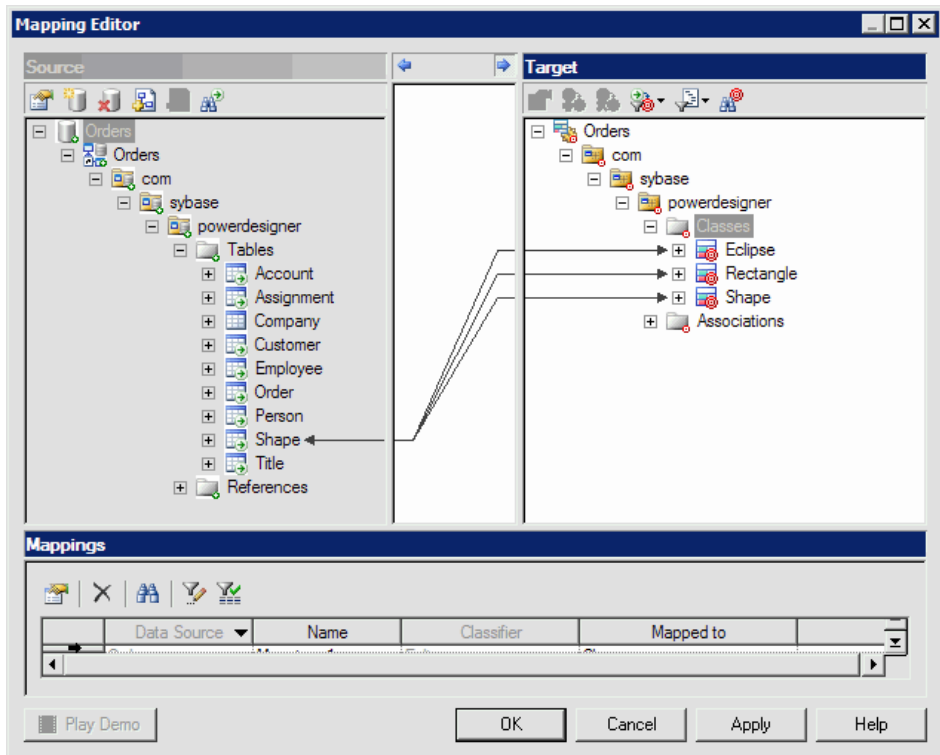
Inheritance can be mapped using a table per class, joined subclass, or table per class hierarchy inheritance mapping strategy. You can apply any of these inheritance mapping strategies or mix them. You should define primary identifier on the entity class that is the root of the entity hierarchy.

Table Per Class Hierarchy Inheritance Mapping Strategy

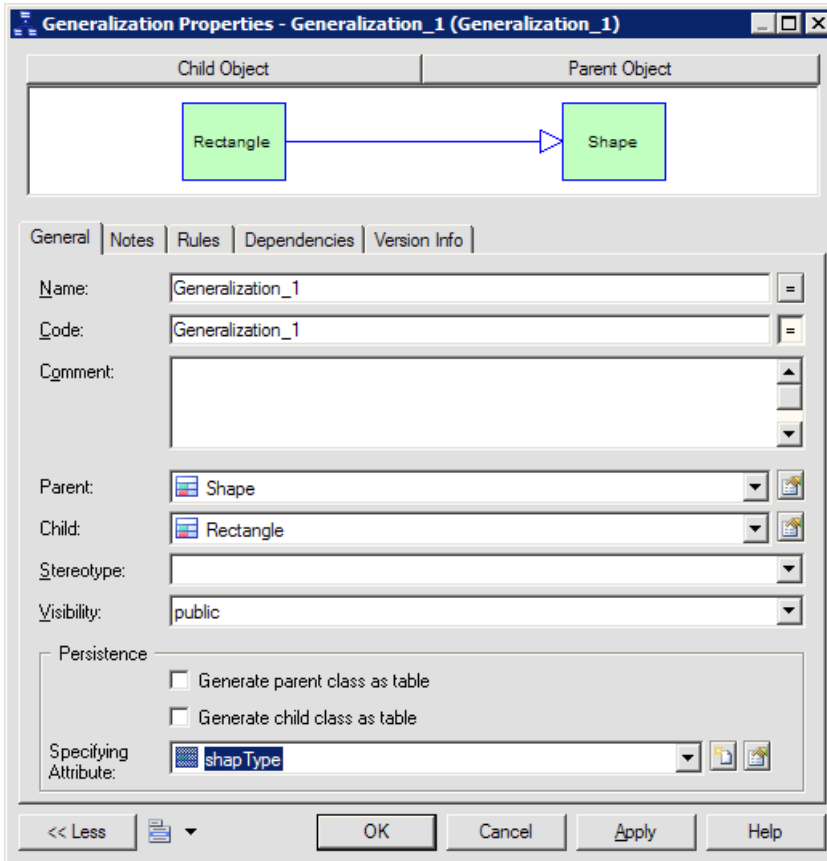
The whole class hierarchy is mapped to one table. One character based type or integer type discriminator column is defined to distinguish instances of difference classes in the hierarchy.



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:



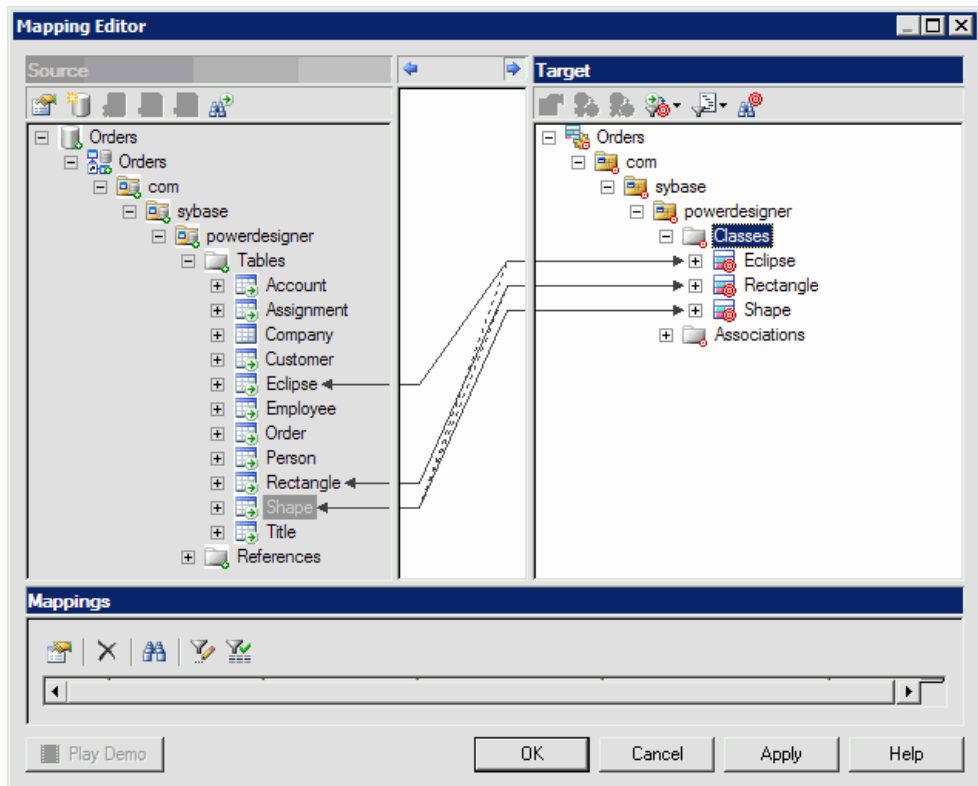
2. Define identifier mapping in the root class.
3. Define attribute mappings or association mappings for each class.
4. Select one of the attributes in the root class, as the Specifying Attribute in the property sheet of one of the children inheritance links of the root class to specify it as a discriminator 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*.

Joined Subclass Inheritance Mapping Strategy

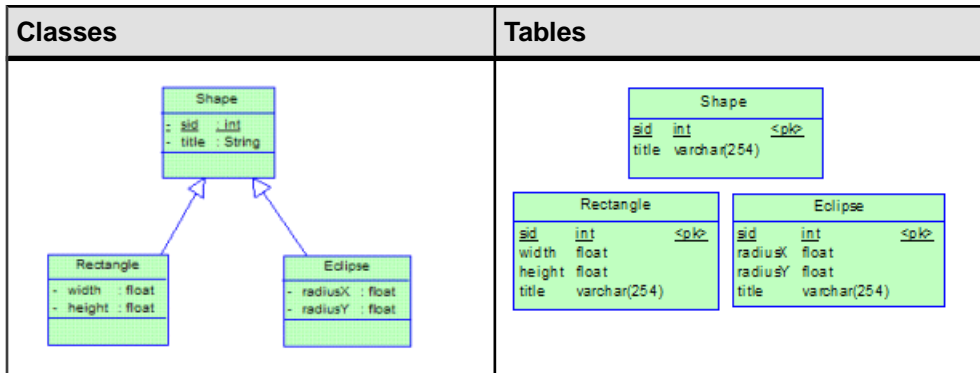
Each entity class is mapped to its own primary table. Each primary table has a reference key referring to a primary table of its parent class except for the primary table of the root class. The reference key should join on the primary key of the primary table.



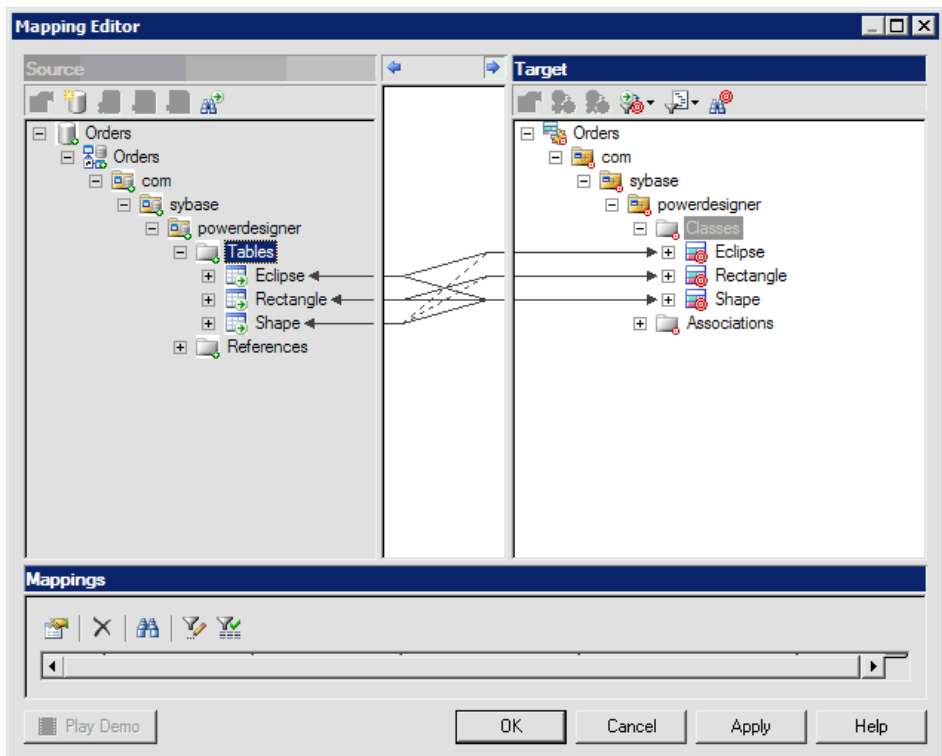
1. Define class mappings for each class in the hierarchy. Each class is mapped to its own primary table.
2. Define identifier mapping in the root class.
3. Define attribute mappings or association mappings for each class.
4. Define persistence generation type for each class.
5. Define persistence generation type of all the classes as *Generate table*.

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.



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*.

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.

PowerDesigner supports the generation of persistent objects for Hibernate and EJB3, as well as JavaServer Faces for Hibernate.

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 **Import** 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.

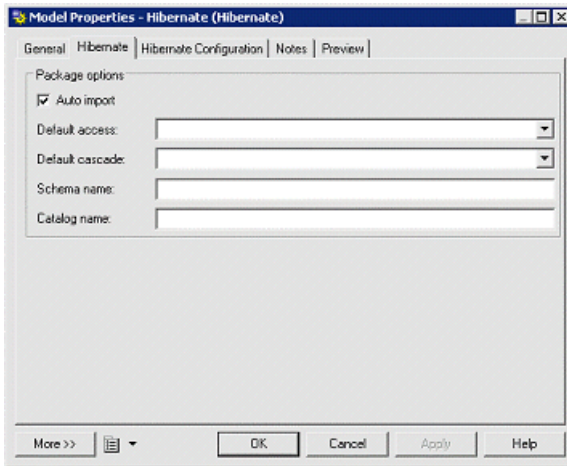
PowerDesigner supports the design of Java classes, database schema and Object/Relational mapping (O/R mapping). Using these metadata, PowerDesigner can generate Hibernate persistent objects including:

- Persistent Java classes (domain specific objects)
- Configuration file
- O/R mapping files
- DAO factory
- Data Access Objects (DAO)
- Unit test classes for automated test

Defining the Hibernate Default Options

You can define these options in the model or a package property sheet.

1. Open the model or a package property sheet, and click the `hibernate` tab:



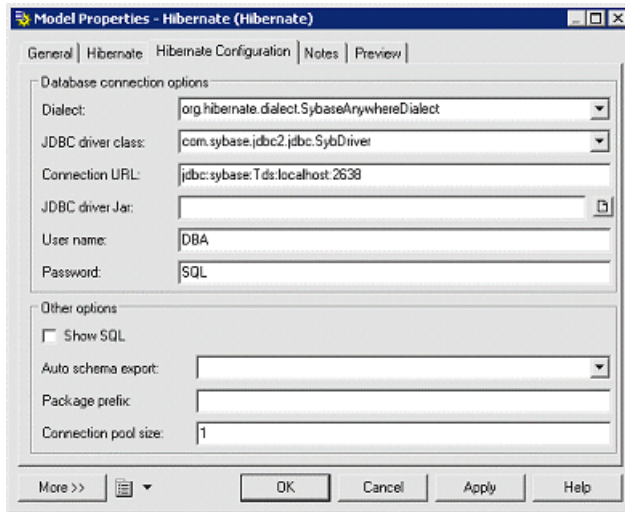
2. Define the model or package level default options.

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.

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`.

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.

Option	Description
Dialect	Specifies the dialect, and hence the type of database. Hibert Tag: dialect
JDBC driver class	Specifies the JDBC driver class. Hibert Tag: connection.driver_class
Connection URL	Specifies the JDBC connection URL string. Hibert Tag: connection.url
JDBC driver jar	Specifies the JDBC driver jar file path. Hibert Tag: N/A
User name	Specifies the database user name. Hibert Tag: connection.username
Password	Specifies the database user password. Hibert Tag: connection.password
Show SQL	Specifies that SQL statements should be shown in the log. Hibert Tag: show_sql

Option	Description
Auto schema export	Specifies the mode of creation from tables. Hibert Tag: hbm2ddl.auto
Package prefix	Specifies a namespace prefix for all the packages in the model. Hibert Tag: N/A
Connection pool size	Specifies the maximum number of pooled connections. Hibert Tag: connection.pool_size

You can verify the configuration parameters in the Preview tab.

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* on page 587 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.

Defining Hibernate Class Mapping Options

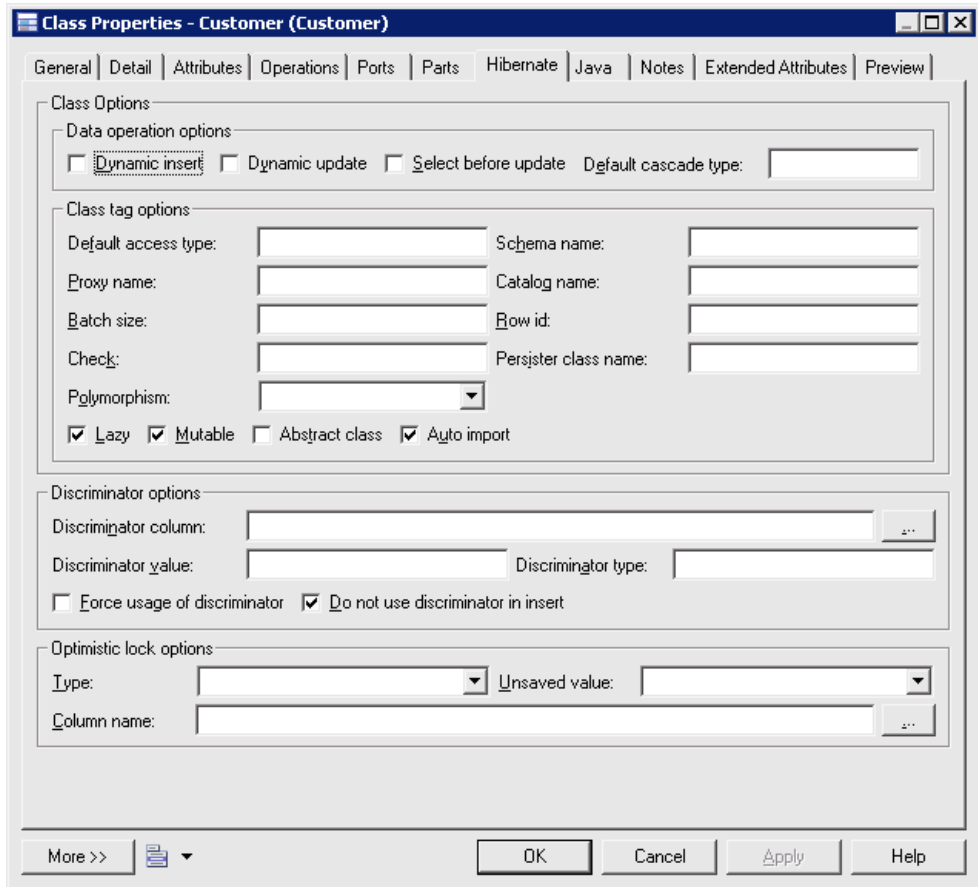
Classes can be mapped to tables or views. Since views have many constraints and limited functionality (for example they do not have primary keys and reference keys), some views cannot be updated, and the mappings may not work properly in some cases.

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	Specifies that INSERT SQL should be generated at runtime and will contain only the columns whose values are not null. Hibernate Tag: dynamic-insert
Dynamic update	Specifies that UPDATE SQL should be generated at runtime and will contain only the columns whose values have changed. Hibernate Tag: dynamic-update

Option	Description
Select before update	Specifies that Hibernate should never perform a SQL UPDATE unless it is certain that an object is actually modified. Hibernate Tag: select-before-update
Default cascade type	Specifies the default cascade style. Hibernate Tag: default-cascade
Default access type	Specifies the default access type (field or property) Hibernate Tag: default-access
Proxy name	Specifies an interface to use for lazy initializing proxies. Hibernate Tag: proxy
Batch size	Specifies a "batch size" for fetching instances of this class by identifier. Hibernate Tag: batch-size
Check	Specifies a SQL expression used to generate a multi-row check constraint for automatic schema generation. Hibernate Tag: check
Polymorphism	Specifies whether implicit or explicit query polymorphism is used. Hibernate Tag: polymorphism
Schema name	Specifies the name of the database schema. Hibernate Tag: schema
Catalog name	Specifies the name of the database catalog. Hibernate Tag: catalog
Row id	Specifies that Hibernate can use the ROWID column on databases which support it (for example, Oracle). Hibernate Tag: rowed
Persister class name	Specifies a custom persistence class. Hibernate Tag: persister
Lazy	Specifies that the class should be lazy fetching. Hibernate Tag: lazy
Mutable	Specifies that instances of the class are mutable. Hibernate Tag: mutable
Abstract class	Specifies that the class is abstract. Hibernate Tag: abstract

Option	Description
Auto import	Specifies that an unqualified class name can be used in a query Hibernate Tag: Auto-import
Discriminator column	Specifies the discriminator column or formula for polymorphic behavior in a one table per hierarchy mapping strategy. Hibernate Tag: discriminator
Discriminator value	Specifies a value that distinguishes individual subclasses, which are used for polymorphic behavior. Hibernate Tag: discriminator-value
Discriminator type	Specifies the discriminator type. Hibernate Tag: type
Force usage of discriminator	Forces Hibernate to specify allowed discriminator values even when retrieving all instances of the root class. Hibernate Tag: force
Do not use discriminator in insert	Forces Hibernate to not include the column in SQL INSERTs Hibernate Tag: insert
Optimistic lock type	Specifies an optimistic locking strategy. Hibernate Tag: optimistic-lock
Optimistic lock column name	Specifies the column used for optimistic locking. A field is also generated if this option is set. Hibernate Tag: version/ timestamp
Optimistic lock unsaved value	Specifies whether an unsaved value is null or undefined. Hibernate Tag: unsaved-value

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:

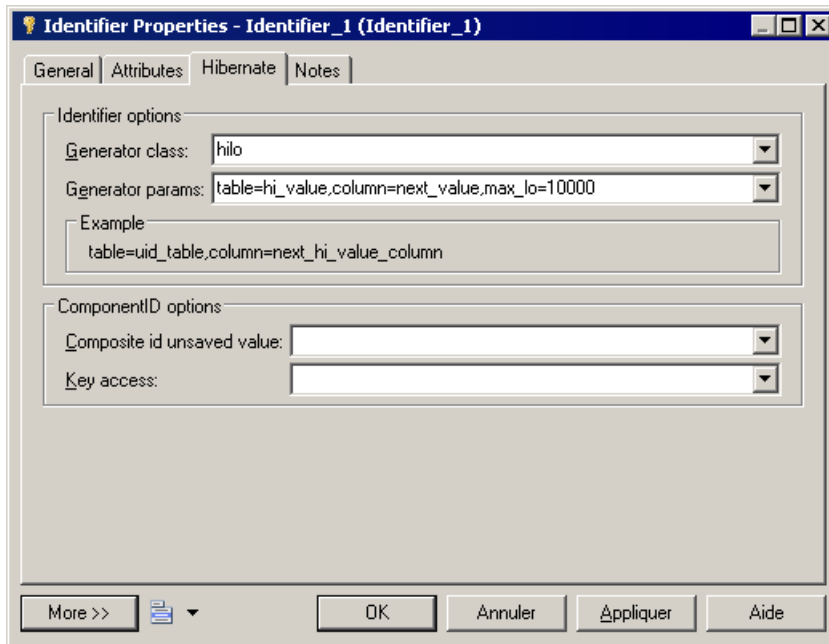
- Simple identifier mapping
- Composite identifier mapping
- Component identifier mapping

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.

