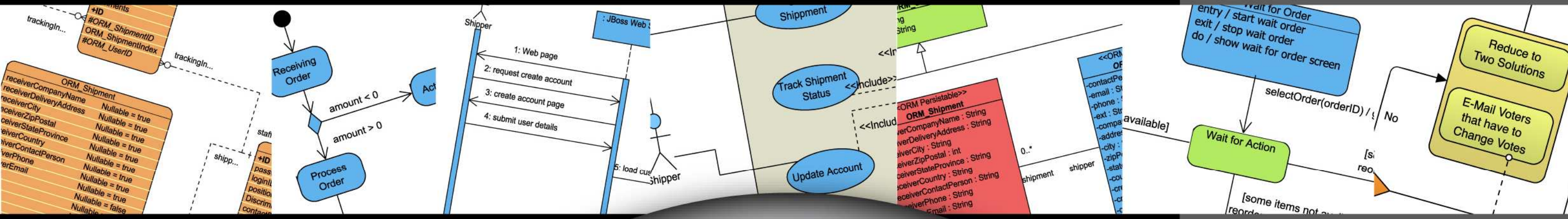
Visual Paradigm

Build Quality Applications Faster, Better and Cheaper

Visual Paradigm SDE for Java User's Guide [Part 2, for ORM]

Streamlined design and development environment



Smart Development Environment 4.0 User's Guide:
SDE 4.0 User's Guide (for ORM)

Acknowledgements

This Product includes software developed by the Apache Software Foundation (http://www.apache.org). Copyright©1999 The Apache Software Foundation. All rights reserved.

Table of Contents

Part 2 – Working with Object-Relational Mapping

Part 2 - Working with Object-Relational Mapping

Welcome to the Smart Development Environment (SDE). SDE is not only a visual UML modeling plugin, but also an Object-Relational Mapping tool supported in the Smart Development Environment Enterprise Edition (SDE EE) and the Smart Development Environment Professional Edition (SDE PE).

Being an Object-Relational Mapping tool, SDE automates the mappings between Java objects, object models, data model and relational database; it supports not only the generation of persistent code and database, but also the synchronization between persistent code, object model, data model and relational database, thus reduces the development time significantly.

SDE supports two types of object-relational mapping, that is, object-relational mapping for Java (called Java ORM) and the other for .NET (called .NET ORM). SDE PE supports Java ORM while SDE EE supports both Java ORM and .NET ORM. This part introduces how to work with object-relational mapping in SDE such that you can develop your object model and data model in an easier way.

This part is intended for database application developers and administrators, software system architects, software engineers and programmers and anyone who uses the major Integrated Development Environments (IDEs) including Eclipse, Borland JBuilder®, NetBeans/Sun™ONE, IntelliJ IDEA™Oracle JDeveloper, BEA WebLogic Workshop™Content in this part of document is based on SDE for Eclipse

In this chapter:

11 Introduction to Object-Relational Mapping

Chapter 11 - Introduction to Object-Relational Mapping

Smart Development Environment (SDE) is not only a visual UML modeling plug-in, but also an Object-Relational Mapping plug-in with IDE which supports building database application faster, better and cheaper. This chapter gives you an introduction to object-relational mapping and describes the key benefits of object-relational mapping.

In this chapter:

- Introduction
- Key Benefits

Introduction

Software applications are most likely to be developed with a database such that all data working with the application system can be retained, resulting in information and knowledge. Hence, database application is widely adopted by businesses of all sizes. In order to access and manipulate the relational database, a standard computer language, Structured Query Language (SQL) has to be used. SQL statements play an important role when developing database application.

Taking a trading system as an example, if the end-user wants to update a Sales Order record, the system has to retrieve the corresponding record from the Sales Order table and display to the end-user. After the end-user confirms the modification of data, the system has to update the record accordingly. It is noticeable that a database application requires a lot of coding for handling SQL statements so as to access and manipulate the database.

Hence, it is inevitable that developers spend almost 50% of development time for implementing the code with SQL statement. Moreover, mapping between the persistent code and database table is maintained throughout the development life cycle. Once there is a change in the structure of a database table, SQL statements which related to the modified table have to be re-written. Developers have to keep an eye on every change in the database schema.

Smart Development Environment (SDE) provides a solution to develop database application by supporting Object-Relational Mapping (ORM). Being supported with object-relational mapping, an ease-to-use environment is provided bridging between object model, data model and relational database. SDE not only provides you a visual modeling for both logical data design and physical data design, but also automates the mapping between object model and data model.

SDE also generates a cost-effective, reliable, scalable and high-performance object to relational mapping layer. The generated mapping layer includes the support of transaction, cache and other optimized feature. SDE increases the productivity and significantly reduces the risk of developing the mapping layer manually.

SDE is capable of generating Java and .NET persistent code; Smart Development Environment Enterprise Edition (SDE EE) supports both object-relational mapping for Java and .NET while Smart Development Environment Professional Edition (SDE PE) supports object-relational mapping for Java. You can thus develop your project in Java or .NET source with object-relational mapping easily.

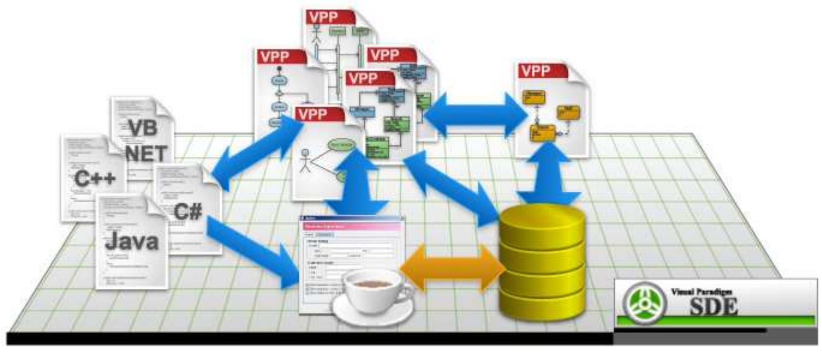
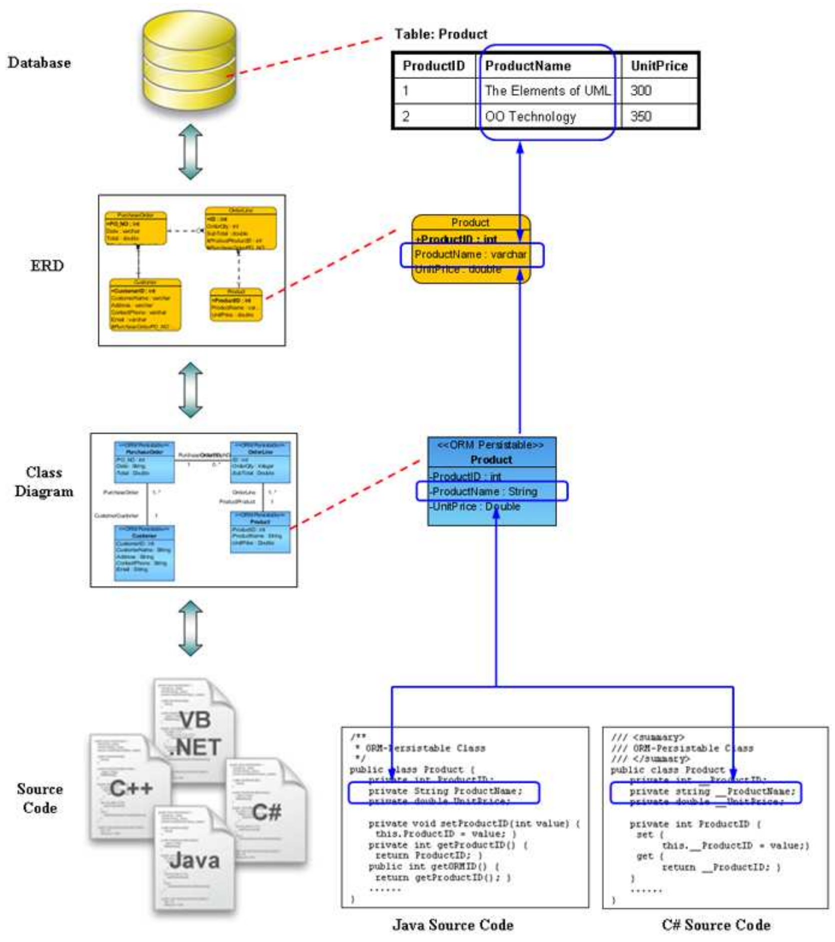


Figure 11.1 - The overview of SDE for Object-Relational Mapping

Overview Diagram



ProductID	ProductName	UnitPrice
1	The Elements of UML	300
2	OO Technology	350

```
/**
 * ORM-Persistable Class
 */
public class Product {
   private int ProductID;
   private String ProductName;
   private double UnitPrice;

   private void setProductID(int value) {
    this.ProductID = value; }
   private int getProductID() {
    return ProductID; }
   public int getORMID() {
    return getProductID(); }
   ......
}
```

```
/// <summary>
/// ORM-Persistable Class
/// </summary>
public class Product
   private int __ProductID;
   private string __ProductName;
   private double __UnitPrice;

   private int ProductID {
    set {
         this.__ProductID = value;}
    get {
         return __ProductID; }
   }
   ......
}
```

Figure 11.2 - The overview of work with database and object

Key Benefits

SDE provides the following key features of object-relational mapping so as to simplify your development:

- Persistence Made Easy

  Traditionally developers spend a lot of effort in saving and loading objects between memory and database which makes the program complicated and difficult to maintain. These tasks can be simplified by generating a persistence layer between object and data models.

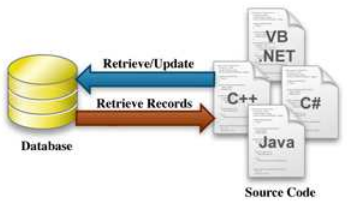


Figure 11.3 - Communication between database and object

- Sophisticated Object-Relational Mapping Generator

  DB-VA generates object-relational mapping layer which incorporates prime features such as transaction support, pluggable cache layer, connection pool and customizable SQL statement. With this mapping layer, developers can keep away from mundane implementation work and focus on the business requirements.

- Enterprise JavaBeans (EJB) Support

  EJB enables distributed, transactional, secure and portable application deployment. Since programming in EJB is a complicated task, SDE simplifies the EJB development by generating beans either from drawn stereotyped class diagram, entity relationship diagram or reversed database.

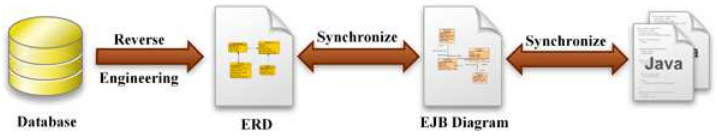


Figure 11.4 - Reverse database and synchronize to EJB

- Model Driven Development

  A true model driven platform for application development is provided. Developers are allowed, not only to start from creating the models using class diagram or entity relationship diagram and generate the executable persistence layer from the models, but also to modify the entity-relational model which is from reversed engineered an existing database, transform into object model and generate persistence layer. With the sophisticated model-code generator, the persistent model will be updated automatically according to any modification.

- Extensive Database Coverage

  A wide range of database are supported, including Oracle, DB2, Cloudscape/Derby, Sybase Adaptive Server Enterprise, Sybase SQL Anywhere, Microsoft SQL Server, PostgreSQL, MySQL and more. By enabling the same set of ORM Java objects to work with different databases, an easy migration between databases is promoted. Besides, the proprietary data type is transformed to suit the default database specified.

- Reverse Database Engineering

  Reverse engineering an existing database through JDBC and/or .NET database connector into the entity-relational model is supported. Developers can transform the entity-relational model to object model and redesign the database for further development.

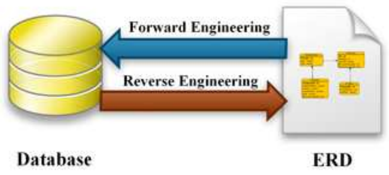


Figure 11.5 - Reverse/Forward engineering of databases

- Visual Modeling for Object and Data Models

  The visual modeling environment is inherited from Visual Paradigm for UML, a well-known UML CASE Tool, it not only provides an intuitive inline editing for both object and data models, but also adopts the resource-centric interface for assisting frequently performed tasks.

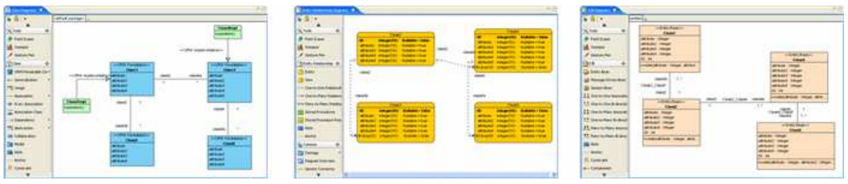


Figure 11.6 - Consistent User Interface within SDE

12 Getting Started with Object-Relational Mapping

Chapter 12 - Getting Started with Object-Relational Mapping

Connecting your database to the working environment facilitates bridging between object model, data model and relational database. This chapter shows you how to configure the database connection, automatically download the database driver files and describes the supported database driver for connecting database in the working environment.

In this chapter:

- Database Configuration
- Automatic Downloading Driver Files
- Supported Database
- Supported JDBC Drivers
- Supported .NET Drivers

Database Configuration

In order to bridge between object model, data model and relational database, you have to configure the database connection to ensure the environment.

To configure the database connection:

1. From the menu, click **Modeling** > **ORM** > **Database Configuration...**

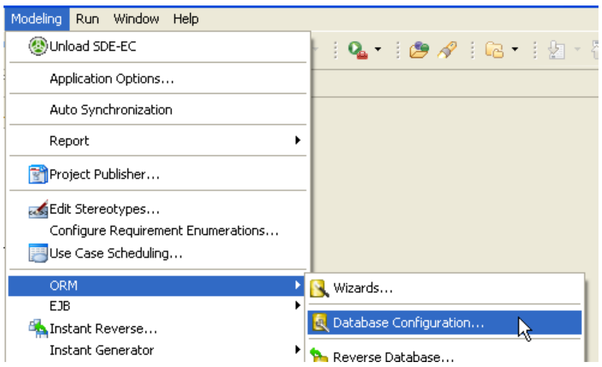


Figure 12.1 - To open the Database Configuration

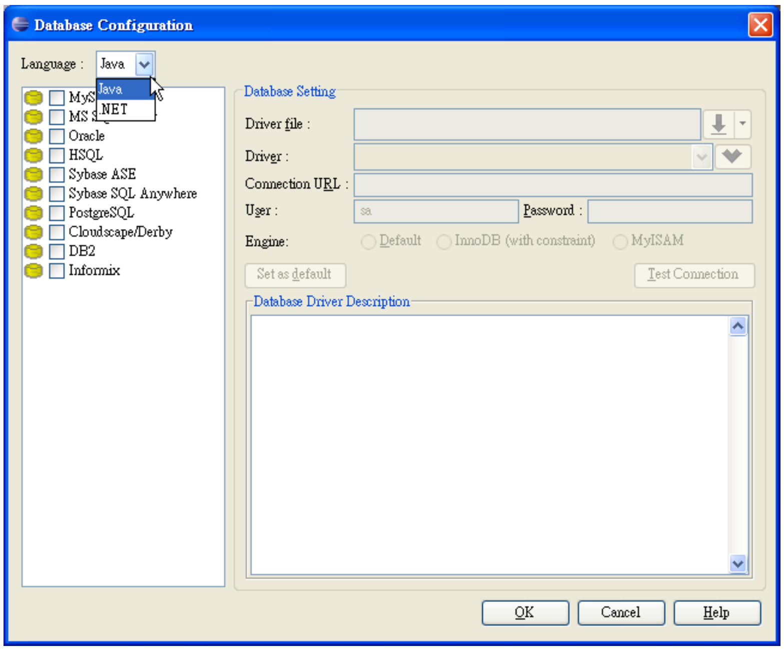
For other SDE:

SDE	Method
SDE for JBuilder	From the menu, click **Tools** > **Modeling** > **ORM** > **Database Configuration...**
SDE for NetBeans	From the menu, click **Modeling** > **ORM** > **Database Configuration...**
SDE for IntelliJ IDEA	From the menu, click **Modeling** > **ORM** > **Database Configuration...**
SDE for JDeveloper	From the menu, click **Model** > **ORM** > **Database Configuration...**
SDE for WebLogic Workshop	From the menu, click **Modeling** > **ORM** > **Database Configuration...**

Table 12.1

The **Database Configuration** dialog box is displayed.

2. Select the **Language** for the project to be developed from the drop-down menu. By default, **Java** language is selected.



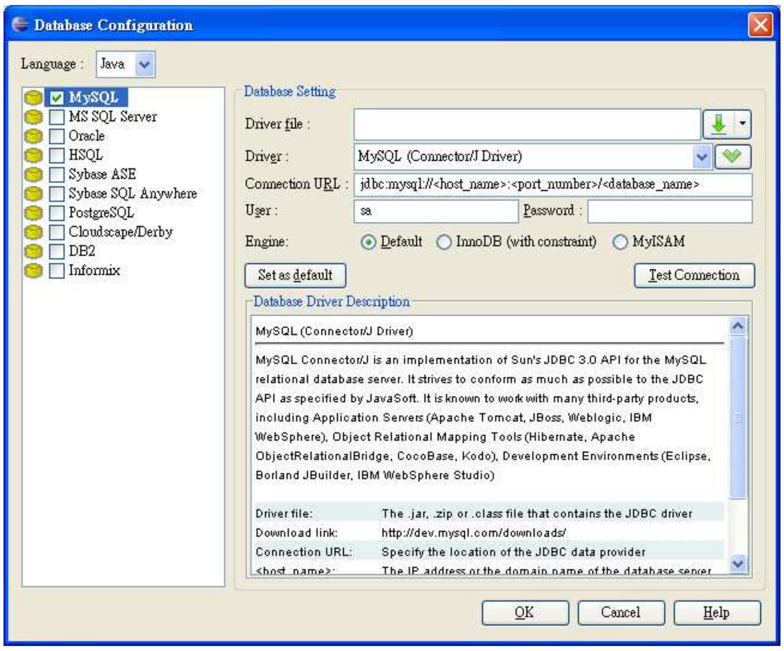
User : sa

Figure 12.2 - Select the generated programming language

Database Configuration for Java Project

For Java project development, continue the following steps for configuring the database connection.

1. Place a check mark beside the desired database for configuration.



Language : Java

☑ MySQL

Driver : MySQL (Connector/J Driver)

Connection URL : jdbc:mysql://<host_name>:<port_number>/<database_name>

User : sa

Password :

Engine: [✓ Default] InnoDB (with constraint) MyISAM

MySQL (Connector/J Driver)

MySQL Connector/J is an implementation of Sun's JDBC 3.0 API for the MySQL relational database server. It strives to conform as much as possible to the JDBC API as specified by JavaSoft. It is known to work with many third-party products, including Application Servers (Apache Tomcat, JBoss, Weblogic, IBM WebSphere), Object Relational Mapping Tools (Hibernate, Apache ObjectRelationalBridge, CocoBase, Kodo), Development Environments (Eclipse, Borland JBuilder, IBM WebSphere Studio)

Driver file:	The .jar, .zip or .class file that contains the JDBC driver
Download link:	http://dev.mysql.com/downloads/
Connection URL:	Specify the location of the JDBC data provider
<host_name>:	The IP address or the domain name of the database server

Figure 12.3 - Database setting for Java

2. Enter the database setting.

   For the **Driver file**, click button to specify the **Driver file**. The Driver file can be specified either by Download, Update or Browse. For more information, refer to the description of Specifying JDBC Driver File, .NET Driver and Adapter File section.

   For the **Driver**, select the JDBC Driver from the drop-down menu. The driver description will be shown in the **Database Driver Description** pane.

   You can press to modify the **Driver class** and **Dialect** manually.

   For the **Connection URL**, enter the information required for the JDBC Driver to connect to the database.
   For the **User**, enter the valid username who has the access right to connect to the database.
   For the **Password**, enter the corresponding password for the user to connect to the database.
   For the **Engine**, select the type of engine used in generating the MySQL database.

The **Engine** option in the **Database Setting** is only provided when configuring **MySQL** database for Java project.

3. Click the **Test Connection** button to test whether the database can be connected.

Figure 12.4 - Test Connection button

If the database can be connected, you will be prompted by a dialog box showing **Connect Successful**. Otherwise, a **Connection Exception** Dialog Box will be prompted.

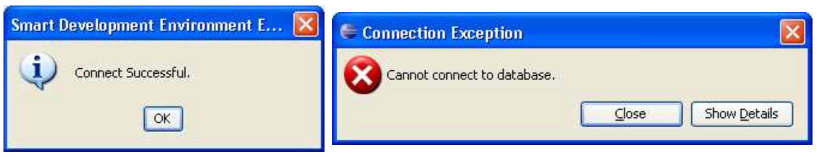

Smart Development Environment E...
Connect Successful.
OK
Connection Exception
Cannot connect to database.
Close
Show Details

Figure 12.5 - The connection successful/failure message

Configuring Multiple Database Settings

Multiple database configurations can be set up in the same project environment. To configure multiple database settings, simply place check marks beside the desired databases and specify the configuration for each database.

Generally, only one database is used for an application. As multiple database configurations are supported, only one database configuration will be applied as the default database connection for the generation of code and database.

You can set the default database connection in one of the two ways:

- Right-click on the desired database, select **Set as default**.

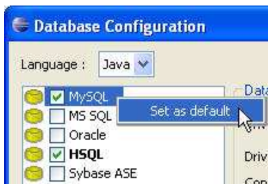


Figure 12.6 - Set the database type as default

- Select the desired database, click **Set as default** button on the **Database Setting**.

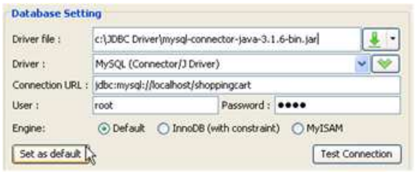

Database Setting
Driver file : c:\JDBC Driver\mysql-connector-java-3.1.6-bin.jar
Driver : MySQL (Connector/J Driver)
Connection URL : jdbc:mysql://localhost/shoppingcart
User : root Password : ●●●●
Engine: [✓ Default] InnoDB (with constraint) MyISAM

Figure 12.7 - Set the database setting as default

Database Configuration for .Net Project

For .Net project development, continue the following steps for configuring the database connection.

1. Place a check mark beside the desired database for configuration.

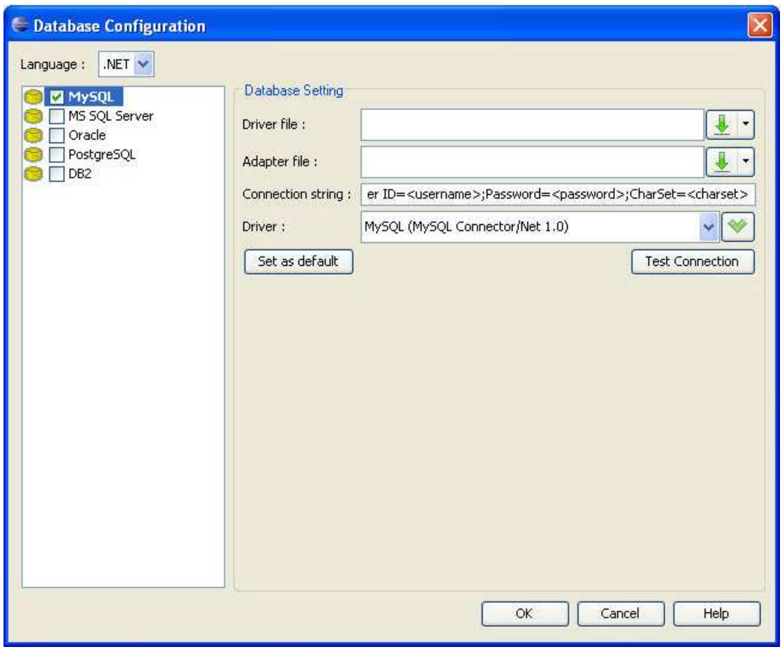

Database Configuration
Language : .NET
✓ MySQL
MS SQL Server
Oracle
PostgreSQL
DB2
Database Setting
Driver file :
Adapter file :
Connection string : er ID=<username>;Password=<password>;CharSet=<charset>
Driver : MySQL (MySQL Connector/Net 1.0)

Figure 12.8 - Database Configuration for .NET

2. Enter the database setting.

For the **Driver file**, click button to specify the **Driver file**. The .NET Driver file can be specified either by Download, Update or Browse. For more information, refer to the description of Specifying JDBC Driver File, .NET Driver and Adapter File section.

For the **Adapter file**, click button to specify the **Adapter file**. The Adapter file can be specified either by Download, Update or Browse. For more information, refer to the description of Specifying JDBC Driver File, .NET Driver and Adapter File section.

For the **Connection String**, enter the information required for the .NET Driver to connect to the database.

For the **Driver**, select the .NET Driver from the drop-down menu. The driver's description will be shown in the **Database Driver Description** pane.

You can press button to modify the **Driver class** and **Dialect** manually.

3. Click the **Test Connection** button to test whether the database can be connected.

Figure 12.9 - Test Connection button

If the database can be connected, you will be prompted by a dialog box showing **Connect Successful**. Otherwise, a **Connection Exception** Dialog Box will be prompted.

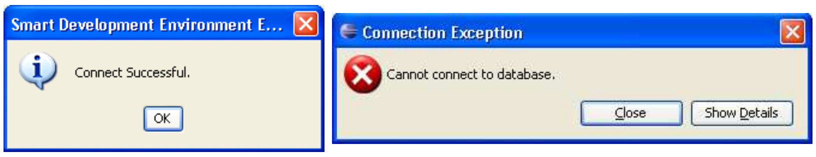

Smart Development Environment E...
Connect Successful.
OK
Connection Exception
Cannot connect to database.
Close
Show Details

Figure 12.10 - The connection successful/failure message

Configuring Multiple Database Settings

Multiple database configurations can be set up in the same project environment. To configure multiple database settings, simply place check marks beside the desired databases and specify the configuration for each database.

Generally, only one database is used for an application. As multiple database configurations are supported, only one database configuration will be applied as the default database connection for the generation of code and database.

You can set the default database connection in one of the two ways:

- Right-click on the desired database, select **Set as default**.

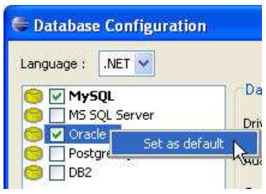
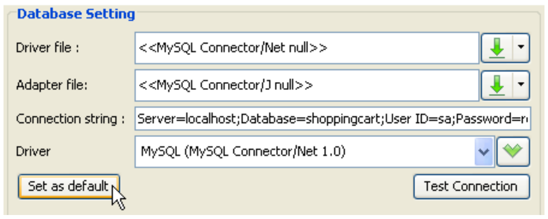


Figure 12.11 - Set the default database type

- Select the desired database, click **Set as default** button on the **Database Setting**.



Driver file :	<<MySQL Connector/Net null>>
Adapter file:	<<MySQL Connector/J null>>
Connection string :	Server=localhost;Database=shoppingcart;User ID=sa;Password=r
Driver	MySQL (MySQL Connector/Net 1.0)

Figure 12.12 - Set the database setting as default

Specifying JDBC Driver File, .NET Driver File and Adapter File

In order to connect the database successfully, JDBC driver file must be specified for Java project. Meanwhile, .NET driver file and adapter file must be specified for .NET project. You are provided with three ways to specify the driver files. They are selecting the suitable driver file manually, downloading driver files automatically and updating the driver files automatically.

Automatic download of JDBC drivers, .NET drivers and Adapter files are supported for database connection in Java and .NET project development respectively. The drivers downloaded automatically are stored in the %VP_Suite_Installation_Directory%/ormlib/driver directory.

When configuring the database connection for any new projects, SDE automatically checks if the desired driver file exists in the driver directory, the detected driver file will be specified in the database configuration automatically.

To specify the driver file, click on the button, either click **Download**, **Update** or **Browse...** from the drop-down menu.

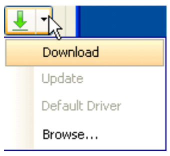


Figure 12.13 - The Driver options

- **Download**

  If Download is clicked, the download of the desired driver file for the desired database proceeds automatically. For more information on downloading driver file automatically, refer to the description of Automatic Downloading Driver File section.

- **Update**

  If Update is clicked, the update on the driver file proceeds automatically if there is an update version for the desired database.

- **Browse**

  If Browse is clicked, a File Chooser is shown, specify the location of the driver file.

Update is only available if the driver file is automatically downloaded and stored in the %VP_Suite_Installation_Directory%/ormlib/driver directory in advance.

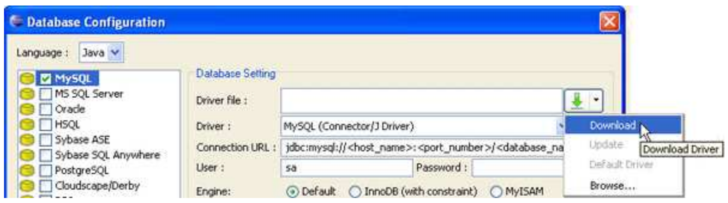
Automatic Downloading Driver files

As the automatic download of the driver files for connecting the database is supported, it reduces the effort to find the desired driver file from the Internet manually.

Automatic Downloading JDBC Driver

The following steps illustrate the automatic download of JDBC Driver for MySQL database as an example:

1. Click on the button, click **Download** from the drop-down menu.



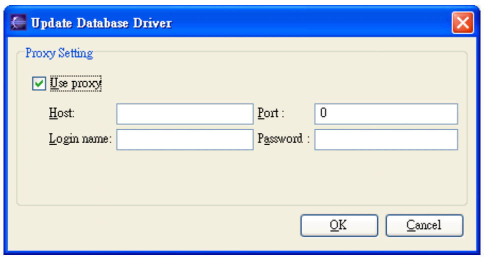
Language : Java

✓ MySQL
MS SQL Server
Oracle
HSQL
Sybase ASE
Sybase SQL Anywhere
PostgreSQL
Cloudscape/Derby

Driver file :
Driver : MySQL (Connector/J Driver)
Connection URL : jdbc:mysql://<host_name>:<port_number>/<database_na
User : sa
Password :
Engine: [✓ Default] InnoDB (with constraint) MyISAM

Figure 12.14 - Select download driver

2. A **Download Database Driver** dialog box is shown allowing the proxy setting. To enable proxy for the Internet connection, check the **Use proxy** option, and then fill in the information for proxy setting.


Update Database Driver

Proxy Setting

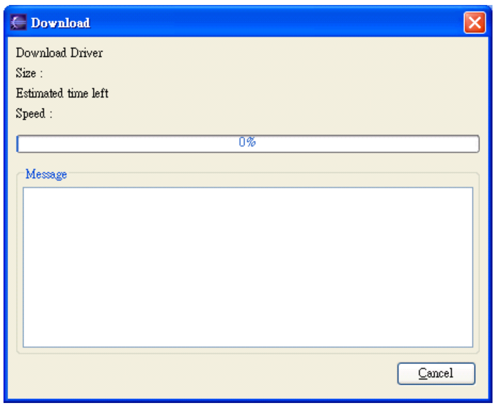
[✓] Use proxy

Host: Port : 0

Login name: Password :

Figure 12.15 - Update Database Driver dialog

The **Download** dialog box is shown indicating the download progress.


Download

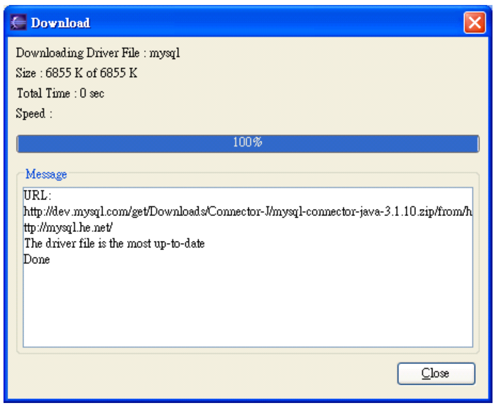
Download Driver
Size :
Estimated time left
Speed :

0%

Message

Figure 12.16 - Download dialog

3. Click **Close** when the download is completed.


Download

Downloading Driver File : mysql
Size : 6855 K of 6855 K
Total Time : 0 sec
Speed :

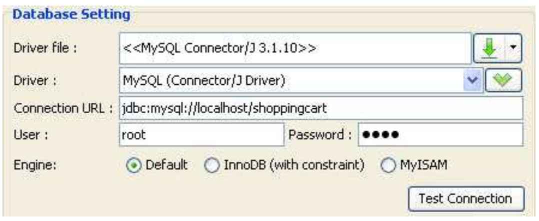
100%

Message

```
URL:
http://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java-3.1.10.zip/from/h
ttp://mysql.he.net/
The driver file is the most up-to-date
Done
```

Figure 12.17 - Download complete with the message

The driver file is shown on the **Driver file** of the **Database Setting** after download is done.


Database Setting

Driver file : <<MySQL Connector/J 3.1.10>>

Driver : MySQL (Connector/J Driver)

Connection URL : jdbc:mysql://localhost/shoppingcart

User : root Password : ●●●●

Engine: [✓ Default] InnoDB (with constraint) MyISAM

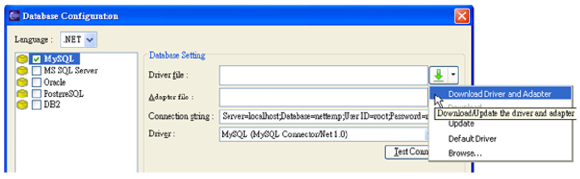
Figure 12.18 - The new database driver is ready

After downloaded the driver file, **<<MySQL Connector/J 3.1.10>>** shown on the **Driver file** indicates that the JDBC driver file is downloaded with the specified version number.

Automatic Downloading .NET Driver and Adapter File

The following steps illustrate the automatic download of .NET Driver and Adapter for MySQL database as an example:

1. Click on the button, click **Download both Driver** from the drop-down menu to download the driver and adapter files at the same time.



Database Configuration
Language : .NET
☑ MySQL
☐ MS SQL Server
☐ Oracle
☐ PostgreSQL
☐ DB2
Database Setting
Driver file :
Adapter file :
Connection string : Server=localhost;Database=nettemp;User ID=root;Password=
Driver : MySQL (MySQL Connector/Net 1.0)

Download Driver and Adapter
Download
Download/Update the driver and adapter
Update
Default Driver
Browse...
Test Conn

Figure 12.19 - Select download driver and adaptor

The **Driver file** and **Adapter file** can be downloaded separately by selecting **Download** from its drop-down menu respectively.

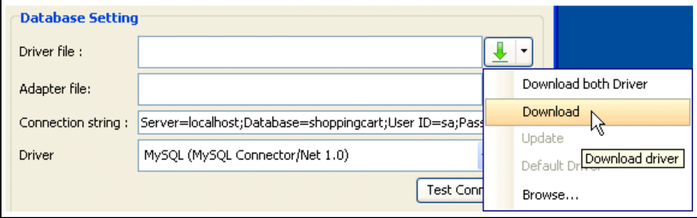
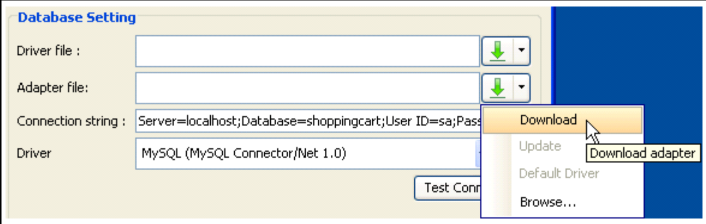
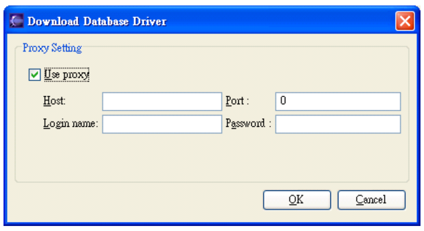
Driver file	 Database Setting Driver file : Adapter file: Connection string : Server=localhost;Database=shoppingcart;User ID=sa;Pass Driver MySQL (MySQL Connector/Net 1.0) Download both Driver Download Update Default Driver Download driver Browse... Test Conn
Adapter file	 Database Setting Driver file : Adapter file: Connection string : Server=localhost;Database=shoppingcart;User ID=sa;Pass Driver MySQL (MySQL Connector/Net 1.0) Download Update Download adapter Default Driver Browse... Test Conn

Table 12.2

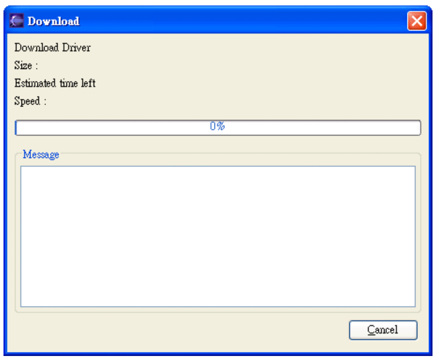
1. A **Download Database Driver** dialog box is shown allowing the proxy setting. To enable proxy for the Internet connection, check the **Use proxy** option, and then fill in the information for proxy setting.



Download Database Driver
Proxy Setting
☑ Use proxy
Host: Port : 0
Login name: Password :

Figure 12.20 - Proxy setting for Download driver

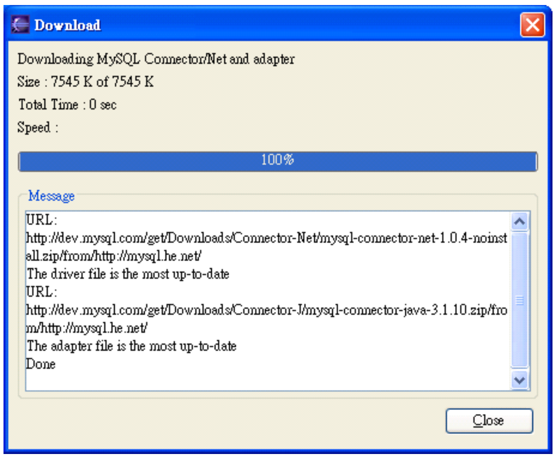
The **Download** dialog box is shown indicating the download progress.



Download
Download Driver
Size :
Estimated time left
Speed :
0%
Message

Figure 12.21 - Download dialog

2. Click **Close** when the download is complete.

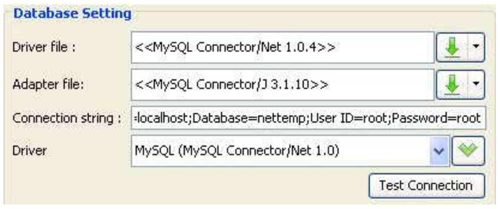


Downloading MySQL Connector/Net and adapter
Size : 7545 K of 7545 K
Total Time : 0 sec
Speed :
100%

```
URL:
http://dev.mysql.com/get/Downloads/Connector-Net/mysql-connector-net-1.0.4-noinst
all.zip/from/http://mysql.he.net/
The driver file is the most up-to-date
URL:
http://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java-3.1.10.zip/fro
m/http://mysql.he.net/
The adapter file is the most up-to-date
Done
```

Figure 12.22 - Download complete message show in Download dialog

The driver file and adapter file are shown on the **Driver file** and **Adapter file** of the **Database Setting** after download is done.



Driver file : <<MySQL Connector/Net 1.0.4>>
Adapter file: <<MySQL Connector/J 3.1.10>>
Connection string : localhost;Database=nettemp;User ID=root;Password=root
Driver MySQL (MySQL Connector/Net 1.0)

Figure 12.23 - The new driver are ready

After downloaded the driver file, **<<MySQL Connector/NET 1.0.4>>** shown on the **Driver file** indicates that the .NET driver file is downloaded with the specified version number.

After downloaded the adapter file, **<<MySQL Connector/J 3.1.10>>** shown on the **Adapter file** indicates that the adapter driver file is downloaded with the specified version number.

Supported Database, JDBC Drivers and .NET Drivers

SDE provides an environment for visual modeling the developing system. By connecting to the relational database, the mapping between models and relational database can be automated. The most common relational database are supported, including Oracle, DB2, Microsoft SQL Server, Sybase Adaptive Server Enterprise, Sybase SQL Anywhere, MySQL, HSQLDB, Cloudscape/Derby and PostgreSQL. Their relative JDBC Drivers and .NET Drivers are listed in the following tables.

In order to connect to any of the supported database, the relative JDBC and .NET Drivers are required for configuring the database connection. All of the required JDBC and .Net Drivers will not be bundled with SDE. You can get the driver files by the automatic download facility provided, or download the driver file manually. For more information on how to get the JDBC and .NET drivers manually, refer to Appendix D - JDBC and .NET Drivers.

Table shows the Supported Database and their relative JDBC Drivers.

Database Name	JDBC Drivers Name
Cloudscape/Derby 10	Cloudscape/Derby (Embedded), Cloudscape/Derby (Server)
DB2 7/8	DB2 (AS/400 Toolbox for Java JDBC Driver) DB2 (App Driver) DB2 (Net Driver)
HSQLDB 1.61-1.8	HSQLDB (In-process) HSQLDB (Server)
IBM Informix	IBM Informix (Client) IBM Informix (Server)

MS SQL Server 2000	MS SQL Server (DataDirect SequeLink Driver) MS SQL Server (JSQL Driver) MS SQL Server (JTURBO Driver) MS SQL Server (Microsoft Driver) MS SQL Server (WebLogic Connect Driver) MS SQL Server (WebSphere Connect Driver) MS SQL Server (jTDS Driver)
MySQL 3/4	MySQL (Connector/J Driver)
Oracle	Oracle (DataDirect SequeLink Driver)
Oracle 8i	Oracle8i (THIN JDBC Driver)
Oracle 9i	Oracle9i
PostgreSQL	PostgreSQL
Sybase Adaptive Server Enterprise 12.5	Sybase Adaptive Server Enterprise (jConnect Driver) Sybase Adaptive Server Enterprise (JDS Driver)
Sybase SQL Anywhere 9	Sybase SQL Anywhere (jConnect Driver)

Table 12.3

Table shows the Supported Database and their relative .NET Drivers.

Database Name	.NET Drivers Name
DB2 7/8	DB2 (DB2 UDB for iSeries .NET Data Provider)
MS SQL Server 2000	MS SQL Server
MySQL 3/4	MySQL (MySQL Connector/Net 1.0)
Oracle	Oracle (.NET Framework Data Provider) Oracle (Oracle Data Provider for .NET)
PostgreSQL	Postgre (Npgsql)

Table 12.4

Supporting Multiple Database

As multiple databases are supported and you are allowed to configure multiple database settings, there may be differences in the data type supported by these databases.

Assigning Data Types from Multiple Database

You are allowed to specify the data type of the column in the database by using the drop-down menu of **Type** in the **Column Specification** dialog box. By default, a list of pre-defined data types which is general to all databases is provided.

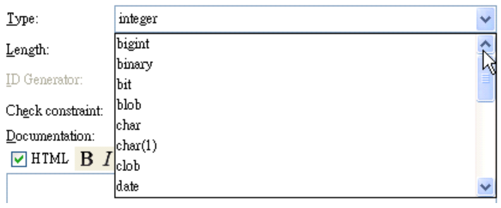


Figure 12.24 - Pre-defined data types

You are also allowed to assign a data type which is database-specific.

1. Place a check mark beside the desired database in the **Database Configuration** dialog box.

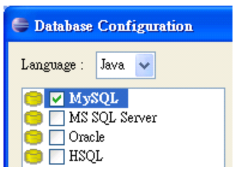


Figure 12.25 - Select database

The database-specific data types will be automatically added to the list.

2. Select the database-specific data types from the drop-down menu. For example, data type, tinyint is specific to MySQL database.

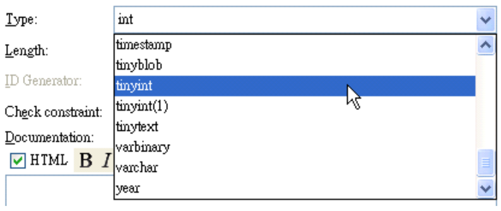


Figure 12.26 - Database-specific data types

If you have checked multiple databases in the **Database Configuration** dialog box, all data types supported by these databases will be added as an option for the drop-down menu.

Porting to Other Database

The generated persistent code, which maps to relational database tables, is capable of manipulating different databases. By default, the persistent code works with the database configured as default connection. On the other hand, you are allowed to port the persistent code to work with other databases by adding a few lines of code.

```
JDBCSettings setting = new JDBCSettings();
JDBCConnectionSetting connectionSetting =
setting.getDefaultJDBCConnectionSetting(JDBCSettings_Constant);
```

The *JDBCSettings_Constant* is the constant for JDBC settings. Modify it by the corresponding constant value stated in the following table.

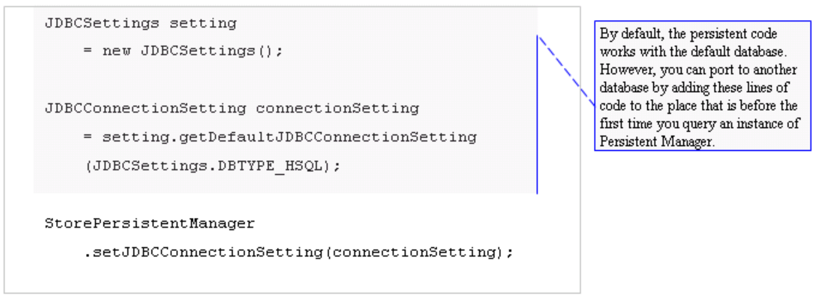
Table shows the constant for JDBC Settings for different database.

Database	Constant for JDBC Settings
MySQL	JDBCSettings.DBTYPE_MY_SQL
HSQLDB	JDBCSettings.DBTYPE_HSQL
MY SQL Server	JDBCSettings.DBTYPE_MS_SQL
Oracle	JDBCSettings.DBTYPE_ORACLE
Sybase ASE	JDBCSettings.DBTYPE_SYBASE
Sybase SQL Anywhere	JDBCSettings.DBTYPE_SYBASE_ANYWHERE

Table 12.5

Porting to Database is only supported in Java project.

Example:



```
JDBCSettings setting
    = new JDBCSettings();

JDBCConnectionSetting connectionSetting
    = setting.getDefaultJDBCConnectionSetting
    (JDBCSettings.DBTYPE_HSQL);

StorePersistentManager
    .setJDBCConnectionSetting(connectionSetting);
```

By default, the persistent code works with the default database. However, you can port to another database by adding these lines of code to the place that is before the first time you query an instance of Persistent Manager.

Figure 12.27 - Example for using JDBCSettings constant

After adding these lines of code, the default database connection will be replaced by the database specified by the *JDBCSettings_Constant*.

Displaying Data Type based on Default Database

As SDE provides a visual data modeling of the database depicted by the Entity Relationship Diagram (ERD), you are allowed to enable and disable the display of data type for columns of the entities in the ERD. Since the default database is configured in the working environment, the data type will be displayed according to the data type supported by the default database.

To display the data type for columns of entities in the ERD:

1. Right-click on the background of the ERD, select **Show Column Types**.

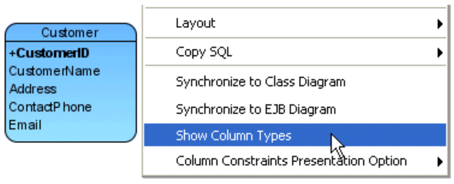

Customer
+CustomerID
CustomerName
Address
ContactPhone
Email
Layout
Copy SQL
Synchronize to Class Diagram
Synchronize to EJB Diagram
Show Column Types
Column Constraints Presentation Option

Figure 12.28 - To show the column types

The data type for columns is displayed.

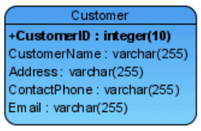

Customer
+CustomerID : integer(10)
CustomerName : varchar(255)
Address : varchar(255)
ContactPhone : varchar(255)
Email : varchar(255)

Figure 12.29 - Entity with column types

If the default database connection is changed, the data types for all columns will be changed with respect to the new database connection automatically.

Example:

There is an entity, Customer in the ERD. Modify the default database connection from MySQL to Oracle, the data types will be changed automatically.

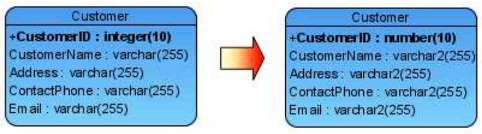

Customer
+CustomerID : integer(10)
CustomerName : varchar(255)
Address : varchar(255)
ContactPhone : varchar(255)
Email : varchar(255)
Customer
+CustomerID : number(10)
CustomerName : varchar2(255)
Address : varchar2(255)
ContactPhone : varchar2(255)
Email : varchar2(255)

Figure 12.30 - Modify the data type

13 Using ORM Wizard

Chapter 13 - Using ORM Wizard

An ORM Wizard is provided to you to generate persistent code and/or database either from database, class diagram or entity relationship diagrams. This chapter shows you how to activate the ORM Wizard and generate code and/or database by one of the three options provided by wizard.

In this chapter:

- Introduction
- Generating Code from Database
- Generating Code and Database from ERD
- Generating Code and Database from Class Diagram

Introduction

Mapping objects to relational database is a complicated and error pound task in the development. Acting as a bridge between object model, data model and relational database, the mappings between these models are automated. Apart from these mappings, the mapping between object models and persistent code is also supported. Hence, the persistent code can thus map to the relational database.

The synchronization between persistent code, object model, data model and relational is supported, which reduces the development time for handling these tedious programming jobs between them. Moreover, your document will always keep up-to-date. To support synchronization in-between persistent code and relational database, you are allowed to generate database and persistent code for your development project.

An ORM Wizard for the generation of persistent code and database. The wizard provides you with three options:

1. **Generate Code from Database**.
2. **Generate Code and Database from Entity Relationship Diagram (ERD).**
3. **Generate Code and Database from Class Diagram**.

To activate the Wizard:

1. On the menu, click **Modeling** > **ORM** > **Wizards...**.

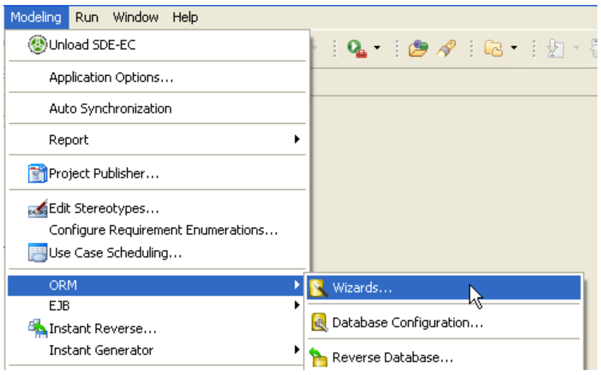


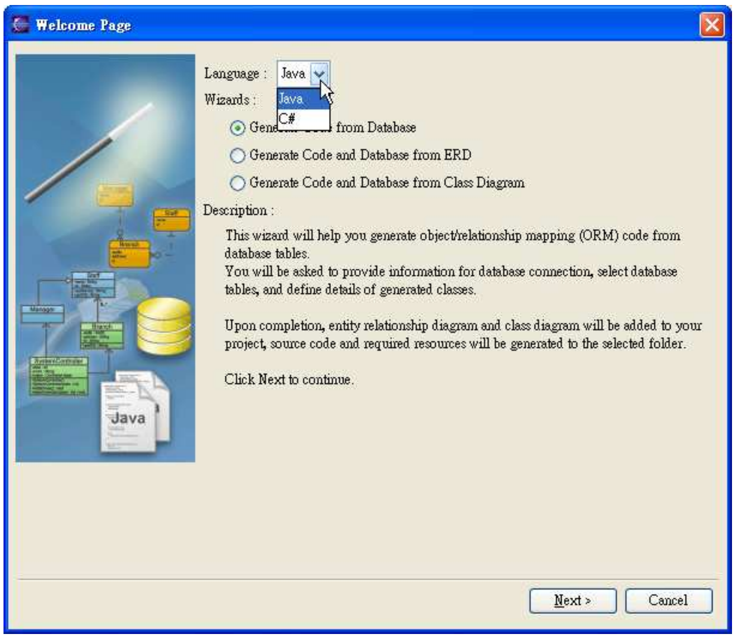
Figure 13.1 - To start the ORM Wizard

For other SDE:

SDE	Method
SDE for JBuilder	On the menu, click **Tools** > **Modeling** > **ORM** > **Wizards...**.
SDE for NetBeans	On the menu, click **Modeling** > **ORM** > **Wizards...**.
SDE for IntelliJ IDEA	On the menu, click **Modeling** > **ORM** > **Wizards...**.
SDE for JDeveloper	On the menu, click **Model** > **ORM** > **Wizards...**.
SDE for WebLogic Workshop	On the menu, click **Modeling** > **ORM** > **Wizards...**.

Table 13.1

2. A Wizard Welcome Page will be shown, select **Language** of the code to be generated from the drop-down menu, either **Java** or **C#.**



Language : Java

Wizards :

- Generate Code from Database
- Generate Code and Database from ERD
- Generate Code and Database from Class Diagram

Description :

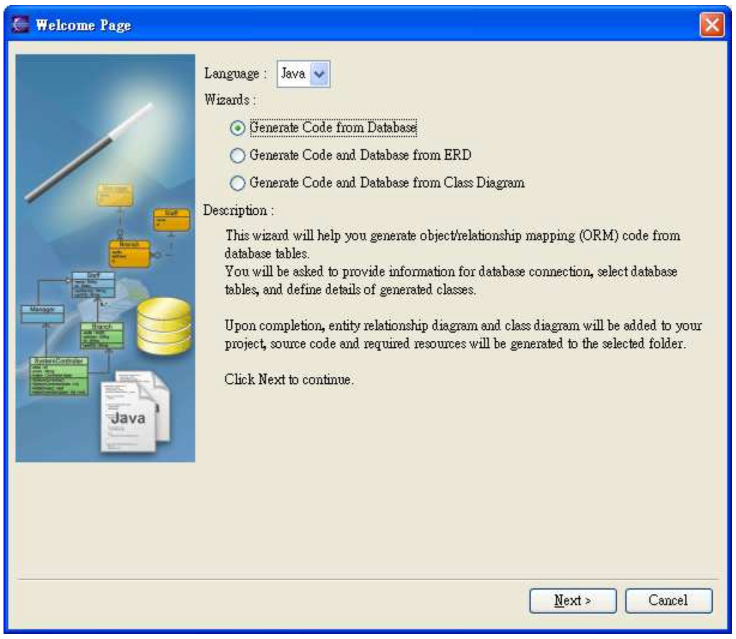
This wizard will help you generate object/relationship mapping (ORM) code from database tables.
You will be asked to provide information for database connection, select database tables, and define details of generated classes.

Upon completion, entity relationship diagram and class diagram will be added to your project, source code and required resources will be generated to the selected folder.

Click Next to continue.

Figure 13.2 - Select the programming language

3. Select one of the wizard options, and then click **Next** to proceed.



Language : Java

Wizards :

- [✓ Generate Code from Database]
- Generate Code and Database from ERD
- Generate Code and Database from Class Diagram

Description :

This wizard will help you generate object/relationship mapping (ORM) code from database tables.
You will be asked to provide information for database connection, select database tables, and define details of generated classes.

Upon completion, entity relationship diagram and class diagram will be added to your project, source code and required resources will be generated to the selected folder.

Click Next to continue.

Figure 13.3 - Select the generate options

Generating Code from Database

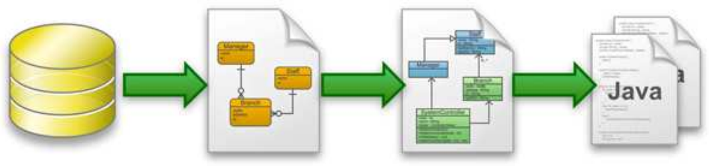
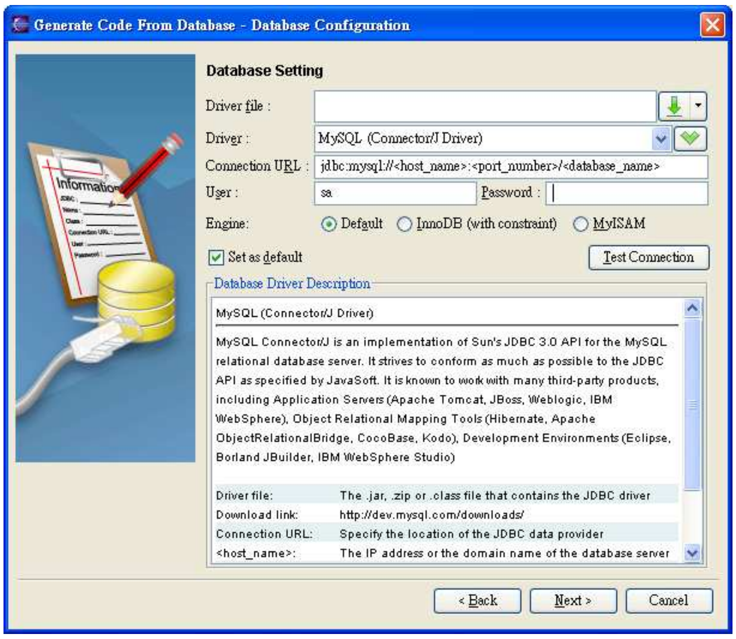


Figure 13.4 - The workflow of generate code from database

Upon selecting the option for **Generate Code from Database**, the wizard helps you generate persistent code from database tables according to the requirements you specified throughout the wizard.

Follow the steps of the **Generate Code From Database** wizard:

1. Database Configuration
    - For Java Language selected



Generate Code From Database - Database Configuration

Database Setting

Driver file :	
Driver :	MySQL (Connector/J Driver)
Connection URL :	jdbc:mysql://<host_name>:<port_number>/<database_name>
User :	sa
Password :	
Engine:	[✓ Default] InnoDB (with constraint) MyISAM

[✓] Set as default

Database Driver Description

MySQL (Connector/J Driver)

MySQL Connector/J is an implementation of Sun's JDBC 3.0 API for the MySQL relational database server. It strives to conform as much as possible to the JDBC API as specified by JavaSoft. It is known to work with many third-party products, including Application Servers (Apache Tomcat, JBoss, Weblogic, IBM WebSphere), Object Relational Mapping Tools (Hibernate, Apache ObjectRelationalBridge, CocoBase, Kodo), Development Environments (Eclipse, Borland JBuilder, IBM WebSphere Studio)

Driver file:	The .jar, .zip or .class file that contains the JDBC driver
Download link:	http://dev.mysql.com/downloads/
Connection URL:	Specify the location of the JDBC data provider
<host_name>:	The IP address or the domain name of the database server

Figure 13.5 - Database Configuration for Java

You are asked to define the database configuration. Refer to the descriptions in the Getting Started with Object-Relational Mapping chapter for information chapter for information on how to configure the database Database Configuration for Java Project section.

- For C# Language selected

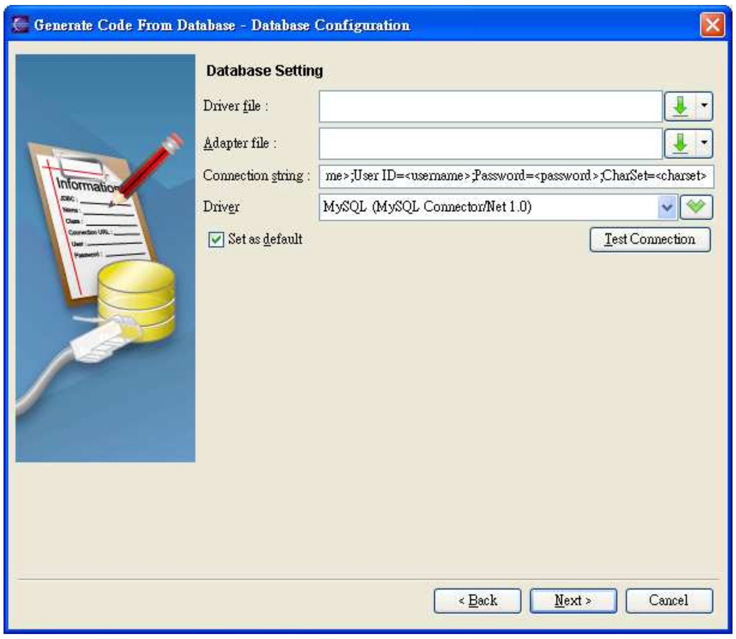

Generate Code From Database - Database Configuration
Database Setting
Driver file :
Adapter file :
Connection string : me>;User ID=<username>;Password=<password>;CharSet=<charset>
Driver MySQL (MySQL Connector/Net 1.0)
Set as default
Test Connection
< Back
Next >
Cancel

Figure 13.6 - Database Configuration for C#

You are asked to define the database configuration. Refer to the descriptions in the Getting Started with Object-Relational Mapping chapter for information on how to configure the database in the Database Configuration for .Net Project section.

2. Selecting Tables

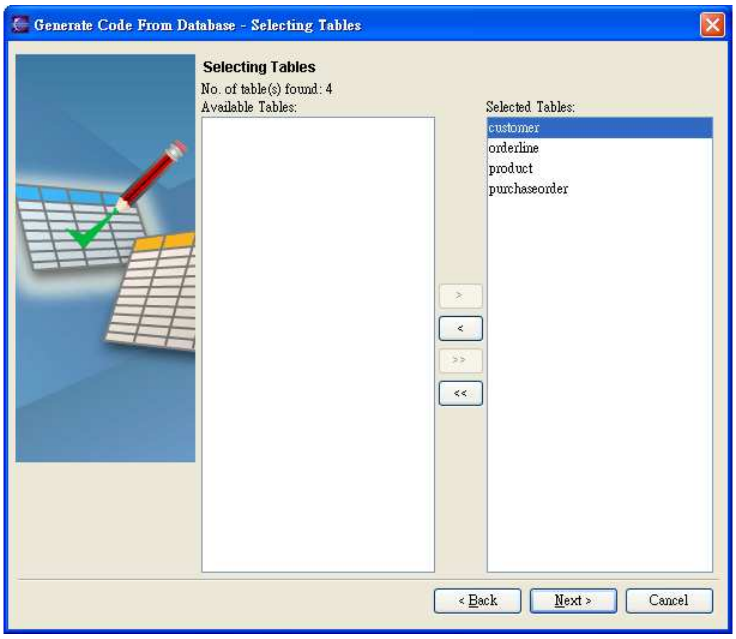


Figure 13.7 - Select the Tables

The database is connected based on your options in the previous database configuration option pane and all database tables are reversed. You are asked to select the database tables which you want to generate persistent class to manipulate those tables. By default, all the database tables are selected for the generation of code shown in the list of **Selected Tables**. You can deselect the table by using the list of buttons between the list of **Available Tables** and **Selected Tables**.

- > Add Selected

  Add the selected table from **Available Tables** to **Selected Tables**.

- < Remove Selected

  Remove the selected table from **Selected Tables** to **Available Tables**.

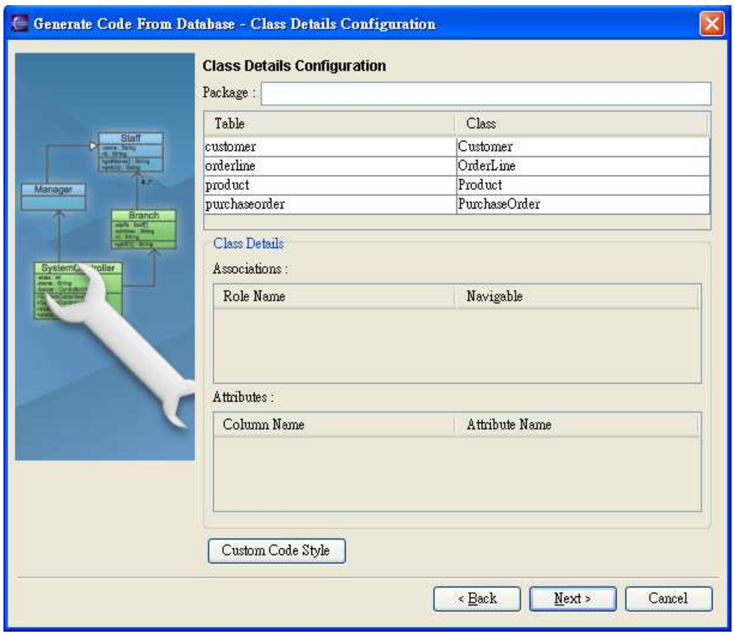
- >> Add All

  Add all tables from **Available Tables** to **Selected Tables**.

- << Remove All

  Remove all tables from **Selected Tables** to **Available Tables**.

3. Class Details Configuration



Generate Code From Database - Class Details Configuration

Class Details Configuration

Package :

Table	Class
customer	Customer
orderline	OrderLine
product	Product
purchaseorder	PurchaseOrder

Class Details

Associations :

Role Name	Navigable

Attributes :

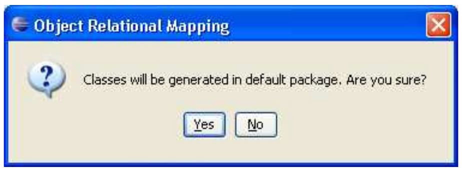
Column Name	Attribute Name

Figure 13.8 - Class Detail Configuration

After selecting tables, you will be directed to a Class Details Configuration pane. You are asked to define the Class Detail for generating code. The persistent classes will be generated based on the information defined here. You can edit the class details by double-clicking the field.

- **Package**

  Enter the package name. A package will be created to store the generated persistent code. If the package name was not defined, you will be prompted by a dialog box warning you the classes will be generated in default package.



Object Relational Mapping

Classes will be generated in default package. Are you sure?

Figure 13.9 - Confirm generate code in default package

- **Class**

  You can edit the class name which will be used as the name of the generated persistent code for a corresponding table.

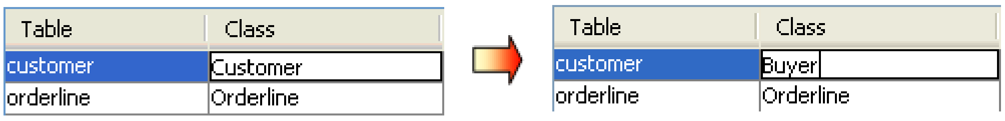


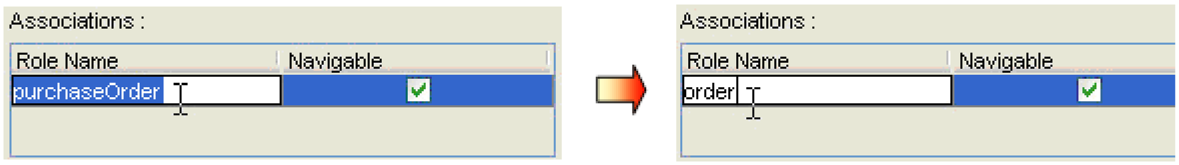
Table	Class
customer	Customer
orderline	Orderline

Table	Class
customer	Buyer
orderline	Orderline

Figure 13.10 - Mapping classes

- **Associations**

  You can edit the role name for a reference in the class.



Associations :

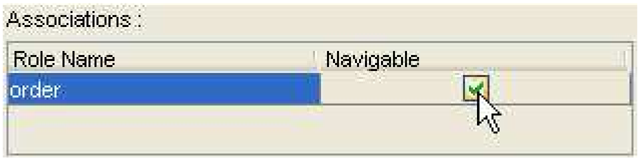
Role Name	Navigable
purchaseOrder	✓

Associations :

Role Name	Navigable
order	✓

Figure 13.11 - Mapping association

You can deselect navigable for an association such that the reference for the target role will not be created.



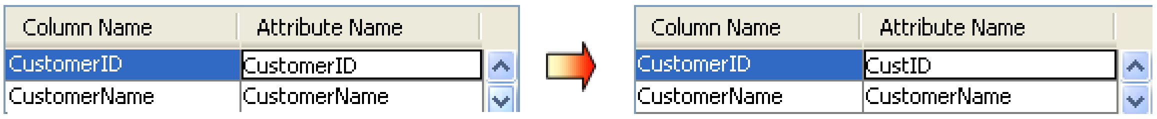
Associations :

Role Name	Navigable
order	✓

Figure 13.12 - Edit the navigable of association

- **Attributes**

  You can edit the attribute name representing the column of the table.



Column Name	Attribute Name
CustomerID	CustomerID
CustomerName	CustomerName

Column Name	Attribute Name
CustomerID	CustID
CustomerName	CustomerName

Figure 13.13 - Mapping attributes

-

- **Custom Code Style**

Click [Custom Code Style] button, **Custom Code Style Setting** dialog box will be displayed. You can modify the prefix or suffix of the **Class**, **Attribute** and **Role Name**.

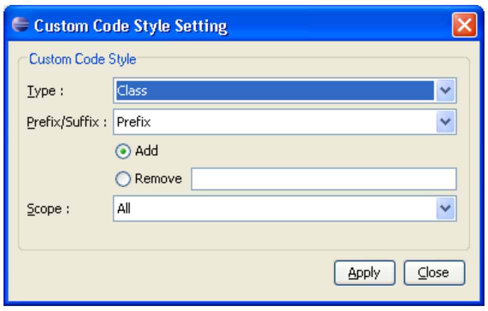

Custom Code Style Setting
Custom Code Style
Type : Class
Prefix/Suffix : Prefix
(● Add) ○ Remove
Scope : All

Figure 13.14 - Customer Code Style Setting dialog

For the **Type**, select the type of Class detail, either Class, Attribute or Role Name (PK) that you want to apply code style.
For the **Prefix/Suffix**, select either Prefix or Suffix to be added or removed.
For the **Add/Remove option**, select the option for the action of code style to be applied.
For the **Textbox**, enter the word for either prefix or suffix.
For the **Scope**, select the scope of the code style to be applied to, either All or Selected.

Table shows the result of applying Code Style.

Code Style	Before Applying	After Applying
Add Prefix (E.g. pre_)	Item	pre_Item
Remove Prefix (E.g. pre_)	pre_Item	Item
Add Suffix (E.g. _suf)	Item	Item_suf
Remove (E.g. _suf)	Item_suf	Item

Table 13.2

4. Generate Code
   - For Java Language selected

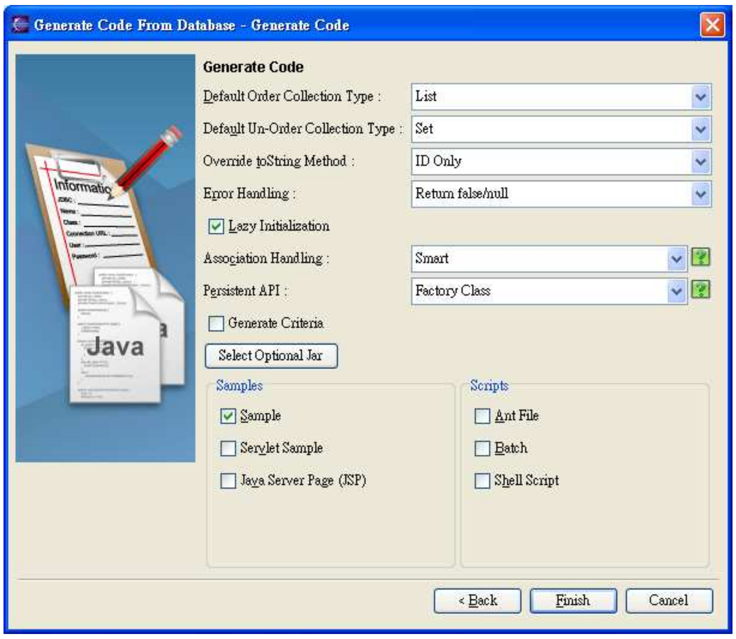
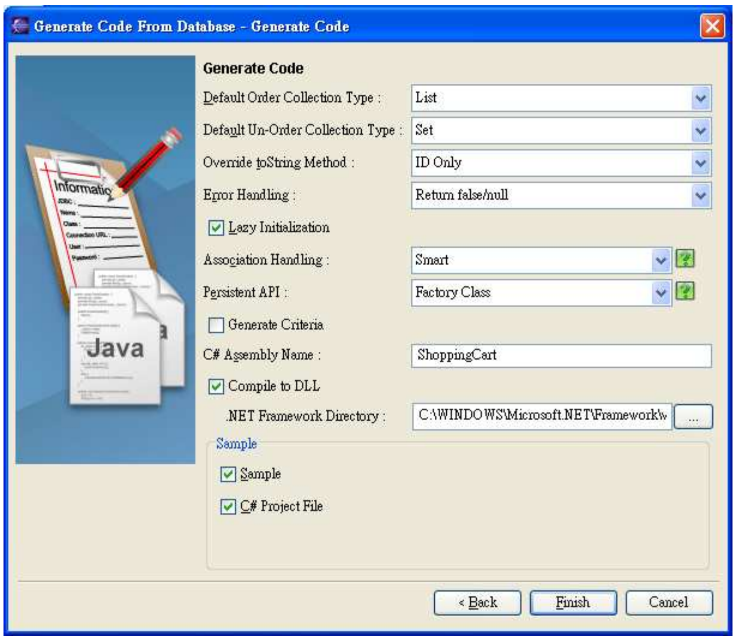

Generate Code From Database - Generate Code
Generate Code
Default Order Collection Type : List
Default Un-Order Collection Type : Set
Override toString Method : ID Only
Error Handling : Return false/null
[✓] Lazy Initialization
Association Handling : Smart
Persistent API : Factory Class
[ ] Generate Criteria
Samples: [✓] Sample, [ ] Servlet Sample, [ ] Java Server Page (JSP)
Scripts: [ ] Ant File, [ ] Batch, [ ] Shell Script

Figure 13.15 - Generate Java Code Setting

You are asked to specify the code details. For information, refer to the description of Configuring Code Generation Setting for Java section in the Implementation chapter.

- For C# Language selected



Generate Code From Database - Generate Code

Generate Code

Default Order Collection Type : List

Default Un-Order Collection Type : Set

Override toString Method : ID Only

Error Handling : Return false/null

[✓] Lazy Initialization

Association Handling : Smart

Persistent API : Factory Class

[ ] Generate Criteria

C# Assembly Name : ShoppingCart

[✓] Compile to DLL

.NET Framework Directory : C:\WINDOWS\Microsoft.NET\Framework\v

Sample

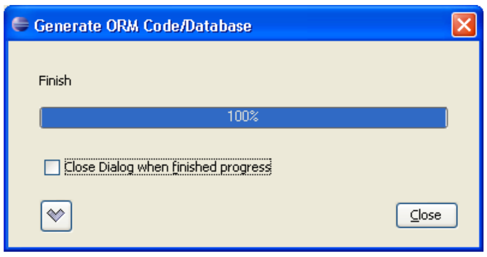
[✓] Sample

[✓] C# Project File

Figure 13.16 - Generate Code Setting for C#

You are asked to specify the code details. For information, refer to the description of Configuring Code Generation Setting for C# section in the Implementation chapter.

5. Click **Finish**, the **Generate ORM Code/Database** dialog box appears showing the progress of code generation. Click **Close** when the generation is complete.



Generate ORM Code/Database

Finish

100%

[ ] Close Dialog when finished progress

Figure 13.17 - Generate ORM Code/Database dialog

A class diagram and an entity relationship diagram will be generated automatically and added to your project. The generated persistent code and required resources will be generated to the specified output path.

Generating Code and Database from ERD

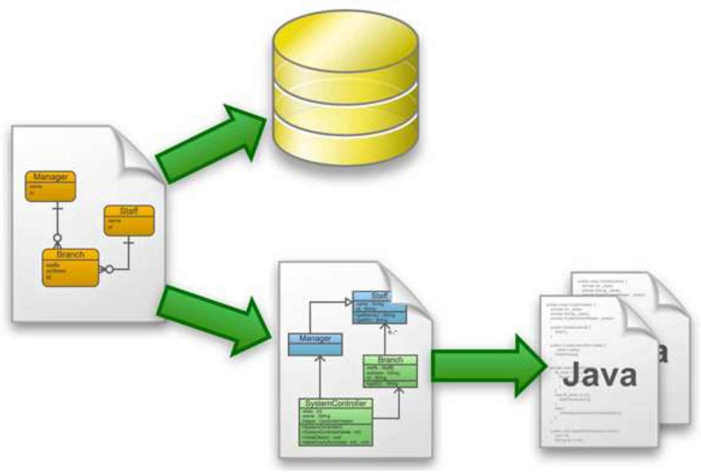


Figure 13.18 - Generate code and database from ERD

Upon selecting the option for Generate Code and Database from ERD, the wizard helps you generate persistent code and database from ERD with respect to the requirements you specified throughout the wizard.

Take the following ERD as an example:

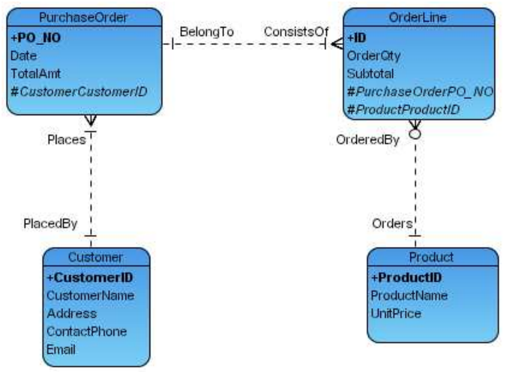
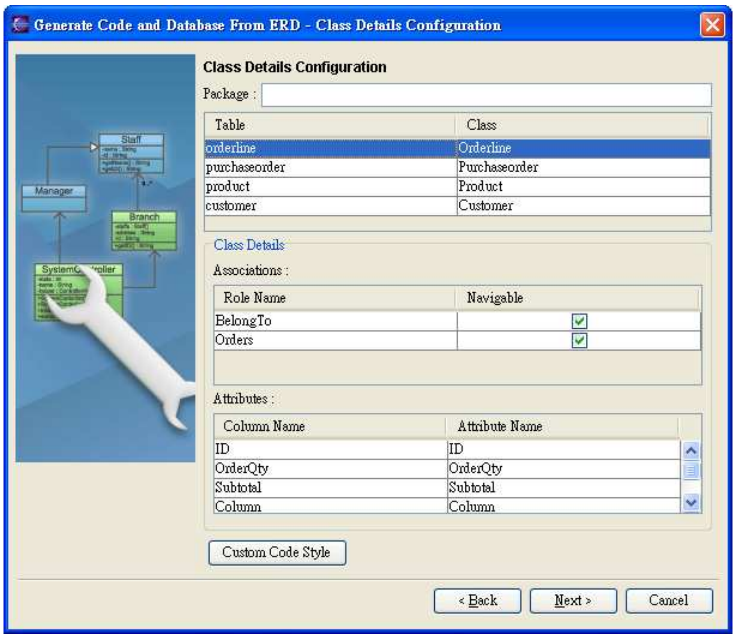


Figure 13.19 - Entity Relationship Diagram (ERD)

Follow the steps of the Generate Code and Database From ERD wizard:

1. Class Details Configuration


Generate Code and Database From ERD - Class Details Configuration
Custom Code Style
< Back
Next >
Cancel

Class Details Configuration

Package :

Table	Class
orderline	Orderline
purchaseorder	Purchaseorder
product	Product
customer	Customer

Class Details

Associations :

Role Name	Navigable
BelongTo	✓
Orders	✓

Attributes :

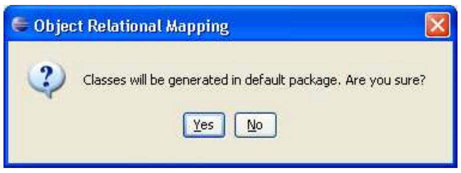
Column Name	Attribute Name
ID	ID
OrderQty	OrderQty
Subtotal	Subtotal
Column	Column

Figure 13.20 - Class Details Configuration

You are asked to define the Class Details for generating code. The persistent classes will be generated based on the information defined here. You can edit the class details by double-clicking the field.