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.

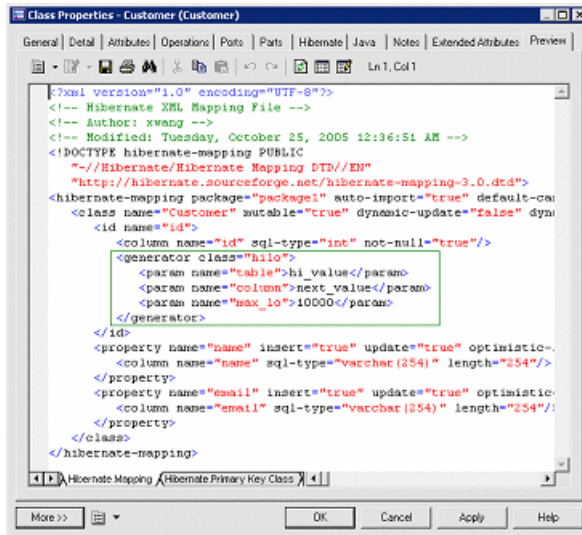
You can define the generator class and parameters in the Hibernate tab of the primary identifier property sheet. The parameters take the form of param1=value1; param2=value2.



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:

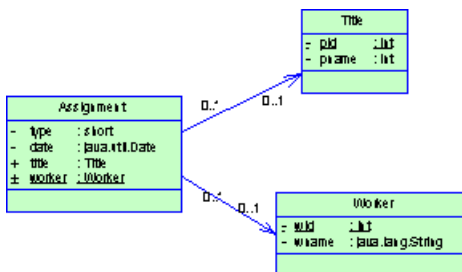


Note that, if there are several Primary identifier attributes, the generator is not used.

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.

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.



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>
```

```

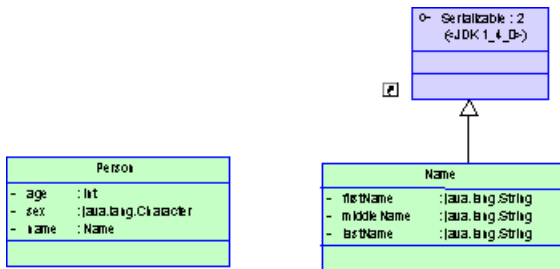
<key-many-to-one name="title">
</key-many-to-one>
<key-many-to-one name="worker">
</key-many-to-one>
</composite-id>

```

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.

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.



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.

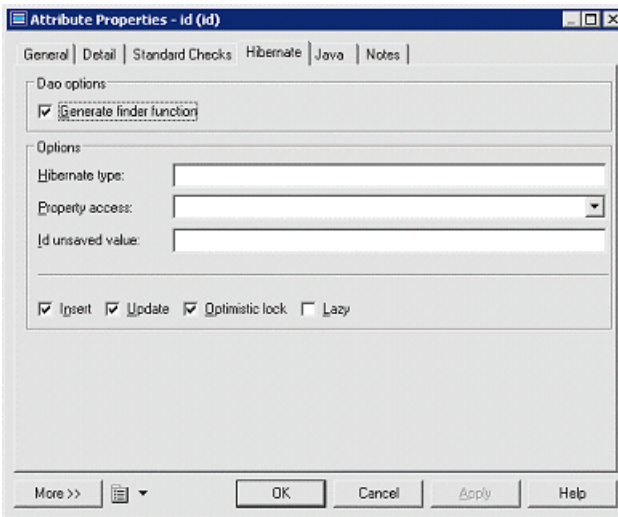
Defining Attribute Mappings

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.

Option	Description
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).

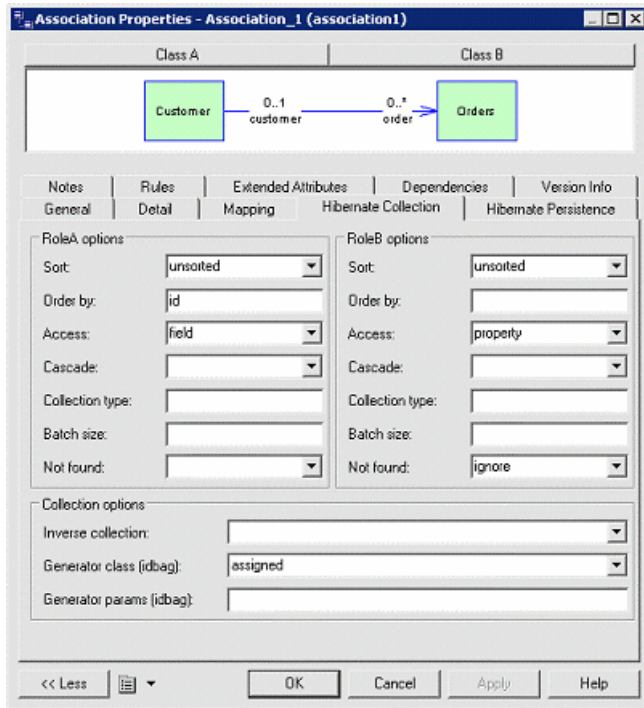
Hibernate Association Mappings

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.

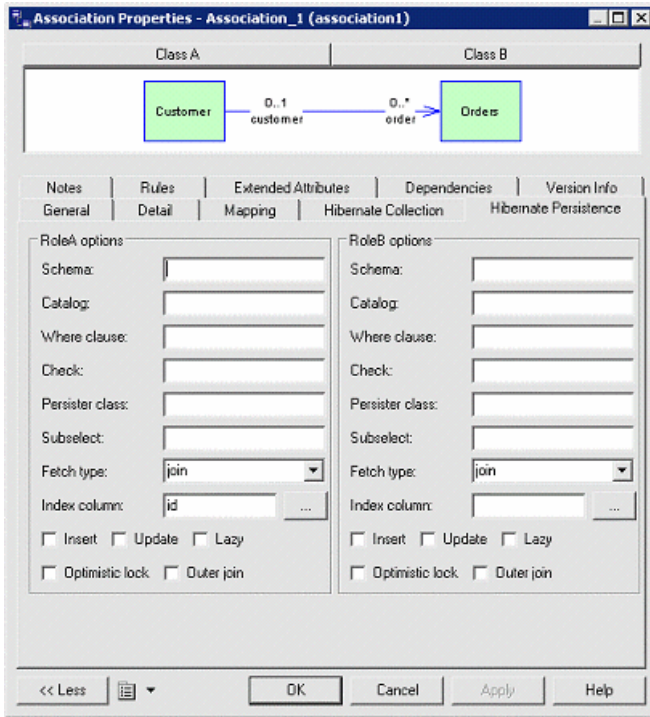
Defining Hibernate Association Mapping Options

You define Hibernate association mapping options as follows:

1. Open the Association property sheet and click the Hibernate Collection tab.
2. Define the collection management options (see *Collection management options* on page 584).



3. Select the Hibernate Persistence tab. Define persistence options (see *Persistence options* on page 585).



Collection Management Options

The following options are available:

Field	Description
Sort	Specifies a sorted collection with natural sort order, or a given comparator class. Hibernate Tag: sort
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. Hibernate Tag: order-by
Access	Specifies the strategy Hibernate should use for accessing the property value. Hibernate Tag: access
Cascade	Specifies which operations should be cascaded from the parent object to the associated object. Hibernate Tag: cascade
Collection type	Specifies a name that indicates the Hibernate type. Hibernate Tag: type

Field	Description
Batch size	Specifies the batch load size. Hibernate Tag: batch-size
Not found	Specifies how foreign keys that reference missing rows will be handled: ignore will treat a missing row as a null association. Hibernate Tag: not-found
Inverse collection	Specifies that the role is the inverse relation of the opposite role. Hibernate Tag: inverse

Persistence Options

The following options are available:

Field	Description
Schema	Specifies the name of the schema. Hibernate Tag: schema
Catalog	Specifies the name of the catalog. Hibernate Tag: catalog
Where clause	Specifies an arbitrary SQL WHERE condition to be used when retrieving objects of this class. Hibernate Tag: where
Check	Specifies a SQL expression used to generate a multi-row check constraint for automatic schema generation. Hibernate Tag: check
Fetch type	Specifies outer-join or sequential select fetching. Hibernate Tag: fetch
Persister class	Specifies a custom persistence class. Hibernate Tag: persister
Subselect	Specifies an immutable and read-only entity to a database subselect. Hibernate Tag: subselect
Index column	Specifies the column name if users use list or array collection type. Hibernate Tag: index
Insert	Specifies that the mapped columns should be included in any SQL INSERT statements. Hibernate Tag: insert

Field	Description
Update	Specifies that the mapped columns should be included in any SQL UPDATE statements. Hibernate Tag: update
Lazy	Specifies that this property should be fetched lazily when the instance variable is first accessed. Hibernate Tag: lazy
Optimistic lock	Specifies that a version increment should occur when this property is dirty. Hibernate Tag: optimistic-lock
Outer join	Specifies to use an outer-join. Hibernate Tag: outer-join

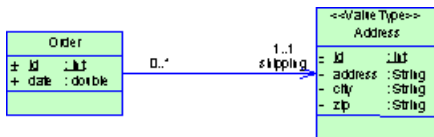
Mapping Collections of Value Types

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.

Mapping Collections of Value Type

You define mapping collections of value type as follows:

1. Create an entity type class.
2. Create another class for value type.
3. Open the property sheet of the class, click the Detail tab, and select the Value type radio button.
4. 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.



5. Generate the PDM with O/R mapping.
6. Open the property sheet of the entity class and click the Preview tab.
7. Verify the mapping file.

A composite entity class may contain components, using the <nested-composite-element> declaration.

Defining Association Collection Type for One-to-many or Many-to-many Associations

You define association collection type for one-to-many or many-to-many associations as follows:

1. Open the association property sheet and click the Detail tab.
2. Specify a Multiplicity on both sides.
3. Specify either unidirectional or bi-directional navigability.
4. Specify role names if necessary.
5. If one role of the association is navigable and the multiplicity is many, you can set the collection container type and batch loading size.
6. 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
<None>	array
<code>java.util.Collection</code>	bag or idbag (many-to-many)
<code>java.util.List</code>	list
<code>java.util.Set</code>	set

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.

Generating Code for Hibernate

Before generating code for Hibernate, you need to:

- Install Hibernate 3.0 or higher.
- Check the model.
- Define generation options.

Checking the Model

When you complete the definition of the model, you need to run the Check Model function to verify if there are errors or warnings in the model. If there are errors, you need to fix them before generating code.

Defining Generation Options

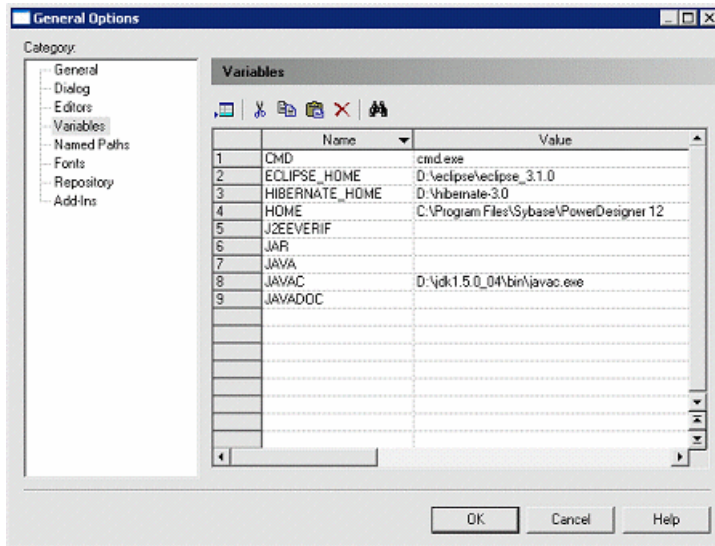
There are two types of generation options:

- Environment variables - to allow your Eclipse or Ant build script to find the Hibernate library Jar files
- Generation options

Defining Environment Variables

You define environment variables as follows:

1. Select **Tools > General Options**.



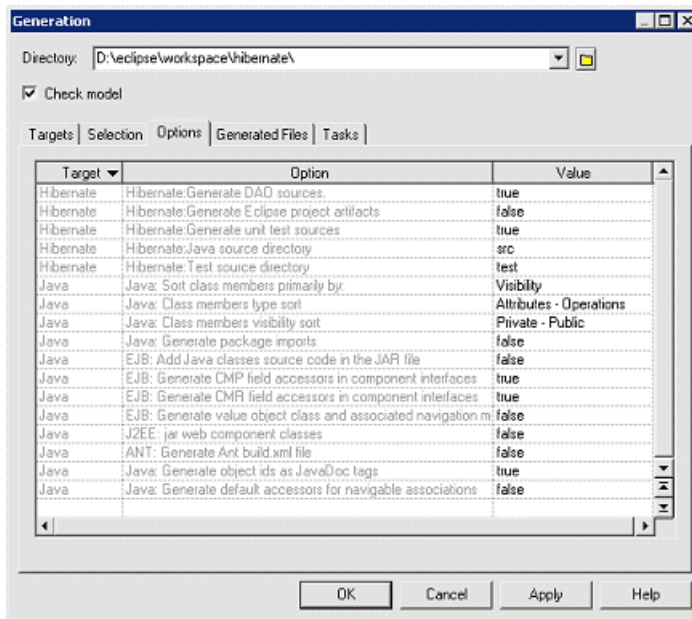
2. Select the Variables node.
3. Add a variable HIBERNATE_HOME and, in the value field, enter the Hibernate home directory path. For example, D:\Hibernate-3.0.

Defining Generation Options

You define generation options as follows:

1. Select **Language > Generate Java Code**.
2. Specify the root directory where you want to generate the code.

3. Click the Options tab.



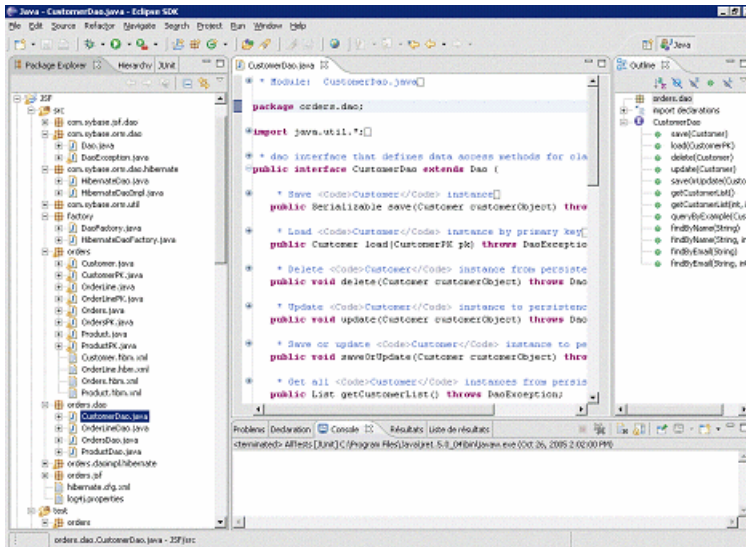
4. [optional] To use DAO, set the Generate DAO sources option to true.
5. [optional] To use Eclipse to compile and test the Java classes, set the Generate Eclipse project artifacts option to true.
6. [optional] To use unit test classes to test the Hibernate persistent objects, set the Generate unit test sources option to true.
7. Click on OK to generate code immediately or Apply and then Cancel to save your changes for later.

Generating Code for Hibernate

Once you have completed your model, checked it, and defined your generation options, you can generate the code for Hibernate.

1. Select **Language > Generate Java Code**.
2. [optional] Click the Select tab to change the object selection.
3. [optional] Click the Options tab to change the Hibernate and Java generation options.
4. [optional] Click the Generated Files tab to review all the files that will be generated.
5. Click OK.

You can use an IDE like Eclipse to modify the generated code, compile, run the unit test and develop your application.



Using the Generated Hibernate Code

To use Eclipse, you need to download and install the Eclipse SDK.

Importing the Generated Project into Eclipse

If you have selected the Generate Eclipse project artifacts generation option, you can import the generated project into Eclipse and use Eclipse to modify, compile and run the tests.

If you use the PowerDesigner Eclipse plugin then, after the code generation, the project is automatically imported or refreshed in Eclipse.

If you use the standalone version of PowerDesigner, you need to import the generated project as follows:

1. In Eclipse, select **File > Import**
2. In the import list, select 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.
 - That the JDK version is the right one. If you use Java 5.0 as the language in OOM, you need to use the JDK 5.0 to compile the code.

Performing the Unit Tests

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 to create and update objects.

After creating, updating, deleting or finding objects, a test asserts that the result is as expected.

If the result is as expected, the test succeeds; otherwise it fails.

Before running the unit tests, you need to:

1. Create the database file.
2. Define an ODBC connection.
3. Generate the database from the PDM using the ODBC connection.
4. Give the test user the permission to connect to the database.
5. Start the database.

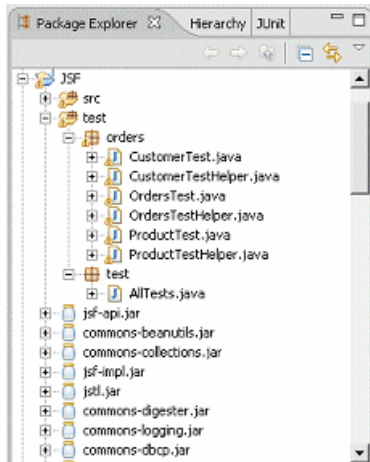
Running Unit Tests in Eclipse

Eclipse integrates JUnit. The JUnit Jar files and JUnit user-interface are provided.

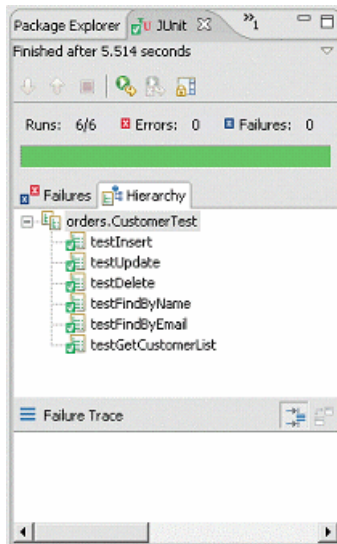
Running a Single Test Case

You run a single test case as follows:

1. Open the Java perspective
2. In the Package Navigator, expand the test package



3. Right-click on a test case (for example, CustomerTest.java) and select **Run As > JUnit Test**.
4. Select the JUnit view to verify the result:



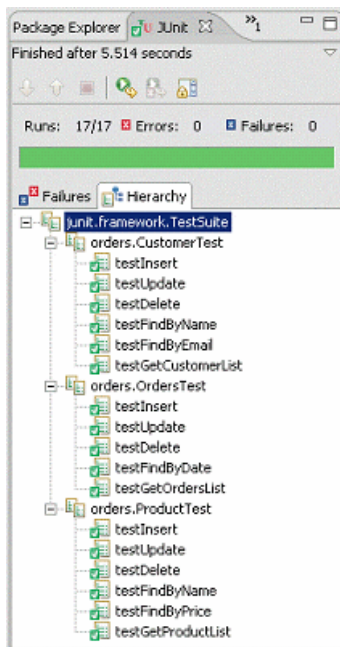
If there are 0 errors, then the test has succeeded. If there are errors, you need to check the Console view to locate the sources of them. The problem could be:

- The database is not started.
- The user name or password is wrong.
- The database is not generated.
- The mapping is wrong.

Running the Test Suite

You run the test suite as follows

1. Open the Java perspective
2. In the Package Navigator, expand the test package
3. Right-click on the AllTests.java test suite and select **Run As > JUnit Test**.
4. Select the JUnit view to verify the result



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.

To use Ant, you need to:

- Download it from <http://www.apache.org> and install it.
- Define an environment variable ANT_HOME and set it to your Ant installation directory.
- Copy junit-3.8.1.jar from HIBERNATE_HOME/lib directory to ANT_HOME/lib directory.
- Make sure that the Hibernate Jar files are defined in the build.xml file or in the CLASSPATH environment variable.
- Make sure that the JDBC driver Jar file of your database is defined in the build.xml file or in the CLASSPATH environment variable.

Running Unit Tests with Ant from PowerDesigner:

You run unit tests with Ant from PowerDesigner as follows:

1. Select **Language > Generate Java Code**.
2. Select the Options tab.
3. Set the Generate Ant build.xml file option to true.
4. Select the Tasks tab.

5. Check the Hibernate: Run the generated unit tests task.
6. Click OK.
7. After you close the generation files list window, the JUnit task runs. You can see the result in output window.