- **Package**

  Enter the package name. A package will be created to store the generated persistent code. If the package name was not defined, you will be prompted by a dialog box warning you the classes will be generated in default package.



Classes will be generated in default package. Are you sure?

Figure 13.21 - Confirm generate code in default package

- **Class**

  You can edit the class name which will be used as the name of the generated persistent code for a corresponding table.

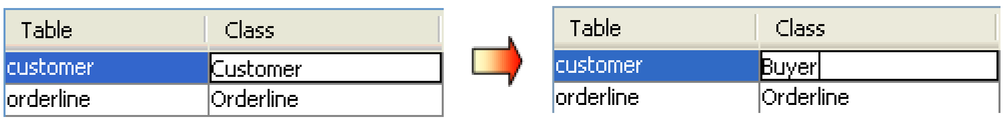


Table	Class
customer	Customer
orderline	Orderline

Table	Class
customer	Buyer
orderline	Orderline

Figure 13.22 - Mapping classes

- **Associations**

You can edit the role name for a reference in the class.

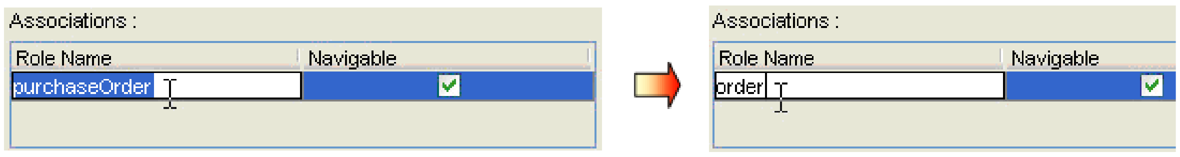


Figure 13.23 - Mapping associations

You can deselect navigable for an association such that the reference for the target role will not be created.

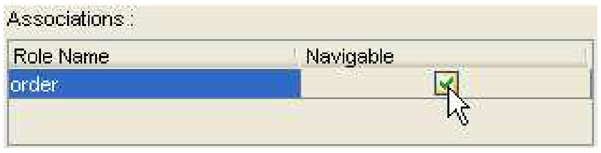


Figure 13.34 - Select the navigable of associations

- **Attributes**

You can edit the attribute name representing the column of the table.

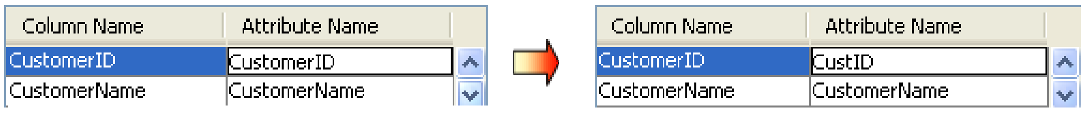
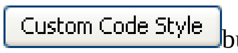


Figure 13.35 - Mapping attributes

- **Custom Code Style**

Click  button, Custom Code Style Setting dialog box will be displayed. You can modify the prefix or suffix of the Class, Attribute and Role Name.

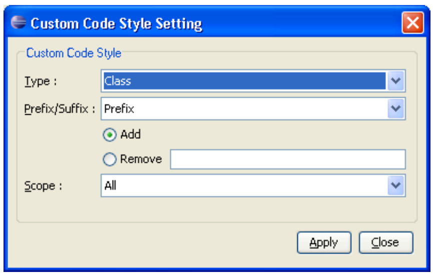


Figure 13.36 - Custom Code Style Setting dialog

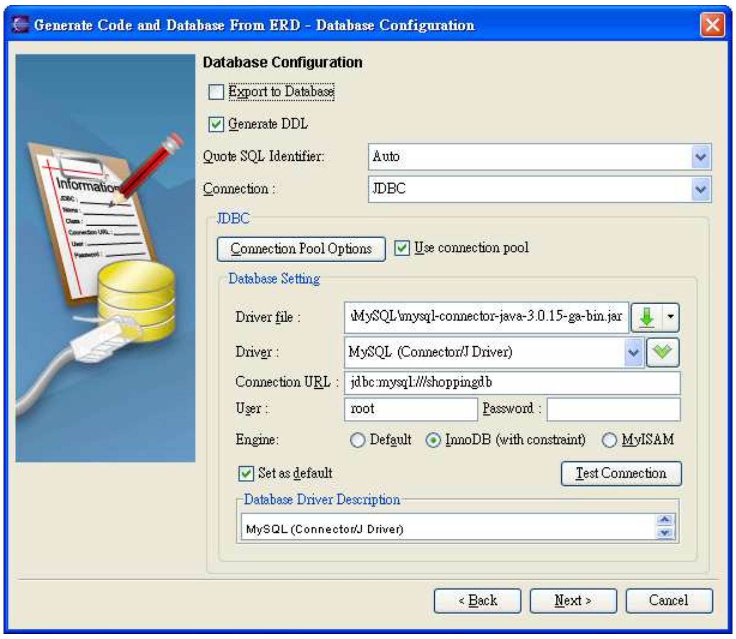
For the **Type**, select the type of Class detail, either Class, Attribute or Role Name (PK) that you want to apply code style.
For the **Prefix/Suffix**, select either Prefix or Suffix to be added or removed.
For the **Add/Remove** option, select the option for the action of code style to be applied.
For the **Textbox**, enter the word for either prefix or suffix.
For the **Scope**, select the scope of the code style to be applied to, either All or Selected.

Table shows the result of applying Code Style.

Code Style	Before Applying	After Applying
Add Prefix (E.g. pre_)	Item	pre_Item
Remove Prefix (E.g. pre_)	pre_Item	Item
Add Suffix (E.g. _suf)	Item	Item_suf
Remove (E.g. _suf)	Item_suf	Item

Table 13.3

2. Database Configuration
   - For Java Language selected


Generate Code and Database From ERD - Database Configuration

Database Configuration

☐ Export to Database

☑ Generate DDL

Quote SQL Identifier: Auto

Connection : JDBC

JDBC

Connection Pool Options ☑ Use connection pool

Database Setting

Driver file : \MySQL\mysql-connector-java-3.0.15-ga-bin.jar

Driver : MySQL (Connector/J Driver)

Connection URL : jdbc:mysql:///shoppingdb

User : root Password :

Engine: ○ Default ◉ InnoDB (with constraint) ○ MyISAM

☑ Set as default Test Connection

Database Driver Description

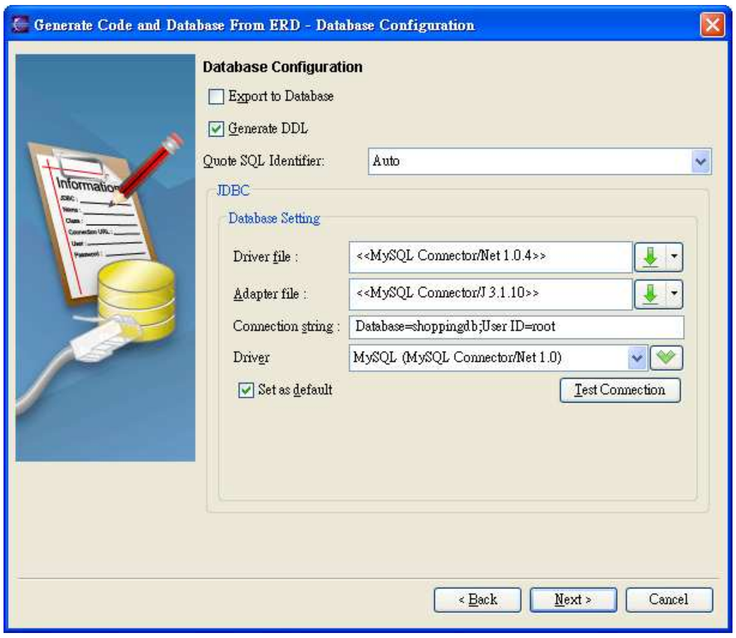
MySQL (Connector/J Driver)

< Back Next > Cancel

Figure 13.37 - Database configuration for Java

You are asked to define the database configuration. Refer to the descriptions in the Getting Started with Object-Relational Mapping chapter for information on how to configure the database in the Database Configuration for Java Project section.

   - For C# Language selected


Generate Code and Database From ERD - Database Configuration

Database Configuration

☐ Export to Database

☑ Generate DDL

Quote SQL Identifier: Auto

JDBC

Database Setting

Driver file : <<MySQL Connector/Net 1.0.4>>

Adapter file : <<MySQL Connector/J 3.1.10>>

Connection string : Database=shoppingdb;User ID=root

Driver MySQL (MySQL Connector/Net 1.0)

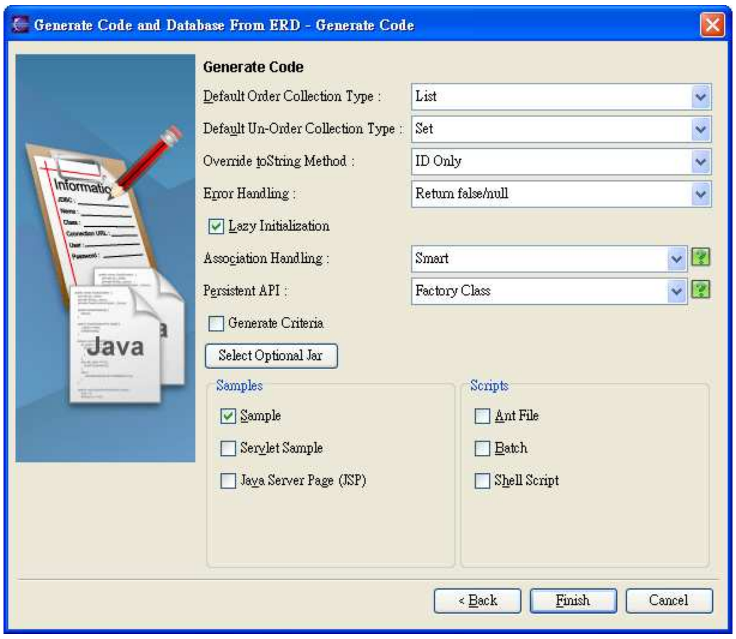
☑ Set as default Test Connection

< Back Next > Cancel

Figure 13.38 - Database configuration for C#

You are asked to define the database configuration. Refer to the descriptions in the Getting Started with Object-Relational Mapping chapter for information on how to configure the database in the Database Configuration for .Net Project section.

3. Generate Code
   - For Java Language selected



Generate Code

Default Order Collection Type : List
Default Un-Order Collection Type : Set
Override toString Method : ID Only
Error Handling : Return false/null
[✓] Lazy Initialization
Association Handling : Smart
Persistent API : Factory Class
[ ] Generate Criteria

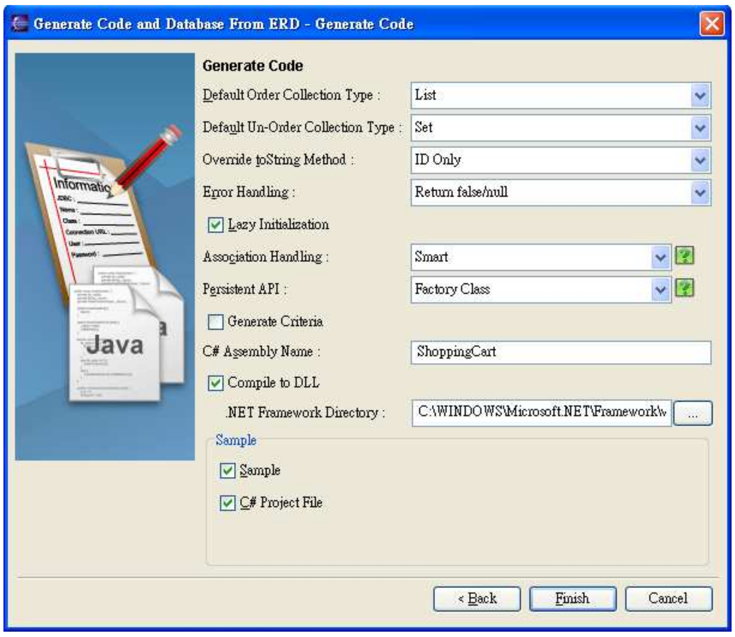
Samples
[✓] Sample
[ ] Servlet Sample
[ ] Java Server Page (JSP)

Scripts
[ ] Ant File
[ ] Batch
[ ] Shell Script

Figure 13.39 - Generate code options for Java

You are asked to specify the code details. For information, refer to the description of Configuring Code Generation Setting for Java section in the Implementation chapter.

   - For C# Language selected



Generate Code

Default Order Collection Type : List
Default Un-Order Collection Type : Set
Override toString Method : ID Only
Error Handling : Return false/null
[✓] Lazy Initialization
Association Handling : Smart
Persistent API : Factory Class
[ ] Generate Criteria
C# Assembly Name : ShoppingCart
[✓] Compile to DLL
.NET Framework Directory : C:\WINDOWS\Microsoft.NET\Framework\v

Sample
[✓] Sample
[✓] C# Project File

Figure 13.40 - Generate code options for C#

You are asked to specify the code details. For information, refer to the description of Configuring Code Generation Setting for C# section in the Implementation chapter.

4. Click **Finish**, the **Generate ORM Code/Database** dialog box appears showing the progress of code generation. Click **Close** when the generation is complete.

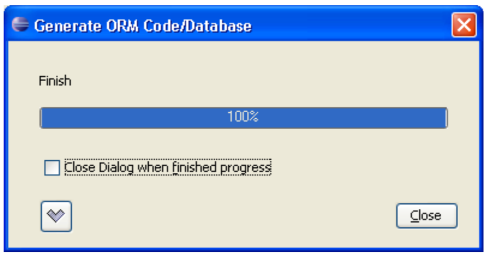

Generate ORM Code/Database
Finish
100%
Close Dialog when finished progress
Close

Figure 13.41 - Generate ORM Code/Database dialog

A class diagram will be generated automatically and added to your project. The generated persistent code and required resources will be generated to the specified output path and the generated database will be set up to the specified database configuration.

Generating Code and Database from Class Diagram

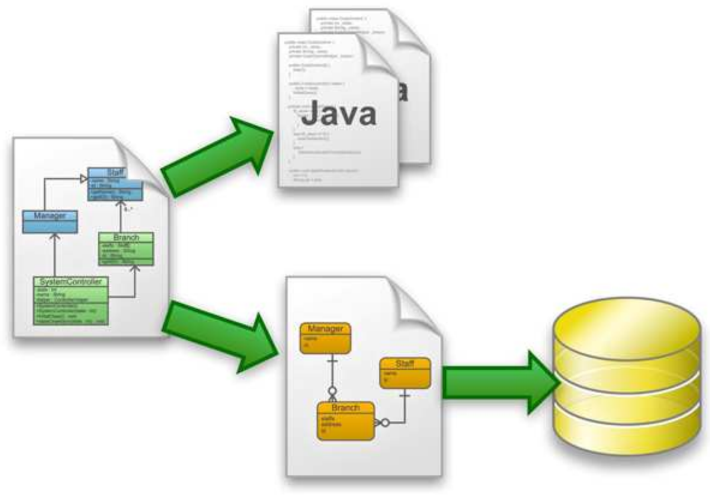


Figure 13.42 -Generate Code and Database from Class Diagram

Upon selecting the option for Generate Code from Class Diagram, the wizard helps you generate persistent code and database from class diagram with respect to the requirements you specified throughout the wizard.

Take the following class diagram as an example:

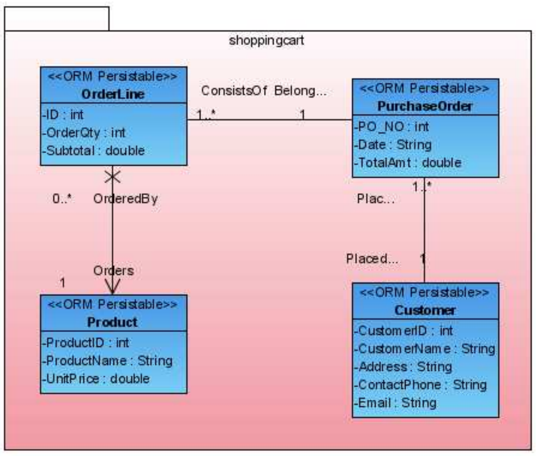


Figure 13.43 - Class Diagram

Follow the steps of the Generate Code and Database from Class Diagram wizard:

1. Selecting Classes

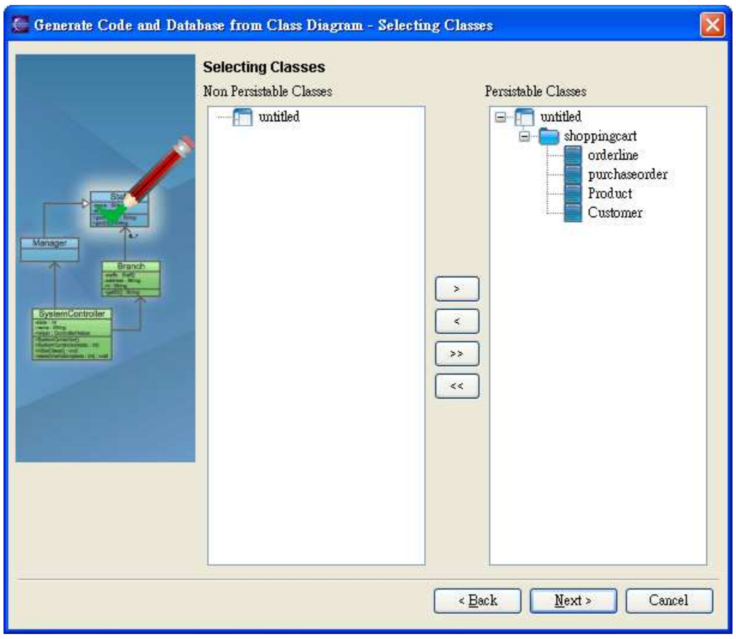


Figure 13.44 - Select the ORM Persistable Classes

You are asked to select the classes on the class diagram which you want to generate persistent class to manipulate persistent data. By default, all the classes stereotyped as ORM-Persistable on the class diagram are selected for the generation of code and database shown in the list of Persistable Classes. You can deselect the persistable classes by using the list of buttons between the list of Non Persistable Classes and Persistable Classes.

- > Add Selected

  Add the selected class from Non Persistable Classes to Persistable Classes.

- < Remove Selected

  Remove the selected class from Persistable Classes to Non Persistable Classes.

- >> Add All

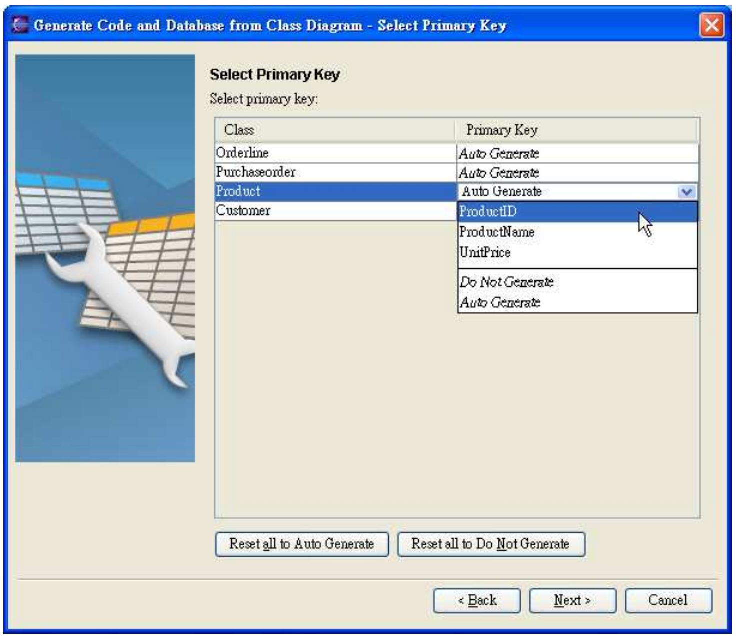
  Add all classes from Non Persistable Classes to Persistable Classes.

- << Remove All

  Remove all classes from Persistable Classes to Non Persistable Classes.

Note: For the classes shown in the list of Persistable Classes, they will be stereotyped as ORM-Persistable on the class diagram after the wizard is finished. Meanwhile, for the classes shown in the list of Non Persistable Classes, they will not be stereotyped on the class diagram after the wizard is finished.

2. Select Primary Key



Generate Code and Database from Class Diagram - Select Primary Key

Select Primary Key

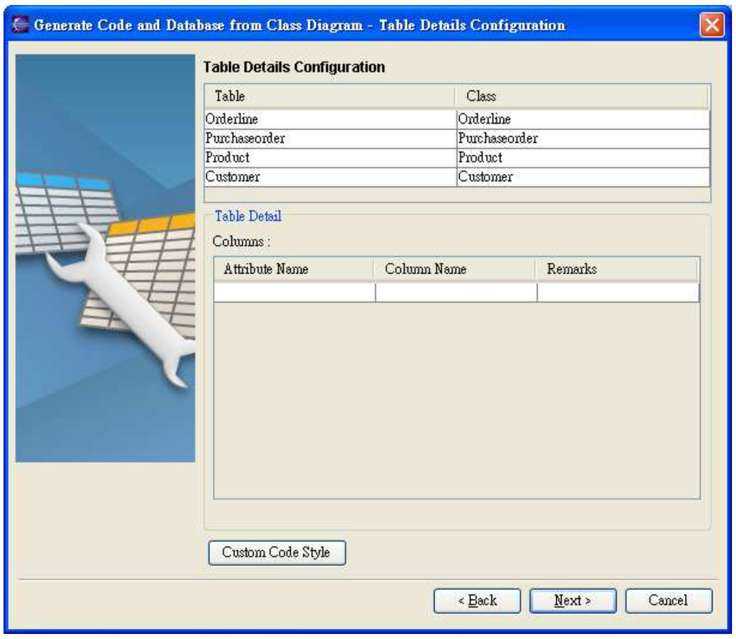
Select primary key:

Class	Primary Key
Orderline	*Auto Generate*
Purchaseorder	*Auto Generate*
Product	Auto Generate
Customer	

Figure 13.45 - Select the primary key

You are asked to select the primary key for each class being mapped to data model and relational database. You can either select an attribute as the primary key or let SDE generate the primary key automatically by using the drop-down menu. For more information, refer to the description of Mapping Primary key in the section of Mapping an Object Model to a Data Model in the Object Model chapter.

3. Table Details Configuration



Generate Code and Database from Class Diagram - Table Details Configuration

Table Details Configuration

Table	Class
Orderline	Orderline
Purchaseorder	Purchaseorder
Product	Product
Customer	Customer

Table Detail

Columns :

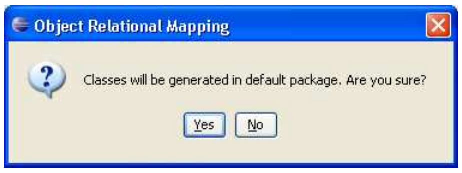
Attribute Name	Column Name	Remarks

Figure 13.46 - Table Details Configuration

You are asked to define the Table Details for generating database and code. The database and persistent classes will be generated based on the information defined here. You can edit the table details by double-clicking the field.

- **Package**

  Enter the package name. A package will be created to store the generated persistent code. If the package name was not defined, you will be prompted by a dialog box warning you the classes will be generated in default package.



Object Relational Mapping

Classes will be generated in default package. Are you sure?

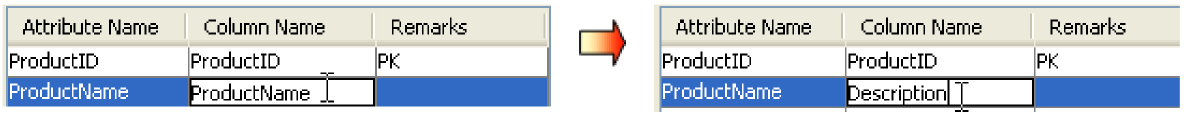
Figure 13.47 - Confirm generate classes in default package

- **Table**

  You can edit the table name which will be used as the name of the generated database table.

- **Columns**

  You can edit the column name represented by the class.



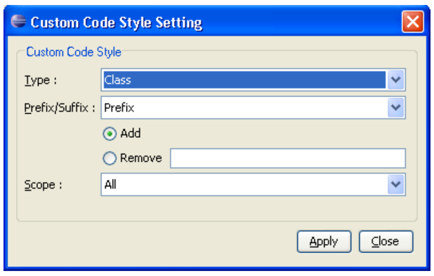
Attribute Name	Column Name	Remarks
ProductID	ProductID	PK
ProductName	ProductName	

Attribute Name	Column Name	Remarks
ProductID	ProductID	PK
ProductName	Description	

Figure 13.48 - Mapping columns

- **Custom Code Style**

  Click button, Custom Code Style Setting dialog box will be displayed. You can modify the prefix or suffix of the Table, Column, Primary Key and Foreign Key.



Custom Code Style Setting

Custom Code Style

Type : Class

Prefix/Suffix : Prefix

(●) Add

( ) Remove

Scope : All

Figure 13.49 - Custom Code Style Setting

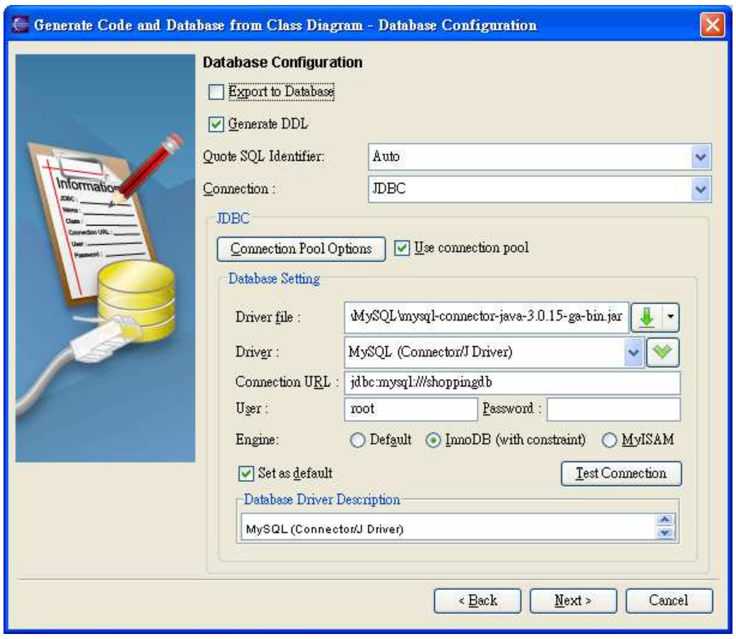
For the **Type**, select the type of Table detail, either Table, Column, Primary Key or Foreign Key that you want to apply code style.
For the **Prefix/Suffix**, select either Prefix or Suffix to be added or removed.
For the **Add/Remove** option, select the option for the action of code style to be applied.
For the **Textbox**, enter the word for either prefix or suffix.
For the **Scope**, select the scope of the code style to be applied to, either All or Selected.

Table shows the result of applying Code Style.

Code Style	Before Applying	After Applying
Add Prefix (E.g. pre_)	Item	pre_Item
Remove Prefix (E.g. pre_)	pre_Item	Item
Add Suffix (E.g. _suf)	Item	Item_suf
Remove (E.g. _suf)	Item_suf	Item

Table 13.4

4. Database Configuration
   - For Java Language selected



Generate Code and Database from Class Diagram - Database Configuration

Database Configuration

☐ Export to Database

☑ Generate DDL

Quote SQL Identifier: Auto

Connection : JDBC

JDBC

☑ Use connection pool

Database Setting

Driver file : \MySQL\mysql-connector-java-3.0.15-ga-bin.jar

Driver : MySQL (Connector/J Driver)

Connection URL : jdbc:mysql:///shoppingdb

User : root Password :

Engine: Default [✓ InnoDB (with constraint)] MyISAM

☑ Set as default

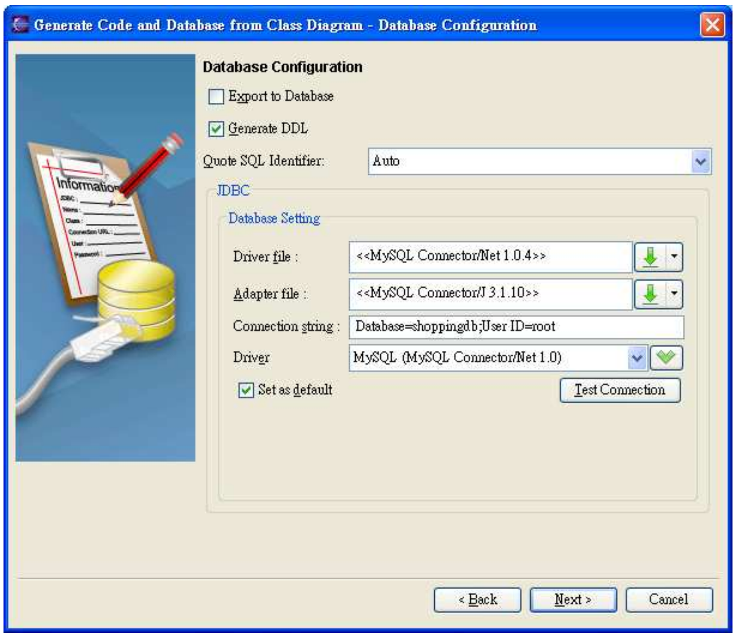
Database Driver Description

MySQL (Connector/J Driver)

Figure 13.50 - Database Configuration for Java

You are asked to define the database configuration. Refer to the descriptions in the Getting Started with Object-Relational Mapping chapter for information on how to configure the database in the Database Configuration for Java Project section

   - For C# Language selected



Generate Code and Database from Class Diagram - Database Configuration

Database Configuration

☐ Export to Database

☑ Generate DDL

Quote SQL Identifier: Auto

JDBC

Database Setting

Driver file : <<MySQL Connector/Net 1.0.4>>

Adapter file : <<MySQL Connector/J 3.1.10>>

Connection string : Database=shoppingdb;User ID=root

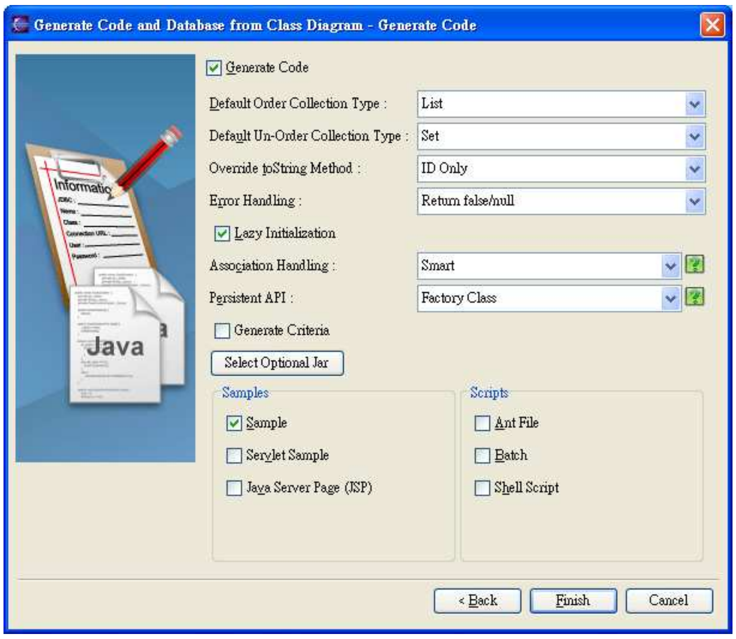
Driver MySQL (MySQL Connector/Net 1.0)

☑ Set as default

Figure 13.51 - Database configuration for C#

You are asked to define the database configuration. Refer to the descriptions in the Getting Started with Object-Relational Mapping chapter for information on how to configure the database in the Database Configuration for .Net Project section.

5. Generate Code
   - For Java Language selected



Generate Code and Database from Class Diagram - Generate Code

☑ Generate Code

Default Order Collection Type : List

Default Un-Order Collection Type : Set

Override toString Method : ID Only

Error Handling : Return false/null

☑ Lazy Initialization

Association Handling : Smart

Persistent API : Factory Class

☐ Generate Criteria

Select Optional Jar

Samples: ☑ Sample ☐ Servlet Sample ☐ Java Server Page (JSP)

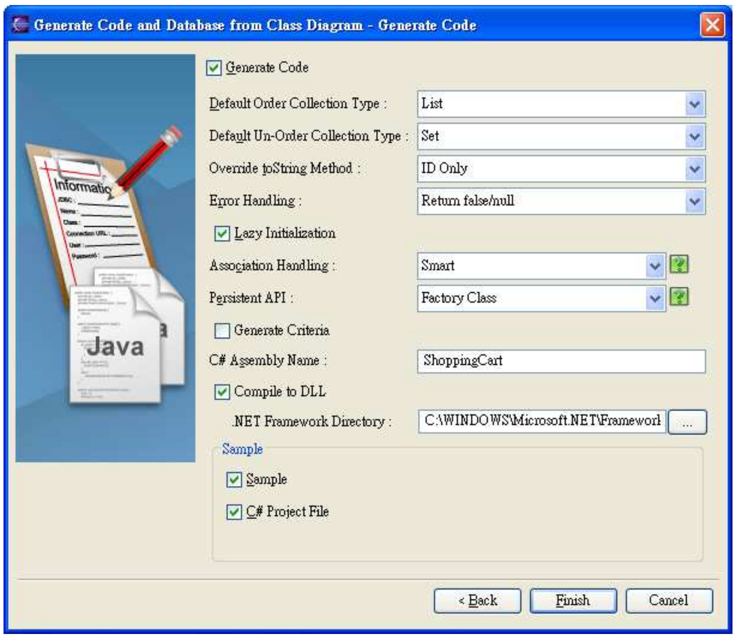
Scripts: ☐ Ant File ☐ Batch ☐ Shell Script

< Back | Finish | Cancel

Figure 13.52 - Generate code options for Java

You are asked to specify the code details. For information, refer to the description of Configuring Code Generation Setting for Java section in the Implementation chapter.

   - For C# Language selected



Generate Code and Database from Class Diagram - Generate Code

☑ Generate Code

Default Order Collection Type : List

Default Un-Order Collection Type : Set

Override toString Method : ID Only

Error Handling : Return false/null

☑ Lazy Initialization

Association Handling : Smart

Persistent API : Factory Class

☐ Generate Criteria

C# Assembly Name : ShoppingCart

☑ Compile to DLL

.NET Framework Directory : C:\WINDOWS\Microsoft.NET\Framework

Sample: ☑ Sample ☑ C# Project File

< Back | Finish | Cancel

Figure 13.53 - Generate code options for C#

6. You are asked to specify the code details. For information, refer to the description of Configuring Code Generation Setting for C# section in the Implementation chapter.

Wizard for Generate Code and Database from Class Diagram option provides an option of generating code to you. By default, the Generate Code option is selected. If you do not want to generate code from class diagram, please deselect the Generate Code option. In this case, only database will be generated while persistent code will not be generated.

7. Click **Finish**, the **Generate ORM Code/Database** dialog box appears showing the progress of code generation. Click **Close** when the generation is complete.

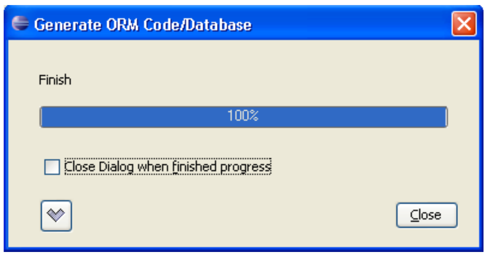

Generate ORM Code/Database
Finish
100%
Close Dialog when finished progress
Close

Figure 13.54 - Generate ORM Code/Database dialog

An entity relationship diagram will be generated automatically and added to your project. The generated persistent code and required resources will be generated to the specified output path and the generated database will be set up to the specified database configuration.

14 Object Model

Chapter 14 - Object Model

Smart Development Environment (SDE) provides you a visual modeling environment for the object model of an application. This chapter shows you how to depict the object models by using a Class Diagram or an EJB diagram and describes how the object model maps to the data model.

In this chapter:

- Introduction
- Creating Object Model
- Mapping Object Model to Data Model
- Creating Enterprise JavaBeans Model
- Mapping Enterprise JavaBeans Model to Data Model

Introduction

An object is a self-contained entity with well-defined characteristics and behaviors while the characteristics and behaviors are represented by attributes and operations respectively. A class is a generic definition for a set of similar objects. Hence, an object is an instance of a class.

An object model provides a static conceptual view of an application. It shows the key components (objects) and their relationships (associations) within the application system. Two types of object models are supported; one is simply called object models referring to the object model for generating the Java model API and .NET model API while the other called Enterprise JavaBeans model which is used to generate the Enterprise JavaBeans. The main difference between these two models is the capability of building a remote and distributed application.

A visual modeling for object models is provided, not only for creating a new object model, but also for transforming from a data model. As object-relational mapping is automated, the database, code and persistent layer can be generated, which in turn streamlines the model-code-deploy software development process.

For visual modeling for Enterprise JavaBeans model, you are allowed to create a new Enterprise JavaBeans model by using EJB Diagram. Synchronizing a data model to Enterprise JavaBeans model is also supported in SDE.

Object Models

A class diagram can be used to describe the objects and classes inside a system and the relationships between them; and thus, a class diagram is also known as an object model. The class diagram identifies the high-level entities of the system. The class diagram truly conforms to a complete UML 2.0.

The following section describes how you can depict an object model using the class diagram. The support for the generation of persistent code based on the object model will be briefly described in ORM-Persistable Class of Implementation chapter.

Creating a Class Diagram

You are provided with two ways to create a Class Diagram:

1. Drawing a Class Diagram
2. Synchronizing from a Data Model to an Object Model

Drawing a Class Diagram

1. You can create a new class diagram in one of the three ways:
   - On the menu, click **File > New Diagram > New Class Diagram**.

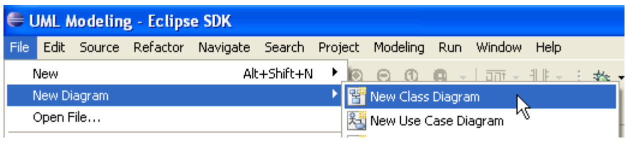

UML Modeling - Eclipse SDK
File Edit Source Refactor Navigate Search Project Modeling Run Window Help
New Alt+Shift+N
New Diagram
Open File...
New Class Diagram
New Use Case Diagram

Figure 14.1 - Create a new Class Diagram

   - On the **Diagram Navigator**, right-click **Class Diagram > New Class Diagram**.

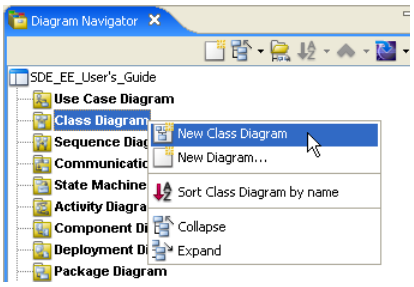


Figure 14.2 - Create Class Diagram by click on Diagram Navigator

   - On the toolbar, click the **New Class Diagram** icon.

A new class diagram pane is displayed.

Creating a new Class element to the Class Diagram

1. On the diagram toolbar, click the **Class** shape icon.

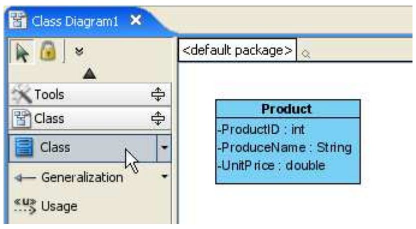


Figure 14.3 - Create a Class

2. Click a location in the diagram pane.

   An icon representing the class element is added to the diagram.

3. Type in a name for the **Class** element.
   - You can edit the name by double-clicking the name or by pressing the **F2** button.

Creating a new ORM-Persistable Class element to the Class Diagram

ORM-Persistable class is capable of manipulating the persistent data with the relational database. Classes added to the class diagram can be stereotyped as ORM Persistable to manipulate the database. For information on how to specify the stereotype of a class, refer to Setting Stereotypes of classes to be ORM Pesistable section.

There is an alternative way to add the ORM-Persistable class easily.

1. On the diagram toolbar, click the drop down button next to Class shape icon, a pop-up menu shows.

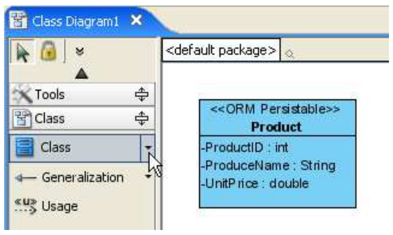


Figure 14.4 - Drop down button next to Class shape icon

2. Select ORM-Persistable Class from the pop-up menu.

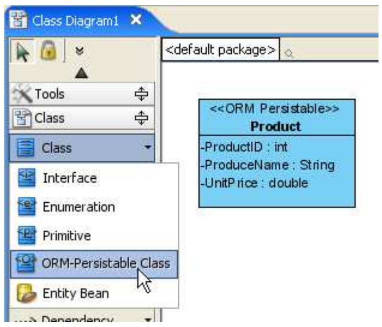


Figure 14.5 - Create ORM-Persistable Class

3. Click a location in the diagram pane.

   A class shape icon which is marked with <<ORM Persistable>> is added to the diagram.

4. Type a name for the ORM-Persistable Class.
   - You can edit the name by double-clicking the name or by pressing the ***F2*** button.

Modifying the Class Specification

A class specification displays the class properties and relationships..

You can display the Class Specification in one of the two ways:

- Click on a class, click the **Open Specification** resource located at the top-right corner of the class.

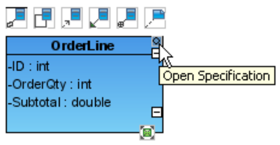
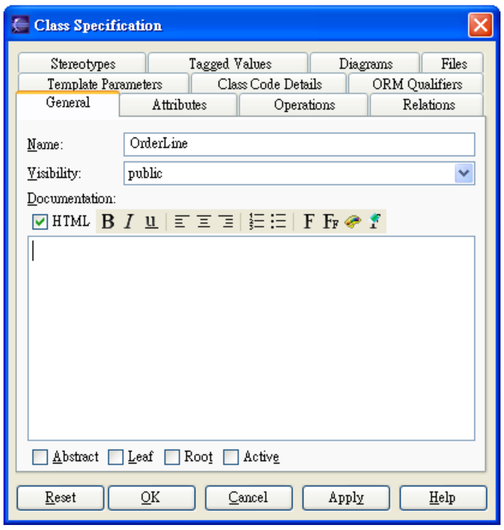


Figure 14.6 - To open specification

- Right-click the class element, click **Open Specification**.

  Class Specification dialog box is displayed, you can modify class properties and relationships.



Class Specification

Stereotypes	Tagged Values	Diagrams	Files
Template Parameters	Class Code Details	ORM Qualifiers	
General	Attributes	Operations	Relations

Name: OrderLine

Visibility: public

Documentation:

[✓] HTML

[ ] Abstract [ ] Leaf [ ] Root [ ] Active

Reset | OK | Cancel | Apply | Help

Figure 14.7 - Class Specification dialog

Adding new Attribute to the Class

An attribute is a property of a class which has a name, a value and also has a type to describe the characteristic of an object.

1. You can add attribute to the class in one of the three ways:
   - Right-click on a class, select **Add** > **Attribute**.

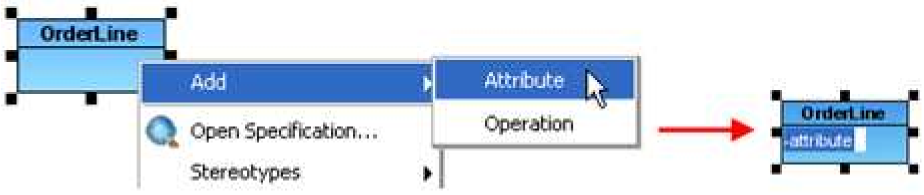
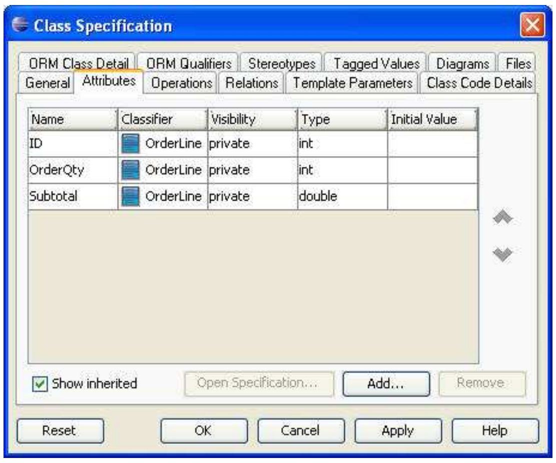


Figure 14.8 - Add an attribute

2. A new attribute is added, type the attribute name and type in the form of "**attribute_name: type**". You can also edit the attribute name by double-clicking the attribute name or by pressing the ***F2*** button.

- Click on a class, press the keyboard shortcut - ***Alt + Shift + A***.
- 1. Right-click the class element, click **Open Specification**.
  2. Click the **Attributes** Tab, then click **Add**.

  Attribute Specification dialog box is displayed, you can modify the attribute name and properties, such as type.



Class Specification

ORM Class Detail | ORM Qualifiers | Stereotypes | Tagged Values | Diagrams | Files
General | Attributes | Operations | Relations | Template Parameters | Class Code Details

Name	Classifier	Visibility	Type	Initial Value
ID	OrderLine	private	int	
OrderQty	OrderLine	private	int	
Subtotal	OrderLine	private	double	

[✓] Show inherited   Open Specification...   Add...   Remove

Reset   OK   Cancel   Apply   Help

Figure 14.9 - Class Specification dialog (Attributes tab)

Adding Association to the Classes

An association refers to the relationship specifying the type of link that exists between objects. It shows how the objects are related to each other.

1. You can add an association to the classes in one of the three ways:
   - Using Resource-Centric Interface
     1. Click on a class, a group of valid editing resources are displayed around the class element.

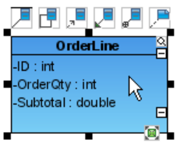


Figure 14.10 - Resource-centric of Class

     2. Mouse over the smart resource of association, drag the resource of "**Association - > Class**" to the associated class.

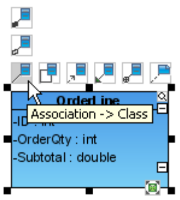


*Figure 14.11 - **Association -> Class** resource-centric*

Smart resource is a kind of resource which groups the resources of similar purpose together and enables the last selected resource (the default resource) of the group to be visible. To see all the resources, mouse over the default resource to expand it.

   - Using Resource-Centric Interface for ORM-Persistable Class element
     1. Click on an ORM-Persistable class, a group of valid editing resources is displayed around the class element.

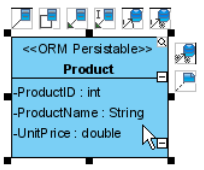


Figure 14.12 - Resource-centric of ORM-Persistable Class

     2. Drag the resource of "**Many-to-Many Association - > Class**" to the associated ORM-Persistable class.

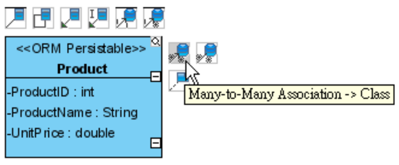


*Figure 14.13 - **Many-to-many -> Class** resource-centric*

- Using Toolbar icon
  1. On the diagram toolbar, click the **Association** icon.

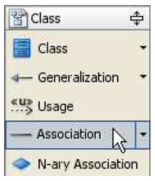


Figure 14.14 - Association Button

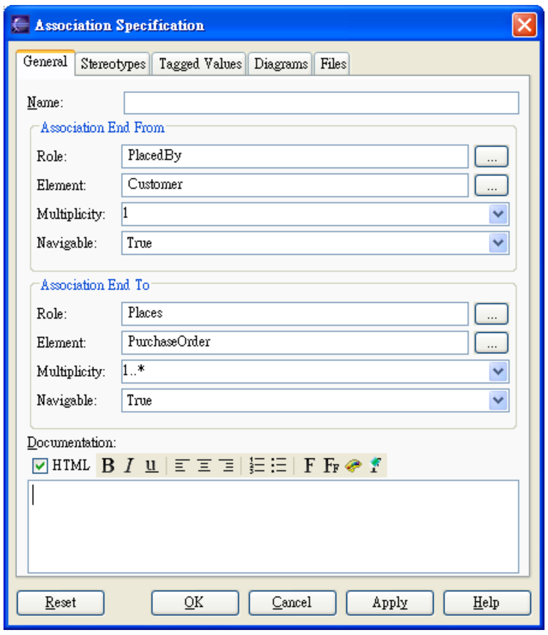
  2. Click on a class, drag to another class.

A line indicating a connection between the two classes is shown.

Editing Association Specification

1. You can edit the association specification in one of the three ways:
- Using Open Specification
  1. Right-click on the connection line, click **Open Specification** from popup menu.

     Association Specification dialog box is displayed, you have to modify the association properties, Roles of classes in Association End From and To, Multiplicity and Navigation etc.



Name:

Association End From

Role: PlacedBy
Element: Customer
Multiplicity: 1
Navigable: True

Association End To

Role: Places
Element: PurchaseOrder
Multiplicity: 1..*
Navigable: True

Documentation:

Figure 14.15 - Association Specification dialog

- Using Pop-up Menu
  1. Right-click on the connection line, the property of the association specification is displayed in the pop-up menu, including **Multiplicity**, **Navigable**, **Aggregation Kind** and **Role Name**.
  2. Select the property that you want to edit, check the desired value.

If you right-click on the connection line towards a class, the pop-up window shows the properties of association specification of the respective class. If you right-click in the middle of the connection line, the pop-up window shows all properties of association specification of both classes.

Role name of the class describes how it acts in the association which will be used in the generation of persistent code. Be sure that you have given the role names to the classes in the association in order to proceed to the generation of code.

- Using Property Pane
  1. On the menu, click **Window** > **Show View** > **Property**.

     The property pane will be displayed.

     For other SDE:

SDE	Method
SDE for JBuilder	Select the **Property** pane on the bottom-left corner.
SDE for NetBeans	From the menu, click **View** > **Property**.
SDE for IntelliJ IDEA	From the menu, click. **View** > **Property**.
SDE for JDeveloper	From the menu, click **View** > **Property**.
SDE for WebLogic Workshop	From the menu, click **View** > **Property**.

Table 14.1

  2. Click on the connection line.

     The properties of the association specification are displayed in the property pane. You can edit the property under the property pane.

As you have completed the class diagram, you can set the stereotypes of the classes to be ORM Persistable while the Class Diagram can be transformed into Entity Relationship Diagram.

Setting Stereotypes of classes be ORM Persistable

Stereotype extends the semantics of the UML metamodel. It classifies the element in what respects it behaves as an instance of metamodel. In order to enable the mapping between object model and relational database, the class has to be stereotyped as ORM persistable.

1. Right-click a class, select **Stereotypes** > **Stereotypes...** .

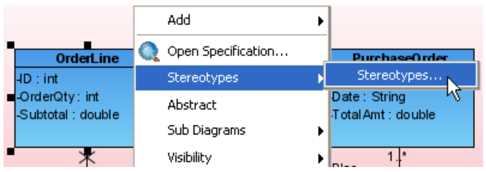


Figure 14.16 - To manage stereotypes

The **Class Specification** dialog box is shown with **Stereotypes** Tab

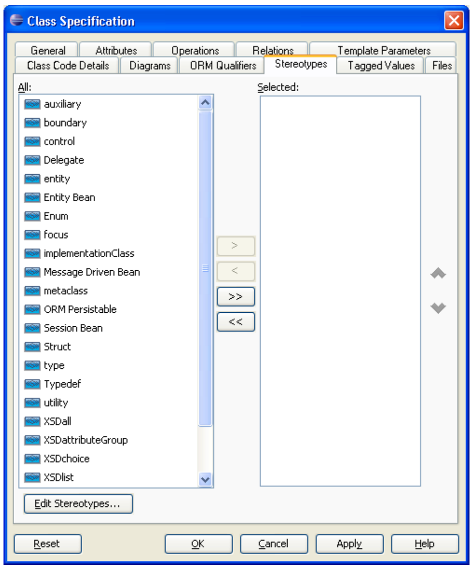


Figure 14.17 - Class Specification (Stereotypes tab)

2. Select **ORM Persistable**, then press > button and **OK**.

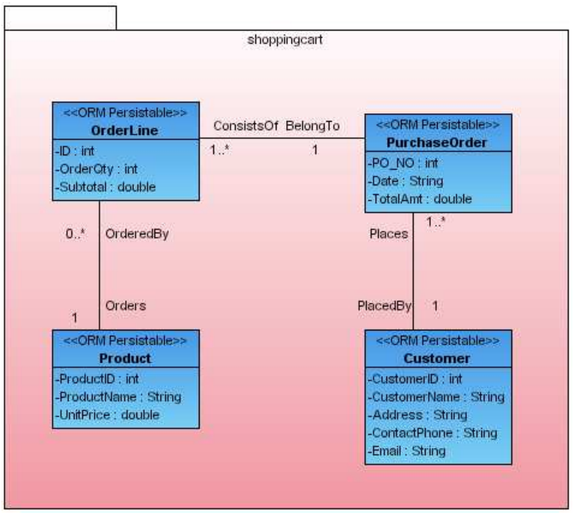


Figure 14.18 - The Classes with Stereotypes

Synchronizing from a Data Model to an Object Model

You can generate a Class Diagram from an ERD by synchronization if there is an ERD.

1. You can synchronize the ERD to Class Diagram in one of the three methods:
   - On the menu, click **Modeling** > **ORM** > **Sync to Class Diagram**.

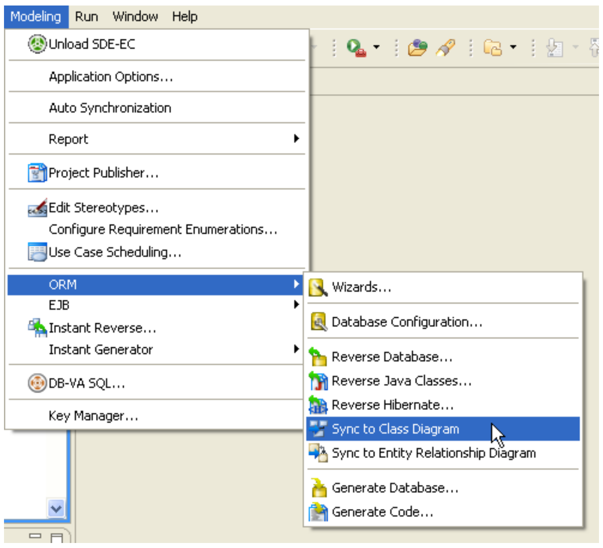


Figure 14.19 - To synchronize ERD to Class Diagram

For other SDE:

SDE	Method
SDE for JBuilder	On the menu, click **Tools** > **Modeling** > **ORM** > **Sync to Class Diagram**.
SDE for NetBeans	On the menu, click **Modeling** > **ORM** > **Sync to Class Diagram..**
SDE for IntelliJ IDEA	On the menu, click **Modeling** > **ORM** > **Sync to Class Diagram**.
SDE for JDeveloper	On the menu, click **Model** > **ORM** > **Sync to Class Diagram**.
SDE for WebLogic Workshop	On the menu, click **Modeling** > **ORM** > **Sync to Class Diagram**.

Table 14.2

- Right-Click on the ERD, select **Synchronize to Class Diagram**.

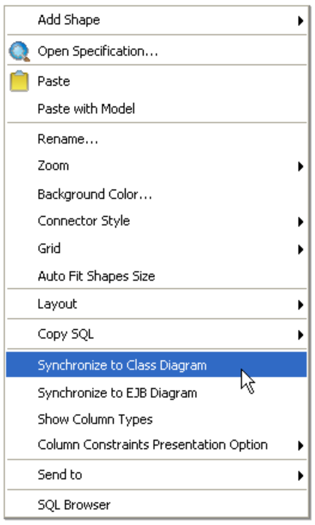


Figure 14.20 - Synchronize to Class Diagram by click on popup menu

- On the ERD, hold down the right-mouse button, move the mouse from right to left to form the gesture. A blue path is shown indicating the gesture.

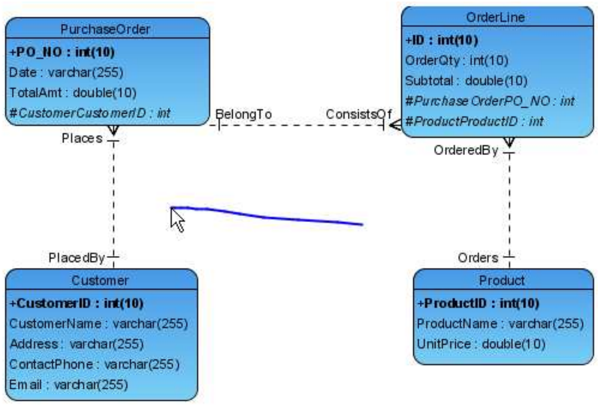


Figure 14.21 - Synchronize to Class Diagram by using gesture

A Class Diagram is generated and can be found under the **Diagram Navigator**.

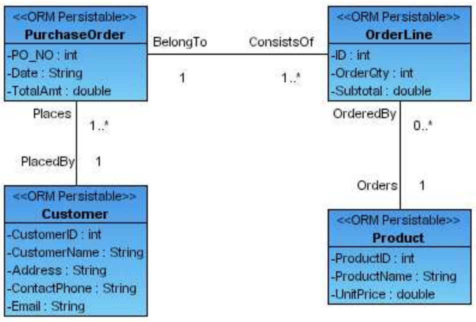


Figure 14.22 - The updated Class Diagram

Defining Package for Classes

There are two ways to define the packages for the classes.

- Enter the package name to the **<default package>** located at the top-left corner of the class diagram by double-clicking the **<default package>.**

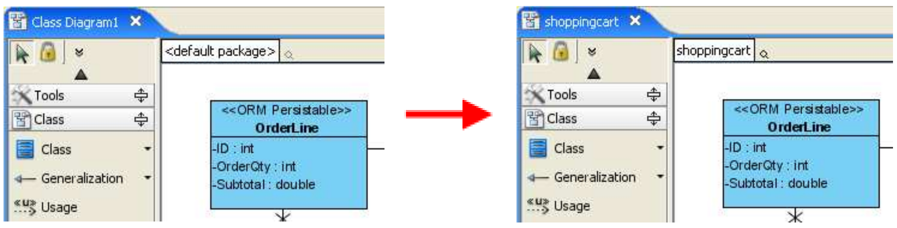


Figure 14.23 - Change the default package of diagram

- Alternative:
  1. On the diagram toolbar, click the **Package** shape icon.

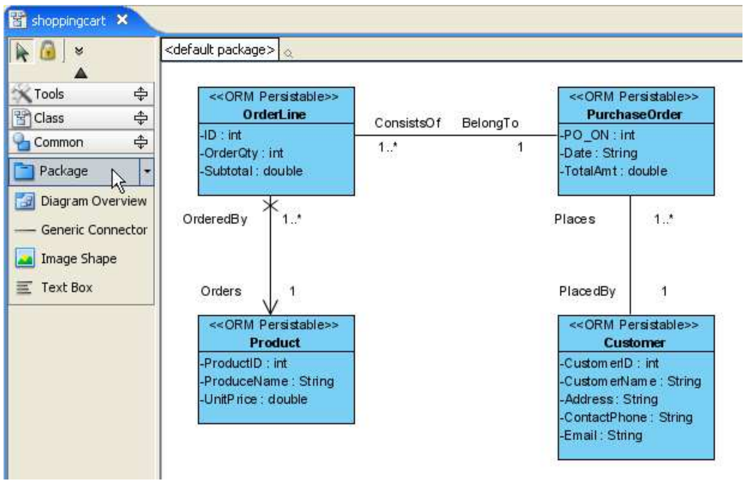


Figure 14.24 - Click on the package icon button

  2. Click a location in the diagram pane to create a package element on the diagram.

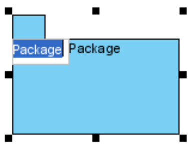


Figure 14.25 - Create and rename the package

  3. Type a name for the **Package** element.

4. Move the desired **Class** elements to the package

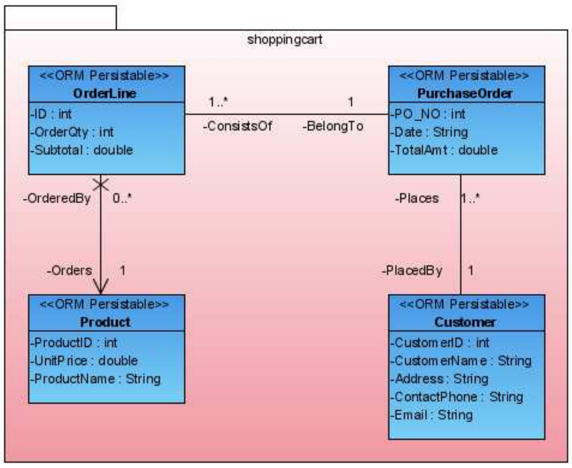


Figure 14.26 - The Classes is moved into the package

After defining the packages to the classes, the classes are inside the package and depicted on the class repository.

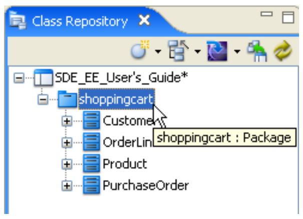


Figure 14.27 - The Class Repository show the classes in a package

Specifying Inheritance Strategy

In a generalization, the subclass inherits all the features of the superclass. Two inheritance strategies - table per class hierarchy and table per subclass are provided for transforming the generalization hierarchy to relational model. By default, table per class hierarchy is used for the generalization.

When transforming generalization into relational model, the generalization is transformed according to the inheritance strategy applied. For more information on the transformation, refer to the description of Mapping Inheritance/Generalization section.

You can specify the inheritance strategy in one of the two ways:

- Specifying from Superclass
  1. Right-click the superclass, select **ORM** > **ORM Class Details...** from the pop-up menu. The **Class Specification** dialog showing the **ORM Class Detail** tab is displayed.

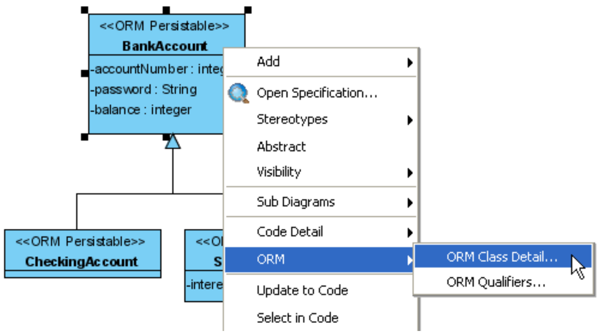
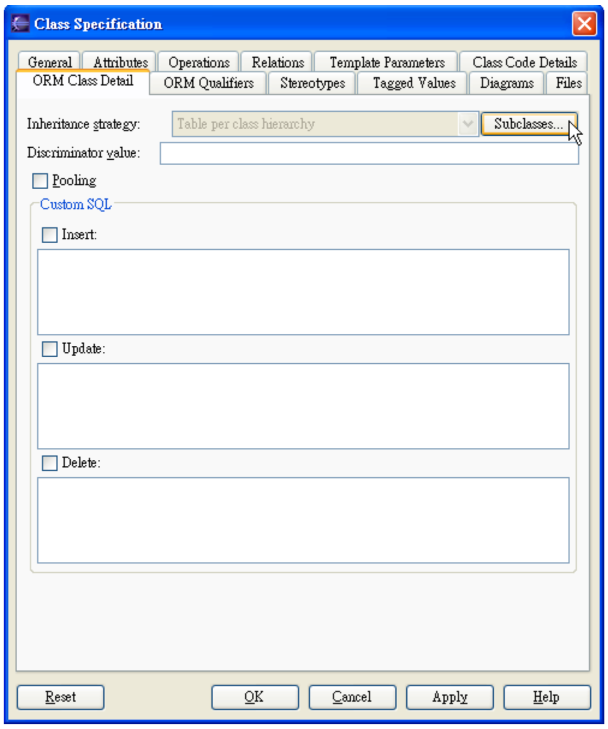


Figure 14.28 - To open the ORM Class Detail

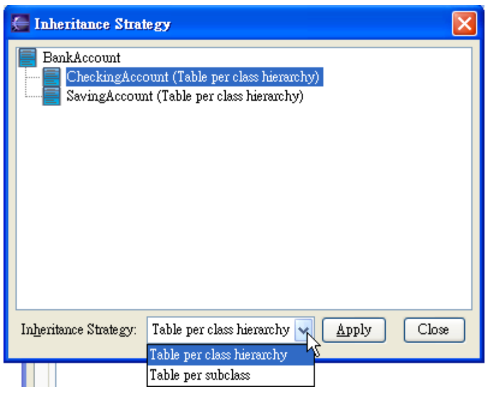
2. Click **Subclasses...** to open the **Inheritance Strategy** dialog box.



Inheritance strategy: Table per class hierarchy
Subclasses...
Discriminator value:
☐ Pooling
☐ Insert:
☐ Update:
☐ Delete:

Figure 14.29 - Class Specification (ORM Class Detail)

3. Select the desired subclass from the generalization tree, select the **Inheritance Strategy** from the drop-down menu, and then click **Apply**.



BankAccount
CheckingAccount (Table per class hierarchy)
SavingAccount (Table per class hierarchy)

Inheritance Strategy: Table per class hierarchy
Table per class hierarchy
Table per subclass

Figure 14.30 - Inheritance Strategy dialog

- Specifying from Subclass
  1. Right-click the subclass, select **ORM > ORM Class Details...** from the pop-up menu. The **Class Specification** dialog box showing the **ORM Class Detail** tab is displayed.

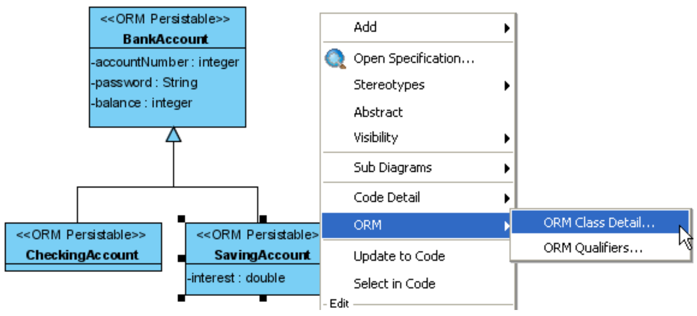


Figure 14.31 - To open the ORM Class Detail of sub-class

2. Select the **Inheritance strategy** from the drop-down menu.

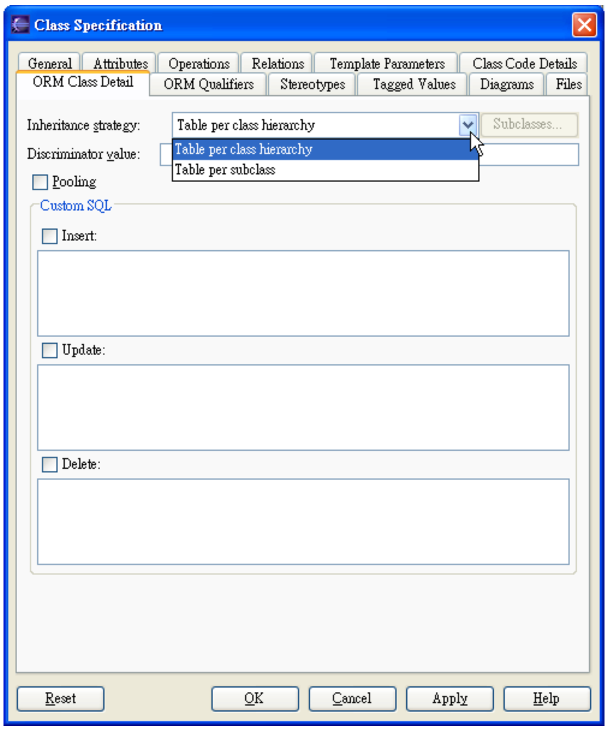


Figure 14.32 - To change the Inherit strategy of sub-class

These two inheritance strategies can be applied to different subclasses within a generalization hierarchy in Java project. Applying two strategies to different subclasses within a generalization in .NET project will result in error when the generation of code and database.

Specifying Collection Type

If one end of an association contains a multiplicity of many, a collection class will be generated for handling the multiple instances. You are allowed to specify the type of collection, including set, bag, list and map.

Set is an unordered collection that does not allow duplication of objects. Bag is an unordered collection that may contain duplicate objects. List is an ordered collection that allows duplication of objects. Map is an ordered collection that maps key to values while each key can map to exactly one value.

For more information on the usage of Collection in Java code, refer to the description of Using Collection with Smart Association Handling and Using Collection with Standard Association Handling in the Manipulating Persistent Data with Java chapter.

For more information on the usage of Collection in .NET source code, refer to the description of Using Collection with Smart Association Handling and Using Collection with Standard Association Handling in the Manipulating Persistent Data with .NET chapter.

1. Right-click on the connection line, click **Open Specification** from popup menu.

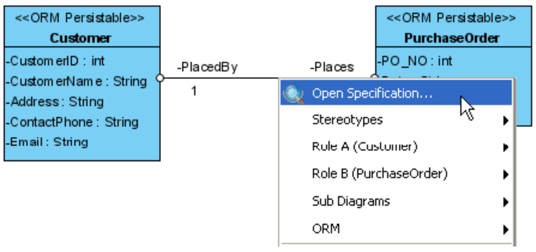


Figure 14.33 - To open the Association Specification

2. Click the **ORM Association Detail** Tab, select the **Collection Type** from the drop-down menu.

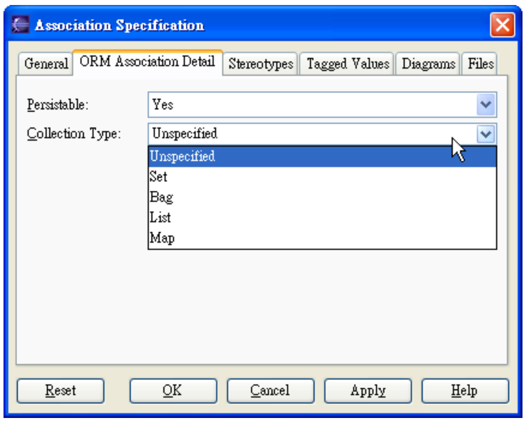


Figure 14.34 - Change the Collection type

Defining ORM Qualifier

ORM Qualifier is used to specify the extra retrieval rules of the generated persistent class querying the database. You are allowed to define the ORM Qualifiers of the classes in the class diagram before the generation of persistent code. For more information on the usage of ORM Qualifier, refer to the description of Using ORM Qualifier in the Manipulating Persistent Data with Java and Manipulating Persistent Data with .NET chapter.

1. Right-click on a class that you want to add extra retrieval rules, click **Open Specification**.

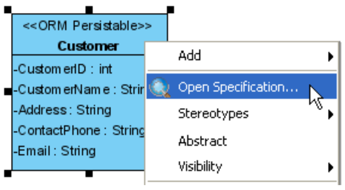


Figure 14.35 -

2. Click the **ORM Qualifiers** Tab, then click **Add**.

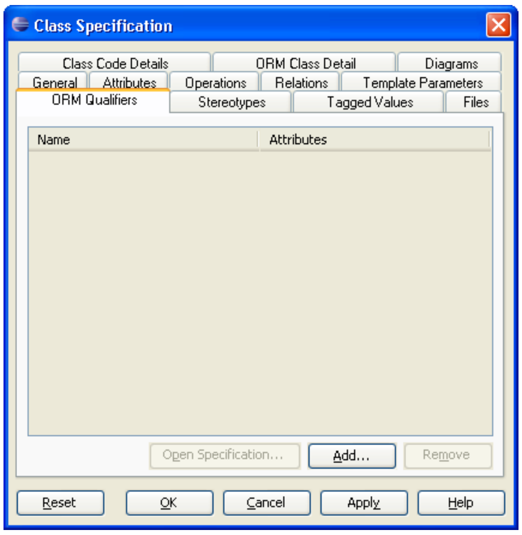
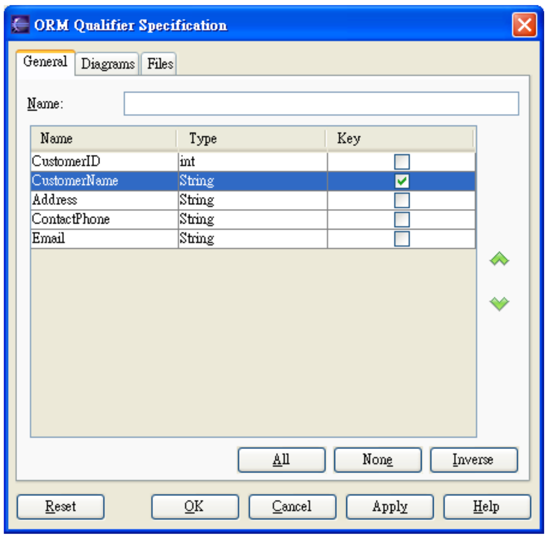


Figure 14.36 - Class Specification (ORM Qualifiers)

ORM Qualifier Specification dialog box is displayed with a list of attributes of the selected class.

3. Enter the name of the ORM Qualifier, place a check mark for the **Key** column of the attribute that will be used in querying the database.


ORM Qualifier Specification

General | Diagrams | Files

Name:

Name	Type	Key
CustomerID	int	☐
CustomerName	String	✓
Address	String	☐
ContactPhone	String	☐
Email	String	☐

All | None | Inverse

Reset | OK | Cancel | Apply | Help

Figure 14.37 - ORM Qualifier Specification dialog

Customizing SQL

Ready-to-use SQL statements will be generated and used by the persistent code to directly manipulate the database. In some cases, you may find the generated SQL statements not appropriate for your needs. You are allowed to override the generated SQL statements, including the Insert, Update and Delete statements whenever you want to.

To customize the generated SQL statements:

1. Right-click on an ORM-Persistable class that you want to customize the SQL statements, select **ORM > ORM Class Details...** from the pop-up menu. The **Class Specification** dialog box showing the **ORM Class Detail** tab is displayed.

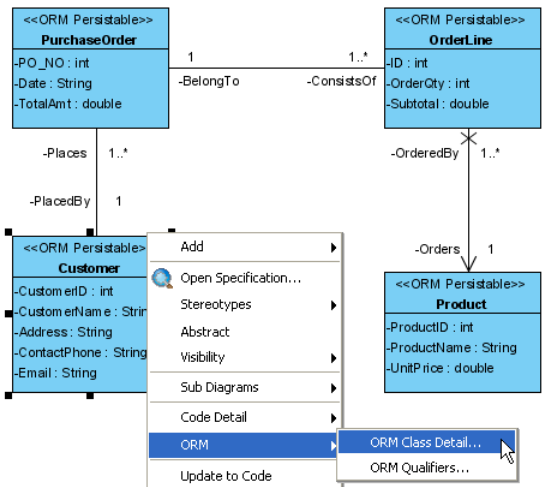


Figure 14.38 - Open the ORM Class detail

2. Select the type of SQL statement that you want to customize.

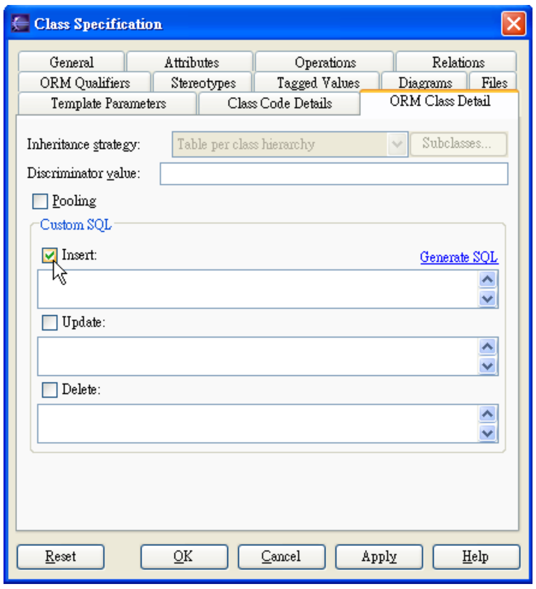


Figure 14.39 - Class Specification (ORM Class Detail)

3. Click **Generate SQL** to generate the ready-to-use SQL statement.

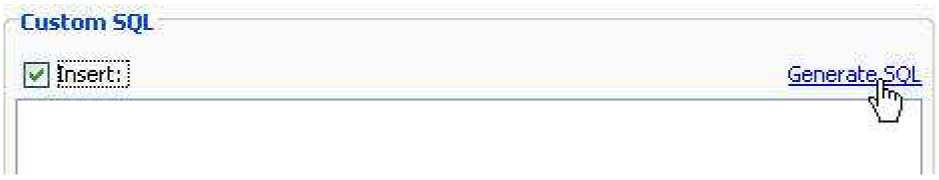
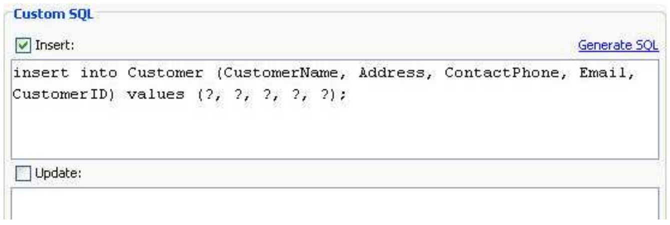


Figure 14.40 - General the default insert SQL statement

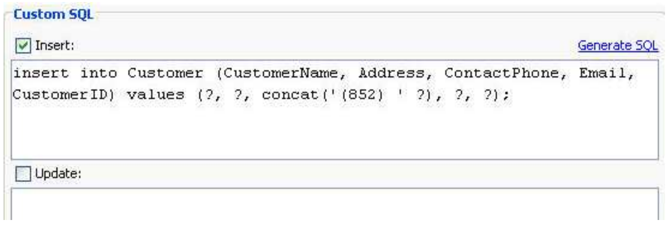
The SQL statement is generated based on the property of class.



```
insert into Customer (CustomerName, Address, ContactPhone, Email,
CustomerID) values (?, ?, ?, ?, ?);
```

Figure 14.41 - The default insert SQL statement

4. Modify the SQL statement to the desired one.



```
insert into Customer (CustomerName, Address, ContactPhone, Email,
CustomerID) values (?, ?, concat('(852) ' ?), ?, ?);
```

Figure 14.42 - Modified the SQL statement

Mapping an Object Model to a Data Model

Object Relational Mapping (ORM) is supported which maps object models to entity relational models and vice versa.

Mapping between objects to relational database preserves not only the data, but also the state, foreign/primary key mapping, difference in data type and business logic. Thus, you are not required to handle those tedious tasks during software development.

Mapping Classes to Entities

Object Model can be mapped to Data Model due to the persistent nature of classes. Persistent classes can act as persistent data storage during the application is running. And hence, all persistent classes can map to entities using a one-to-one mapping.

Example:

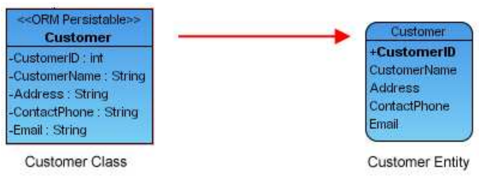


Figure 14.43 - Mapping class to entity

In the above example, the Customer Class is mapped with the Customer Entity as the Customer instance can store the customer information from the Customer Entity.

Mapping Attributes to Columns

Since the persistent classes map to the entities, persistent attributes map to columns accordingly. All non-persistent attributes such as derived values are ignored during the transformation.

Example:

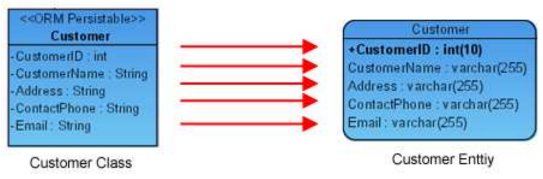


Figure 14.44 - Mapping attribute to column

In the above example, the following table shows the mapping between the attributes of the Customer Class and the columns of the Customer Entity.

Customer Class	Customer Entity
CustomerID	CustomerID
CustomerName	CustomerName
Address	Address
ContactPhone	ContactPhone
Email	Email

Table 14.3

Mapping Data Type

The persistent attribute type automatically maps to an appropriate column data type of the database you desired.

Example:

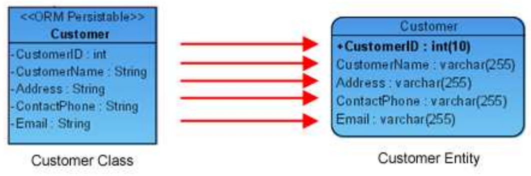


Figure 14.45 - Mapping data type

In the above example, the following table shows the mapping between data types

Customer Class	Customer Entity
int	int (10)
String	varchar(255)

Table 14.4

A table shows the data type mapping between Object model and Data model.

Object Model	Data Model
Boolean	Bit (1)
Byte	Tinyint (3)
Byte[]	Binary(1000)
Blob	Blob
Char	Char(1)
Character	Char(1)
String	varchar(255)
Int	Integer(10)
Integer	Integer(10)
Double	Double(10)
Decimal	Integer
Bigdecimal	Decimal(19)
Float	Float(10)
Long	Bigint(19)
Short	Smallint(5)
Date	Date
Time	Time(7)
Timestamp	Timestamp(7)

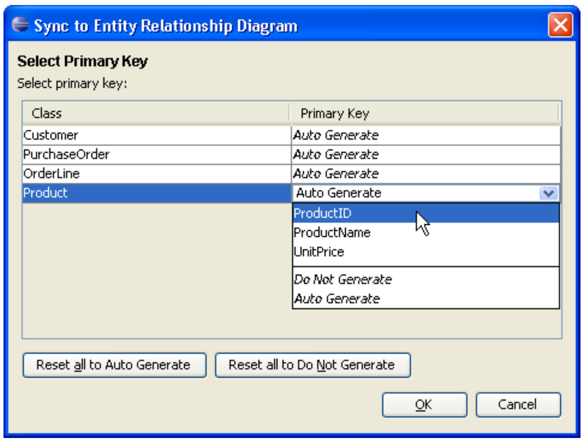
Figure 14.5

Mapping Primary Key

You can map an attribute to a primary key column. When you synchronize the ORM-Persistable Class to the ERD, you will be prompted by a dialog box to select primary key.

- You can select an attribute as the primary key.
- You can let SDE generate the primary key automatically.

Example:



Sync to Entity Relationship Diagram

Select Primary Key

Select primary key:

Class	Primary Key
Customer	*Auto Generate*
PurchaseOrder	*Auto Generate*
OrderLine	*Auto Generate*
Product	Auto Generate

ProductID
ProductName
UnitPrice
Do Not Generate
Auto Generate

Figure 14.46 - Sync to Entity Relationship Diagram dialog

In the above example, when synchronizing the class of Product to entity relationship diagram, the above dialog box is shown to prompt you to select the primary key of the Product class.

Under the drop-down menu, you can select either one of the attributes of the Product class to be the primary key, or assign SDE to generate the primary key automatically, or select "**Do Not Generate**" to leave the generated entity without primary key.

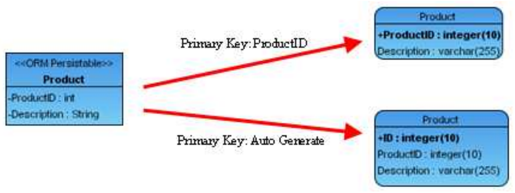


Figure 14.47 - User specify and auto generate primary key

The above diagram shows if you assign ProductID as primary key, the ProductID of the generated entity, Product will become bold; whereas if you select "**Auto Generate**" for the primary key, an additional attribute ID is generated as the primary key of the Product entity.

Mapping Association

Association represents a binary relationship among classes. Each class of an association has a role. A role name is attached at the end of an association line. The role name maps to a phrase of relationship in the data model.

Mapping Aggregation

Aggregation is a stronger form of association. It represents the "has-a" or "part-of" relationship.

Example:

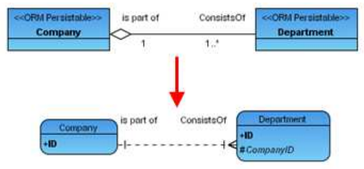


Figure 14.48 - Mapping aggregation

In the above example, it shows that a company consists of one or more department while a department is a part of the company.

You have to give the role names, "ConsistsOf" and "is Part Of" to the classes, Company and Department in the association respectively in order to proceed to the generation of code.

Mapping Composite Aggregation

Composite aggregation implies exclusive ownership of the "part-of" classes by the "whole" class. It means that parts may be created after a composite is created, meanwhile such parts will be explicitly removed before the destruction of the composite.

Example:

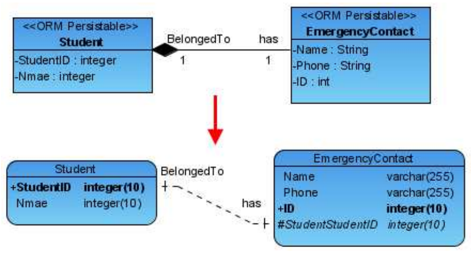


Figure 14.49 - Mapping Composite Aggregation

In the above example, the Primary/Foreign Key Column Mapping is automatically executed. StudentID of the student entity is added to the entity, EmergencyContact as primary and foreign key column.

Mapping Multiplicity

Multiplicity refers to the number of objects associated with a given object. There are six types of multiplicity commonly found in the association. The following table shows the syntax to express the Multiplicity.

Table shows the syntax expressing the Multiplicity

Type of Multiplicity	Description
0	Zero instance
0..1	Zero or one instances
0..*	Zero or more instances
1	Exactly one instance
1..*	One or more instances
*	Unlimited number of instances

Table 14.6

Example:

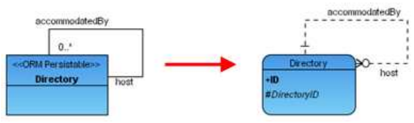
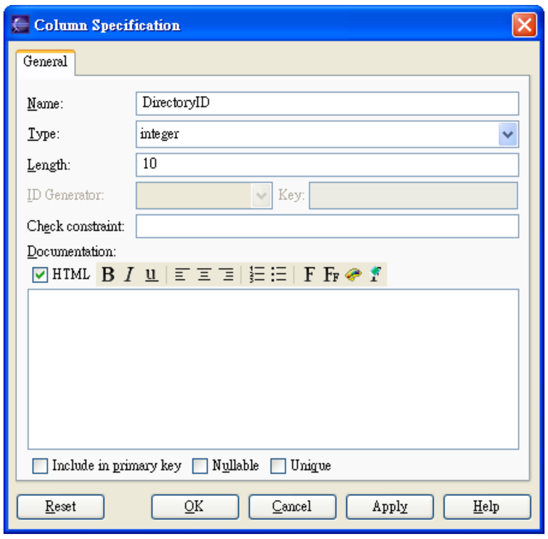


Figure 14.50 - Mapping multiplicity

In the above example, it shows that a parent directory (role: host) contains zero or more subdirectories (role: accommodated by).

When transforming a class with multiplicity of zero, the foreign key of parent entity can be nullable in the child entity. It is illustrated by the DirectoryID.


Column Specification

General

Name: DirectoryID
Type: integer
Length: 10
ID Generator: Key:
Check constraint:
Documentation:
[✓] HTML

[ ] Include in primary key [ ] Nullable [ ] Unique

Figure 14.51 - Column Specification dialog

Table shows the typical mapping between Class Diagram and Entity Relationship Diagram.

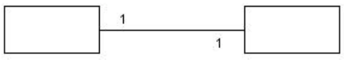
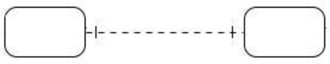
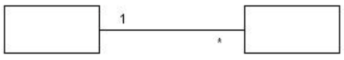
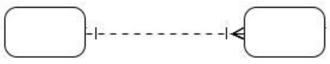
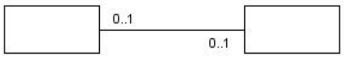
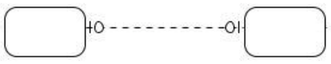
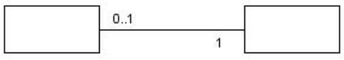
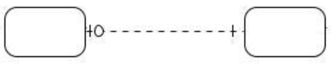
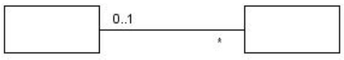
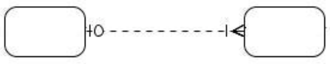
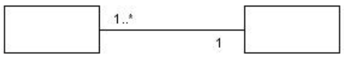
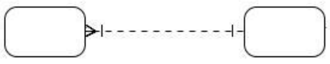
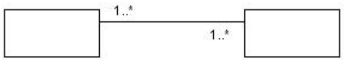
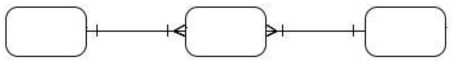
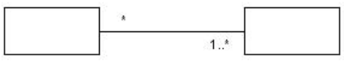
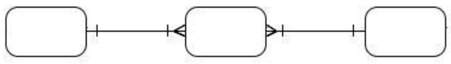
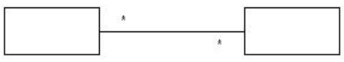
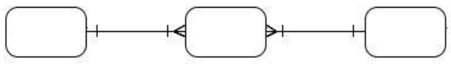
Class Diagram	Entity Relationship Diagram
	
	
	
	
	
	
	
	
	

Table 14.7

Mapping Many-to-Many Association

For a many-to-many association between two classes, a Link Entity will be generated to form two one-to-many relationships in-between two generated entities. The primary keys of the two entities will migrate to the link entity as the primary/foreign keys.

Example:

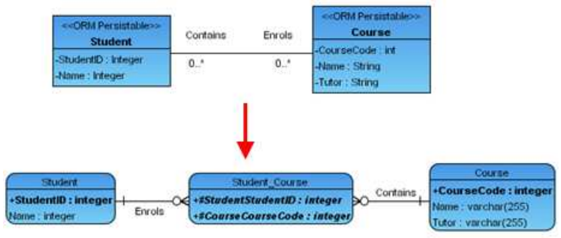


Figure 14.52 - Mapping Many-to-many association

In the above example, the link entity, Student_Course is generated between entities of Student and Course when transforming the many-to-many association.

Mapping Inheritance/Generalization

Generalization distributes the commonalities from the superclass among a group of similar subclasses. The subclass inherits all the superclass's attributes and it may contain specific attributes.

Two strategies are provided for transforming the generalization hierarchy to relational model. The two strategies for transformation are table per class hierarchy and table per subclass. For information on how to specify the inheritance strategy to subclasses, refer to the description of *Specifying Inheritance Strategy* section.

Using Table per Class Hierarchy Strategy

Transforming generalization hierarchy to relational model with the table per class hierarchy strategy, all the classes within the hierarchy will be combined into one single entity containing all the attributes, and a discriminator column will also be generated to the entity. The discriminator is a unique value identifying the entity which hierarchy it belongs to.

By using the table per class hierarchy strategy, the time used for reading and writing objects can be saved. However, more memory is used for storing data. It is useful if the class will be loaded frequently.

Example:

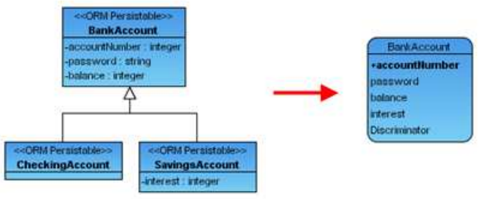


Figure 14.53 - Using table per Class Hierarchy Strategy

In the above example, it shows how the generalization with CheckingAccount, SavingsAccount and their superclass, BankAccount is transformed by applying the table per class hierarchy strategy.

Using Table per Subclass Strategy

When a generalization hierarchy using the table per subclass strategy is transformed to relational model, each subclass will be transformed to an entity with a one-to-one identifying relationship with the entity of the superclass.

By using the table per subclass strategy, it can save memory for storing data. However, it takes time for reading and writing an object among several tables which slows down the speed for accessing the database. It is useful if the class, which contains a large amount of data, is not used frequently.

Example:

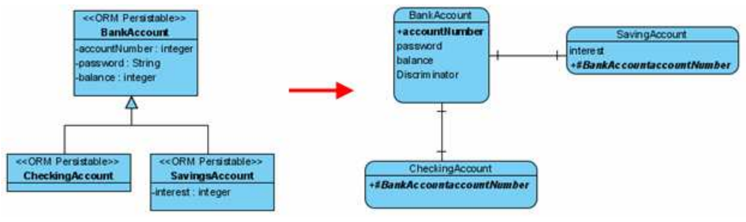


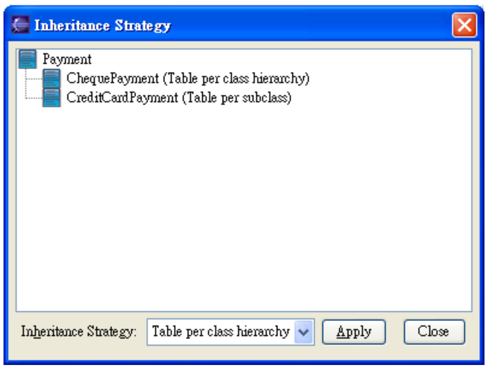
Figure 14.54 - Using table per subclass strategy

In the above example, it shows how the generalization which applies the table per subclass hierarchy strategy is transformed to the CheckingAccount and SavingsAccount.

Using Mixed Strategies

The two inheritance strategies are allowed to be applied to different subclasses within a generalization hierarchy in Java project. By applying different strategies to different subclasses within a generalization hierarchy, the generalization hierarchy will be transformed with respect to the specified inheritance strategies.

Example:


Inheritance Strategy

Payment
- ChequePayment (Table per class hierarchy)
- CreditCardPayment (Table per subclass)

Inheritance Strategy: Table per class hierarchy

Figure 14.55 - Inheritance Strategy dialog

Applying the above inheritance strategy to the generalization with the ChequePayment and CreditCardPayment and their superclass, Payment, two entities are resulted after transformation as shown below.

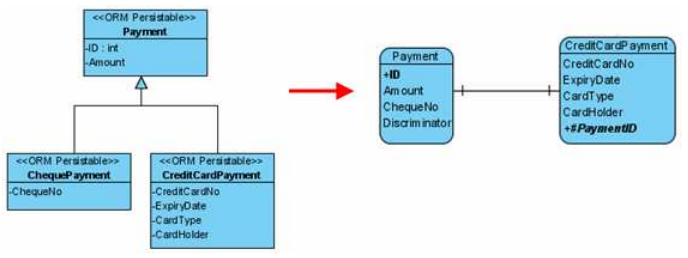


Figure 14.56 - Mapping using mixed strategies

In the above example, it shows that applying table per hierarchy inheritance strategy will result in combining the attributes of the superclass and subclass into one single entity while table per subclass will result in forming an entity of subclass and a one-to-one identifying relationship between entities of superclass and subclass.

Note

Applying two inheritance strategies to different subclasses within a generalization hierarchy is only available in Java project. If mixed strategies are applied to the generalization hierarchy in .NET project, it will result in error when the generation of code and database.

Mapping Collection of Objects to Array Table

For a persistent class acting as persistent data storage, it may consist of a persistent data containing a collection of objects. The Array Table is promoted to be used to allow users retrieve objects in the form of primitive array.

When transforming a class with an attribute of array type modifier, this attribute will be converted to an Array Table automatically. The generated entity and the array table form a one-to-many relationship.

Example:

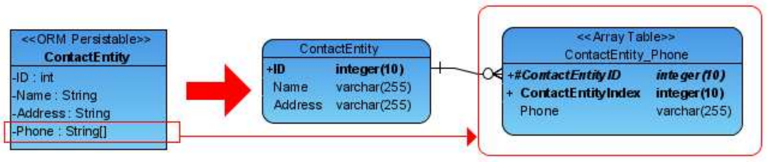


Figure 14.57- Mapping Collection of Object to Array Table

In the above example, the phonebook has a contact entry for each contact person. Each contact person may have more than one phone numbers. In order to ease the retrieval of a collection of phone objects, the phone attribute is converted into a ContactEntry_Phone array table.

Mapping Object Model Terminology

Table shows the shift from Object model to data model terminology.

Object Model Term	Data Model Term
Class	Entity
Object	Instance of an entity
Association	Relationship
Generalization	Supertype/subtype
Attribute	Column
Role	Phrase
Multiplicity	Cardinality

Table 14.8

Showing Mapping between Object and Data Models by ORM Diagram

In order to identify the mapping between the object and data models clearly, an ORM diagram is provided to show the mappings between the ORM-Persistable class and its corresponding entity.

As you are allowed to name the ORM-Persistable class and its corresponding entity differently, and also the attributes and columns as well, you may find it difficult to identify the mappings not only between the ORM-Persistable classes and the corresponding entities, but also the attributes and columns. Taking the advantage of ORM diagram, the mappings between ORM-Persistable classes and entities and between attributes and columns can be clearly identified.

There are two ways to create the ORM diagram for showing the mapping between the object and data models:

1. Creating an ORM diagram from the existing class diagram and/or ERD.
2. Drawing an ORM Diagram

Creating an ORM Diagram from Existing Diagrams

The following example shows you how to show the mapping between the existing object and data models by the ORM diagram.

Let us assume the following class diagram has been created and synchronized to the entity relationship diagram (ERD).

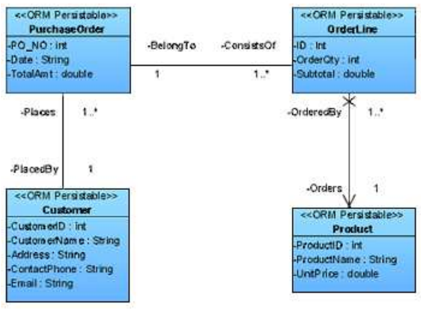
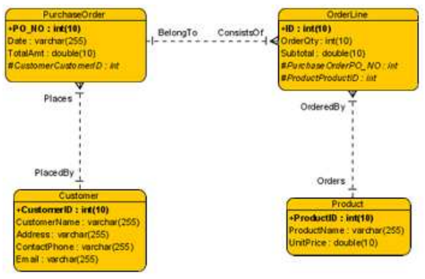
Class Diagram	Synchronized ERD
 <<ORM Persistable>> PurchaseOrder -PO_NO : int -Date : String -TotalAmt : double -BelongTo 1 -ConsistsOf 1..* <<ORM Persistable>> OrderLine -ID : int -OrderQty : int -Subtotal : double -Places 1..* -PlacedBy 1 -OrderedBy 1..* -Orders 1 <<ORM Persistable>> Customer -CustomerID : int -CustomerName : String -Address : String -ContactPhone : String -Email : String <<ORM Persistable>> Product -ProductID : int -ProductName : String -UnitPrice : double	 PurchaseOrder +PO_NO : int(10) Date : varchar(255) TotalAmt : double(10) #CustomerCustomerID : int BelongTo ConsistsOf OrderLine +ID : int(10) OrderQty : int(10) Subtotal : double(10) #PurchaseOrderPO_NO : int #ProductProductID : int Places PlacedBy OrderedBy Orders Customer +CustomerID : int(10) CustomerName : varchar(255) Address : varchar(255) ContactPhone : varchar(255) Email : varchar(255) Product +ProductID : int(10) ProductName : varchar(255) UnitPrice : double(10)

Table 14.9

You can create an ORM diagram by either the object model, data model, or both. The following steps show you how to create the ORM diagram by using the object model.

1. Right-click on the class diagram, select **Send to > ORM Diagram > New ORM Diagram** from the pop-up menu.

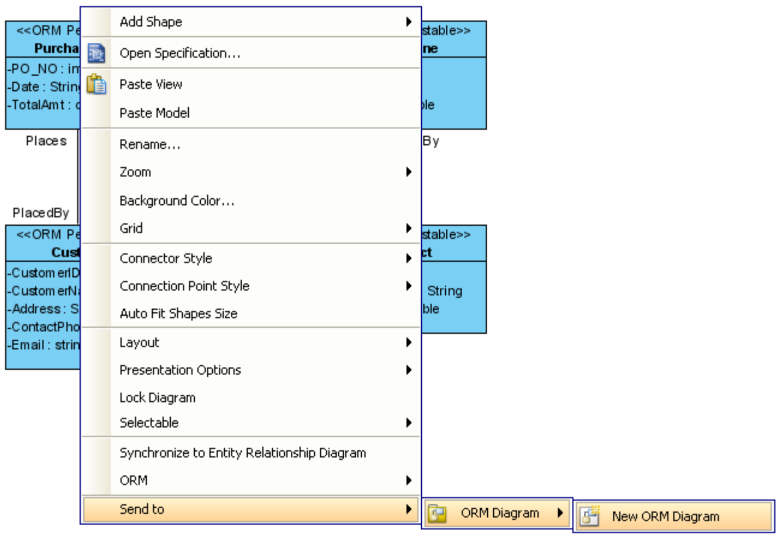
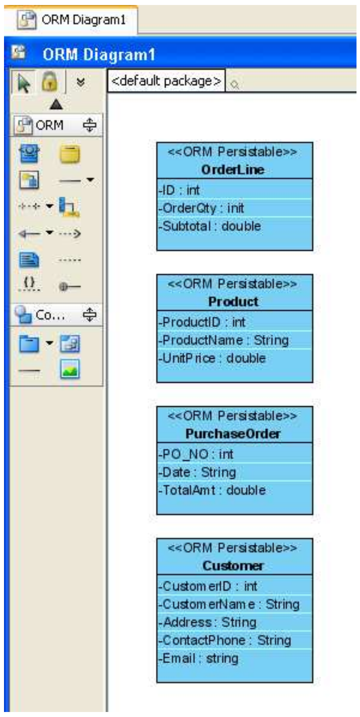


Figure 14.58 - Send Classes to a New ORM Diagram

A new ORM diagram is created which displays the ORM-Persistable classes.



<<ORM Persistable>>
OrderLine
-ID : int
-OrderQty : init
-Subtotal : double

<<ORM Persistable>>
Product
-ProductID : int
-ProductName : String
-UnitPrice : double

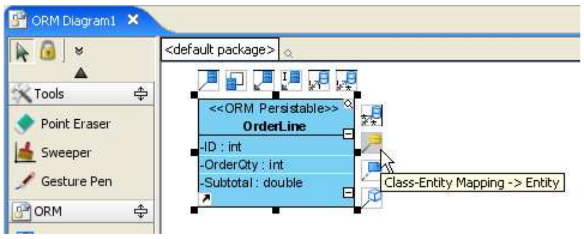
<<ORM Persistable>>
PurchaseOrder
-PO_NO : int
-Date : String
-TotalAmt : double

<<ORM Persistable>>
Customer
-CustomerID : int
-CustomerName : String
-Address : String
-ContactPhone : String
-Email : string

Figure 14.59 - The Class sent to ORM Diagram

Alternatively, you can create the ORM diagram from the data model by right-clicking on the ERD, and selecting **Send to > ORM Diagram > New ORM Diagram** from the pop-up menu.

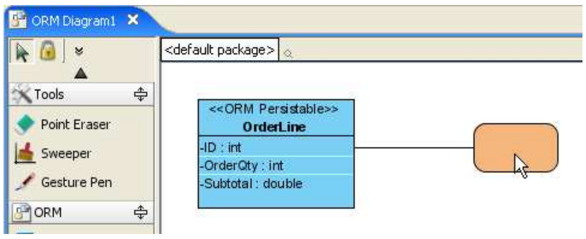
2. Mouse over the ORM Persistable class, click the **Class-Entity Mapping - > Entity** resource.



<<ORM Persistable>>
OrderLine
-ID : int
-OrderQty : int
-Subtotal : double

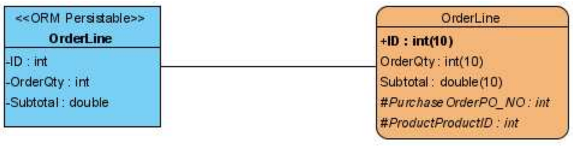
*Figure 14.60 - **Class-Entity Mapping - > Entity** resource-centric*

3. Drag the resource on the ORM diagram to show the corresponding entity associated in the mapping.



<<ORM Persistable>>
OrderLine
-ID : int
-OrderQty : int
-Subtotal : double

Figure 14.61 - Drag to the diagram



<<ORM Persistable>>
OrderLine
-ID : int
-OrderQty : int
-Subtotal : double

OrderLine
+ID : int(10)
OrderQty : int(10)
Subtotal : double(10)
#PurchaseOrderPO_NO : int
#ProductProductID : int

Figure 14.62 - The Mapping Entity will be created

A class-to-entity mapping is shown on the diagram.

4. By repeating steps 2 and 3, all class-to-entity mappings for all classes can be shown.

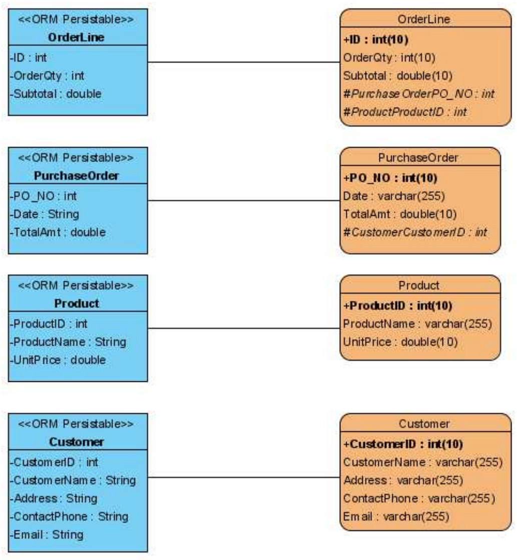


Figure 14.63 - Mapping of the classes

If you have created the ORM diagram from the data model, you can show the class-to-entity mapping by dragging the **Class-Entity Mapping - > Class** resource.

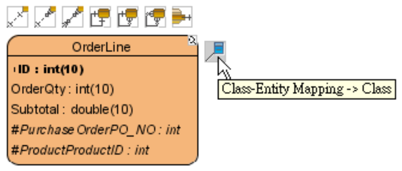


Figure 14.64 - The Class-Entity Mapping resource-centric on Entity

Drawing an ORM Diagram

1. You can create a new ORM diagram in one of the three ways:
   - On the menu, click **File > New Diagram > New ORM Diagram**.

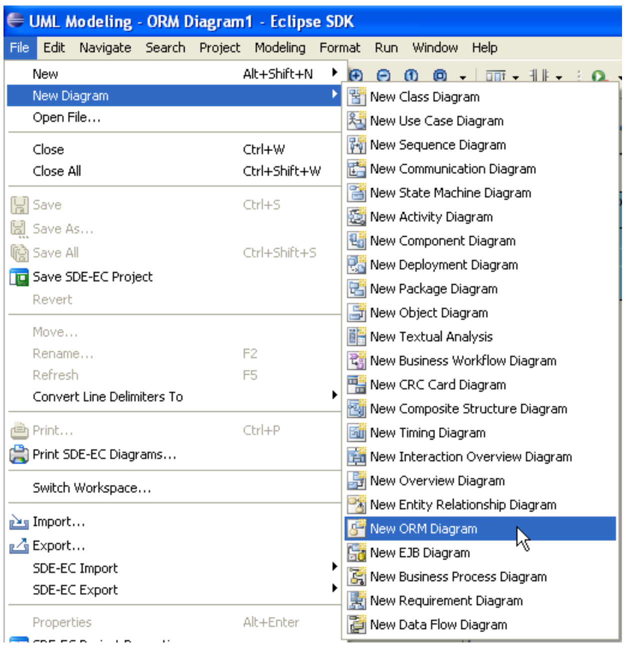


Figure 14.65 - Create new ORM Diagram

- On the **Diagram Navigator,** right-click **ORM Diagram > New ORM Diagram**.

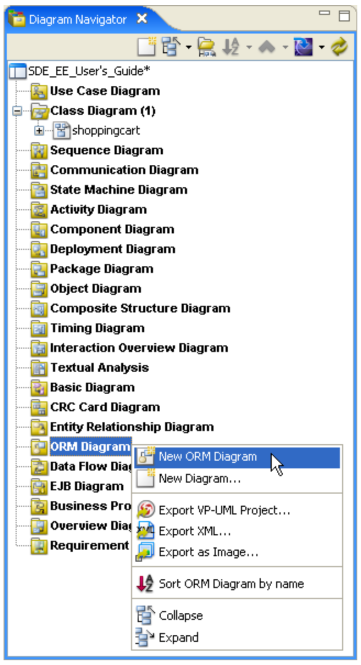


Figure 14.66 - Create new ORM Diagram in Diagram Navigator

- On the toolbar, click the **New ORM Diagram** icon.

A new ORM diagram is displayed.

Creating ORM-Persistable Class and Mapping Entity to the ORM Diagram

After created a new ORM diagram, you can create ORM-Persistable class and its mapping entity on the ORM diagram. To create an ORM-Persistable class on ORM diagram:

1. On the diagram toolbar, click **ORM-Persistable Class** shape icon.

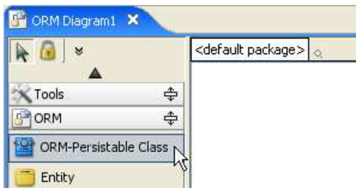


Figure 14.67 - Create ORM-Persistable Class button

2. Click a location in the diagram pane.

   A class shape icon which is marked with **<<ORM-Persistable>>** is added to the digram.

3. Type the name for the **ORM-Persistable Class**.
   - You can edit the name by double-clicking the name or by pressing the ***F2*** button.

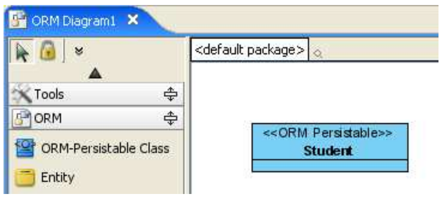


Figure 14.68 - Rename the Class name

Creating Associated ORM-Persistable Class to the ORM Diagram

You can create an associated ORM-Persistable class by using the association resources of an ORM-Persistable class.

1. Mouse over the ORM-Persistable class, drag the **One-to-One Association - > Class** resource to the diagram to create another ORM-Persistable class with a one-to-one directional association.

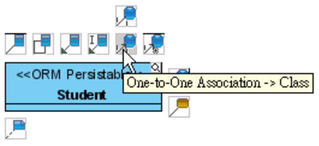


*Figure 14.69 - **One-to-one Association** -> **Class** resources-centric*

2. Enter the name to the newly created class.

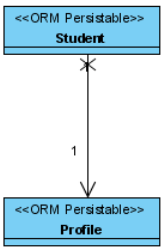


Figure 14.70 - Create a new Class with one-to-one association

Creating Mapping Entity to the ORM Diagram

To create the mapping entity of the ORM-Persistable class:

1. Mouse over the ORM-Persistable class, click and drag the **Class-Entity Mapping - > Entity** resource to the diagram.

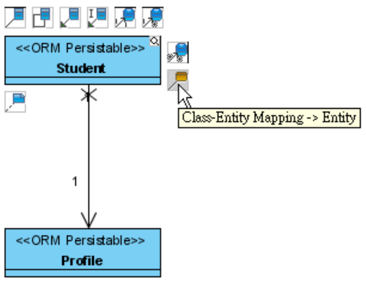


*Figure 14.71 - **Class-Entity Mapping** -> **Class** resource-centric*

The corresponding mapping entity, **Student** is created automatically.

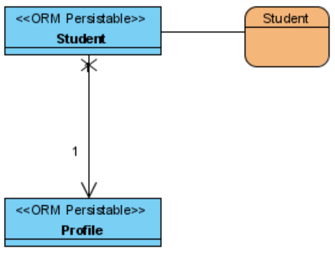


Figure 14.72 - The mapping entity are created

2. Create the mapping entity of Profile class by using the **Class-Entity Mapping - > Entity** resource.

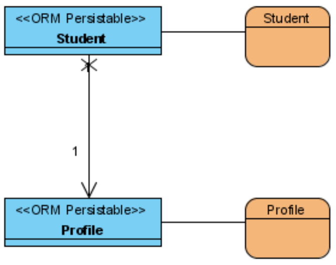


Figure 14.73 - Mapping Classes to Entities

You can create the Entity and the mapping ORM-Persistable class using the same approach of Creating ORM-Persistable Class and Mapping Entity by using the **Entity** icon on the diagram toolbar and the **Class-Entity Mapping - > Class** resource.

Showing Attribute Mapping

The object-relational mapping exists not only between the ORM-Persistable class and entity, but also the attributes and columns. You can investigate the mapping between the attributes and columns by using the Attribute Mapping feature.

To view the attribute mapping:

1. Right-click on the ORM diagram, select **View > Attribute Mapping** from the pop-up menu.

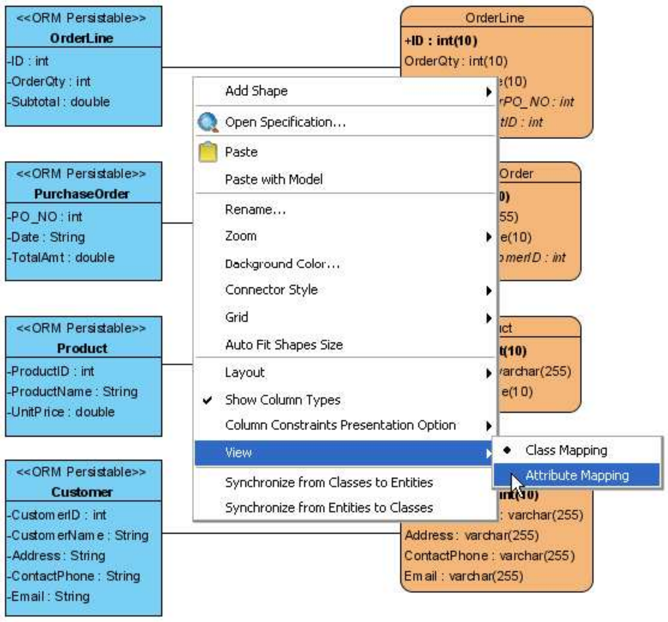


Figure 14.74 - Mapping view options

The class-to-entity mapping shown on the ORM diagram is changed to attribute-to-column mapping automatically.

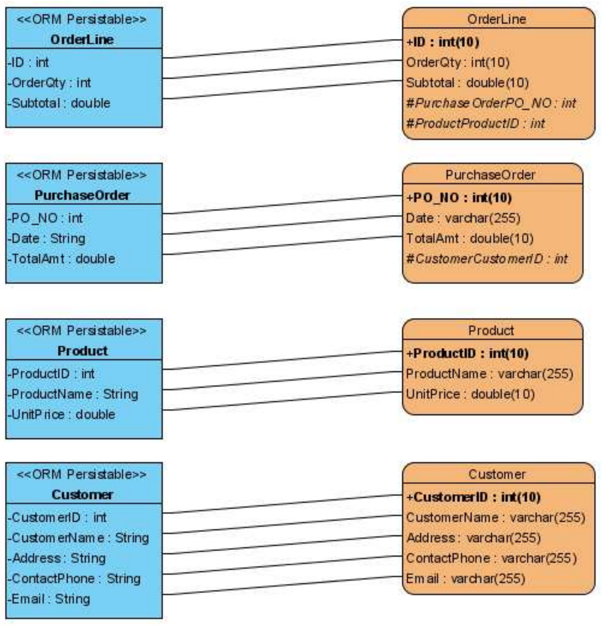


Figure 14.75 - Show mapping of attribute

Supporting Real-time Synchronization

ORM diagram supports real-time synchronization; that is, any change in the class diagram, entity relationship diagram and/or ORM diagram will automatically synchronize to each other.

Drawing ORM Diagram section as an example to modify the ORM-Persistable Class and Entity.

Forming a Class Diagram

You can create a class diagram from the existing

1. Create a new class diagram by using **New Class Diagram** icon.
2. Select the classes from the **Class Repository**, drag to the newly created class diagram.

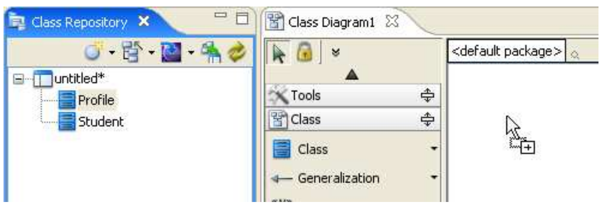


Figure 14.76 - Drag the Classes to Class Diagram

The following class diagram is created.

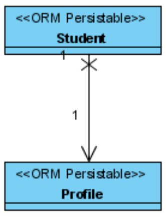


Figure 14.77 - The original classes

Modifying ORM-Persistable Class

You can modify the ORM-Persistable class such as renaming the class name and adding attributes to the class.

1. Right-click the **Student** class, select **Add** > **Attribute** from the pop-up menu.

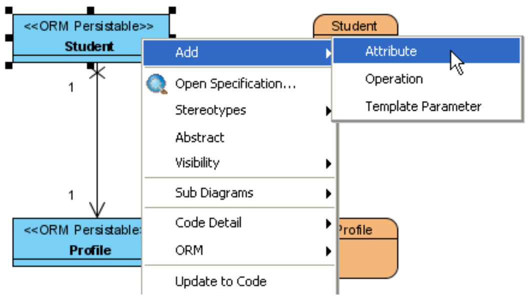


Figure 14.78 - Add an attribute to the Class

An attribute is added to the Student class, and the mapping attribute is added to the mapping Student entity automatically.

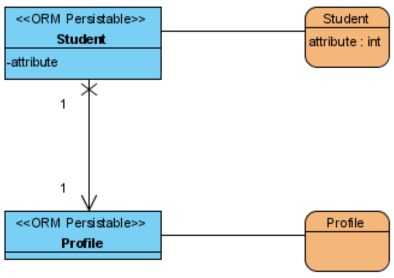


Figure 14.79 - The Entity is updated automatically

2. Enter "**StudentID : String**" to the attribute of Student by double-clicking on the attribute. The type of the mapping column is mapped to varchar(255) automatically.

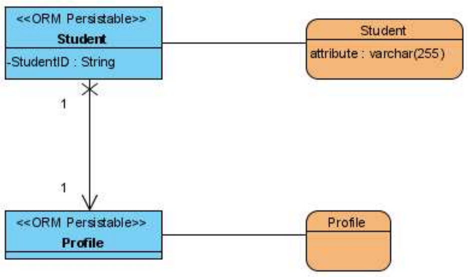


Figure 14.80 - Change the data type

Modifying Entity

You can modify the entity such as renaming the entity name and adding columns to the entity.

1. Rename the column of **Student** entity from **attribute** to **ID** by double-clicking on it.

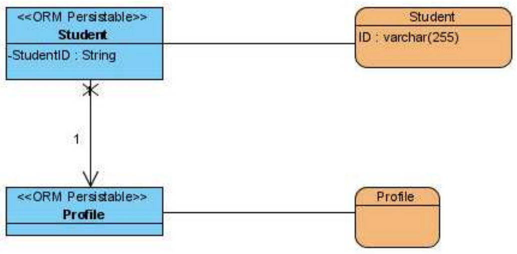


Figure 14.81 - Change the name of Class and Entity

2. Right-click the **Student** entity, select **New Column** from the pop-up menu.

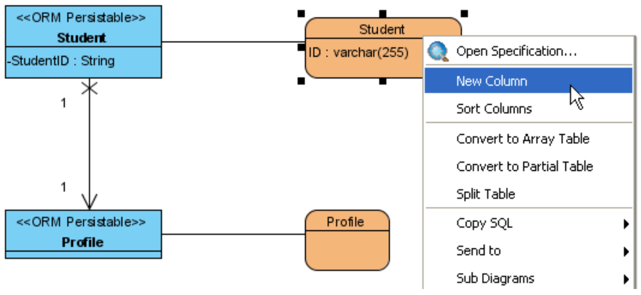


Figure 14.82 - Create Column in Entity

A new column is added to the Student entity and the corresponding attribute is added to the Student ORM-Persistable class automatically.

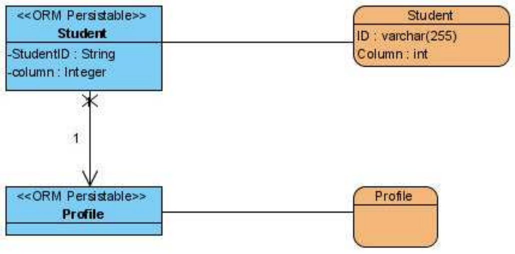


Figure 14.83 - Class is updated automatically

3. Enter "**Name : varchar(255)"** to the column by double-clicking on it. The type of the mapping attribute is mapped to String automatically.

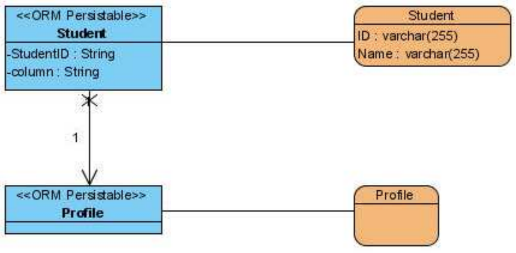


Figure 14.84 - Change the data type of Entity

4. Double-click the column attribute of the Student class, rename from **column** to **Name**.

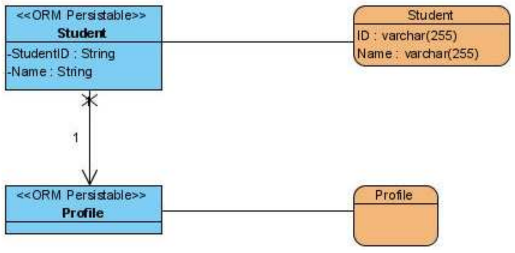


Figure 14.85 - Change the name of Column

5. Modify the classes and entities on the ORM diagram as shown below:

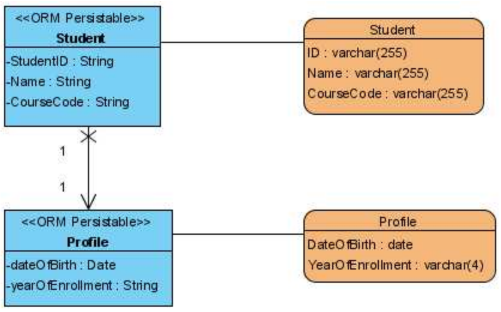


Figure 14.86 - The modified Classes and Entities

6. Navigate to the class diagram, the class diagram is updated automatically.

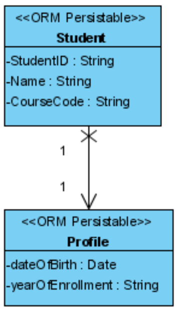


Figure 14.87 - Classes updated in Class Diagram

Switching the View of Mapping

As ORM diagram provides two views of mapping, including the mapping between ORM-Persistable class and entity (called Class Mapping), and the mapping between attributes and columns (called Attribute Mapping).

To change the view of mapping, right-click on the ORM-diagram, select the desired view from the sub-menu of **View**.

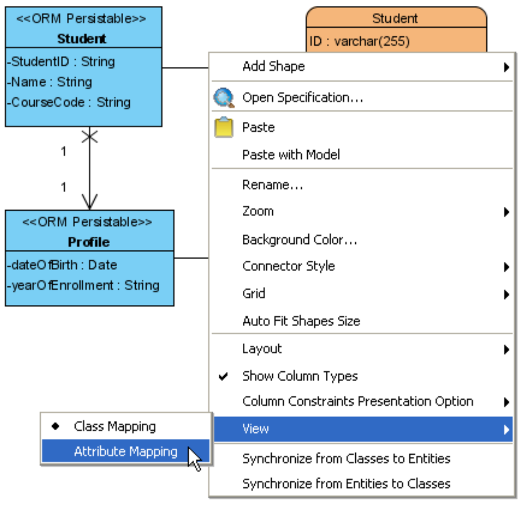


Figure 14.88 - To View attribute Mapping

By selecting the **View > Attribute Mapping** from the pop-up menu, the class-to-entity mapping shown on the ORM diagram is changed to attribute-to-column mapping automatically.

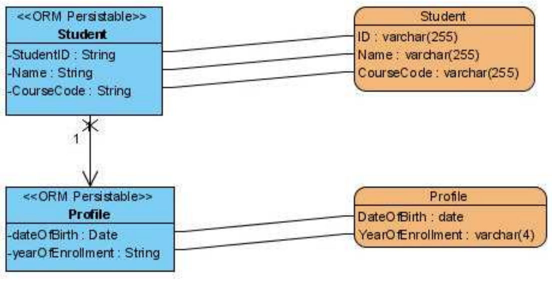


Figure 14.89 - The mapping also updated automatically

Using ORM Pane

An ORM pane is provided to generate persistent model and entity from an existing object model in Java classes and database respectively. Using the ORM pane, the existing object model and database will be transformed to ORM-Persistable class and entity; you can further develop the ORM-Persistable classes and entities by adding the desired classes and entities to the class diagram and entity relationship diagram respectively.

The ORM pane provides two views, including Class View and Database View. The class view allows you to transform the existing object model to class model while the database view allows you to transform the existing database to entity.

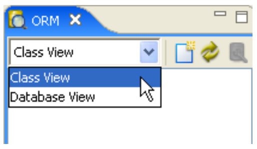


Figure 14.90 - ORM Pane

Class View

As the class view of the ORM pane supports the transformation of the existing object model into ORM-Persistable class, you are allowed to further your development based on the transformed object model.

1. Select the **Class View** of the **ORM** pane.
2. Click the **Classpath Configuration** icon.

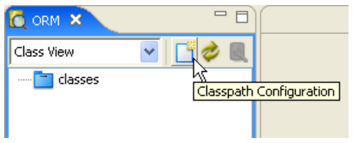


Figure 14.91 - ORM Pane (Class View)

The **Select Classpaths** dialog box is displayed.

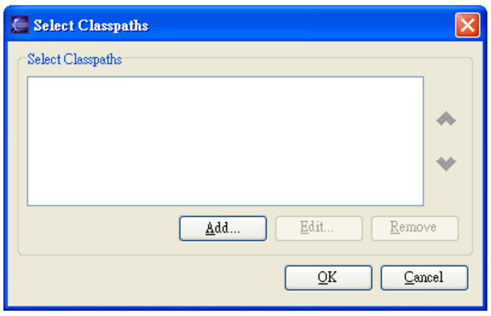


Figure 14.92 - Select Classpaths dialog

3. Click **Add...** button to select the desired classpath.

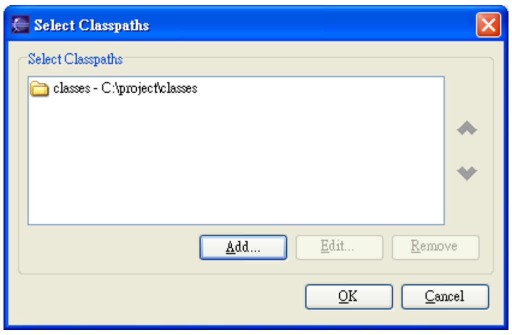


Figure 14.93 - Selected a classpath

All the available classes found from the specified classpath(s) are transformed and shown on the **ORM** pane.

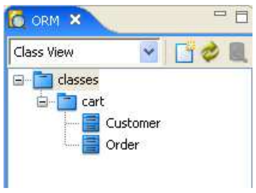


Figure 14.94 - The Classes locate in the classpath is shown

4. Create a new class diagram by using the **New Class Diagram** icon.
5. Select the desired classes and drag to the class diagram.

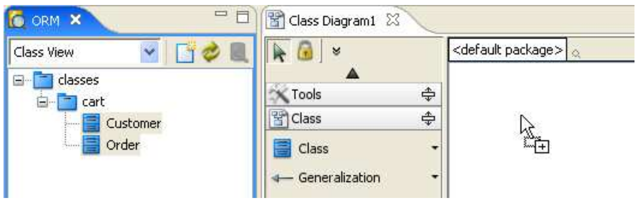


Figure 14.95 - Drag the Classes from ORM Pane to Class Diagram

The classes are added to the class diagram such that you can further develop the model by using the visual modeling feature.

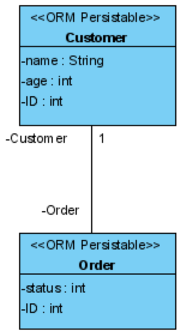


Figure 14.96 - The Classes is created in Class Diagram

Database View

As the database view of the ORM pane supports the transformation of the existing database into entity, you are allowed to alter the database schema by modeling with the entity relationship diagram and exporting to the existing database.

1. Select the **Database View** of the **ORM** pane.

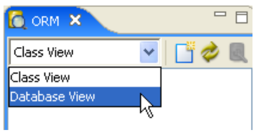


Figure 14.97 - switch to Database View

2. Click the **Database Configuration** icon.

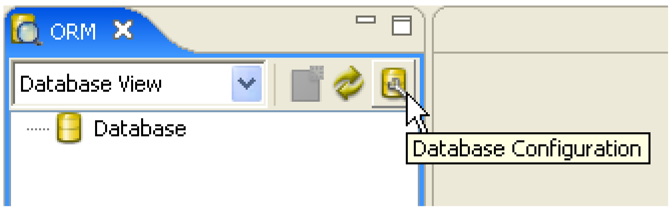
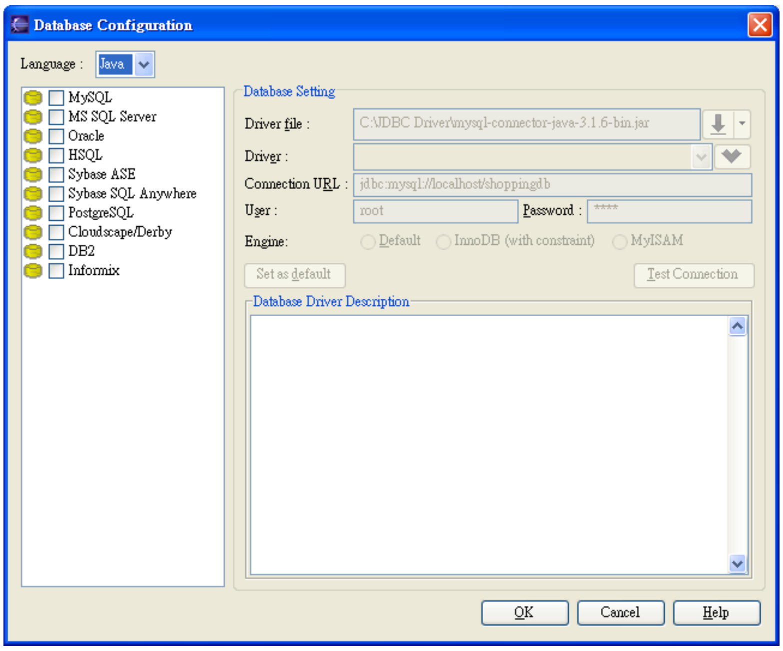


Figure 14.98 - To set the Database Configuration

The **Database Configuration** dialog box is displayed.



Database Configuration

Language : Java

MySQL
MS SQL Server
Oracle
HSQL
Sybase ASE
Sybase SQL Anywhere
PostgreSQL
Cloudscape/Derby
DB2
Informix

Database Setting

Driver file : C:\JDBC Driver\mysql-connector-java-3.1.6-bin.jar
Driver :
Connection URL : jdbc:mysql://localhost/shoppingdb
User : root Password : ****
Engine: Default InnoDB (with constraint) MyISAM

Set as default Test Connection

Database Driver Description

OK Cancel Help

Figure 14.99 - Database Configuration dialog

3. Configure the database connection by using the **Database Configuration** dialog box. Refer to the description in the Getting Started with Object-Relational Mapping chapter for information on how to configure the database in the Database Configuration for Java Project and Database Configuration for .Net Project sections. If the database is successfully connected, the tables of the connected database are transformed into entities and shown on the **ORM** pane.

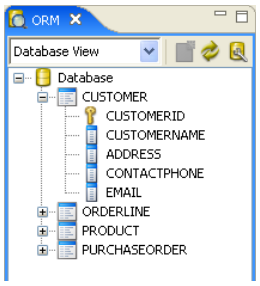


Figure 14.100 - The Table in database is shown

4. Create a new entity relationship diagram by using the **New Entity Relationship Diagram** icon.
5. Select the desired entities from the **ORM** pane, drag to the entity relationship diagram.

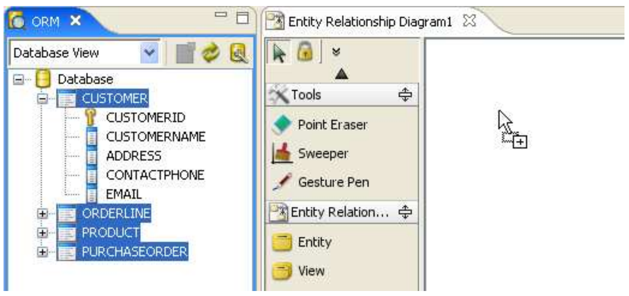


Figure 14.101 - Drag the entity from ORM Pane to ERD

The selected entities are added to the class diagram allowing you alter the database schema by visual modeling.

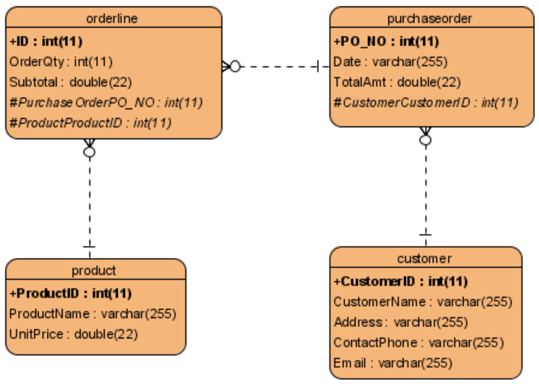


Figure 14.102 - The entities is created on the ERD

Reverse Engineering Java Classes to Object Model

You can reverse engineer the Java classes into object model with ORM-Persistable stereotyped.

To reverse engineer Java classes:

1. On the menu, click **Modeling** > **ORM** > **Reverse Java Classes...** .

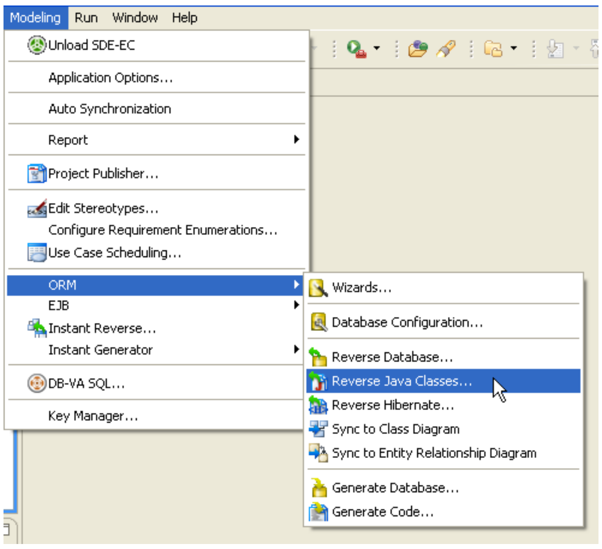


Figure 14.103 - To reverse Java Classes

For other SDE:

SDE	Method
SDE for JBuilder	From the menu, click **Tools** > **Modeling** > **ORM** > **Reverse Java Classes...** .
SDE for NetBeans	From the menu, click **Modeling** > **ORM** > **Reverse Java Classes...** .
SDE for IntelliJ IDEA	From the menu, click **Modeling** > **ORM** > **Reverse Java Classes...** .
SDE for JDeveloper	From the menu, click **Model** > **ORM** > **Reverse Java Classes...** .
SDE for WebLogic Workshop	From the menu, click **Modeling** > **ORM** > **Reverse Java Classes...** .

Table 14.10

The **Reverse Java Classes** dialog box is displayed.

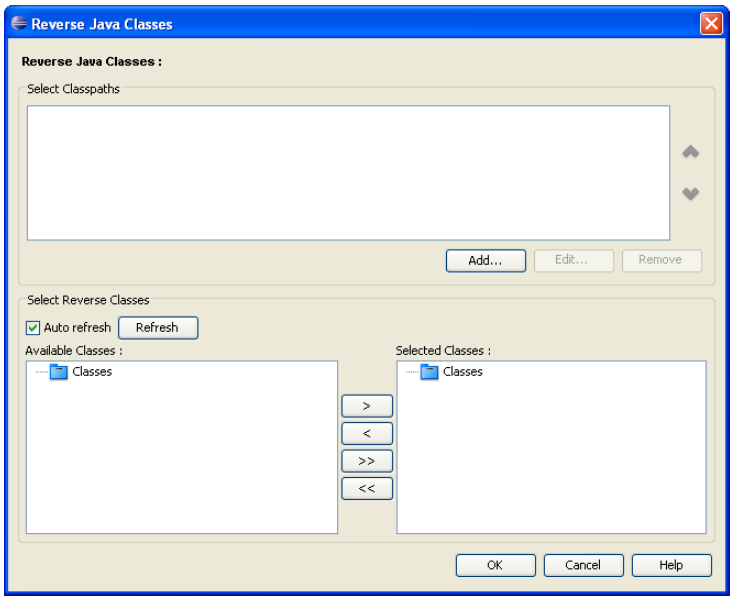


Figure 14.104 - Reverse Java Classes dialog

2. Click **Add...** to select the classpath of the Java classes to be reversed. The classpath can be a folder, zip file or jar file. After finished selecting the classpath, the classpath is added to the list of **Select Classpaths**, and the classes identified from the classpath are shown in the list of **Available Classes**.

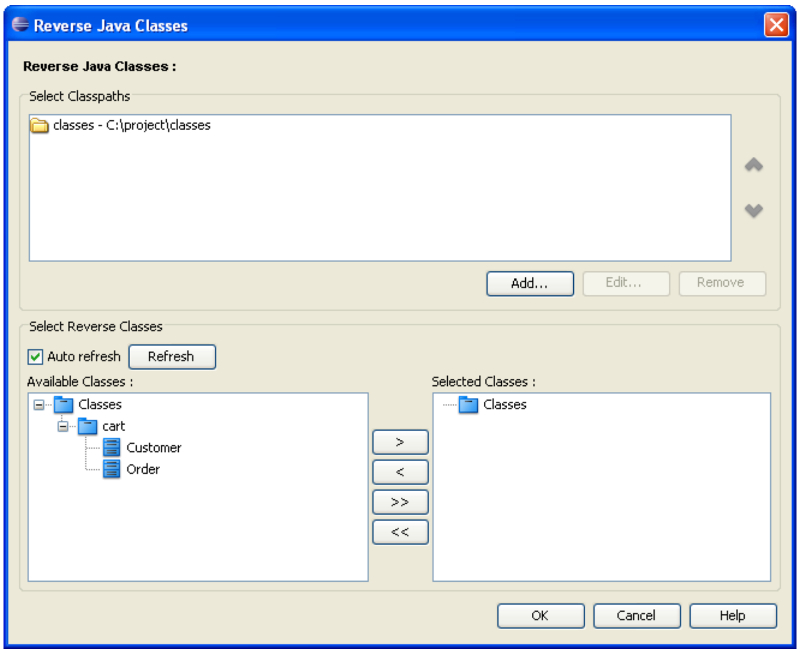


Figure 14.105 - The classes in the classpath is shown

3. Select the desired classes by using the list of buttons between the list of **Available Classes** and **Selected Classes**.

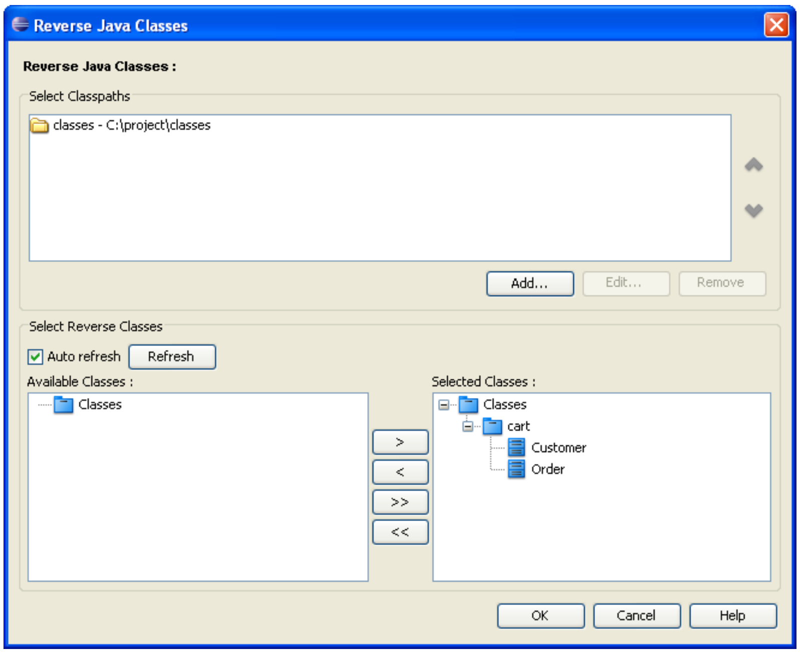


Figure 14.106 - Select the classes for reverse

4. Click **OK**. The selected classes are reverse engineered to class models which can be found under the **Model** tree.

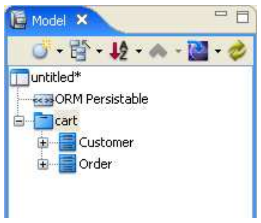


Figure 14.107 - The classes is imported and show in Model Pane

To work on the reversed class models, simply add the reversed models to the

1. Create a new class diagram by using the **New Class Diagram** icon.
2. Select the classes from the **Model** tree, drag the classes to the newly created class diagram.

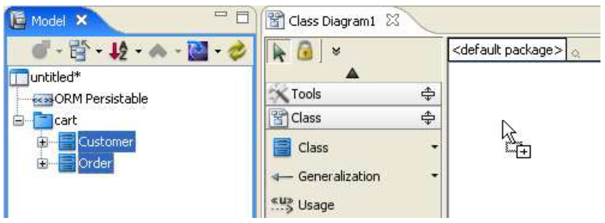


Figure 14.108 - Drag the Classes from Model Pane to Class Diagram

The classes are added to the class diagram accordingly. The classes shown on the class diagram are stereotyped as ORM Persistable; meaning that the Java classes are reversed engineered to ORM Persistable classes supporting the object relational mapping.

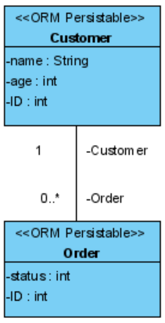


Figure 14.109 - The Classes show in the Class Diagram

Reverse Engineering Hibernate Model to Object Model

You are allowed to reverse engineer not only the Java classes, but also the hibernate model to object model with ORM-Persistable stereotyped. The database configuration is also reverse engineered as the database setting is defined in the hibernate model.

To reverse engineer Hibernate model:

1. On the menu, click **Modeling** > **ORM** > **Reverse Hibernate...** .

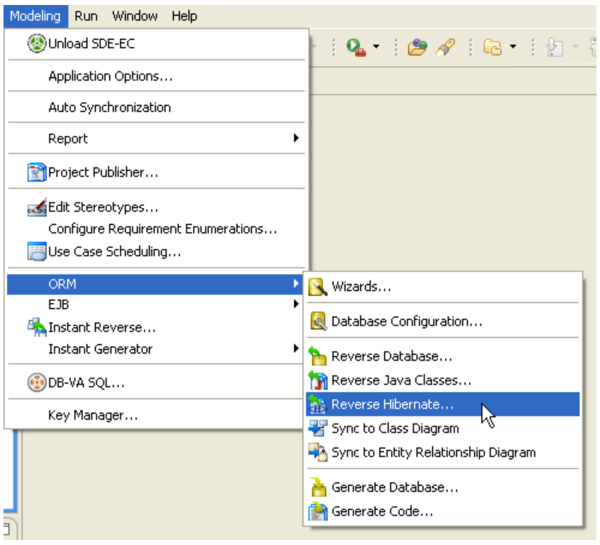


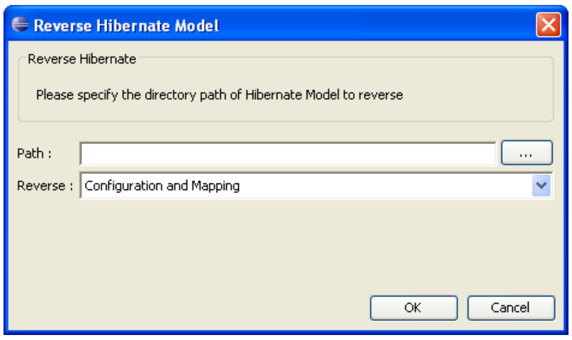
Figure 14.110 - To reverse Hibernate

For other SDE:

SDE	Method
SDE for JBuilder	From the menu, click **Tools** > **Modeling** > **ORM** > **Reverse Hibernate...** .
SDE for NetBeans	From the menu, click **Modeling** > **ORM** > **Reverse Hibernate...** .
SDE for IntelliJ IDEA	From the menu, click **Modeling** > **ORM** > **Reverse Hibernate...** .
SDE for JDeveloper	From the menu, click **Model** > **ORM** > **Reverse Hibernate...** .
SDE for WebLogic Workshop	From the menu, click **Modeling** > **ORM** > **Reverse Hibernate...**.

Table 14.11

The **Reverse Hibernate Model** dialog box is displayed.


Reverse Hibernate Model

Reverse Hibernate

Please specify the directory path of Hibernate Model to reverse

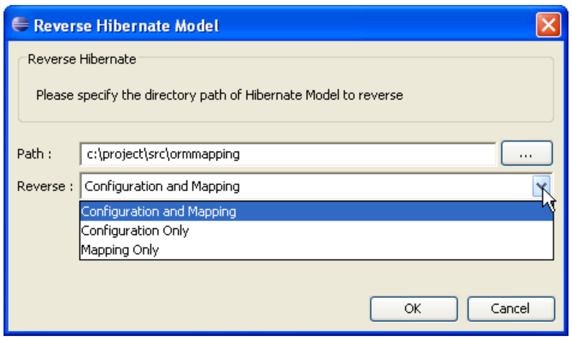
Path :

Reverse : Configuration and Mapping

Figure 14.111 - Reverse Hibernate Model dialog

2. Select the path of the Hibernate xml files by using the ... button.

3. Select the type of reverse engineering to be performed from the drop-down menu of Reverse, either **Configuration and Mapping**, **Configuration only** or **Mapping only**.


Reverse Hibernate Model

Reverse Hibernate

Please specify the directory path of Hibernate Model to reverse

Path : c:\project\src\ormmapping

Reverse : Configuration and Mapping

Figure 14.112 - Select the type of reverse engineering to performed

4. Click **OK**. The hibernate model are reverse engineered to class models and entities which can be found under the **Model** tree.

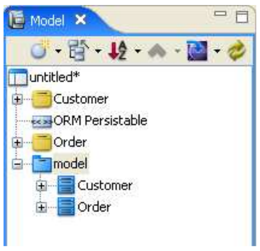


Figure 14.113 - The hibernate model are reversed

After reverse engineered the hibernate model, you can use an ORM diagram to

1. Create a new ORM diagram by using the **New ORM Diagram** icon.
2. Add the reversed classes to the ORM diagram by dragging the class models from the **Model** tree to the ORM diagram.

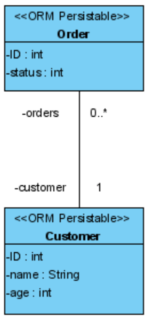


Figure 14.114- Drag the classes to ORD Diagram

3. Drag the entities from the **Model** tree to the ORM diagram to add the entities to the ORM diagram.

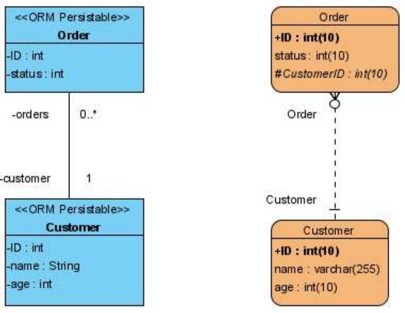


Figure 14.115 Add the Entity to the ORM Diagram

4. Right-click the ORM diagram, select **View** > **Attribute Mapping** from the pop-up menu.

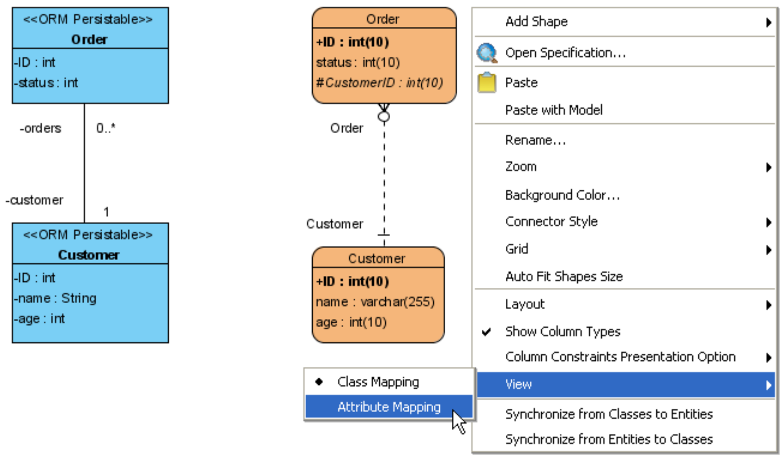


Figure 14.116 - To view the attribute mapping

The mapping between the attributes of class models and columns of entities are shown.

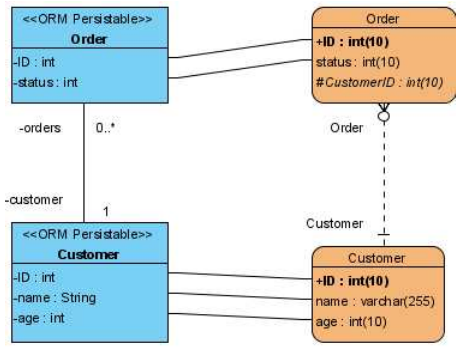


Figure 14.117 - The mapping of attribute is shown

You can also check the reversed engineered database connection by the following steps:

1. Click the **Database Configuration** icon, to open the **Database Configuration** dialog box.

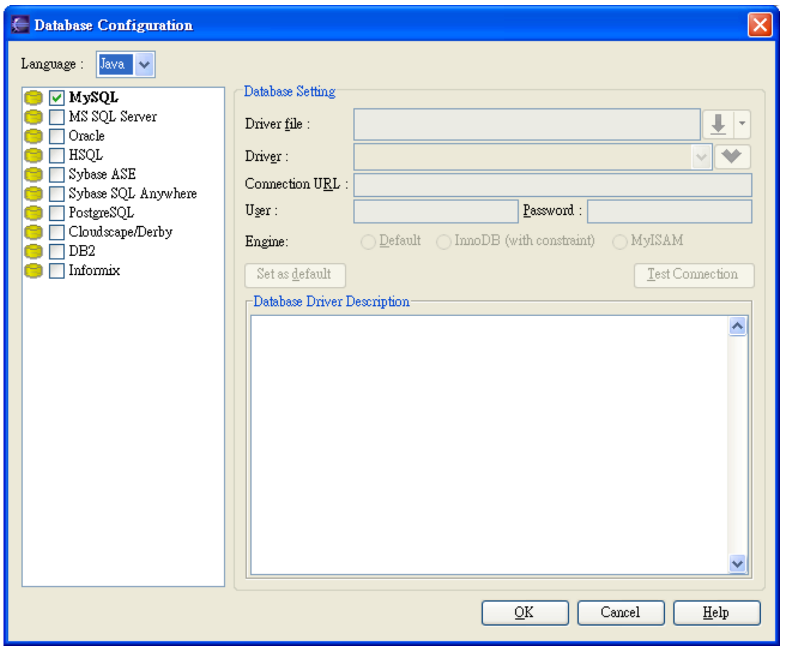
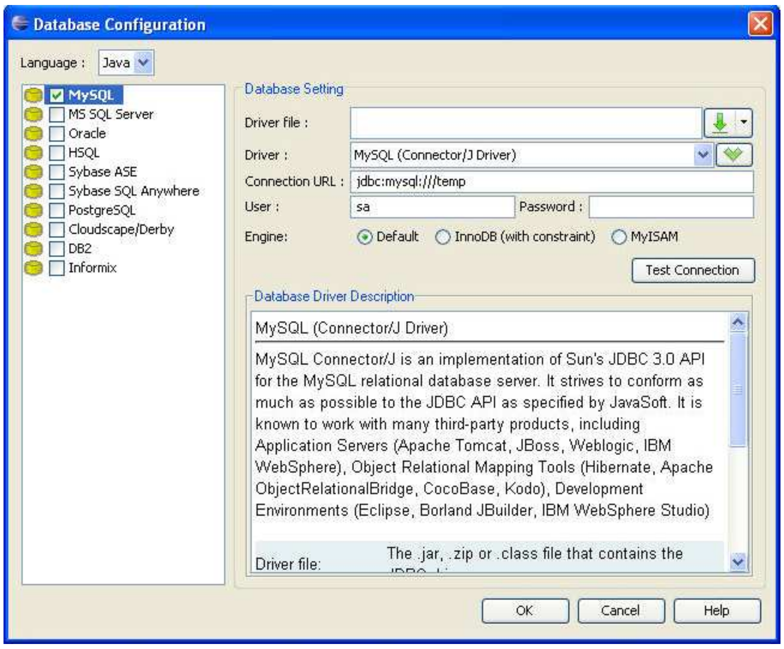


Figure 14.118 - Database configuration dialog

2. Select the connected database from the **Database Configuration** dialog box, the **Database Setting** is shown which shows the database configuration has been reversed successfully.



Database Configuration

Language : Java

- [✓] MySQL
- [ ] MS SQL Server
- [ ] Oracle
- [ ] HSQL
- [ ] Sybase ASE
- [ ] Sybase SQL Anywhere
- [ ] PostgreSQL
- [ ] Cloudscape/Derby
- [ ] DB2
- [ ] Informix

Database Setting

Driver file :

Driver : MySQL (Connector/J Driver)

Connection URL : jdbc:mysql:///temp

User : sa Password :

Engine: [✓ Default] InnoDB (with constraint) MyISAM

Test Connection

Database Driver Description

MySQL (Connector/J Driver)

MySQL Connector/J is an implementation of Sun's JDBC 3.0 API for the MySQL relational database server. It strives to conform as much as possible to the JDBC API as specified by JavaSoft. It is known to work with many third-party products, including Application Servers (Apache Tomcat, JBoss, Weblogic, IBM WebSphere), Object Relational Mapping Tools (Hibernate, Apache ObjectRelationalBridge, CocoBase, Kodo), Development Environments (Eclipse, Borland JBuilder, IBM WebSphere Studio)

Driver file: The .jar, .zip or .class file that contains the

OK Cancel Help

Figure 14.119 - Select the Database and fill in the setting

Enterprise JavaBeans Modeling

Enterprise JavaBeans (EJB) model is another object model which describes the components of the application system. It is built on the JavaBeans technology which supports the remote and distributed application. It also supports distributing program components from the server to the clients in a network.

There are three Enterprise JavaBeans model, including Entity Bean, Session Bean and Message-Driven Bean. Entity Bean represents the persistent data maintained in a database. Session Bean is created by a client and only exists during a single client-server session which contains operations for transaction. Message-Driven Bean is a bean which allows the J2EE application to process messages asynchronously.

An EJB diagram can be used to describe the objects and classes inside a system and the relationships between them. It is commonly used to depict a distributed application. You can create an Enterprise JavaBeans model by using an EJB diagram.

Creating an EJB Diagram

There are two ways to create an EJB Diagram:

1. Drawing an EJB Diagram
2. Synchronizing from a Data Model to an Enterprise JavaBeans Model

Drawing an EJB Diagram

1. You can create a new EJB Diagram in one of the three ways:
   - On the menu, click **File > New Diagram > New EJB Diagram**.

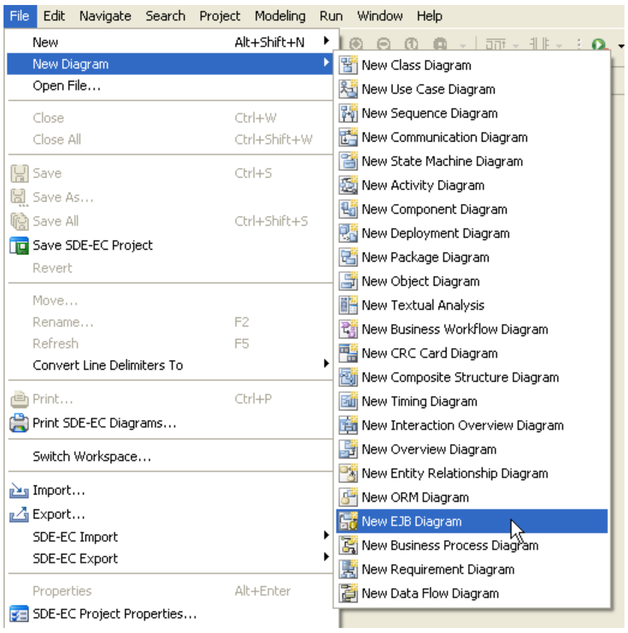


Figure 14.120 - Create EJB Diagram

   - On the **Diagram Navigator**, right-click **EJB Diagram > New EJB Diagram**.

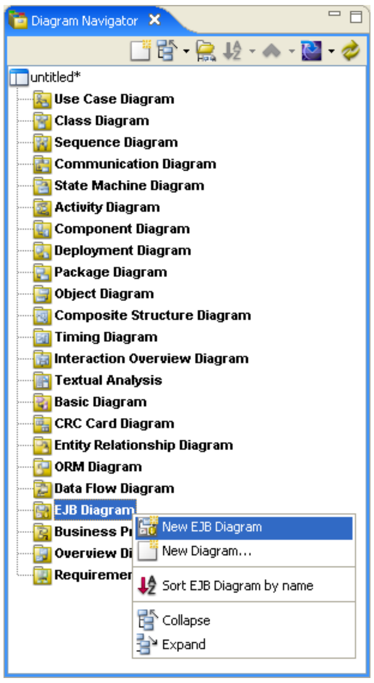


Figure 14.121 - Create EJB Diagram by click on Diagram Navigator

   - On the toolbar, click the **New EJB Diagram** icon.