```

compiletest:
[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_InnerInnerClass Test
[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

```

Running Unit Tests with Ant from the Command Line

You run unit tests with Ant from the command line as follows:

1. Open a command line window.
2. Go to the directory where you have generated the code.
3. Run the JUnit test task: Ant junit
4. Check the output result.

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 **Import** 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

Generating Entities for EJB 3.0

You generate entities for EJB 3.0 as follows:

1. Create an OOM and a PDM, and then define your O/R mappings. For detailed information, see *Chapter 22, Object/Relational (O/R) Mapping* on page 539.
2. Define the EJB 3 persistence settings.
3. Generate Java code.
4. Run unit tests.

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* on page 541).
- 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.

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 *Chapter 22, Object/Relational (O/R) Mapping* on page 539.

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.
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 for information on how to map one Entity class to multiple tables, see *Chapter 22, Object/Relational (O/R) Mapping* on page 539.

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; }
```

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() {}
    .....
}
```


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 *Chapter 22, Object/Relational (O/R) Mapping* on page 539.

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 `OneToOne` 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)
@JoinColumn({
    @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 *Chapter 22, Object/Relational (O/R) Mapping* on page 539.

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 `OneToMany` 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
@JoinColumn({
    @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 *Chapter 22, Object/Relational (O/R) Mapping* on page 539.

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 *Chapter 22, Object/Relational (O/R) Mapping* on page 539.

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.
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.

Defining EJB 3 Inheritance Mappings

EJB 3 persistence supports all three popular inheritance mapping strategies and also mixed strategies.

- Table per class hierarchy - SINGLE_TABLE
- Joined subclass - JOINED
- Table per concrete class - TABLE_PER_CLASS

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.

Mapped Superclasses

In EJB 3.0, Mapped superclasses are used to define state and mapping information that is common to multiple entity classes. They are not mapped to separate tables of their own. You cannot currently define Mapped superclasses in PowerDesigner.

Table Per Class Hierarchy Strategy

In this strategy, 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.

Option	Description
Discriminator value	Specifies a 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.

For more informations about mapping, see *Chapter 22, Object/Relational (O/R) Mapping* on page 539.

```
@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 Strategy

In this strategy, 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.

For more informations about mapping, see *Chapter 22, Object/Relational (O/R) Mapping* on page 539.

```
@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.

Applying Table Per Class Strategy

In this strategy, 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 { ... }

```

Defining EJB 3 Persistence Default Options

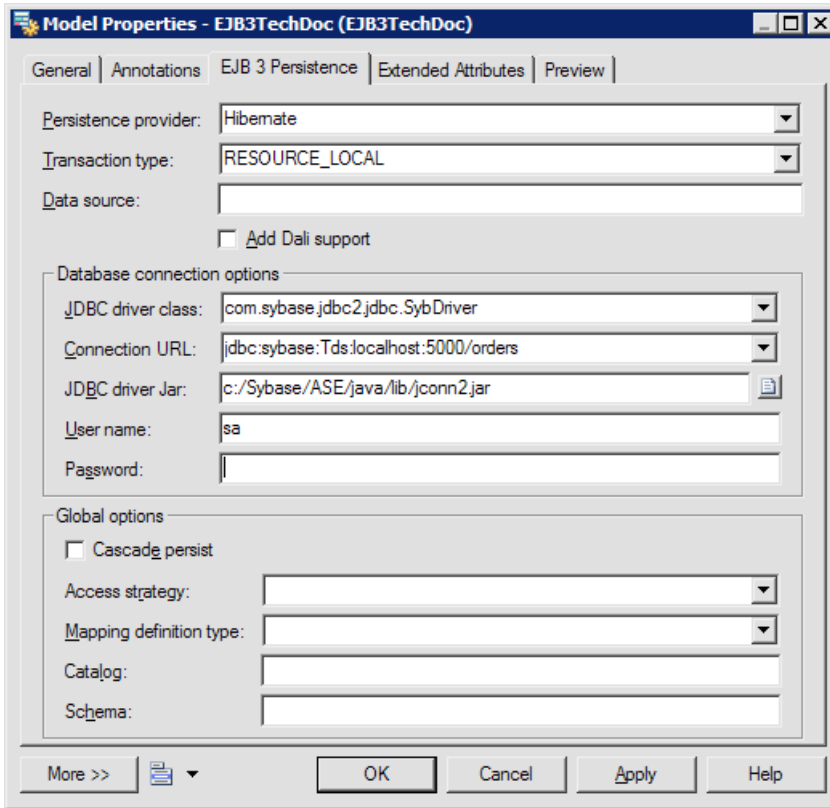
The following default persistent options can be set at the model, package or class level:

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.

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.

1. Open EJB 3 Persistence Configuration form from 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. 3. Define JDBC driver class, connection URL, JDBC driver jar file path, user name and password.

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.

Option	Description
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.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.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.SybD
river</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>
```

Checking the Model

To keep the model and mappings correct and consistent, you need to run a model check. If there are errors, you need to fix them. You can also run model checking before code generation.

Generating Code for EJB 3 Persistence

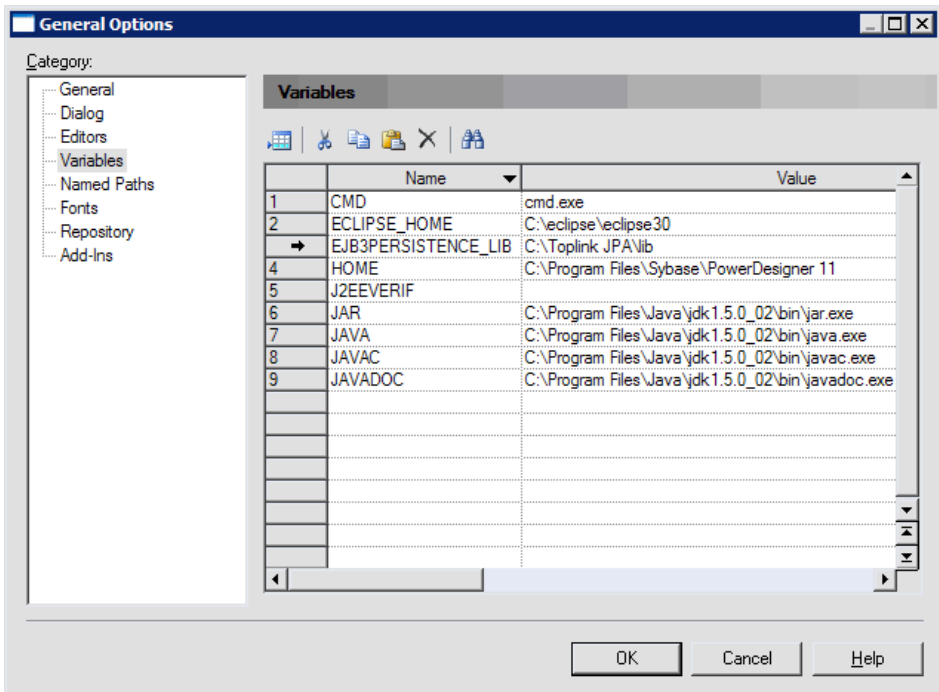
In order to generate code for EJB 3 persistence, you must.

- Download an EJB 3 persistence provider such as Hibernate Entity Manager, Kodo, TopLink and GlassFish.
- Define an environment variable to specify the location of the persistence library directory.
- Generate code - define model selection and generation options and preview generated file list.
- Run unit test.

Defining the Environment Variable

PowerDesigner uses an environment variable to generate library configuration for Eclipse project or Ant build script.

1. Select Tools >> **Generation Options**
2. Select the Variables node



3. Add a variable EJB3PERSISTENCE_LIB and, in the value field, enter directory path which you put your persistence provider libraries, for example D:\EJB 3.0\Hibernate Entity Manager\lib.

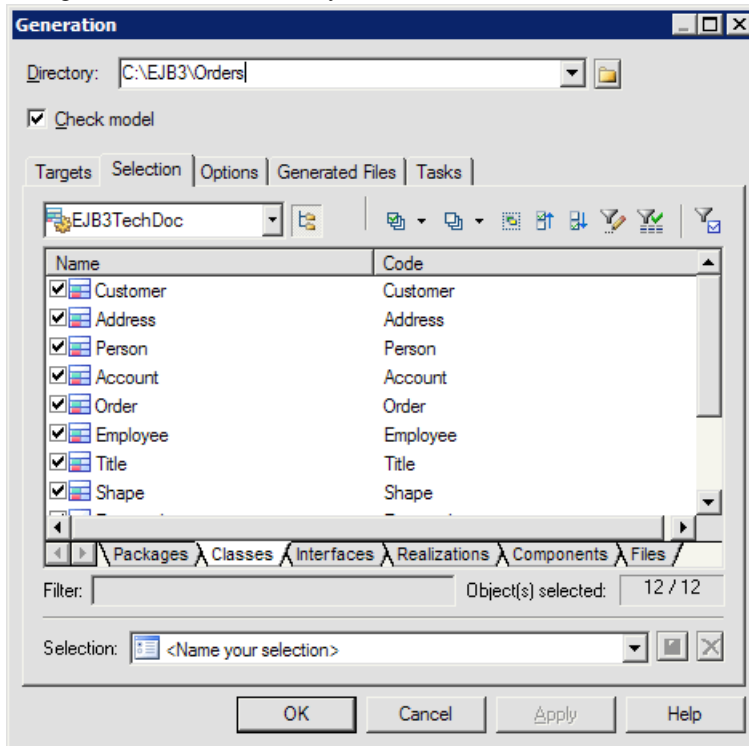
You can also define this as a Windows system environment variable, but you need to restart PowerDesigner to have it take effect.

Generate Code

To generate code, select **Language > Generate Java code** (Ctrl + G). Specify the directory where you want to put your generated code. On the Targets tab, make sure the O/R Mapping target is selected.

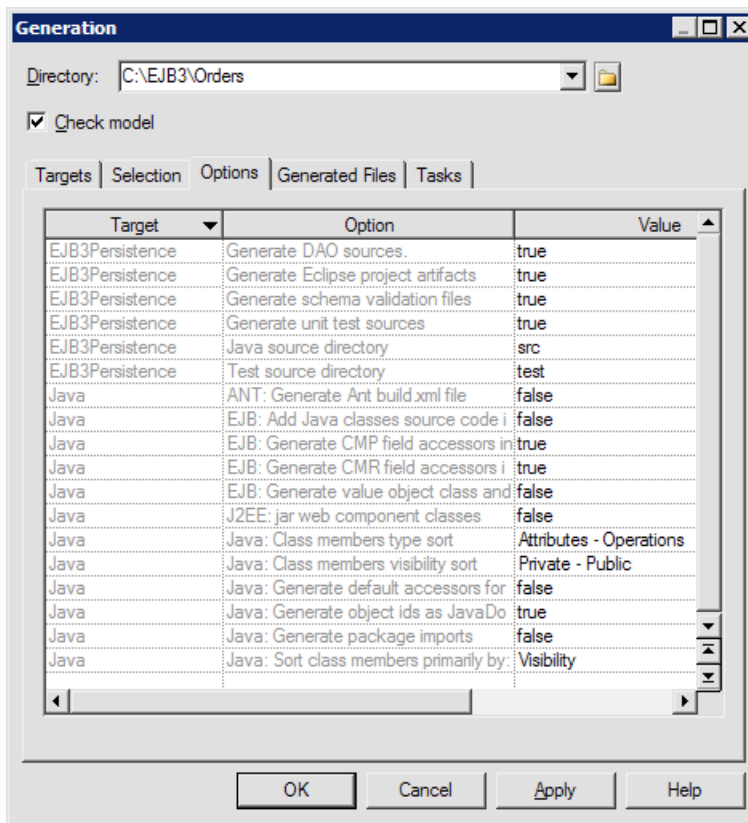
Select Model Elements

Select model elements to be generated on the Selection tab. The model should be selected for there are some important artifacts generated on the model level, such as persistence configuration file, DAO factory classes, DAO base classes etc.



Define Generation Options

- Define generation options on the Options tab.



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.
Generate Ant build.xml file	Specifies whether Ant build.xml file should be generated.

Preview Generated File List

You can get a preview of generated file list on the Generated files tab.

Specify Post Generation Tasks

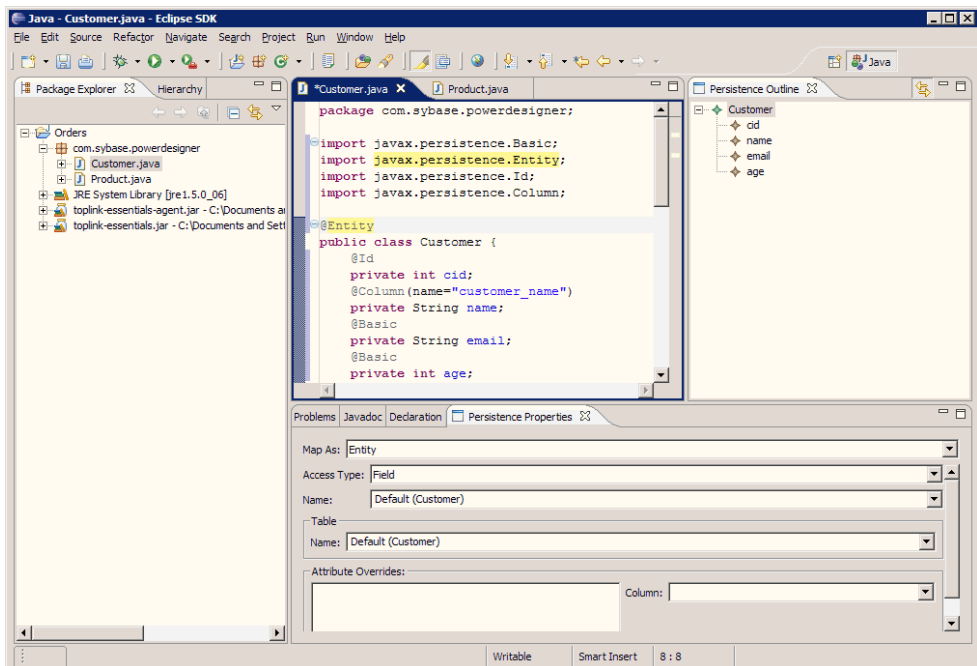
There are some tasks that can be run after the generation. You can select them on Tasks tab. The useful one is Run generated unit tests. PowerDesigner will run unit tests by Ant script generated after generation if you select the task. You can also run it on the command window. There are some prerequisites before you can run the task. We will show you how to run unit tests in the coming section.

Authoring in Dali Tools

Dali JPA Tools provide support for the definition, editing, and deployment of Object-Relational (O/R) mappings for EJB 3.0 Entity Beans. It simplifies mapping definition and editing through:

- Creation and automated mapping wizards
- Intelligent mapping assistance
- Dynamic problem identification

You can import generated Eclipse project and do further editing in Dali tools if you had selected Add Dali support in model's property sheet.



Run Unit Tests

There are two ways you can run unit tests generated. One is running Ant task. The other is running them in Eclipse.

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 options tab.

To use Ant, you need to:

- Download Ant from <http://www.apache.org> and install it.
- Define an environment variable ANT_HOME and set it to your Ant installation directory.
- Download junit-3.8.1.jar if you don't have it.
- Copy junit-3.8.1.jar to \$ANT_HOME/lib directory.
- Make sure that you have defined database connection parameters and JDBC driver jar correctly.

Running Unit Tests with Ant from PowerDesigner

You can run unit tests with Ant from PowerDesigner.
Select the Run unit tests task when generating code.

Running Unit Tests with Ant from the Command Line

You can run unit tests with Ant from the command line.

1. Open a command line window.
2. Go to the directory where you have generated the code.
3. Run the JUnit test task by issuing command: Ant junit
4. Check the output result in ejb3-persistence.log and testout directory.

Running Unit Test in Eclipse

To use Eclipse, you need to download and install the Eclipse SDK.

If you have selected the Generate Eclipse project artifacts generation option, you can import the generated project into Eclipse and use Eclipse to modify, compile and run the tests.

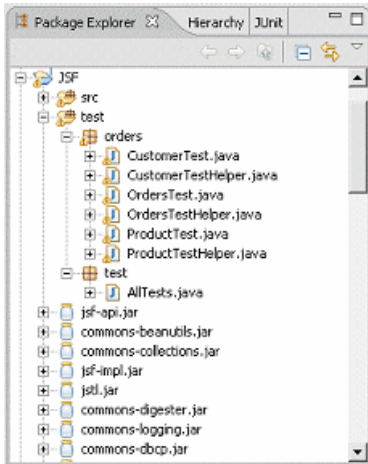
If you use the PowerDesigner Eclipse plugin, after the code generation, the project is automatically imported or refreshed in Eclipse.

You can run a single test case each time or run them as suite.

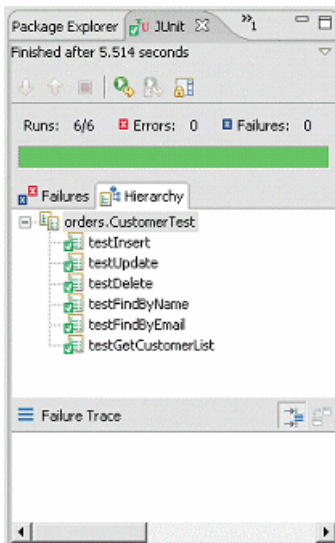
Running a Single Test Case

You can run a single test case.

1. Open the Java perspective
2. In the Package Navigator, expand the test package



3. Select a test case class, for example CustomerTest.java and run it as Unit Test
4. Select the JUnit view to verify the result:



Running the Test Suite

You can run test cases as a suite.

1. Select AllTests class under test package.
2. Run it as Application. All the tests will be run as suite.

Generated File List

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 - Primary key classes are generated to ease find-by-primary-key operation. It is also mandatory for composite primary key.
- EJB 3 Configuration File - The EJB 3 persistence configuration file persistence.xml is generated in META-INFO sub directory of Java source directory.
- Log4J Configuration File - PowerDesigner uses Log4j as the default logging framework to log messages. The Log4j properties file log4j.properties is generated in Java source directory.
- Utility Class - The Util.java class contains some utility functions that are used by unit tests, such as compare date by precision. It is defined in the com.sybase.orm.util package.
- EJB 3 O/R Mapping Files - If mapping definition type specified includes mapping file, 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 - generated to help user perform test to prove that:
 - The mappings are correctly defined
 - The CRUD (Create, Read, Update and Delete) work properly
 - The find methods work
 - The navigations work

Unit test classes include:

- Test helper classes - provide some utility functions for unit test classes, such as creating new instances, modifying state of instances, saving instances etc.
- Unit Test Classes - For each persistent entity, PowerDesigner generates a unit test class. The generated test cases are:
 - Insert test method - to test instance insert.
 - Update test method - to test instance update.
 - Delete test method - to test instance delete.
 - Property finder test methods - to test every property finder method defined in Dao.
 - Get all instance list test method - to test get all instances method.
 - Navigation test method - to test association mapping.
 - Inheritance test method - to test inheritance mapping.

- User defined operation test methods - skeleton test methods for user defined functions.
- AllTest Class - a test suite that runs all the unit test cases.
- Ant Build File - PowerDesigner can generate an Ant build file to help you to compile and run unit tests if you set the Generate Ant build.xml file option to true in the Java code generation window. The Ant build.xml contains customized elements for EJB 3:
 - Custom properties - that specify directories and class path.
 - Custom target definitions - to define JUnit tasks.
 - Custom tasks - to run JUnit test or generate JUnit test reports.

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 **Import** 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.

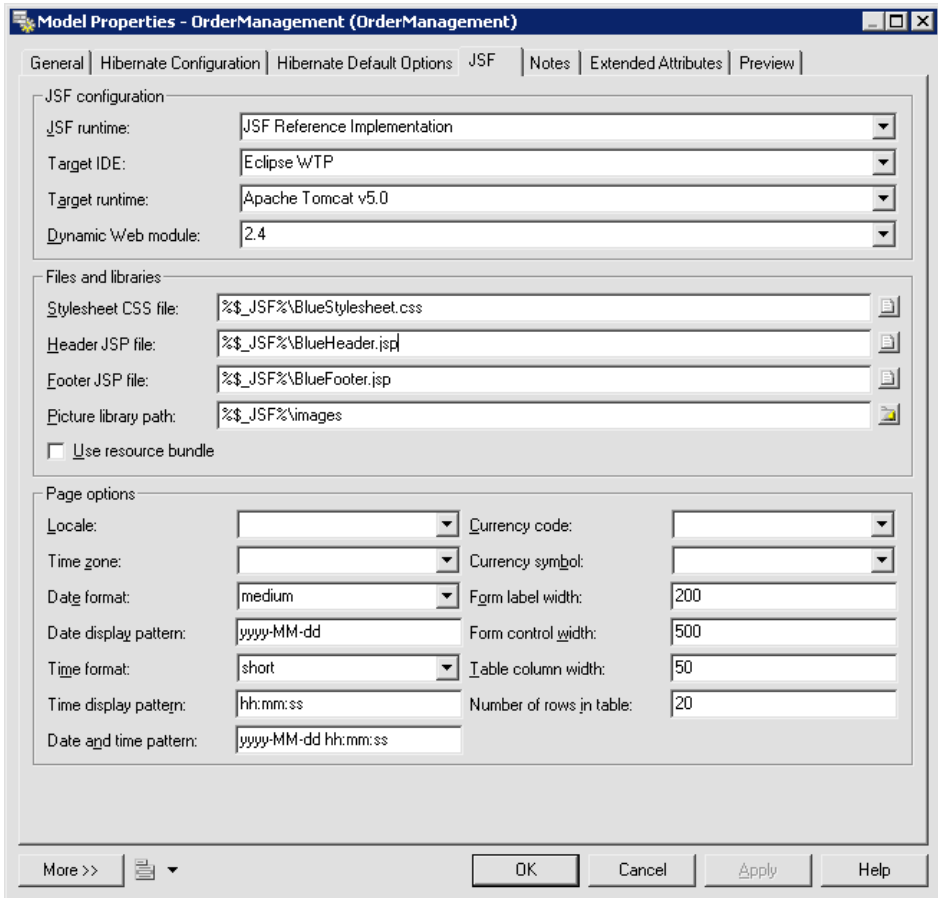
Defining Global Options

Each page could use a style sheet, a header file and a footer file to define its standard presentation.

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.

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.

The following options are available:

Option	Description
JSF runtime	Specifies the JSF runtime. It can be JSF Reference Implementation or Apache My Faces.
Target IDE	Specifies the target IDE. List/Default Values: Eclipse WTP, Sybase WorkSpace

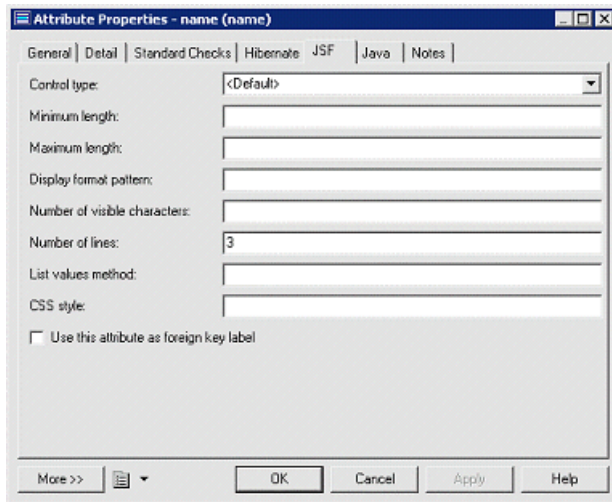
Option	Description
Target runtime	Specifies the target runtime. List/Default Values: Apache Tomcat v5.0, Sybase EAServer v5.x, etc.
Dynamic Web Module	Specifies the dynamic web module's version List/Default Values: 2.2, 2.3, and 2.4.
Stylesheet CSS File	Specifies the stylesheet file for JSF pages. List/Default Values: %\$_JSF%\stylesheet.css
Header JSP file	Specifies the header file for JSF pages. List/Default Values: %\$_JSF%\header.jsp
Footer JSP file	Specifies the footer file for JSF pages. List/Default Values: %\$_JSF%\footer.jsp
Picture library path	Specifies the path that contains pictures for JSF pages. 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 Default: short
Date display pattern	Specifies the date pattern
Time format	Specifies the date format could be default, short, medium, long, full Default: short
Time display pattern	Specifies the time pattern
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 Default: 200

Option	Description
Form control width	Specifies the width of the control in pixel in a form Default: 500
Table column width	Specifies the width of the control in pixel in a table Default: 50
Table number rows	Specifies the number of rows that can be displayed in a table Default: 20

Defining Attribute Options

You can define attribute-level options for validation or presentation style.

1. Open Attribute Property sheet, select JSF tab.
2. Define Attribute options.

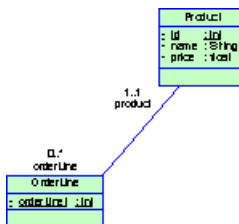


Option	Description
Control type	Specifies the type of control. Note: You should select the type that can support the Attribute Java type. <ul style="list-style-type: none"> • String - EditBox, MultilineEdit • Boolean - CheckBox • Date - Date, Time, DateTime, Year, Month, Day • <Contains List of Value> - ListBox, ComboBox, RadioButtons

Option	Description
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	Specifies the number of lines for multiline edit control Default: 3
List values method	Specifies the method that provides the list of values for ListBox, ComboBox or radioButtons.
CSS style	Specifies the CSS formatting style
Use the attribute as foreign key label	Specifies that the column associated to the attribute will be used as the foreign key label for the foreign key selection. If no FK label column is defined, PowerDesigner will choose the first not-PK and non FK column for the default label column. Default: False

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: 1

Quantity: 10

Orders: 1

Product: PowerDesigner

Create Cancel

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.

Attribute Validation Rules and Default Values

PowerDesigner can generate validation and default values for the edit boxes in the Create and Edit pages.

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.

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.

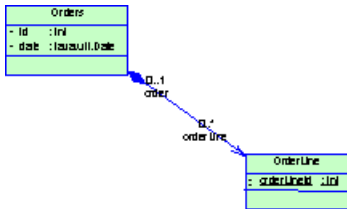
You can also select which control style to use:

- Combobox
- Listbox
- Radio buttons (if the number of values is low) For example: Mr. Ms.

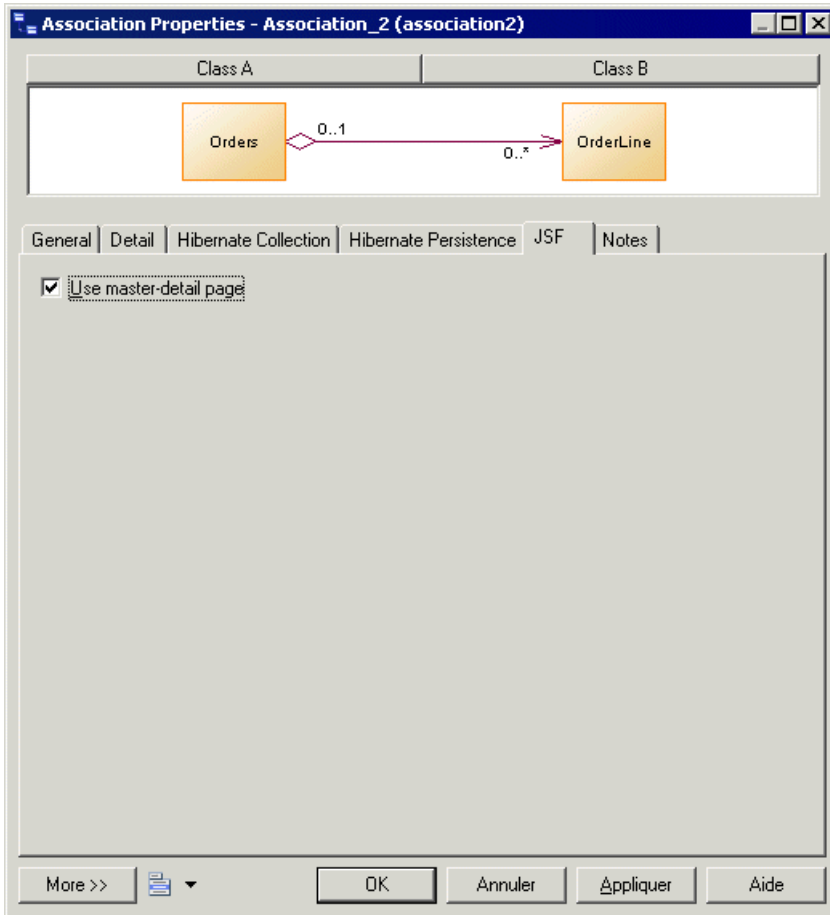
Defining Master-Detail Pages

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.

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:



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

Id	Date	Total	Customer	OrderLines	Edit	Delete
16	Jan 1, 2005	3000.0	Customer	OrderLines	Edit	Delete

[Create](#) [Cancel](#)

OrderLine List

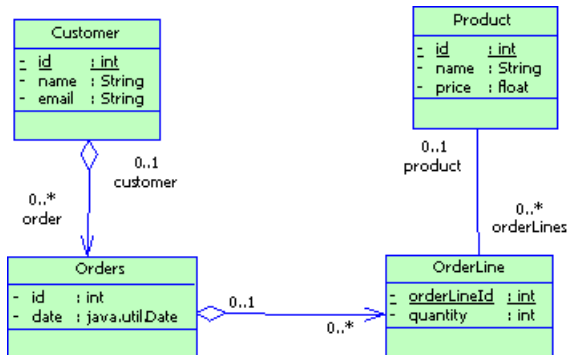
OrderLineId	Quantity	Order	Product	Edit	Delete
1	10	Order	Product	Edit	Delete
2	5	Order	Product	Edit	Delete
4	1	Order	Product	Edit	Delete
5	2	Order	Product	Edit	Delete

[Create](#) [Cancel](#)

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.

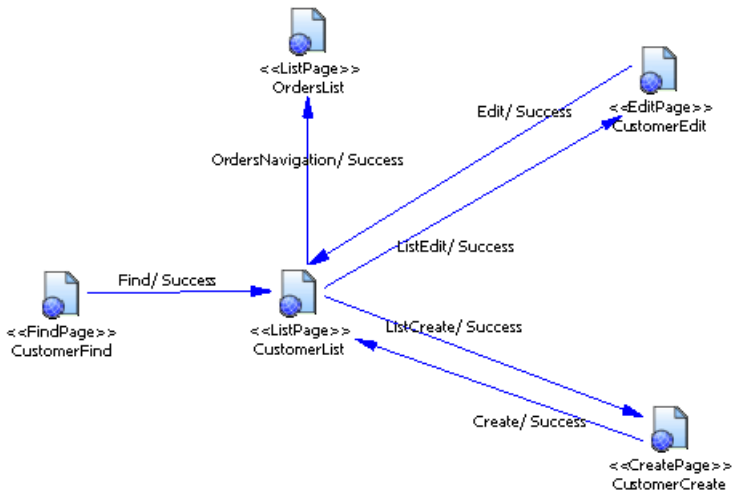
You can Generate PageFlow Diagram diagram in three levels, Model, Package and Class. Next description will use the following example.



Generating a class level PageFlow diagram

You generate a class level PageFlow diagram as follows:

1. Select one class in the ClassDiagram, e.g., Customer. Right click and select the context menu "Generate PageFlow Diagram".
2. A new PageFlow will be automatically generated, e.g., CustomerPageFlow.



Generating a Package Level PageFlow Diagram:

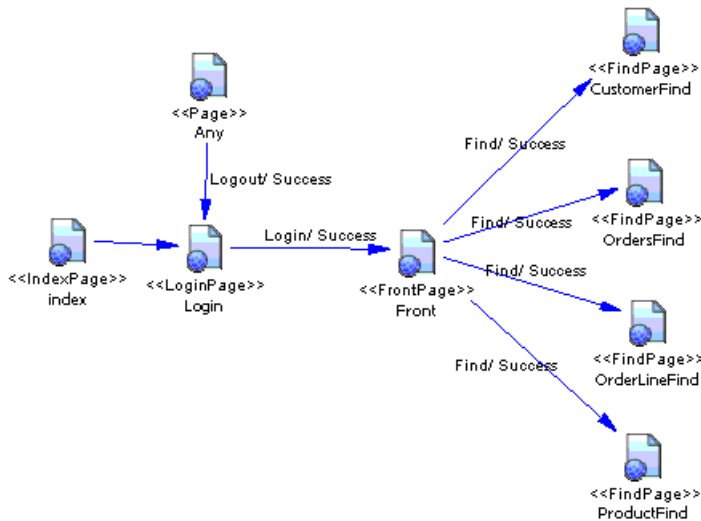
You generate a package level PageFlow diagram as follows:

1. Select one package, e.g., "Orders". Right click it and select the context menu "Generate PageFlow Diagram".
2. The PageFlow will be generated for each class under this package and its sub packages recursively, e.g., CustomerPageFlow, OrdersPageFlow, ProductPageFlow, OrderLinePageFlow.

Generating a Model Level PageFlow Diagram

You generate a model level PageFlow diagram as follows:

1. Right click on the model, e.g., "JSF", and select the context menu "Generate PageFlow Diagram".
2. A model level PageFlow will be generated, e.g., "JSFPageFlow" automatically. At the same time, the PageFlow will be generated for each class under this package and its sub packages recursively, e.g., CustomerPageFlow, OrdersPageFlow, ProductPageFlow, OrderLinePageFlow.



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.

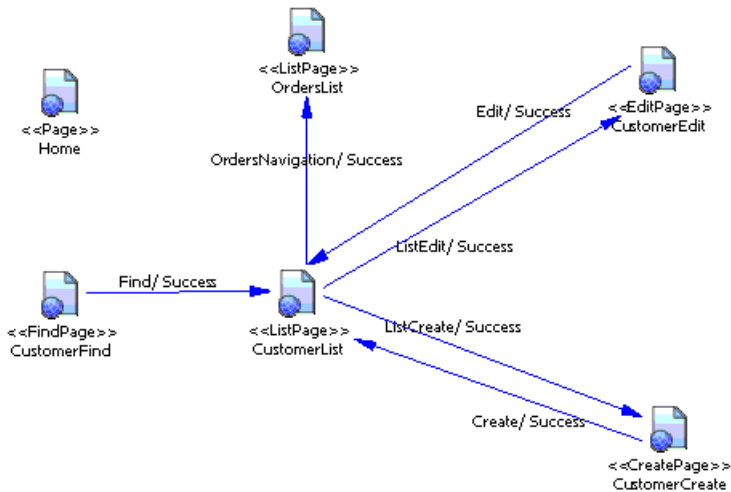
All the pages in the default High Level PageFlow diagram have their pre-defined stereotype, e.g., The stereotype for CustomerFind is "FindPage", CustomerEdit is "EditPage", etc. You can add your customized Page.

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.

Adding a New Page

You add a new page from the Toolbox.

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.

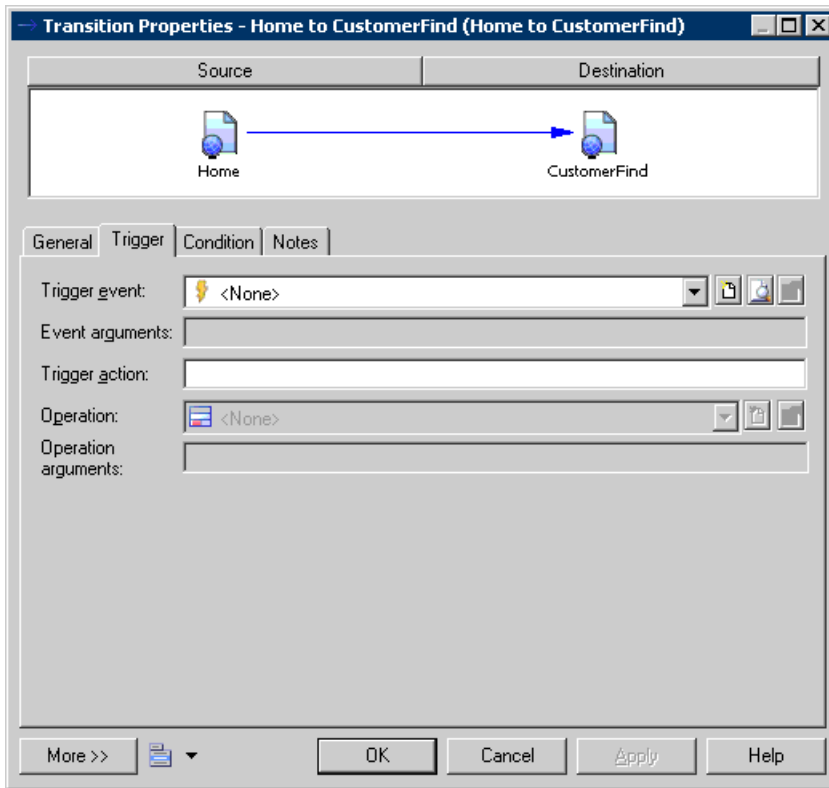


After create a new Page, when the code generation, a default JSF page and its page bean will be generated.

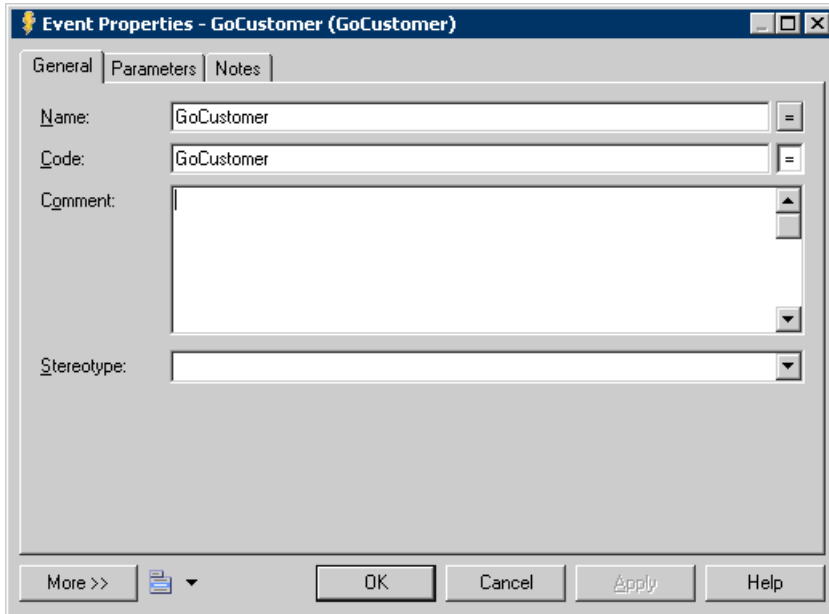
Adding a New PageFlow

You add a new PageFlow from the Toolbox.

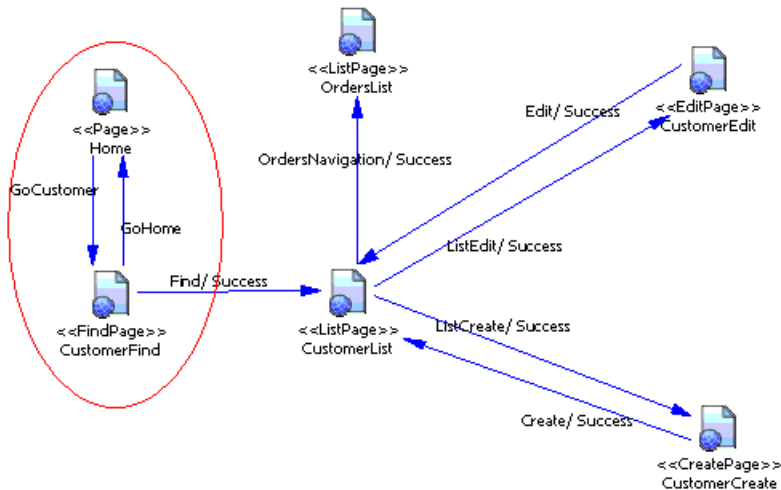
1. Select **Transition** from the Toolbox.
2. Select the source state, e.g., "Home", to the target state, e.g., "CustomerFind". A default transition will be generated.