A new EJB Diagram pane is displayed.

Creating a new Entity Bean element to the EJB Diagram

1. On the diagram toolbar, click the **Entity Bean** shape icon.

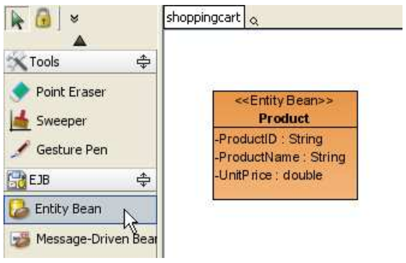


Figure 14.122 - Entity Bean

2. Click a location in the diagram pane.

   An icon representing the entity bean element is added to the diagram.

3. Type in a name for the **Entity Bean** element.
   - You can edit the name by double-clicking the name or by pressing the ***F2*** button.

Creating a new Message-Driven Bean element to the EJB Diagram

1. On the diagram toolbar, click the **Message-Driven Bean** shape icon.

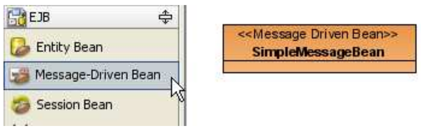


Figure 14.123 - Message Driven Bean

2. Click a location in the diagram pane.

   An icon representing the message driven bean element is added to the diagram.

3. Type a name for the **Message-Driven Bean** element.
   - You can edit the name by double-clicking the name or by pressing the **F2** button.

Creating a new Session Bean to the EJB Diagram

1. On the diagram toolbar, click the **Session Bean** shape icon.

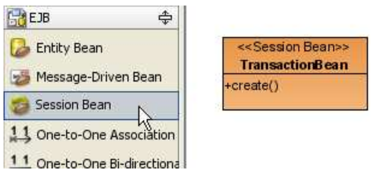


Figure 14.124 - Session Bean

2. Click a location in the diagram pane.

   An icon representing the session bean element is added to the diagram.

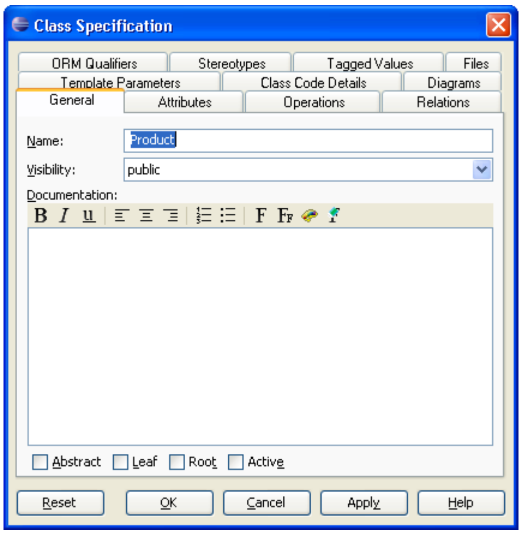
3. Type a name for the **Session Bean** element.
   - You can edit the name by double-clicking the name or by pressing the **F2** button.

Modifying the Class Specification of Entity, Message Driven and Session Bean

A class specification of a bean displays the bean properties and relationships.

1. To display the **Class Specification** of a bean, right-click the bean element, click **Open Specification**.

   Class Specification dialog box is displayed, you can modify bean properties and relationships.


Class Specification

ORM Qualifiers | Stereotypes | Tagged Values | Files
Template Parameters | Class Code Details | Diagrams
General | Attributes | Operations | Relations

Name: Product
Visibility: public
Documentation:

Abstract | Leaf | Root | Active

Reset | OK | Cancel | Apply | Help

Figure 14.125 - Class Specification dialog

Adding new Attribute to the Entity, Message Driven and Session Bean

An attribute is a property of an Entity, Message Driven and Session Bean which has a name, a value and also has a type to describe the characteristic of an object.

1. You can add attribute to the Entity, Message Driven and Session Bean in one of the two ways:
   - Right-click on a bean, select **Add** > **Attribute**.

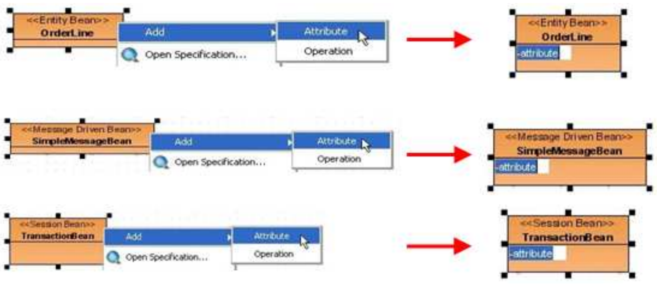
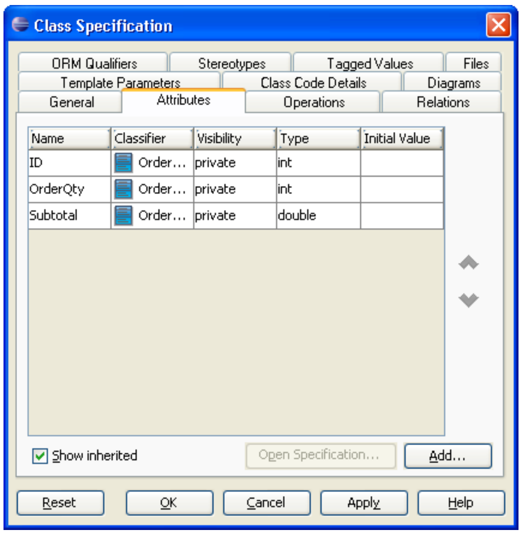


Figure 14.126 - Add Attributes on the classes

   A new attribute is added, type the attribute name and type in the form of "attribute_name:type". You can also edit the attribute name by double-clicking the attribute name or by pressing the ***F2*** button.

   - Alternative
     1. Right-click the bean element, click **Open Specification**.
     2. Click the **Attributes** Tab, the click **Add**.

Attribute Specification dialog box is displayed, you can modify the attribute name and properties, such as type.



Name	Classifier	Visibility	Type	Initial Value
ID	Order...	private	int	
OrderQty	Order...	private	int	
Subtotal	Order...	private	double	

Figure 14.127 - Class Specification dialog (Attributes tab)

Adding Association to the Entity Bean

An association refers to the relationship specifying the type of link that exists between objects. In the Enterprise JavaBeans Modeling, an association can only exist between entity beans; and hence, an association shows how the entity beans are related to each other.

1. You can add an association to the Entity Bean in one of the two ways:
   - Using Resource-Centric Interface
     1. Click on an Entity Bean, a group of valid editing resources are displayed around the Entity Bean element.
     2. Drag the selected resource of association such as "**One-to-One Association - > Entity Bean**" to the associated Entity Bean class.

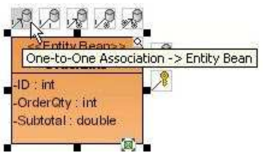


*Figure 14.128 - **One-to-one Association** resource-centric*

   - Using Toolbar icon
     1. On the diagram toolbar, click the **Association** icon, such as "**One-to-One Association**".

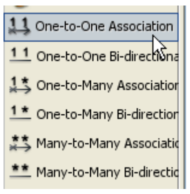


*Figure 14.129 - **One-to-one Association** button*

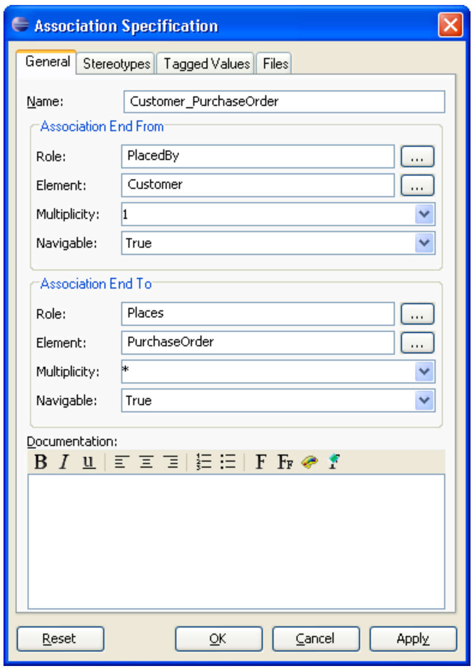
     2. Click on an Entity Bean class, drag to another Entity Bean element.

A line indicating a connection between the two Entity Beans is shown.

Editing Association Specification

1. You can edit the association specification in one of the three ways:
   - Using Open Specification
     1. Right-click on the connection line, click **Open Specification** from popup menu.

        Association Specification dialog box is displayed, you have to modify the association properties, Roles of Entity Beans in Association End From and To, Multiplicity and Navigation etc.



Association Specification

General | Stereotypes | Tagged Values | Files

Name: Customer_PurchaseOrder

Association End From
- Role: PlacedBy
- Element: Customer
- Multiplicity: 1
- Navigable: True

Association End To
- Role: Places
- Element: PurchaseOrder
- Multiplicity: *
- Navigable: True

Documentation:

Reset | OK | Cancel | Apply

Figure 14.130 - Association Specification dialog

   - Using Pop-up Menu
     1. Right-click on the connection line, the property of the association specification is displayed in the pop-up menu, including **Multiplicity**, **Navigable**, **Aggregation Kind** and **Role Name**.
     2. Select the property that you want to edit, check the desired value.

If you right-click on the connection line towards an entity bean, the pop-up window shows the properties of association specification of the respective entity bean. If you right-click in the middle of the connection line, the pop-up window shows all properties of association specification of both entity beans.

Role name of the entity bean describes how it acts in the association which will be used in the generation of persistent code. Be sure that you have given the role names to the entity bean in the association in order to proceed to the generation of code.

   - Using Property Pane
     1. On menu, click **Window** > **Show View** > **Property**.
     2. The property pane will be displayed.
     3. Click on the connection line.

        The properties of the association specification are displayed in the property pane. You can edit the property under the property pane.

Synchronizing from a Data Model to an Enterprise JavaBeans Model

You can generate an EJB Diagram from an ERD by synchronization if there is an ERD.

1. You can synchronize the ERD to EJB Diagram in one of the two methods:
    - On the menu, click **Modeling > EJB > Sync to EJB Diagram**.

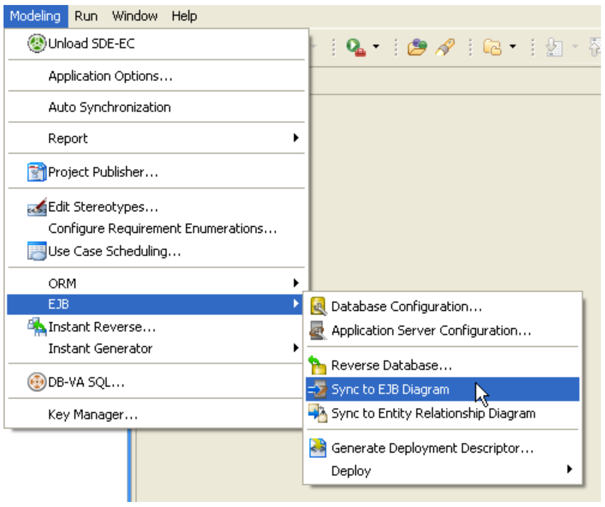


Figure 14.131 - Synchronize ERD to EJB Diagram

For other SDE:

SDE	Method
SDE for JBuilder	On the menu, click **Tools > Modeling > EJB > Sync to EJB Diagram**.
SDE for NetBeans	On the menu, click **Modeling > EJB > Sync to EJB Diagram**.
SDE for IntelliJ IDEA	On the menu, click **Modeling > EJB > Sync to EJB Diagram**.
SDE for JDeveloper	On the menu, click **Model > EJB > Sync to EJB Diagram**.
SDE for WebLogic Workshop	On the menu, click **Modeling > EJB > Sync to EJB Diagram**.

Table 14.12

- Right-Click on the ERD, select **Sync to EJB Diagram**.

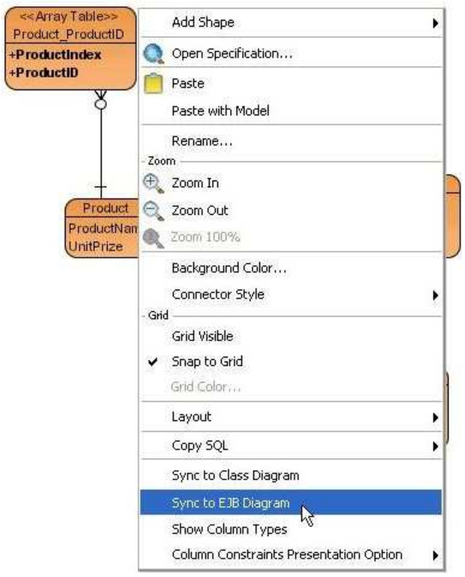


Figure 14.132 - Sync to EJB Diagram by click on popup menu

An EJB Diagram is generated and can be found under the **Diagram Navigator**.

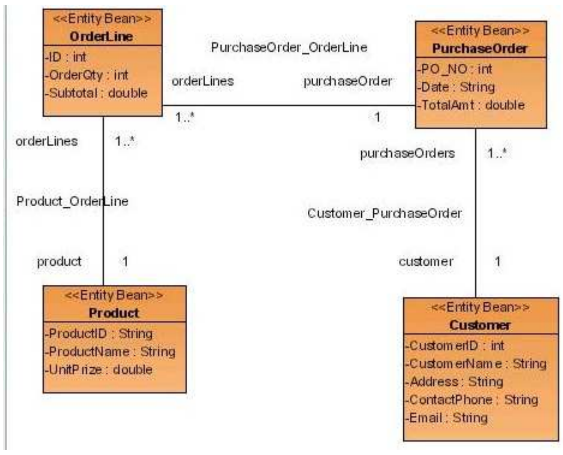


Figure 14.133 - The generated EJB Diagram

Defining Primary Key

You can define an attribute to a primary key column in one of the three ways:

- Using **Include in Primary Key**
  1. Select and right-click on attribute, select **Include in primary key**.

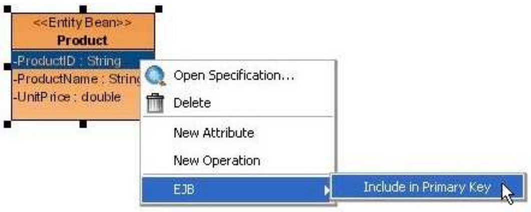
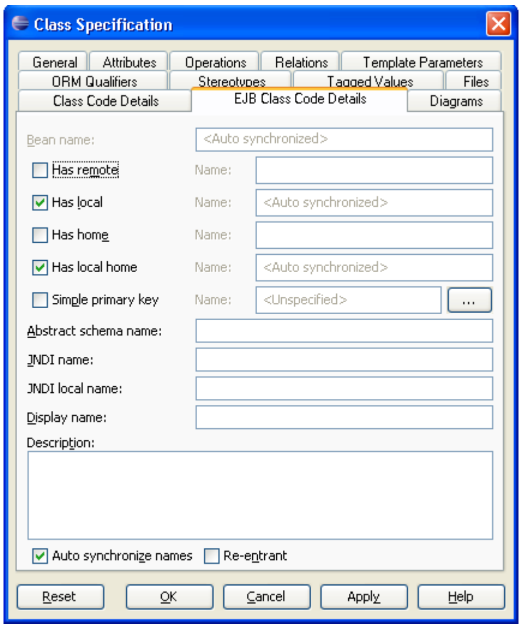


Figure 14.134 - Add a attribute to a primary key

- Using **Class Specification** dialog box
  1. Right-click on the Entity Bean, click **Open Specification** from popup menu to open the **Class Specification** dialog box.
  2. Select **EJB Class Code Details** Tab; click the "**...** " button to open the **Select CMP Fields** dialog box.



Bean name: <Auto synchronized>

☐ Has remote	Name:	
☑ Has local	Name:	<Auto synchronized>
☐ Has home	Name:	
☑ Has local home	Name:	<Auto synchronized>
☐ Simple primary key	Name:	<Unspecified> ...

Abstract schema name:

JNDI name:

JNDI local name:

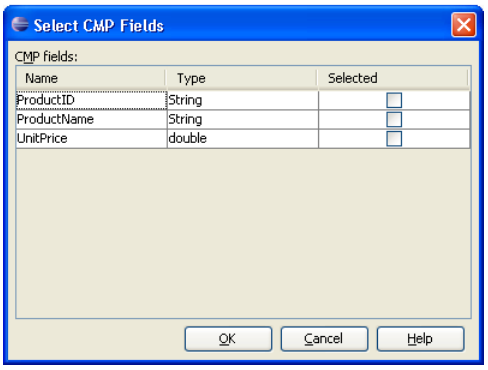
Display name:

Description:

☑ Auto synchronize names ☐ Re-entrant

Figure 14.135 - Class Specification dialog(EJB Code Details tab)

3. Check the **Selected** option for the **CMP fields** which will be included in a primary key.



Name	Type	Selected
ProductID	String	☐
ProductName	String	☐
UnitPrice	double	☐

Figure 14.136 - Select CMP Fields dialog

If the **Simple primary key** option is checked, simple primary key is allowed; that is, only one attribute can be defined as the primary key. Otherwise, more than one attribute can be selected to form a composite primary key in a Primary Key Class.

- Using Resource-Centric Interface
  1. Click on an Entity Bean, a group of valid editing resources are displayed around the Entity Bean Class.
  2. Drag the resource of "**Create Primary Key**" to the diagram pane.

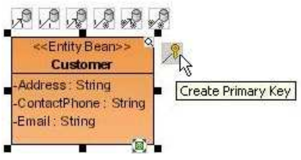


*Figure 14.137 - **Create Primary Key** resource-centric*

A class stereotyped as **Entity Bean Primary Key** is created.

1. Add the attributes for the Primary Key Class which will be added to the corresponding Entity Bean automatically.

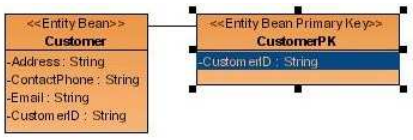


Figure 14.138 - The Entity Bean Primary Key Class

The Primary Key Class for the Entity Bean will not map to an additional entity in the entity relationship diagram. The Primary Key Class will be used in the generation of code.

Creating a Creator Method

A Creator Method is used to create an entity bean instance at the server side such that the server will return a remote interface to the client.

You can add a creator method to an Entity Bean by one of the two ways.

- Using **Creator Method Wizard**
  1. To activate the **Creator Method Wizard**, right-click on the Entity Bean, select **EJB > Create Creator Method**.

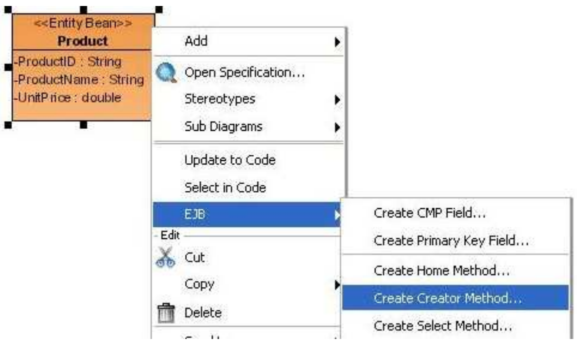
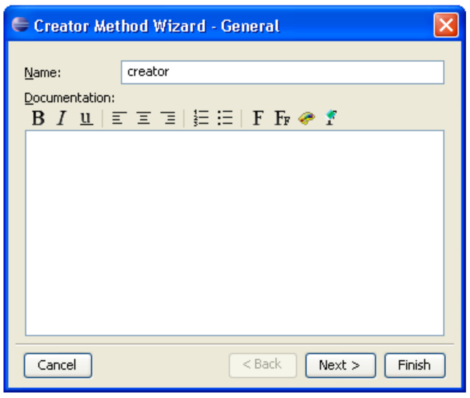


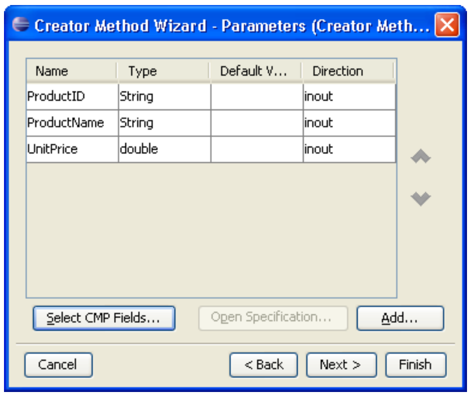
Figure 14.139 - To create Creator Methods

  2. The **Creator Method Wizard** will be shown, you can modify the **Name** for the Creator Method and then click **Next >.**



Name: creator

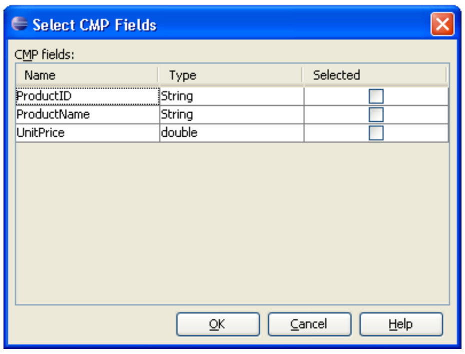
Figure 14.140 - Creator Method Wizard dialog



Name	Type	Default V...	Direction
ProductID	String		inout
ProductName	String		inout
UnitPrice	double		inout

Figure 14.141 - Add a parameter

3. To specify the parameters for the Creator Method, you can add the parameters in one of the two ways:
   - Click **Select CMP Fields...** button, the Select CMP Fields dialog box will be displayed. Check the **Selected** option for the **CMP fields** which will be added as the parameters for the Creator Method.

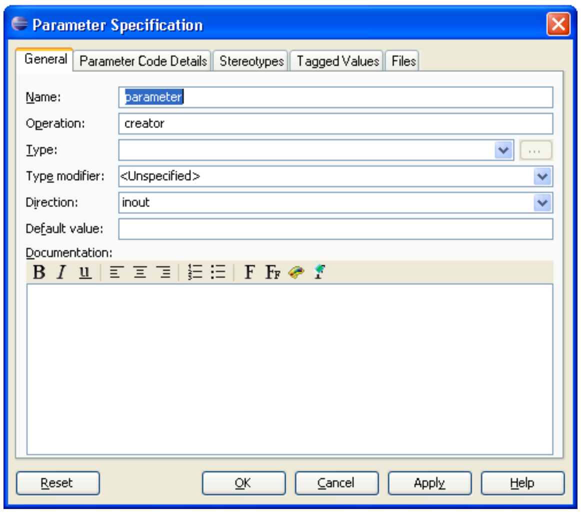


CMP fields:

Name	Type	Selected
ProductID	String	☐
ProductName	String	☐
UnitPrice	double	☐

Figure 14.142 - Select CMP Fields dialog

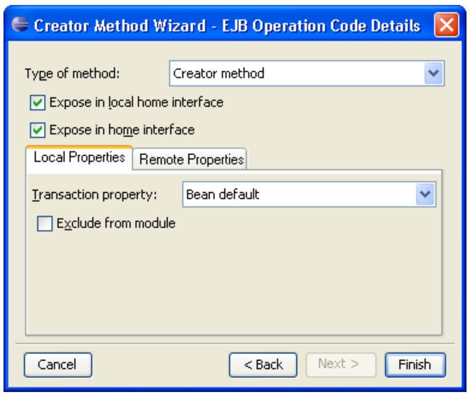
   - Click **Add...** button, a **Parameter Specification** dialog box is displayed, and you can modify the parameter name and properties, such as type.



Name: parameter
Operation: creator
Type:
Type modifier: <Unspecified>
Direction: inout
Default value:
Documentation:

Figure 14.143 - Parameter Specification dialog

4. Click **Next >,** the **EJB Operation Code Details** is shown which allows you to define the **Local Properties** and **Remote Properties** for the Creator Method.



Type of method: Creator method
☑ Expose in local home interface
☑ Expose in home interface
Transaction property: Bean default
☐ Exclude from module

Figure 14.144 - EJB Operation Code Details

As **Create Creator Method...** was selected from the pop-up menu of the entity bean, the Type of method of the **EJB Operation Code Details** of the wizard will be set as **Creator method** by default.

5. Click **Finish**, the **Creator Method** is added to the Entity Bean.

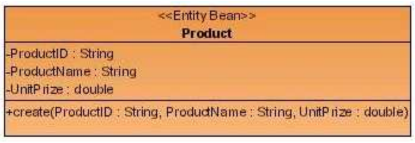


Figure 14.145 - The Entity Bean Class with creator method

- Using **Operation Specification** dialog box.
  5. Right-click on the Entity Bean, select **Add > Operation** to add a new operation.

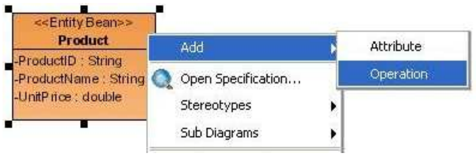


Figure 14.146 - To Add operation

6. Right-click on the operation, select **Open Specification**.

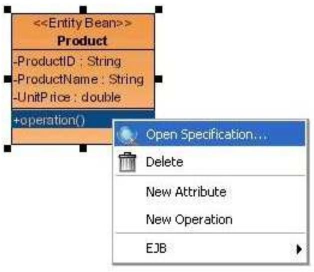


Figure 14.147 - To open operation Specification

7. An **Operation Specification** dialog box is displayed, click the **Parameters** Tab, click **Add...** button.

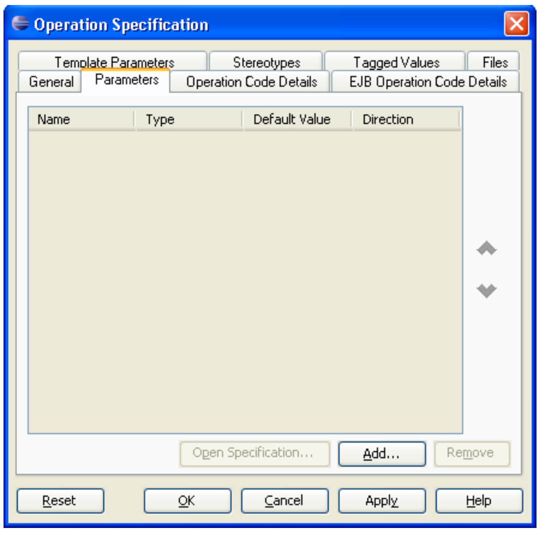
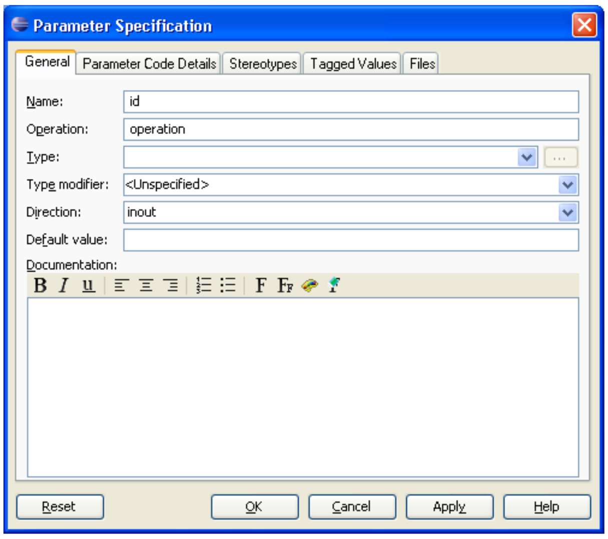


Figure 14.148 - Operation Specification dialog

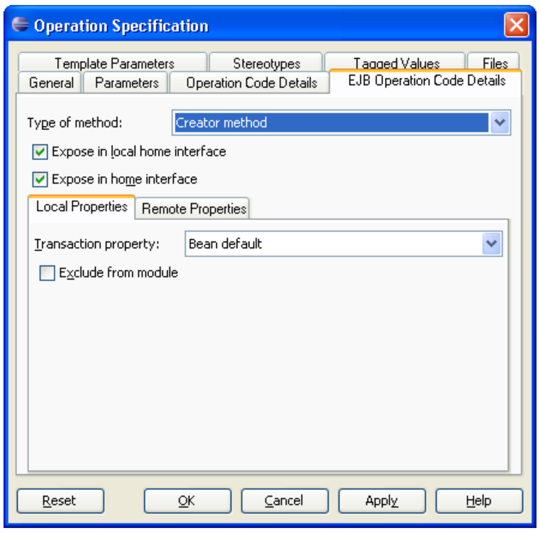
8. A **Parameter Specification** dialog box is displayed, you can edit the parameter name and properties such as type, click **OK**.



Name: id
Operation: operation
Type:
Type modifier: <Unspecified>
Direction: inout
Default value:
Documentation:

Figure 14.149 - Parameter Specification dialog

9. Click the **EJB Operation Code Details** Tab of the **Operation Specification** dialog box, select **Creator method** from the drop-down menu of **Type of Method**.



Type of method: Creator method
☑ Expose in local home interface
☑ Expose in home interface
Transaction property: Bean default
☐ Exclude from module

Figure 14.150 - Operation specification dialog (EJB Operation Code Details)

10. Click **OK**. The Creator Method is added to the Entity Bean.

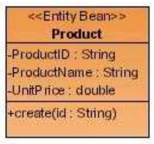


Figure 14.151 - Class with Creator methods

Creating a Finder Method

A Finder Method is used to specify a retrieval rule for the generated beans querying the database. You are allowed to define the finder method before the generation of Enterprise Java Bean. For more information on the usage of Finder Method, refer to the description of Using Finder Method in the Manipulating Persistent Data with Java chapter.

1. Right-Click on an Entity Bean, select **EJB > Create Finder Method**.

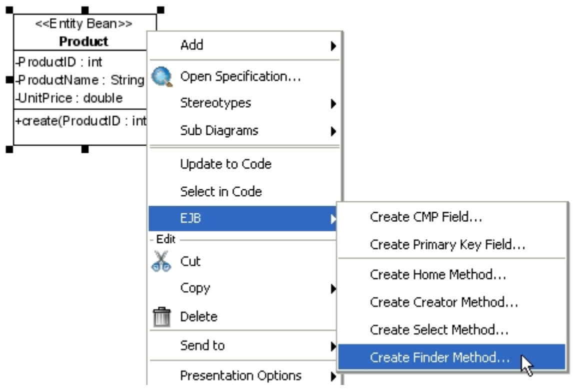
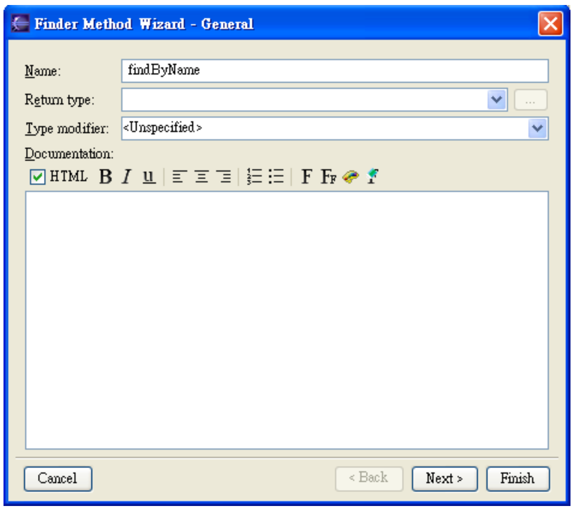


Figure 14.152 - To create finder method

2. The **Finder Method Wizard** will be shown, you can edit the general information for the Finder Method such as **Name**, **Return type** and **Type modifier**, and then click **Next >.**

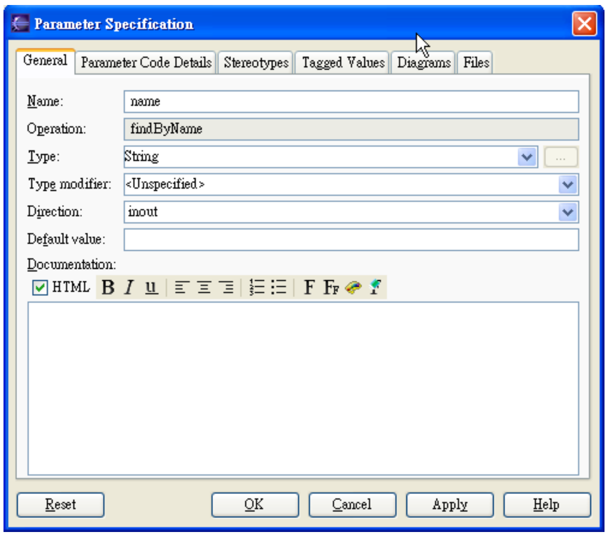


Name: findByName
Return type:
Type modifier: <Unspecified>

Figure 14.153 - Finder Method Wizard

You are directed to the **Finder Method Wizard - Parameters** screen.

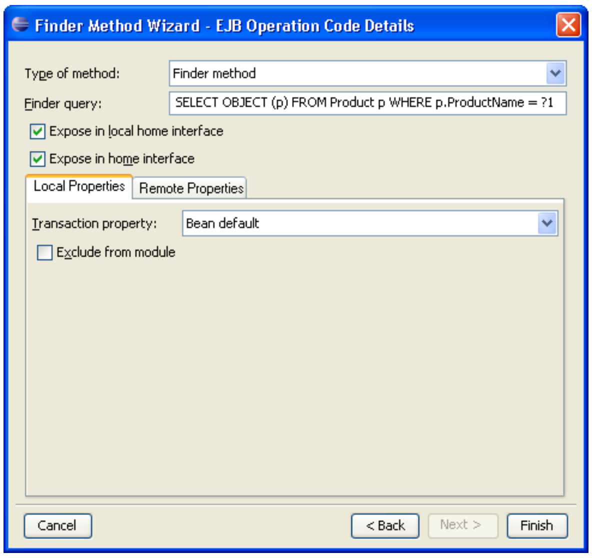
3. Click **Add...** button, a **Parameter Specification** dialog box is displayed, and you can modify the parameter name and properties, such as type.



Name: name
Operation: findByName
Type: String
Type modifier: <Unspecified>
Direction: inout
Default value:

Figure 14.154 - Parameter Specification dialog

4. Click **Next >,** the **EJB Operation Code Details** is shown which allows you to define the **Finder query** for the Finder Method.



Type of method: Finder method
Finder query: SELECT OBJECT (p) FROM Product p WHERE p.ProductName = ?1
[✓] Expose in local home interface
[✓] Expose in home interface
Transaction property: Bean default
[ ] Exclude from module

Figure 14.155 - EJB Operation Code Details

Note: As **Create Finder Method...** was selected from the pop-up menu of the entity bean, the Type of method of the **EJB Operation Code Details** of the wizard will be set as **Finder method** by default.

5. Click **Finish**, the **Finder Method** is added to the Entity Bean.

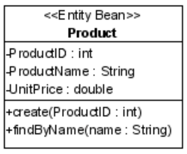


Figure 14.156 - Entity Bean Class with finder method

Mapping an Enterprise JavaBeans Model to a Data Model

Entity Bean to Database Mapping is supported which maps Enterprise JavaBeans models to entity relational models and vice versa.

Mapping between entity beans to relational database preserves the data, state, primary key, and difference in data type. Thus, you are not required to handle those tedious tasks during software development.

Mapping Entity Beans to Entities

Enterprise JavaBeans Model can be mapped to Data Model. Entity Bean classes can act as persistent data storage during the application is running, and hence, all entity bean classes can map to entities using a one-to-one mapping:

Example:

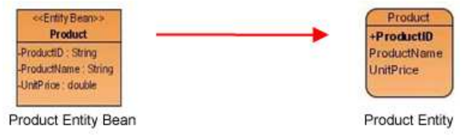


Figure 14.157 - Mapping Entity Beans to Entities

In the above example, the Product Entity Bean is mapped with the Product Entity as the Product Entity Bean instance can store the product information from the Product Entity.

Mapping Attributes to Columns

Since the Entity Bean map to the entities, attributes map to columns accordingly.

Example:

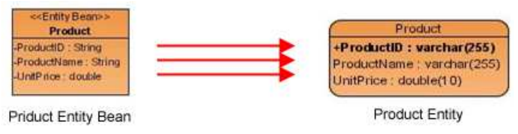


Figure 14.158 - Mapping attributes to Columns

In the above example, the following table shows the mapping between the attributes of the Customer Class and the columns of the Customer Entity.

Product Entity Bean	Product Entity
ProductID	ProductID
ProductName	ProductName
UnitPrice	UnitPrice

Table 14.13

Mapping Data Type

The attribute type automatically maps to an appropriate column data type of the database you desired.

Example:

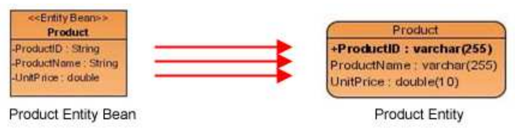


Figure 14.159 - Mapping Data type

In the above example, the following table shows the mapping between data types

Product Entity Bean	Product Entity
String	varchar(255)
Double	double(10)

Table 14.14

A table shows the data type mapping between Enterprise JavaBeans model and Data model.

Enterprise JavaBeans Model	Data Model
Boolean	Bit (1)
Byte	Tinyint (3)
Byte[]	Binary(1000)
Blob	Blob
Char	Char(1)
Character	Char(1)
String	varchar(255)
Int	Integer(10)
Integer	Integer(10)
Double	Double(10)
Decimal	Integer
Bigdecimal	Decimal(19)
Float	Float(10)
Long	Bigint(19)
Short	Smallint(5)
Date	Date
Time	Time(7)
Timestamp	Timestamp(7)

Table 14.15

Mapping Primary Key

The user-defined primary key of the Entity Bean maps to a primary key column of the corresponding entity.

Example:

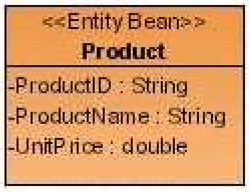


Figure 14.160 - Entity Bean

In the above example, when synchronizing the Product entity bean to entity relationship diagram, the attribute, ProductID which is defined as primary key by **Include in Primary Key** method maps to the primary key ProductID of the Product entity.

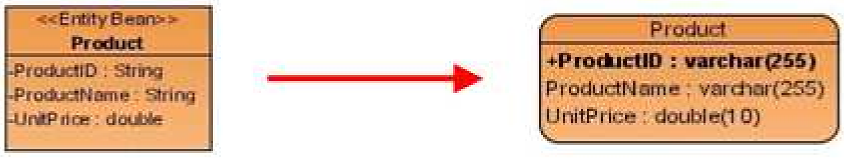


Figure 14.161 - Mapping Primary Key

Mapping Association

Association represents a binary relationship among Entity Beans. Each Entity Bean of an association has a role. A role name is attached at the end of an association line. The role name maps to a phrase of relationship in the data model.

Mapping Multiplicity

Multiplicity refers to the number of objects associated with a given object. There are six types of multiplicity commonly found in the association. The following table shows the syntax to express the Multiplicity.

Table shows the syntax expressing the Multiplicity

Type of Multiplicity	Description
0	Zero instance
0..1	Zero or one instances
0..*	Zero or more instances
1	Exactly one instance
1..*	One or more instances
*	Unlimited number of instances

Table 14.16

Example:

In the above example, it shows that a directory (role: host) contains zero or more files (role: accommodated by).

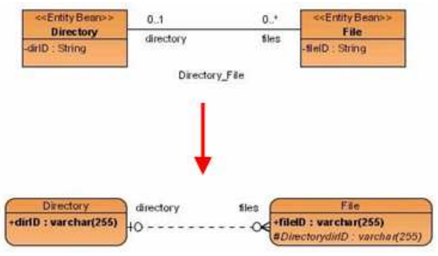
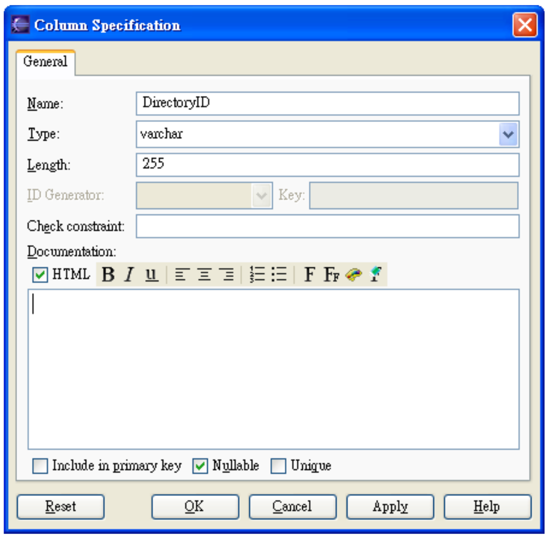


Figure 14.162 - Mapping Multiplicity

Transforming an Entity Bean with multiplicity of zero, the foreign key of parent entity can be nullable in the child entity. It is illustrated by the DirectoryID.


Column Specification

General

Name:	DirectoryID
Type:	varchar
Length:	255
ID Generator:	Key:
Check constraint:	
Documentation:	

[✓] HTML

[ ] Include in primary key [✓] Nullable [ ] Unique

Figure 14.163 - Column Specification dialog

Table shows the typical mapping between EJB Diagram and Entity Relationship Diagram.

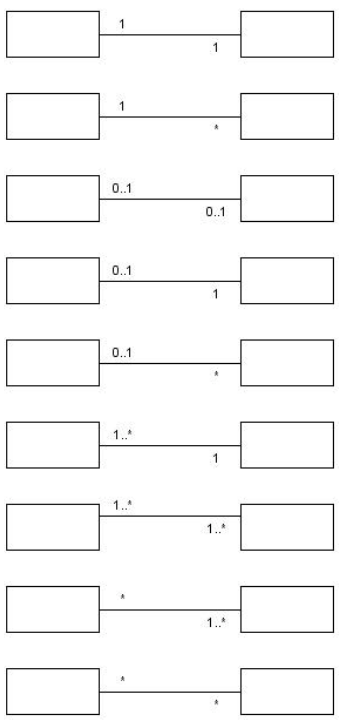
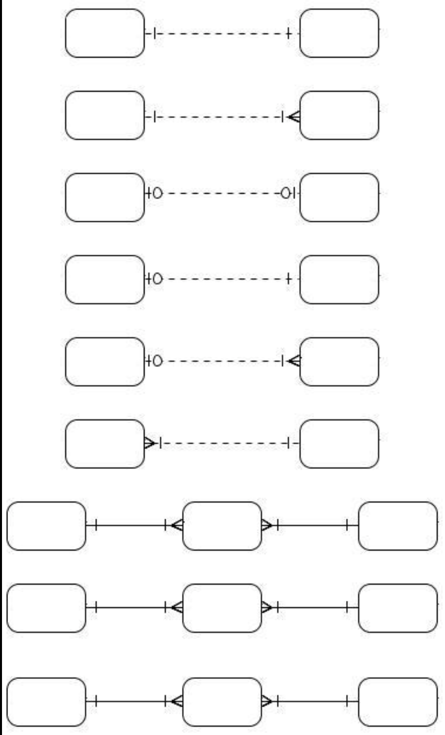
EJB Diagram	Entity Relationship Diagram
	

Table 14.17

Mapping Many-to-Many Association

For a many-to-many association between two Entity Beans, a Link Entity will be generated to form two one-to-many relationships in-between two generated entities. The primary keys of the two entities will migrate to the link entity as the primary/foreign keys.

Example:

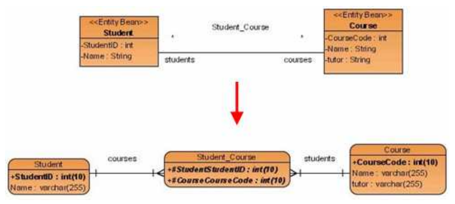


Figure 14.164 - Mapping Many-to-many association

In the above example, the link entity, Student_Course is generated between entities of Student and Course when transforming the many-to-many association.

Mapping Enterprise JavaBeans Model Terminology

Table shows the shift from Enterprise JavaBeans model to data model terminology.

Enterprise JavaBeans Model Term	Data Model Term
Entity Bean	Entity
Entity Bean Instance	Instance of an entity
Association	Relationship
Attribute	Column
Role	Phrase
Multiplicity	Cardinality

Table 14.18

15 Data Model

Chapter 15 - Data Model

You are provided with a visual modeling environment for the data model of an application, and also reverse database engineering. This chapter shows you how to depict the object models by using Entity Relationship Diagram and how to reverse database, and describes how the data model maps to the object model.

In this chapter:

- Introduction
- Creating Data Model
- Reverse Database Engineering
- Creating Array Table in Data Model
- Creating Partial Table in Data Model
- Copying SQL Statements
- Mapping Data Model to Object Model
- Mapping Data Model to Enterprise JavaBeans Model

Introduction

An entity is an object in the business or system with well-defined characteristics which are represented by columns showing what information can be stored. In relational databases, an entity refers to a record structure, i.e. table.

A data model provides the lower-level detail of a relational database of an application. It shows the physical database models and their relationships in an application. An entity relationship diagram can be used to describe the entities inside a system and their relationships with each other; the entity relationship diagram is also known as a data model.

Visual modeling for data models is supported, not only by creating a new data model, but also by transforming from either an object model, or an Enterprise JavaBeans model. Reversing the existing relational model into data models is also supported. As object-relational mapping is automated, object model can thus be generated from the data model by reversing the existing database.

The following section describes how to reverse the relational model into data models and depict the data models using the Entity Relationship Diagram.

Entity Relationship Diagram

Entity relationship diagram is a graphical representation of a data model of an application. It acts as the basis for mapping the application to the relational database.

Creating an Entity Relationship Diagram

You are provided with four ways to create an Entity Relationship Diagram.

1. Drawing an Entity Relationship Diagram(ERD)
2. Reverse Engineering an existing Relational Database
3. Synchronizing from the Object Model to Data Model
4. Synchronizing from the Enterprise JavaBeans Model to Data Model

Drawing an Entity Relationship Diagram (ERD)

1. You can create a new ERD in one of the three ways:
   - On the menu, click **File > New Diagram >New Entity Relationship Diagram**.

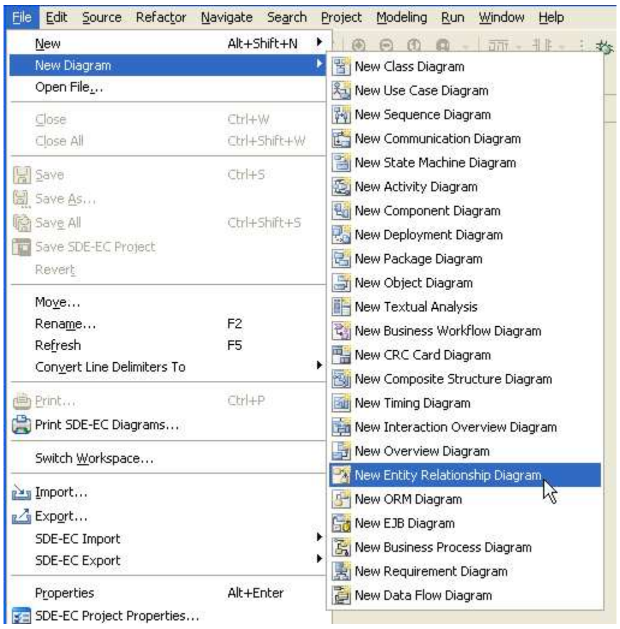


Figure 15.1 - Create new Entity Relationship Diagram

   - On the **Diagram Navigator**, right-click **Entity Relationship Diagram > Create Entity Relationship Diagram**.

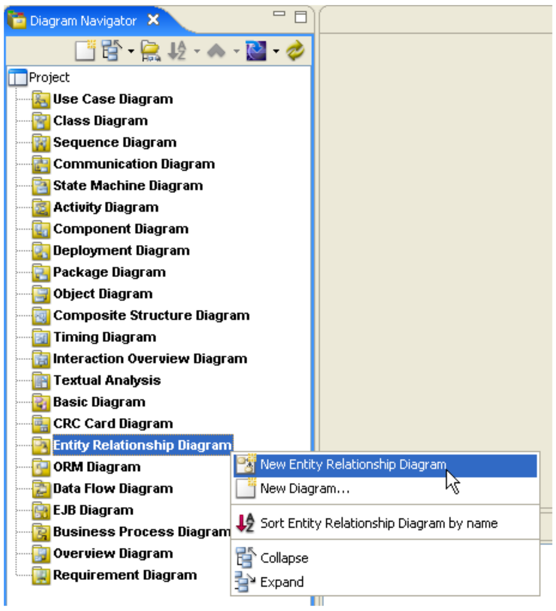


Figure 15.2 - Create ERD by click on Diagram Navigator

   - On the toolbar, click the **New Entity Relationship Diagram** icon.

A new Entity Relationship Diagram pane is displayed.

Creating a new Entity element to the ERD

1. On the diagram toolbar, click the **Entity** shape icon.

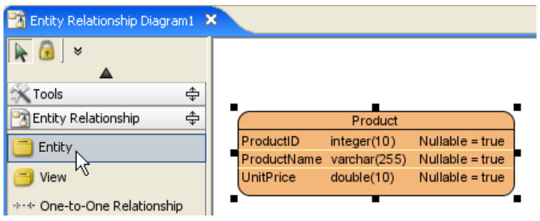

Entity Relationship Diagram1
Tools
Entity Relationship
Entity
View
One-to-One Relationship
Product
ProductID integer(10) Nullable = true
ProductName varchar(255) Nullable = true
UnitPrice double(10) Nullable = true

Figure 15.3 - Entity resource-centric.

2. Click a location in the diagram pane.

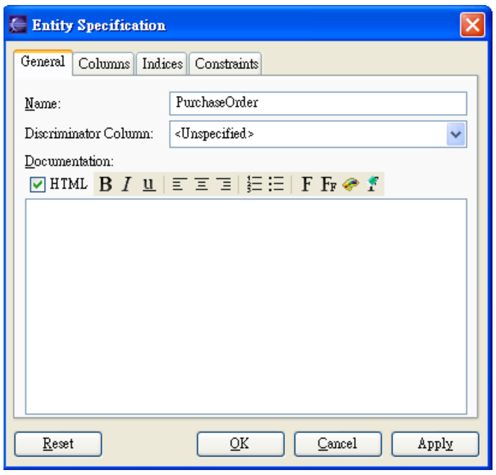
   An icon representing the entity element is added to the diagram.

3. Type in a name for the **Entity** element.
   - You can edit the name by double-clicking the name or by pressing the ***F2*** button.

Modifying the Entity Specification

1. To display the **Entity Specification**, right-click the entity element, click **Open Specification**.

   Entity Specification dialog box is displayed, you can modify the entity properties and constraints.



Name: PurchaseOrder
Discriminator Column: <Unspecified>
Documentation:

Figure 15.4 - Entity Specification dialog

Adding new Column to the Entity

1. You can add a new column to the entity in one of the three ways:
   - Right-click on an entity, select **New Column**.

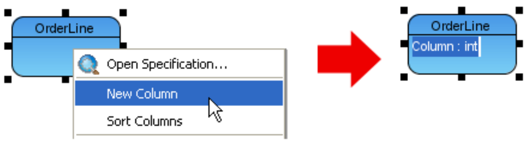


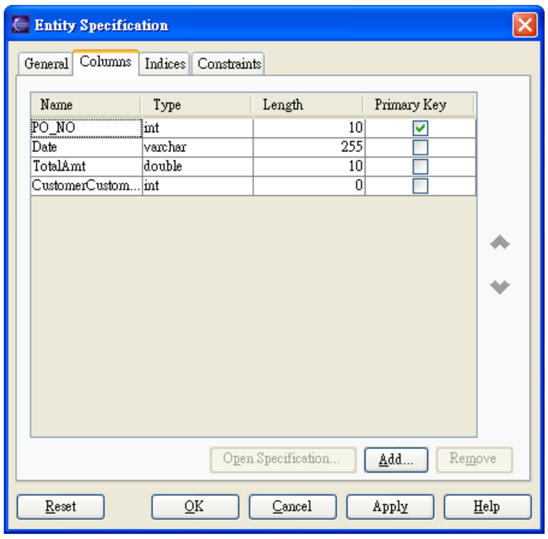
Figure 15.5 - Add new column

A new column is added, type the column name and type in the form of "**column_name: type**". You can also edit the column name by double-clicking the column name or by pressing the ***F2*** button.

- Click on an entity, press the keyboard shortcut -***Alt + Shift + C***.
- 1. Right-click the entity element, click **Open Specification**.

  2. Click the **Columns** Tab, then click **Add**.

  Column Specification dialog box is displayed, you can modify the column name and properties, such as type.



Name	Type	Length	Primary Key
PO_NO	int	10	✓
Date	varchar	255	☐
TotalAmt	double	10	☐
CustomerCustom...	int	0	☐

Figure 15.6 - Entity Specification dialog

Adding Relationship to the Entities

1. You can add relationship to the entities in one of the two ways:
   - Using Resource-Centric Interface
     1. Click on an entity, a group of valid editing resources are displayed around the entity.
     2. Drag the resource of "**One-to-One Relationship**" to the associated class.

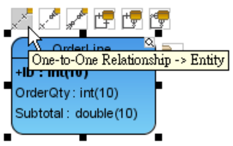


Figure 15.7 - One-to-one Relationship resource-centric

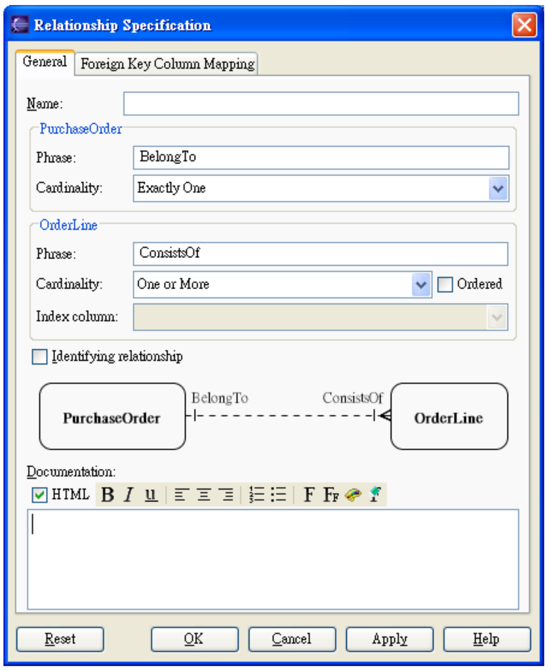
   - Using Toolbar icon
     1. On the diagram toolbar, click the **Relationship** icon.
        - **One-to-One Relationship**
        - **One-to-Many Relationship**
        - **Many-to-Many Relationship**
     2. Click on an entity, drag to another entity.

A line indicating a connection between the two entities is shown.

Editing Relationship Specification

1. You can edit the relationship specification in the following way:
   - Right-click on the connection line, click **Open Specification** from the pop-up menu.

   Relationship Specification dialog box is displayed, you have to modify the relationship properties, **Phrase** and **Cardinality**.



Relationship Specification

General | Foreign Key Column Mapping

Name:

PurchaseOrder
Phrase: BelongTo
Cardinality: Exactly One

OrderLine
Phrase: ConsistsOf
Cardinality: One or More ☐ Ordered
Index column:

☐ Identifying relationship

Documentation:
☑ HTML

Reset | OK | Cancel | Apply | Help

Figure 15.8 - Relationship Specification

Once you assign a primary key to a column of an entity, a foreign key column is automatically added to all entities associated with it.

An ERD is created.

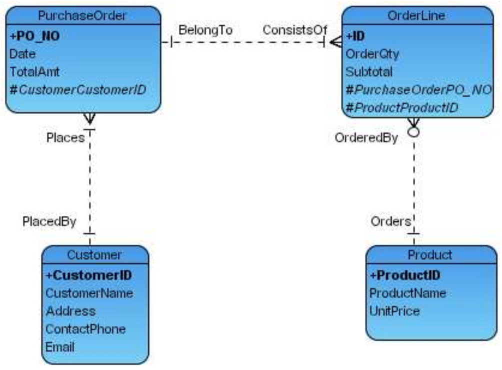


Figure 15.9 - Entity Relationship Diagram

Reverse Engineering an existing Relational Database

1. You can create an Entity Relationship Diagram by reverse engineering an existing relational database.
   - On the menu, click **Modeling** > **ORM** > **Reverse Database...**.

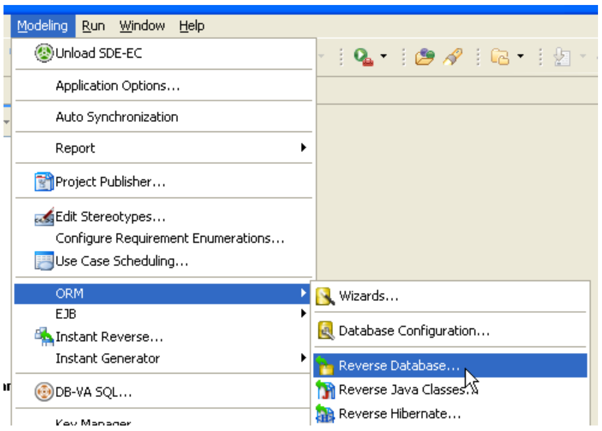


Figure 15.10 - Reverse databases

For other SDE:

SDE	Method
SDE for JBuilder	On the menu, click **Tools** > **Modeling** > **ORM** > **Reverse Database...**.
SDE for NetBeans	On the menu, click **Modeling** > **ORM** > **Reverse Database...**.
SDE for IntelliJ IDEA	On the menu, click **Modeling** > **ORM** > **Reverse Database...**.
SDE for JDeveloper	On the menu, click **Model** > **ORM** > **Reverse Database...**.
SDE for WebLogic Workshop	On the menu, click **Modeling** > **ORM** > **Reverse Database...**.

Table 15.1

The **Database to Data Model** dialog box is displayed.

Step 1: Select Language

Select the language of the project from the drop-down menu, either **Java** or **C#,** and then click **Next >** to proceed to Step 2.

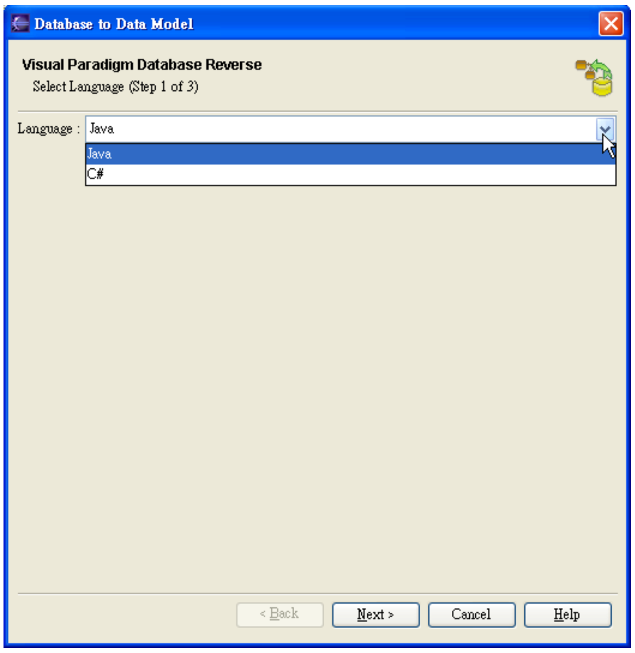
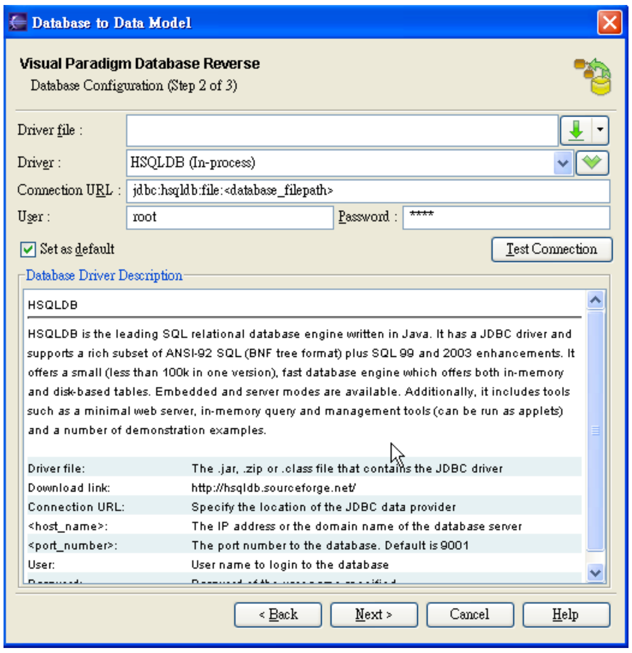

Database to Data Model
Visual Paradigm Database Reverse
Select Language (Step 1 of 3)
Language : Java

Figure 15.11 - Select the programming language

Step 2: Database Configuration

You can configure the database connection for the desired database to be reversed.

1. You are asked to define the database configuration. Refer to the descriptions in the Database Configuration chapter for information on how to configure the database.


Database to Data Model
Visual Paradigm Database Reverse
Database Configuration (Step 2 of 3)

Driver file :	
Driver :	HSQLDB (In-process)
Connection URL :	jdbc:hsqldb:file:<database_filepath>
User :	root
Password :	****

✓ Set as default

Test Connection
Database Driver Description

HSQLDB

HSQLDB is the leading SQL relational database engine written in Java. It has a JDBC driver and supports a rich subset of ANSI-92 SQL (BNF tree format) plus SQL 99 and 2003 enhancements. It offers a small (less than 100k in one version), fast database engine which offers both in-memory and disk-based tables. Embedded and server modes are available. Additionally, it includes tools such as a minimal web server, in-memory query and management tools (can be run as applets) and a number of demonstration examples.

Driver file:	The .jar, .zip or .class file that contains the JDBC driver
Download link:	http://hsqldb.sourceforge.net/
Connection URL:	Specify the location of the JDBC data provider
<host_name>:	The IP address or the domain name of the database server
<port_number>:	The port number to the database. Default is 9001
User:	User name to login to the database

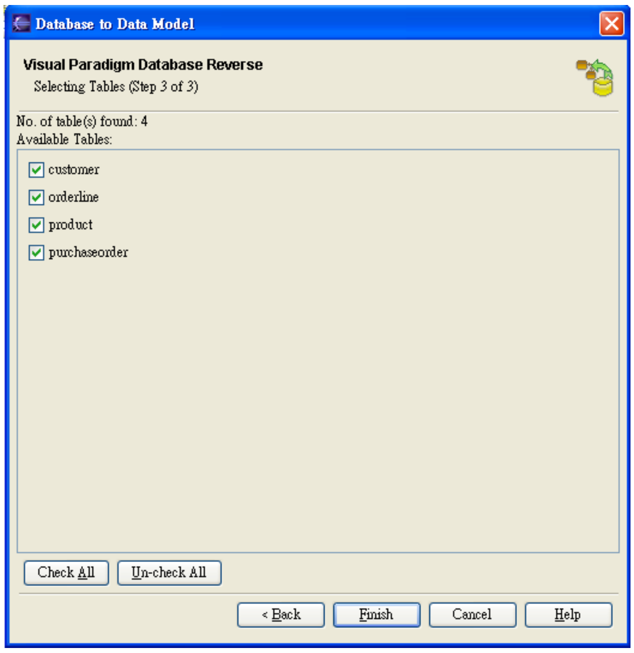
Figure 15.12 - Database Configuration

2. Click **Next>,** go to Step 3 of Reverse Database.

Step 3: Selecting Tables

All the available tables found from the connected database are listed.

1. Select the tables that you want to reverse to Data Model.
2. Click **Finish**.


Database to Data Model
Visual Paradigm Database Reverse
Selecting Tables (Step 3 of 3)

No. of table(s) found: 4
Available Tables:

- ✓ customer
- ✓ orderline
- ✓ product
- ✓ purchaseorder

Figure 15.13 - Select the tables for reverse

An Entity Relationship Diagram is automatically generated and displayed. It can be found under the **Diagram Navigator**.

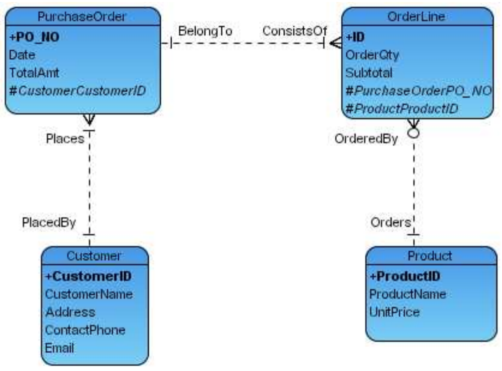


Figure 15.14 - The reversed Entity Relationship Diagram

Synchronizing from an Object Model to a Data Model

You can generate the ERD from a class diagram by synchronization if there is a class diagram.

1. You can synchronize the Class Diagram to ERD in one of the three methods:
   - On the menu, click **Modeling> ORM > Sync to Entity Relationship Diagram**.

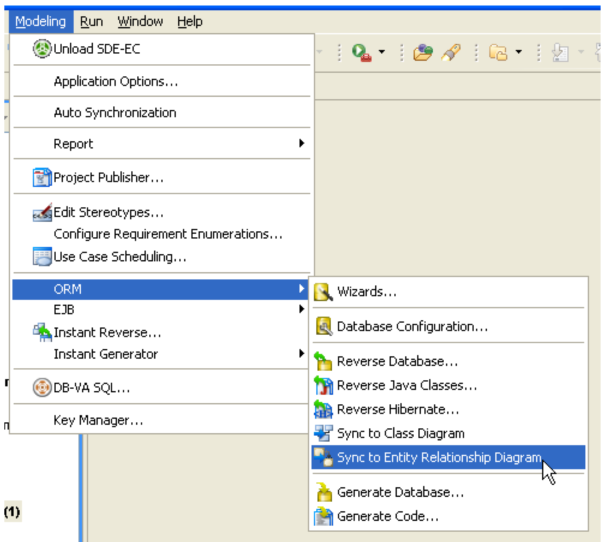


Figure 15.15 - Synchronize ORM to ERD

For other SDE:

SDE	Method
SDE for JBuilder	On the menu, click **Tools > Modeling > ORM > Sync to Entity Relationship Diagram**.
SDE for NetBeans	On the menu, click **Modeling > ORM > Sync to Entity Relationship Diagram**.
SDE for IntelliJ IDEA	On the menu, click **Modeling > ORM > Sync to Entity Relationship Diagram**.
SDE for JDeveloper	On the menu, click **Model > ORM > Sync to Entity Relationship Diagram**.
SDE for WebLogic Workshop	On the menu, click **Modeling > ORM > Sync to Entity Relationship Diagram**.

Table 16.2

- Right-Click on the Class Diagram, select **Synchronize to Entity Relationship Diagram**.

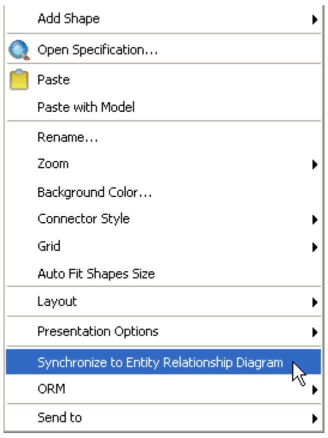


Figure 15.16 - Synchronize to ERD by click on popup menu

- On the class diagram, hold down the right-mouse button, move the mouse from left to right to form the gesture. A blue path is shown indicating the gesture.

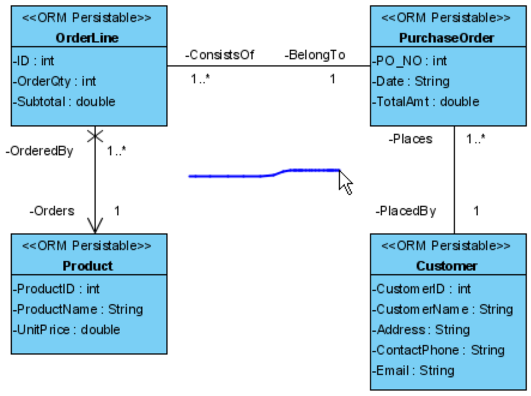


Figure 15.17 - Synchronize to ERD by using gesture

2. An Entity Relationship Diagram is generated and can be found under the **Diagram Navigator**.

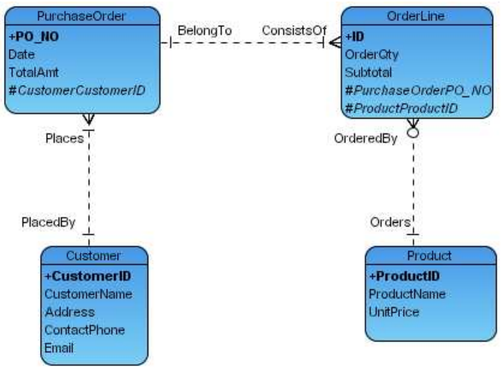


Figure 15.18 - The synchronized ERD

Synchronizing from the Enterprise JavaBeans Model to Data Model

You can generate the ERD from an EJB diagram by synchronization if there is an EJB diagram.

1. You can synchronize the EJB Diagram to ERD in one of the three methods:
   - On the menu, click **Modeling** > **EJB** > **Sync to Entity Relationship Diagram.**

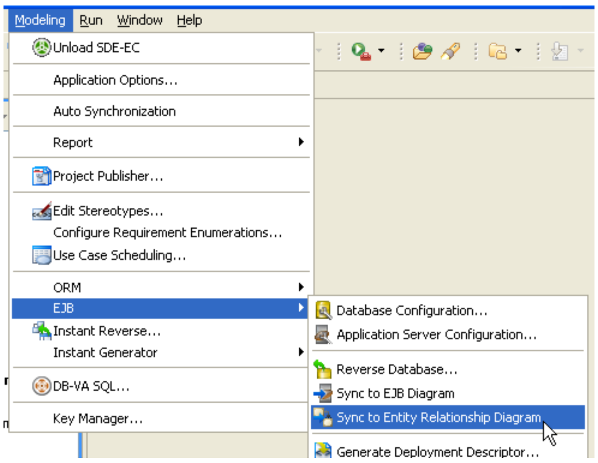


Figure 15.19 - synchronize EJB to ERD

For other SDE:

SDE	Method
SDE for JBuilder	On the menu, click **Tools** > **Modeling** > **EJB** > **Sync to Entity Relationship Diagram**.
SDE for NetBeans	On the menu, click **Modeling** > **EJB** > **Sync to Entity Relationship Diagram**.
SDE for IntelliJ IDEA	On the menu, click **Modeling** > **EJB** > **Sync to Entity Relationship Diagram**.
SDE for JDeveloper	On the menu, click **Model** > **EJB** > **Sync to Entity Relationship Diagram**.
SDE for WebLogic Workshop	On the menu, click **Modeling** > **EJB** > **Sync to Entity Relationship Diagram**.

Table 16.3

   - Right-Click on the EJB Diagram, select **Synchronize to Entity Relationship Diagram**.

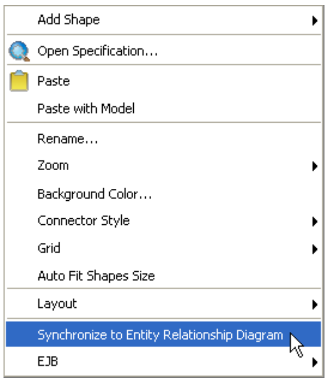


Figure 15.20 - Synchronize to ERD by click on popup menu

- On the EJB diagram, hold down the right-mouse button, move the mouse from left to right to form the gesture. A blue path is shown indicating the gesture.

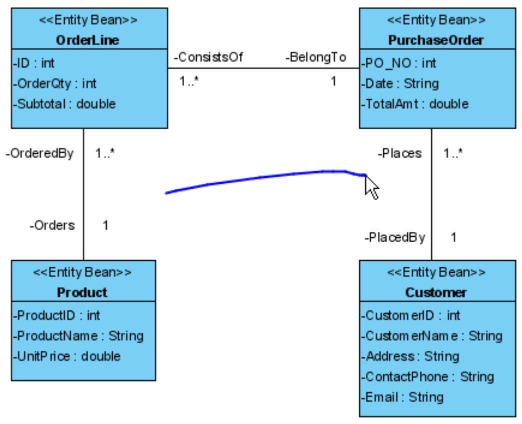


Figure 15.21 - synchronize to EJB by using gesture

An Entity Relationship Diagram is generated and can be found under the **Diagram Navigator**.

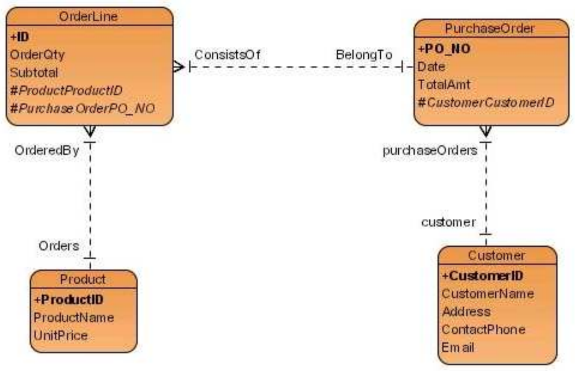


Figure 15.22 - The generated Entity Relationship Diagram

Specifying Index Column

If a relationship with cardinality of many at one end, a corresponding collection class will be used for handling its multiple cardinality. You can specify an index column to sort the collection.

1. Right-click on the connection line, click **Open Specification** from the pop-up menu.

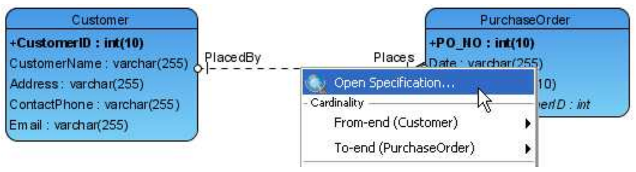
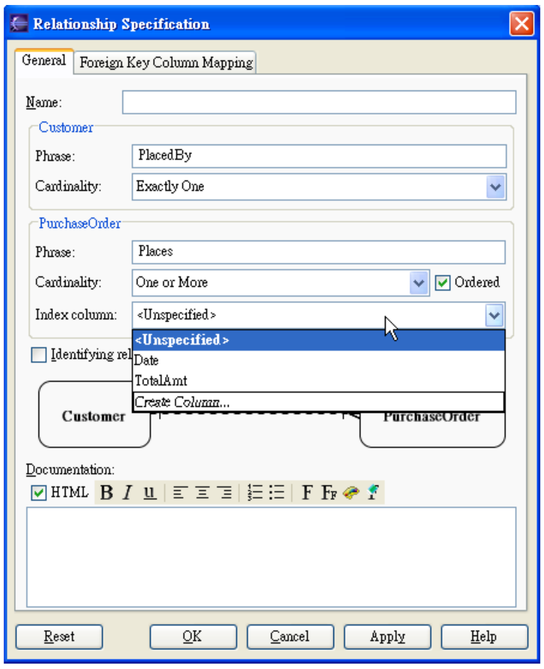


Figure 15.23 - To open association specification

Relationship Specification dialog box is displayed.

2. Check the option for **Ordered**.

3. Select the index column from the drop-down menu of **Index column**, click **OK**.



Name:

Customer
Phrase: PlacedBy
Cardinality: Exactly One

PurchaseOrder
Phrase: Places
Cardinality: One or More ☑ Ordered
Index column: <Unspecified>

<Unspecified>
Date
TotalAmt
Create Column...

☐ Identifying rel

Figure 15.24 - Relationship Specification dialog

You can select **Create Column** from the drop-down menu to create a new index column for sorting.

Using the ID Generator

As the primary key is unique, the generation of primary key is supported. The ID Generator is specialized for generating a primary key value at runtime.

1. Right-click on the primary key of an entity, select **Open Specification** from the pop-up menu.

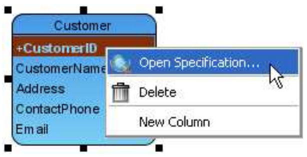
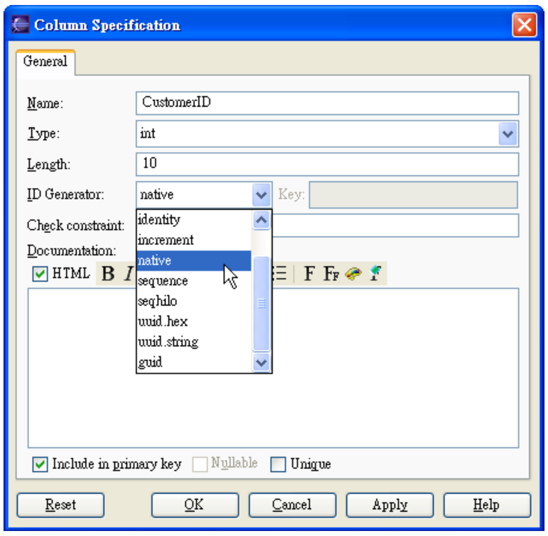


Figure 15.25 - To open column specification

Column Specification of the primary key is displayed.

2. Select the ID generator from the drop-down menu of **ID Generator**, click **OK** to confirm setting.



Column Specification

General

Name: CustomerID

Type: int

Length: 10

ID Generator: native

Figure 15.26 - Column Specification dialog

If the ID Generator is specified as either sequence, seqhilo or hilo, you have to enter the key for the sequence/table name.

Defining Discriminator

In generalization, the superclass distributes its commonalities to a group of similar subclasses. The subclass inherits all superclass's attributes and it may contain specific attributes. The entities within the hierarchy are combined into one single entity containing all the attributes and a discriminator column. The discriminator contains a unique value which is used for identifying the entity which hierarchy it belongs to.

You are allowed to define the discriminator in the entity and discriminator value in the classes.

Defining Discriminator Column for Entity

You can add a new column acting as the discriminator column for an entity.

1. Right-click on an entity, select **New Column**.

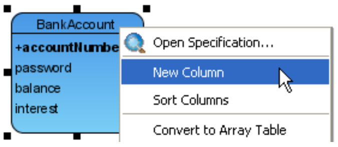
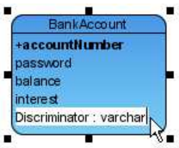


Figure 15.27 - add new column

2. Enter the name and type for the discriminator in the form of "**discriminator_name: type**".



Discriminator : varchar

Figure 15.28 - Enter the name and type of column

3. Right-click on the entity, select **Open Specification...**.

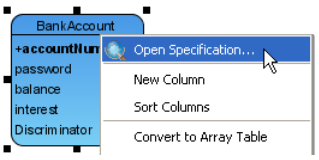


Figure 15.28 - To open entity specification

Entity Specification dialog box is displayed.