3. Double click the default transition, or right click the default transition and select **Properties** in the context menu, its property dialog will pop up.



1. Click **Create** button beside Trigger Event, it will pop up Trigger Event creation dialog. You can input the name, e.g., GoCustomer. Click "OK" to finish the creation of Trigger Event.



1. Click "OK" in the property dialog of Transition. A new PageFlow will be created.



After you modify the default pageflow, and generate JSF codes, the corresponding default JSF pages, page beans and faces-config file will be updated.

Installing JSF Runtime Jar Files

You can use an IDE like Eclipse WTP or Sybase WorkSpace to edit, deploy and test JSF pages.

If the IDE does not include JSF runtime Jar files, you will need to download a copy and install them.

You can use the following JSF runtimes:

JSF reference implementation

Apache My Faces

Installing the JSF Reference Implementation

You install the JSF Reference Implementation as follows:

1. Download the JSF reference implementation from the Sun JavaServer Faces website: <http://java.sun.com/javaee/javaserverfaces/>
2. Unzip the downloaded file into a folder. This folder contains the required library files:
 - commons-beanutils.jar
 - commons-collections.jar
 - commons-digester.jar
 - commons-logging.jar
 - jsf-api.jar
 - jsf-impl.jar
3. The JSF reference implementation requires an additional jar file:
 - jstl.jar

You can download it from the Apache Jakarta Project website: <http://jakarta.apache.org/>.

Unzip the downloaded file to obtain the folder "jakarta-taglibs\standard\lib", and copy the jstl.jar file to the JSF reference implementation lib folder.

4. Define a "JSF_LIB" variable in the PowerDesigner "General Options" window, Variables tab to indicate the JSF reference implementation lib folder path.

Installing Apache My Faces

You install Apache My Faces as follows:

1. Download the Apache MyFaces JSF implementation from the Apache MyFaces Project website: <http://myfaces.apache.org/>

Select the current release of Apache MyFaces, for example, myfaces-1.1.1.zip.

2. Download the current release of Apache MyFaces, and unzip it into a folder, which will contain the following jar files:

- myfaces-all.jar
- myfaces-api.jar
- myfaces-impl.jar

Remove the myfaces-api.jar and myfaces-impl.jar files.

3. Download the dependency jar files from the same site and copy the jar files of the folder myfaces-blank-example\WEB-INF\lib to the Apache MyFaces lib folder. You should have 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
4. Define a "JSF_LIB" variable in the PowerDesigner "General Options" window, Variables tab to indicate the Apache MyFaces implementation lib folder path.

Configuring for JSF Generation

You configure for JSF generation as follows:

1. Select **Tools > General Options**.
2. Select the Variables Category.
3. Add a new row in the Variables list:
 - Name: JSF_LIB
 - Value: Select JSF Jar file library folder.
4. Select the Named Paths Category.
5. If there is no _JSF named path, add a new row in the Named Paths list:
 - Name: _JSF
 - Value: Select the JSF folder where it contains JSF style sheets, headers, footers and images.

Generating JSF Pages

Before generation, make sure that you have attached the Hibernate Extension file to the model, and checked the model for errors.

1. Java CodeSelect **Language > Generate Java Code**.
2. Specify a target directory.

3. Define generation options.
4. Click OK.

The generation produces the following files:

- Persistent files (persistent classes, DAO, ...)
- Eclipse and Eclipse WTP project artifacts
- A home page
- JSF pages for persistent classes:
 - A find page for searching objects
 - A list page for displaying find results
 - A create page for creating new objects
 - An edit page for updating objects
- Managed beans
- Page flows (face configuration files)

Testing JSF Pages

You can deploy a JSF Web application in a Web server or an application server that supports JSP. For example, Apache Tomcat, JBoss.

You can use an IDE like Eclipse to edit the generated Java classes, and use the Eclipse WTP (Web Tools Project) to edit JSF pages and face config files.

Testing JSF Pages with Eclipse WTP

You test JSF Pages with Eclipse WTP as follows:

1. Install a Web server such as Tomcat or an application server such as JBoss.
2. Generate Java code
3. Import the JSF project into Eclipse. The project is built.
4. Configure your Web server or application server.
5. Start the database.
6. Start the Web server or application server using the WTP Servers view.
7. Right-click the index.jsp under the webroot folder and select **Run As > Run on Server**.
8. Select the server you want to use in the list.

Testing JSF Pages with Apache Tomcat

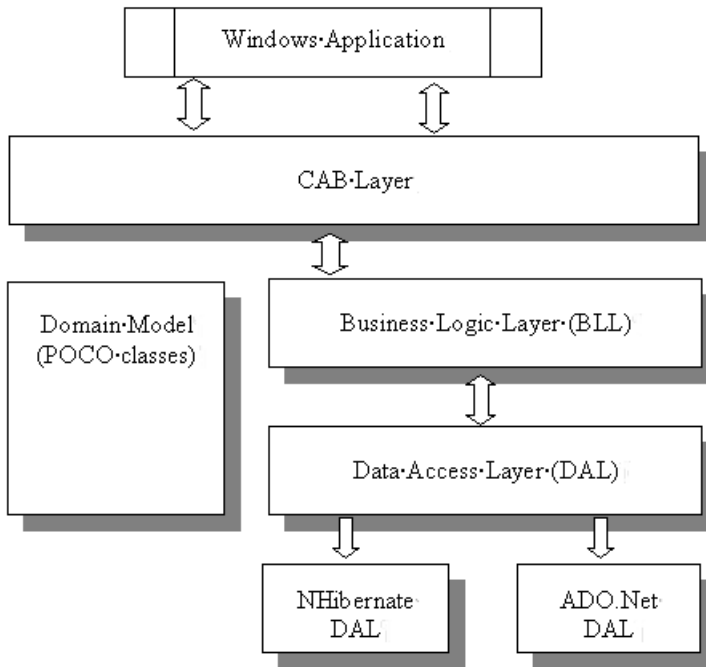
You test JSF Pages with Apache Tomcat as follows:

1. Install a Web server such as Tomcat or an application server such as JBoss.
2. Generate Java code
3. Import the JSF project into Eclipse. The project is built.

4. Copy the <ProjectName> folder under the .deployables folder into the Apache Tomcat webapps folder. Where <ProjectName> is the Eclipse project name.
5. Start the database.
6. Start Apache Tomcat server.
7. Run the Web application using the URL: <http://<hostname>:8080/<ProjectName>/index.jsp>. If Apache Tomcat is installed locally, <hostname> is localhost.

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.

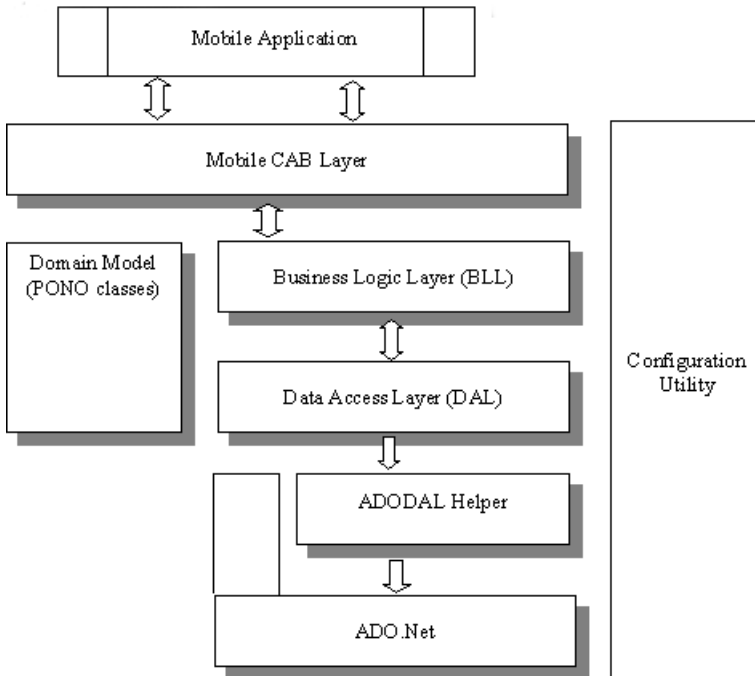
The DAL Helper provides common features used by all the DAL implementations, such as connection and transaction management, and the supply of SQL command parameters. Some common classes, such as Session, Criteria, and Exception, are also defined.

PowerDesigner supports two kinds of DAL implementation:

- ADO.NET (see *Generating ADO.NET and ADO.NET CF Persistent Objects* on page 633)
- Nhibernate (see *Generating NHibernate Persistent Objects* on page 641)
- 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* on page 664).

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* on page 633)

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 **Import** 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)

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.

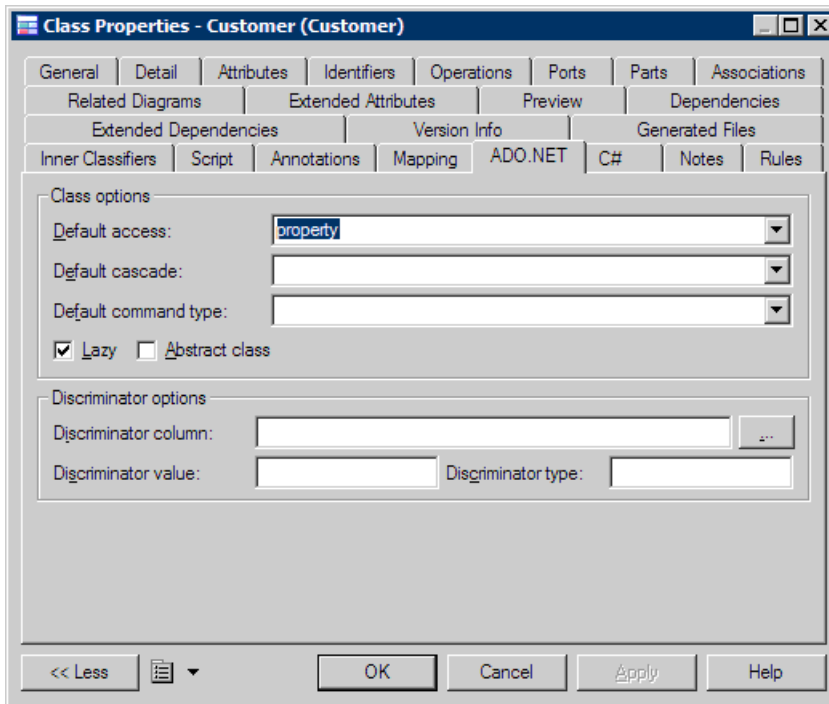
Option	Description
Data Provider	<p>Specifies which data provider you want to use. For ADO.NET, you can choose between:</p> <ul style="list-style-type: none"> • OleDb • SqlClient • ODBC • Oracle <p>For ADO.NET CF, you can choose between:</p> <ul style="list-style-type: none"> • Microsoft SQL Server 2005 Mobile Edition • Sybase ASA Mobile Edition
Connection String	<p>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 <i>Configuring Connection Strings</i> on page 656.</p>
Default access	<p>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.</p>
Default cascade	<p>Specifies the default cascade type.</p>
Default command type	<p>Specifies the default command type, which can be overridden by concrete class.</p>
DALContainer	<p>Specifies the collection type returned from database. You can choose between Generic List Collection and System.Collections.ArrayList.</p>
Logging Type	<p>[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</p>

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.

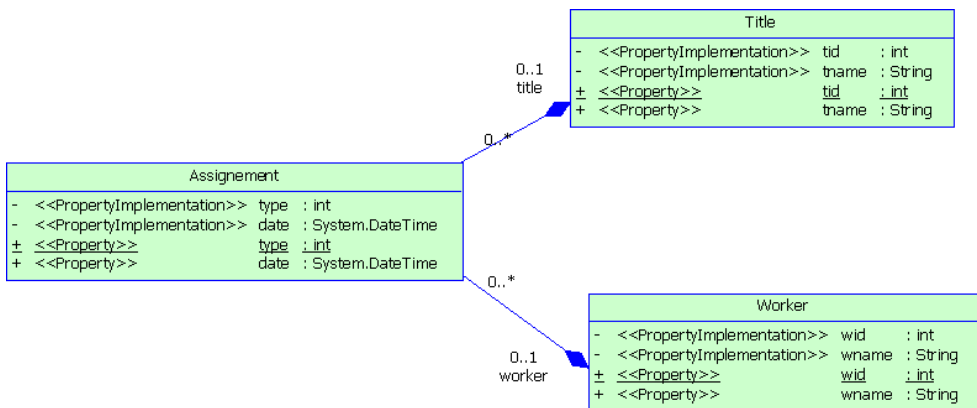
Primary Identifier Mappings

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.

Person	
- <<PropertyImplementation>>	name : Name
- <<PropertyImplementation>>	sex : System.Char
- <<PropertyImplementation>>	age : int
+ <<Property>>	name : Name
+ <<Property>>	sex : System.Char
+ <<Property>>	age : int

Name	
- <<PropertyImplementation>>	firstName : String
- <<PropertyImplementation>>	middleName : String
- <<PropertyImplementation>>	lastName : String
+ <<Property>>	firstName : String
+ <<Property>>	middleName : String
+ <<Property>>	lastName : String

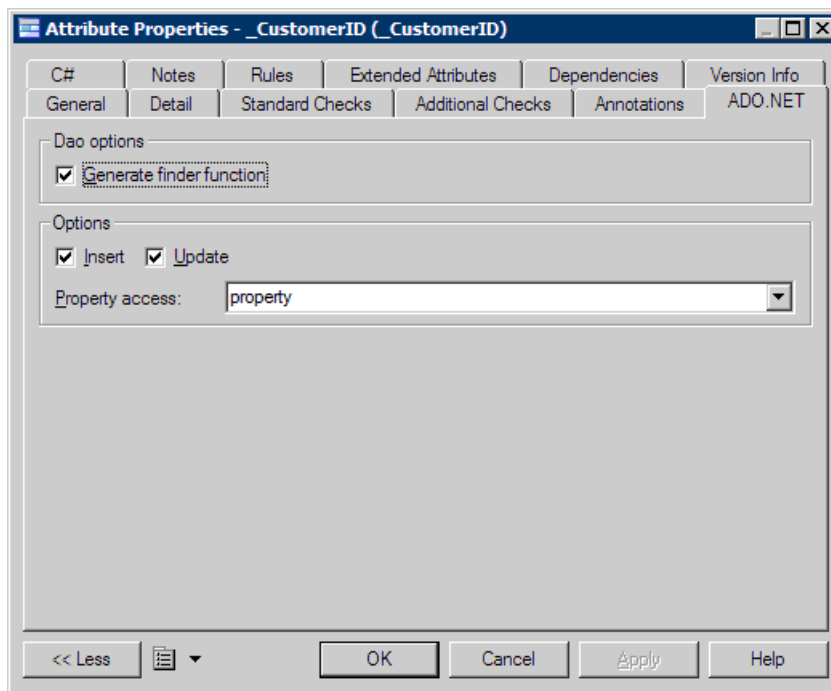
Attribute Mappings

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.

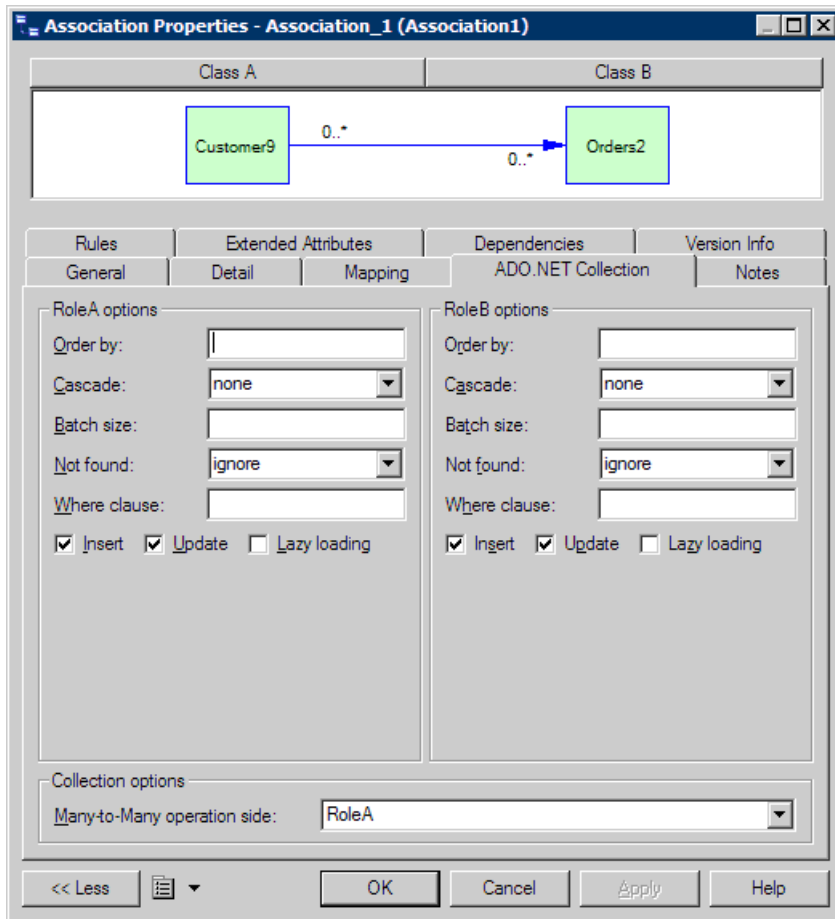


Option	Description
Generate finder function	Generates a finder function for the attribute.
Insert	Specifies that the mapped columns should be included in any SQL INSERT statements.
Update	Specifies that the mapped columns should be included in any SQL UPDATE statements.
Lazy	Specifies that this property 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.

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.

Open the Association property sheet and click the ADO.NET or ADO.NET CF Collection tab.



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.

Option	Description
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.

Defining Inheritance Mappings

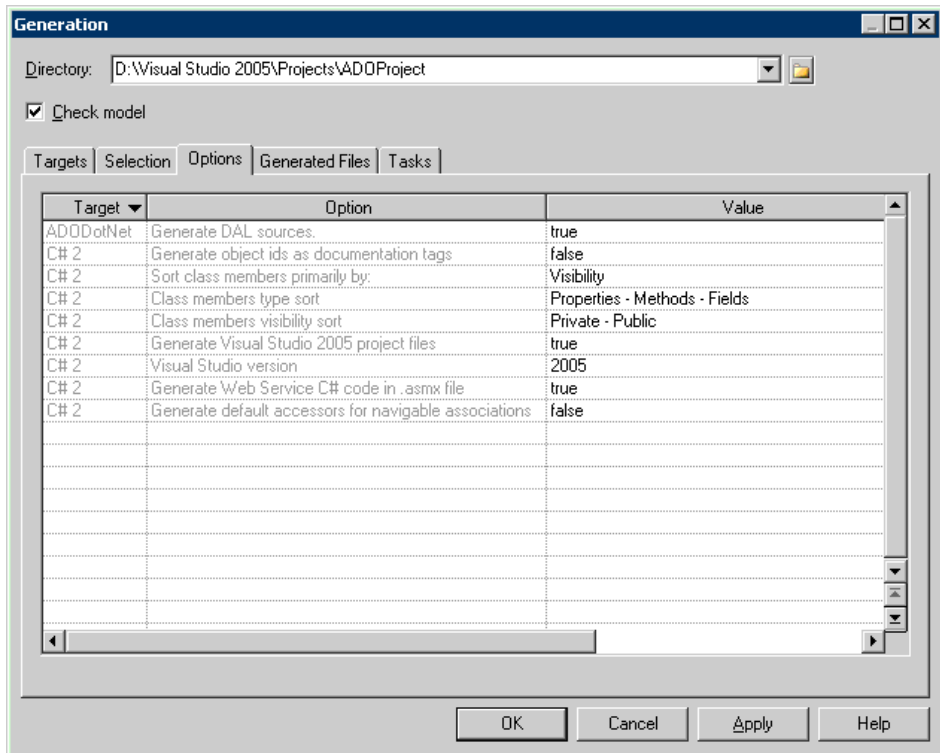
ADO.NET and ADO.NET CF support the three basic inheritance mapping strategies:

- Table per class hierarchy
- Table per subclass
- Table per concrete class
- These strategies all follow the standard inheritance mapping definitions.

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:

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:



- [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 Visual Basic 2005 Files* on page 476 or *Generating C# 2.0 Files* on page 520.
- Click OK to generate code immediately or Apply and then Cancel to save your changes for later.

Once generation is complete, you can use an IDE such as Visual Studio.NET 2005 to modify the code, compile, and develop your application.

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 **Import** 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)

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. NHibernate Tag: dialect
Driver class	Specifies the driver class. NHibernate Tag: connection.driver_class
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 <i>Configuring Connection Strings</i> on page 656. NHibernate Tag: connection.url
Auto import	Specifies that users may use an unqualified class name in queries.
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.
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.
Show SQL	Specifies that SQL statements should be shown in the log. NHibernate Tag: show_sql

Option	Description
Auto schema export	Specifies the mode of creation from tables. NHibernate Tag: hbm2ddl.auto

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:

Class Properties - Customer (Customer)

General | Detail | Attributes | Operations | Ports | Parts | Annotations | C#

NHibernate | Notes | Extended Attributes | Preview

Class Options:

Data operation options:

Dynamic insert Dynamic update Select before update Default cascade type:

Class tag options:

Default access type: Schema name:

Proxy name: Catalog name:

Batch size: Row id:

Check: Persister class name:

Polymorphism:

Lazy Mutable Auto import

Discriminator options:

Discriminator column:

Discriminator value: Discriminator type:

Force usage of discriminator Do not use discriminator in insert

Optimistic lock options:

Type: Unsaved value:

Column name:

More >>

Option	Description
Dynamic insert	Specifies that INSERT SQL should be generated at runtime and will contain only the columns whose values are not null. NHibernate Tag: dynamic-insert
Dynamic update	Specifies that UPDATE SQL should be generated at runtime and will contain only the columns whose values have changed. NHibernate Tag: dynamic-update
Select before update	Specifies that NHibernate should never perform a SQL UPDATE unless it is certain that an object is actually modified. NHibernate Tag: select-before-update
Default cascade	Specifies the default cascade style. NHibernate Tag: default-cascade

Option	Description
Default access	Specifies the default access type (field or property) NHibernate Tag: default-access
Proxy name	Specifies an interface to use for lazy initializing proxies. NHibernate Tag: proxy
Batch size	Specifies a "batch size" for fetching instances of this class by identifier. NHibernate Tag: batch-size
Check	Specifies a SQL expression used to generate a multi-row check constraint for automatic schema generation. NHibernate Tag: check
Polymorphism	Specifies whether implicit or explicit query polymorphism is used. NHibernate Tag: polymorphism
Schema name	Specifies the name of the database schema. NHibernate Tag: schema
Catalog name	Specifies the name of the database catalog. NHibernate Tag: catalog
Row id	Specifies that NHibernate can use the ROWID column on databases which support it (for example, Oracle). NHibernate Tag: rowed
Persister class name	Specifies a custom persistence class. NHibernate Tag: persister
Lazy	Specifies that the class should be lazy fetching. NHibernate Tag: lazy
Mutable	Specifies that instances of the class are mutable. NHibernate Tag: mutable
Abstract class	Specifies that the class is abstract. NHibernate Tag: abstract
Auto import	Specifies that an unqualified class name can be used in a query NHibernate Tag: Auto-import
Discriminator column	Specifies the discriminator column or formula for polymorphic behavior in a one table per hierarchy mapping strategy. NHibernate Tag: discriminator

Option	Description
Discriminator value	Specifies a value that distinguishes individual subclasses, which are used for polymorphic behavior. NHibernate Tag: discriminator-value
Discriminator type	Specifies the discriminator type. NHibernate Tag: type
Force usage of discriminator	Forces NHibernate to specify allowed discriminator values even when retrieving all instances of the root class. NHibernate Tag: force
Do not use discriminator in insert	Forces NHibernate to not include the column in SQL INSERTs NHibernate Tag: insert
Optimistic lock type	Specifies an optimistic locking strategy. NHibernate Tag: optimistic-lock
Optimistic lock column name	Specifies the column used for optimistic locking. A field is also generated if this option is set. NHibernate Tag: version/ timestamp
Optimistic lock unsaved value	Specifies whether an unsaved value is null or undefined. NHibernate Tag: unsaved-value

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.

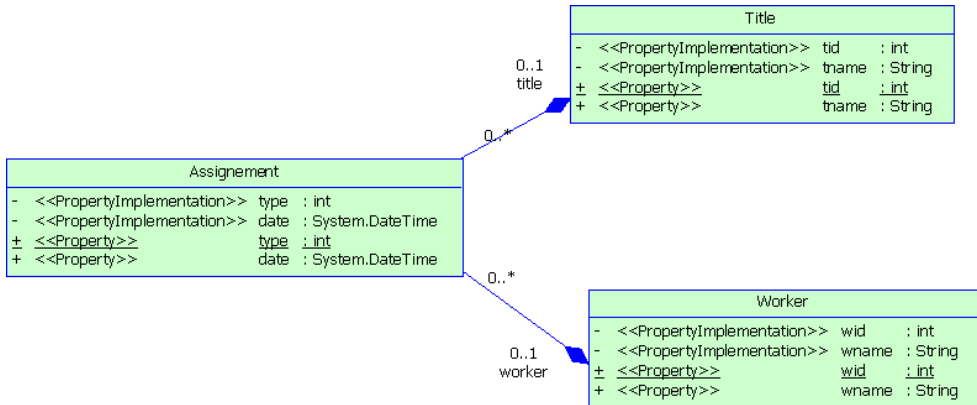
There are three kinds of primary identifier mapping in NHibernate:

- 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.



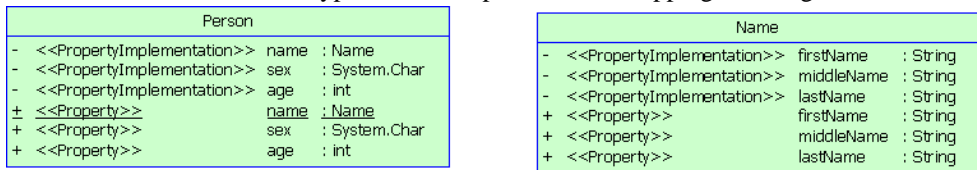
In the above example, the Assignment class has a primary identifier with three attributes: one basic type attribute and two migrated attributes. The primary identifier mapping is as follows:

```

<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.



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>
</composite-id>
    
```

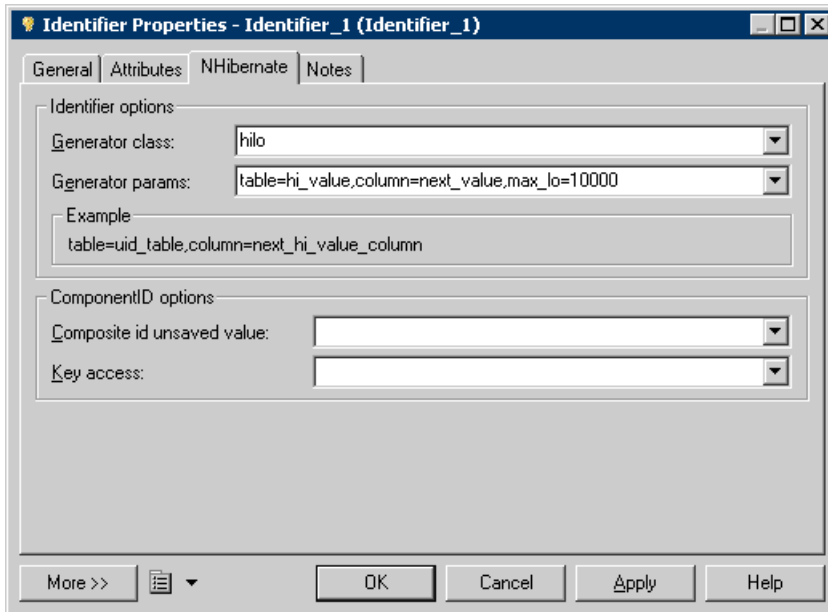
```
<key-property name="lastName">
  <column name="name_lastName"
    sql-type="text" />
</key-property>
</composite-id>
```

Note: The value type class must implement the `java.io.Serializable` interface, which implements the `equals()` and `hashCode()` methods.

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.

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`.



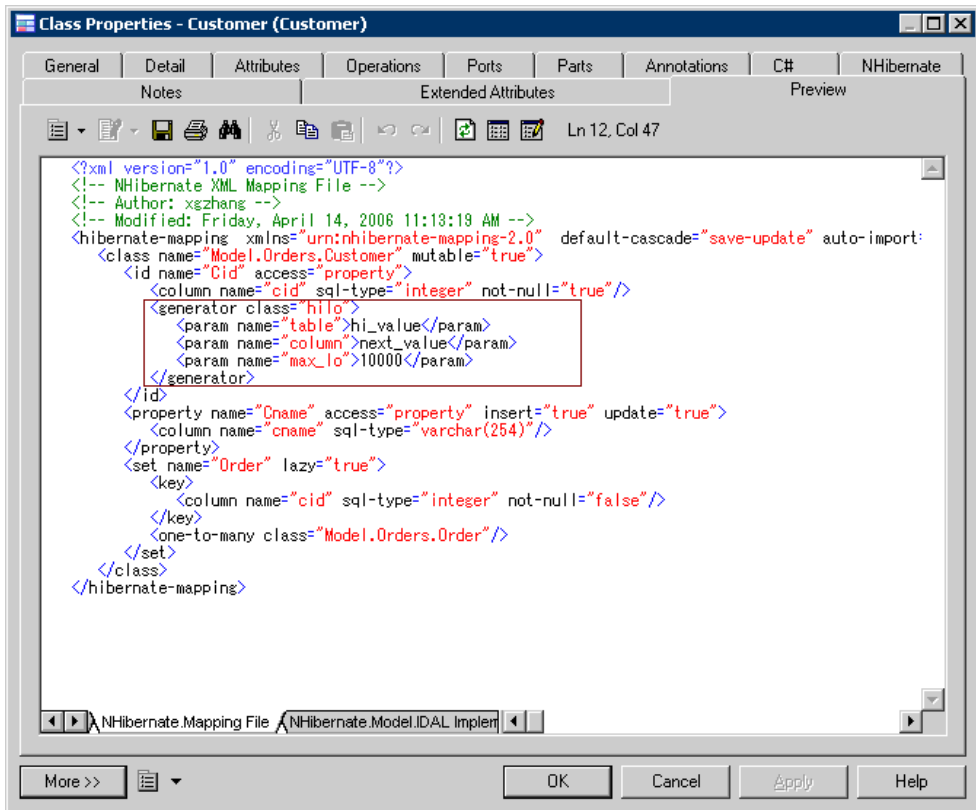
1. Open the class property sheet and click the Identifiers tab. Double-click the entry to open its property sheet.
2. Click the NHibernate tab, select a generator class and define its parameters.

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:



Note that, if there are several Primary identifier attributes, the generator is not used.

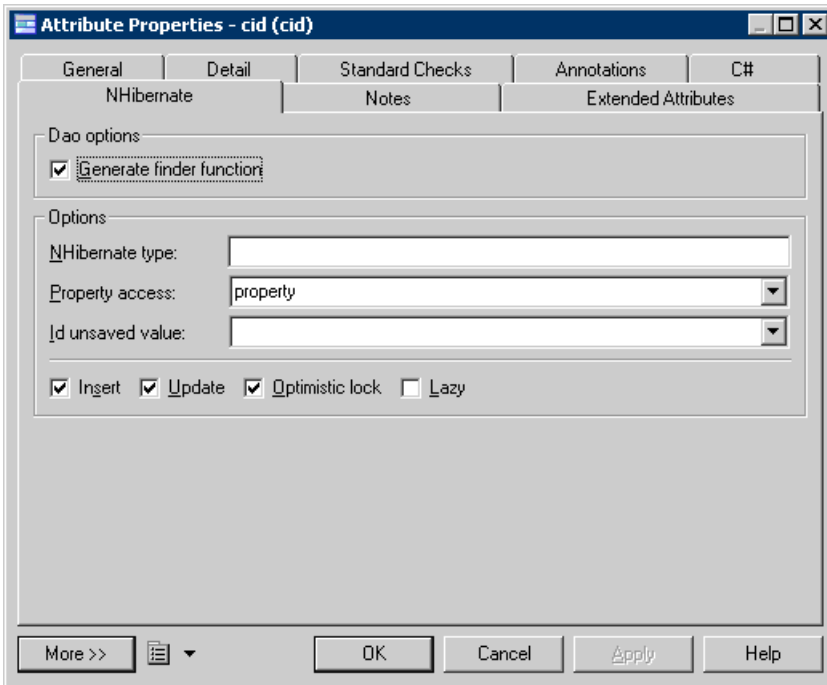
Attribute Mappings

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.

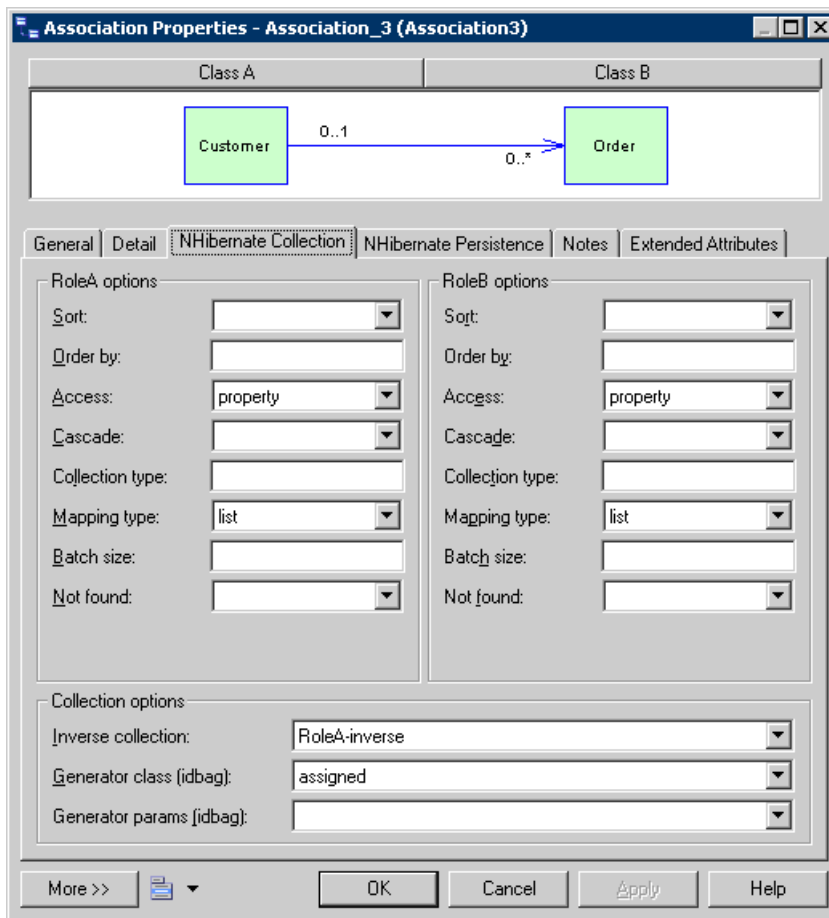


Option	Description
Generate finder function	Generates a finder function for the attribute.
NHibernate type	Specifies a name that indicates the NHibernate type.
Property access	Specifies the strategy that NHibernate should use for accessing the property value.
Id unsaved value	Specifies the value of an 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).

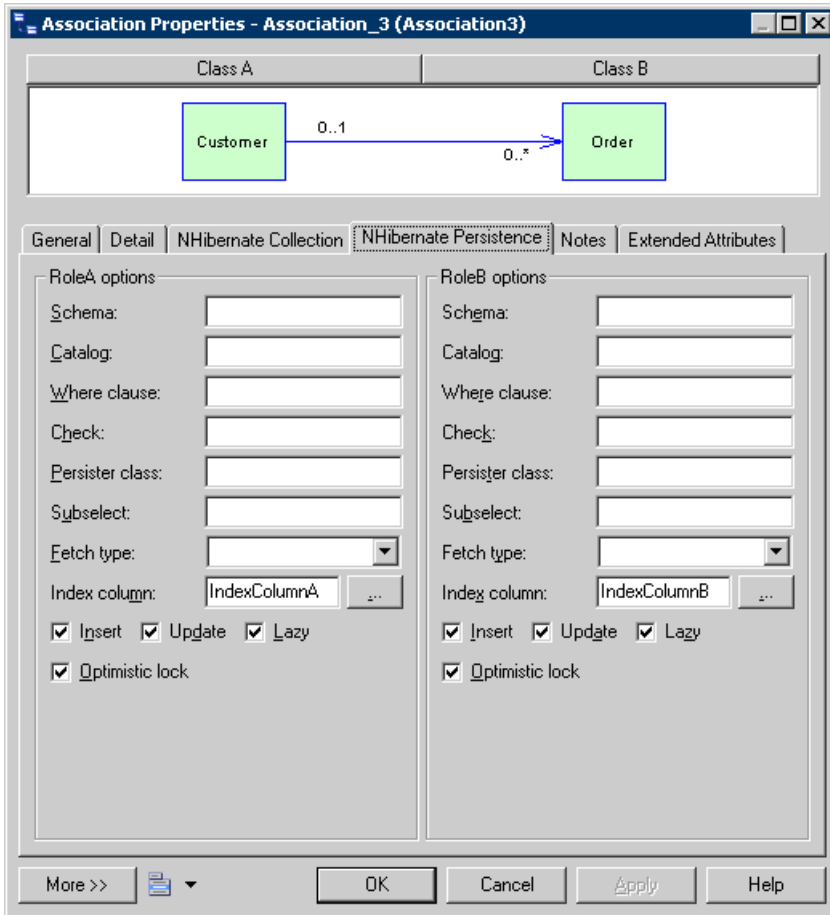
Defining Association Mappings

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.

1. Open the Association property sheet and click the NHibernate Collection tab.
2. Define the collection management options (see *Defining NHibernate Collection options* on page 652).



3. Select the NHibernate Persistence tab, and define the persistence options (see *Defining NHibernate Persistence options* on page 653).



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. NHibernate Tag: sort
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. NHibernate Tag: order-by
Access	Specifies the strategy Nhibernate should use for accessing the property value. NHibernate Tag: access

Option	Description
Cascade	Specifies which operations should be cascaded from the parent object to the associated object. NHibernate Tag: cascade
Collection type	Specifies a name that indicates the NHibernate type. NHibernate Tag: type
Batch size	Specifies the batch load size. NHibernate Tag: batch-size
Not found	Specifies how foreign keys that reference missing rows will be handled: ignore will treat a missing row as a null association. NHibernate Tag: not-found
Inverse collection	Specifies that the role is the inverse relation of the opposite role. NHibernate Tag: inverse
Mapping type	Specifies the collection mapping type NHibernate Tag: Set, Array, Map, or List.

Defining NHibernate Persistence Options

The following options are available:

Option	Description
Schema	Specifies the name of the schema. NHibernate Tag: schema
Catalog	Specifies the name of the catalog. NHibernate Tag: catalog
Where clause	Specifies an arbitrary SQL WHERE condition to be used when retrieving objects of this class. NHibernate Tag: where
Check	Specifies a SQL expression used to generate a multi-row check constraint for automatic schema generation. NHibernate Tag: check
Fetch type	Specifies outer-join or sequential select fetching. NHibernate Tag: fetch
Persister class	Specifies a custom persistence class. NHibernate Tag: persister

Option	Description
Subselect	Specifies an immutable and read-only entity to a database subselect. NHibernate Tag: subselect
Index column	Specifies the column name if users use list or array collection type. NHibernate Tag: index
Insert	Specifies that the mapped columns should be included in any SQL INSERT statements. NHibernate Tag: insert
Update	Specifies that the mapped columns should be included in any SQL UPDATE statements. NHibernate Tag: update
Lazy	Specifies that this property should be fetched lazily when the instance variable is first accessed. NHibernate Tag: lazy
Optimistic lock	Specifies that a version increment should occur when this property is dirty. NHibernate Tag: optimistic-lock
Outer join	Specifies to use an outer-join. NHibernate Tag: outer-join

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	Iesi.Collections.ISet
Bag	System.Collections.ICollection
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.

Defining Inheritance Mappings

NHibernate supports the two basic inheritance mapping strategies:

- Table per class hierarchy

- Table per subclass

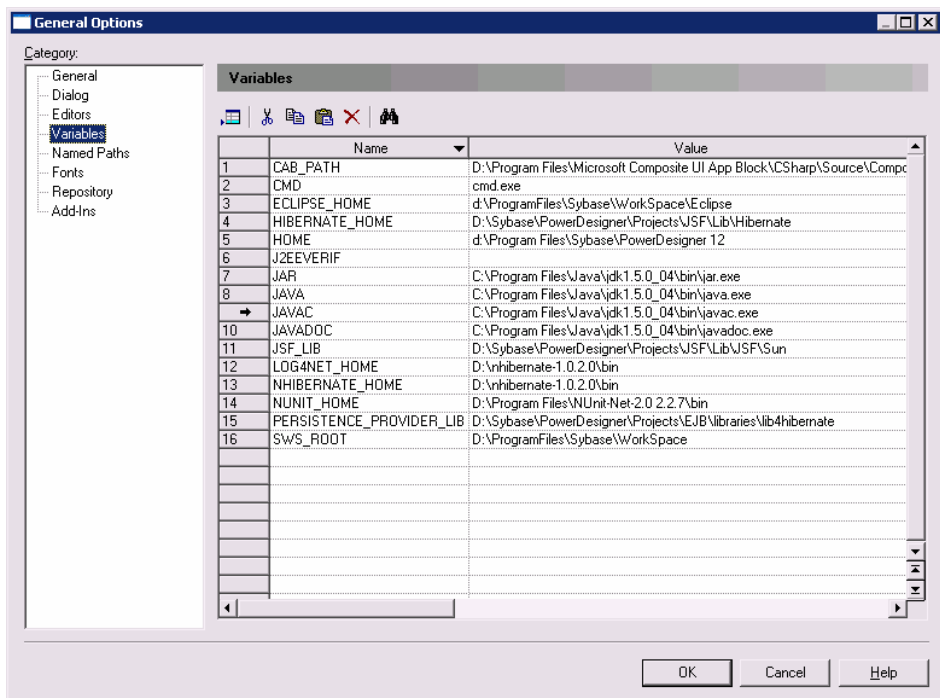
It does not support the "Table per concrete class" mapping strategy.

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.

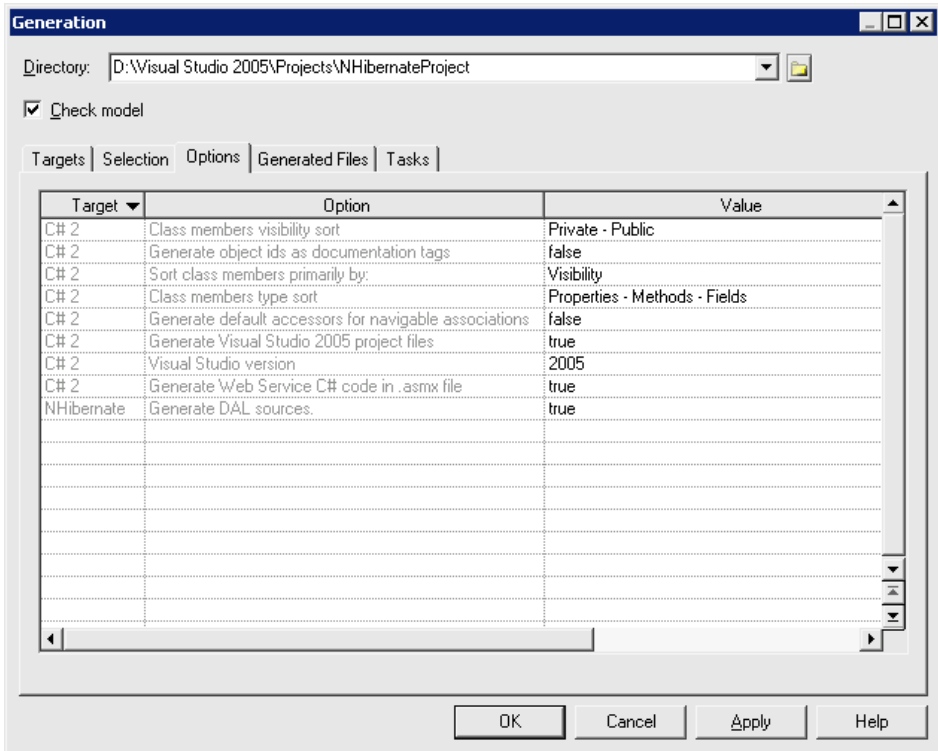
Generating Code for NHibernate

Before generating code for NHibernate, you must have NHibernate 1.0.2 or higher installed.

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 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. [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.

Once generation is complete, you can use an IDE such as Visual Studio.NET 2005 to modify the code, compile, and develop your application.

Configuring Connection Strings

PowerDesigner supports multiple types of database connection with each of the .NET frameworks.

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.

Configuring a Connection String from the ADO.NET or ADO.NET CF Tab

You configure a connection string from the ADO.NET or ADO.NET CF tab as follows:

1. Select a data provider.
2. Click the ellipsis button to open a provider-specific connection string dialog.
3. [ADO.NET only] Enter the necessary parameters and click Test Connection to validate them.
4. Click Apply to Connection String and then Close to return to the ADO.NET or ADO.NET CF tab.

Configuring a Connection String from the NHibernate Tab

You configure a connection string from the NHibernate tab as follows:

1. Select a dialect and driver class.
2. Click the ellipsis button to open a provider-specific connection string dialog.
3. Enter the necessary parameters and click Test Connection to validate them.
4. Once the connection tests correctly, click Close to return to the NHibernate tab.

OLEDB Connection String Parameters

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.
Password	Specifies the database user 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.

ODBC Connection String Parameters

The following parameters are required to configure an ODBC connection string:

Option	Description
ODBC source name	Specifies the ODBC source name
User name	Specifies the database user name.
Password	Specifies the database user password.

Microsoft SQL Server and Microsoft SQL Server Mobile Edition Connection String Parameters

The following parameters are required to configure a Microsoft SQL Server and Microsoft SQL Server Mobile Edition connection string:

Option	Description
Server name	Specifies the server name.
User name	Specifies the database user name.
Password	Specifies the database user password.
Authentication type	Specifies authentication type, Use SQL Server Authentication, or Use Windows Authentication
Database name	Specifies the database name
Database file name	Specifies the database's file name.
Logical name	Specifies the logical name for the database file.

Oracle Connection String Parameters

The following parameters are required to configure an Oracle connection string:

Option	Description
Server name	Specifies the server name.
User name	Specifies the database user name.
Password	Specifies the database user password.

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.

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:

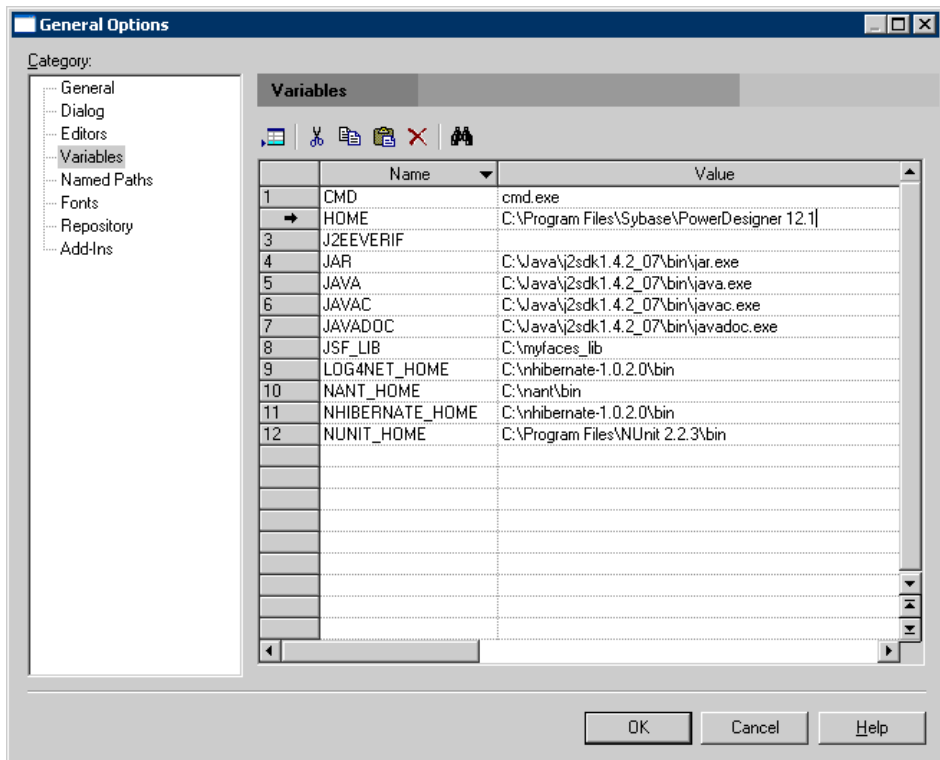
- Mapping of the class should be defined.
- The class can be instantiated. Unit test classes cannot be generated for abstract classes.
- The class is not a value type.
- The Mutable property is set to true. If Mutable is set to false, the class can not be updated or deleted.
- The class has no unfulfilled 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 **Import** tool, select the `UnitTest.NET` or `UnitTest.NET CF` file (on the **Unit Test** tab), and click **OK** to attach it.

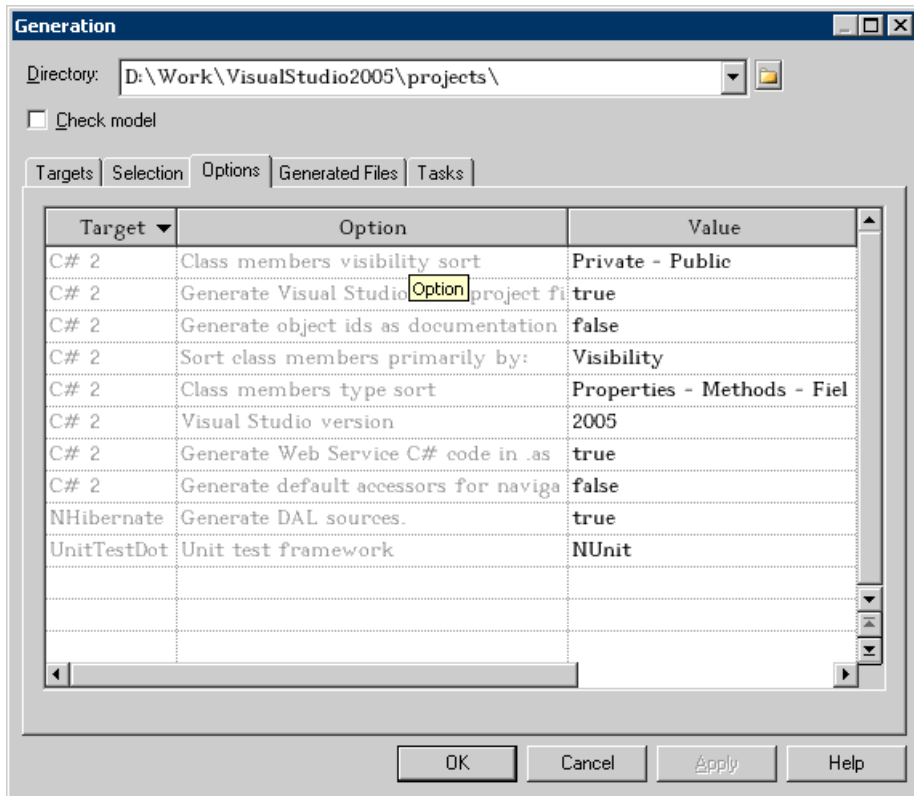
Before generating code for UnitTest, you must have NUnit 2.2.3 or higher installed (available at <http://www.nunit.org>).

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.

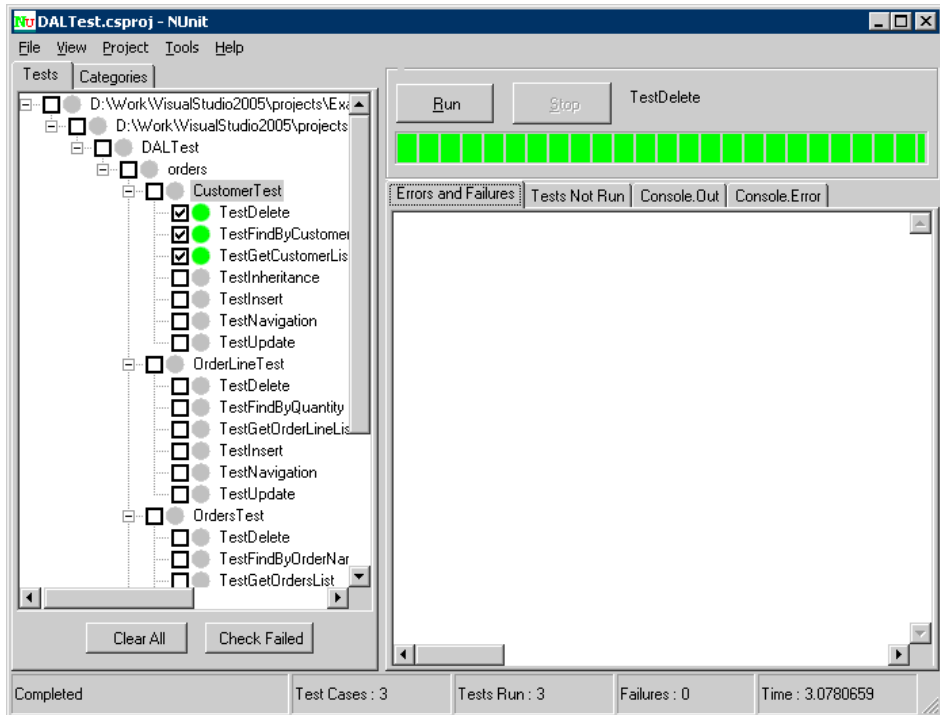
Running NUnit Unit Tests

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.

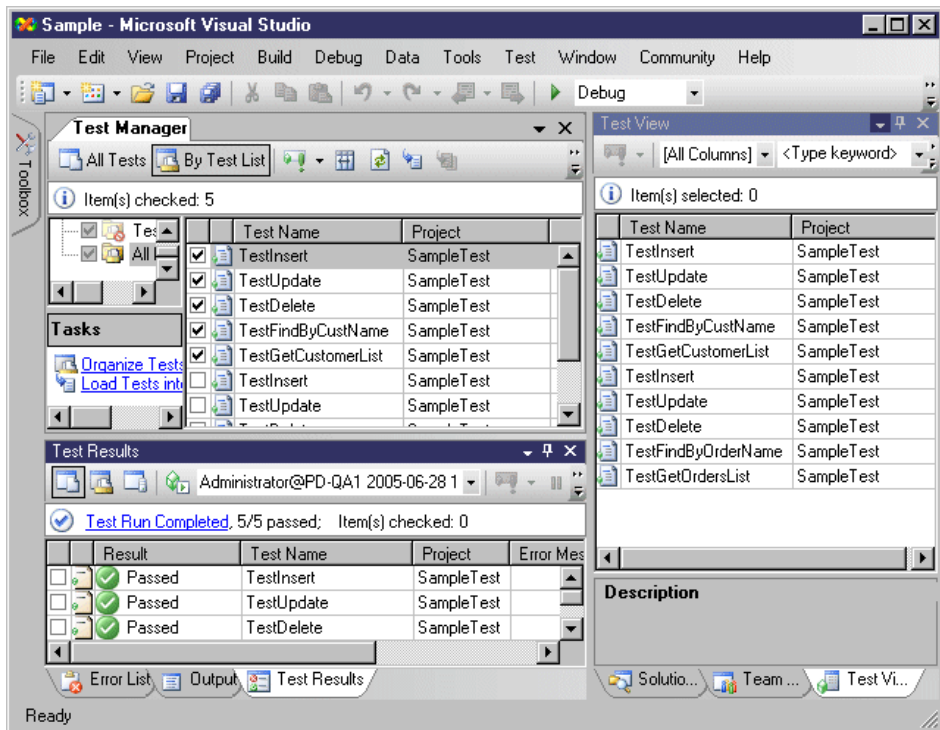
Running Visual Studio Test System Unit Tests

The Visual Studio Team System Test tools offer a number of ways to run tests, both from the Visual Studio integrated development environment (IDE) and from a command prompt.

Running Tests in Visual Studio.NET 2005 IDE

You run tests in Visual Studio.NET 2005 IDE as follows:

1. Use the Test Manager or Test View window. You can also rerun tests from the Test Result window.
2. In the Test Manager window, select tests by selecting the check boxes in the test's row, and then either click Run Tests on the Test Manager toolbar or right-click the selected tests and then click Run Checked Tests.
3. In the Test View window, select the tests you want to run and then click Run Tests on the Test View toolbar, or right-click, and select Run Selection.

**Running Tests from the Command Line**

You run tests from the command line as follows:

1. Open the Visual Studio command prompt
2. Either navigate to your solution folder directory or, when you run the MSTest.exe program in step, specify a full or relative path to the metadata file or to the test container.
3. Run the MSTest.exe program

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 **Import** 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.

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.

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.

Controlling the Data Grid View

You can specify the length of your ADO.NET CF datagrid views.

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.

Defining Attributes Display Options

You can define attribute-level options for presentation style.

1. Open `Attribute Property` sheet, and select the `Windows Application` tab.

2. Set the appropriate options and then click OK

The following options are available:

Option	Description
Control Type	You can choose TextBox or ComoboBox 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.

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.

1. Open the attribute property sheet, and click the Standard Checks tab.
2. [optional] Define minimum value and maximum values to control the value range.
3. [optional] Define a default value. A string value must be enclosed in quotes. You can also define the Initial value in the Details tab.
4. [optional] Define a list of values. PowerDesigner will automatically generate a combo box that includes these values.
5. Click OK to return to the model.

Generating Code for a Windows Application

Before generating code for Windows Application, you need to install CAB (available from <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/cab.asp>).

Check for the following required files in your installation directory:

- Microsoft.Practices.CompositeUI.dll
- Microsoft.Practices.CompositeUI.WinForms.dll
- Microsoft.Practices.ObjectBuilder.dll

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

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.

Generating Code for a Smart Device Application

PowerDesigner can generate code for smart device applications.

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* on page 664).
 - 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* on page 633).
 - 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
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.

Deploying Code to a Smart Device

You deploy code to a smart device as follows:

1. Compile your generated code in Visual Studio.
2. Deploy the start up project, i.e. the <model>Test project or User Interface project
3. Deploy the SystemFramework project separately with the database file and required DLLs (such as ulnet10.dll for UltraLite support).

Testing the Application on the Device

You test the Application on the Device as follows:

1. 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.
2. 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.

Index

.NET

- compile source files 257
- generate server side code 256
- generation options 256
- generation tasks 257
- Web service method 246
- Web service proxy code 257
- Web service support 232

A

- abstract data type 288, 291
- access point URL 249
- accessibility
 - C# 484
 - VB.NET 438
- action 205
 - check model 315
 - create 207
 - properties 207
- action type 165, 166
- activation 137
 - attach message 152
 - create 152
 - detach message 153
 - move 154
 - overlap 153
 - procedure call 152
 - resize 154
 - sequence diagram 151
- activity 160
 - action type 165, 166
 - attached to organization unit 175
 - check model 317
 - committee activity 176
 - create 161
 - decomposed 170
 - parameter 164
 - properties 162
- activity diagram 131
 - activity 160
 - convert to decomposed activity 172
 - object node 192
 - organization unit 173
 - start 182, 183, 185

actor 20

- check model 298
- create 22
- drag and drop in other diagram 24
- primary actor 20
- properties 22
- secondary actor 20
- show symbol 24

add package hierarchy 272

ADO.NET

- association mapping 638
- attribute mapping 637
- component primary identifier mapping 636
- composite identifier mapping 636
- database connection strings 656
- extension file 633
- generating code 640
- inheritance mapping 640
- O/R mapping 634
- ODBC connection string 658
- OleDb connection string 657
- options 633
- Oracle connection string 658
- simple identifier mapping 636