4. Select the desired column from the drop-down menu of **Discriminator Column**, click **OK** to confirm setting.

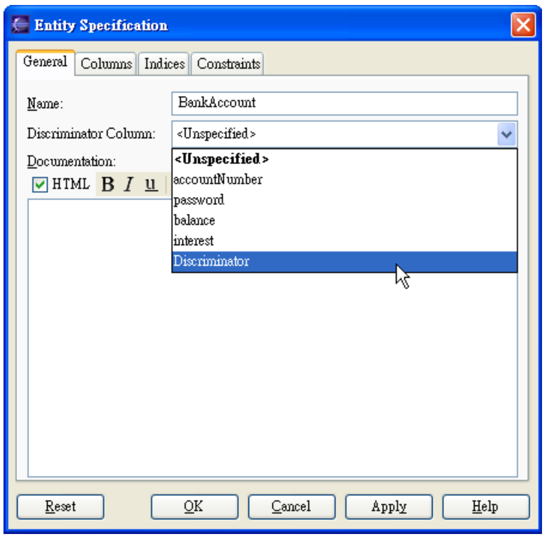


Figure 15.29 - Select the discriminator Column

Defining Discriminator Value for Class

You can specify the discriminator value for each sub-class.

1. Right-click on the relative sub-class for adding discriminator, select **ORM** > **ORM Class Details...**from the pop-up menu. The **Class Specification** dialog box showing the **ORM Class Detail** tab is displayed.

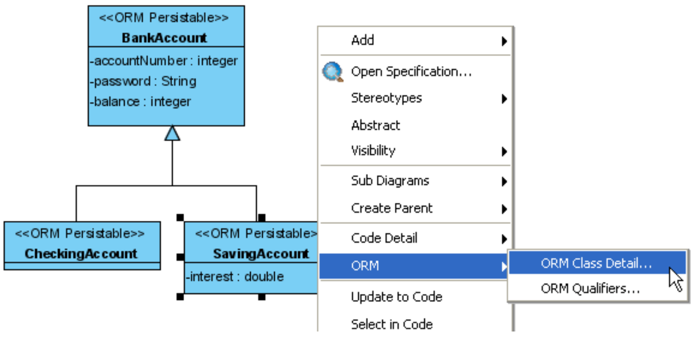
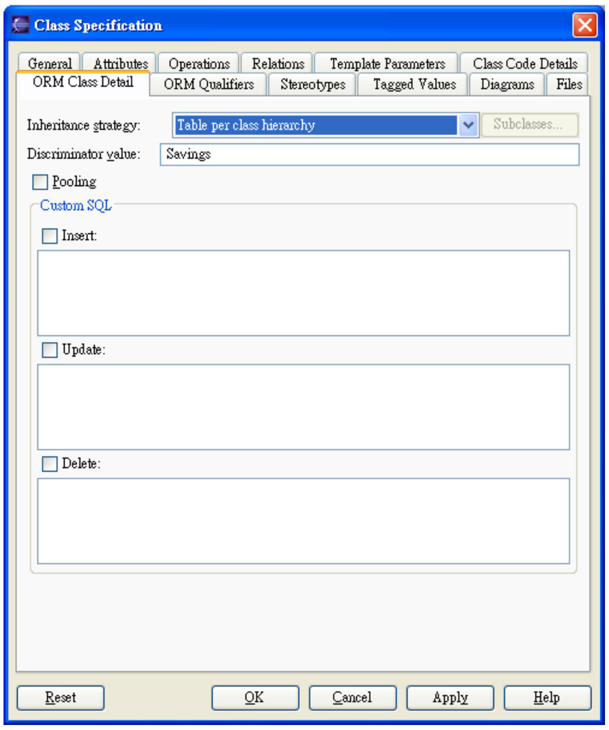


Figure 15.30 - To open ORM Class Details

2. Enter the discriminator value for identifying the sub-class.



Inheritance strategy: Table per class hierarchy
Discriminator value: Savings
☐ Pooling
Custom SQL
☐ Insert:
☐ Update:
☐ Delete:

Figure 15.31 - Class Specification(ORM Class Detail)

Creating an Array Table

In a one-to-many relationship, a collection is used for handling the multiple objects such that it is simpler to retrieve each object from the collection one by one.

The idea of Array Table is promoted which allows users to retrieve objects in the form of primitive array, instead of a collection when handling a data column with cardinality of many.

You are allowed to create an array table in the entity and define an array type in the classes.

Defining an Array Table

You can create an Array Table for the Entity with a column containing more than one instance of data.

1. Create a one-to-many relationship between the entity and one of its columns that may contain more than one instance of data.

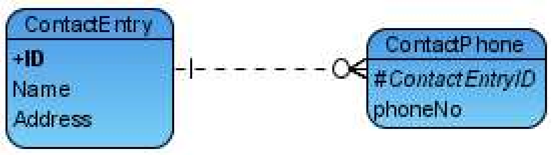


Figure 15.32 - Defining a array table

In the above case, the phonebook has a contact entry for each contact person. Each contact person may have more than one phone numbers. A one-to-many relationship between contact entry and contact phone can be built.

2. Right-click on the entity for the data column with cardinality of many, select **Convert to Array Table** from the pop-up menu.

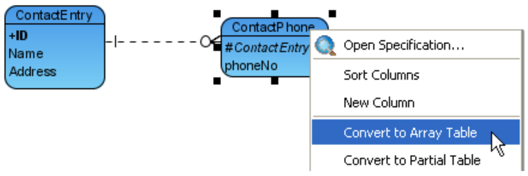
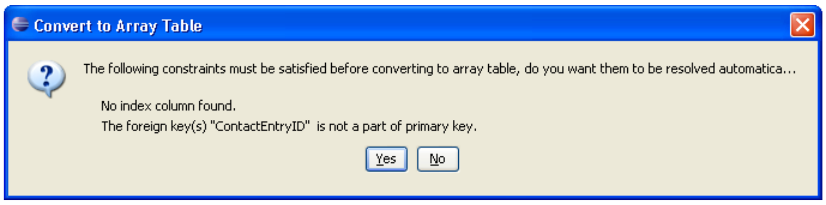


Figure 15.33 - Convert table to array table

3. A warning message will be displayed, showing that the listed constraints are not satisfied for converting to array table. Click **Yes** to let SDE to resolve the constraints automatically. Click **No** to cancel the conversion to array table.


Convert to Array Table

The following constraints must be satisfied before converting to array table, do you want them to be resolved automatica...

No index column found.

The foreign key(s) "ContactEntryID" is not a part of primary key.

Yes No

Figure 15.34 - Confirm convert to Array Table

The conversion to Array Table is completed and the entity for the data column is stereotyped as Array Table.

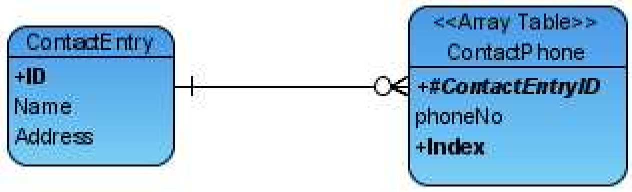


Figure 15.35 - The diagram is converted to array table

Defining an Array Type for Attribute in Class

A class with an attribute of array type modifier means that the attribute may contain more than one data; thus it implies the idea of Array Table.

You can define the array type for the attribute in one of the two ways:

- Using Inline Editing
  1. Right-click on a class, click **Add** > **Attribute**.

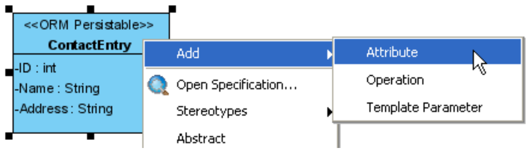


Figure 15.36 - Add an attribute

  2. Enter the name and type for the attribute in the form of "**attribute_name :type[]**", the sign, **"[ ]"** indicates the attribute is an array.

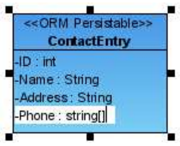


Figure 15.37 - Enter the type of attribute as array

- Using **Class Specification** Dialog Box
    1. Right-click on a class, click **Open Specification**.

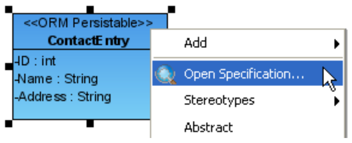


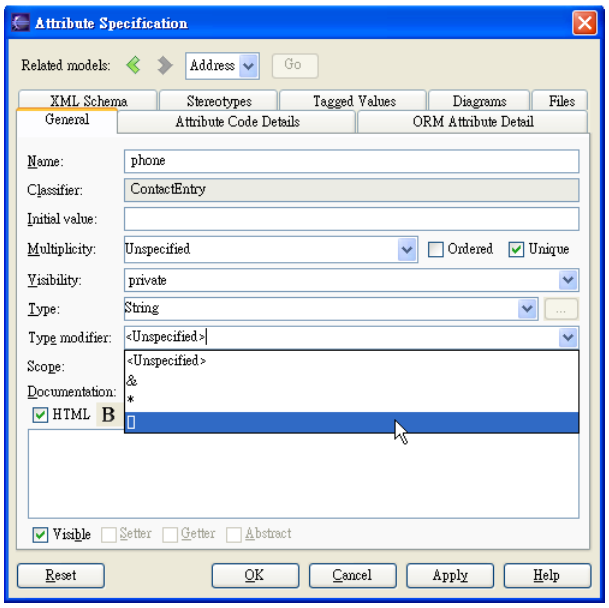
Figure 15.38 - To open the attribute Specification

- The **Class Specification** dialog box is displayed

1. Click **Attribute** Tab, click **Add**.

   Attribute Specification is displayed.

2. Enter attribute name and type, select [] from the drop-down menu of **Type modifier**, then click **OK** to confirm setting.



Name: phone
Classifier: ContactEntry
Initial value:
Multiplicity: Unspecified ☐ Ordered ☑ Unique
Visibility: private
Type: String
Type modifier: <Unspecified>
<Unspecified>
&
*
[]
☑ Visible

Figure 15.39 - Attribute Specification dialog

Creating a Partial Table

In a one-to-one identifying relationship, an entity may be a subordinate of the related entity; that is, the subordinate entity has columns which also belong to its superior entity in the real world situation.

The idea of Split Table with stereotype of Partial is also promoted, which allows developers to optimize the size of database, and minimizes the redundant persistent classes for handling one-to-one identifying relationship. In order to reduce the risk of appending a new column to an existing database table, Split table supports developers to add new columns to the partial table with a one-to-one identifying relationship linked to the existing table.

You are allowed to split the entity into two and convert the subordinate entity to be a Partial Table in a one-to-one identifying relationship.

Splitting Table

You can split an entity into two associated with a one-to-one identifying relationship.

1. You can activate the **Split Table** dialog box in one of the two ways:
   - Using Pop-up Menu
     1. Right-click an entity, select **Split Table**.

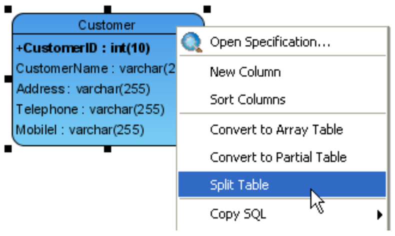


Figure 15.40 - To split Table

   - Using Resource-Centric Interface
     1. Click on an entity, a group of valid editing resources are displayed around the entity.
     2. Click the resource of "**One-to-One Relationship -> Partial Table**".

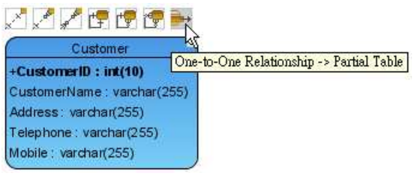


*Figure 15.41 - **One-to-one Relationship** -> **Partial Table** resource-centric*

Split Table dialog box is displayed.

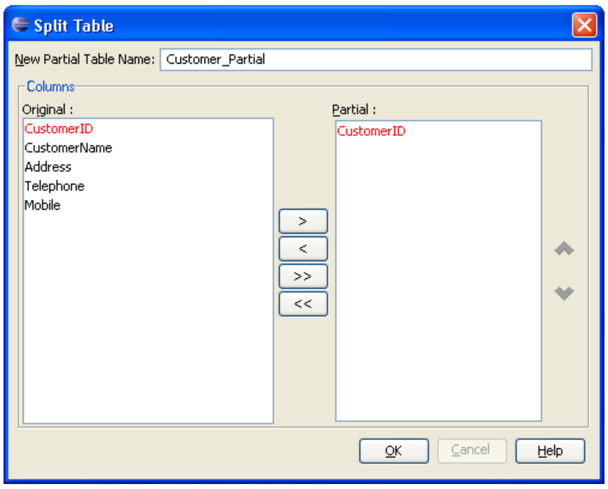


Figure 15.42 - Split Table dialog

2. Edit the **New Partial Table Name,** select the columns from the list of **Original** to **Partial**, and click **OK**.

   An entity stereotyped as Partial is created.

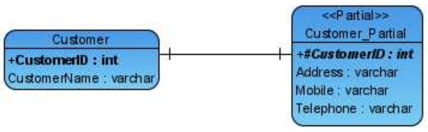


Figure 15.43 - Partial Table created

Converting to a Partial Table

You can convert an entity to a Partial Table in a one-to-one identifying relationship.

1. Right-click on the entity, select **Convert to Partial Table** from the pop-up menu.

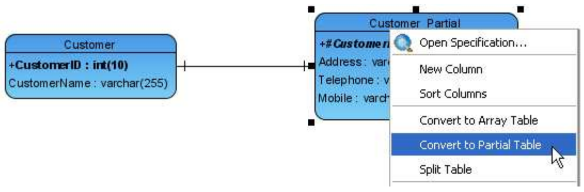


Figure 15.44 - Convert to partial table

The entity is stereotyped as Partial.

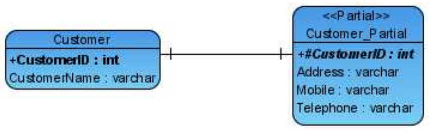


Figure 15.45 - Modified to Partial Table

Copying SQL statements from Tables

Copying SQL statements from the ERD entities is provided. It allows the developers to copy the SQL statements from the entity relationship diagram easily such that developers can use and modify the SQL statement on the database directly.

In order to copy the SQL statement, you must configure the database setting in advance as the SQL statements will be generated according to the default database server type.

1. To configure database connection, on the menu, click **Modeling > ORM > Database Configuration...**or **Modeling > EJB > Database Configuration...**.

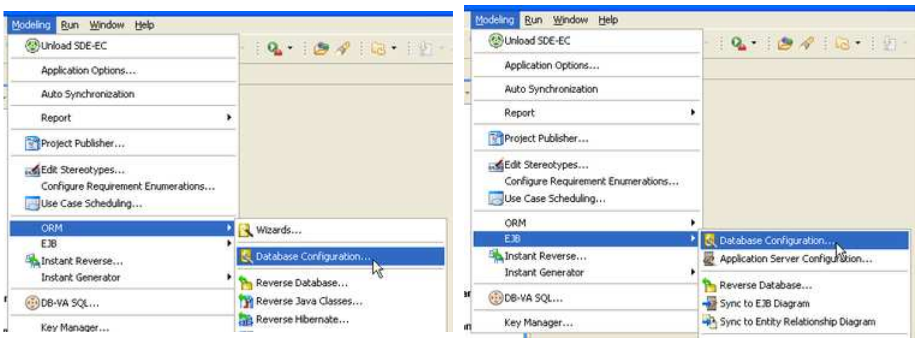


Figure 15.46 - To setup the Database Configuration

For other SDE:

SDE	Method
SDE for JBuilder	On the menu, click **Tools** > **Modeling** > **ORM** > **Database Configuration...**. On the menu, click **Tools** > **Modeling** > **EJB** > **Database Configuration...**.
SDE for NetBeans	On the menu, click **Modeling** > **ORM** > **Database Configuration...**. On the menu, click **Modeling** > **EJB** > **Database Configuration...**.
SDE for IntelliJ IDEA	On the menu, click **Modeling** > **ORM** > **Database Configuration...**. On the menu, click **Modeling** > **EJB** > **Database Configuration...**.
SDE for JDeveloper	On the menu, click **Model** > **ORM** > **Database Configuration...**. On the menu, click **Model** > **EJB** > **Database Configuration...**.
SDE for WebLogic Workshop	On the menu, click **Modeling** > **ORM** > **Database Configuration...**. On the menu, click **Modeling** > **EJB** > **Database Configuration...**.

Table 15.4

Database Configuration dialog box will be displayed. Refer to the descriptions in the *Database Configuration* chapter for information on how to configure the database.

Note	SDE will only provide you the function of copying SQL if the default database connection is set.
Note	If there are multiple database settings, the SQL statements will be generated for all these database servers.

Example:

There are two database settings selected in the working environment.

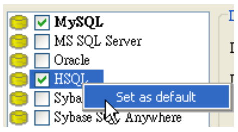


Figure 15.47 - Set the database type as default

1. Right-click on the ERD, select **Copy SQL** > **Detail...**from the pop-up menu.

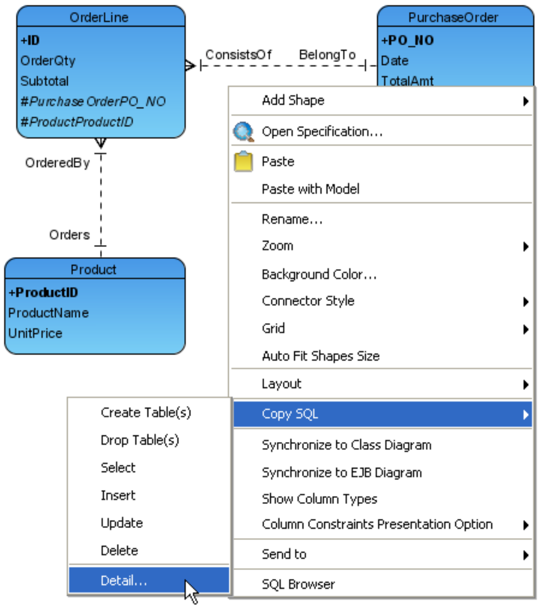
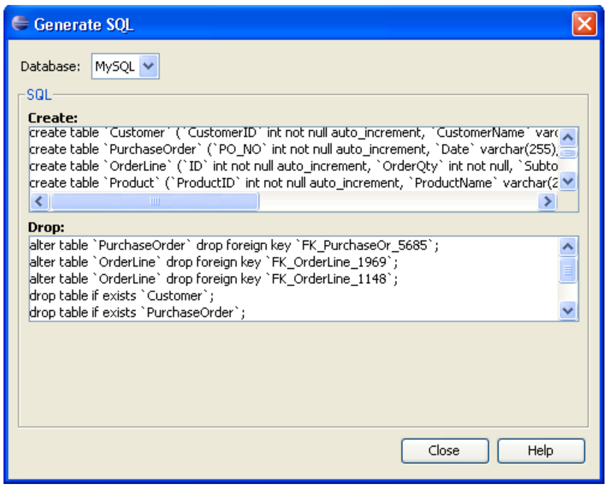


Figure 15.48 - To open the Generate SQL dialog

Note	You can select **Create Table(s), Drop Table(s), Select**, **Insert**, **Update** and **Delete** from the **Copy SQL** submenu to directly copy the SQL statements to clipboard.

2. **Generate SQL** dialog box is displayed, select the database server from the drop-down menu of **Database**, the corresponding SQL statements will be displayed accordingly.



Database: MySQL

SQL

Create:

```
create table `Customer` (`CustomerID` int not null auto_increment, `CustomerName` varc
create table `PurchaseOrder` (`PO_NO` int not null auto_increment, `Date` varchar(255),
create table `OrderLine` (`ID` int not null auto_increment, `OrderQty` int not null, `Subto
create table `Product` (`ProductID` int not null auto_increment, `ProductName` varchar(2
```

Drop:

```
alter table `PurchaseOrder` drop foreign key `FK_PurchaseOr_5685`;
alter table `OrderLine` drop foreign key `FK_OrderLine_1969`;
alter table `OrderLine` drop foreign key `FK_OrderLine_1148`;
drop table if exists `Customer`;
drop table if exists `PurchaseOrder`;
```

Figure 15.49 - Generate SQL dialog

You are allowed to copy the SQL statements from the **Generate SQL** dialog box.

Copying SQL statements from Specified Scope

You can specify the scope on the ERD to generate the SQL statements.

You can specify one of the three scopes:

- All entities on the ERD
    1. Select all entities on the ERD.
    2. Right-click on the entity, select **Copy SQL > Detail...**.

As copying SQL without specifying scope, SQL statements will be generated for all components including both entities and relationships on the ERD.

Example:

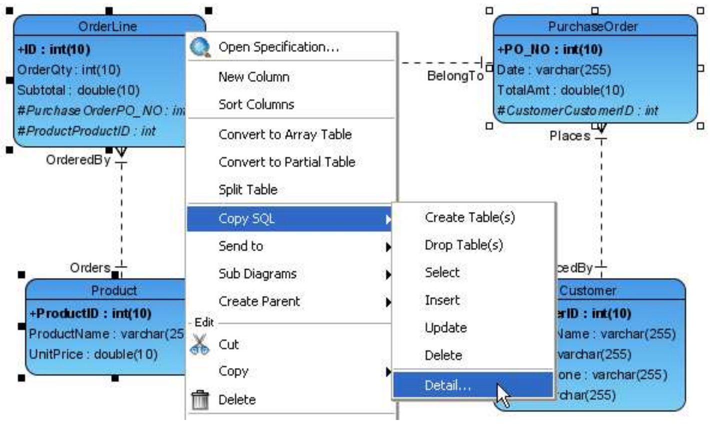
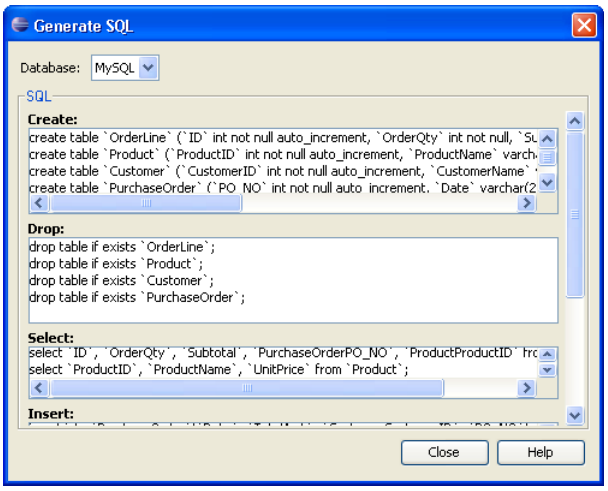


Figure 15.50 - To generate SQL for select entities



```
create table `OrderLine` (`ID` int not null auto_increment, `OrderQty` int not null, `Su
create table `Product` (`ProductID` int not null auto_increment, `ProductName` varch
create table `Customer` (`CustomerID` int not null auto_increment, `CustomerName`
create table `PurchaseOrder` (`PO_NO` int not null auto_increment, `Date` varchar(2
```

```
drop table if exists `OrderLine`;
drop table if exists `Product`;
drop table if exists `Customer`;
drop table if exists `PurchaseOrder`;
```

```
select `ID`, `OrderQty`, `Subtotal`, `PurchaseOrderPO_NO`, `ProductProductID` fro
select `ProductID`, `ProductName`, `UnitPrice` from `Product`;
```

Figure 15.51 - Generate SQL for selected entities

- Multiple entities and connection lines on the ERD
    1. Select several entities and relationships on the ERD, press **Alt**, right-click on the diagram pane, select **Copy SQL** from the pop-up menu.

As copying SQL with specifying a particular scope, SQL statements will be generated only for the components included in the specified scope.

Example:

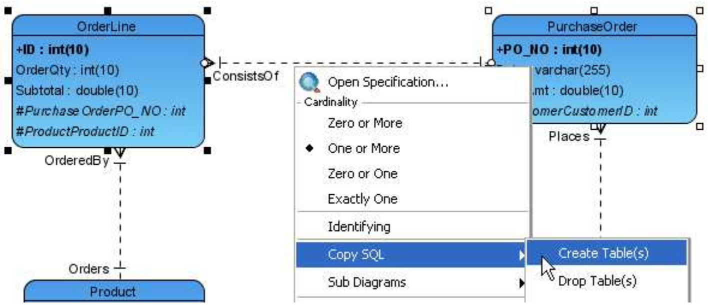


Figure 15.52 - To copy the SQL for create table

- Connection lines on the ERD
    1. Select connection line, press **Alt**, right-click on the diagram pane, select Copy SQL from the pop-up menu.

As copying SQL with connection lines

Example:

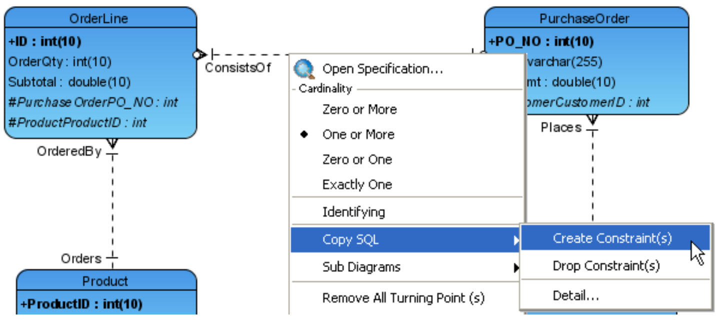


Figure 15.53 - To copy the create constraint SQL statement

Create Constraint :
```
alter table `OrderLine` add index `FK_OrderLine_1969` (`PurchaseOrderPO_NO`),
add constraint `FK_OrderLine_1969` foreign key (`PurchaseOrderPO_NO`)
references `PurchaseOrder` (`PO_NO`);
```

Drop Constraint :
```
alter table `OrderLine` drop foreign key `FK_OrderLine_1969`;
```

Mapping a Data Model to an Object Model

Object Relational Mapping (ORM) is supported which maps data models to object models and vice versa.

Mapping between objects to relational database preserves not only the data, but also the state, foreign/primary key mapping, difference in data type and business logic. Thus, you are not required to handle those tedious tasks during software development.

Mapping Entities to Classes

All entities map one-to-one to persistent classes in an object model.

Example:

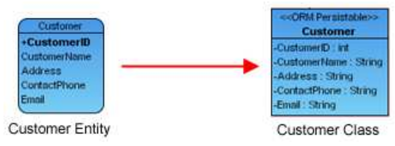


Figure 15.54 - Mapping Entities to Classes

In the above example, the Customer Entity map one-to-one the Customer Class as the Customer instance can store the customer information from the Customer Entity.

Mapping Columns to Attributes

Since all entities map one-to-one to persistent classes in an object model, columns in turn map to attributes in a one-to-one mapping. All specialty columns such as computed columns and foreign key columns are ignored during the transformation.

Example:

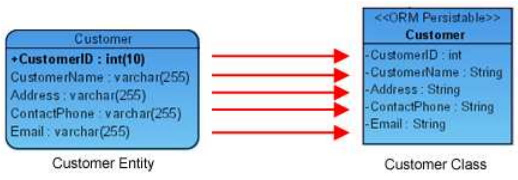


Figure 15.55 - Mapping Columns to Attributes

In the above example, the following table shows the mapping between the columns of the Customer Entity and the attributes of the Customer Class.

Customer Entity	Customer Class
CustomerID	CustomerID
CustomerName	CustomerName
Address	Address
ContactPhone	ContactPhone
Email	Email

Table 15.5

Mapping Data Type

The column data type automatically maps to an appropriate attribute type of object model.

Example:

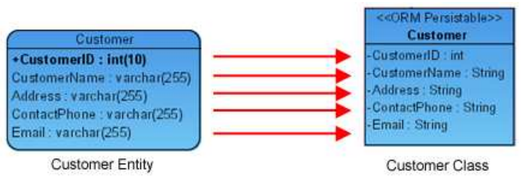


Figure 15.56 - Mapping Data Type

In the above example, the following table shows the mapping between data types

Customer Entity	Customer Class
int (10)	int
varchar(255)	String

Table 15.6

A table shows the data type mapping between Object model and Data model.

Data Model	Object Model
Bigint	Long
Binary	Byte[]
Bit	Boolean
Blob	Blob
Varchar	String
Char	Character
Char(1)	Character
Clob	String
Date	Date
Decimal	BigDecimal
Double	Double
Float	Float
Integer	Integer
Numeric	BigDecimal
Real	Float
Time	Time
Timestamp	Timestamp
Tinyint	Byte
Smallint	Short
Varbinary	Byte[]

Table 15.7

Mapping Primary Key

As the columns map to attributes in a one-to-one mapping, primary key columns in the entity map to attributes as a part of a class.

Example:

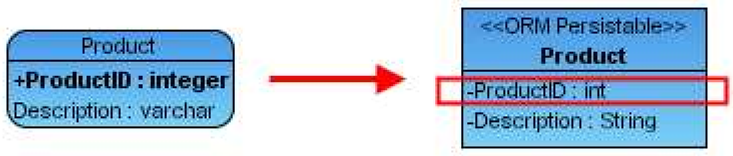


Figure 15.57 - Mapping Primary Key

In the example, the primary key of entity Product, **ProductID** maps to an attribute ProductID of the class Product.

Mapping Relationship

Relationship represents the correlation among entities. Each entity of a relationship has a role, called Phrase describing how the entity acts in it. The phrase is attached at the end of a relationship connection line. The phrase maps to role name of association in the object model.

There are two types of relationships in data model mapping to object model -identifying and non-identifying.

Identifying relationship specifies the part-of-whole relationship. It means that the child instance cannot exist without the parent instance. Once the parent instance is destroyed, the child instance becomes meaningless.

Non-identifying relationship implies weak dependency relationship between parent and child entities. There are two kinds of non-identifying relationships, including optional and mandatory. The necessity of the parent entity is "exactly one" and "zero or one" in the mandatory and optional non-identifying relationship respectively.

Mapping Identifying Relationship

Since the identifying relationship specifies the part-of-whole relationship, it map to composite aggregations which implies that the part cannot exist without its corresponding whole.

Example:

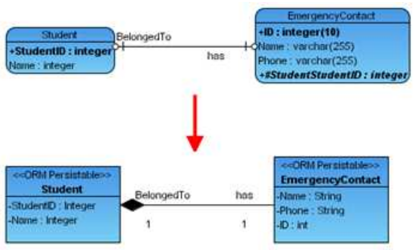


Figure 15.58 - Mapping Identifying Relationship

In the above example, the identifying relationship between the entities of EmergencyContact and Student maps to composition aggregation.

Mapping Non-identifying Relationship

Since non-identifying relationship implies weak relationship between entities, it maps to association.

Example:

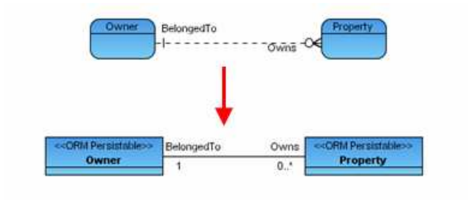


Figure 15.59 - Mapping Non-identifying Relationship

In the above example, non-identifying relationship between entities Owner and Property maps to association between Classes of Owner and Property.

Mapping Cardinality

Cardinality refers to the number of possible instances of an entity relate to one instance of another entity. The following table shows the syntax to express the Cardinality.

Table shows the syntax expressing the Cardinality

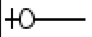
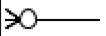
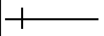
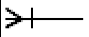
Type of Cardinality	Description
	Zero or one instance
	Zero or more instances
	Exactly one instance
	One or more instances

Table 15.8

Table shows the typical mapping between Entity Relationship Diagram and Class Diagram.

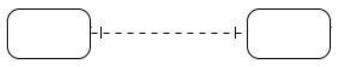
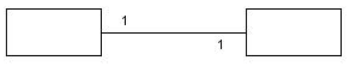
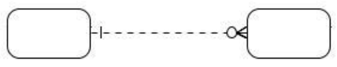
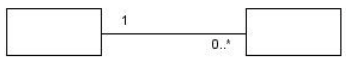
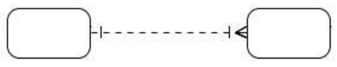
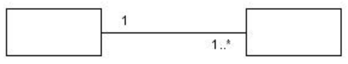
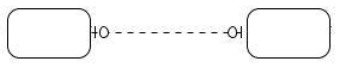
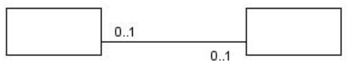
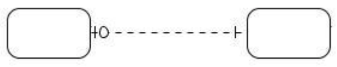
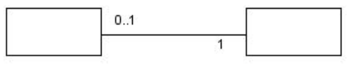
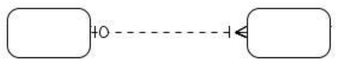
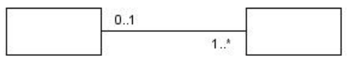
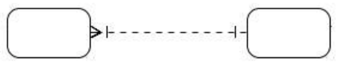
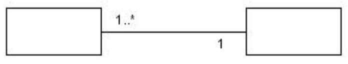
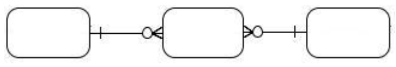
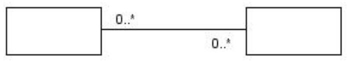
Entity Relationship Diagram	Class Diagram
	
	
	
	
	
	
	
	

Table 15.9

Mapping Many-to-Many Relationship

For each many-to-many relationship between entities, a Link Entity will be generated to form two one-to-many relationships in between. The primary keys of the two entities will automatically migrate to the link entity to form the primary/foreign keys.

Example:

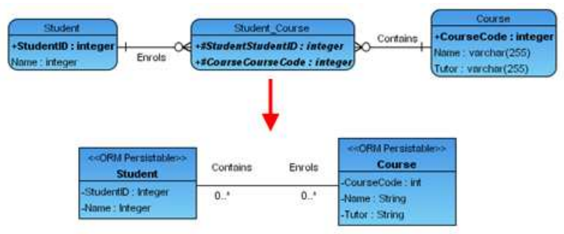


Figure 15.59 - Mapping Many-to-many Relationship

In the above example, SDE generates the link entity once a many-to-many relationship is setup between two entities. To transform the many-to-many relationship, the many-to-many relationship maps to many-to-many association.

Mapping Array Table to Collection of Objects

The Array Table is promoted to allow users retrieve objects in the form of primitive array.
When transforming an array table, the array table will map to an attribute with array type modifier.

Example:

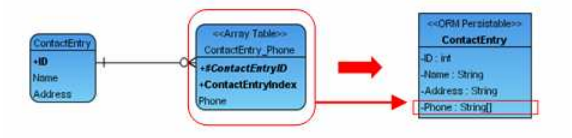


Figure 15.60 - Mapping Array Table to Collection of Objects

In the above example, the phonebook has a contact entry for each contact person. Each contact person may have more than one phone numbers. The array table of ContactEntry_Phone maps into the phone attribute with array type modifier in the ContactEntry class.

Mapping Data Model Terminology

The following table shows the shift from data model to object model terminology.

Data Model Term	Object Model Term
Entity	Class
Instance of an entity	Object
Relationship	Association
Supertype/subtype	Generalization
Column	Attribute
Phrase	Role
Cardinality	Multiplicity

Table 15.10

Mapping a Data Model to an Enterprise JavaBeans Model

Entity Bean to Database Mapping is supported which maps data models to Enterprise JavaBeans models and vice versa.

Mapping between entity beans to relational database preserves not only the data, but also the state, foreign/primary key mapping, difference in data type and business logic. Thus, you are not required to handle those tedious tasks during software development.

Mapping Entities to Entity Bean Classes

All entities map one-to-one to entity bean in an Enterprise JavaBeans model.

Example:

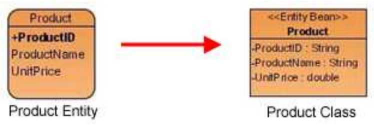


Figure 15.61 - Mapping Entities to Entity Bean Classes

In the above example, the Product Entity map one-to-one the Product Entity Bean as the Product Entity Bean instance can store the product information from the Product Entity.

Mapping Columns to Attributes

Since all entities map one-to-one to entity bean in an Enterprise JavaBeans model, columns in turn map to attributes in a one-to-one mapping. All specialty columns such as computed columns and foreign key columns are ignored during transformation.

Example:

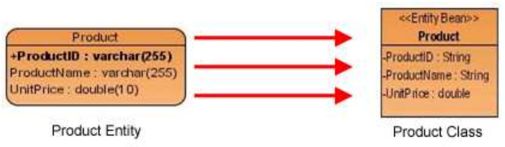


Figure 15.62 - Mapping Columns to Attributes

In the above example, the following table shows the mapping between the columns of the Product Entity and the attributes of the Product Entity Bean Class.

Product Entity	Product Entity Bean
ProductID	ProductID
ProductName	ProductName
UnitPrice	UnitPrice

Table 15.11

Mapping Data Type

The column data type automatically maps to an appropriate attribute type of Enterprise JavaBeans model.

Example:

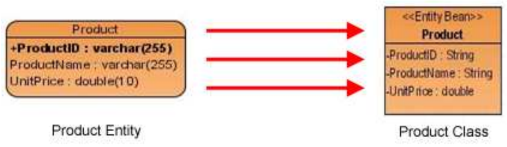


Figure 15.63 - Mapping Data Type

In the above example, the following table shows the mapping between data types

Product Entity	Product Entity Bean
varchar(255)	String
double(10)	double

Table 15.12

A table shows the data type mapping between Enterprise JavaBeans model and Data model.

Data Model	Enterprise JavaBeans Model
Bigint	Long
Binary	Byte[]
Bit	Boolean
Blob	Blob
Varchar	String
Char	Character
Char(1)	Character
Clob	String
Date	Date
Decimal	BigDecimal
Double	Double
Float	Float
Integer	Integer
Numeric	BigDecimal
Real	Float
Time	Time
Timestamp	Timestamp
Tinyint	Byte
Smallint	Short
Varbinary	Byte[]

Table 15.13

Mapping Primary Key

As the columns map to attributes in a one-to-one mapping, primary key columns in the entity map to attributes as a part of an entity bean class.

Example:

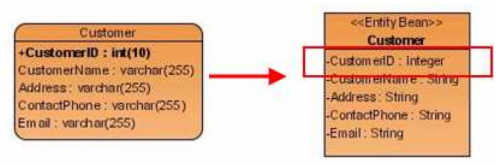


Figure 15.64 - Mapping Primary Key

In the example, the primary key of entity Customer, CustomerID maps to an attribute CustomerID of the entity bean Customer.

Mapping Relationship

Relationship represents the correlation among entities. Each entity of a relationship has a role, called Phrase describing how the entity acts in it. The phrase is attached at the end of a relationship connection line. The phrase maps to role name of association in the object model.

There is only one type of relationship in data model mapping to EJB Diagram - non-identifying.

Non-identifying relationship implies weak dependency relationship between parent and child entities. There are two kinds of non-identifying relationships, including optional and mandatory. The necessity of the parent entity is "exactly one" and "zero or one" in the mandatory and optional non-identifying relationship respectively.

Mapping Non-identifying Relationship

Since non-identifying relationship implies weak relationship between entities, it maps to association.

Example:

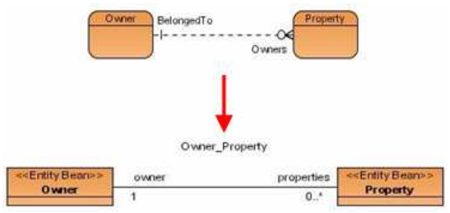


Figure 15.65 - Mapping Non-identifying Relationship

In the above example, non-identifying relationship between entities Owner and Property maps to association between Entity Beans of Owner and Property.

Mapping Cardinality

Cardinality refers to the number of possible instances of an entity relate to one instance of another entity. The following table shows the syntax to express the Cardinality.

Table shows the syntax expressing the Cardinality

Type of Cardinality	Description
┼○──	Zero or one instance
>○──	Zero or more instances
┼──	Exactly one instance
>┼──	One or more instances

Table 15.14

Table shows the typical mapping between Entity Relationship Diagram and EJB Diagram.

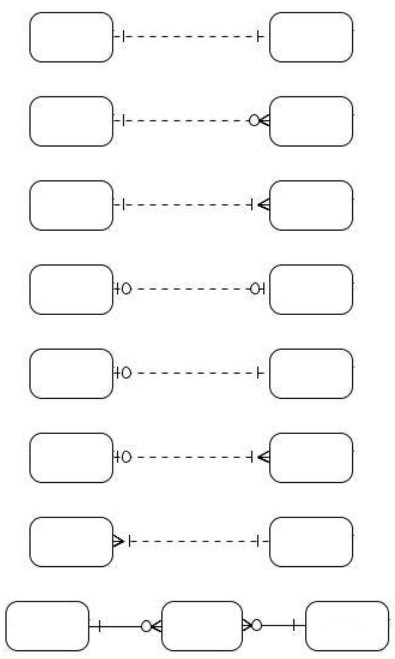
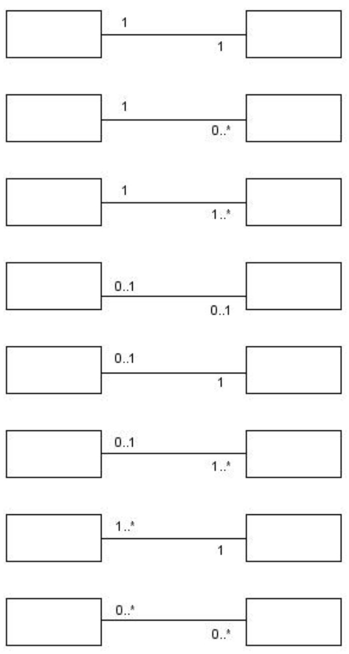
Entity Relationship Diagram	EJB Diagram
	 1 1 1 0..* 1 1..* 0..1 0..1 0..1 1 0..1 1..* 1..* 1 0..* 0..*

Table 15.15

Mapping Many-to-Many Relationship

For each many-to-many relationship between entities, a Link Entity will be generated to form two one-to-many relationships in between. The primary keys of the two entities will automatically migrate to the link entity to form the primary/foreign keys.

Example:

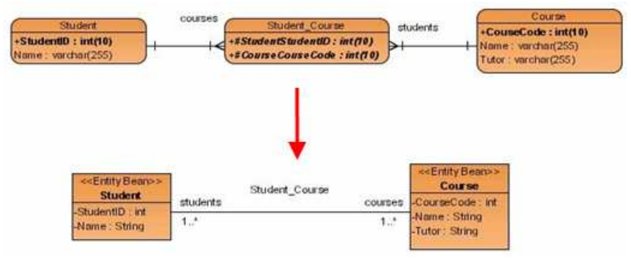


Figure 15.66 - Mapping Many-to-many Relationship

In the above example, the link entity is generated once a many-to-many relationship is setup between two entities. To transform the many-to-many relationship, the many-to-many relationship maps to many-to-many association.

Mapping Data Model Terminology

The following table shows the shift from data model to Enterprise JavaBeans model terminology.

Data Model Term	Enterprise JavaBeans Model Term
Entity	Entity Bean Class
Instance of an entity	Object
Relationship	Association
Column	Attribute
Phrase	Role
Cardinality	Multiplicity

Table 15.16

16 Database Schema

Chapter 16 - Database Schema

The database can be generated from either Entity Relationship Diagram or Class Diagram. You are allowed to configure the database connection and generate the database schema by exporting the Entity Relationship Diagram or Class Diagram to relational database. This chapter shows you how to generate the database and data definition language and describes how the data model maps the data definition language.

In this chapter:

- Introduction
- Generating Data Definition Language and Database
- Mapping Data Model to Data Definition Language

Introduction

Database schema refers to the database structure while data definition language (DDL) is a database language which describes the data structure in a database; that is, the DDL is used to define the database schema. The DDL statements support the creation and destruction of a database and/or table.

As the visual modeling of data model is supported, database connection is allowed to configure to the working environment. According to the data model, the database and DDL can also be generated.

Generating Data Definition Language and Database

SDE provides you with two ways to generate a relational database:

1. Generating Database from ERD
2. Generating Database from Class Diagram

Generating Database from Data Model

You can generate the database from data model in one of the two ways:

1. Using Database Code Generation Dialog Box
2. Using Wizard

Using Database Code Generation Dialog Box

1. On the menu, click **Modeling** > **ORM** > **Generate Database...**.

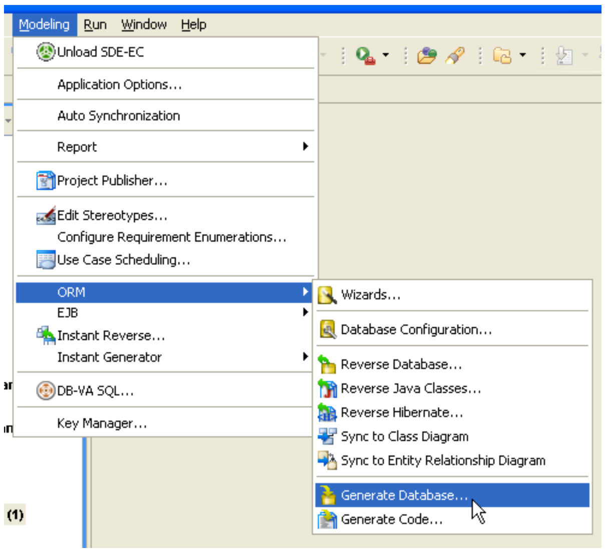


Figure 16.1 - To generate database

For other SDE:

SDE	Method
SDE for JBuilder	On the menu, click **Tools** > **Modeling** > **ORM** > **Generate Database...**.
SDE for NetBeans	On the menu, click **Modeling** > **ORM** > **Generate Database...**.
SDE for IntelliJ IDEA	On the menu, click **Modeling** > **ORM** > **Generate Database...**.
SDE for JDeveloper	On the menu, click **Model** > **ORM** > **Generate Database...**.
SDE for WebLogic Workshop	On the menu, click **Modeling** > **ORM** > **Generate Database...**.

Table 16.1

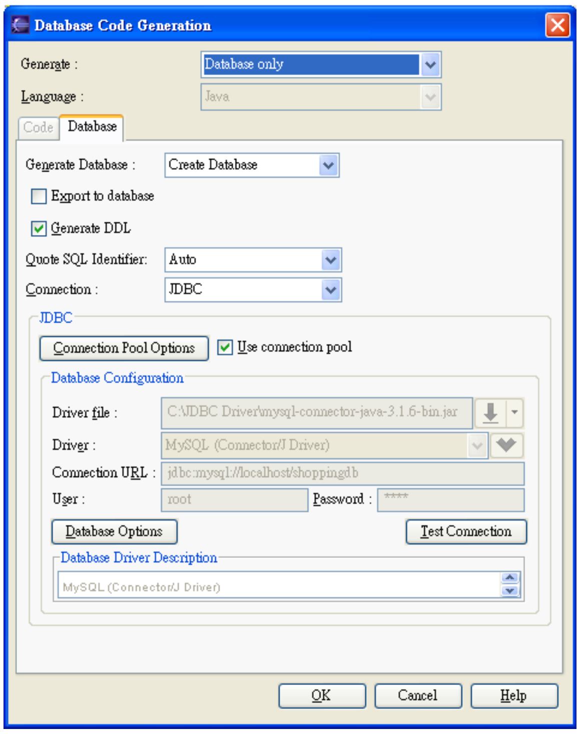
The **Database Code Generation** dialog box is displayed.

Generating Database for Java Project

For generating database for a Java project, configure the following options.

1. Click **Database Options** button, **Database Configuration** dialog box is displayed.

   You are allowed to define the database configuration. Refer to the descriptions in the Database Configuration for Java Project in the Getting Started with Object-Relational Mapping chapter for information on how to configure the database for Java project.



Database Code Generation

Generate : Database only
Language : Java
Code | Database
Generate Database : Create Database
☐ Export to database
☑ Generate DDL
Quote SQL Identifier: Auto
Connection : JDBC
JDBC
Connection Pool Options ☑ Use connection pool
Database Configuration
Driver file : C:\JDBC Driver\mysql-connector-java-3.1.6-bin.jar
Driver : MySQL (Connector/J Driver)
Connection URL : jdbc:mysql://localhost/shoppingdb
User : root Password : ****
Database Options Test Connection
Database Driver Description
MySQL (Connector/J Driver)
OK Cancel Help

Figure 16.2 - Database Code Generation dialog

2. Select the option **Export to database** to allow altering the database.

☑ Export to database

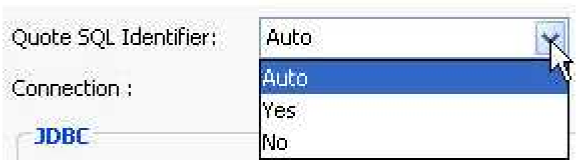
Figure 16.3 - Export to database checkbox

3. Select the option **Generate DDL** to allow the generation of DDL file.

☑ Generate DDL

Figure 16.4 - Generate DLL checkbox

4. Select the option for enabling **Quote SQL Identifier** from the drop-down menu. By enabling **Quote SQL Identifier**, the reserved word used in database can be used as ordinary word in generating database.



Quote SQL Identifier: Auto
Connection :
JDBC
Auto
Yes
No

Figure 16.5 - Quote SQL Identifier options

5. Select the type of connection from the drop-down menu of **Connection**, either **JDBC** or **Datasource**.

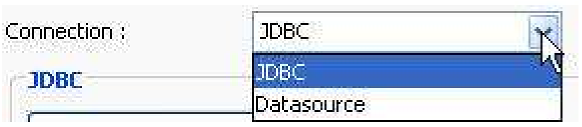
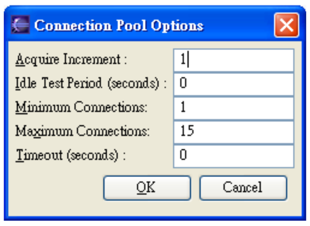


Figure 16.6 - Connection Type

6. Select the option for **Use connection pool** to enable using the connection pool for Java project.

Figure 16.7 - Use connection pool

7. Click **Connection Pool Options** button to open the **Connection Pool Options** dialog box to configure the connection pool settings.

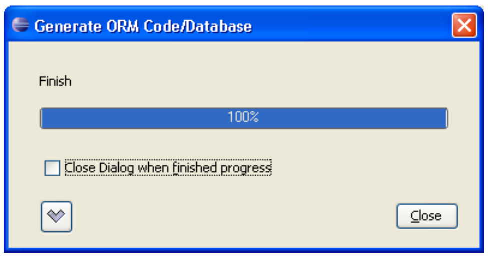


Connection Pool Options

Acquire Increment : 1
Idle Test Period (seconds) : 0
Minimum Connections: 1
Maximum Connections: 15
Timeout (seconds) : 0

Figure 16.8 - Connection Pool Options dialog

8. Click **OK**, you will be prompted by the **Generate ORM Code/Database** dialog box showing the progress of database generation. Click **Close** when the progress is 100% finished.



Generate ORM Code/Database

Finish

100%

Close Dialog when finished progress

Figure 16.9 - Generate ORM Code/Database dialog

The database tables are generated.

You are allowed to generate the database by selecting **Create Database**, **Update Database**, **Drop and Create Database** or **Drop Database** from the drop down menu.

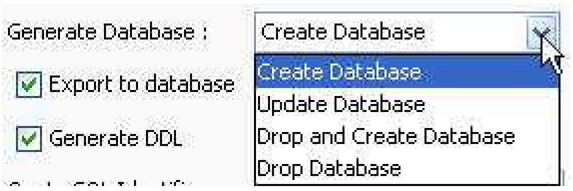


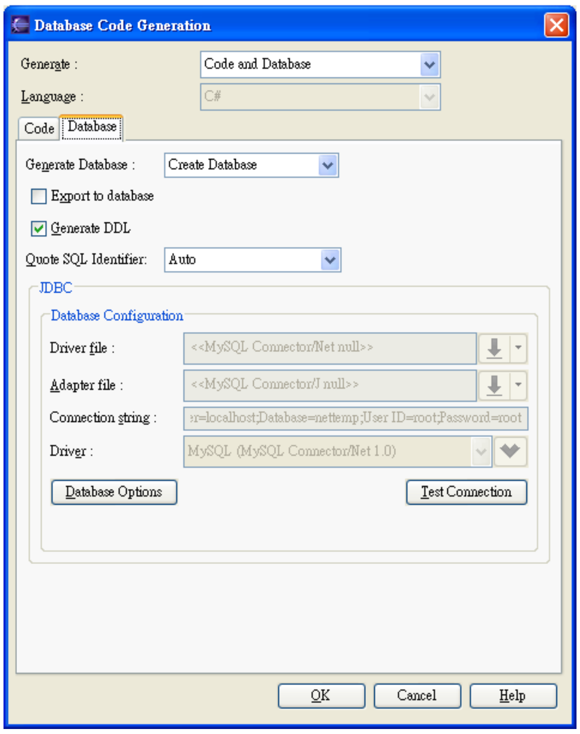
Figure 16.10 - Generate database options

Generating Database for .NET Project

For generating database for a .NET project, configure the following options.

1. Click **Database Options** button, **Database Configuration** dialog box is displayed.

   You are allowed to define the database configuration. Refer to the descriptions in the Database Configuration for .NET Project in the Getting Started with Object-Relational Mapping chapter for information on how to configure the database for .NET project.



Database Code Generation

Generate : Code and Database

Language : C#

Code | Database

Generate Database : Create Database

☐ Export to database

☑ Generate DDL

Quote SQL Identifier: Auto

JDBC

Database Configuration

Driver file : <<MySQL Connector/Net null>>

Adapter file : <<MySQL Connector/J null>>

Connection string : r=localhost;Database=nettemp;User ID=root;Password=root

Driver : MySQL (MySQL Connector/Net 1.0)

Figure 16.11 - Database Code Generation dialog

2. Select the option for **Export to database** to allow altering the database.

☑ Export to database

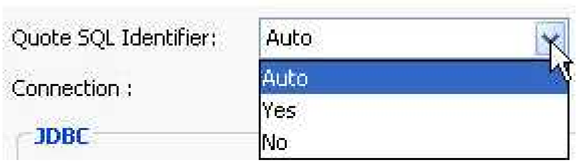
Figure 16.12 - Export to database checkbox

3. Select the option for **Generate DDL** to allow the generation of DDL file.

☑ Generate DDL

Figure 16.13 - Generate DDL checkbox

4. Select the option for enabling **Quote SQL Identifier** from the drop-down menu. By enabling **Quote SQL Identifier**, the reserved word used in database can be used as ordinary word in generating database.



Quote SQL Identifier: Auto

Connection :

JDBC

Figure 16.14 - Quote SQL Identifier options

5. Click **OK**, you will be prompted by the **Generate ORM Code/Database** dialog box showing the progress of database generation. Click **Close** when the progress is 100% finished.

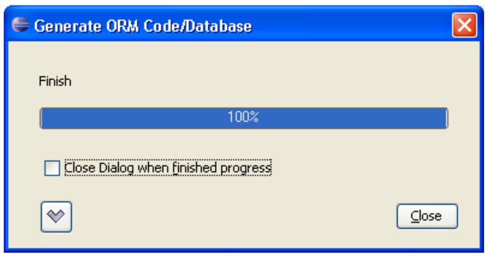

Generate ORM Code/Database
Finish
100%
Close Dialog when finished progress
Close

Figure 16.15 - Generate ORM Code/Database

he database tables are generated.

You are allowed to generate the database by selecting **Create Database**, **Update Database**, **Drop and Create Database** or **Drop Database** from the drop-down menu.

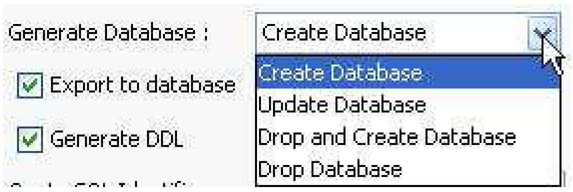


Figure 16.16 - Generate database options

Using Wizard

For information on generating database from data model by using wizard, refer to the descriptions in the Using ORM Wizard chapter.

Generating Database from Object Model

You can generate the database from object model by using wizard. For information, refers to the descriptions in the Using ORM Wizard chapter.

Mapping Data Model to Data Definition Language

As the database can be generated, it supports mapping from Data Model to Data Definition Language (DDL)

The following table shows the mapping between Data Model and DDL.

Data Model	DDL
Entity	Table
Column	Column
Data Type	Data Type
Primary Key	Primary Key
Foreign Key	Foreign Key

Table 16.2

Example:

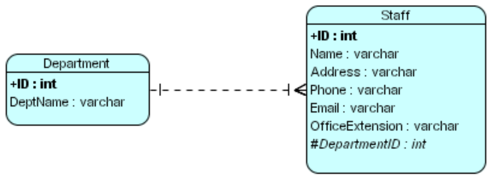


Figure 16.17 - Entity Relationship Diagram

The following DDL statements are generated by the above data model:

```
create table 'Department' ('ID' int not null auto_increment, 'DeptName' varchar(255), primary
key ('ID'))
create table 'Staff' ('ID' int not null auto_increment, 'Name' varchar(255), 'Address'
varchar(255), 'Phone' varchar(255), 'Email' varchar(255), 'OfficeExtension' varchar(255),
'DepartmentID' int not null, primary key ('ID'))
alter table 'Staff' add index 'FK_Staff_4271' ('DepartmentID'), add constraint 'FK_Staff_4271'
foreign key ('DepartmentID') references 'Department' ('ID')
```

The following figure illustrates the mapping between Data Model and DDL.

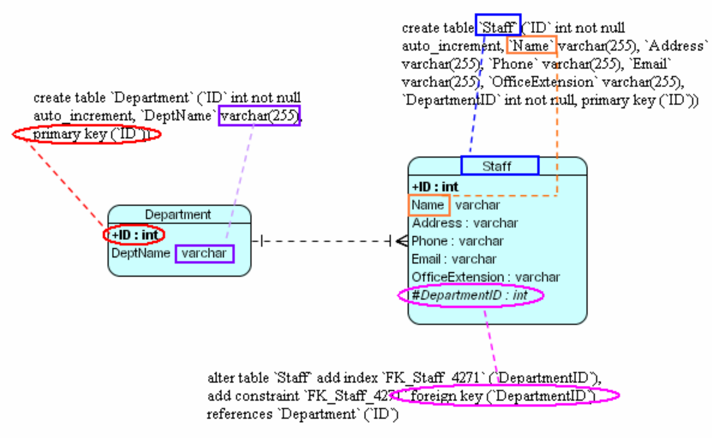

create table `Staff` (`ID` int not null auto_increment, `Name` varchar(255), `Address` varchar(255), `Phone` varchar(255), `Email` varchar(255), `OfficeExtension` varchar(255), `DepartmentID` int not null, primary key (`ID`))
create table `Department` (`ID` int not null auto_increment, `DeptName` varchar(255), primary key (`ID`))
Department
+ID : int
DeptName varchar
Staff
+ID : int
Name varchar
Address : varchar
Phone : varchar
Email : varchar
OfficeExtension : varchar
#DepartmentID : int
alter table `Staff` add index `FK_Staff_4271` (`DepartmentID`), add constraint `FK_Staff_4271` foreign key (`DepartmentID`) references `Department` (`ID`)

Figure 16.18 - Mapping between Data Model and DDL

The Staff entity maps to Staff table of the DDL; the primary key of Department entity maps to DDL for creating "primary key (ID)" constraint; the column Name of Staff entity maps to a column Name of the staff table to be generated; the data type of DeptName, which is varchar maps to "varchar(255)" to the DDL; the column DepartmentID of Staff Entity maps to the DDL for creating a foreign key constraint.

17 Implementation

Chapter 17 - Implementation

Class Diagram, Entity Relationship Diagram and Database can be used to generate the persistent code while the EJB Diagram can be used to generate the Enterprise JavaBeans. This chapter shows you how to generate the persistent code and Enterprise JavaBeans, and describes how the Object Model maps to ORM-Persistent Class.

In this chapter:

- Introduction
- Generating ORM-Persistable Class from Data, Object Models and Database
- Mapping Object Model to ORM-Persistable Java Class
- Mapping Object Model to ORM-Persistable .NET class
- Generating Enterprise JavaBeans (EJB) from Enterprise JavaBeans Model
- Mapping Enterprise JavaBeans Model to Enterprise JavaBeans
- Deploying Enterprise JavaBeans on Application Server
- Developing a Client Program

Introduction

Persistent code is the object that exposes the ability to store and retrieve data in relational databases permanently. An instance of the persistence class represents a row of a database table. As the persistent code is generated based on the object models, a mapping between object models and relational models is automatically built, which in turn supports your software development in database applications in an easier way.

ORM-Persistable Class

You are provided with three ways to generate persistent code.

1. Generating Persistent Code from ERD
2. Generating Persistent Code from Class Diagram
3. Generating Persistent Code from Database

Generating ORM-Persistable Class from Data Model

You can generate the persistent code from data model in one of the two ways:

1. Using Database Code Generation Dialog Box
2. Using ORM Wizard

Using Database Code Generation Dialog Box

As persistent code is generated based on the object model, and it will be used to manipulate the relational database which is the data model, if the object and data models were not synchronized, the generated persistent code may become malfunction.

Hence, there is a prerequisite of using Database Code Generation dialog box to generate persistent code, which is the synchronization between object and data models. The persistent code will be generated successfully as if the object and data models are synchronized in advance. For information on synchronization from data model to object model, refer to Synchronizing from a Data Model to an Object Model in the Object Model chapter.

To generate the persistent code from data model, synchronize the data model to object model before performing the following steps.

1. On the menu, click **Modeling** > **ORM** > **Generate Code...**.

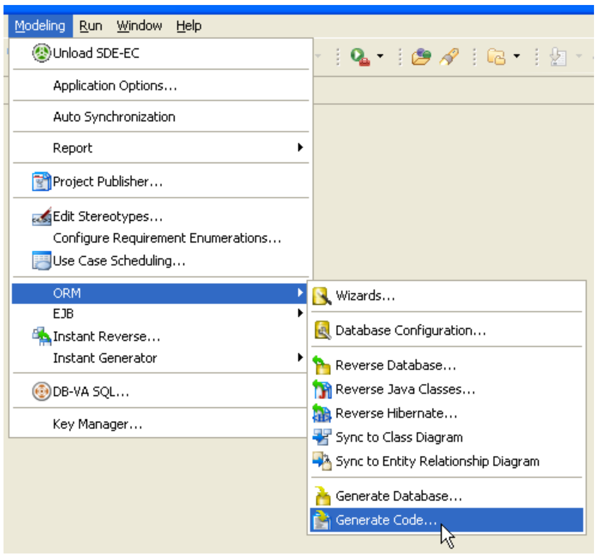


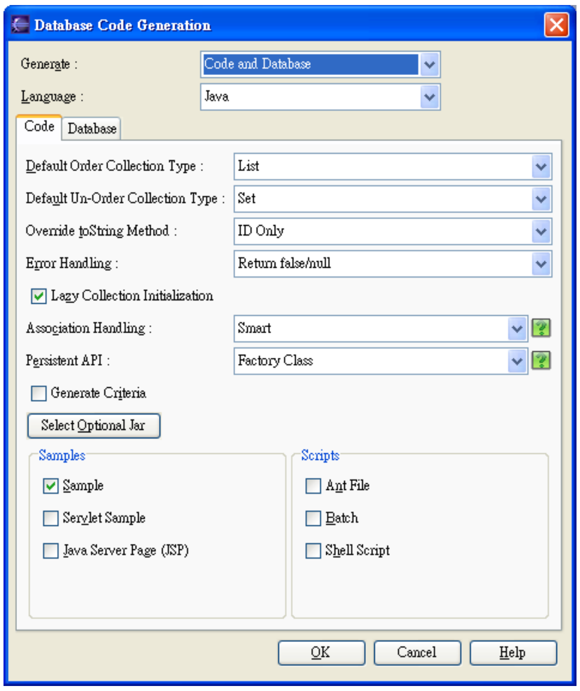
Figure 17.1 - To generate code

For other SDE:

SDE	Method
SDE for JBuilder	On the menu, click **Tools** > **Modeling** > **ORM** > **Generate Code...**.
SDE for NetBeans	On the menu, click **Modeling** > **ORM** > **Generate Code...**.
SDE for IntelliJ IDEA	On the menu, click **Modeling** > **ORM** > **Generate Code...**.
SDE for JDeveloper	On the menu, click **Model** > **ORM** > **Generate Code...**.
SDE for WebLogic Workshop	On the menu, click **Modeling** > **ORM** > **Generate Code...**.

Table 17.1

The **Database Code Generation** dialog box is displayed.


Database Code Generation

Generate : Code and Database
Language : Java

Code | Database

Default Order Collection Type : List
Default Un-Order Collection Type : Set
Override toString Method : ID Only
Error Handling : Return false/null
[✓] Lazy Collection Initialization
Association Handling : Smart
Persistent API : Factory Class
[ ] Generate Criteria
Select Optional Jar

Samples
[✓] Sample
[ ] Servlet Sample
[ ] Java Server Page (JSP)

Scripts
[ ] Ant File
[ ] Batch
[ ] Shell Script

OK Cancel Help

Figure 17.2 - Database code generation dialog

2. Select the language of code to be generated from the drop-down menu of **Language**, either **Java** or **C#.**

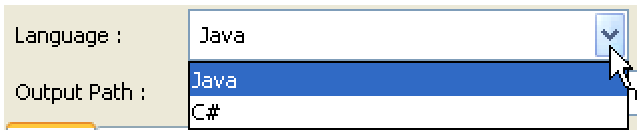
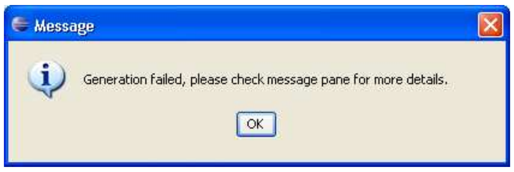


Figure 17.3 - The language options

3. Configure the code generation setting with respect to the selected language of code. Refer to the descriptions of Configuring Code Generation Setting for Java and Configuring Code Generation Setting for C# for information on how to configure the code generation setting for Java and C# respectively.
4. Click **OK**, you will be prompted by the **Generate ORM Code/Database** dialog box showing the progress of database generation. Click **Close** when the generation is complete; whereas if generation is failed, you will be prompted by a **Message** dialog box informing you to find the error/warning message on the **Message Pane**.



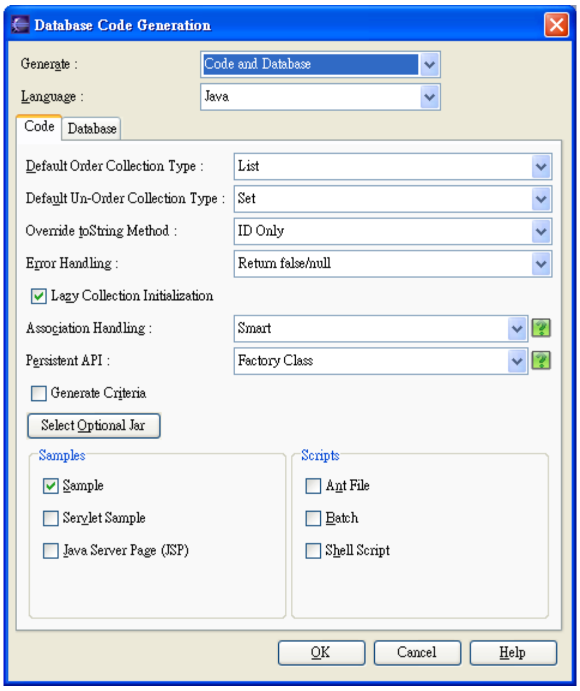
Generation failed, please check message pane for more details.

Figure 17.4 - Message dialog

The persistent code is generated.

Configuring Code Generation Setting for Java

By specifying the settings of code generation for Java, the Java persistence class will be generated accordingly.



Generate : Code and Database
Language : Java
Default Order Collection Type : List
Default Un-Order Collection Type : Set
Override toString Method : ID Only
Error Handling : Return false/null
☑ Lazy Collection Initialization
Association Handling : Smart
Persistent API : Factory Class
☐ Generate Criteria
☑ Sample
☐ Servlet Sample
☐ Java Server Page (JSP)
☐ Ant File
☐ Batch
☐ Shell Script

Figure 17.5 - Configuring Code Generation Setting for Java

- **Default Order Collection Type**

  Select the type of ordered collection to be used in handling the multiple cardinality relationship, either **List** or **Map**.

- **Default Un-Order Collection Type**

  Select the type of un-ordered collection to be used in handling the multiple cardinality relationship, either **Set** or **Bag**.

- **Override toString Method**

  Select the way that you want to override the toString method of the object. There are three options provided to override the toString method as follows:

  - **ID Only** - the toString method returns the value of the primary key of the object as string.
  - **All Properties** - the toString method returns a string with the pattern "Entity[<column1_name>=<column1_value><column2_name>=(column2_value>...]"
  - **No** - the toString method will not be overridden.
- **Error Handling**

  Select the way to handle errors when they occur.

  - **Return false/null** - It returns false/null in the method to terminate its execution.
  - **Throw PersistentException** - It throws a PersistentException which will be handled by the caller.
  - **Throw RuntimeException** - It throws a RuntimeException which will be handled by the caller.

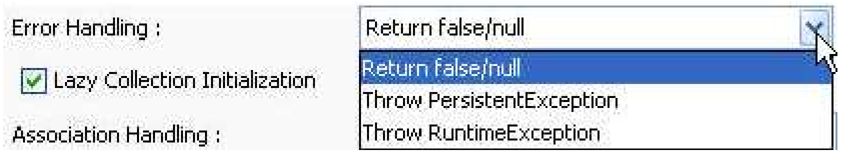


Figure 17.6 - Error Handle options

- **Lazy Collection Initialization**

  Check the option to avoid the associated objects from being loaded when the main object is loaded. Uncheck the option will result in loading the associated objects when the main object is loaded.

- **Association Handling**

  Select the type of association handling to be used, either **Smart**, or **Standard**.

  - **Smart**

    With smart association handling, when you update one end of a bi-directional association, the generated persistent code is able to update the other end automatically. Besides, you do not need to cast the retrieved object(s) into its corresponding persistence class when retrieving object(s) from the collection.

  - **Standard**

    With standard association handling, you must update both ends of a bi-directional association manually to maintain the consistency of association. Besides, casting of object(s) to its corresponding persistence class is required when retrieving object(s) from the collection.

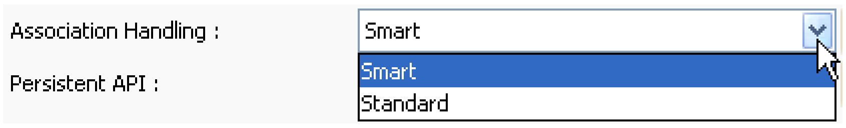


Figure 17.7 - Association Handling options

- **Persistent API**

  Select the type of persistent code to be generated, either **Static Methods**, **Factory Class**, **DAO** or **POJO**.

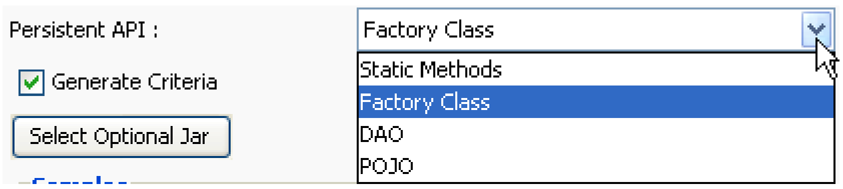


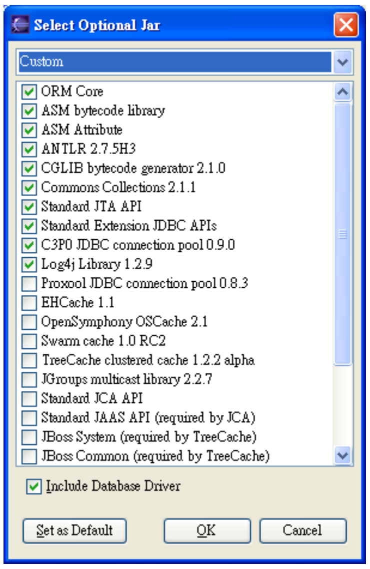
Figure 17.8 - Persistent API options

- **Generate Criteria**

  You can check the option for Generate Criteria to generate the criteria class for each ORM Persistable class. The criteria class supports querying the database by specifying the searching criteria. For information on using the criteria, refer to the description of Using Criteria Class in the Manipulating Persistent Data with Java chapter.

- **Select Optional Jar**

  You are allowed to select the libraries and JDBC driver to be included in the generation of the **orm.jar** file. Click on the **Select Optional Jar** button, the **Select Optional Jar** dialog box is displayed. Check the desired libraries to be included in the **orm.jar** which will be built at runtime.



Select Optional Jar

Custom

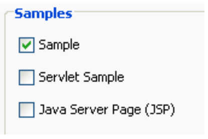
- [✓] ORM Core
- [✓] ASM bytecode library
- [✓] ASM Attribute
- [✓] ANTLR 2.7.5H3
- [✓] CGLIB bytecode generator 2.1.0
- [✓] Commons Collections 2.1.1
- [✓] Standard JTA API
- [✓] Standard Extension JDBC APIs
- [✓] C3P0 JDBC connection pool 0.9.0
- [✓] Log4j Library 1.2.9
- [ ] Proxool JDBC connection pool 0.8.3
- [ ] EHCache 1.1
- [ ] OpenSymphony OSCache 2.1
- [ ] Swarm cache 1.0 RC2
- [ ] TreeCache clustered cache 1.2.2 alpha
- [ ] JGroups multicast library 2.2.7
- [ ] Standard JCA API
- [ ] Standard JAAS API (required by JCA)
- [ ] JBoss System (required by TreeCache)
- [ ] JBoss Common (required by TreeCache)

[✓] Include Database Driver

Figure 17.9 - Select Optional Jar

- **Samples**

  Sample files, including Java application sample, servlet sample and Java Server Page (JSP) sample are available for generation. The generated sample files guide you through the usage of the Java persistence class. You can check the options to generate the sample files for reference.



Samples
- [✓] Sample
- [ ] Servlet Sample
- [ ] Java Server Page (JSP)

Figure 17.10 - General samples options

- **Scripts**

  Scripts, including Ant File, Batch and Shell Script are available for generation. You can execute the generated scripts directly.

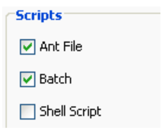
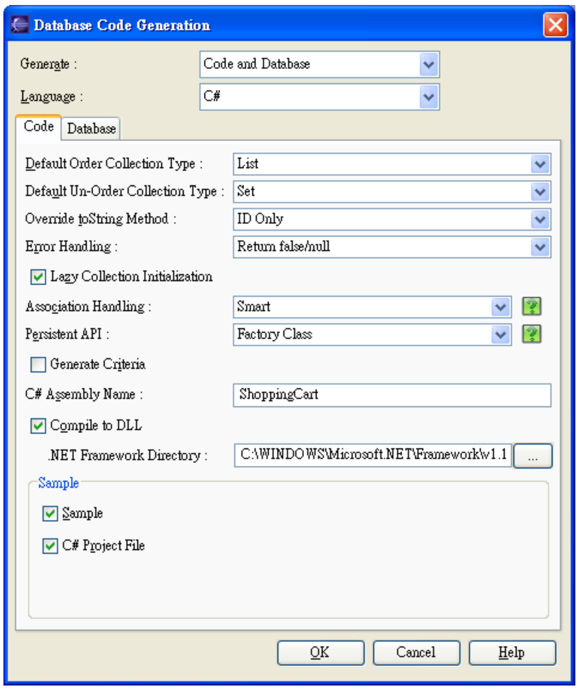

Scripts
☑ Ant File
☑ Batch
☐ Shell Script

Figure 17.11 - Generate scripts options

Configuring Code Generation Setting for C#

By specifying the settings of code generation for C#, the C# persistence class will be generated accordingly.


Database Code Generation

Generate : Code and Database
Language : C#

Code | Database

Default Order Collection Type : List
Default Un-Order Collection Type : Set
Override toString Method : ID Only
Error Handling : Return false/null
☑ Lazy Collection Initialization
Association Handling : Smart
Persistent API : Factory Class
☐ Generate Criteria
C# Assembly Name : ShoppingCart
☑ Compile to DLL
.NET Framework Directory : C:\WINDOWS\Microsoft.NET\Framework\v1.1

Sample
☑ Sample
☑ C# Project File

Figure 17.12 - Configuring Code Generation Setting for C#

- **Default Order Collection Type**

  Select the type of ordered collection to be used in handling the multiple cardinality relationship, either List or Map.

- **Default Un-Order Collection Type**

  Select the type of un-ordered collection to be used in handling the multiple cardinality relationship, either Set or Bag.

- **Override toString Method**

  Select the way that you want to override the toString method of the object. There are three options provided to override the toString method as follows:

  - **ID Only** - the toString method returns the value of the primary key of the object as string.
  - **All Properties** - the toString method returns a string with the pattern

    "Entity[<column1_name>=<column1_value><column2_name>=(column2_value>...]"

  - **No** - the toString method will not be overridden.
- **Error Handling**

  Select the way to handle errors when they occur

.

- **Return false/null** - It returns false/null in the method to terminate its execution.
- **Throw PersistentException** - It throws a PersistentException which will be handled by the caller.

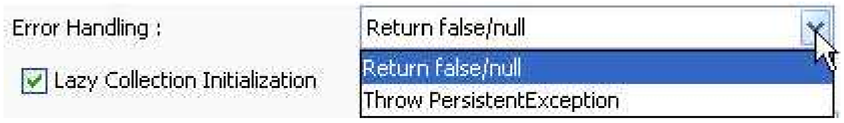


Figure 17.13 - Error Handling checkbox

- **Lazy Collection Initialization**

  Check the option to avoid the associated objects from being loaded when the main object is loaded. Uncheck the option will result in loading the associated objects when the main object is loaded.

- **Association Handling**

  Select the type of association handling to be used, either Smart, or Standard.

  - **Smart**

    With smart association handling, updating one end of a bi-directional association will update the other end automatically. Besides, casting of object(s) is not needed when retrieving object(s) from the collection.

  - **Standard**

    With standard association handling, both ends of a bi-directional association must be updated manually to maintain the consistency of association. Besides, casting of object(s) is required when retrieving object(s) from the collection.

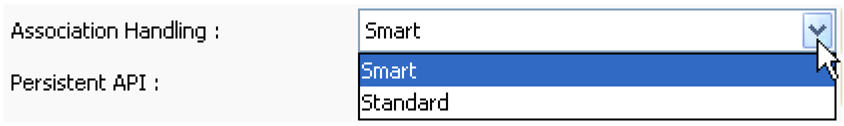


Figure 17.14 - Association Handling options

- **Persistent API**

  Select the type of persistent code to be generated, either Static Methods, Factory Class, DAO or POJO.

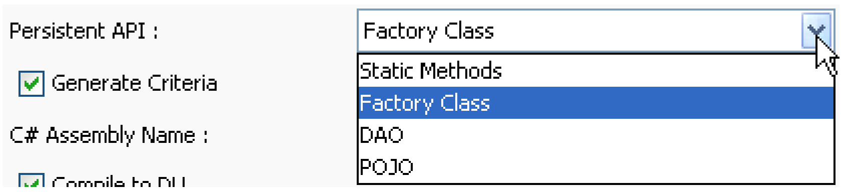


Figure 17.15 - Persistent API options

- **Generate Criteria**

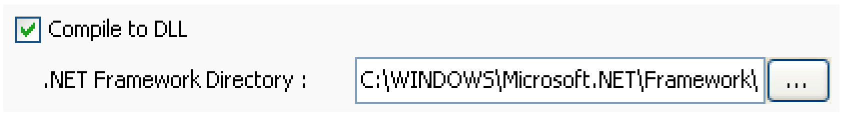
  You can check the option for Generate Criteria to generate the criteria class for each ORM Persistable class. The criteria class supports querying the database by specifying the searching criteria. For information on using the criteria, refer to the description of Using Criteria Class in the <u>Manipulating Persistent Data with .NET</u> chapter.