- SQL Server connection string 658

ADO.NET CF

- options 633

aggregation association 88

Analysis (object language) 269

annotation 111

- assigning 111
- Java 353

archive Java 403

argument

- event 205

ASMX file

- generate 257

ASP.NET 457

- artifact file object 458
- ASP file 457
- create 458
- create from class diagram 458
- create from selected file object 458
- default template 457
- file object 458

Index

- generate 460
 - TemplateContent 458
 - wizard 458
- assembly
 - C# 2.0 505
 - Visual Basic 2005 463
- assembly connector 107
 - check model 329
 - create 108
 - properties 108
- association 24, 88
 - add association class 95
 - change to instance link 98
 - check model 330
 - container type 92
 - create 89
 - default implementation 92
 - generate PDM 280
 - generated code 94
 - implementation 92
 - implementation class 92
 - migrate navigable roles 97
 - pdGenerated 94
 - pdRoleInfo 94
 - properties 90
 - role 90
 - transformation 544
- association class 90, 95
 - transformation 546
- association mapping 559
- attribute 65
 - add 72
 - add operation 71
 - attach 65
 - C# 487
 - check model 308
 - complex data type 542
 - constraint 70, 71
 - create 66, 72
 - duplicate 72
 - identifier 77
 - initial value 74
 - interface 65
 - migrate 49, 286
 - operation 71
 - override 74
 - properties 66
 - simple data type 542
 - transformation 542

- update using domain 124
 - VB.NET 442
- attribute mapping 555
- AXIS EJB 255, 256
- AXIS RPC 254

B

- bean class
 - adding an operation 364
- bidirectional association in C++ 535
- bindingTemplate 229
- BMP entity bean 356
- bound classifier wizard 45
- businessEntity 229
- businessService 229

C

C#

- accessibility 484
- annotation 111
- attribute 487
- class 485
- constructor 493
- conversion operator method 494
- custom attributes 487
- delegate 493
- destructor 493
- documentation tag 495
- enumeration 485
- event 494
- field 487
- generation 269, 496, 520
- implementation 483
- indexer 489
- inheritance 483
- interface 485
- method 491
- method implementation 491
- method parameter 491
- namespace 483
- operator method 494
- preprocessing 503
- preprocessing (reverse engineering) 500
- preprocessing symbol (reverse engineering) 502
- project 483
- property 488

- reverse engineering 498, 500–503
- struct 485
- C# 2.0
 - assembly 505
 - class 510
 - compilation unit 507
 - custom attributes 520
 - delegate 512
 - enum 513
 - event 517
 - field 514
 - implementation 520
 - indexer 517
 - inheritance 520
 - interface 511
 - method 514
 - namespace 509
 - partial type 508
 - property 517
 - reverse-engineering code 522
 - struct 511
- C# generation option
 - code in .ASMXML 256
- C# reverse engineering 503
 - preprocessing directives 501
- C++ 536
 - bidirectional association 535
 - generate 536
 - generation 269
 - unsupported features 535
- CDM
 - generate 280
- check model 295
 - action 315
 - activity 317
 - actor 298
 - assembly connector 329
 - association 330
 - attribute 308
 - class 298
 - class part 327
 - component 323
 - component instance 325
 - data format 325
 - data source 296
 - decision 318
 - domain 295
 - EJB 366
 - end 321
 - event 316
 - flow 322
 - generalization 311
 - identifier 305
 - input parameter 330
 - instance link 313
 - interaction reference 326
 - interface 305
 - junction point 317
 - message 313
 - node 324
 - object 312
 - object node 319
 - operation 309
 - organization unit 320
 - output parameter 330
 - package 297
 - port 328
 - realization 311
 - start 321
 - state 314
 - synchronization 321
 - transition 322
 - use case 298
- circular dependency 297
- circular inheritance 297
- class 34, 409
 - association 49
 - C# 2.0 510
 - check model 298
 - circular dependency 297
 - composite classifier 46
 - copy and paste in communication diagram 59
 - copy and paste in object diagram 59
 - copy and paste in sequence diagram 59
 - create 35, 59
 - create from an interface 35
 - drag and drop in communication diagram 59
 - drag and drop in object diagram 59
 - drag and drop in sequence diagram 59
 - EJB 361
 - generate in C# 485
 - generate in VB.NET 439
 - generate PDM 280
 - implement interface 82
 - inherited association 49
 - instantiation 59
 - migrate attribute 49
 - operation 79

Index

- part 60
- port 62
- properties 35
- realization 35, 105
- Visual Basic 2005 468
- class diagram 27
 - annotation 111
 - association 88
 - attribute 65
 - class 34
 - create a JSP 394
 - create a servlet 385
 - create an ASP.NET 458
 - create an EJB 358
 - create for component 219
 - create Web service 235
 - dependency 101
 - domain 120
 - generalization 98
 - identifier 75
 - interface 53
 - operation 78
 - part 60
 - port 62
 - realization 105
 - require link 106
 - servlet 385
- class part
 - check model 327
- class persistence 281
- classifier
 - attach to data type 47
 - attach to return type 47
 - fully qualified name 47
 - interface 46
 - operation return type 47
 - parameter data type 47
 - use case 46
- CLASSPATH 251
 - J2EE_HOME 355
 - JAVA_HOME 355
- CMP entity bean 356, 363
- code
 - comment 407
 - preview 8
- code generation 411
- comment
 - generate 353
 - Java code 407
 - Javadoc 353
 - Web service component instance 250
- committee activity 176
- communication diagram 125
 - actor 20
 - create from sequence diagram 125
 - create sequence diagram 127
 - instance link 116
 - message 137
- compilation unit 465, 507
- complex data type 281, 288
- component 214
 - check model 323
 - class diagram 219
 - create 215
 - create class diagram 219
 - deploy to node 220
 - open class diagram 219
 - part 60
 - port 62
 - properties 216
 - servlet 384
 - update class diagram 219
- component diagram 211
 - ASP.NET 457
 - component 214
 - delegation connector 109
 - dependency 101
 - EJB 355
 - generalization 98
 - JSP 393
 - part 60
 - port 62
 - servlet 384
 - Web service 232, 237
- component instance 222
 - cardinality 224
 - check model 325
 - create 223
 - properties 224
 - Web service 224, 248
- component primary identifier mapping 556
- composite activity
 - package 50
- composite classifier
 - create diagram 47
- composite primary identifier mapping 556
- composite state
 - package 50

- composite structure diagram 29
 - association 88
 - delegation connector 109
- composite view 50
 - editable mode 170, 197
 - node 220
 - read-only (sub-diagram) mode 170, 197
- composition association 88
- conditional branch 185
- constraint 70, 71
 - data format 70
- constructor 80
 - generate in C# 493
 - generate in VB.NET 446
- container
 - type 92
- conversion operator 494
- convert to decomposed 172, 199
- create associations
 - reverse engineering Java 405
- create symbols
 - reverse engineering Java 405
- custom attribute
 - C# 487
 - VB.NET 441
- custom attributes
 - C# 2.0 520
 - Visual Basic 2005 476

D

- data format 70
 - check model 325
- data grid view 664
- data profiling 70, 71
- data source 551
 - check model 296
- data type
 - default 11
 - options 11
 - parameter 47
 - Web service 238
- data type generation
 - multiplicity 291
- data type link (rebuild) 97
- de-serialization class for Web service class 238
- decision 185
 - check model 318
 - conditional branch 185
 - create 187

- merge 185
 - properties 187
- decomposed activity 170, 172
 - committee activity 176
- decomposed state 197, 199
- delegate 446, 470, 493
 - C# 2.0 512
- delegation connector 109
 - create 110
 - properties 110
- dependency 101
 - create 103
 - properties 103
- deploy
 - .NET Web services 257, 259
- deploy component to node 220
- deployment descriptor
 - EJB 369
 - JAR 369
 - reverse engineering Java 405
 - Web service 250
 - XML file 369
- deployment diagram 213
 - component instance 222
 - dependency 101
 - file object 224
 - node 220
 - node association 227
 - node diagram 222
 - Web service 248
- derivation constraint (generic type) 42
- Destroy ()message 144
- destructor 80
 - generate in C# 493
 - generate in VB.NET 446
- directives for preprocessing 454, 501
- disable swimlane mode 173
- display preferences 13
- documentation tag 495
- domain 120
 - check model 295
 - constraint 70, 71
 - create 121
 - properties 121
 - update attributes 124

E

- EEnum 409

Index

- EJB 355
 - check model 366
 - code generation 366
 - create 358
 - create from a class diagram 358
 - create from selected class 358
 - create operation from bean class 363
 - create operation from interface 363
 - define class 361
 - define interface 361
 - define operation 363
 - generate 370
 - generate JAR 375
 - generate source code 374
 - initialization 366
 - linked method 365
 - operation stereotype 363
 - persistence 373
 - properties 357
 - reverse engineering 376
 - source code 373
 - stereotype 361
 - synchronization 366
 - transaction 358
 - version 355
 - wizard 358
- EJB entity bean 356
- EJB message driven bean 356
- EJB session bean 356
 - stateful 356
 - stateless 356
- EJB Web service
 - create 260
 - svc_ejb file 262
- ejb-jar.XML 375
- EJB3 377
 - creating 377
 - properties 382
- EJB3 BeanClass
 - properties 381
- EJB3 operation
 - properties 383
- ejbCreate method 363
- ejbFinder method 363
- ejbPostCreate method 363
- ejbSelect method 363
- EMF
 - annotation 410
 - association 410
 - attribute 410
 - class 409
 - datatype 409
 - EEnum 409
 - generating code 411
 - operation 411
 - package 409
 - parameter 411
 - reference 410
 - reverse-engineering code 412
- enable swimlane mode 173
- end
 - check model 321
 - create 184
 - define 183
 - properties 184
- enterprise java bean v3.0 377
- Enterprise Java Bean Wizard 377
- entity
 - class transformation 541
- entity class
 - mapping 552
- entity/relationships
 - OOM 280
- enum 471
 - C# 2.0 513
- enum in Java 343
- enumeration
 - generate in C# 485
 - generate in VB.NET 439
- event 202
 - arguments 205
 - check model 316
 - create 203
 - design as attribute 447
 - design as operation 447
 - generate in C# 494
 - generate in VB.NET 447
 - properties 203
- event handler 448
- exception
 - synchronize 365
- export
 - XMI file 339
 - XML 339
- extension 14, 272
 - WSDL for .NET 232
 - WSDL for Java 232
- extension file 14

external method 448

F

field

C# 2.0 514

field in C# 487

file

create 225

file encoding in reverse engineering 274

file object 224

ASP.NET 458

ASP.NET stereotype 458

JSP 393

properties 226

flow 190

check model 322

create 191

properties 191

fork 188

format variable 10

G

gate (sequence diagram) 147

generalization 98, 286

check model 311

create 100

properties 100

generate

add package hierarchy 272

ADT 286

C# 269

C++ 269

CDM 280

generation 269

IDL-CORBA 269

Java BeanInfo 39

JavaBean 39

object language 269

PowerBuilder 269

table 286

VB.NET 269

WSDL 269

XML 269

generation 411

.NET commands 257

.NET options 256

.NET Web services 257

ASMX file 256

ASP.NET 460

AXIS EJB 255, 256

AXIS RPC 254

C# 496, 520

C++ 536

EJB 370, 374

IDL-CORBA 422

JAR 375

Java 399

Java Web service 262

Javadoc 353

Javadoc comment 353

JAX-RPC 252

JAXM 251

JSP 395

JSP Web deployment descriptor 395

PowerBuilder 428

server side code in .NET 256

servlet 387

servlet Web deployment descriptor 390

Stateless Session Bean Web 253

VB.NET 449

Visual Basic 2005 476

Web services 250

generation target 272

generic classifier specialization wizard 43

generic type 42

generic type (derivation constraint) 42

Getter operation 71

guard condition for decision 185

H

Hibernate

database configuration parameters 572

default options 571

entity type classes 574

extensions 571

generating code 587

generation options 588

PowerDesigner support for 571

using Ant 593

using Eclipse 590

value type classes 574

Hibernate JavaServer Faces

attribute options 616

attribute validation rules 618

computed attributes 618

default values 618

Index

- generating 628
- global page options 613
- JSF runtime environments 627
- master-detail pages 619
- testing 629
- Hibernate O/R mapping
 - association mapping 582
 - attribute mapping 581
 - basic 574
 - collections of value types 586
 - component primary identifier mapping 580
 - composite identifier mapping 579
 - inheritance mapping 587, 601
 - options 574
 - simple identifier mapping 578
- I**
- identifier 75
 - add attributes 77
 - check model 305
 - create 75
 - properties 77
- IDL
 - reverse engineering 423
- IDL-CORBA 422
 - definition file 422
 - generation 269, 422
 - objects 413
- implement
 - association 92
 - Web service method 241
- implementation
 - C# 483
 - VB .NET 437
 - Visual Basic 2005 476
 - Web service 235
- implementation class 92
- implementation operation 82
- import
 - interface WSDL in Web service component
 - instance 250
 - Rational Rose 331
 - XMI file 339
 - XML 339
- indexer parameters 489
- inheritance 74, 82, 286
 - C# 483
 - joined subclass 547
 - table per class 547
 - table per class hierarchy 547
 - transformation 547
 - VB .NET 437
 - Visual Basic 2005 476
- inheritance mapping 564
- inherited attribute 65, 286
 - PowerBuilder 431
- inherited operation 78
- initial value 74
- inner classifier 46
 - create 46
 - inner link 46
 - reverse engineering 273
- inner link 46
 - inner classifier 46
- input flow
 - fork 188
 - join 188
- input parameter
 - check model 330
- instance link 116
 - check model 313
 - create 119
 - instance of association 98
 - properties 120
- instantiation
 - class 59
- interaction activity 155
 - create 156
 - properties 156
- interaction fragment 157
 - create 157
 - manipulating 159
 - properties 157
- interaction overview diagram 136
 - interaction activity 155
 - start 182, 183, 185
- interaction reference 155
 - check model 326
 - create 155
 - properties 156
- interface 53, 511
 - attribute 65
 - check model 305
 - composite classifier 46
 - create 53
 - create a class 35
 - EJB 361
 - generate in C# 485

- generate in VB.NET 439
 - properties 53
 - realization 105
 - Visual Basic 2005 469
 - Web service 235, 239
- J**
- J2EE 355
- JAR 375
- Java 269, 343
 - annotation 111, 353
 - code comment 407
 - enum 343
 - generate 399
 - JSP 393
 - public class 343
 - reverse engineering 403
 - script 10
 - servlet 384
 - strictfp keyword 354
 - Web service support 232
- Java BeanInfo 39
 - create 39
 - generate 39
- Java class package 261
- JAVA_HOME path 355
- Java IDE 355
- Java reverse engineering
 - options 405
- Java Web service
 - create 260
 - define Java class package 261
 - generate 262
 - generate for Sybase WorkSpace 260
 - svc_java file 262
- JavaBean
 - generate 39
- JAVACLASSPATH 251
- Javadoc 351, 353
 - comments 348
- Javadoc comment 353
- JavaServer Faces
 - extension file 613
- JAX-RPC 229, 235, 252
- JAXM 229, 235, 251
- JDK 355
- JMS 356
- join 188
- joined subclass 567
- JSP 393, 394
 - file object 393
 - generate 395
 - generate WAR file 390
 - reverse engineering 398
- JSP component
 - default template 393
- JSR specification 229, 235, 250
- junction point 209
 - check model 317
 - create 209
 - properties 209
- justify variable 10
- L**
- library
 - PowerBuilder 434
 - reverse engineering Java 405
- linked method 365, 366
- Local2SOAP 238
- Local2XSD 238
- M**
- many-to-many association mapping 563
- mark classifiers
 - reverse engineering Java 405
- merge 185
- Merise 280
- message 137
 - check model 313
 - create 139
 - Create 143
 - create sequence number 149
 - decrease sequence number 151
 - Destroy 143, 144
 - increase sequence number 151
 - properties 139
 - recursive 145
 - Self-Destroy 144
 - sequence diagram 147
- method 514
 - BaseInitializer 491
 - de-serialization 241
 - delegate 446, 493
 - generate in C# 491
 - generate in VB.NET 444
 - implementation 444, 491

Index

- parameter 444, 491
- serialization 241
- shadowing 444
- ThisInitializer 491
- Visual Basic 2005 475
- migrate
 - association role 96
 - attribute 96, 286
 - column 286
 - navigable roles in association 97
- model
 - copy object language 5
 - create 5
 - diagram 3
 - options 11
 - PowerBuilder 434
 - preview code 8
 - properties 7
 - share object language 5
- model option
 - data type 11
- modeling environment
 - customize 11
- module
 - generate in VB.NET 440
- multiplicity
 - data type generation 283, 291
- N**
- namespace 272, 509
 - generate in C# 483
 - generate in VB.NET 437
 - package 51
 - Visual Basic 2005 467
 - Web service 235
- network address 250
- New (C# method) 491
- NHibernate
 - association mapping 651
 - attribute mapping 649
 - component primary identifier mapping 646
 - composite identifier mapping 646
 - database connection strings 656
 - extension file 641
 - generating code 655
 - inheritance mapping 654
 - O/R mapping 643
 - ODBC connection string 658
 - OLEDB connection string 657
 - options 642
 - Oracle connection string 658
 - simple identifier mapping 646
 - SQL Server connection string 658
- node 220
 - check model 324
 - component deployed 220
 - composite view 220
 - create 221
 - network address 250
 - node diagram 222
 - properties 221
- node association 227
 - create 227
 - properties 227
 - role 227
- node diagram 222
 - deployment diagram 222
- NUnit
 - running 661
- O**
- O/R mapping 214, 539
- object
 - check model 312
 - class instantiation 59
 - communication diagram 56
 - create 57
 - instance of class 59
 - objects diagram 56
 - OOM 56
 - properties 58
 - sequence diagram 56
- object diagram
 - dependency 101
 - instance link 116
- object language
 - C++ 536
 - generate 269
 - IDL-CORBA 422
- object node 192
 - check model 319
 - create 193
 - properties 193
- object persistence 281, 286
 - complex data type 281, 282, 286, 288
 - simple data type 281, 286
- ODBC
 - connection string 658

- OLEDB
 - connection string 657
 - one-to-many association mapping 561
 - one-to-one association mapping 560
 - OOM
 - activity diagram 131
 - changing 13
 - check model 295
 - class diagram 27
 - communication diagram 125
 - component diagram 211
 - composite structure diagram 29
 - create 5
 - deployment diagram 213
 - edit definition file 13
 - generate CDM 280
 - generate PDM 539, 550
 - interaction overview diagram 136
 - object diagram 31
 - options 11
 - overview 3
 - package diagram 33
 - sequence diagram 127
 - statechart diagram 133
 - use case diagram 17
 - operation 78
 - add 83
 - attribute 71
 - check model 309
 - class 79
 - create 79
 - duplicate 83
 - EJB 363
 - Getter 71
 - implement 244
 - override 82
 - parameters 84, 242
 - properties 84
 - return type 47, 241
 - synchronize 365
 - operator method 494
 - Oracle
 - connection string 658
 - organization unit 173–176
 - attached to activity 175
 - check model 320
 - committee activity 176
 - creating 174
 - parent organization 174
 - properties 174
 - swimlane 173
 - See also swimlane
 - Organization Unit Swimlane tool 174
 - output flow
 - fork 188
 - join 188
 - output parameter
 - check model 330
 - override 74, 82, 431
- P**
- package 50
 - check model 297
 - composite view 50
 - default diagram 52
 - hierarchy 272
 - properties 51
 - sub-package 50
 - parameter
 - in a Web service method 242
 - operation 84
 - part 60
 - create 60
 - properties 61
 - partial type
 - C# 2.0 508
 - Visual Basic 2005 467
 - PBD libraries not supported 425
 - PBL libraries 425
 - pdGenerated 94
 - PDM
 - generate from association 280
 - generate from class 280
 - generate from OOM 539, 550
 - pdRoleInfo 94
 - persistence 281
 - attribute migration 286
 - inter-model generation 35
 - OOM to CDM 281
 - OOM to PDM 286
 - persistent
 - attribute (class diagram) 282, 288
 - class generation 282, 288
 - port 62
 - check model 328
 - create 63
 - properties 63

Index

PowerBuilder

- application 429
- binary object 429
- control 429
- data window 429
- design 425
- function 429
- generation 269, 428
- libraries 429
- library 434
- load 434
- menu 429
- model 434
- objects 429
- open 434
- overriding attribute 431
- PBD not supported 425
- PBL 425
- pipe line 429
- project 429
- proxy object 429
- query 429
- reverse engineering 429, 432
- structure 429
- target/application 432
- user object 429
- window 429

preprocessing

- C# 502
- C# reverse engineering 500, 503
- directives 454
- directives (C# reverse engineering) 501
- symbol (VB .NET) 454
- VB .NET 453

preview code 8

previous version model 46

primary identifier mapping 556

procedure call

- activation 152

project

- C# 483
- VB.NET 437

property

- generate in C# 488

public class in Java 343

R

Rational Rose

- importing into an OOM 331

realization 105

- check model 311
- class 35, 105
- create 105
- interface 105
- properties 105

reflexive association 88

require link 106

- create 107
- properties 107

return type

- operation 47
- Web service method 241

reverse engineering 273

- .class 403
- .jar 403
- .java 403
- .zip 403
- C# 2.0 498
- C# 2.0 code 522
- EJB 376
- EMF code 412
- encoding 274
- IDL 423
- into an existing OOM 275
- Java 403
- Javadoc comment 353
- JSP 398
- JSP deployment descriptor 398
- new OOM 273
- options (C#) 499
- PowerBuilder 432
- preprocessing (VB .NET) 455
- servlet 391
- synonym creation 273
- target/application 432
- VB .NET 451
- Visual Basic 2005 478
- Web service 263

role

- association 90
- migrate from association 96

Rose

- importing into an OOM 331

S

script

- Java 10
- search WSDL 265

- Self-Destroy message 144
- sequence diagram 127
 - activation 137
 - actor 20
 - create communication diagram 125
 - create from communication diagram 127
 - gate 147
 - interaction fragment 157
 - interaction frame 127
 - interaction reference 155
 - message 137
 - objects 130
- sequence number
 - create in communication diagram 149
 - insert 151
 - move 149
- serialization class for Web service class 238
- servlet 384
 - class 385
 - component 384
 - create 385
 - create from class diagram 385
 - create from selected class 385
 - generate 387
 - initialization 386
 - reverse engineering 391
 - synchronization 386
 - Web service 235
 - wizard 385
- Setter operation 71
- shadows in VB.NET 441
- shortcut
 - generated as child table 297
- simple data type 281, 286
- simple primary identifier mapping 556
- Smart Device Application
 - generating code 666
- Smart Device applications
 - data grid view 664
- SOAP 229
 - fault in WSDL schema 247
 - input in WSDL schema 247
 - output in WSDL schema 247
- SOAP extension class 241
 - Web service operation 241
- specialized class 42
- SQL Server
 - connection string 658
- standard
 - data type generation 282
- Standard Component Wizard 215
- start 182
 - check model 321
 - create 183
 - properties 183
- state 194
 - check model 314
 - create 195
 - decomposed 197
 - properties 195
- statechart diagram 133
 - action 205
 - convert to decomposed state 199
 - default classifier 135
 - event 202
 - junction point 209
 - start 182, 183
 - state 194
 - transition 200
- stateless session bean
 - Web service 235
- Stateless Session Bean Web 253
- strictfp keyword 354
- struct 469
 - C# 485, 511
- structure in VB.NET 439
- sub in VB.NET 444
- sub-class generalization 98
- sub-package hierarchy 50
- super class generalization 98
- svc_ejb file 262
- svc_java file 262
- swimlane 174
 - changing format 182
 - changing orientation 181
 - copying and pasting 177
 - creating 174
 - creating links between pools 180
 - grouping 178
 - moving 177
 - organization unit 173
 - resizing 182
 - selecting symbol 176
 - ungrouping 178
 - See also organization unit
 - See also organization unit

Index

- Sybase WorkSpace
 - generate Java Web service 260
 - synchronization 188
 - change to horizontal 189
 - change to vertical 189
 - check model 321
 - create 189
 - exception 365
 - operation 365
 - properties 189
 - synchronize
 - code editor 276
 - generated file 276
 - model 276
 - synonym creation in reverse engineering 273
 - syntax variable 10
- ## T
- table per class 568
 - table per class hierarchy 564
 - target namespace
 - Web service component instance 250
 - templatecontent
 - ASP.NET 458
 - templatecontent (JSP) 394
 - test .NET Web services 259
 - tModel 229
 - traceability link 15
 - transaction
 - EJB 358
 - type 358
 - transition 200
 - check model 322
 - create 200
 - link to event 200
 - properties 201
 - trigger event
 - transition 200
- ## U
- UDDI 229
 - operator URL 265
 - version 265
 - UML
 - activity diagram 131
 - class diagram 27
 - communication diagram 125
 - component diagram 211
 - composite structure diagram 29
 - deployment diagram 213
 - interaction overview diagram 136
 - object diagram 31
 - package diagram 33
 - sequence diagram 127
 - statechart diagram 133
 - use case diagram 17
 - unit test 590
 - unit tests
 - extension file 659
 - generating code 659
 - use case 18
 - check model 298
 - create 19
 - properties 19
 - use case association
 - create 25
 - properties 25
 - use case diagram 17
 - actor 20
 - association 24
 - dependency 101
 - generalization 98
- ## V
- value type
 - transformation 543
 - variable
 - CLASSPATH 251
 - creating 169
 - format 10
 - JAVACLASSPATH 251
 - justify 10
 - properties 169
 - reading 169
 - syntax 10
 - writing to 169
 - variable in VB.NET 442
 - VB .NET
 - preprocessing directives 454
 - reverse engineering 451, 453–456
 - reverse engineering options 451
 - reverse engineering preprocessing 453
 - VB.NET
 - accessibility 438
 - annotation 111
 - attribute 442

- class 439
 - constructor 446
 - custom attributes 441
 - delegate 446
 - destructor 446
 - enumeration 439
 - event 447
 - event handler 448
 - external method 448
 - generation 269, 449
 - implementation 437
 - inheritance 437
 - interface 439
 - method 444
 - method implementation 444
 - method parameter 444
 - module 440
 - namespace 437
 - project 437
 - property 443
 - shadows 441
 - structure 439
 - sub 444
 - variable 442
 - Visual Basic 2005
 - assembly 463
 - class 468
 - compilation unit 465
 - custom attributes 476
 - delegate 470
 - enum 471
 - event 472
 - field 472
 - generation 476
 - implementation 476
 - inheritance 476
 - interface 469
 - introduction 463
 - method 475
 - namespace 467
 - partial type 467
 - property 472
 - reverse engineering 478
 - struct 469
 - VSTS
 - running 662
- W**
- WAR 390
 - Web service 229
 - component diagram 232
 - component instance 248
 - create 235
 - create from class diagram 235
 - create from selected class 235
 - create in component diagram 237
 - data type 238
 - deploy in .NET 257, 259
 - deployment descriptor 250
 - deployment diagram 248
 - generate client side code 250
 - generate EAR 250
 - generate for Sybase WorkSpace 260
 - generate in .NET 257
 - generate JAR 250
 - generate WAR 250
 - implementation 229
 - implementation class 235
 - interface 239
 - method 239
 - namespace 235
 - node 250
 - port type 239
 - properties 232
 - reverse engineering 263
 - test in .NET 259
 - type 235
 - wizard 235
 - XSD data type 238
 - Web service component 232
 - Web service interface 229
 - Web service method
 - create 239
 - extended attributes 246
 - implement 241
 - implement in .NET 246
 - implementation class 239
 - interface 239
 - operation implementation 244
 - operation parameters 242
 - return type 241
 - Web service proxy code in .NET generation tasks 257
 - web.xml 250
 - Windows applications
 - attribute display options 664
 - attribute validation rules 665
 - extension file 664

Index

- generating code 665
 - image library 664
 - wizard
 - create a JSP 394
 - create a servlet 385
 - create an ASP.NET 458
 - create an EJB 358
 - create Web service 235
 - WSDL 229
 - data type 238
 - generation 269
 - implementation 229
 - import 263
 - interface 229
 - reverse options 263
 - WSDL data type
 - select 238
 - Web service operation 241
 - WSDL editor
 - user-defined button 250
 - WSDL for .NET extension 232
 - WSDL for Java extension 232
 - WSDL schema
 - SOAP fault 247
 - SOAP input 247
 - SOAP output 247
 - WSDL URL
 - Web service component instance 249
 - WSDL2Local 238
- ## X
- xem 14
 - XMI
 - save as XML 339
 - saved as XML 339
 - XMI file
 - export 339
 - import 339
 - XML
 - designing for 527
 - export 339
 - generating for 531
 - generation 269
 - import 339
 - modeling 527
 - reverse engineering 532