- **C# Assembly Name**

  Specify the name of the assembly for the .NET application which holds the assembly metadata.

- **Compile to DLL**

  By checking the option of Compile to DLL, the DLL files will be generated, which can be referenced by .NET projects of language other than C# source.



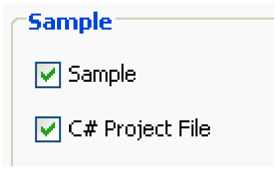
✓ Compile to DLL

.NET Framework Directory : C:\WINDOWS\Microsoft.NET\Framework\

Figure 17.16 - Compile to DLL setting

- **Sample**

  Sample files, C# project file are available for generation for reference. The generated sample files guide you through the usage of the .NET persistence class. You can check the options to generate the sample files.



Sample

✓ Sample

✓ C# Project File

Figure 17.17 - generate sample options

Using ORM Wizard

For information on generating persistent code from data model by using wizard, refer to the descriptions of Generating Code and Database from ERD in the Using ORM Wizard chapter.

Generating ORM-Persistable Class from Object Model

You can generate the persistent code from object model in one of the two ways:

1. Using Database Code Generation Dialog Box
2. Using ORM Wizard

Using Database Code Generation Dialog Box

Persistent code is designed to manipulate the relational database. The persistent code maps to object model; and meanwhile, data model maps to relational database. Thus, synchronization between object model and data model is important such that the object-relational mapping environment can be built successfully.

Hence, synchronizing the object model to data model is the prerequisite of using **Database Code Generation** dialog box. For more information, refer to Synchronizing from an Object Model to a Data Model in the Data Model chapter.

For information about using Database Code Generation dialog box, refer to Using Database Code Generation Dialog Box section.

Using ORM Wizard

For information on generating persistent code from object model by using wizard, refer to the descriptions of Generating Code and Database from Class Diagram in the Using ORM Wizard chapter.

Generating ORM-Persistable Class from Database

You can generate the persistent code from database by using Wizard. For more information, refer to the descriptions of Generating Code from Database in the Using ORM Wizard chapter.

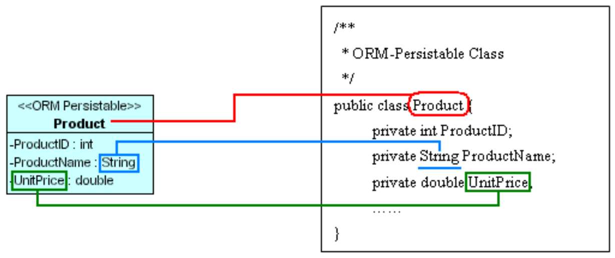
Mapping Object Model to ORM-Persistable Java Class

The persistent code can be generated with the support on mapping from Object Model to Persistent Code. In this section, examples will be given to show the mapping between object model and Java persistence class.

Mapping Classes, Attributes and Data Type

For each class element which may have attributes and methods in the object model, a corresponding Java class will be generated. The generated ORM persistable Java class will have the name as same as the name of the class element for identification. As if the class element has attributes with specified data types, all the attributes and data types will map to the Java class as well.

Example:



```
/**
 * ORM-Persistable Class
 */
public class Product {
        private int ProductID;
        private String ProductName;
        private double UnitPrice;
        ......
}
```

Figure 17.18 - Mapping Classes, Attributes and Data Type

In the above example, an ORM-Persistable Java class, Product was generated according to the class Product in the object model. There are three types of mapping shown above:

1. Mapping Class Name

   The name of the class, **Product** maps to the name of the Java class, **Product**. (It is illustrated by the red line in the diagram.)

2. Mapping Attribute

   The attributes of the class map to the attributes of the Java class. (It is illustrated by the green line in the diagram.) Besides, a pair of getter and setter methods for the corresponding attribute is generated to the Java class.

3. Mapping Data Type

   The data type of the class maps to the data type of the Java Class. (It is illustrated by the blue line in the diagram.)

Mapping Data Type

The following table shows the data type mapping between object model and Java persistent code.

Object Model	Java Persistent Code
Boolean	boolean
Byte	byte
Byte[]	byte[]
Blob	java.sql.Blob
Char	char
Character	character
String	String
Int	int
Integer	Integer

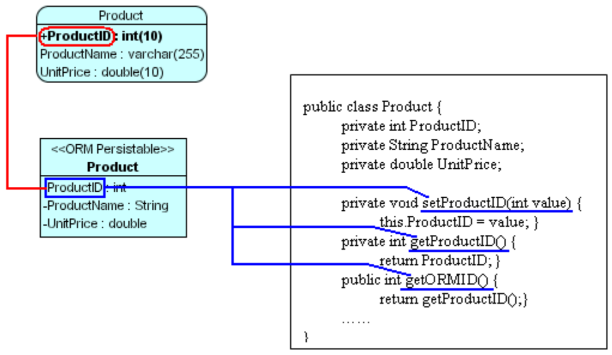
Double	int
Bigdecimal	BigDecimal
Float	float
Long	long
Short	short
Date	java.util.Date
Time	java.sql.Time
Timestamp	java.sql.Timestamp

Table 17.2

Mapping Primary Key

As object model maps to data model bridging the environment between object model and relational database, primary key is mapped. As the object model maps to persistent code, primary key is in turn mapped.

Example:



```
public class Product {
        private int ProductID;
        private String ProductName;
        private double UnitPrice;

        private void setProductID(int value) {
                this.ProductID = value; }
        private int getProductID() {
                return ProductID; }
        public int getORMID() {
                return getProductID();}
        ......
}
```

Figure 17.19 - Mapping Primary Key

In the above example, the primary key of Product entity, ProductID maps to an attribute, ProductID of class element, Product, which in turn becomes an attribute of the Java class.

In the ORM-Persistable class, a pair of getter and setter methods for the attribute is generated for accessing its value. Besides, the ORM-Persistable class has a method named as getORMID which returns the primary key value of the class.

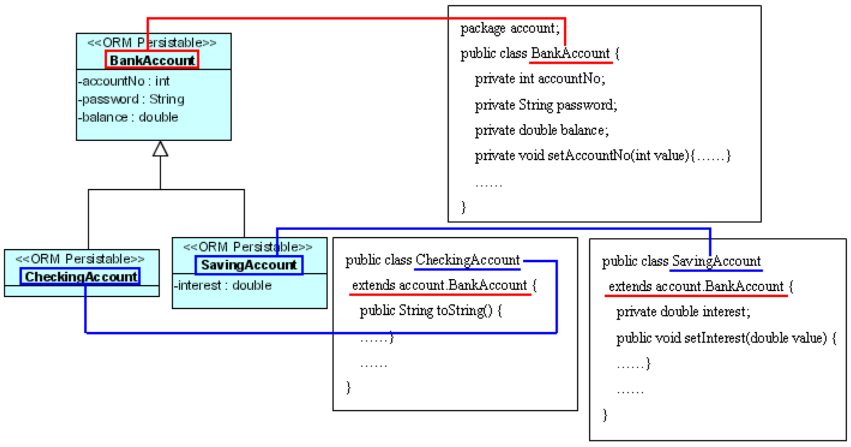
In the above example, the getORMID() method returns the value by calling the method getProductID() which returns the value of ProductID. It thus signifies ProductID is the primary key of the class.

Mapping Generalization

Generalization distributes the commonalities from the superclass among a group of subclasses. The subclass is consistent with the superclass and it may contain additional attributes and methods to make it specific.

Mapping generalization hierarchy to persistent code, the superclass and its subclasses are generated to individual Java class while the subclasses inherit all the superclass attributes and methods.

Example:



```
package account;
public class BankAccount {
    private int accountNo;
    private String password;
    private double balance;
    private void setAccountNo(int value){……}
    ……
}
```

```
public class CheckingAccount
  extends account.BankAccount {
    public String toString() {
    ……}
    ……
}
```

```
public class SavingAccount
  extends account.BankAccount {
    private double interest;
    public void setInterest(double value) {
    ……}
    ……
}
```

Figure 17.20 - Mapping generalization

In the above example, the generalization hierarchy of BankAccount, CheckingAccount and SavingAccount maps to three ORM-Persistable Java classes where the subclasses, CheckingAccount and SavingAccount extend their superclass, BankAccoount so as to inherit the BankAccount's attributes and methods.

Mapping Association and Multiplicity

An association specifies the semantic relationship between two classes in the object model with a role attached at one end. The role specifies the detailed behavior of a class associated in the association, including its role name, which indicates how the class acts in the association; and multiplicity, which specifies the number of links between each instance of the associated classes.

The role name of the association maps to an attribute which specifies how the class associates with its associated supplier class with respect to multiplicity. If a class has a multiplicity of many in an association, the class maps not only to an ORM persistent Java class, but also to a collection class which supports manipulating the objects within a collection.

You are allowed to define the type of collection class for code generation. Meanwhile, List and Set are the default types for ordered and un-ordered collection respectively.

Example:

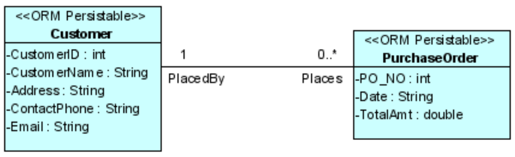
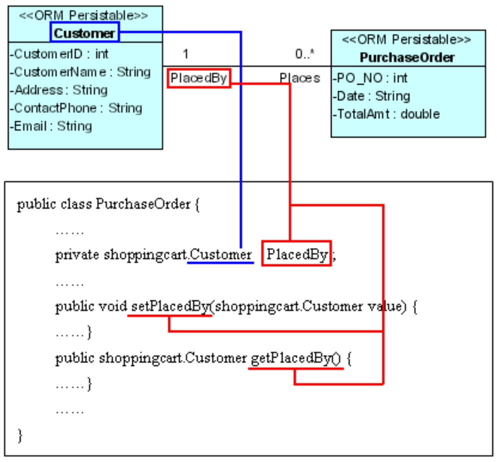


Figure 17.21 - Entity Relation Diagram

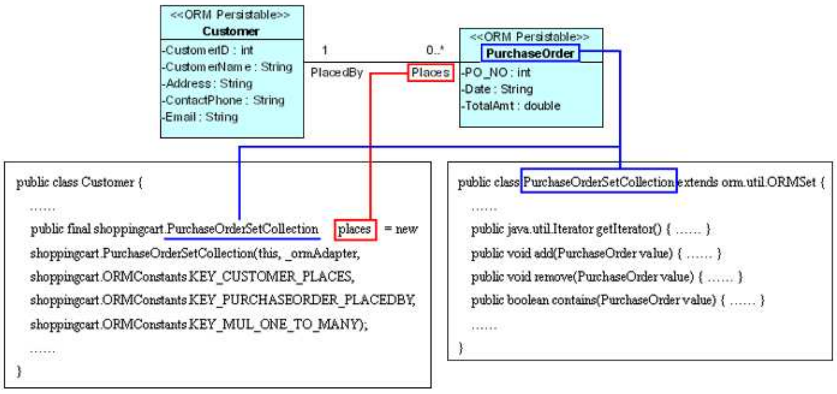
In the above object model, there is a one-to-many association between Customer class and PurchaseOrder class; defining that a customer can place many purchase order meanwhile a purchase order is placed by one customer. As object model maps to persistent code, three ORM-Persistable classes will be generated, including Customer, PurchaseOrder and PurchaseOrderSetCollection classes.



```
public class PurchaseOrder {
        ……
        private shoppingcart.Customer PlacedBy;
        ……
        public void setPlacedBy(shoppingcart.Customer value) {
        ……}
        public shoppingcart.Customer getPlacedBy() {
        ……}
        ……
}
```

Figure 17.22 - Mapping Association and Multiplicity

From the above diagram, PurchaseOrder class has the role of "PlacedBy" in the association and it is associated with the Customer class with multiplicity of one. The role, "PlacedBy" maps as an attribute to the PurchaseOrder class which is defined as the data type of Customer class; it implies that the attribute "PlacedBy" stores a customer object which places the particular purchase order.



```
public class Customer {
   ……
   public final shoppingcart.PurchaseOrderSetCollection places = new
   shoppingcart.PurchaseOrderSetCollection(this, _ormAdapter,
   shoppingcart.ORMConstants.KEY_CUSTOMER_PLACES,
   shoppingcart.ORMConstants.KEY_PURCHASEORDER_PLACEDBY,
   shoppingcart.ORMConstants.KEY_MUL_ONE_TO_MANY);
   ……
}
```

```
public class PurchaseOrderSetCollection extends orm.util.ORMSet {
   ……
   public java.util.Iterator getIterator() { …… }
   public void add(PurchaseOrder value) { …… }
   public void remove(PurchaseOrder value) { …… }
   public boolean contains(PurchaseOrder value) { …… }
   ……
}
```

Figure 17.23 - Mapping an association

From the above diagram, Customer has the role of "Places" in the association which associates with many PurchaseOrder instances. As the PurchaseOrder class has the multiplicity of many in the association, a collection class is generated so as to manipulate the multiple instances of PurchaseOrder. The role, "Places" maps as an attribute to the Customer class with the data type of PurchaseOrderSetCollection class; it implies that the attribute "Places" represents a collection of PurchaseOrder objects which was placed by a particular customer.

Mapping Navigable Association

An association, which specifies a semantic relationship occurring between typed instances, has at least two ends represented by properties. When a property at an end of the association is owned by an end class, it represents the association is navigable from the opposite ends. On the contrary, when a property does not appear in the namespace of any associated class, it represents a non-navigable end of the association.

An open arrowhead on the end of an association indicates the end is navigable. Meanwhile, a small cross (x) on the end of an association indicates the end is not navigable. If an association has two navigable ends, it is considered as bi-directional associations.

As the role name of the association maps to an attribute which specifies how the class associates with its associated class, however if the association has a non-navigable end, the corresponding role name will not map to an attribute, suppressing the ownership of the class.

Example:

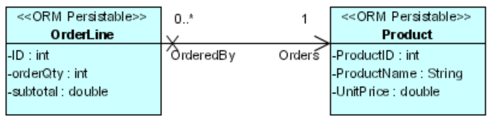
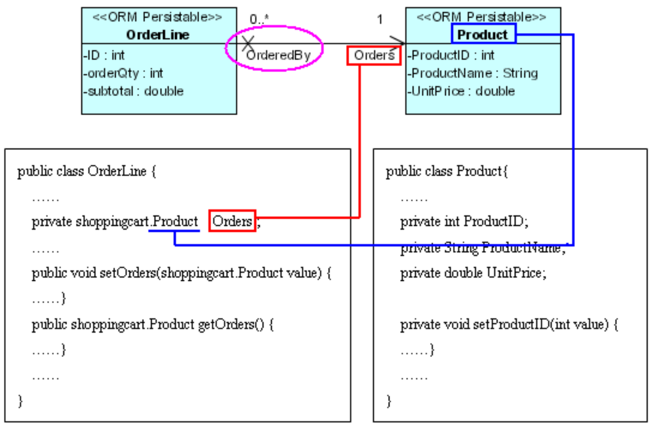


Figure 17.24 - Class Diagram with navigable

In the above object model, it shows a binary association with one end navigable and the other non-navigable. It indicates that Product is owned by OrderLine while Product cannot own OrderLine. As the object model maps to persistent code, the ownership of OrderLine instance in Product class will not be generated in the persistent code.



```
public class OrderLine {
   ......
   private shoppingcart.Product Orders;
   ......
   public void setOrders(shoppingcart.Product value) {
   ......}
   public shoppingcart.Product getOrders() {
   ......}
   ......
}
```

```
public class Product{
   ......
   private int ProductID;
   private String ProductName;
   private double UnitPrice;

   private void setProductID(int value) {
   ......}
   ......
}
```

Figure 17.25 - Mapping with navigable

From the above diagram, OrderLine class has the role of "Orders" in the association and associated with the one Product instance. The role, "Orders" maps as an attribute to OrderLine class which defines as the data type of Product; it implies that the attribute "Orders" stores a product instance.

On the other hand, Product class has the role of "OrderedBy" associated with many OrderLine instances in a non-navigable association; that is, Product class does not have the ownership of OrderLine. Even though the OrderLine class has the multiplicity of many in the association, the corresponding collection class of OrderLine for the Product class is not generated to manipulate it.

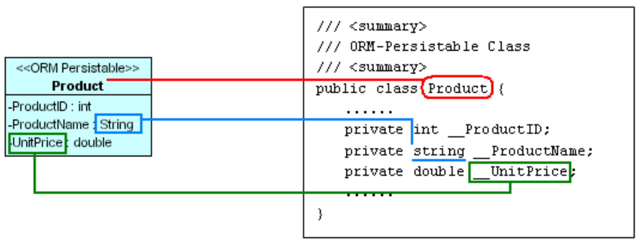
Mapping Object Model to ORM-Persistable .NET Class

As the .NET persistence class can be generated with the support on mapping from Object Model to Persistent Code. In this section, examples will be given to show the mapping between object model and .Net persistence class.

Mapping Classes, Attributes and Data Type

For each class element which may have attributes and methods in the object model, a corresponding persistence class of C# will be generated. The generated ORM persistable .NET class will have the name as same as the name of the class element for identification. As if the class element has attributes with specified data types, all the attributes and data types will map to the .NET persistence class as well.

Example:



```
/// <summary>
/// ORM-Persistable Class
/// <summary>
public class Product {
   ......
   private int __ProductID;
   private string __ProductName;
   private double __UnitPrice;
   ......
}
```

Figure 17.26 - Mapping Classes, Attributes and Data type

In the above example, an ORM-Persistable .NET class, Product was generated according to the class Product in the object model. There are three types of mapping shown above:

1. Mapping Class Name

   The name of the class, **Product** maps to the name of the .NET persistence class, **Product**. (It is illustrated by the red line in the diagram.)

2. Mapping Attribute

   The attributes of the class map to the attributes of the .NET persistence class. (It is illustrated by the green line in the diagram.) Besides, a pair of getter and setter methods for the corresponding attribute is generated to the persistence class.

3. Mapping Data Type

   The data type of the class maps to the data type of the .NET persistence class. (It is illustrated by the blue line in the diagram.)

Mapping Data Type

The following table shows the data type mapping between object model and .NET persistent code.

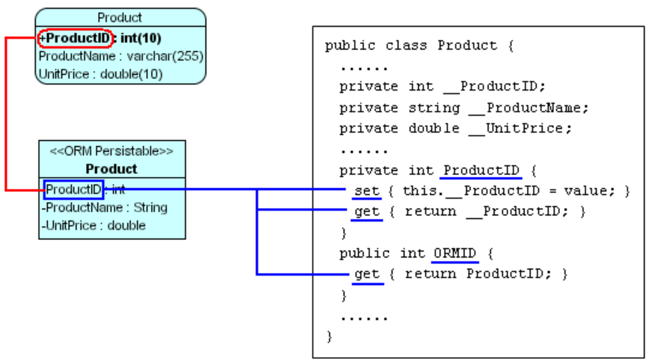
Object Model	.NET Persistent Code
Boolean	bool
Byte	byte
Byte[]	byte[]
Blob	byte[]
Char	char
Character	char
String	string
Int	int
Integer	int
Double	double
Decimal	Decimal
Float	float
Long	long
Short	short
Date	DateTime
Time	DateTime
Timestamp	DateTime

Table 17.3

Mapping Primary Key

As object model maps to data model bridging the environment between object model and relational database, primary key is mapped. As the object model maps to persistent code, primary key is in turn mapped.

Example:



```
public class Product {
  ......
  private int __ProductID;
  private string __ProductName;
  private double __UnitPrice;
  ......
  private int ProductID {
    set { this.__ProductID = value; }
    get { return __ProductID; }
  }
  public int ORMID {
    get { return ProductID; }
  }
  ......
}
```

Figure 17.27 - Mapping Primary Key

In the above example, the primary key of Product entity, ProductID maps to an attribute, ProductID of class element, Product, which in turn becomes an attribute of the .NET persistence class.

In the ORM-Persistable .NET class, a pair of getter and setter methods for the attribute is generated for accessing its value. Besides, the ORM-Persistable .NET class has a method of getting the ORMID which returns the primary key value of the class.

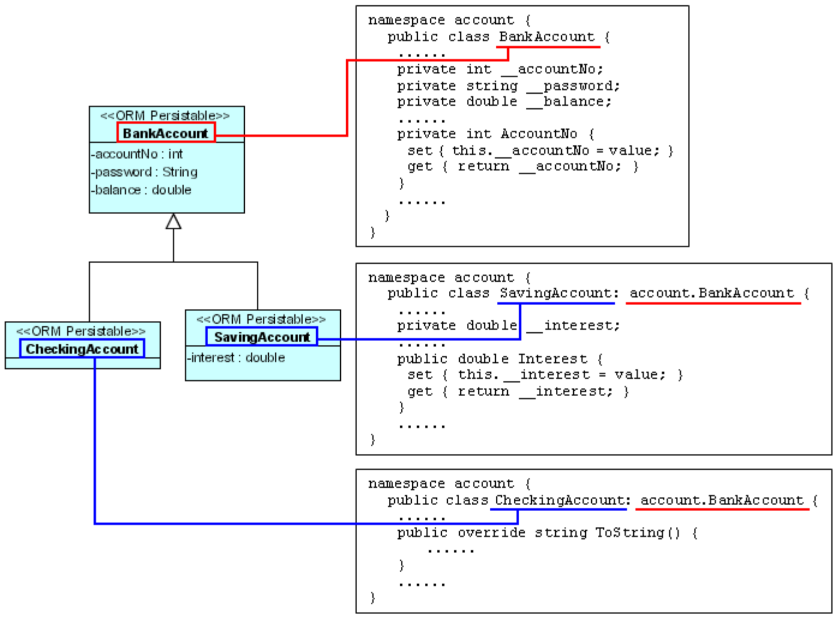
In the above example, the getter method of ORMID returns the value by calling the getter method of ProductID which returns the value of __ProductID. It thus signifies ProductID is the primary key of the class.

Mapping Generalization

Generalization distributes the commonalities from the superclass among a group of subclasses. The subclass is consistent with the superclass and it may contain additional attributes and methods to make it specific.

Mapping generalization hierarchy to persistent code, the superclass and its subclasses are generated to individual .NET persistence class while the subclasses inherit all the superclass attributes and methods.

Example:



```
namespace account {
  public class BankAccount {
    ......
    private int __accountNo;
    private string __password;
    private double __balance;
    ......
    private int AccountNo {
      set { this.__accountNo = value; }
      get { return __accountNo; }
    }
    ......
  }
}
```

```
namespace account {
  public class SavingAccount: account.BankAccount {
    ......
    private double __interest;
    ......
    public double Interest {
      set { this.__interest = value; }
      get { return __interest; }
    }
    ......
}
```

```
namespace account {
  public class CheckingAccount: account.BankAccount {
    ......
    public override string ToString() {
        ......
    }
    ......
}
```

Figure 17.28 - Mapping generalization

In the above example, the generalization hierarchy of BankAccount, CheckingAccount and SavingAccount maps to three ORM-Persistable .NET classes where the subclasses, CheckingAccount and SavingAccount extend their superclass, BankAccoount so as to inherit the BankAccount's attributes and methods.

Mapping Association and Multiplicity

An association specifies the semantic relationship between two classes in the object model with a role attached at one end. The role specifies the detailed behavior of a class associated in the association, including its role name, which indicates how the class acts in the association; and multiplicity, which specifies the number of links between each instance of the associated classes.

The role name of the association maps to an attribute which specifies how the class associates with its associated supplier class with respect to multiplicity. If a class has a multiplicity of many in an association, the class maps not only to an ORM persistent .NET class, but also to a collection class which supports manipulating the objects within a collection.

You are allowed to define the type of collection class for code generation. Meanwhile, List and Set are the default types for ordered and un-ordered collection respectively.

Example:

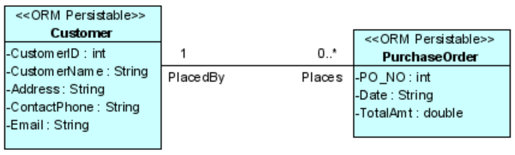
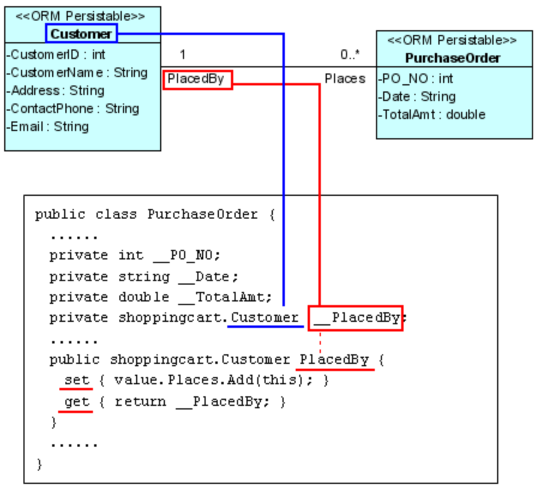


Figure 17.29 - Class Diagram with association and multiplicity

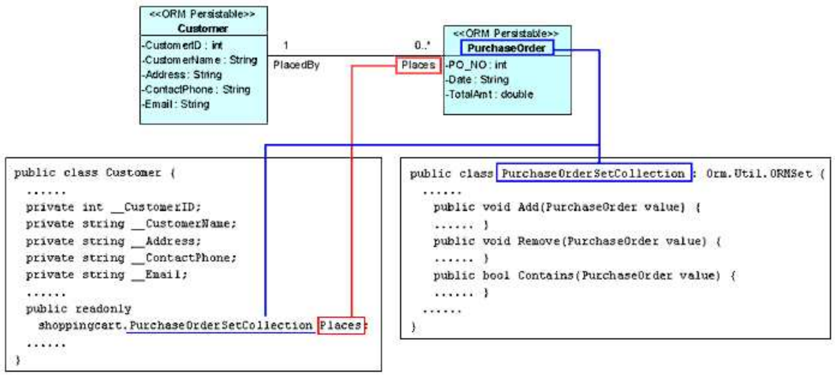
In the above object model, there is a one-to-many association between Customer class and PurchaseOrder class; defining that a customer can place many purchase order meanwhile a purchase order is placed by one customer. As object model maps to persistent code, three ORM-Persistable .NET classes will be generated, including Customer, PurchaseOrder and PurchaseOrderSetCollection classes.



```
public class PurchaseOrder {
  ......
  private int __PO_NO;
  private string __Date;
  private double __TotalAmt;
  private shoppingcart.Customer __PlacedBy;
  ......
  public shoppingcart.Customer PlacedBy {
    set { value.Places.Add(this); }
    get { return __PlacedBy; }
  }
  ......
}
```

Figure 17.30 - Mapping Association

From the above diagram, PurchaseOrder class has the role of "PlacedBy" in the association and it is associated with the Customer class with multiplicity of one. The role, "PlacedBy" maps as an attribute to the PurchaseOrder class which is defined as the data type of Customer class; it implies that the attribute "PlacedBy" stores a customer object which places the particular purchase order.



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private string __Address;
  private string __ContactPhone;
  private string __Email;
  ......
  public readonly
    shoppingcart.PurchaseOrderSetCollection Places;
  ......
}
```

```
public class PurchaseOrderSetCollection : Orm.Util.ORMSet {
  ......
    public void Add(PurchaseOrder value) {
    ...... }
    public void Remove(PurchaseOrder value) {
    ...... }
    public bool Contains(PurchaseOrder value) {
    ...... }
  ......
}
```

Figure 17.31 - Mapping Multiplicity

From the above diagram, Customer has the role of "Places" in the association which associates with many PurchaseOrder instances. As the PurchaseOrder class has the multiplicity of many in the association, a collection class is generated so as to manipulate the multiple instances of PurchaseOrder. The role, "Places" maps as an attribute to the Customer class with the data type of PurchaseOrderSetCollection class; it implies that the attribute "Places" represents a collection of PurchaseOrder objects which was placed by a particular customer.

Mapping Navigable Association

An association, which specifies a semantic relationship occurring between typed instances, has at least two ends represented by properties. When a property at an end of the association is owned by an end class, it represents the association is navigable from the opposite ends. On the contrary, when a property does not appear in the namespace of any associated class, it represents a non-navigable end of the association.

An open arrowhead on the end of an association indicates the end is navigable. Meanwhile, a small cross (x) on the end of an association indicates the end is not navigable. If an association has two navigable ends, it is considered as bi-directional associations.

As the role name of the association maps to an attribute which specifies how the class associates with its associated class, however if the association has a non-navigable end, the corresponding role name will not map to an attribute, suppressing the ownership of the class.

Example:

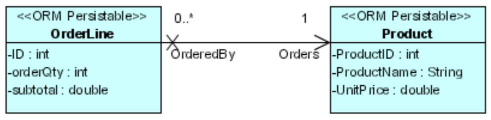
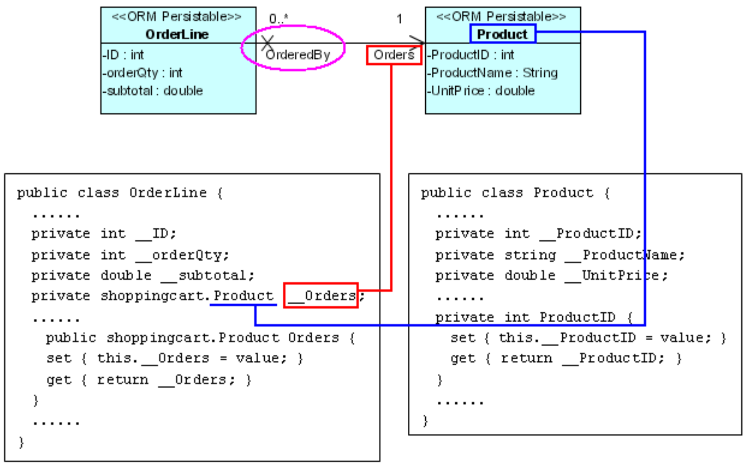


Figure 17.32 - Class Diagram with Navigable

In the above object model, it shows a binary association with one end navigable and the other non-navigable. It indicates that Product is owned by OrderLine while Product cannot own OrderLine. As the object model maps to persistent code, the ownership of OrderLine instance in Product class will not be generated in the persistent code.



```
public class OrderLine {
  ......
  private int __ID;
  private int __orderQty;
  private double __subtotal;
  private shoppingcart.Product __Orders;
  ......
    public shoppingcart.Product Orders {
    set { this.__Orders = value; }
    get { return __Orders; }
  }
  ......
}
```

```
public class Product {
  ......
  private int __ProductID;
  private string __ProductName;
  private double __UnitPrice;
  ......
  private int ProductID {
    set { this.__ProductID = value; }
    get { return __ProductID; }
  }
  ......
}
```

Figure 17.33 - Mapping with Navigable

From the above diagram, OrderLine class has the role of "Orders" in the association and associated with the one Product instance. The role, "Orders" maps as an attribute to OrderLine class which defines as the data type of Product; it implies that the attribute "Orders" stores a product instance.

On the other hand, Product class has the role of "OrderedBy" associated with many OrderLine instances in a non-navigable association; that is, Product class does not have the ownership of OrderLine. Even though the OrderLine class has the multiplicity of many in the association, the corresponding collection class of OrderLine for the Product class is not generated to manipulate it.

Enterprise JavaBeans (EJB)

Enterprise JavaBeans can be generated based on the Enterprise JavaBeans model, the generated code ensures that all developers produce consistent java persistence. Application development cycles are shortened using a code generation approach. As common database and application servers are supported, you are allowed to use the same Enterprise JavaBeans model to develop application with several supported application and database servers.

Generating Enterprise JavaBeans from Enterprise JavaBeans Model

You can generate EJB code from EJB Diagram.

You can obtain the EJB Diagram by drawing or from ERD. For more information, refer to the descriptions of Creating an EJB Diagram in Object Model chapter.

In order to generate Enterprise JavaBeans from the Enterprise JavaBeans model, you have to specify the class code detail for the models.

Specifying Entity Bean Code Detail

1. Right-click on Entity Bean, select **Open Specification...**.

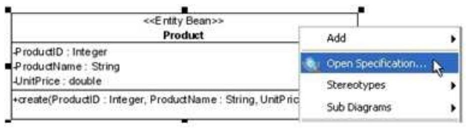
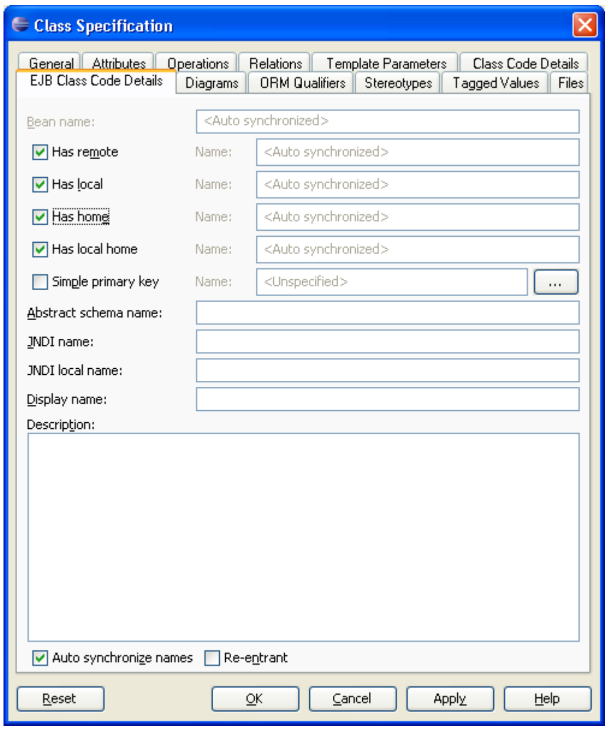


Figure 17.34 - To open specification

2. Click **EJB Class Code Details** Tab on the **Class Specification** dialog box.
3. Enter the code detail for the Entity Bean and click **OK** to confirm.



Bean name: <Auto synchronized>
☑ Has remote Name: <Auto synchronized>
☑ Has local Name: <Auto synchronized>
☑ Has home Name: <Auto synchronized>
☑ Has local home Name: <Auto synchronized>
☐ Simple primary key Name: <Unspecified> ...
Abstract schema name:
JNDI name:
JNDI local name:
Display name:
Description:
☑ Auto synchronize names ☐ Re-entrant

Figure 17.35 - Class Specification dialog (EJB Class Code Details)

Field	Description
Has remote	Generate the remote interface of the entity bean if the option is selected.
Has local	Generate the local interface of the entity bean if the option is selected.
Has home	Generate the home interface of the entity bean if the option is selected.
Has local home	Generate the local home interface of the entity bean if the option is selected.
Simple primary key	Disable the generation of primary key class if the option is selected; that is, the selected field will be used as a primary key of the entity bean.
Abstract schema name	The abstract schema name of the entity bean. The name must be a valid java identifier and unique. It is used by EJB-QL in CMP entity bean for retrieving data from the database.
JNDI name	The JNDI name of the entity bean which is used for the lookup process of the home interface.
JNDI local name	The local JNDI name of the entity bean which is used for the lookup process of the local home interface.
Display name	The short name of this entity bean.

Table 17.4

Specifying Message-Driven Bean Code Detail

1. Right-click on Message Driven Bean, select **Open Specification...**.

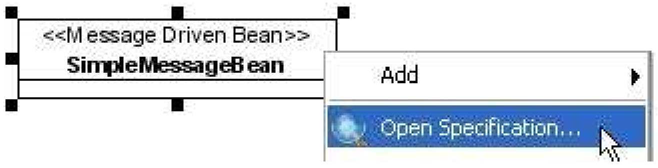
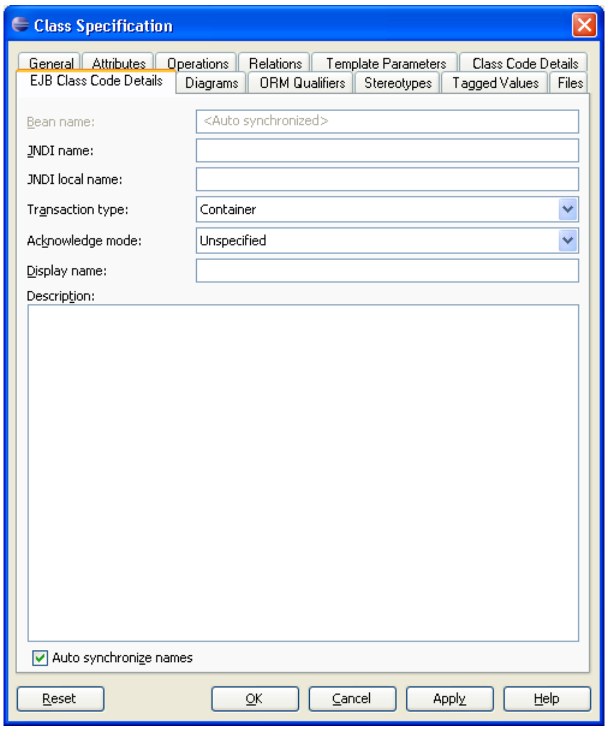


Figure 17.36 - To open specification for Message Driven Bean

2. Click **EJB Class Code Details** Tab on the **Class Specification** dialog box.
3. Enter the code detail for the Message Driven Bean and click **OK** to confirm.



Bean name: <Auto synchronized>
JNDI name:
JNDI local name:
Transaction type: Container
Acknowledge mode: Unspecified
Display name:
Description:
☑ Auto synchronize names

Figure 17.37 - Class Specification dialog (EJB Class Code Details tab)

Field	Description
JNDI name	The JNDI name of the entity bean which is used for the lookup process of the home interface.
JNDI local name	The local JNDI name of the entity bean which is used for the lookup process of the local home interface.
Transaction type	The transaction type of the message-driven bean.
Acknowledge mode	The acknowledge mode of the message-driven bean.
Display name	The short name of this message-driven bean.

Table 17.5

Specifying Session Bean Code Detail

1. Right-click on Session Bean and select **Open Specification...**.

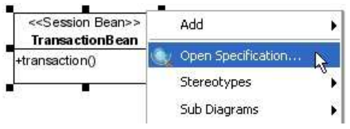
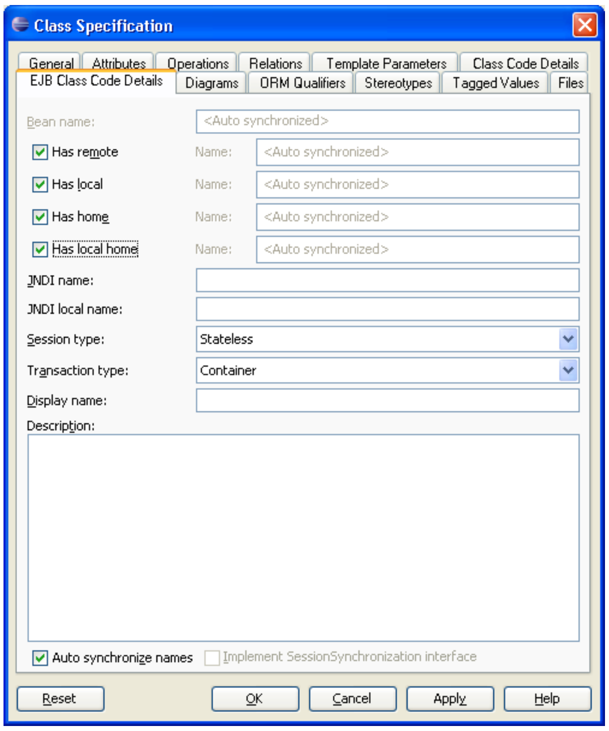


Figure 17.38 - Open specification for Session Bean

2. Click **EJB Class Code Details** Tab on the **Class Specification** dialog box.
3. Enter the code detail for the Session Bean and click **OK** to confirm.



Bean name: <Auto synchronized>
☑ Has remote Name: <Auto synchronized>
☑ Has local Name: <Auto synchronized>
☑ Has home Name: <Auto synchronized>
☑ Has local home Name: <Auto synchronized>
JNDI name:
JNDI local name:
Session type: Stateless
Transaction type: Container
Display name:
Description:
☑ Auto synchronize names ☐ Implement SessionSynchronization interface

Figure 17.39 - Class Specification Dialog (EJB Class Code Details tab)

Field	Description
Has remote	Generate the remote interface of the session bean if the option is selected.
Has local	Generate the local interface of the session bean if the option is selected.
Has home	Generate the home interface of the session bean if the option is selected.
Has local home	Generate the local home interface of the session bean if the option is selected.
JNDI name	The JNDI name of the entity bean which is used for the lookup process of the home interface.

JNDI local name	The local JNDI name of the entity bean which is used for the lookup process of the local home interface.
Session type	The type of the session bean, either Stateless or Stateful.
Transaction type	The transaction type of the session bean.
Display name	The short name of this session bean.

Table 17.6

Using Update Code

After specifying the code details for the Enterprise JavaBeans model, you can generate the beans and deployment descriptor.

1. Click the **Update Code** on toolbar.

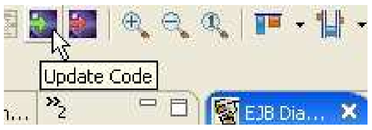
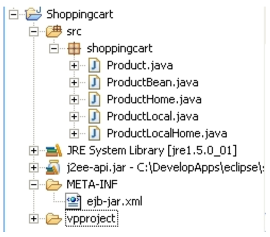


Figure 17.40 - Update the code

The Java class for beans and the deployment descriptor are generated. The Java class can be found under the package of your project and the deployment descriptor, **ejb-jar.xml** can be found inside the **META-INF** folder of your project.



```
Shoppingcart
  src
    shoppingcart
      Product.java
      ProductBean.java
      ProductHome.java
      ProductLocal.java
      ProductLocalHome.java
  JRE System Library [jre1.5.0_01]
  j2ee-api.jar - C:\DevelopApps\eclipse\
  META-INF
    ejb-jar.xml
  vpproject
```

Figure 17.41 - The generated files

Mapping Enterprise JavaBeans Model to Enterprise JavaBeans

The Enterprise JavaBeans is generated and supports mapping from Enterprise JavaBeans Model to Enterprise JavaBeans code. In this section, examples will be given to show the mapping between Enterprise JavaBeans model and Enterprise JavaBeans code.

Mapping Entity Bean

For each Entity Bean element which may have attributes and methods in the Enterprise JavaBeans model, a corresponding Java class will be generated. By configuring the EJB Class Code Details of Entity Bean, remote, local, home and local home interface can be generated. By default, the name of the interface will be generated as appending the type of interface to the name of the entity bean. You are allowed to assign the name for the interface.

Example:

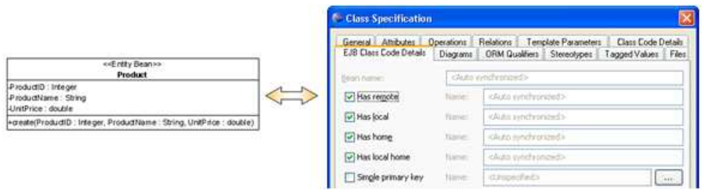
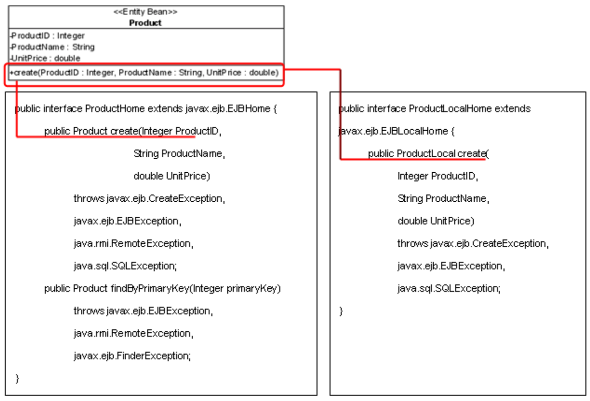

<<Entity Bean>>
Product
-ProductID : Integer
-ProductName : String
-UnitPrice : double
+create(ProductID : Integer, ProductName : String, UnitPrice : double)
Class Specification
General
Attributes
Operations
Relations
Template Parameters
Class Code Details
EJB Class Code Details
Diagrams
ORM Qualifiers
Stereotypes
Tagged Values
Files
Bean name: <Auto synchronized>
[✓] Has remote Name: <Auto synchronized>
[✓] Has local Name: <Auto synchronized>
[✓] Has home Name: <Auto synchronized>
[✓] Has local home Name: <Auto synchronized>
[ ] Single primary key Name: <Unspecified>

Figure 17.42 - The Class Configuration for Home/Local interface

In the above example, the Product Entity Bean model maps to a class ProductBean and four interface including Product, ProductLocal, ProductHome and ProductLocalHome.



```
<<Entity Bean>>
Product
-ProductID : Integer
-ProductName : String
-UnitPrice : double
+create(ProductID : Integer, ProductName : String, UnitPrice : double)
```

```
public interface ProductHome extends javax.ejb.EJBHome {
        public Product create(Integer ProductID,
                              String ProductName,
                              double UnitPrice)
            throws javax.ejb.CreateException,
            javax.ejb.EJBException,
            java.rmi.RemoteException,
            java.sql.SQLException;
        public Product findByPrimaryKey(Integer primaryKey)
            throws javax.ejb.EJBException,
            java.rmi.RemoteException,
            javax.ejb.FinderException;
}
```

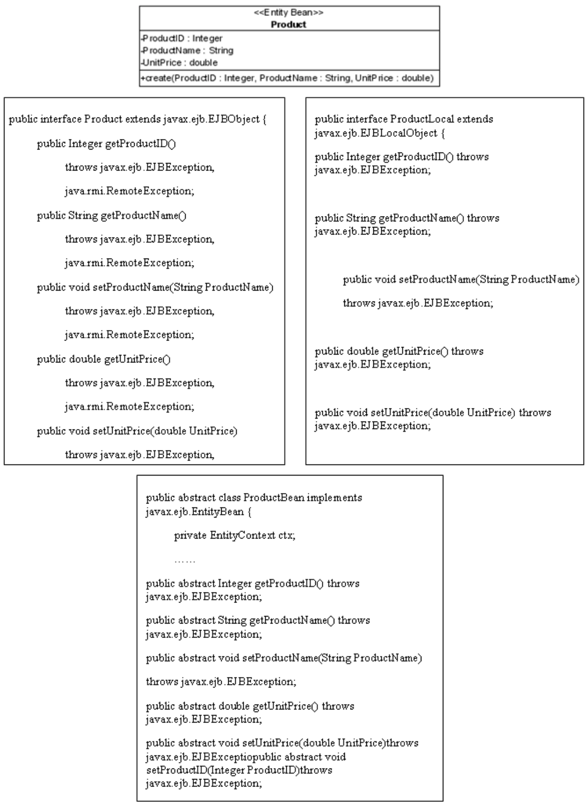
```
public interface ProductLocalHome extends
javax.ejb.EJBLocalHome {
        public ProductLocal create(
                Integer ProductID,
                String ProductName,
                double UnitPrice)
                throws javax.ejb.CreateException,
                javax.ejb.EJBException,
                java.sql.SQLException;
}
```

Figure 17.43 - Mapping the creator method

- Mapping Create Method

  In the above example, the create method of the Product entity bean model maps to the create method of ProductHome and ProductLocalHome interface which has the same signature of the model with the same parameters in order. (It is illustrated by the red line in the above diagram.)

  The ProductHome interface contains create and finder methods. The create method takes the Product value and primary key as parameters. When entity bean instantiates the home interface and calls its create method, the container create a ProductBean instance and calls its ejbCreate method. The ProductHome.create() and ProductHomeBean.ejbCreate() methods must have the same signatures, such that the product and primary key values can be passed from the home interface to the entity bean by the entity bean's container.



```
<<Entity Bean>>
Product
-ProductID : Integer
-ProductName : String
-UnitPrice : double
+create(ProductID : Integer, ProductName : String, UnitPrice : double)
```

```
public interface Product extends javax.ejb.EJBObject {
        public Integer getProductID()
                throws javax.ejb.EJBException,
                java.rmi.RemoteException;
        public String getProductName()
                throws javax.ejb.EJBException,
                java.rmi.RemoteException;
        public void setProductName(String ProductName)
                throws javax.ejb.EJBException,
                java.rmi.RemoteException;
        public double getUnitPrice()
                throws javax.ejb.EJBException,
                java.rmi.RemoteException;
        public void setUnitPrice(double UnitPrice)
                throws javax.ejb.EJBException,
```

```
public interface ProductLocal extends
javax.ejb.EJBLocalObject {
public Integer getProductID() throws
javax.ejb.EJBException;

public String getProductName() throws
javax.ejb.EJBException;

        public void setProductName(String ProductName)
        throws javax.ejb.EJBException;

public double getUnitPrice() throws
javax.ejb.EJBException;

public void setUnitPrice(double UnitPrice) throws
javax.ejb.EJBException;
```

```
public abstract class ProductBean implements
javax.ejb.EntityBean {
        private EntityContext ctx;
        ……
public abstract Integer getProductID() throws
javax.ejb.EJBException;
public abstract String getProductName() throws
javax.ejb.EJBException;
public abstract void setProductName(String ProductName)
throws javax.ejb.EJBException;
public abstract double getUnitPrice() throws
javax.ejb.EJBException;
public abstract void setUnitPrice(double UnitPrice)throws
javax.ejb.EJBExceptiopublic abstract void
setProductID(Integer ProductID)throws
javax.ejb.EJBException;
```

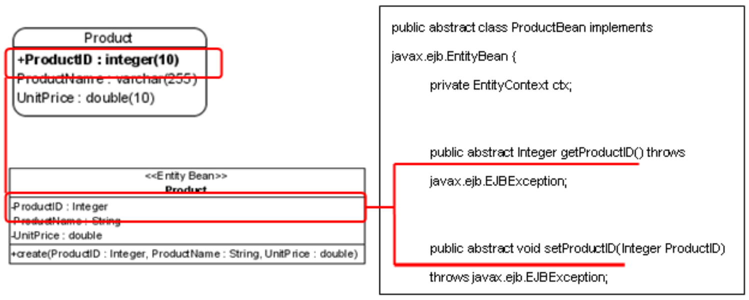
Figure 17.44 - The generated Class

  From the above example, the attributes of Product map to the attributes of the Product Bean. A pair of getter and setter methods will be generated in the Product interface, ProductLocal interface and ProduceBean. This pair of getter and setter methods of attributes allows the access to the Entity Bean attributes' value.

Mapping Primary Key

The primary key of the entity bean can be mapped to either an attribute of the generated entity bean, or a Primary Key class.

Example:



```
public abstract class ProductBean implements
javax.ejb.EntityBean {
        private EntityContext ctx;


        public abstract Integer getProductID() throws
        javax.ejb.EJBException;


        public abstract void setProductID(Integer ProductID)
        throws javax.ejb.EJBException;
```

Figure 17.45 - Mapping Primary Key

In the above example, the primary key of Product entity, ProductID maps to an attribute, ProductID of the entity bean, Product, which in turn becomes an attribute of the ProductBean class.

Mapping Association and Multiplicity

An association specifies the relationship between two entity bean classes in the EJB model with a role attached at one end. The role specifies the detailed behavior of an entity bean associated in the association, including its role name, which indicates how the entity bean class acts in the association; and multiplicity, which specifies the number of links between each instance of the associated classes.

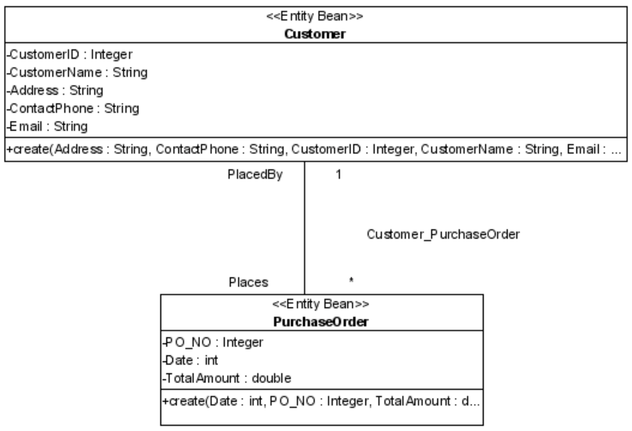
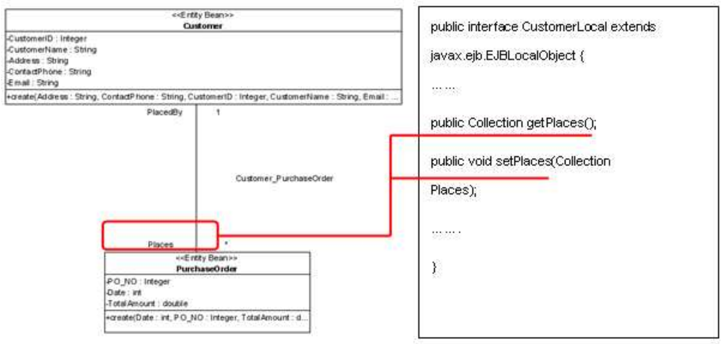


Figure 17.46 - Mapping Association and Multiplicity

In the above Enterprise JavaBeans model, there is a one-to-many association between Customer and PurchaseOrder entity beans, defining that a customer can place many purchase order meanwhile a purchase order is placed by one customer.



```
public interface CustomerLocal extends
javax.ejb.EJBLocalObject {
… …
public Collection getPlaces();
public void setPlaces(Collection
Places);
… … .
}
```

Figure 17.47 - Mapping the association

From the above diagram, Customer entity bean has the role of "Places" in the association which associates with many PurchaseOrder instances. As the PurchaseOrder entity has the multiplicity of many in the association, a collection type is generated so as to manipulate the multiple instances of PurchaseOrder. The role, "Places" maps as an attribute to the Customer class with the data type Collection class.

Mapping Navigable Association

An association, which specifies a semantic relationship occurring between typed instances, has at least two ends represented by properties. When a property at an end of the association is owned by an end class, it represents the association is navigable from the opposite ends. On the contrary, when a property does not appear in the namespace of any associated class, it represents a non-navigable end of the association.

An open arrowhead on the end of an association indicates the end is navigable. Meanwhile, a small cross (x) on the end of an association indicates the end is not navigable. If an association has two navigable ends, it is considered as bi-directional associations.

As the role name of the association maps to an attribute which specifies how the entity bean associates with its associated entity bean, however if the association has a non-navigable end, the corresponding role name will not map to an attribute, suppressing the ownership of the entity bean.

Example:

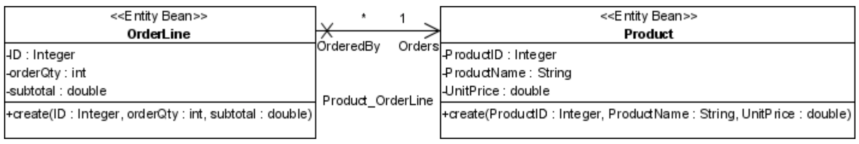
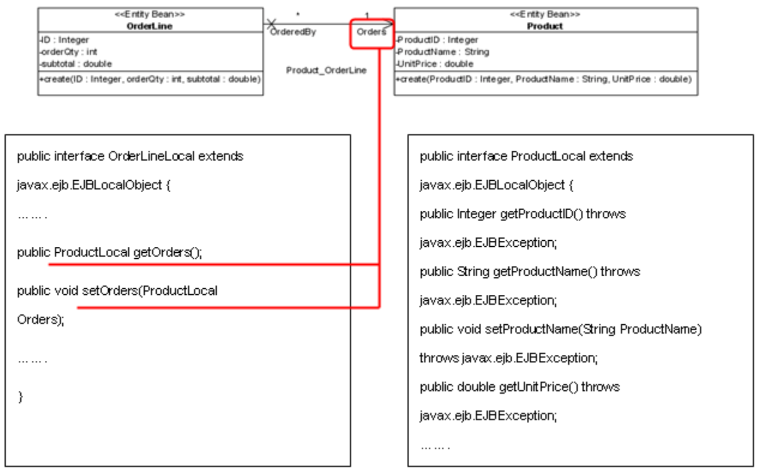

<<Entity Bean>>
OrderLine
-ID : Integer
-orderQty : int
-subtotal : double
+create(ID : Integer, orderQty : int, subtotal : double)
*
1
OrderedBy
Orders
Product_OrderLine
<<Entity Bean>>
Product
-ProductID : Integer
-ProductName : String
-UnitPrice : double
+create(ProductID : Integer, ProductName : String, UnitPrice : double)

Figure 17.48 - The Class Diagram with Navigable

In the above object model, it shows a binary association with one end navigable and the other non-navigable. It indicates that Product is owned by OrderLine while Product cannot own OrderLine. As the Enterprise JavaBeans model maps to Enterprise JavaBeans, the ownership of OrderLine instance in Product entity bean will not be generated in the bean code.


<<Entity Bean>>
OrderLine
-ID : Integer
-orderQty : int
-subtotal : double
+create(ID : Integer, orderQty : int, subtotal : double)
*
1
OrderedBy
Orders
Product_OrderLine
<<Entity Bean>>
Product
-ProductID : Integer
-ProductName : String
-UnitPrice : double
+create(ProductID : Integer, ProductName : String, UnitPrice : double)

```
public interface OrderLineLocal extends
javax.ejb.EJBLocalObject {
……. 
public ProductLocal getOrders();
public void setOrders(ProductLocal
Orders);
…….
}
```

```
public interface ProductLocal extends
javax.ejb.EJBLocalObject {
public Integer getProductID() throws
javax.ejb.EJBException;
public String getProductName() throws
javax.ejb.EJBException;
public void setProductName(String ProductName)
throws javax.ejb.EJBException;
public double getUnitPrice() throws
javax.ejb.EJBException;
…….
```

Figure 17.49 - Mapping with Navigable Association

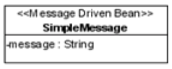
From the above diagram, OrderLineLocal interface has the role of "Orders" in the association and associated with the pair of getter and setter for ProductLocal instance. The role, "Orders" maps as an attribute to OrderLineLocal class which defines as the data type of ProductLocal; it implies that the attribute "Orders" stores a product local instance.

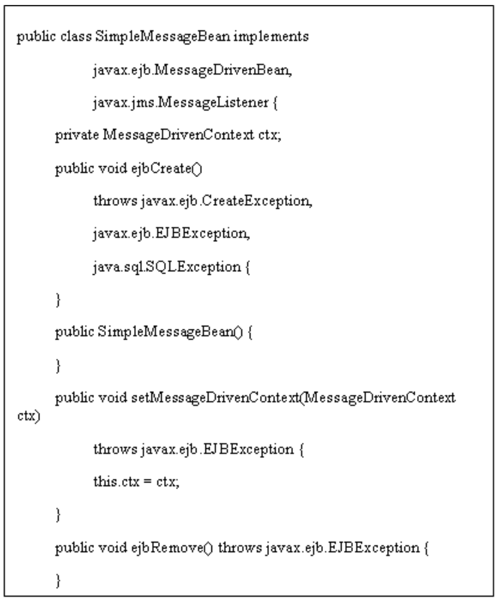
On the other hand, Product class has the role of "OrderedBy" associated with many OrderLine instances in a non-navigable association; that is, ProductLocal class does not have the ownership of OrderLine. Even though the OrderLine class has the multiplicity of many in the association, the corresponding collection class of OrderLine for the ProductLocal class is not generated to manipulate it.

Mapping Message-Driven Bean

The message-driven bean allows you to process the JMS message asynchronously. The message may be sent by a client application, another enterprise java bean, a web component, any other J2EE component or a JMS application. The main difference between the message-driven beans and entity beans and session beans is the client can access the message-driven bean directly.

Example:



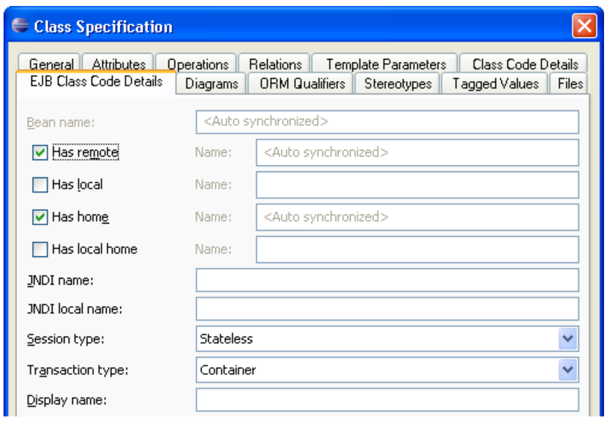


```
public class SimpleMessageBean implements
            javax.ejb.MessageDrivenBean,
            javax.jms.MessageListener {
      private MessageDrivenContext ctx;
      public void ejbCreate()
            throws javax.ejb.CreateException,
            javax.ejb.EJBException,
            java.sql.SQLException {
      }
      public SimpleMessageBean() {
      }
      public void setMessageDrivenContext(MessageDrivenContext
ctx)
            throws javax.ejb.EJBException {
            this.ctx = ctx;
      }
      public void ejbRemove() throws javax.ejb.EJBException {
      }
```

Figure 17.50 - Mapping Message Driven Bean

Mapping Session Bean

For each Session Bean element which may have attributes and methods in the Enterprise JavaBeans model, a corresponding Java class will be generated. By configuring the EJB Class Code Details of Session Bean, remote, local, home and local home interface can be generated. By default, the name of the interface will be generated as appending the type of interface to the name of session bean. You are allowed to assign the name of the interface.

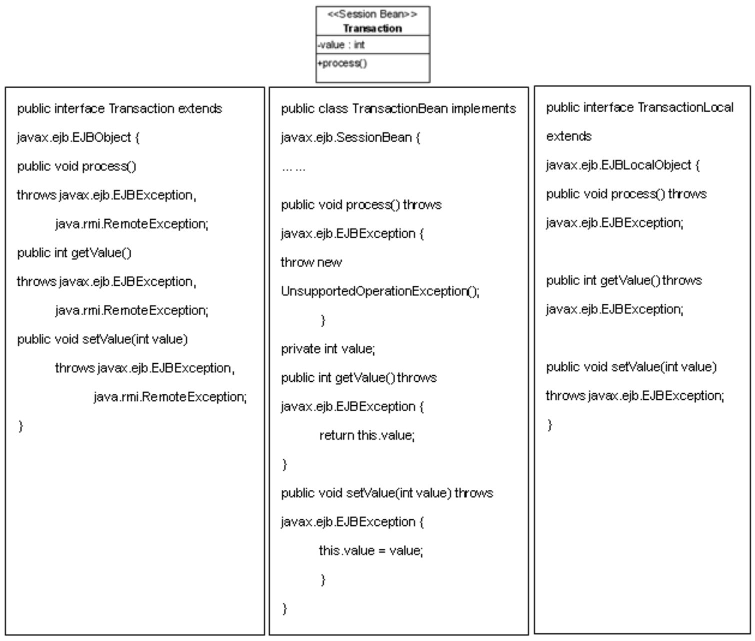


Bean name:		<Auto synchronized>
[✓] Has remote	Name:	<Auto synchronized>
[ ] Has local	Name:	
[✓] Has home	Name:	<Auto synchronized>
[ ] Has local home	Name:	
JNDI name:		
JNDI local name:		
Session type:		Stateless
Transaction type:		Container
Display name:		

Figure 17.51 - Class Specification (EJB Class Code Details)

Example:

The Session Bean, Transaction is selected to generate remote, local, home and local home.



```
public interface Transaction extends
javax.ejb.EJBObject {
public void process()
throws javax.ejb.EJBException,
        java.rmi.RemoteException;
public int getValue()
throws javax.ejb.EJBException,
        java.rmi.RemoteException;
public void setValue(int value)
        throws javax.ejb.EJBException,
                java.rmi.RemoteException;
}
```

```
public class TransactionBean implements
javax.ejb.SessionBean {
... ...

public void process() throws
javax.ejb.EJBException {
throw new
UnsupportedOperationException();
        }
private int value;
public int getValue() throws
javax.ejb.EJBException {
        return this.value;
}
public void setValue(int value) throws
javax.ejb.EJBException {
        this.value = value;
        }
}
```

```
public interface TransactionLocal
extends
javax.ejb.EJBLocalObject {
public void process() throws
javax.ejb.EJBException;

public int getValue() throws
javax.ejb.EJBException;

public void setValue(int value)
throws javax.ejb.EJBException;
}
```

Figure 17.52 - Mapping Session Bean

Deploying Enterprise JavaBeans on Application Servers

The deployment of Enterprise JavaBeans is supported on different application servers. A simple Application Server function provided to user to configure different kind of application server easily.

Configuring Application Servers

1. On the menu, click **Modeling** > **EJB** > **Application Server Configuration...**.

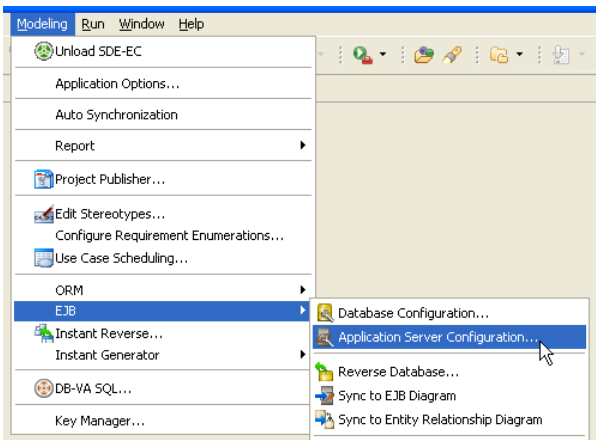


Figure 17.53 - Application Server Configuration

For other SDE:

SDE	Method
SDE for JBuilder	On the menu, click **Tools** > **Modeling** > **EJB** > **Application Server Configuration...**.
SDE for NetBeans	On the menu, click **Modeling** > **EJB** > **Application Server Configuration...**.
SDE for IntelliJ IDEA	On the menu, click **Modeling** > **EJB** > **Application Server Configuration...**.

SDE for JDeveloper	On the menu, click **Model > EJB > Application Server Configuration...**.
SDE for WebLogic Workshop	On the menu, click **Modeling > EJB > Application Server Configuration...**.

Table 17.7

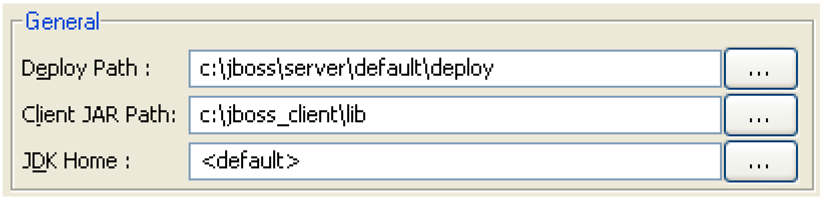
The **Application Server Configuration** dialog box is displayed.

2. Place a check mark beside the desired application servers for configuration.
3. Enter the general application server setting.

For the **Deploy Path**, click ... to specify the location to deploy the EJB application JAR file which will be run on the application server.

For the **Client JAR Path**, click ... to specify the location for the user to get the client JAR file to develop their client application which will be deployed to the application server.

For the **JDK Home**, click ... to specify the location of JDK Home which can be configured instead of using the default **JDK Home** in the system. By default, the value of the JDK Home is "**default**".



General

Deploy Path : c:\jboss\server\default\deploy

Client JAR Path: c:\jboss_client\lib

JDK Home : <default>

Figure 17.54 - The general deploy setting

4. Right-click on the desired server, select **Set as Default Application Server** if more than one application server is configured.

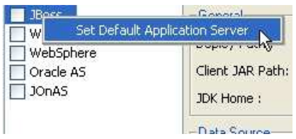
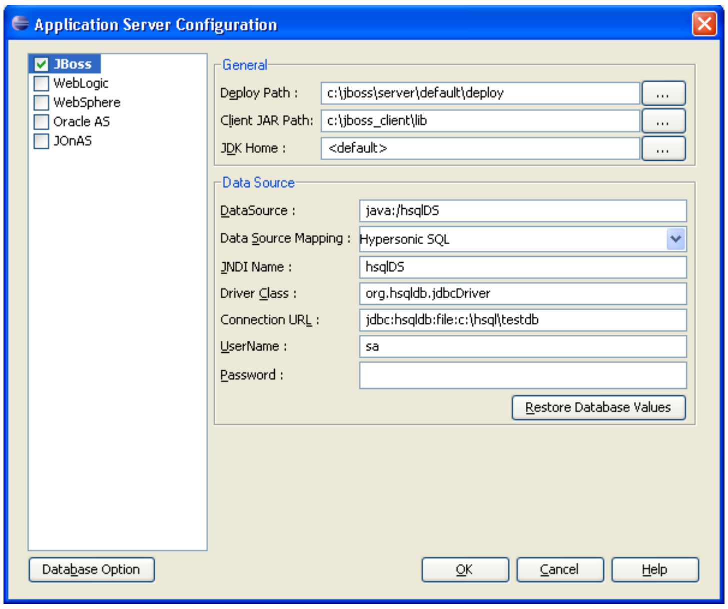


Figure 17.55 - Set the database as default

JBoss Application Server

Example:



Application Server Configuration

- [✓ JBoss]
- WebLogic
- WebSphere
- Oracle AS
- JOnAS

General

Deploy Path :	c:\jboss\server\default\deploy
Client JAR Path:	c:\jboss_client\lib
JDK Home :	<default>

Data Source

DataSource :	java:/hsqlDS
Data Source Mapping :	Hypersonic SQL
JNDI Name :	hsqlDS
Driver Class :	org.hsqldb.jdbcDriver
Connection URL :	jdbc:hsqldb:file:c:\hsql\testdb
UserName :	sa
Password :	

Figure 17.56 - JBoss application server configuration

Field	Description
DataSource	The data source.
Data Source Mapping	The type mapping.
JNDI Name	The JNDI name used for lookup this datasource.
Driver Class	The JDBC driver class.
Connection URL	The JDBC connection string.
UserName	The JDBC username.
Password	The JDBC password.

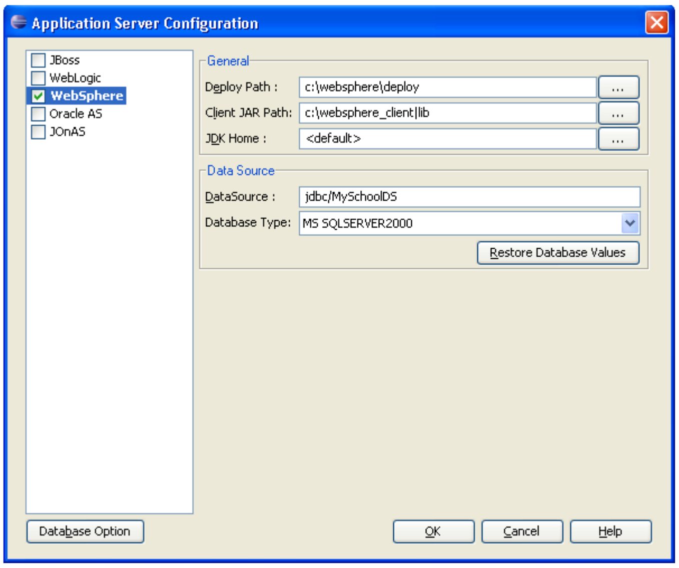
Table 17.8

For the JBoss Application Server configuration, you are asked to configure the Data Source settings including Driver Class, Connection URL, UserName and Password which can be configured with the **Database Option** button. The Database Configuration dialog box is displayed if the **Database Option** button is clicked. Refer to the descriptions in the Database Configuration chapter for information on how to configure database.

You are allowed to synchronize the setting between the JBoss Application Server and Database Configuration by using the **Restore Database Values** button.

IBM WebSphere Application Server

Example:


Application Server Configuration

- [ ] JBoss
- [ ] WebLogic
- [✓] WebSphere
- [ ] Oracle AS
- [ ] JOnAS

General

Deploy Path : c:\websphere\deploy

Client JAR Path: c:\websphere_client\lib

JDK Home : <default>

Data Source

DataSource : jdbc/MySchoolDS

Database Type: MS SQLSERVER2000

Figure 17.57 - WebSphere application server configuration

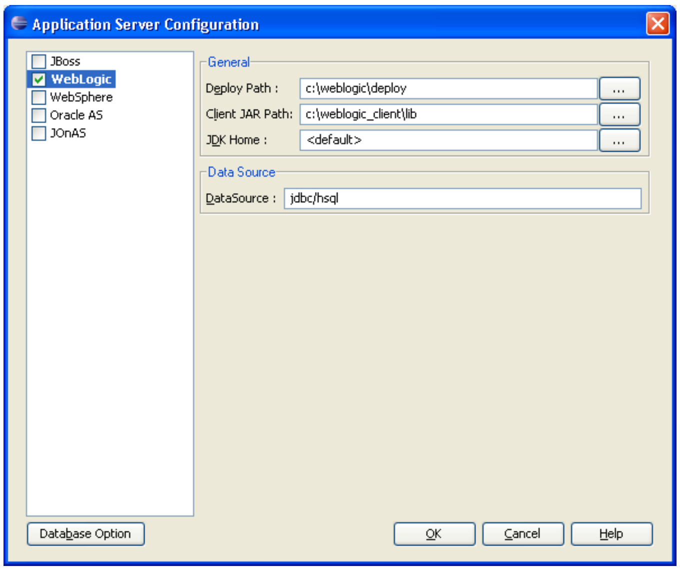
Field	Description
DataSource	The data source.
Database Type	The type of database.

Table 17.9

The IBM WebSphere Application Server requires you to specify the DataSource and Database Type for configuring the application server. You can select one of the database types supported.

BEA WebLogic Application Server

Example:


Application Server Configuration

- [ ] JBoss
- [✓] WebLogic
- [ ] WebSphere
- [ ] Oracle AS
- [ ] JOnAS

General

Deploy Path : c:\weblogic\deploy

Client JAR Path: c:\weblogic_client\lib

JDK Home : <default>

Data Source

DataSource : jdbc/hsql

Figure 17.58 - WebLogic Application Server configuration

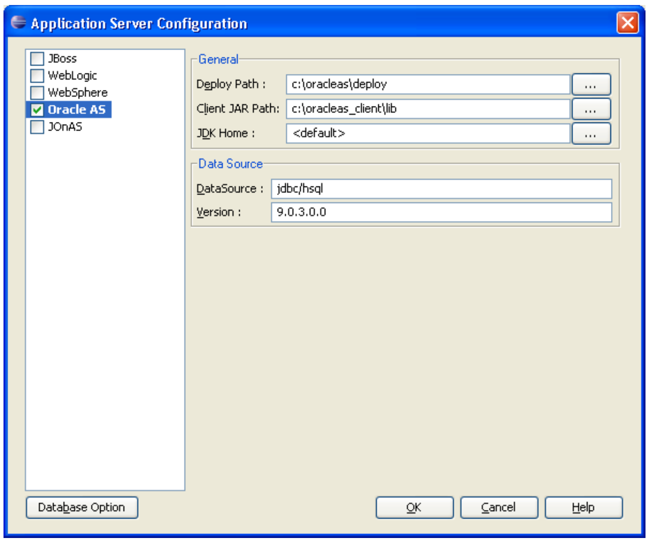
Field	Description
DataSource	The data source.

Table 17.10

The BEA WebLogic application server requires you to provide the DataSource for setting up the application server.

Oracle Application Server

Example:



Deploy Path : c:\oracleas\deploy
Client JAR Path: c:\oracleas_client\lib
JDK Home : <default>
DataSource : jdbc/hsql
Version : 9.0.3.0.0

Figure 17.59 - Oracle AS configuration

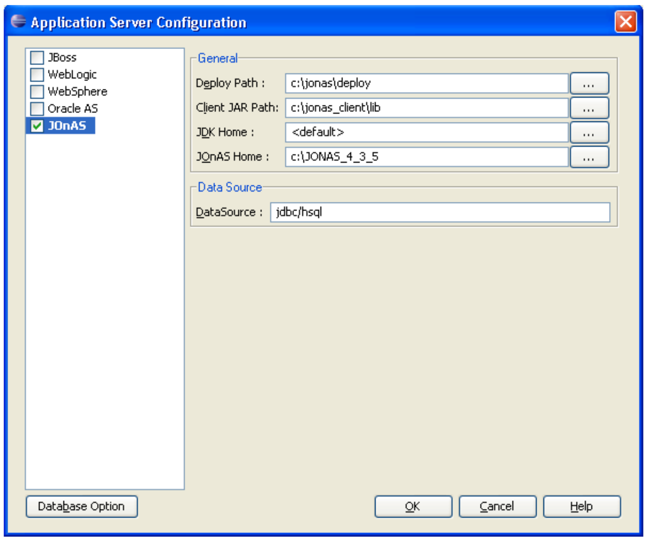
Field	Description
DataSource	The data source.
Version	The version of the Oracle Application Server.

Table 17.11

The Oracle application server requires you to provide the DataSource and Version for setting up the application server.

JOnAS Application Server

Example:



Deploy Path : c:\jonas\deploy
Client JAR Path: c:\jonas_client\lib
JDK Home : <default>
JOnAS Home : c:\JONAS_4_3_5
DataSource : jdbc/hsql

Figure 17.60 - JOnAS configuration

Field	Description
JOnAS Home	The install location of JOnAS application server.
DataSource	The data source.

Table 17.12

The JOnAS application server requires you to specify the installed location for the JOnAS application server and provide the DataSource for setting the application server.

After configure the application server, you can deploy the EJB code to your selected application server.

Deploying Beans on the Application Server

1. On the menu, click **Modeling** > **EJB** > **Deploy**, select one of the application servers which have been configured from the sub-menu.

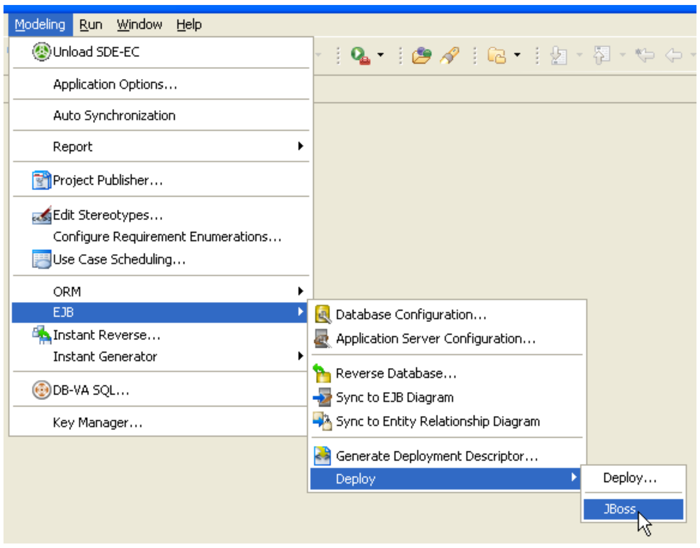


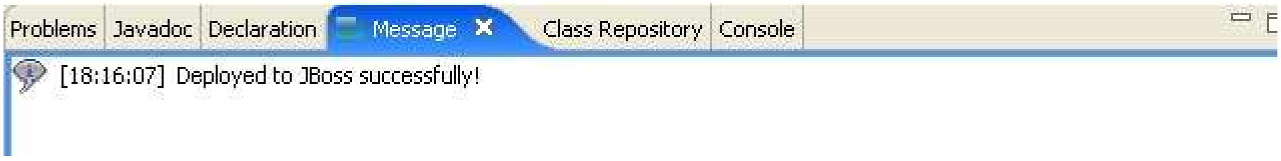
Figure 17.61 - Deploy to Application Server

For other SDE:

SDE	Method
SDE for JBuilder	On the menu, click **Tools** > **Modeling** > **EJB** > **Deploy**.
SDE for NetBeans	On the menu, click **Modeling** > **EJB** > **Deploy**.
SDE for IntelliJ IDEA	On the menu, click **Modeling** > **EJB** > **Deploy**.
SDE for JDeveloper	On the menu, click **Model** > **EJB** > **Deploy**.
SDE for WebLogic Workshop	On the menu, click **Modeling** > **EJB** > **Deploy**.

Table 17.13

After deploying the EJB application, a message will be displayed on the message pane showing the result of deployment.



[18:16:07] Deployed to JBoss successfully!

Figure 17.62 - Message Pane show the deploy message

Deploying Beans on WebSphere Application Sever

A backend database can be generated for deploying beans to WebSphere Application Server. The backend database helps you to perform Entity Bean to Database mapping for specified database type. In order to generate the backend database, you must ensure the EJB Diagram is synchronized with the ERD Diagram.

Example:

1. Synchronize the EJB Diagram to ERD Diagram.

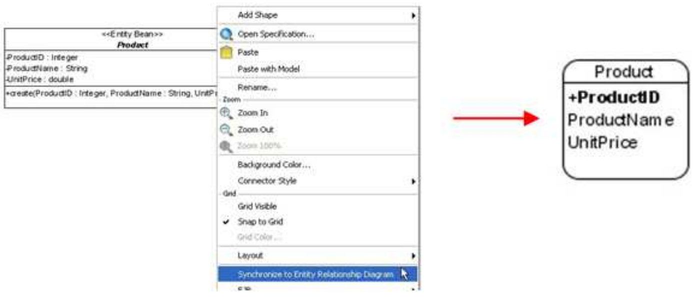


Figure 17.63 - Synchronize EJB Diagram to ERD

2. Deploy the project to WebSphere Application server.

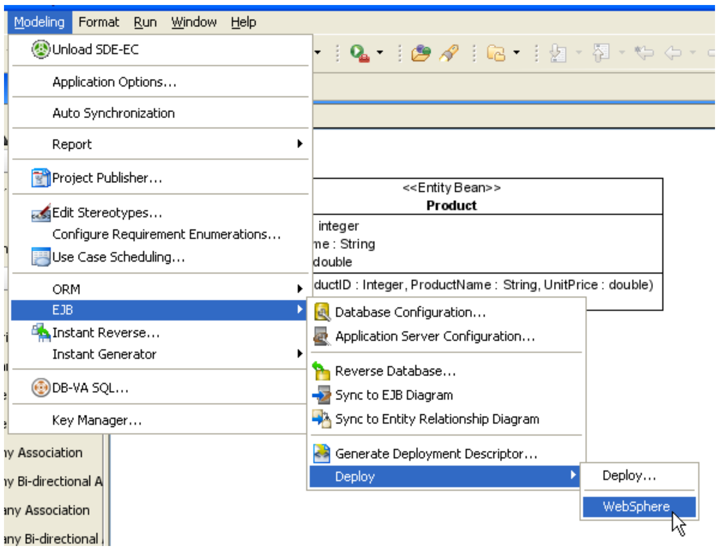


Figure 17.64 - Deploy EJB to Application Server

3. The backend database is generated and included in the deployment JAR file.

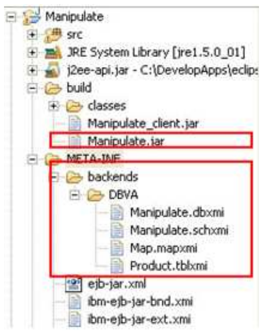


Figure 17.65 - The generated Server configuration files

Developing a Client Program

After deploying the EJB application to application server, you can write a client program with **Client JAR** file to test the application. A sample code for the client program is provided. The following example demonstrates how to get the deployed session bean from the application and instance to call the session bean information.

Example 1:

```
public staic void main(String[] args){
	Hashtable props = new Hashtable();
	props.put("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
	props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
	props.put("java.naming.provider.url", "jnp://localhost:1099");
	try{
		Context ctx = new InitialContext(props);
		FacadeSessionHome facadeSessionHome = (FacadeSessionHome)
		PortableRemoteObject.narrow(ctx.lookup("ejb/FacadeSession()"),
		FacadeSessionHome.class);
		FacadeSession facadeSession = facadeSessionHome.create();
		facadeSession.test();
	} catch (Exception e){
		e.printStackTrace();
	}
}
```

Example 2:

The following example shows a client program developed without specifying the environment parameter of the context. When executing the program, the following statement should be used:

```
java - Djava.naming.factory.initial = org.jnp.interfaces.NamingContextFactory -
Djava.naming.factory.url.pkgs = org.jboss.naming:org.jnp.interfaces -
Djava.naming.provider.url = jnp://localhost:1099 ...

public staic void main(String[] args){
	try{
		Context ctx = new InitialContext();
		FacadeSessionHome facadeSessionHome = (FacadeSessionHome)
		PortableRemoteObject.narrow(ctx.lookup("ejb/FacadeSession()"),
		FacadeSessionHome.class);
		FacadeSession facadeSession = facadeSessionHome.create();
		facadeSession.test();
	} catch (Exception e){
		e.printStackTrace();
	}
}
```

As the client program may connect to different application server, the parameter values for the context's environment are different. The following tables show the values to be used in different application server.

Parameter Value for Context Environment in JBoss Client Program

Environment Parameter	Parameter Value
java.naming.factory.initial	org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs	org.jboss.naming:org.jnp.interfaces
java.naming.provider.url	jnp://<hostname>:<port>
java.naming.security.principal	<user>
java.naming.security.credentials	<password>

Table 17.14

* port default 1099

Parameter Value for Context Environment in WebLogic Client Program

Environment Parameter	Parameter Value
java.naming.factory.initial	weblogic.jndi.WLInitialContextFactory
java.naming.provider.url	t3://<hostname>:<port>
java.naming.security.principal	<user>
java.naming.security.credentials	<password>

Table 17.15

* port default 7001

Parameter Value for Context Environment in WebSphere Client Program

Environment Parameter	Parameter Value
java.naming.factory.initial	com.ibm.websphere.naming.WsnInitialContextFactory
java.naming.provider.url	iiop://<hostname>:<port>
java.naming.security.principal	<user>
java.naming.security.credentials	<password>

Table 17.16

* port default 2809

Parameter Value for Context Environment in Oracle Client Program

Environment Parameter	Parameter Value
java.naming.factory.initial	com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url	ormi://<hostname>:<port>/<application>
java.naming.security.principal	<user>
java.naming.security.credentials	<password>

Table 17.17

* port default 3201

Parameter Value for Context Environment in JOnAS Client Program

s Environment Parameter	Parameter Value
java.naming.factory.initial	com.sun.jndi.rmi.registry.RegistryContextFactory
java.naming.provider.url	rmi://<hostname>:<port>
java.naming.security.principal	<user>
java.naming.security.credentials	<password>

Table 17.18

* port default 1099

18 Manipulating Persistent Data with Java

Chapter 18 - Manipulating Persistent Data with Java

Java and .NET persistent code can be generated to manipulate persistent data with the database. This chapter shows the use of the generated Java persistent code by inserting, retrieving, updating and deleting persistent data and demonstrates how to run the generated sample code.

In this chapter:

- Introduction
- Using ORM-Persistable Class to manipulate persistent data
- Using Criteria Class to retrieve persistent data
- Using ORM Implementation
- Using Entity Bean to manipulate persistent data
- Using Transactions in Enterprise JavaBeans

Introduction

With the Smart Development Environment Enterprise Edition (SDE EE) and Professional Edition (SDE PE), you can generate Java persistence code to manipulate the persistent data of the relational database easily.

In the working environment, you are allowed to configure the database connection for your development project. As the database was configured before the generation of persistent code, the ORM persistable Java classes and a set of ORM files are generated together; while the set of ORM files configures the environment for connecting the generated persistence classes and the database. And hence, you are allowed to access the database directly by using the Java persistence class without using any code for setting up the database connection in your project.

There are several mappings in the SDE environment:

1. Mapping between data model and relational database.
2. Mapping between data model and object model.
3. Mapping between object model and persistent code.

And hence, there is an indirect mapping between persistent code and relational database. Each persistence class represents a table in the database and an instance of the class represents one record of the table. In the generated persistence class, there is not only a pair of getter and setter methods for manipulating the corresponding attribute, but also a set of methods for manipulating records with the database.

Using ORM-Persistable Class

Java ORM-Persistable class is generated based on the object model defined in the class diagram. The generated Java persistence code can be identified into two categories - Model API and Persistent API. The Model API refers to the manipulation of attributes of models and associations between models while the Persistent API refers to the persistent code used to manipulate the persistent data with relational database.

Model API

Model API refers to the generated Java persistent code which is capable of manipulating the properties of the object model in terms of attributes and associations.

Manipulating Attributes

In order to specify and retrieve the value to the attributes of the ORM-Persistable class, a pair of getter and setter methods for each attribute is generated to the Java ORM-Persistable class.

Table shows the method summary of the persistence class to be used for inserting a row to database.

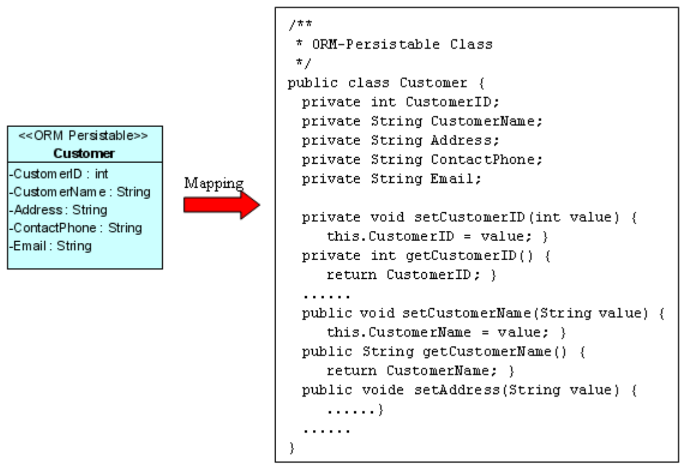
Return Type	Method Name	Description
void	setAttribute(DataType value)	Set the value to the property of an instance.
DataType	getAttribute()	Get the value of the property of an instance.

Table 18.1

Remark:

1. **Attribute** should be replaced by the name of the generated persistence class.
2. **DataType** should be replaced by the data type of the attribute defined in the object model.

Example:



```
/**
 * ORM-Persistable Class
 */
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;

  private void setCustomerID(int value) {
     this.CustomerID = value; }
  private int getCustomerID() {
     return CustomerID; }
  ......
  public void setCustomerName(String value) {
     this.CustomerName = value; }
  public String getCustomerName() {
     return CustomerName; }
  public voide setAddress(String value) {
     ......}
  ......
}
```

Figure 18.1 - ORM Persisteable Class mapping

From the above example, an ORM-Persistable object model of Customer maps to an ORM-Persistable Java class generated with the getter and setter methods for each attribute.

To set the values of properties of the Customer object, the following lines of code should be used.

```
customer.setCustomerName("Joe Cool");
customer.setAddress("1212, Happy Building");
customer.setContactPhone("23453256");
customer.setEmail("joe@cool.com");
```

To get the values of properties of the Customer object:

```
String name = customer.getCustomerName();
String address = customer.getAddress();
String phone = customer.getContactPhone();
String email = customer.getEmail();
```

Manipulating Association

When mapping a navigable association to persistence code, the role name of the association will become an attribute of the class. A pair of getter and setter methods for the role will be generated to the persistence class so as to manipulate its role associated with its supplier class. There are two ways in manipulating association, including Smart Association Handling and Standard Association Handling.

Smart Association Handling

Using smart association handling, the generated persistent code is capable of defining one end of the association which updates the other end of association automatically in a bi-directional association regardless of multiplicity. Examples are given to show how the generated persistent code manipulates the one-to-one, one-to-many and many-to-many associations with smart association handling.

One-to-One Association

In order to manipulate the directional association, implement the program with the following steps:

1. Set the properties of role name of the object.

Table shows the method summary of the persistence class to be used for setting the directional association.

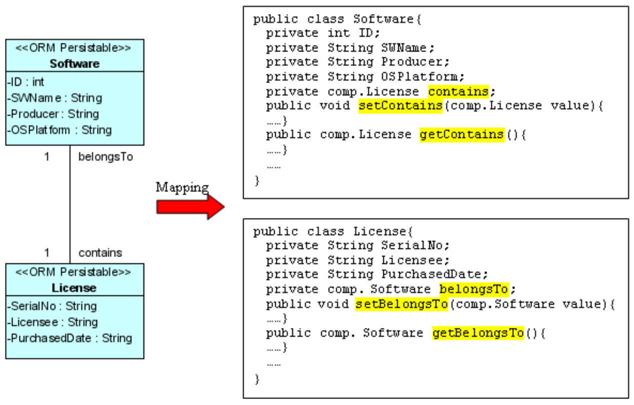
Return Type	Method Name	Description
void	setRoleName(Class value)	Set the value to the role of an instance.

Table 18.2

Remark:

1. **RoleName** should be replaced by the role of the navigable association.
2. **Class** should be replaced by the data type of the supplier class in the navigable association.

Example:



```
public class Software{
  private int ID;
  private String SWName;
  private String Producer;
  private String OSPlatform;
  private comp.License contains;
  public void setContains(comp.License value){
  ……}
  public comp.License getContains(){
  ……}
  ……
}
```

```
public class License{
  private String SerialNo;
  private String Licensee;
  private String PurchasedDate;
  private comp. Software belongsTo;
  public void setBelongsTo(comp.Software value){
  ……}
  public comp. Software getBelongsTo(){
  ……}
  ……
}
```

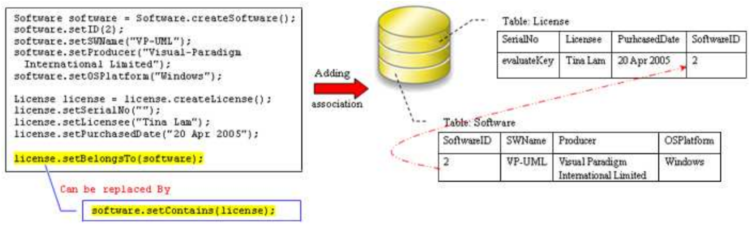
Figure 18.2 - Mapping one-to-one association

From the above example, an ORM-Persistable object model of Software maps to an ORM-Persistable Java class with an attribute of role, "contains" in the association typed as its associated persistence class, License. Meanwhile, the object model of License maps to a Java class with an attribute of role, "belongsTo" typed as Software. It specifies the association that Software contains a particular License and a License belongs to particular software.

To manipulate the association, the following lines of code should be implemented.

```
Software software = Software.createSoftware();
License license = License.createLicense();
license.setBelongsTo(software);
```

In the above association, the primary key of the Software table will become a foreign key in License table when transforming to data model. Thus, after executing the lines of code, the primary key of the software will be referenced as a foreign key of the license record.



```
Software software = Software.createSoftware();
software.setID(2);
software.setSWName("VP-UML");
software.setProducer("Visual-Paradigm
  International Limited");
software.setOSPlatform("Windows");

License license = license.createLicense();
license.setSerialNo("");
license.setLicensee("Tina Lam");
license.setPurchasedDate("20 Apr 2005");

license.setBelongsTo(software);
```

```
software.setContains(license);
```

Table: License

SerialNo	Licensee	PurhcasedDate	SoftwareID
evaluateKey	Tina Lam	20 Apr 2005	2

Table: Software

SoftwareID	SWName	Producer	OSPlatform
2	VP-UML	Visual Paradigm International Limited	Windows

Figure 18.3 - Code for one-to-one association

It is a bi-directional association with smart association handling. The association link can be created by setting either the role of Software, or the role of License, i.e. setting one of the roles builds up a bi-directional association automatically. It is not required to set one role after the other. Hence, both license.setBelongsTo(software) and software.setContains(license) result in building up the bi-directional association.

One-to-Many Association

In a bi-directional one-to-many association, a class has multiplicity of one while the other has many. If the class has multiplicity of many, the corresponding collection class will be automatically generated for manipulating the objects. When transforming the association to persistent code, the role name will map to an attribute with data type of a collection class. For more detailed information on the collection class, refer to the description in Using Collection section.

A bi-directional one-to-many association is shown below.

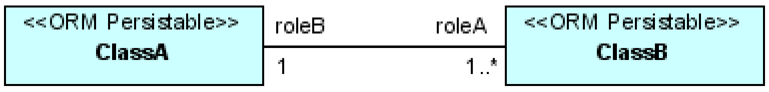


Figure 18.4 - One-to-Many Association

To manipulate the bi-directional one-to-many association, you can create the association in one of the two ways:

- Set the properties of role with an instance of the associated class; i.e. classB.setRoleB(classA)
- Add the objects to the collection of the associated class; i.e. classA.roleA.add(classB)

where classA is an object of ClassA; classB is an object of ClassB;
roleA is the collection of ClassB; setRoleB is the setter method of property, roleB.

After specifying the association, the value of primary key of the object of ClassA will be referenced as a foreign key of the associated object of ClassB.

Setting the property of role

For information on setting the properties of role, refer to the description in the One-to-One Association section.

Adding objects to the collection

In order to add an object to the collection, implement the program with the following steps:

1. Create a new instance of the class which associates with more than one instance of associated class.
2. Create a new object of the associated class.
3. Add a new object or an existing object of the associated class to the collection belonging to the class.

Table shows the method summary of the collection class to be used for adding a new object to it

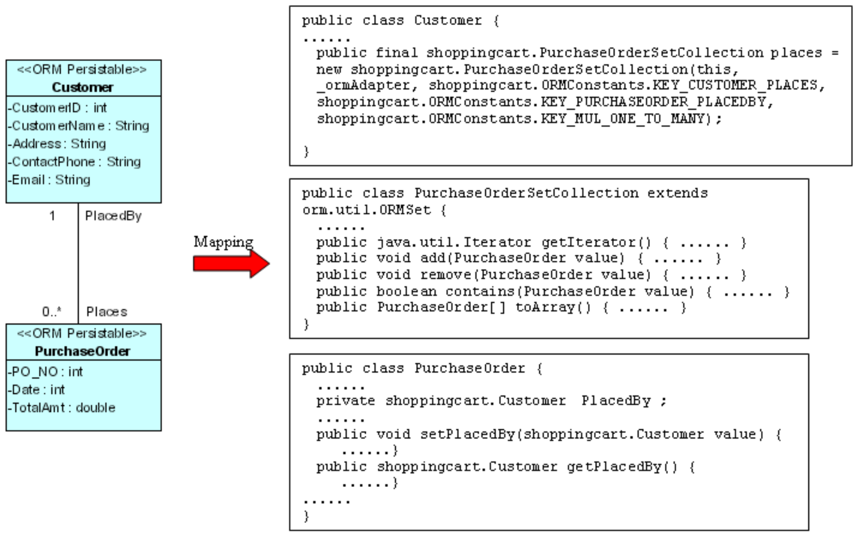
Return Type	Method Name	Description
void	add(Class value)	Add a new object to the collection of the associated class.

Table 18.3

Remark:

1. **Class** should be replaced by the name of the associated class.

Example:



```
public class Customer {
......
  public final shoppingcart.PurchaseOrderSetCollection places =
  new shoppingcart.PurchaseOrderSetCollection(this,
  _ormAdapter, shoppingcart.ORMConstants.KEY_CUSTOMER_PLACES,
  shoppingcart.ORMConstants.KEY_PURCHASEORDER_PLACEDBY,
  shoppingcart.ORMConstants.KEY_MUL_ONE_TO_MANY);

}
```

```
public class PurchaseOrderSetCollection extends
orm.util.ORMSet {
  ......
  public java.util.Iterator getIterator() { ...... }
  public void add(PurchaseOrder value) { ...... }
  public void remove(PurchaseOrder value) { ...... }
  public boolean contains(PurchaseOrder value) { ...... }
  public PurchaseOrder[] toArray() { ...... }
}
```

```
public class PurchaseOrder {
  ......
  private shoppingcart.Customer  PlacedBy ;
  ......
  public void setPlacedBy(shoppingcart.Customer value) {
     ......}
  public shoppingcart.Customer getPlacedBy() {
     ......}
......
}
```

Figure 18.5 - One-to-many association

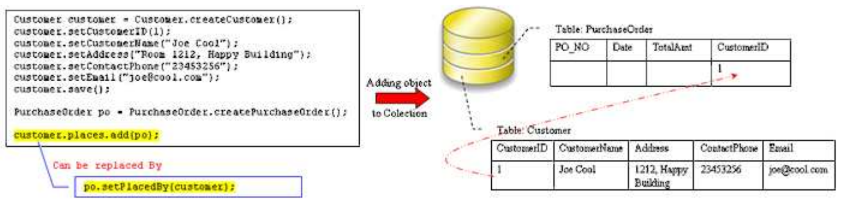
From the above example, an ORM-Persistable object model of PurchaseOrder maps to an ORM-Persistable Java class with an attribute of role, "PlacedBy" in the association typed as an instance of Customer, specifying that the particular purchase order is placed by a particular customer. Moreover, the object model of Customer maps to a Java class with an attribute of role, "Places" in the association typed as a collection class, PurchaseOrderSetCollection which manipulates instances of PurhcaseOrder.

To add a PurchaseOrder object to the collection of PurchaseOrder, the following lines of code should be implemented.

```
Customer customer = Customer.createCustomer();
PurchaseOrder po = PurchaseOrder.createPurchaseOrder();
customer.places.add(po);
```

After executing these lines of code, an object is added to the collection representing the association. When inserting records to the database tables, the primary key of the customer will be referenced to as the foreign key of the purchase order record.

The alternative way to create the association is using po.setPlacedBy(customer);



```
Customer customer = Customer.createCustomer();
customer.setCustomerID(1);
customer.setCustomerName("Joe Cool");
customer.setAddress("Room 1212, Happy Building");
customer.setContactPhone("23453256");
customer.setEmail("joe@cool.com");
customer.save();

PurchaseOrder po = PurchaseOrder.createPurchaseOrder();

customer.places.add(po);
```

Can be replaced By

```
po.setPlacedBy(customer);
```

Table: PurchaseOrder

PO_NO	Date	TotalAmt	CustomerID
			1

Table: Customer

CustomerID	CustomerName	Address	ContactPhone	Email
1	Joe Cool	1212, Happy Building	23453256	joe@cool.com

Figure 18.6 - Code for one-to-many association

Retrieving objects from the collection

In order to retrieve an object from the collection, implement the program with the following steps:

1. Retrieve an instance of the class which associates with more than one instance of associated class.
2. Get the collection from the class, and convert the collection into an array.

Table shows the method summary of the collection class to convert the collection into an array.

Return Type	Method Name	Description
Class[]	toArray()	Convert the collection into an array which stores the objects of the associated class.

Table 18.4

Remark:

1. **Class** should be replaced by the name of the associated class.

Example:

Refer to the example of the one-to-many association between the ORM-Persistable object models of Customer and PurchaseOrder, implement the following lines of code to retrieve a collection of PurchaseOrder objects from the customer.

```
Customer customer = Customer.loadByName("Joe Cool");
PurchaseOrder[] orders = customer.places.toArray();
```

After executing these lines of code, the purchase order records associated with the customer are retrieved and stored in the array.

Retrieve the customer record by using the loadByName() method which is the method provided by the persistent API. For more information on retrieving persistent object, refer to the Persistent API section.

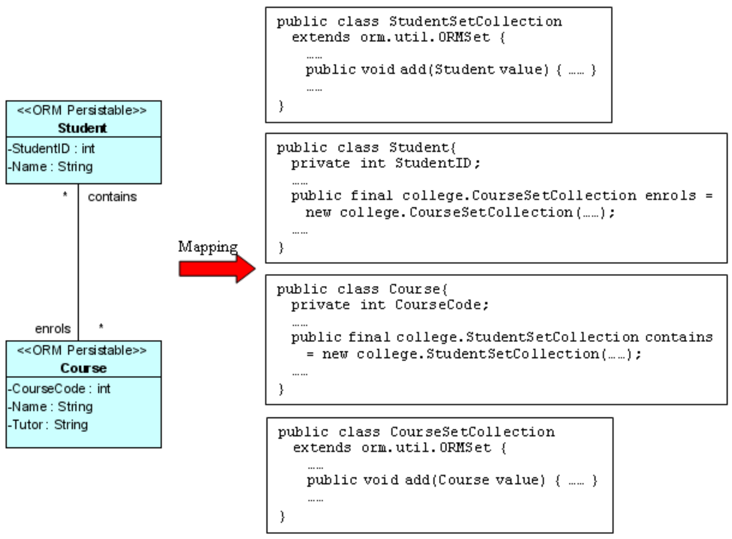
Many-to-Many Association

When transforming a many-to-many association between two ORM-Persistable classes, the corresponding persistent and collection Java class will be generated simultaneously such that the collection class is able to manipulate its related objects within the collection.

In order to specify the many-to-many association, add the objects to the corresponding collection of the associated class. For information on adding objects to the collection, refer to the description in the One-to-Many Association section.

In addition, a many-to-many association in the object model is transformed to data model by generating a Link Entity to form two one-to-many relationships between two generated entities. The primary keys of the two entities will migrate to the link entity as the primary/foreign keys. When specifying the association by adding objects to the collection, the primary key of the two related instances will be inserted as a pair for a row of the Link Entity automatically.

Example:



```
public class StudentSetCollection
  extends orm.util.ORMSet {
    ……
    public void add(Student value) { …… }
    ……
}
```

```
public class Student{
  private int StudentID;
  ……
  public final college.CourseSetCollection enrols =
    new college.CourseSetCollection(……);
    ……
}
```

```
public class Course{
  private int CourseCode;
  ……
  public final college.StudentSetCollection contains
    = new college.StudentSetCollection(……);
  ……
}
```

```
public class CourseSetCollection
  extends orm.util.ORMSet {
    ……
    public void add(Course value) { …… }
    ……
}
```

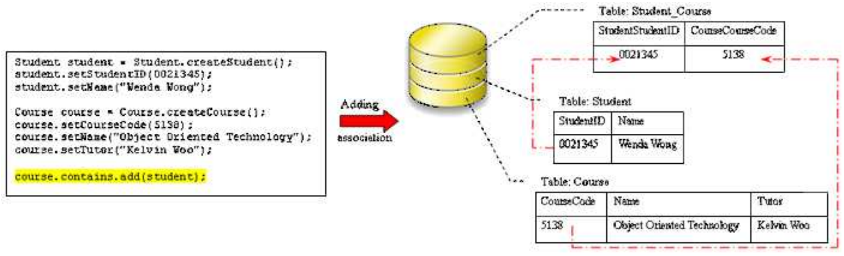
Figure 18.7 - Mapping many-to-many association

Four classes including Student, StudentSetCollection, Course and CourseSetCollection are generated from the above object model.

By executing the following lines of code:

```
Student student = Student.createStudent();
student.setStudentID(0021345);
student.setName("Wenda Wong");
Course course = Course.createCourse();
course.setCourseCode(5138);
course.setName("Object Oriented Technology");
course.setTutor("Kelvin Woo");
course.contains.add(student);
```

Both the Student table and Course table are inserted with a new record. After specifying the association by course.contains.add(student) the corresponding primary keys of student record and course record migrate to the Student_Course Link Entity to form a row of record.



```
Student student = Student.createStudent();
student.setStudentID(0021345);
student.setName("Wenda Wong");

Course course = Course.createCourse();
course.setCourseCode(5138);
course.setName("Object Oriented Technology");
course.setTutor("Kelvin Woo");

course.contains.add(student);
```

Table: Student_Course

StudentStudentID	CourseCourseCode
0021345	5138

Table: Student

StudentID	Name
0021345	Wenda Wong

Table: Course

CourseCode	Name	Tutor
5138	Object Oriented Technology	Kelvin Woo

Figure 18.8 - Code for many-to-many association

Using Collection

A collection represents a group of objects. Some collections allow duplicate objects and others do not. Some collections are ordered and other unordered. A collection class thus supports the manipulation of the objects within a collection. There are four types of collection class supported, including set, bag, list and map.

The type of collection can be specified in advance of generating persistence code. Refer to Specifying Collection Type in the Object Model chapter for more information.

Set

Set is an unordered collection that does not allow duplication of objects. It is the default type for unordered collection.

Table shows the method summary of the collection class typed as Set.

Return Type	Method Name	Description
void	add(Class value)	Add the specified persistent object to this set if it is not already present.
void	clear()	Remove all of the persistent objects from this set.
boolean	contains(Class value)	Return true if this set contains the specified persistent object.
Iterator	getIterator()	Return an iterator over the persistent objects in this set.
boolean	isEmpty()	Return true if this set contains no persistent object.
void	remove(Class value)	Remove the specified persistent object from this set if it is present.
int	size()	Return the number of persistent objects in this set.
Class[]	toArray()	Return an array containing all of the persistent objects in this set.

Table 18.5

Remark:

1. **Class** should be replaced by the persistence class.

Bag

Bag is an unordered collection that may contain duplicate objects.

Table shows the method summary of the collection class typed as Bag.

Return Type	Method Name	Description
void	add(Class value)	Add the specified persistent object to this bag.
void	clear()	Remove all of the persistent objects from this bag.
boolean	contains(Class value)	Return true if this bag contains the specified persistent object.
Iterator	getIterator()	Return an iterator over the persistent objects in this bag.
boolean	isEmpty()	Return true if this bag contains no persistent object.
void	remove(Class value)	Remove the specified persistent object from this bag.
int	size()	Return the number of persistent objects in this bag.
Class[]	toArray()	Return an array containing all of the persistent objects in this bag.

Table 18.6

Remark:

1. **Class** should be replaced by the persistence class.

List

List is an ordered collection that allows duplication of objects. It is the default type for ordered collection.

Table shows the method summary of the collection class typed as List.

Return Type	Method Name	Description
void	add(Class value)	Append the specified persistent object to the end of this list.
void	add(int index, Class value)	Insert the specified persistent object at the specified position in this list.
void	clear()	Remove all of the persistent objects from this list.
boolean	contains(Class value)	Return true if this list contains the specified persistent object.
Class	get(int index)	Return the persistent object at the specified position in this list.
Iterator	getIterator()	Return an iterator over the persistent objects in this list in proper sequence.

boolean	isEmpty()	Return true if this list contains no persistent object.
void	remove(Class value)	Remove the first occurrence in this list of the specified persistent object.
Class	remove(int index)	Remove the persistent object at the specified position in this list.
int	set(int index, Class value)	Replace the persistent object at the specified position in this list with the specified persistent object.
int	size()	Return the number of persistent objects in this list.
Class[]	toArray()	Return an array containing all of the persistent objects in this list in proper sequence.

Table 18.7

Remark:

1. **Class** should be replaced by the persistence class.

Map

Map is an ordered collection which is a set of key-value pairs while duplicate keys are not allowed.

Table shows the method summary of the collection class typed as Map.

Return Type	Method Name	Description
void	add(Object key, Class value)	Add the specified persistent object with the specified key to this map.
void	clear()	Remove all mappings from this map.
boolean	contains(Object key)	Return true if this map contains a mapping for the specified key.
Class	get(Object key)	Return the persistent object to which this map maps the specified key.
Iterator	getIterator()	Return an iterator over the persistent objects in this map.
Iterator	getKeyIterator()	Return an iterator over the persistent objects in this map.
boolean	isEmpty()	Return true if this map contains no key-value mappings.
void	remove(Object key)	Remove the mapping for this key from this map if it is present.
int	size()	Return the number of key-value mappings in this map.
Class[]	toArray()	Return an array containing all of the persistent objects in this map.

Table 18.8

Remark:

1. **Class** should be replaced by the persistence class.

Standard Association Handling

With standard association handling, when updating one end of the association, the generated persistent code will not update the other end of a bi-directional association automatically. Hence, you have to define the two ends of the bi-directional association manually to maintain consistency. Examples are given to show how to manipulate the one-to-one, one-to-many and many-to-many associations with standard association handling.

One-to-One Association

In order to manipulate the directional association, implement the program with the following steps:

1. Set the properties of role name of the object.

Table shows the method summary of the persistence class to be used for setting the directional association.

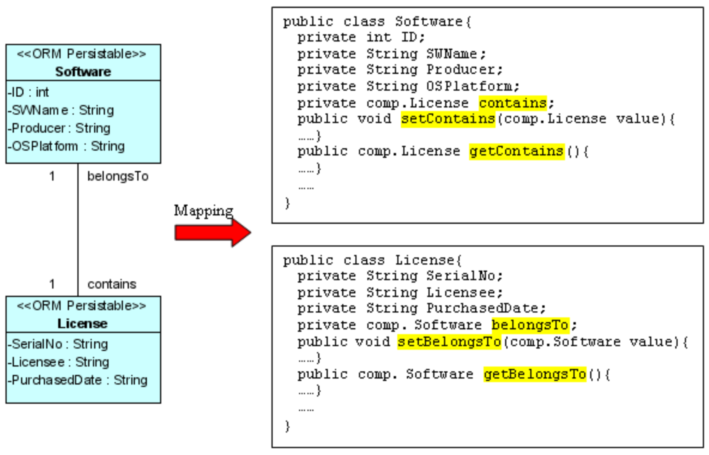
Return Type	Method Name	Description
void	setRoleName(Class value)	Set the value to the role of an instance.

Table 18.9

Remark:

1. RoleName should be replaced by the role of the navigable association.
2. Class should be replaced by the data type of the supplier class in the navigable association.

Example:



```
public class Software{
  private int ID;
  private String SWName;
  private String Producer;
  private String OSPlatform;
  private comp.License contains;
  public void setContains(comp.License value){
  ……}
  public comp.License getContains(){
  ……}
  ……
}
```

```
public class License{
  private String SerialNo;
  private String Licensee;
  private String PurchasedDate;
  private comp. Software belongsTo;
  public void setBelongsTo(comp.Software value){
  ……}
  public comp. Software getBelongsTo(){
  ……}
  ……
}
```

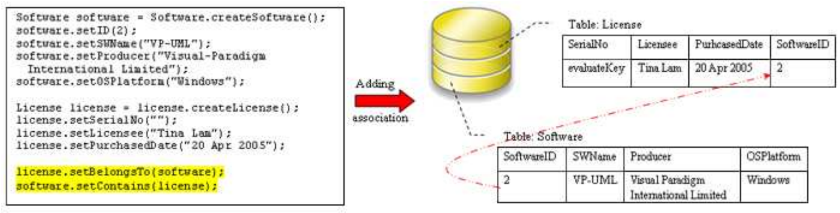
Figure 18.9 - Mapping one-to-one association

From the above example, an ORM-Persistable object model of Software maps to an ORM-Persistable Java class with an attribute of role, "contains" in the association typed as its associated persistence class, License. Meanwhile, the object model of License maps to a Java class with an attribute of role, "belongsTo" typed as Software. It specifies the association that Software contains a particular License and a License belongs to particular software.

To manipulate the association, the following lines of code should be implemented.

```
Software software = Software.createSoftware();
License license = License.createLicense();
license.setBelongsTo(software);
software.setContains(license);
```

In the above association, the primary key of the Software table will become a foreign key in License table when transforming to data model. Thus, after executing the lines of code, the primary key of the software will be referenced as a foreign key of the license record.



```
Software software = Software.createSoftware();
software.setID(2);
software.setSWName("VP-UML");
software.setProducer("Visual-Paradigm
  International Limited");
software.setOSPlatform("Windows");

License license = license.createLicense();
license.setSerialNo("");
license.setLicensee("Tina Lam");
license.setPurchasedDate("20 Apr 2005");

license.setBelongsTo(software);
software.setContains(license);
```

Table: License

SerialNo	Licensee	PurhcasedDate	SoftwareID
evaluateKey	Tina Lam	20 Apr 2005	2

Table: Software

SoftwareID	SWName	Producer	OSPlatform
2	VP-UML	Visual Paradigm International Limited	Windows

Figure 18.10 - Code for one-to-one association

It is a bi-directional association with standard association handling. The association link must be created by setting both the roles of Software and License.

One-to-Many Association

In a bi-directional one-to-many association, a class has multiplicity of one while the other has many. When transforming the association to persistent code, the role name will map to an attribute of the class with data type of a Java collection class. The type of collection is specified in the object model, refer to the description in Using Collection section for mapping the type of collection with Java collection class.

A bi-directional one-to-many association is shown below.

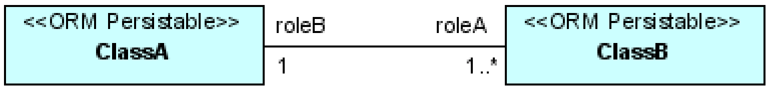


Figure 18.11 - An one-to-many association

With standard association handling, you have to create the association with the following steps in order to manipulate the bi-directional one-to-many association:

- Set the properties of role with an instance of the associated class; i.e. classB.setRoleB(classA)
- Add the objects to the collection of the associated class; i.e. classA.getRoleA().add(classB)

where classA is an object of ClassA; classB is an object of ClassB;
getRoleA returns the collection of ClassB; setRoleB is the setter method of property, roleB.

After specifying the association, the value of primary key of the object of ClassA will be referenced as a foreign key of the associated object of ClassB.

Setting the property of role

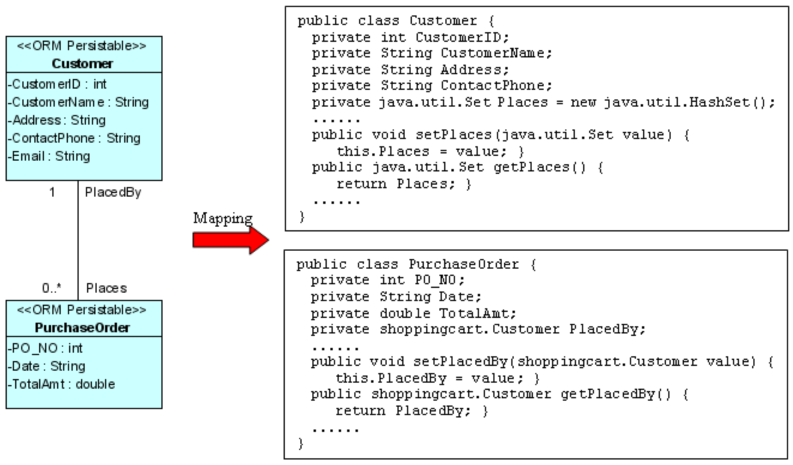
For information on setting the properties of role, refer to the description in the One-to-One Association section.

Adding objects to the collection

In order to add an object to the collection, implement the program with the following steps:

1. Create a new instance of the class which associates with more than one instance of associated class.
2. Create a new object of the associated class.
3. Add a new object or an existing object of the associated class to the collection belonging to the class.

Example:



```
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private java.util.Set Places = new java.util.HashSet();
  ......
  public void setPlaces(java.util.Set value) {
     this.Places = value; }
  public java.util.Set getPlaces() {
     return Places; }
  ......
}
```

```
public class PurchaseOrder {
  private int PO_NO;
  private String Date;
  private double TotalAmt;
  private shoppingcart.Customer PlacedBy;
  ......
  public void setPlacedBy(shoppingcart.Customer value) {
     this.PlacedBy = value; }
  public shoppingcart.Customer getPlacedBy() {
     return PlacedBy; }
  ......
}
```

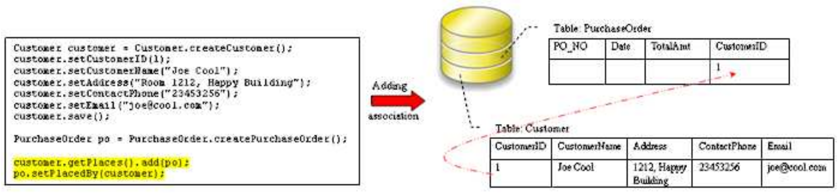
Figure 18.12 - Mapping one-to-many association with using Collection

From the above example, an ORM-Persistable object model of PurchaseOrder maps to an ORM-Persistable Java class with an attribute of role, "PlacedBy" in the association typed as an instance of Customer, specifying that the particular purchase order is placed by a particular customer. Moreover, the object model of Customer maps to a Java class with an attribute of role, "Places" in the association typed as a collection, Set which is the specified type of collection in the model manipulating instances of PurhcaseOrder.

To add a PurchaseOrder object to the collection of PurchaseOrder, the following lines of code should be implemented.

```
Customer customer = Customer.createCustomer();
PurchaseOrder po = PurchaseOrder.createPurchaseOrder();
customer.getPlaces().add(po);
po.setPlacedBy(customer);
```

After executing these lines of code, an object is added to the collection representing the association. When inserting records to the database tables, the primary key of the customer will be referenced to as the foreign key of the purchase order record.



```
Customer customer = Customer.createCustomer();
customer.setCustomerID(1);
customer.setCustomerName("Joe Cool");
customer.setAddress("Room 1212, Happy Building");
customer.setContactPhone("23453256");
customer.setEmail("joe@cool.com");
customer.save();

PurchaseOrder po = PurchaseOrder.createPurchaseOrder();

customer.getPlaces().add(po);
po.setPlacedBy(customer);
```

Table: PurchaseOrder

PO_NO	Date	TotalAmt	CustomerID
			1

Table: Customer

CustomerID	CustomerName	Address	ContactPhone	Email
1	Joe Cool	1212, Happy Building	23453256	joe@cool.com

Figure 18.13 - Code for one-to-many association with using Collection

Retrieving objects from the collection

In order to retrieve an object from the collection, implement the program with the following steps:

1. Retrieve an instance of the class which associates with more than one instance of associated class.
2. Get the collection from the class, and convert the collection into an object array.

Example:

Refer to the example of the one-to-many association between the ORM-Persistable object models of Customer and PurchaseOrder, implement the following lines of code to retrieve a collection of PurchaseOrder objects from the customer.

```
Customer customer = Customer.loadByName("Joe Cool");
Object[] orders = customer.getPlaces().toArray();
```

After executing these lines of code, the purchase order records associated with the customer are retrieved and stored in the object array.

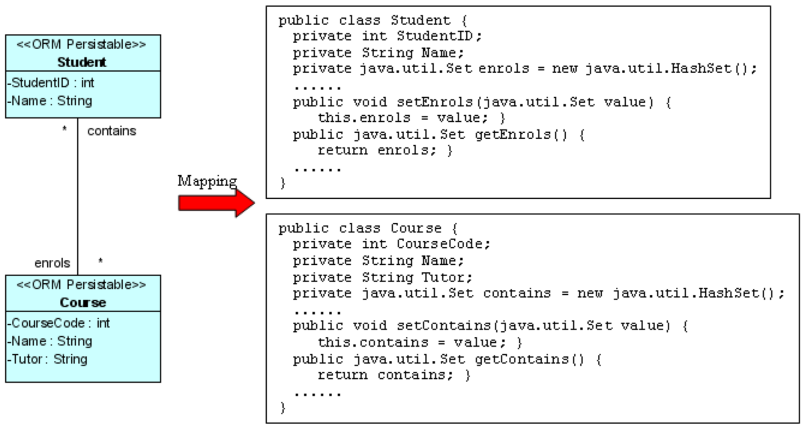
Retrieve the customer record by using the loadByName() method which is the method provided by the persistent API. For more information on retrieving persistent object, refer to the Persistent API section.

Many-to-Many Association

When transforming a many-to-many association between two ORM-Persistable classes with standard association handling, the role names will map to one type of collection defined in the object model. In order to specify the many-to-many association, add the objects to the corresponding collection of the associated class. For information on adding objects to the collection, refer to the description in the One-to-Many Association section.

In addition, a many-to-many association in the object model is transformed to data model by generating a Link Entity to form two one-to-many relationships between two generated entities. The primary keys of the two entities will migrate to the link entity as the primary/foreign keys. When specifying the association by adding objects to the collection, the primary key of the two related instances will be inserted as a pair for a row of the Link Entity automatically.

Example:



```
public class Student {
  private int StudentID;
  private String Name;
  private java.util.Set enrols = new java.util.HashSet();
  ......
  public void setEnrols(java.util.Set value) {
     this.enrols = value; }
  public java.util.Set getEnrols() {
     return enrols; }
  ......
}
```

```
public class Course {
  private int CourseCode;
  private String Name;
  private String Tutor;
  private java.util.Set contains = new java.util.HashSet();
  ......
  public void setContains(java.util.Set value) {
     this.contains = value; }
  public java.util.Set getContains() {
     return contains; }
  ......
}
```

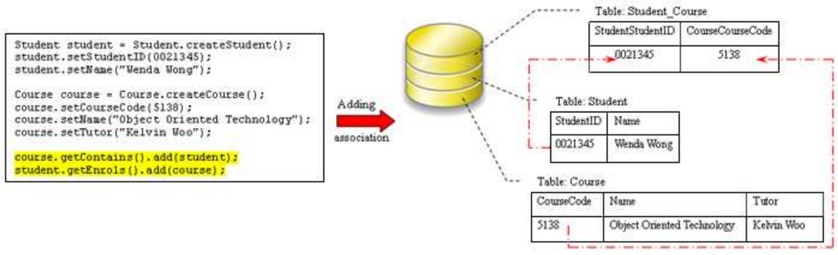
Figure 18.14 - Mapping Many-to-many association with using Collection

With standard association handling, only two classes including Student and Course are generated from the above object model.

By executing the following lines of code:

```
Student student = Student.createStudent();
student.setStudentID(0021345);
student.setName("Wenda Wong");
Course course = Course.createCourse();
course.setCourseCode(5138);
course.setName("Object Oriented Technology");
course.setTutor("Kelvin Woo");
course.getContains().add(student);
student.getEnrols().add(course);
```

Both the Student table and Course table are inserted with a new record. After specifying the association by course.getContains().add(student) and student.getEnrols().add(course), the corresponding primary keys of student record and course record migrate to the Student_Course Link Entity to form a row of record.



```
Student student = Student.createStudent();
student.setStudentID(0021345);
student.setName("Wenda Wong");

Course course = Course.createCourse();
course.setCourseCode(5138);
course.setName("Object Oriented Technology");
course.setTutor("Kelvin Woo");

course.getContains().add(student);
student.getEnrols().add(course);
```

Table: Student_Course

StudentStudentID	CourseCourseCode
0021345	5138

Table: Student

StudentID	Name
0021345	Wenda Wong

Table: Course

CourseCode	Name	Tutor
5138	Object Oriented Technology	Kelvin Woo

Figure 18.15 - Code for many-to-many association with using Collection

Using Collection

With standard association handling, the role name which associates with more than one instance of the supplier class is transformed into one type of Java collection class. The type of collection can be specified before the generation of code, refer to the description of Specifying Collection Type in the Object Model chapter.

The following table shows the mapping between the collection type defined in the association specification and the Java collection class.

Collection Type	Java Collection Class
Set	java.util.Set
Bag	java.util.Collection
List	java.util.List
Map	java.util.Map

Table 18.10

Persistent API

Persistent API refers to the persistent code used to manipulate the persistent data. There are four types of persistent API available for generating the Java persistence code. The four types of persistent API, which include Static Method, Factory Class, POJO and Data Access Object (DAO), are capable of manipulating the persistent data with the relational database, i.e., inserting, retrieving, updating and deleting records.

Using Static Method

Using static method persistent API, a persistence class for the ORM-Persistable class is generated with static methods which is capable of creating, retrieving persistent object and persisting data. The following class diagram shows the dependency relationship between the client program and the generated persistence class.

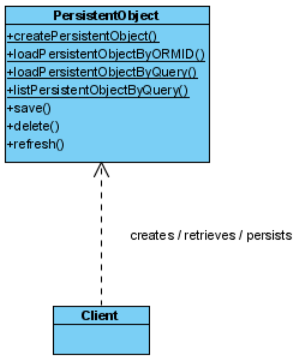

PersistentObject
+createPersistentObject()
+loadPersistentObjectByORMID()
+loadPersistentObjectByQuery()
+listPersistentObjectByQuery()
+save()
+delete()
+refresh()
creates / retrieves / persists
Client

Figure 18.16 - Class with static methods

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram. For example, the ORM-Persistable class, Customer persists with the Customer data in the Customer table.

Example:

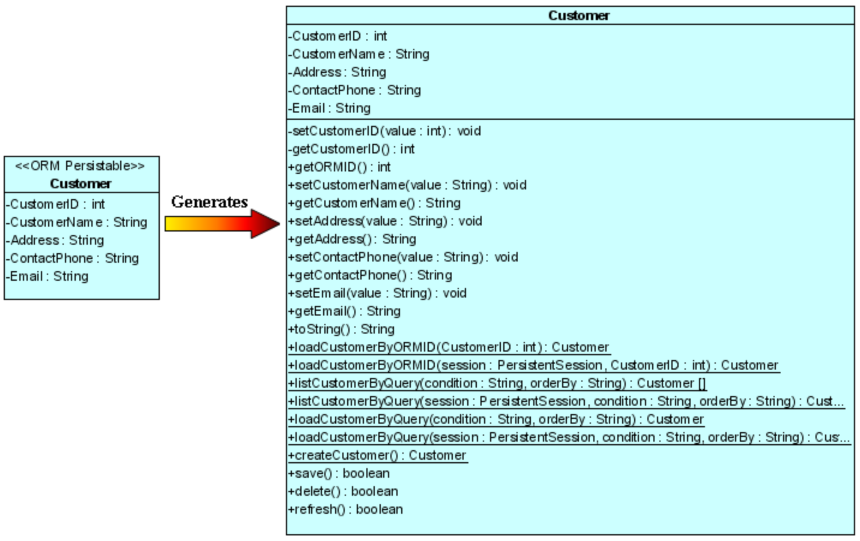

<<ORM Persistable>>
Customer
-CustomerID : int
-CustomerName : String
-Address : String
-ContactPhone : String
-Email : String
Generates
Customer
-CustomerID : int
-CustomerName : String
-Address : String
-ContactPhone : String
-Email : String
-setCustomerID(value : int) : void
-getCustomerID() : int
+getORMID() : int
+setCustomerName(value : String) : void
+getCustomerName() : String
+setAddress(value : String) : void
+getAddress() : String
+setContactPhone(value : String) : void
+getContactPhone() : String
+setEmail(value : String) : void
+getEmail() : String
+toString() : String
+loadCustomerByORMID(CustomerID : int) : Customer
+loadCustomerByORMID(session : PersistentSession, CustomerID : int) : Customer
+listCustomerByQuery(condition : String, orderBy : String) : Customer []
+listCustomerByQuery(session : PersistentSession, condition : String, orderBy : String) : Cust...
+loadCustomerByQuery(condition : String, orderBy : String) : Customer
+loadCustomerByQuery(session : PersistentSession, condition : String, orderBy : String) : Cus...
+createCustomer() : Customer
+save() : boolean
+delete() : boolean
+refresh() : boolean

Figure 18.17 - Mapping with static methods

From the above example, a Customer persistence class is generated with a set of methods supporting the database manipulation.

In this section, it introduces how to use the static methods of the generated persistence classes to manipulate the persistent data with the relational database.

Creating a Persistent Object

As a persistence class represents a table in the database and an instance of the class represents a record of the table, creating a persistent object in the application system is the same as adding a row of new record to the table.

In order to insert a new row to the database table, implement the program with the following steps:

1. Create a new instance of the class.
2. Set the properties of the object.
3. Insert the object as a row to the database.

Table shows the method summary of the persistence class

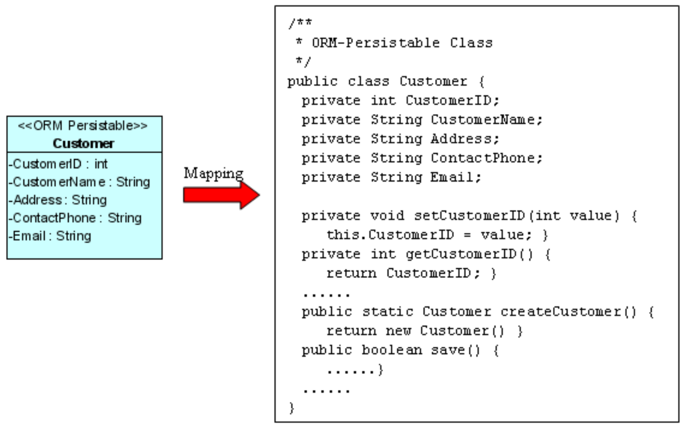
Return Type	Method Name	Description
Class	createClass()	Create a new instance of the class.
void	setAttribute(DataType value)	Set the value to the property of an instance.
boolean	save()	Insert the object as a row to the database table.

Table 18.11

Remark:

1. **Class** should be replaced by the name of the generated persistence class.
2. **Attribute** should be replaced by the name of the attribute.
3. **DataType** should be replaced by the data type of the attribute defined in the object model.

Example:



```
/**
 * ORM-Persistable Class
 */
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;

  private void setCustomerID(int value) {
     this.CustomerID = value; }
  private int getCustomerID() {
     return CustomerID; }
  ......
  public static Customer createCustomer() {
     return new Customer() }
  public boolean save() {
     ......}
  ......
}
```

Figure 18.18 - Mapping ORM Persistent Class

From the above example, an ORM-Persistable object model of Customer maps to an ORM-Persistable Java class generated with methods for creating a new instance and inserting the instance as a row of the database table.

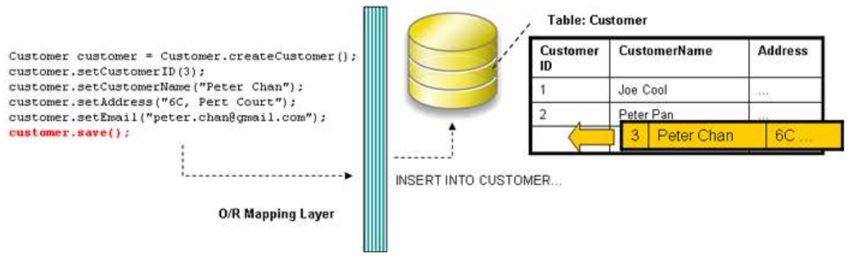
To insert a new Customer record to the table, Customer of the database, the following lines of code should be implemented.

```
Customer customer = Customer.createCustomer();
customer.setCustomerID(3);
customer.setCustomerName("Peter Chan");
customer.setAddress("6C, Pert Court");
customer.setEmail("peter.chan@gmail.com");
customer.save();
```

After executing these lines of code, a row of record is inserted to the database table.

An alternative way to create a new instance of the class is using the new operator:
Class c = new Class();

From the above example, Customer customer = Customer.createCustomer() can be replaced by Customer customer = new Customer() to create a Customer object.



```
Customer customer = Customer.createCustomer();
customer.setCustomerID(3);
customer.setCustomerName("Peter Chan");
customer.setAddress("6C, Pert Court");
customer.setEmail("peter.chan@gmail.com");
customer.save();
```

Table: Customer

Customer ID	CustomerName	Address
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	6C ...

Figure 18.19 - Save a Data object to the database

Loading a Persistent Object

As the database table is represented by a persistence class, a record of the table can thus be represented by an instance. A record retrieved from the table will be stored as an object.

In order to retrieve a record from the database table, you have to specify the condition for finding the record. To retrieve a record, implement the program with the following steps:

1. Specify the condition for searching the record.
2. Load the retrieved record to an object.

Table shows the method summary of the persistence class to be used for retrieving a record from database.

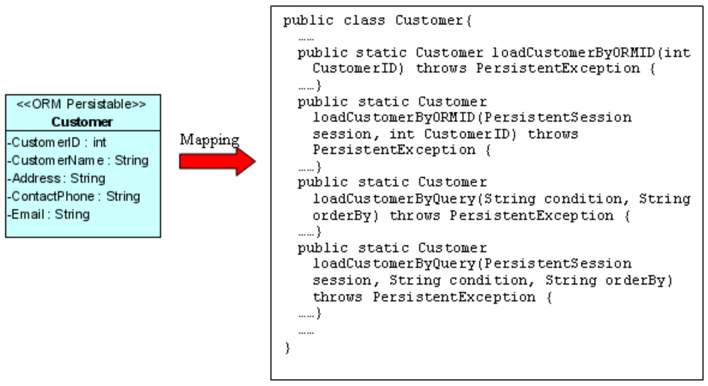
Return Type	Method Name	Description
Class	loadClassByORMID(DataType PrimaryKey)	Retrieve a record matching with the specified value of primary key.
Class	loadClassByORMID(PersistentSession session, DataType PrimaryKey)	Retrieve a record matching with the specified value of primary key and specified session.
Class	loadClassByQuery(String condition, String orderBy)	Retrieve the first record matching the user defined condition while the matched records are ordered by a specified attribute.
Class	loadClassByQuery(PersistentSession session, String condition, String orderBy)	Retrieve the first record matching the user defined condition and specified session while the matched records are ordered by a specified attribute.

Table 18.12

Remark:

1. **Class** should be replaced by the name of the persistence class.
2. **DataType** should be replaced by the data type of the attribute defined in the object model.

Example:



```
public class Customer{
  ……
  public static Customer loadCustomerByORMID(int
    CustomerID) throws PersistentException {
  ……}
  public static Customer
    loadCustomerByORMID(PersistentSession
    session, int CustomerID) throws
    PersistentException {
  ……}
  public static Customer
    loadCustomerByQuery(String condition, String
    orderBy) throws PersistentException {
  ……}
  public static Customer
    loadCustomerByQuery(PersistentSession
    session, String condition, String orderBy)
    throws PersistentException {
  ……}
  ……
}
```

Figure 18.20 - Mapping with load method

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable Java class generated with methods for retrieving a matched record.

To retrieve a record from the Customer table, the following line of code should be implemented.

Loading an object by passing the value of primary key:

```
Customer customer = Customer.loadCustomerByORMID(2);
```

Loading an object by specifying a user defined condition:

```
Customer customer = Customer.loadCustomerByQuery("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

After executing the code, a matched row is retrieved and loaded to a Customer object.

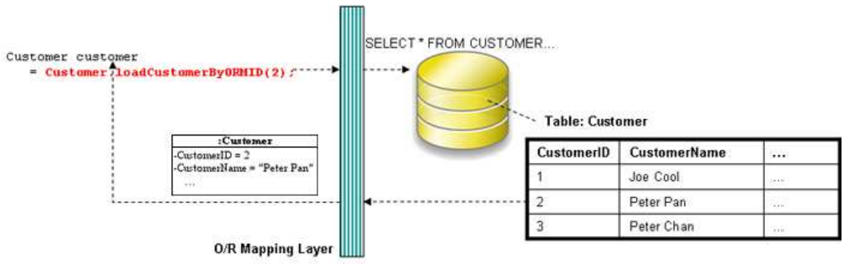


Figure 18.21 - Load record from database

Updating a Persistent Object

As a record can be retrieved from the table and loaded to an object of the persistence class, the record is allowed to update by simply using the setter method of the property.

In order to update a record, you have to retrieve the row being updated, update the value by setting the property to the database. To update the record, implement the program with the following steps:

1. Retrieve a record from database table and load as an object.
2. Set the updated value to the property of the object.
3. Save the updated record to the database.

Table shows the method summary of the persistence class to be used for updating a record.

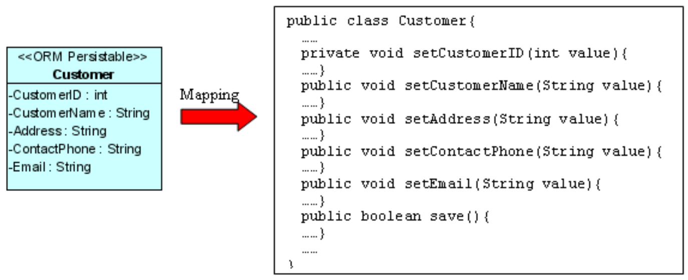
Return Type	Method Name	Description
void	setAttribute(DataType value)	Set the value to the property of an instance.
boolean	save()	Update the value to database.

Table 18.13

Remark:

1. **Attribute** should be replaced by the name of the attribute.
2. **DataType** should be replaced by the data type of the attribute defined in the object model.

Example:



```
public class Customer{
  ……
  private void setCustomerID(int value){
  ……}
  public void setCustomerName(String value){
  ……}
  public void setAddress(String value){
  ……}
  public void setContactPhone(String value){
  ……}
  public void setEmail(String value){
  ……}
  public boolean save(){
  ……}
  ……
}
```

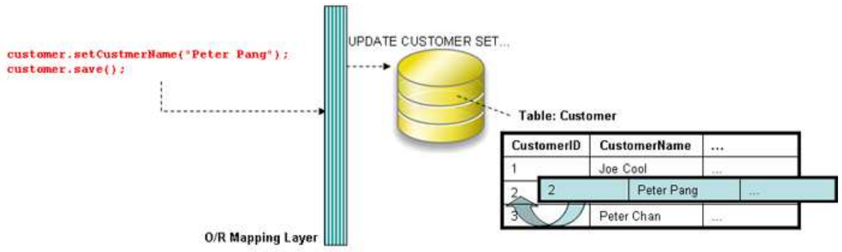
Figure 18.22 - Mapping with save methods

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable Java class generated with the methods for setting the properties and updating the row.

To update a Customer record, the following lines of code should be implemented.

```
customer.setCustomerName("Peter Pang");
customer.save();
```

After executing the above lines of code, the customer name is updated to "Peter Pang" in the database.



```
customer.setCustmerName("Peter Pang");
customer.save();
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pang	...
3	Peter Chan	...

Figure 18.23 - Code for update record

Deleting a Persistent Object

As a record can be retrieved from the table and loaded to an object of the persistence class, the record can be deleted by simply using the delete method of the persistence class.

In order to delete a record, implement the program with the following steps:

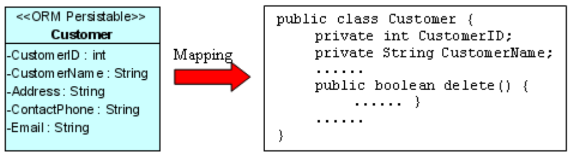
1. Retrieve a record from database table and load as an object.
2. Delete the retrieved record.

Table shows the method summary of the persistence class to be used for

Return Type	Method Name	Description
boolean	delete()	Delete the current instance.

Table 18.14

Example:



```
public class Customer {
    private int CustomerID;
    private String CustomerName;
    ......
    public boolean delete() {
        ...... }
    ......
}
```

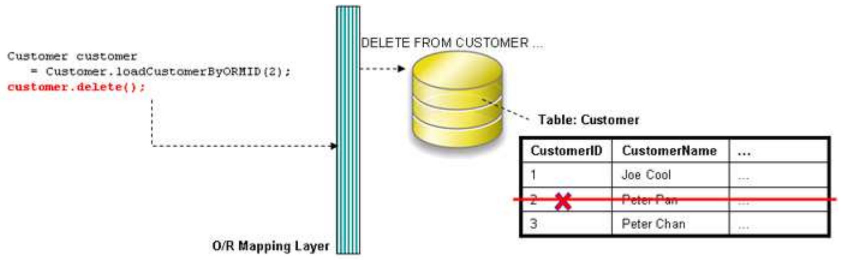
Figure 18.24 - Mapping delete methods

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable Java class generated with the methods for deleting the specified record from the database.

To delete a Customer record, the following lines of code should be implemented.

```
Customer customer = Customer.loadCustomerByORMID(2);
customer.delete();
```

After executing the above code, the specified customer record is deleted from the database.



```
Customer customer
    = Customer.loadCustomerByORMID(2);
customer.delete();
```

CustomerID	CustomerName	...
1	Joe Cool	...
~~2~~	~~Peter Pan~~	~~...~~
3	Peter Chan	...

Figure 18.25 - Code for delete record

Querying

For most of the database application, the database is enriched with information. Information may be requested from the database so as to performing an application function, such as retrieving the unit price of products for calculating the total amount for a purchase order.

As the persistence class represents a table, the ORM-Persistable Java class is generated with methods for retrieving information from the database.

Loading a Collection of Persistent Objects

As the database table usually contains many records, you may want to query the tables by a specified condition. The generated persistent code supports querying the database, the matched records will be retrieved and loaded as an object array.

In order to retrieve records from the table, you have to specify the condition for querying. To retrieve a number of records, implement the program with the following steps:

1. Specify the condition for searching the record.
2. Load the retrieved records as an object array.

Table shows the method summary of the persistence class to be used for retrieving records from database table.

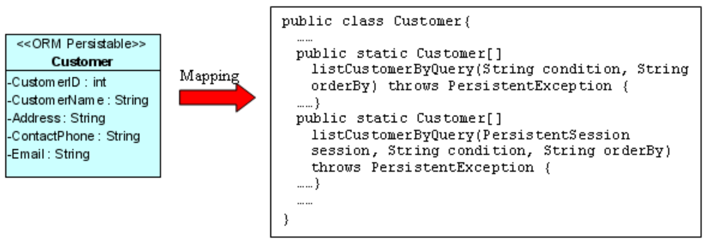
Return Type	Method Name	Description
Class[]	listClassByQuery(String condition, String orderBy)	Retrieve the records matched with the user defined condition and ordered by a specified attribute.
Class[]	listClassByQuery(PersistentSession session, String condition, String orderBy)	Retrieve the records matched with the user defined condition and specified session and ordered by a specified attribute.

Table 18.15

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer{
  ……
  public static Customer[]
    listCustomerByQuery(String condition, String
    orderBy) throws PersistentException {
  ……}
  public static Customer[]
    listCustomerByQuery(PersistentSession
    session, String condition, String orderBy)
    throws PersistentException {
  ……}
  ……
}
```

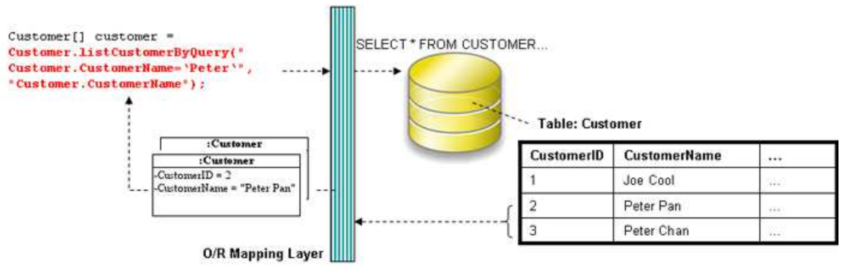
Figure 18.26 - Mapping with list methods

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable Java class generated with methods for retrieving records.

To retrieve records from the Customer table, the following line of code should be implemented.

```
Customer[] customer = Customer.listCustomerByQuery("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

After executing the code, the matched rows are retrieved and loaded to an object array of Customer.



```
Customer[] customer =
Customer.listCustomerByQuery("
Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 18.27 - Code for retrieve a list of records

Using ORM Qualifier

ORM Qualifier is an additional feature which allows you to specify the extra data retrieval rules apart from the system pre-defined rules. The ORM Qualifier can be defined in order to generate persistence code. Refer to Defining ORM Qualifier in the Object Model chapter for more information.

By defining the ORM Qualifier in a class, the persistence class will be generated with additional data retrieval methods, load and list methods.

Table shows the method summary generated by defining the ORM Qualifier.

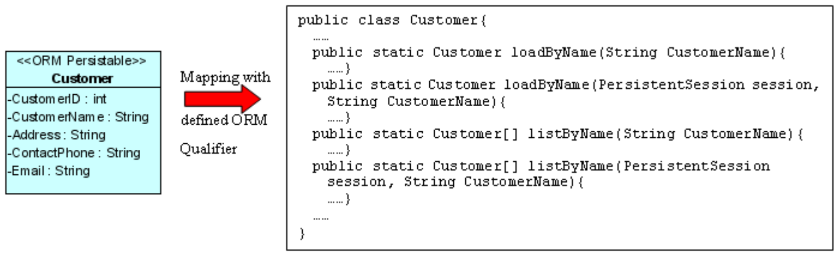
Return Type	Method Name	Description
Class	loadByORMQualifier(DataType attribute)	Retrieve the first record that matches the specified value with the attribute defined in the ORM Qualifier.
Class	loadByORMQualifier (PersistentSession session, DataType attribute)	Retrieve the first record that matches the specified value with the attribute defined in the ORM Qualifier and specified session.
Class[]	listByORMQualifier (DataType attribute)	Retrieve the records that match the specified value with the attribute defined in the ORM Qualifier.
Class[]	listByORMQualifier (PersistentSession session, DataType attribute)	Retrieve the records that match the specified value with the attribute defined in the ORM Qualifier and specified session.

Table 18.16

Remark:

1. **Class** should be replaced by the name of the persistence class.
2. **ORMQualifier** should be replaced by the Name defined in the ORM Qualifier.
3. **DataType** should be replaced by the data type of the attribute which associated with the ORM Qualifier.
4. **attribute** is the specified value to be used for querying the table.

Example:



```
public class Customer{
  ……
  public static Customer loadByName(String CustomerName){
    ……}
  public static Customer loadByName(PersistentSession session,
    String CustomerName){
    ……}
  public static Customer[] listByName(String CustomerName){
    ……}
  public static Customer[] listByName(PersistentSession
    session, String CustomerName){
    ……}
  ……
}
```

Figure 18.28 - Mapping with load and list by qualifier

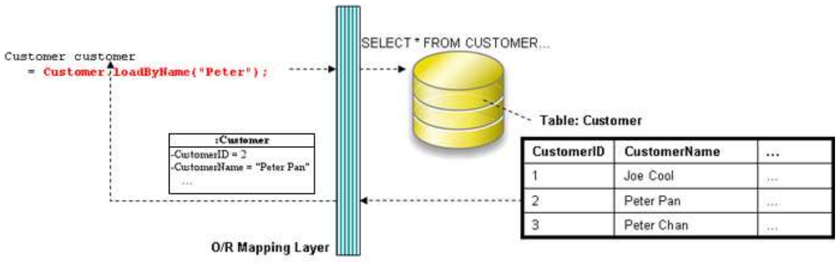
In the above example, a customer object model is defined with an ORM Qualifier named as Name and qualified with the attribute, CustomerName.

To query the Customer table with the ORM Qualifier in one of the two ways

- By Load method

  ```
  Customer customer = Customer.loadByName("Peter");
  ```

  After executing the code code, the first occurrence of "Peter" in the CustomerName column in the Customer table will be loaded to the object identified as customer.



```
Customer customer
  = Customer.loadByName("Peter");
```

```
-CustomerID = 2
-CustomerName = "Peter Pan"
...
```

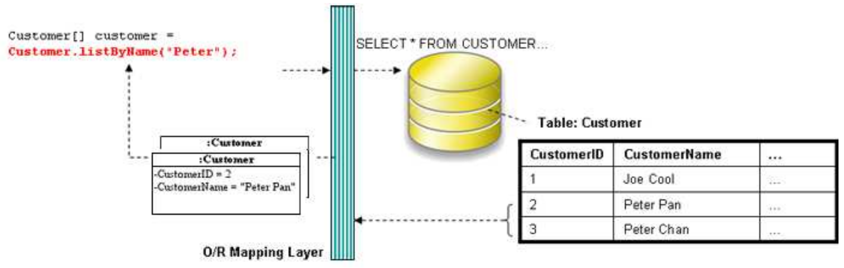
CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 18.29 - Code for retrieve a record by qualifier

- By List method

  ```
  Customer[] customer = Customer.listByName("Peter");
  ```

After executing the code, all rows which contain "Peter" in the CustomerName column in the Customer table will be retrieved and stored in an array of Customer object.



```
Customer[] customer =
Customer.listByName("Peter");
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 18.30 - Code for retrieve a list of records by qualifier

Using Criteria Class

When generating the persistence class for each ORM-Persistable class defined in the object model, the corresponding criteria class can also be generated. Criteria class is a helper class which provides an additional way to specify the condition to retrieve records from the database. Refer to Using Criteria Class section for more detailed information on how to specify the conditions to query the database by the criteria class.

You can get the retrieved records from the criteria class in one of the two ways:

- Use the retrieval methods of the criteria class.

  For information on the retrieval methods provided by the criteria class, refer to the description of Loading Retrieved Records section.

- Use the retrieval by criteria methods of the persistence class.

Table shows the method summary generated to the persistence class to retrieve records from the criteria class.

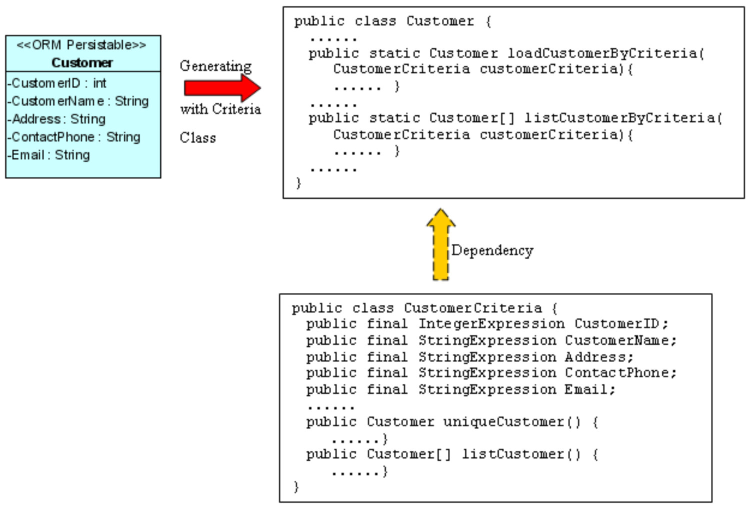
Return Type	Method Name	Description
Class	loadClassByCriteria(ClassCriteria value)	Retrieve the single record that matches the specified conditions applied to the criteria class.
Class[]	listClassByCriteria(ClassCriteria value)	Retrieve the records that match the specified conditions applied to the criteria class.

Table 18.17

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  ......
  public static Customer loadCustomerByCriteria(
     CustomerCriteria customerCriteria){
     ...... }
  ......
  public static Customer[] listCustomerByCriteria(
     CustomerCriteria customerCriteria){
     ...... }
  ......
}
```

```
public class CustomerCriteria {
  public final IntegerExpression CustomerID;
  public final StringExpression CustomerName;
  public final StringExpression Address;
  public final StringExpression ContactPhone;
  public final StringExpression Email;
  ......
  public Customer uniqueCustomer() {
     ......}
  public Customer[] listCustomer() {
     ......}
}
```

Figure 18.31 - Mapping with dependency

To retrieve records from the Criteria Class in one of the two ways:

- By Load method

  ```
  CustomerCriteria customerCriteria = new CustomerCriteria();
  customerCriteria.CustomerName.like("%Peter%");
  Customer customer = Customer.loadCustomerByCriteria(customerCriteria);
  ```

  After executing the code, the first occurrence of "Peter" in the CustomerName column in the Customer table will be loaded to the object identified as customer.

- By List method

  ```
  CustomerCriteria customerCriteria = new CustomerCriteria();
  CustomerCriteria.CustomerName.like("%Peter%");
  Customer[] customer = Customer.listCustomerByCriteria(customerCriteria);
  ```

  After executing the code, all rows which contain "Peter" in the CustomerName column in the Customer table will be retrieved and stored in an array of Customer object.

Using Factory Class

Using factory class persistent API, not only the persistence class will be generated, but also the corresponding factory class for each ORM-Persistable class. The generated factory class is capable of creating a new instance of the persistence class, which assists in inserting a new record to the database, and retrieving record(s) from the database by specifying the condition for query. After an instance is created by the factory class, the persistence class allows setting the values of its property and updating into the database. The persistence class also supports the deletion of records.

The following class diagram shows the relationship between the client program, persistent object and factory object.

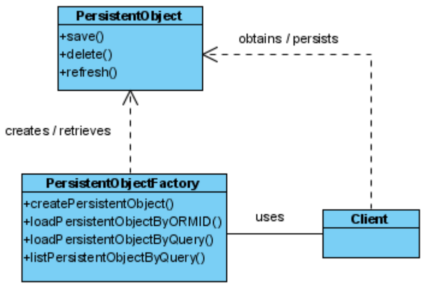

PersistentObject
+save()
+delete()
+refresh()
obtains / persists
creates / retrieves
PersistentObjectFactory
+createPersistentObject()
+loadPersistentObjectByORMID()
+loadPersistentObjectByQuery()
+listPersistentObjectByQuery()
uses
Client

Figure 18.32 - Mapping with using Factory Class

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram while the PersistentObjectFactory refers to the generated factory class of the ORM-Persistable class. For example, the CustomerFactory class creates and retrieves Customer persistent object while the ORM-Persistable class, Customer persists with the Customer data in the Customer table.

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram while the PersistentObjectFactory refers to the generated factory class of the ORM-Persistable class. For example, the CustomerFactory class creates and retrieves Customer persistent object while the ORM-Persistable class, Customer persists with the Customer data in the Customer table.

Example:

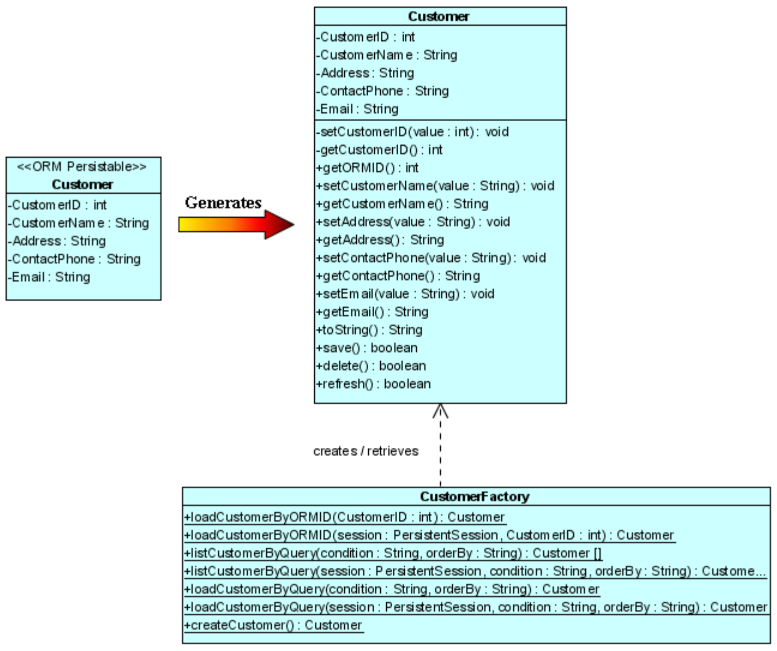

<<ORM Persistable>>
Customer
-CustomerID : int
-CustomerName : String
-Address : String
-ContactPhone : String
-Email : String
Generates
Customer
-CustomerID : int
-CustomerName : String
-Address : String
-ContactPhone : String
-Email : String
-setCustomerID(value : int) : void
-getCustomerID() : int
+getORMID() : int
+setCustomerName(value : String) : void
+getCustomerName() : String
+setAddress(value : String) : void
+getAddress() : String
+setContactPhone(value : String) : void
+getContactPhone() : String
+setEmail(value : String) : void
+getEmail() : String
+toString() : String
+save() : boolean
+delete() : boolean
+refresh() : boolean
creates / retrieves
CustomerFactory
+loadCustomerByORMID(CustomerID : int) : Customer
+loadCustomerByORMID(session : PersistentSession, CustomerID : int) : Customer
+listCustomerByQuery(condition : String, orderBy : String) : Customer []
+listCustomerByQuery(session : PersistentSession, condition : String, orderBy : String) : Custome...
+loadCustomerByQuery(condition : String, orderBy : String) : Customer
+loadCustomerByQuery(session : PersistentSession, condition : String, orderBy : String) : Customer
+createCustomer() : Customer

Figure 18.33 - Class Diagram with Factory Class

From the above example, the CustomerFactory class supports the creation of Customer persistent object, the retrieval of Customer records from the Customer table while the Customer persistence class supports the update and deletion of customer record.

Creating a Persistent Object

An instance of a persistence class represents a record of the corresponding table, creating a persistent object corresponds to inserting a new record to the table.

In order to insert a new row to the database table, implement the program with the following steps:

1. Create a new instance of the class by the factory class.
2. Set the properties of the object by the persistence class.
3. Insert the object as a row to the database by the persistence class.

Table shows the method summary of the factory and persistence classes to be used for inserting a row to database.

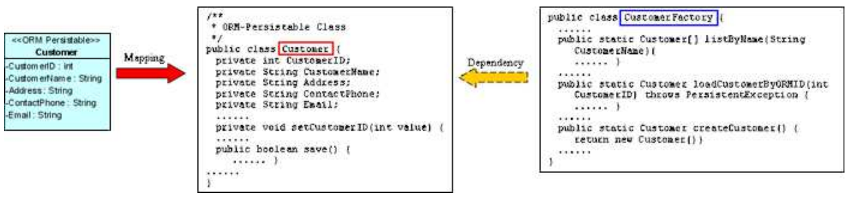
Class Type	Return Type	Method Name	Description
Factory Class	Class	createClass()	Create a new instance of the class.
Persistence Class	void	setAttribute(DataType value)	Set the value to the property of an instance.
Persistence Class	boolean	save()	Insert the object as a row to the database table.

Table 18.18

Remark:

1. **Class** should be replaced by the name of the generated persistence class.
2. **Attribute** should be replaced by the name of the attribute.
3. **DataType** should be replaced by the data type of the attribute defined in the object model.

Example:



```
/**
 * ORM-Persistable Class
 */
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
  ......
  public boolean save() {
     ...... }
......
}
```

```
public class CustomerFactory {
  ......
  public static Customer[] listByName(String
     CustomerName){
     ...... }
  ......
  public static Customer loadCustomerByORMID(int
     CustomerID) throws PersistentException {
     ...... }
  ......
  public static Customer createCustomer() {
     return new Customer()}
  ......
}
```

Figure 18.34 - Mapping with Factory Class

From the above example, an ORM-Persistable object model of Customer maps to an ORM-Persistable Java class with methods for setting the properties. An ORM-Persistable factory class is generated with method for creating a new instance and; and thus these methods allow inserting the instance as a row of the database table.

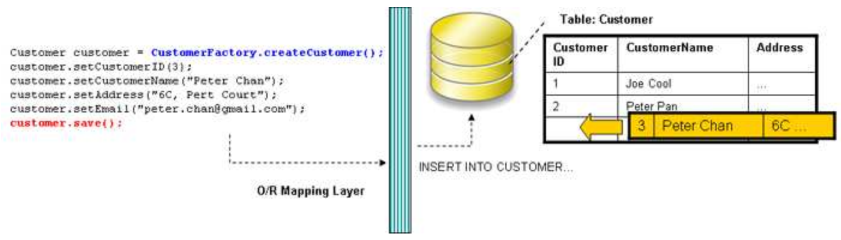
To insert a new Customer record to the table, Customer of the database, the following lines of code should be implemented.

```
Customer customer = CustomerFactory.createCustomer();
customer.setCustomerID(3);
customer.setCustomerName("Peter Chan");
customer.setAddress("6C, Pert Court");
customer.setEmail("peter.chan@gmail.com");
customer.save();
```

After executing these lines of code, a row of record is inserted to the database table.

An alternative way to create a new instance of the class is using the new operator:
Class c = new Class();

From the above example, Customer customer = CustomerFactory.createCustomer() can be replaced by Customer customer = new Customer() to create a Customer object.



```
Customer customer = CustomerFactory.createCustomer();
customer.setCustomerID(3);
customer.setCustomerName("Peter Chan");
customer.setAddress("6C, Pert Court");
customer.setEmail("peter.chan@gmail.com");
customer.save();
```

Table: Customer

Customer ID	CustomerName	Address
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	6C ...

Figure 18.35 - Insert record by using Factory Class

Loading a Persistent Object

As an instance of a persistence class represents a record of the corresponding table, a record retrieved from the table will be stored as an object.

In order to retrieve a record from the database table, you have to specify the condition for finding the record. To retrieve a record, implement the program with the following steps:

1. Specify the condition for searching the record by the factory class.
2. Load the retrieved record to an object.

Table shows the method summary of the factory class to be used for

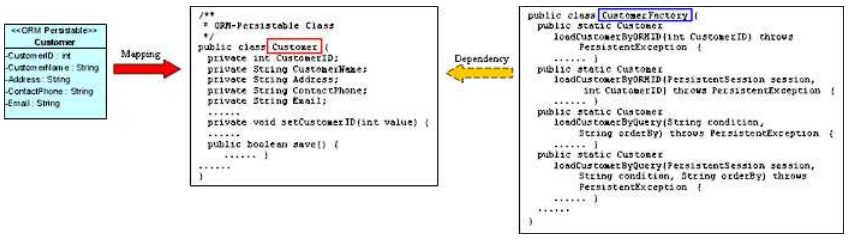
Class Type	Return Type	Method Name	Description
Factory Class	Class	loadClassByORMID(DataType PrimaryKey)	Retrieve a record matching with the specified value of primary key.
Factory Class	Class	loadClassByORMID(PersistentSession session, DataType PrimaryKey)	Retrieve a record matching with the specified value of primary key and specified session.
Factory Class	Class	loadClassByQuery(String condition, String orderBy)	Retrieve the first record matching the user defined condition while the matched records are ordered by a specified attribute.
Factory Class	Class	loadClassByQuery(PersistentSession session, String condition, String orderBy)	Retrieve the first record matching the user defined condition and specified session while the matched records are ordered by a specified attribute.

Table 18.19

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
/**
 * ORM-Persistable Class
 */
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
  ......
  public boolean save() {
     ...... }
......
}
```

```
public class CustomerFactory {
  public static Customer
    loadCustomerByORMID(int CustomerID) throws
        PersistentException {
    ...... }
  public static Customer
    loadCustomerByORMID(PersistentSession session,
         int CustomerID) throws PersistentException {
    ...... }
  public static Customer
    loadCustomerByQuery(String condition,
        String orderBy) throws PersistentException {
    ...... }
  public static Customer
    loadCustomerByQuery(PersistentSession session,
        String condition, String orderBy) throws
        PersistentException {
    ...... }
  ......
}
```

Figure 18.36 - Mapping load and list methods in Factory Class

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable Java class. A factory class is generated with methods for retrieving a matched record.

To retrieve a record from the Customer table, the following line of code should be implemented.

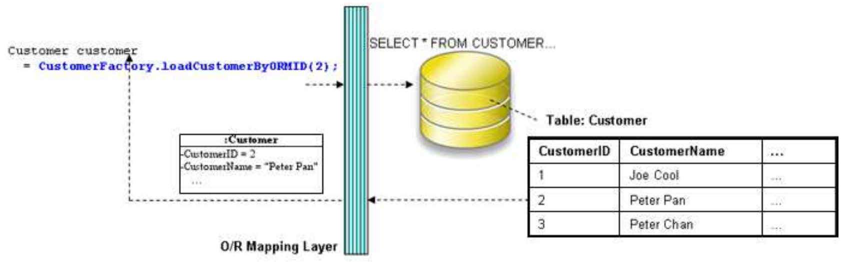
Loading an object by passing the value of primary key:

```
Customer customer = CustomerFactory.loadCustomerByORMID(2);
```

Loading an object by specifying a user defined condition:

```
Customer customer = CustomerFactory.loadCustomerByQuery("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

After executing the code, a matched row is retrieved and loaded to a Customer object.



```
Customer customer
  = CustomerFactory.loadCustomerByORMID(2);
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 18.37 - Retrieve a record by using Factory Class

Updating a Persistent Object

As a record can be retrieved from the table and loaded as an instance of the persistence class, setting a new value to the attribute by its setter method supports the update of record.

In order to update a record, you have to first retrieve the row to be updated, and then set a new value to the property, finally update to the database. To update the record, implement the program with the following steps:

1. Retrieve a record from database table and load as an object by the factory class.
2. Set the updated value to the property of the object by the persistence class.
3. Save the updated record to the database by the persistence class.

Table shows the method summary of the persistence class to be used for updating a record.

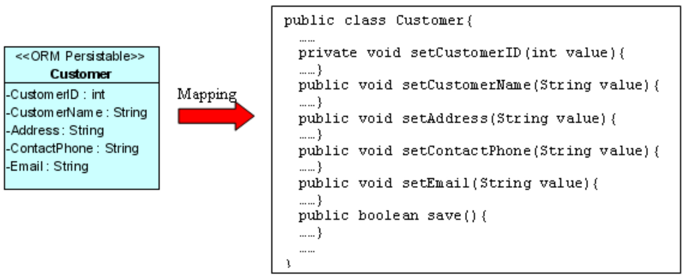
Class Type	Return Type	Method Name	Description
Persistence Class	void	setAttribute(DataType value)	Set the value to the property of an instance.
Persistence Class	boolean	save()	Update the value to database.

Table 18.20

Remark:

1. **Attribute** should be replaced by the name of the attribute.
2. **DataType** should be replaced by the data type of the attribute defined in the object model.

Example:



```
public class Customer{
  ……
  private void setCustomerID(int value){
  ……}
  public void setCustomerName(String value){
  ……}
  public void setAddress(String value){
  ……}
  public void setContactPhone(String value){
  ……}
  public void setEmail(String value){
  ……}
  public boolean save(){
  ……}
  ……
}
```

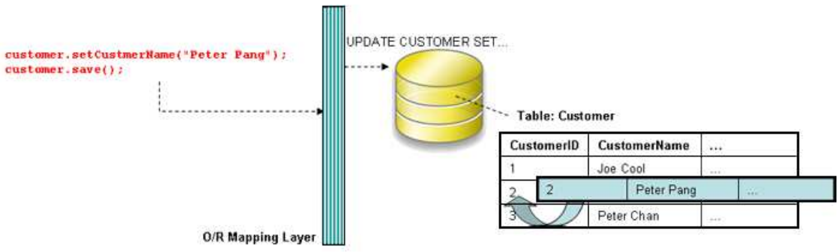
Figure 18.38 - Mapping ORM Persistable Class

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable Java class generated with the methods for setting the properties and updating the row.

To update a Customer record, the following lines of code should be implemented.

```
customer.setCustomerName("Peter Pang");
customer.save();
```

After executing the above lines of code, the customer name is updated to "Peter Pang" in the database.



```
customer.setCustmerName("Peter Pang");
customer.save();
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pang	...
3	Peter Chan	...

Figure 18.39 - Update a record

Deleting a Persistent Object

As a record can be retrieved from the table and loaded to an object of the persistence class, the record can be deleted by simply using the delete method of the persistence class.

In order to delete a record, implement the program with the following steps:

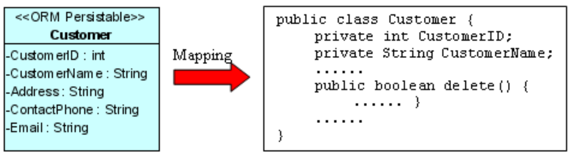
1. Retrieve a record from database table and load as an object by the factory class.
2. Delete the retrieved record by the persistence class.

Table shows the method summary of the persistence class to be used for deleting a record from database

Class Type	Return Type	Method Name	Description
Persistence Class	boolean	delete()	Delete the current instance.

Table 18.21

Example:



```
public class Customer {
     private int CustomerID;
     private String CustomerName;
     ......
     public boolean delete() {
          ...... }
     ......
}
```

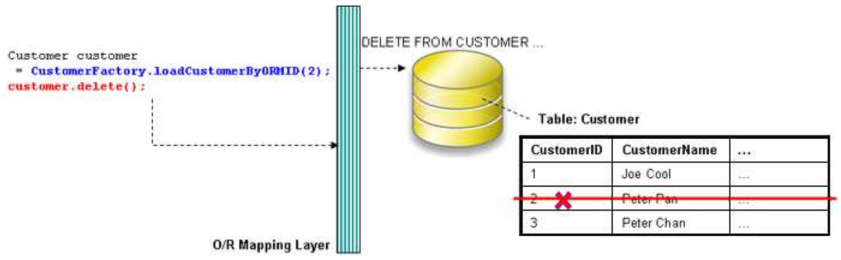
Figure 18.40 - Mapping delete method

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable Java class generated with the methods for deleting the specified record from the database.

To delete a Customer record, the following lines of code should be implemented.

```
Customer customer = CustomerFactory.loadCustomerByORMID(2);
customer.delete();
```

After executing the above code, the specified customer record is deleted from the database.



```
Customer customer
  = CustomerFactory.loadCustomerByORMID(2);
customer.delete();
```

CustomerID	CustomerName	...
1	Joe Cool	...
~~2~~	~~Peter Pan~~	~~...~~
3	Peter Chan	...

Figure 18.41 - Code for delete a record

Querying

It is well known that the database is enriched with information. In some cases, information may be retrieved from the database to assist another function of the application, such as retrieving the unit price of products for calculating the total amount for a purchase order.

As the persistence factory class supports the retrieval of records, using the methods of the factory class can retrieve records from the database according to the specified condition.

Loading a Collection of Persistent Objects

As the database table usually contains many records, you may want to query the tables by a specified condition. The generated factory class supports querying the database, the matched records will be retrieved and loaded as an object array.

In order to retrieve records from the table, you have to specify the condition for querying. To retrieve a number of records, implement the program with the following steps:

1. Specify the condition for searching the record by the factory class.
2. Load the retrieved records as an object array by the factory class.

Table shows the method summary of the factory class to be used for

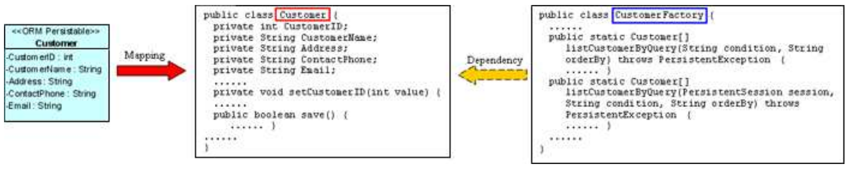
Class Type	Return Type	Method Name	Description
Factory Class	Class[]	listClassByQuery(String condition, String orderBy)	Retrieve the records matched with the user defined condition and ordered by a specified attribute.
Factory Class	Class[]	listClassByQuery(PersistentSession session, String condition, String orderBy)	Retrieve the records matched with the user defined condition and specified session and ordered by a specified attribute.

Table 18.22

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
  ......
  public boolean save() {
     ...... }
......
}
```

```
public class CustomerFactory {
  ......
  public static Customer[]
     listCustomerByQuery(String condition, String
     orderBy) throws PersistentException  {
     ...... }
  public static Customer[]
     listCustomerByQuery(PersistentSession session,
     String condition, String orderBy) throws
     PersistentException  {
     ...... }
  ......
}
```

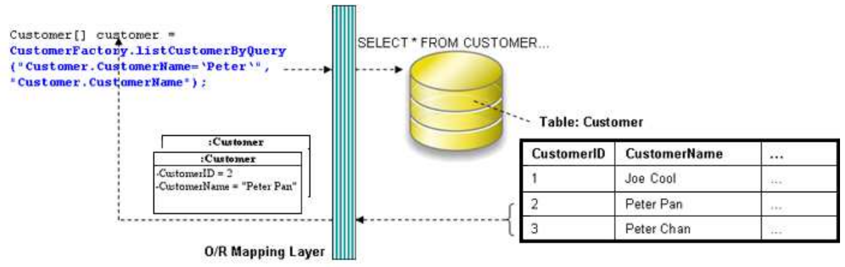
Figure 18.42 - Mapping List methods

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable Java factory class generated with methods for retrieving records.

To retrieve records from the Customer table, the following line of code should be implemented.

```
Customer[] customer = CustomerFactory.listCustomerByQuery("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

After executing the code, the matched rows are retrieved and loaded to an object array of Customer.



```
Customer[] customer =
CustomerFactory.listCustomerByQuery
("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 18.43 - Retrieve a list of records

Using ORM Qualifier

ORM Qualifier is an additional feature which allows you to specify the extra data retrieval rules apart from the system pre-defined rules. The ORM Qualifier can be defined in order to generate persistence code. Refer to Defining ORM Qualifier in the Object Model chapter for more information.

By defining the ORM Qualifier in a class, the factory class will be generated with additional data retrieval methods, load and list methods.

Table shows the method summary generated to the factory class by defining the ORM Qualifier.

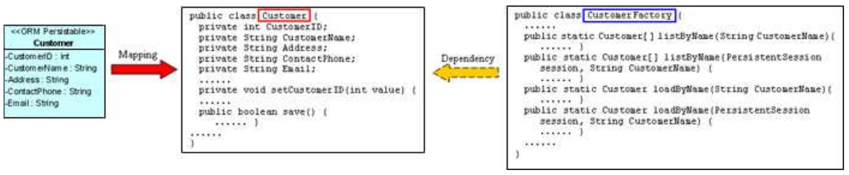
Class Type	Return Type	Method Name	Description
Factory Class	Class	loadByORMQualifier(DataType attribute)	Retrieve the first record that matches the specified value with the attribute defined in the ORM Qualifier.
Factory Class	Class	loadByORMQualifier (PersistentSession session, DataType attribute)	Retrieve the first record that matches the specified value with the attribute defined in the ORM Qualifier and specified session.
Factory Class	Class[]	listByORMQualifier (DataType attribute)	Retrieve the records that match the specified value with the attribute defined in the ORM Qualifier.
Factory Class	Class[]	listByORMQualifier (PersistentSession session, DataType attribute)	Retrieve the records that match the specified value with the attribute defined in the ORM Qualifier and specified session.

Table 18.23

Remark:

1. **Class** should be replaced by the name of the persistence class.
2. **ORMQualifier** should be replaced by the Name defined in the ORM Qualifier.
3. **DataType** should be replaced by the data type of the attribute which associated with the ORM Qualifier.
4. **attribute** is the specified value to be used for querying the table.

Example:



```
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
  ......
  public boolean save() {
     ...... }
......
}
```

```
public class CustomerFactory {
  ......
  public static Customer[] listByName(String CustomerName) {
     ...... }
  public static Customer[] listByName(PersistentSession
     session, String CustomerName) {
     ...... }
  public static Customer loadByName(String CustomerName) {
     ...... }
  public static Customer loadByName(PersistentSession
     session, String CustomerName) {
     ...... }
  ......
}
```

Figure 18.44 - Mapping load and list method with Qualifier

In the above example, a customer object model is defined with an ORM Qualifier named as Name and qualified with the attribute, CustomerName.

To query the Customer table with the ORM Qualifier in one of the two ways

- By Load method

```
Customer customer = CustomerFactory.loadByName("Peter");
```

After executing the code, the first occurrence of "Peter" in the CustomerName column in the Customer table will be loaded to the object identified as customer.

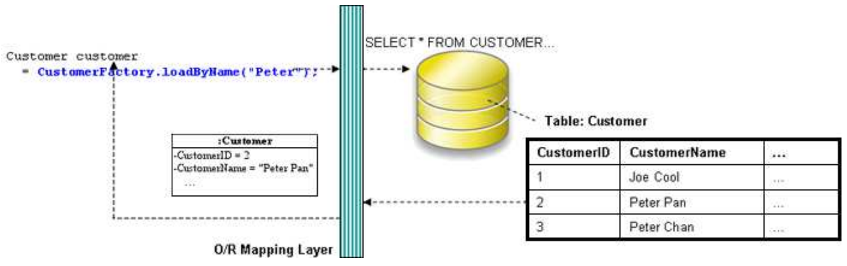


Figure 18.45 - Retrieve a record by ORM qualifier

- By List method

```
Customer[] customer = CustomerFactory.listByName("Peter");
```

After executing the code, all rows which contain "Peter" in the CustomerName column in the Customer table will be retrieved and stored in an array of Customer object.

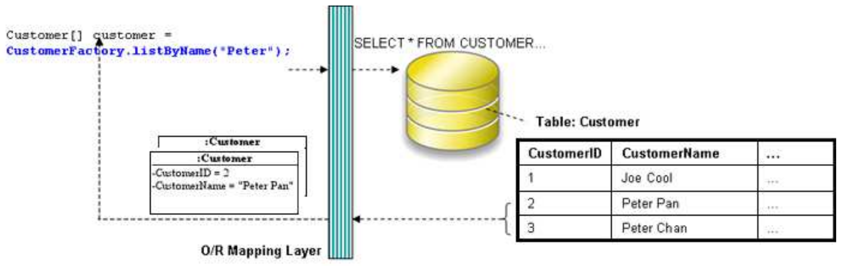


Figure 18.46 - Retrieve a list of records by ORM qualifier

Using Criteria Class

When generating the persistence class for each ORM-Persistable class defined in the object model, the corresponding criteria class can also be generated. Criteria class is a helper class which provides an additional way to specify the condition to retrieve records from the database. Refer to Using Criteria Class section for more detailed information on how to specify the conditions to query the database by the criteria class.

You can get the retrieved records from the criteria class in one of the two ways:

- Use the retrieval methods of the criteria class.

  For information on the retrieval methods provided by the criteria class, refer to the description of Loading Retrieved Records section.

- Use the retrieval by criteria methods of the factory class.

Table shows the method summary generated to the persistence class to retrieve records from the criteria class.

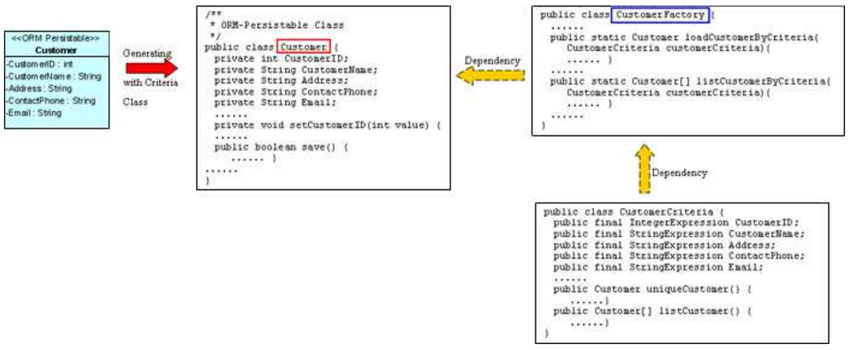
Class Type	Return Type	Method Name	Description
Factory Class	Class	loadClassByCriteria(ClassCriteria value)	Retrieve the single record that matches the specified conditions applied to the criteria class.
Factory Class	Class[]	listClassByCriteria(ClassCriteria value)	Retrieve the records that match the specified conditions applied to the criteria class.

Table 18.24

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
/**
 * ORM-Persistable Class
 */
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
  ......
  public boolean save() {
     ...... }
......
}
```

```
public class CustomerFactory {
  ......
  public static Customer loadCustomerByCriteria(
     CustomerCriteria customerCriteria){
     ...... }
  ......
  public static Customer[] listCustomerByCriteria(
     CustomerCriteria customerCriteria){
     ...... }
  ......
}
```

```
public class CustomerCriteria {
  public final IntegerExpression CustomerID;
  public final StringExpression CustomerName;
  public final StringExpression Address;
  public final StringExpression ContactPhone;
  public final StringExpression Email;
  ......
  public Customer uniqueCustomer() {
     ......}
  public Customer[] listCustomer() {
     ......}
}
```

Figure 18.47 - Mapping with Criteria Class

To retrieve records from the Criteria Class in one of the two ways:

- By Load method

  ```
  CustomerCriteria customerCriteria = new CustomerCriteria();
  customerCriteria.CustomerName.like("%Peter%");
  Customer customer = CustomerFactory.loadCustomerByCriteria(customerCriteria);
  ```

  After executing the code, the first occurrence of "Peter" in the CustomerName column in the Customer table will be loaded to the object identified as customer.

- By List method

  ```
  CustomerCriteria customerCriteria = new CustomerCriteria();
  CustomerCriteria.CustomerName.like("%Peter%");
  Customer[] customer = CustomerFactory.listCustomerByCriteria(customerCriteria);
  ```

  After executing the code, all rows which contain "Peter" in the CustomerName column in the Customer table will be retrieved and stored in an array of Customer object.

Using POJO

Generating the persistence code with POJO, the persistence class will be generated only with the attributes and a pair of getter and setter methods for each attribute. As the persistence class is generated without the methods for database manipulation, the generated PersistentManager and PersistentSession classes are responsible for manipulating the database.

By also generating the corresponding Data Access Object (DAO) class for each ORM-Persistable class inside the defined package of orm package, you are allowed to use the generated DAO class as same as the DAO class generated from the DAO persistent API. For information on using the DAO class to manipulate the database, refer to the description in Using DAO section.

When using the DAO class generated with POJO persistent API, manipulate the database with the same persistent code of DAO class generated with DAO persistent API by replacing the DAO class with the DAO class inside the orm package.

Note: When using the DAO class generated with POJO persistent API, manipulate the database with the same persistent code of DAO class generated with DAO persistent API by replacing the DAO class with the DAO class inside the orm package.

The following class diagram shows the relationship between the client program, persistent object, persistent session and persistent manager.

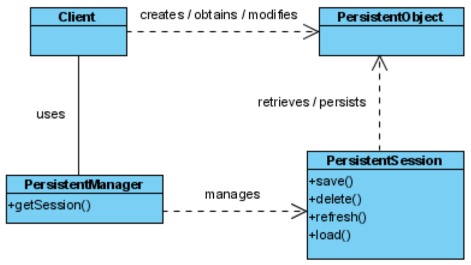


Figure 18.48 - Class Diagram for POJO

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram.

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram.

Example:

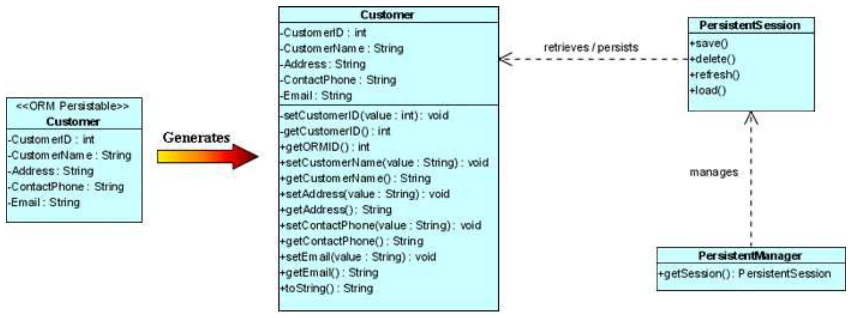


Figure 18.49 - Class Diagram for using POJO

From the above example, the Customer persistence class is generated with only the getter and setter methods for each attribute while the Customer persistent object is managed by the generated PersistentManager and PersistentSession.

Creating a Persistent Object

As a persistence class represents a table, an instance of a persistence class represents a record of the corresponding table. Creating a persistent object represents a creation of new record. With the support of the PersistentManager class, the newly created object can be saved as a new record to the database.

In order to insert a new row to the database table, implement the program with the following steps:

1. Create a new instance of the class by using the new operator.
2. Set the properties of the object by the persistence class.
3. Insert the object as a row to the database by the PersistentSession class.

Table shows the method summary of the persistence class and the PersistentSession class to be used for inserting a row to database.

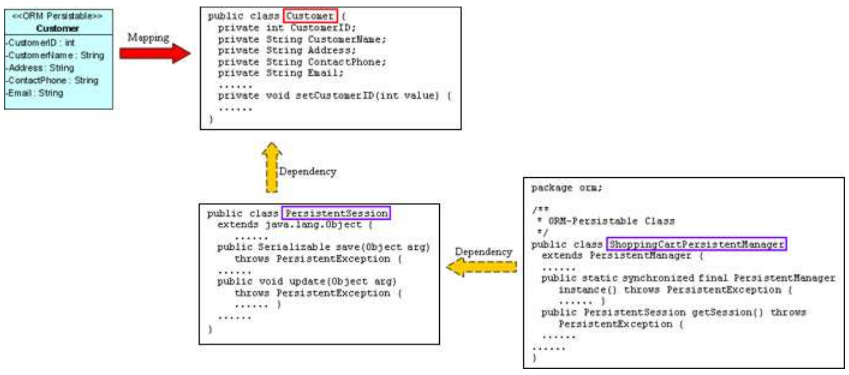
Class Type	Return Type	Method Name	Description
Persistence Class	void	setAttribute(DataType value)	Set the value to the property of an instance.
PersistentSession Class	Serializable	save(Object value)	Insert the object as a row to the database table.

Table 18.25

Remark:

1. **Attribute** should be replaced by the name of the attribute.
2. **DataType** should be replaced by the data type of the attribute defined in the object model.
3. **Object** represents the newly created instance of the persistence class to be added to the database.

Example:



```
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
  ......
}
```

```
public class PersistentSession
  extends java.lang.Object {
    ......
  public Serializable save(Object arg)
     throws PersistentException {
  ......
  public void update(Object arg)
     throws PersistentException {
     ...... }
  ......
}
```

```
package orm;

/**
 * ORM-Persistable Class
 */
public class ShoppingCartPersistentManager
  extends PersistentManager {
  ......
  public static synchronized final PersistentManager
     instance() throws PersistentException {
     ...... }
  public PersistentSession getSession() throws
     PersistentException {
  ......
......
}
```

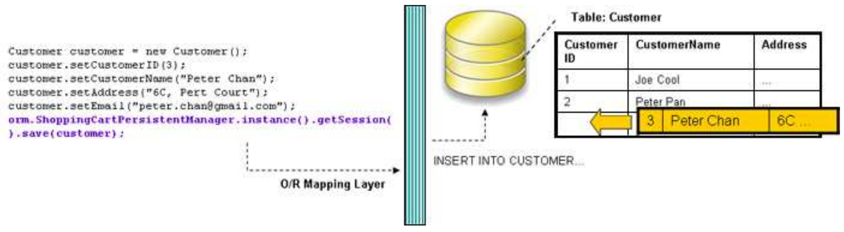
Figure 18.50 - Mapping with POJO

From the above example, an ORM-Persistable object model of Customer maps to an ORM-Persistable class with a pair of getter and setter methods for each attributes. With the PersistentManager class, the PersistentSession object can assist in inserting the instance as a row of the database table.

To insert a new Customer record to the table, Customer of the database, the following lines of code should be implemented.

```
Customer customer = new Customer();
customer.setCustomerID(3);
customer.setCustomerName("Peter Chan");
customer.setAddress("6C, Pert Court");
customer.setEmail("peter.chan@gmail.com");
customer.save();
orm.ShoppingCartPersistentManager.instance().getSession().save(customer);
```

After executing these lines of code, a row of record is inserted to the database table.



```
Customer customer = new Customer();
customer.setCustomerID(3);
customer.setCustomerName("Peter Chan");
customer.setAddress("6C, Pert Court");
customer.setEmail("peter.chan@gmail.com");
orm.ShoppingCartPersistentManager.instance().getSession(
).save(customer);
```

Customer ID	CustomerName	Address
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	6C ...

Figure 18.51 - Insert record with using POJO

Loading a Persistent Object

As an instance of a persistence class represents a record of a table, a record retrieved from the table will be stored as an object. By specifying the query-like condition to the PersistentManager class, records can be retrieved from the database.

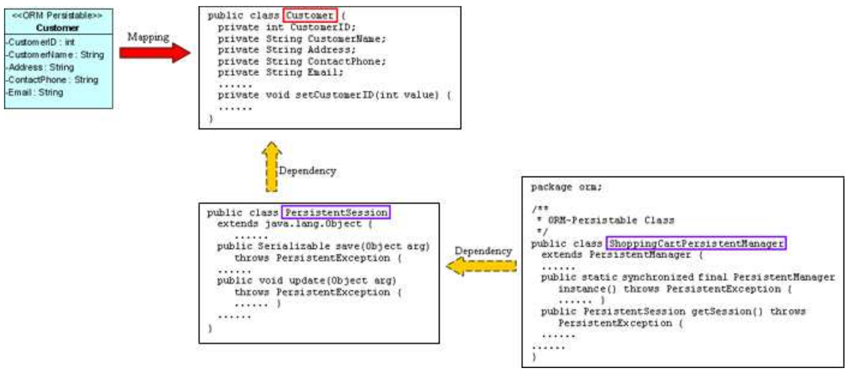
To retrieve a record, simply get the session by PersistentManager class and retrieve the records by defining the condition for query to the PersistentSession object and specify to return a single unique record.

Table shows the method summary of the PersistentSession class to be used for managing the retrieval of records

Class Type	Return Type	Method Name	Description
PersistentSession Class	Query	createQuery(String arg)	Retrieve the records matched with the user defined condition and ordered by a specified attribute.

Table 18.26

Example:



```
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
  ......
}
```

```
public class PersistentSession
  extends java.lang.Object {
     ......
  public Serializable save(Object arg)
     throws PersistentException {
  ......
  public void update(Object arg)
     throws PersistentException {
     ...... }
  ......
}
```

```
package orm;

/**
 * ORM-Persistable Class
 */
public class ShoppingCartPersistentManager
  extends PersistentManager {
  ......
  public static synchronized final PersistentManager
     instance() throws PersistentException {
     ...... }
  public PersistentSession getSession() throws
     PersistentException {
  ......
......
}
```

Figure 18.52 - Mapping for POJO

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable class with a pair of getter and setter methods for each attribute. The PersistentManager class gets the PersistentSession object which helps to query the database by specifying a user defined condition.

To retrieve a record from the Customer table, the following line of code should be implemented.

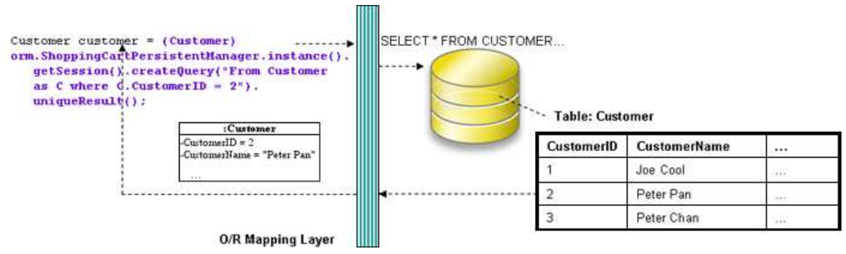
Loading an object by specifying a user defined condition with a passing value typed as "int":

```
Customer customer = (Customer)
orm.ShoppingCartPersistentManager.instance().getSession().createQuery("From Customer as C
where C.CustomerID = 2").uniqueResult();
```

Loading an object by specifying a user defined condition with a passing value typed as "String":

```
Customer customer = (Customer)
orm.ShoppingCartPersistentManager.instance().getSession().createQuery("From Customer as C
where C.CustomerName = 'Peter Pan'").uniqueResult();
```

After executing the code, a matched row is retrieved and loaded to a Customer object.



```
Customer customer = (Customer)
orm.ShoppingCartPersistentManager.instance().
   getSession().createQuery("From Customer
   as C where C.CustomerID = 2").
   uniqueResult();
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 18.53 - retrieve a record from database

Updating a Persistent Object

As a record can be retrieved from the database table and loaded as an instance of the persistence class, updating a record can be achieved by setting a new value to the attribute by its setter method and saving the new value to the database.

In order to update a record, you have to retrieve the row which will be updated, and then set a new value to the property, finally update the database record. To update the record, implement the program with the following steps:

1. Retrieve a record from database table and load as an object by the PersistentManager class.
2. Set the updated value to the property of the object by the persistence class.
3. Save the updated record to the database by the PersistentManager class.

Table shows the method summary of the persistence class and PersistentSession class to be used for updating a record

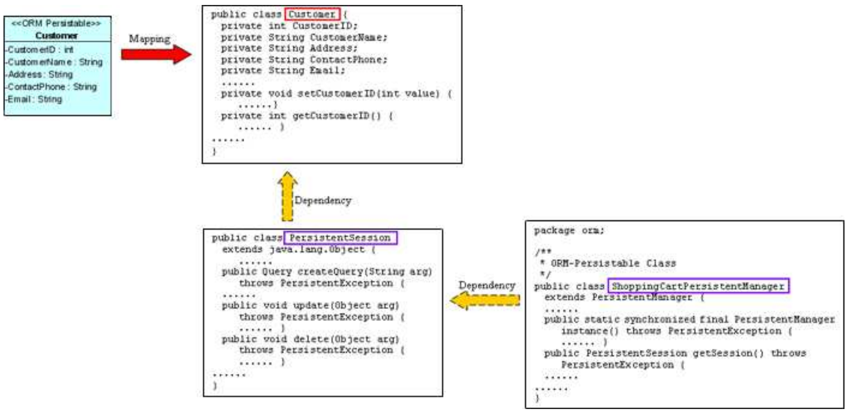
Class Type	Return Type	Method Name	Description
Persistence Class	void	setAttribute(DataType value)	Set the value to the property of an instance.
PersistentSession Class	void	update(Object arg)	Update the value to database.

Table 18.27

Remark:

1. **Attribute** should be replaced by the name of the attribute.
2. **DataType** should be replaced by the data type of the attribute defined in the object model.
3. **Object** represents the instance of the persistence class to be updated to the corresponding database record.

Example:



```
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
    ......}
  private int getCustomerID() {
    ...... }
......
}
```

```
public class PersistentSession
  extends java.lang.Object {
    ......
  public Query createQuery(String arg)
    throws PersistentException {
  ......
  public void update(Object arg)
    throws PersistentException {
    ...... }
  public void delete(Object arg)
    throws PersistentException {
    ...... }
......
}
```

```
package orm;

/**
 * ORM-Persistable Class
 */
public class ShoppingCartPersistentManager
  extends PersistentManager {
  ......
  public static synchronized final PersistentManager
    instance() throws PersistentException {
    ...... }
  public PersistentSession getSession() throws
    PersistentException {
  ......
......
}
```

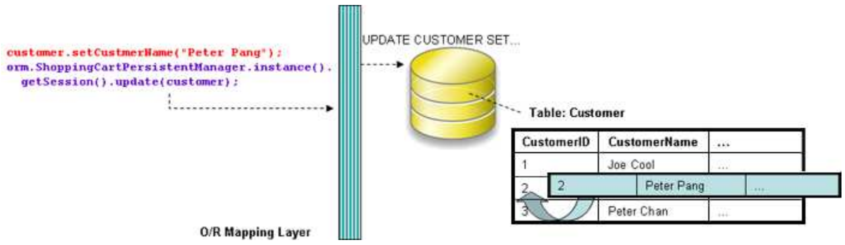
Figure 18.54 - Mapping for create query

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable class with a pair of getter and setter methods for each attributes. The PersistentManager class gets the PersistentSession object which helps to save the updated record to database.

To update a Customer record, the following line of code should be implemented.

```
customer.setCustomerName("Peter Pang");
orm.ShoppingCartPersistentManager.instance().getSession().update(customer);
```

After executing the above lines of code, the customer name is updated to "Peter Pang" in the database.



```
customer.setCustmerName("Peter Pang");
orm.ShoppingCartPersistentManager.instance().
  getSession().update(customer);
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pang	...
3	Peter Chan	...

Figure 18.55 - Insert record into database

Deleting a Persistent Object

As a record can be retrieved from the table and loaded as an object of the persistence class, the record can be deleted with the help of the PersistentManager class.

In order to delete a record, implement the program with the following steps:

1. Retrieve a record from database table and load as an object by the PersistentManager class.
2. Delete the retrieved record by the PersistentManager class.

Table shows the method summary of the PersistentSession class to be used for deleting a record from database

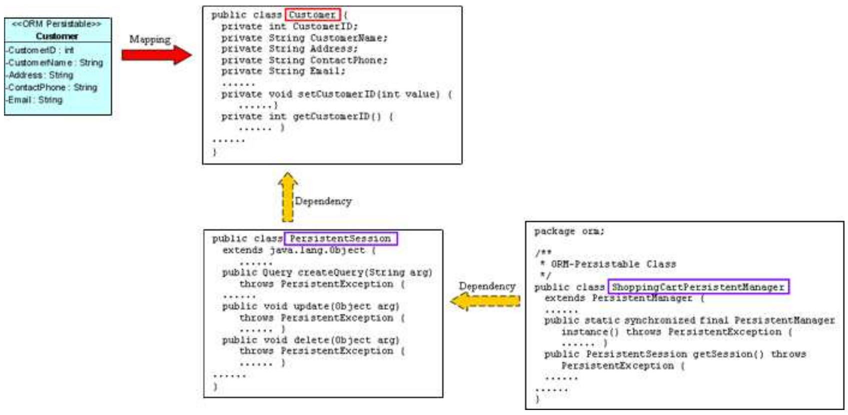
Class Type	Return Type	Method Name	Description
PersistentSession Class	void	delete(Object arg)	Delete the current instance.

Table 18.28

Remark:

1. **Object** represents the instance of the persistence class corresponding to a record that will be deleted from the database.

Example:



```
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
     ......}
  private int getCustomerID() {
     ...... }
......
}
```

```
public class PersistentSession
  extends java.lang.Object {
     ......
  public Query createQuery(String arg)
     throws PersistentException {
  ......
  public void update(Object arg)
     throws PersistentException {
     ...... }
  public void delete(Object arg)
     throws PersistentException {
     ...... }
......
}
```

```
package orm;

/**
 * ORM-Persistable Class
 */
public class ShoppingCartPersistentManager
  extends PersistentManager {
  ......
  public static synchronized final PersistentManager
     instance() throws PersistentException {
     ...... }
  public PersistentSession getSession() throws
     PersistentException {
  ......
......
}
```

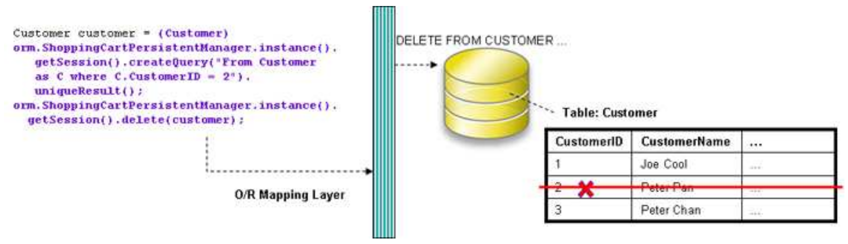
Figure 18.56 - Mapping for delete records

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable class with a pair of getter and setter methods for each attributes. The PersistentManager class gets the PersistentSession object which helps to delete the record from database.

To delete a Customer record, the following line of code should be implemented.

```
Customer customer = (Customer)
orm.ShoppingCartPersistentManager.instance().getSession().createQuery("From Customer as C
where C.CustomerID = 2").uniqueResult();
orm.ShoppingCartPersistentManager.instance().getSession().delete(customer);
```

After executing the above lines of code, the specified customer record is deleted from the database.



```
Customer customer = (Customer)
orm.ShoppingCartPersistentManager.instance().
   getSession().createQuery("From Customer
   as C where C.CustomerID = 2").
   uniqueResult();
orm.ShoppingCartPersistentManager.instance().
  getSession().delete(customer);
```

CustomerID	CustomerName	...
1	Joe Cool	...
~~2~~	~~Peter Pan~~	~~...~~
3	Peter Chan	...

Figure 18.57 - Delete a record from database

Querying

It is well known that the database is enriched with information. In some cases, information may be retrieved from the database to assist another function of the application, such as retrieving the unit price of products for calculating the total amount for a purchase order.

As the persistence class is not capable of retrieving records from the database, the PersistentManager class makes use of the PersistentSession object to retrieve records from the database.

Loading a Collection of Persistent Objects

As the database table usually contains many records, you may want to query the tables by a specified condition. The PersistentManager and PersistentSession support querying the database, the matched records will be retrieved and loaded to a list of objects.

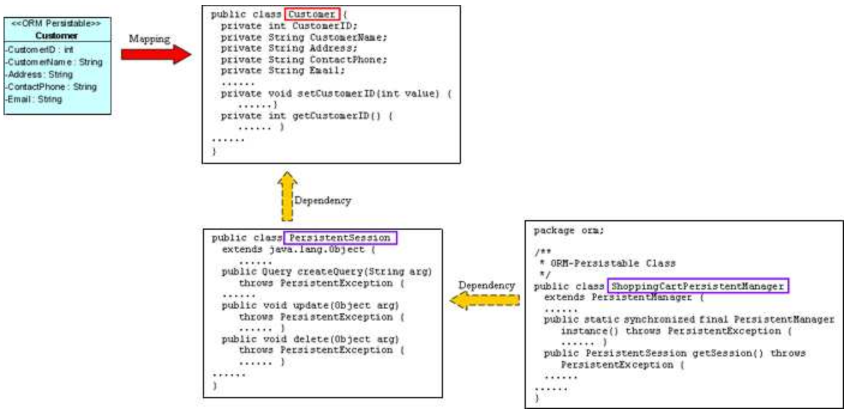
In order to retrieve records from the table, simply get the session by PersistentManager class and retrieve the records by defining the query condition to the PersistentSession object and specify to return a list of matched records.

Table shows the method summary of the PersistentSession class to be used for managing the retrieval of records.

Class Type	Return Type	Method Name	Description
PersistentSession Class	Query	createQuery(String arg)	Retrieve the records matched with the user defined condition and ordered by a specified attribute.

Table 18.29

Example:



```
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
     ......}
  private int getCustomerID() {
     ...... }
......
}
```

```
public class PersistentSession
  extends java.lang.Object {
     ......
  public Query createQuery(String arg)
     throws PersistentException {
  ......
  public void update(Object arg)
     throws PersistentException {
     ...... }
  public void delete(Object arg)
     throws PersistentException {
     ...... }
......
}
```

```
package orm;

/**
 * ORM-Persistable Class
 */
public class ShoppingCartPersistentManager
  extends PersistentManager {
  ......
  public static synchronized final PersistentManager
     instance() throws PersistentException {
     ...... }
  public PersistentSession getSession() throws
     PersistentException {
  ......
......
}
```

Figure 18.58 - Mapping with using POJO

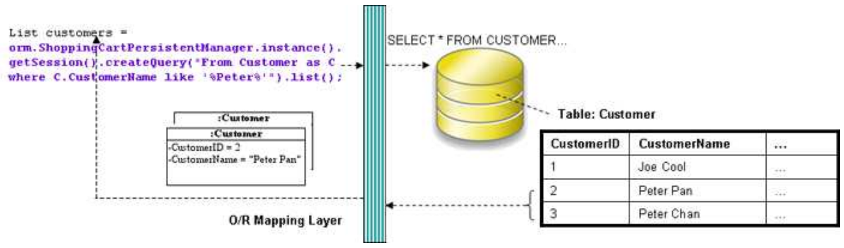
From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable class generated with a pair of getter and setter methods for each attribute. The PersistentManager class gets the PersistentSession object which helps to query the database by giving a user defined condition.

To retrieve records from the Customer table, the following line of code should be implemented.

Loading objects by specifying a user defined condition with a passing value typed as "String":

```
List customers = orm.ShoppingCartPersistentManager.instance().getSession().createQuery("From 
Customer as C where C.CustomerName like '%Peter%'").list();
```

After executing the code, the matched rows are retrieved and loaded to a list containing the Customer objects.



```
List customers =
orm.ShoppingCartPersistentManager.instance().
getSession().createQuery("From Customer as C
where C.CustomerName like '%Peter%'").list();
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

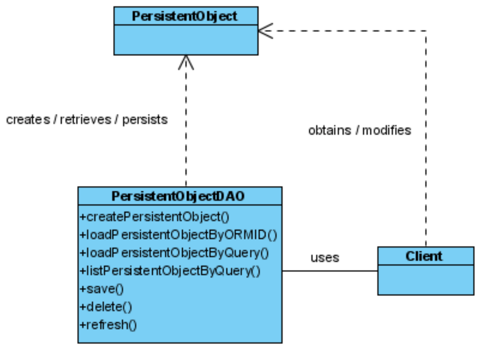
Figure 18.59 - Retrieve a list of records

Using DAO

Generating the persistence code with data access object (DAO), not only the persistence class will be generated, but also the corresponding DAO class for each ORM-Persistable class.

The generated persistence class using DAO persistent API is as same as that using POJO; that is, the persistence class contains attributes and a pair of getter and setter methods for each attribute only. Instead of using the persistence class to manipulate the database, the DAO class supports creating a new instance for the addition of a new record to the database, and retrieving record(s) from the database, updating and deleting the records.

The following class diagram shows the relationship between the client program, persistent object and DAO object.



PersistentObject
creates / retrieves / persists
obtains / modifies
PersistentObjectDAO
+createPersistentObject()
+loadPersistentObjectByORMID()
+loadPersistentObjectByQuery()
+listPersistentObjectByQuery()
+save()
+delete()
+refresh()
uses
Client

Figure 18.60 - Class Diagram for DAO

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram while the PersistentObjectDAO refers to the generated DAO class of the ORM-Persistable class. For example, the CustomerDAO class persists with the Customer persistent object and the database.

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram while the PersistentObjectDAO refers to the generated DAO class of the ORM-Persistable class. For example, the CustomerDAO class persists with the Customer persistent object and the database.

Example:

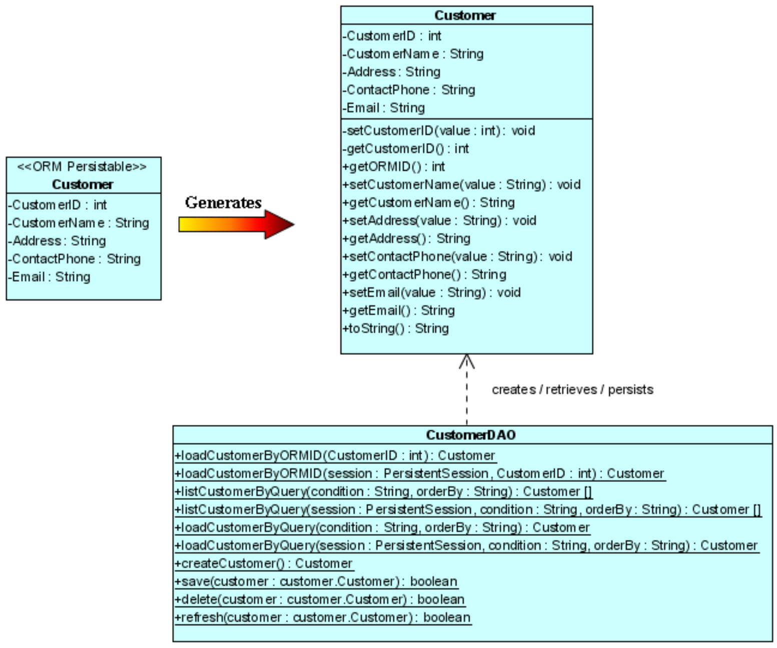

<<ORM Persistable>>
Customer
-CustomerID : int
-CustomerName : String
-Address : String
-ContactPhone : String
-Email : String
Generates
Customer
-CustomerID : int
-CustomerName : String
-Address : String
-ContactPhone : String
-Email : String
-setCustomerID(value : int) : void
-getCustomerID() : int
+getORMID() : int
+setCustomerName(value : String) : void
+getCustomerName() : String
+setAddress(value : String) : void
+getAddress() : String
+setContactPhone(value : String) : void
+getContactPhone() : String
+setEmail(value : String) : void
+getEmail() : String
+toString() : String
creates / retrieves / persists
CustomerDAO
+loadCustomerByORMID(CustomerID : int) : Customer
+loadCustomerByORMID(session : PersistentSession, CustomerID : int) : Customer
+listCustomerByQuery(condition : String, orderBy : String) : Customer []
+listCustomerByQuery(session : PersistentSession, condition : String, orderBy : String) : Customer []
+loadCustomerByQuery(condition : String, orderBy : String) : Customer
+loadCustomerByQuery(session : PersistentSession, condition : String, orderBy : String) : Customer
+createCustomer() : Customer
+save(customer : customer.Customer) : boolean
+delete(customer : customer.Customer) : boolean
+refresh(customer : customer.Customer) : boolean

Figure 18.61 - Class Diagram for DAO

From the above example, the CustomerDAO class supports the creation of Customer persistent object, the retrieval of Customer records from the Customer table while the Customer persistence class supports the update and deletion of customer record.

Creating a Persistent Object

An instance of a persistence class represents a record of the corresponding table, creating a persistent object supports the addition of new record to the table.

In order to add a new row to the database table, implement the program with the following steps:

1. Create a new instance of the class by the DAO class.
2. Set the properties of the object by the persistence class.
3. Insert the object as a row to the database by the DAO class.

Table shows the method summary of the DAO and persistence classes to be used for inserting a row to database.

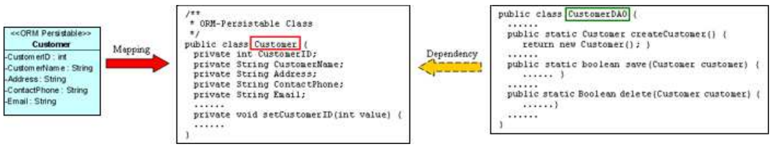
Class Type	Return Type	Method Name	Description
DAO Class	Class	createClass()	Create a new instance of the class.
Persistence Class	void	setAttribute(DataType value)	Set the value to the property of an instance.
DAO Class	boolean	save(Class value)	Insert the object as a row to the database table.

Table 18.30

Remark:

1. **Class** should be replaced by the name of the generated persistence class.
2. **Attribute** should be replaced by the name of the attribute.
3. **DataType** should be replaced by the data type of the attribute defined in the object model.

Example:



```
/**
 * ORM-Persistable Class
 */
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
  ......
}
```

```
public class CustomerDAO {
  ......
  public static Customer createCustomer() {
     return new Customer(); }
  ......
  public static boolean save(Customer customer) {
     ...... }
  ......
  public static Boolean delete(Customer customer) {
     ......}
  ......
}
```

Figure 18.62 - Mapping with DAO

From the above example, an ORM-Persistable object model of Customer maps to an ORM-Persistable class with methods for setting the properties. An ORM-Persistable DAO class is generated supporting the creation of persistent object and adding it into the database.

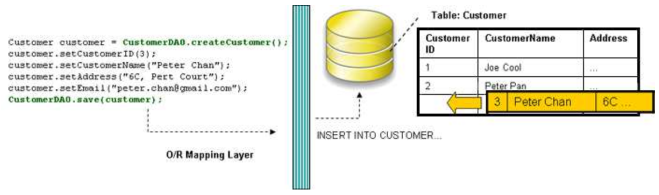
To add a new Customer record to the table, Customer of the database, the following lines of code should be implemented.

```
Customer customer = CustomerDAO.createCustomer();
customer.setCustomerID(3);
customer.setCustomerName("Peter Chan");
customer.setAddress("6C, Pert Court");
customer.setEmail("peter.chan@gmail.com");
CustomerDAO.save(customer);
```

After executing these lines of code, a row of record is inserted to the database table.

An alternative way to create a new instance of the class is using the new operator:
Class c = new Class();

From the above example, Customer customer = CustomerDAO.createCustomer() can be replaced by Customer customer = new Customer() to create a Customer object.



```
Customer customer = CustomerDAO.createCustomer();
customer.setCustomerID(3);
customer.setCustomerName("Peter Chan");
customer.setAddress("6C, Pert Court");
customer.setEmail("peter.chan@gmail.com");
CustomerDAO.save(customer);
```

Customer ID	CustomerName	Address
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	6C ...

Figure 18.63 - Insert a record by using DAO

Loading a Persistent Object

As an instance of a persistence class represents a record of the table, a record retrieved from the table will be stored as an object.

In order to retrieve a record from the database table, you have to specify the condition for finding the record. To retrieve a record, implement the program with the following steps:

1. Specify the condition for searching the record by the DAO class.
2. Load the retrieved record to an object.

Table shows the method summary of the DAO class to be used for retrieving a record from database

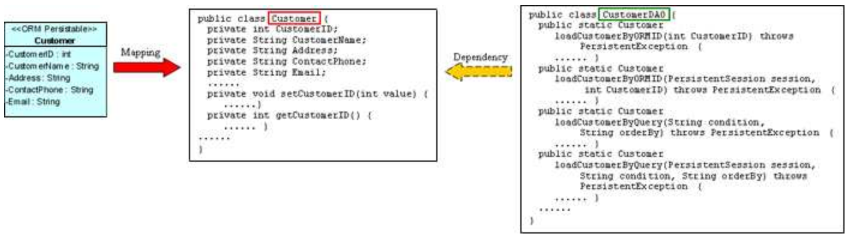
Class Type	Return Type	Method Name	Description
DAO Class	Class	loadClassByORMID(DataType PrimaryKey)	Retrieve a record matching with the specified value of primary key.
DAO Class	Class	loadClassByORMID(PersistentSession session, DataType PrimaryKey)	Retrieve a record matching with the specified value of primary key and specified session.
DAO Class	Class	loadClassByQuery(String condition, String orderBy)	Retrieve the first record matching the user defined condition while the matched records are ordered by a specified attribute.
DAO Class	Class	loadClassByQuery(PersistentSession session, String condition, String orderBy)	Retrieve the first record matching the user defined condition and specified session while the matched records are ordered by a specified attribute.

Table 18.31

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
     ......}
  private int getCustomerID() {
     ...... }
......
}
```

```
public class CustomerDAO {
 public static Customer
    loadCustomerByORMID(int CustomerID) throws
       PersistentException {
    ...... }
 public static Customer
    loadCustomerByORMID(PersistentSession session,
       int CustomerID) throws PersistentException {
    ...... }
 public static Customer
    loadCustomerByQuery(String condition,
       String orderBy) throws PersistentException {
    ...... }
 public static Customer
    loadCustomerByQuery(PersistentSession session,
       String condition, String orderBy) throws
       PersistentException {
    ...... }
 ......
}
```

Figure 18.63 - Mapping load methods in DAO

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable Java class. A DAO class for the customer class is generated with methods for retrieving a matched record.

To retrieve a record from the Customer table, the following line of code should be implemented.

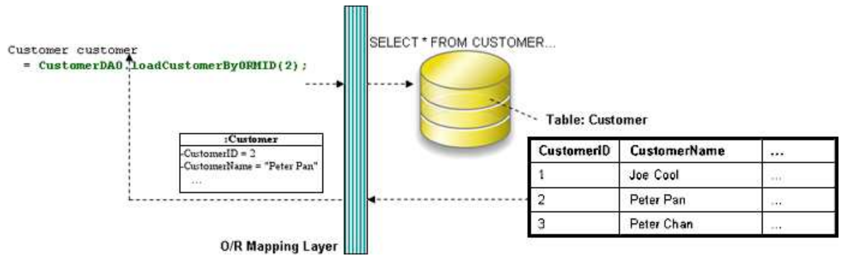
Loading an object by passing the value of primary key:

```
Customer customer = CustomerDAO.loadCustomerByORMID(2);
```

Loading an object by specifying a user defined condition:

```
Customer customer = CustomerDAO.loadCustomerByQuery("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

After executing the code, a matched row is retrieved and loaded to a Customer object.



```
Customer customer
  = CustomerDAO.loadCustomerByORMID(2);
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 18.64 - Code for load a record with DAO

Updating a Persistent Object

The DAO class not only supports the retrieval of record, but also the update on the record with the assistance of the setter method of the persistence class.

In order to update a record, you have to retrieve the row to be updated first, and then set a new value to the property, and finally save the updated record to the database. To update the record, implement the program with the following steps:

1. Retrieve a record from database table and load as an object by the DAO class.
2. Set the updated value to the property of the object by the persistence class.
3. Save the updated record to the database by the DAO class.

Table shows the method summary of the persistence class and DAO class to be used for updating a record

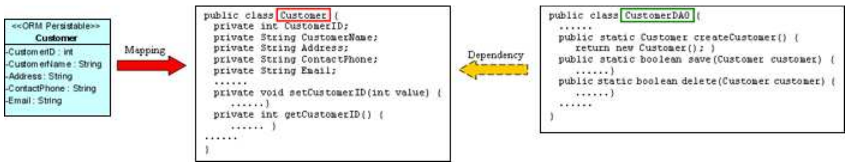
Class Type	Return Type	Method Name	Description
Persistence Class	void	setAttribute(DataType value)	Set the value to the property of an instance.
DAO Class	boolean	save(Class value)	Update the value to database.

Table 18.32

Remark:

1. **Attribute** should be replaced by the name of the attribute.
2. **DataType** should be replaced by the data type of the attribute defined in the object model.
3. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
     ......}
  private int getCustomerID() {
     ...... }
......
}
```

```
public class CustomerDAO {
  ......
  public static Customer createCustomer() {
     return new Customer(); }
  public static boolean save(Customer customer) {
     ......}
  public static boolean delete(Customer customer) {
     ......}
  ......
}
```

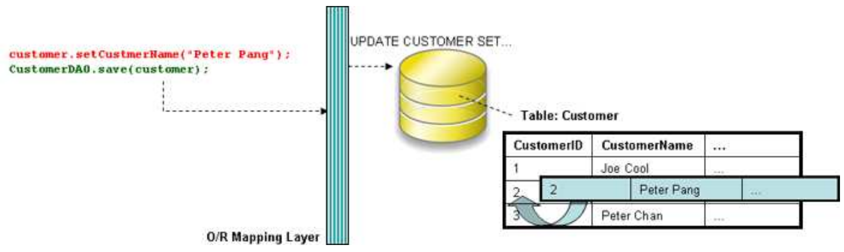
Figure 18.65 - Mapping save method in DAO

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable Java class with a pair of getter and setter methods for each attributes. The generated DAO class provides method for updating the record.

To update a Customer record, the following lines of code should be implemented.

```
customer.setCustomerName("Peter Pang");
CustomerDAO.save(customer);
```

After executing the above lines of code, the customer name is updated to "Peter Pang" in the database.



```
customer.setCustmerName("Peter Pang");
CustomerDAO.save(customer);
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pang	...
3	Peter Chan	...

Figure 18.66 - Code for update a record

Deleting a Persistent Object

The DAO class also supports deleting a record from the database. In order to delete a record, implement the program with the following steps:

1. Retrieve a record from database table and load as an object by the DAO class.
2. Delete the retrieved record by the DAO class.

Table shows the method summary of the DAO class to be used for deleting a record from database

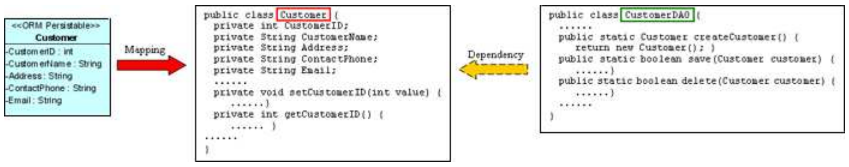
Class Type	Return Type	Method Name	Description
DAO Class	boolean	delete(**Class** value)	Delete the current instance.

Table 18.33

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
     ......}
  private int getCustomerID() {
     ...... }
......
}
```

```
public class CustomerDAO {
  ......
  public static Customer createCustomer() {
     return new Customer(); }
  public static boolean save(Customer customer) {
     ......}
  public static boolean delete(Customer customer) {
     ......}
  ......
}
```

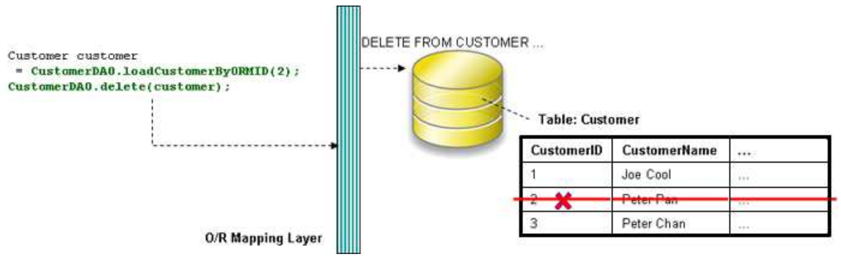
Figure 18.67 - Mapping delete methods in DAO

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable Java class generated with the methods for deleting the specified record from the database.

To delete a Customer record, the following lines of code should be implemented.

```
Customer customer = CustomerDAO.loadCustomerByORMID(2);
CustomerDAO.delete();
```

After executing the above code, the specified customer record is deleted from the database.



```
Customer customer
 = CustomerDAO.loadCustomerByORMID(2);
CustomerDAO.delete(customer);
```

CustomerID	CustomerName	...
1	Joe Cool	...
~~2~~	~~Peter Pan~~	~~...~~
3	Peter Chan	...

Figure 18.68 - Code for delete record by using DAO

Querying

It is well known that the database is enriched with information. In some cases, information may be retrieved from the database to assist another function of the application, such as retrieving the unit price of products for calculating the total amount for a purchase order.

As the DAO class is capable of querying the database, records can be retrieved by specifying the searching condition.

Loading a Collection of Persistent Objects

As the database table usually contains many records, you may want to query the tables by a specified condition. The generated DAO class supports querying the database, the matched records will be retrieved and loaded as an object array.

In order to retrieve records from the table, you have to specify the condition for querying. To retrieve a number of records, implement the program with the following steps:

1. Specify the condition for searching the record by the DAO class.
2. Load the retrieved records as an object array by the DAO class.

Table shows the method summary of the DAO class to be used for retrieving records from database table.

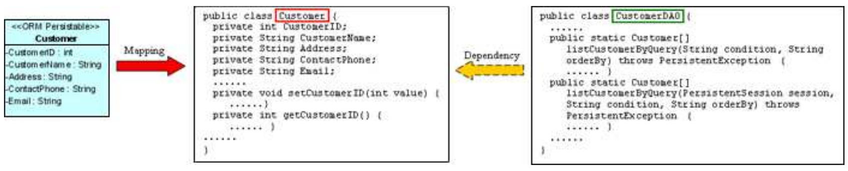
Class Type	Return Type	Method Name	Description
DAO Class	Class[]	listClassByQuery(String condition, String orderBy)	Retrieve the records matched with the user defined condition and ordered by a specified attribute.
DAO Class	Class[]	listClassByQuery(PersistentSession session, String condition, String orderBy)	Retrieve the records matched with the user defined condition and specified session and ordered by a specified attribute.

Table 18.34

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
     ......}
  private int getCustomerID() {
     ...... }
......
}
```

```
public class CustomerDAO {
  ......
  public static Customer[]
     listCustomerByQuery(String condition, String
     orderBy) throws PersistentException {
     ...... }
  public static Customer[]
     listCustomerByQuery(PersistentSession session,
     String condition, String orderBy) throws
     PersistentException {
     ...... }
  ......
}
```

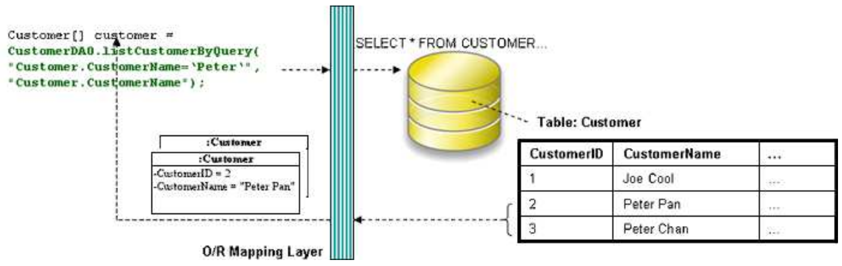
Figure 18.69 - Mapping list method in DAO

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable Java class. The DAO class for the customer is generated with methods for retrieving records.

To retrieve records from the Customer table, the following line of code should be implemented.

```
Customer[] customer = CustomerDAO.listCustomerByQuery("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

After executing the code, the matched rows are retrieved and loaded to an object array of Customer.



```
Customer[] customer =
CustomerDAO.listCustomerByQuery(
"Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

```
SELECT * FROM CUSTOMER...
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 18.70 - Code for retrieve a list of records using DAO

Using ORM Qualifier

ORM Qualifier is an additional feature which allows you to specify the extra data retrieval rules apart from the system pre-defined rules. The ORM Qualifier can be defined in order to generate persistence code. Refer to Defining ORM Qualifier in the Object Model chapter for more information.

By defining the ORM Qualifier in a class, the persistence DAO class will be generated with additional data retrieval methods, load and list methods.

Table shows the method summary generated to the DAO class by defining the ORM Qualifier.

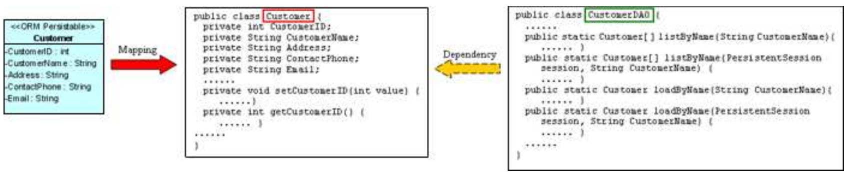
Class Type	Return Type	Method Name	Description
DAO Class	Class	loadByORMQualifier(DataType attribute)	Retrieve the first record that matches the specified value with the attribute defined in the ORM Qualifier.
DAO Class	Class	loadByORMQualifier (PersistentSession session, DataType attribute)	Retrieve the first record that matches the specified value with the attribute defined in the ORM Qualifier and specified session.
DAO Class	Class[]	listByORMQualifier (DataType attribute)	Retrieve the records that match the specified value with the attribute defined in the ORM Qualifier.
DAO Class	Class[]	listByORMQualifier (PersistentSession session, DataType attribute)	Retrieve the records that match the specified value with the attribute defined in the ORM Qualifier and specified session.

Table 18.35

Remark:

1. **Class** should be replaced by the name of the persistence class.
2. **ORMQualifier** should be replaced by the Name defined in the ORM Qualifier.
3. **DataType** should be replaced by the data type of the attribute which associated with the ORM Qualifier.
4. **attribute** is the specified value to be used for querying the table.

Example:



```
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
     ......}
  private int getCustomerID() {
     ...... }
......
}
```

```
public class CustomerDAO {
  ......
  public static Customer[] listByName(String CustomerName){
     ...... }
  public static Customer[] listByName(PersistentSession
     session, String CustomerName) {
     ...... }
  public static Customer loadByName(String CustomerName){
     ...... }
  public static Customer loadByName(PersistentSession
     session, String CustomerName) {
     ...... }
  ......
}
```

Figure 18.71 - Mapping load and list by qualifier methods in DAO

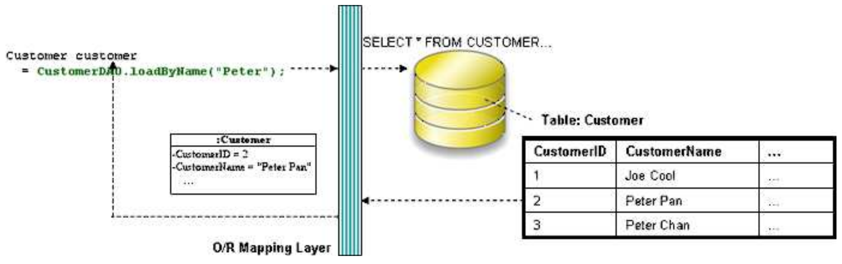
In the above example, a customer object model is defined with an ORM Qualifier named as Name and qualified with the attribute, CustomerName.

To query the Customer table with the ORM Qualifier in one of the two ways

- By Load method

```
Customer customer = CustomerDAO.loadByName("Peter");
```

After executing the code, the first occurrence of "Peter" in the CustomerName column in the Customer table will be loaded to the object identified as customer.



```
Customer customer
  = CustomerDAO.loadByName("Peter");
```

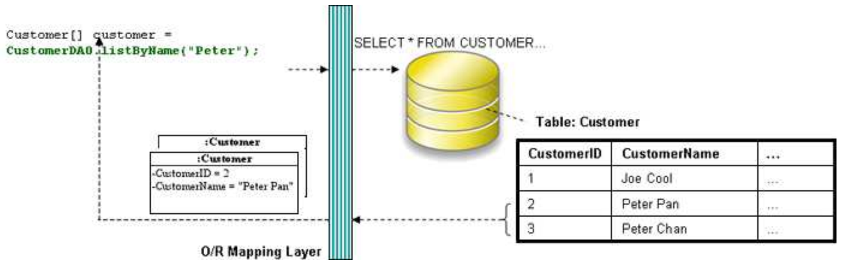
CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 18.72 - Code for load a record by qualifier

- By List method

```
Customer[] customer = CustomerDAO.lsitByName("Peter");
```

After executing the code, all rows which contain "Peter" in the CustomerName column in the Customer table will be retrieved and stored in an array of Customer object.



```
Customer[] customer =
CustomerDAO.listByName("Peter");
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 18.73 - Code for list a record by qualifier

Using Criteria Class

When generating the persistence class for each ORM-Persistable class defined in the object model, the corresponding criteria class can also be generated. Criteria class is a helper class which provides an additional way to specify the condition to retrieve records from the database. Refer to Using Criteria Class section for more detailed information on how to specify the conditions to query the database by the criteria class.

You can get the retrieved records from the criteria class in one of the two ways:

- Use the retrieval methods of the criteria class.

  For information on the retrieval methods provided by the criteria class, refer to the description of Loading Retrieved Records section.

- Use the retrieval by criteria methods of the DAO class.

Table shows the method summary generated to the persistence class to retrieve records from the criteria class

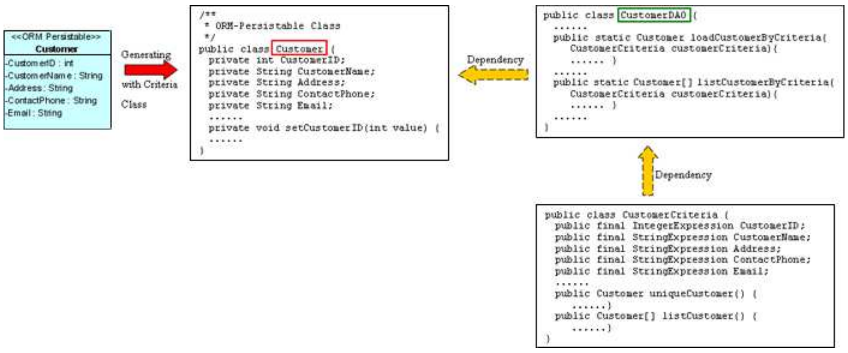
Class Type	Return Type	Method Name	Description
DAO Class	Class	loadClassByCriteria(ClassCriteria value)	Retrieve the single record that matches the specified conditions applied to the criteria class.
DAO Class	Class[]	listClassByCriteria(ClassCriteria value)	Retrieve the records that match the specified conditions applied to the criteria class.

Table 18.36

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
/**
 * ORM-Persistable Class
 */
public class Customer {
  private int CustomerID;
  private String CustomerName;
  private String Address;
  private String ContactPhone;
  private String Email;
  ......
  private void setCustomerID(int value) {
  ......
}
```

```
public class CustomerDAO {
  ......
  public static Customer loadCustomerByCriteria(
     CustomerCriteria customerCriteria){
     ...... }
  ......
  public static Customer[] listCustomerByCriteria(
     CustomerCriteria customerCriteria){
     ...... }
  ......
}
```

```
public class CustomerCriteria {
  public final IntegerExpression CustomerID;
  public final StringExpression CustomerName;
  public final StringExpression Address;
  public final StringExpression ContactPhone;
  public final StringExpression Email;
  ......
  public Customer uniqueCustomer() {
     ......}
  public Customer[] listCustomer() {
     ......}
}
```

Figure 18.74 - Mapping with DAO and Criteria Class

To retrieve records from the Criteria Class in one of the two ways:

- By Load method

  ```
  CustomerCriteria customerCriteria = new customerCriteria.CustomerName.like("%Peter%");
  Customer customer = CustomerDAO.loadCustomerByCriteria(customerCriteria);
  ```

  After executing the code, the first occurrence of "Peter" in the CustomerName column in the Customer table will be loaded to the object identified as customer.

- By List method

  ```
  CustomerCriteria customerCriteria = new CustomerCriteria.CustomerName.like("%Peter%");
  Customer[] customer = CustomerDAO.listCustomerByCriteria(customerCriteria);
  ```

  After executing the code, all rows which contain "Peter" in the CustomerName column in the Customer table will be retrieved and stored in an array of Customer object.

Using Criteria Class

As a database is normally enriched with information, there are two ways to retrieve records from the database by the generated persistent code. Firstly, the list and load methods to the persistence code are generated which retrieve matched records from the database with respect to the user defined condition. Secondly, a criteria class can be generated for each ORM-Persistable class which supports searching the records from the corresponding table.

By using the criteria class, it supports querying the database with multiple criteria. To generate the criteria class for query, simply check the Generate Criteria option before the generation of code. For more information on the setting of code generation, refer to Configuring Code Generation Setting for Java in the Implementation chapter.

Figure 18.75 - Generate Criteria checkbox

Having selected the Generate Criteria option for the code generation, a criteria class is generated in addition to the classes generated with respect to the persistent API. The generated criteria class is named as "ClassCriteria " in which Class is the name of the ORM Persistable class accordingly. The criteria class is generated with attributes, which are defined in the object model, with type of one of Expression with respect to the type of attribute defined in the object model and two operations for specifying the type of record retrieval.

In order to query the database by the criteria class, implement the program with the following steps:

1. Create an instance of the criteria class.
2. Apply restriction to the property of the class which specifies the condition for query.
3. Apply ordering to the property if it is necessary.
4. Set the range of the number of records to be retrieved if it is necessary.
5. Load the retrieved record(s) to an object or array.

Applying Restriction to Property

To apply the restriction to the property, implement with the following code template:

```
criteria.property.expression(parameter);
```

where criteria is the instance of the criteria class; property is the property of the criteria;
expression is the expression to be applied on the property; parameter is the parameter(s) of the expression.

Table shows the expression to be used for specifying the condition for query.

Expression	Description
eq(value)	The value of the property is equal to the specified value.
ne(value)	The value of the property is not equal to the specified value.
gt(value)	The value of the property is greater than to the specified value.
ge(value)	The value of the property is greater than or equal to the specified value.
lt(value)	The value of the property is less than the specified value.
le(value)	The value of the property is less than or equal to the specified value.
isEmpty()	The value of the property is empty.
isNotEmpty()	The value of the property is not empty.
isNull()	The value of the property is NULL.
isNotNull()	The value of the property is not NULL.
in(values)	The value of the property contains the specified values in the array.
between(value1, value2)	The value of the property is between the two specified values, value1 and value2.
like(value)	The value of the property matches the string pattern of value; use % in value for wildcard.
ilike(value)	The value of the property matches the string pattern of value, ignoring case differences.

Table 18.37

Sorting Retrieved Records

There are two types of ordering to sort the retrieved records, that is, ascending and descending order.

To sort the retrieved records with respect to the property, implement the following code template:

```
criteria.property.order(ascending_order);
```

where the value of ascending_order is either true or false. True refers to sort the property in ascending order while false refers to sort the property in descending order.

Setting the Number of Retrieved Records

To set the range of the number of records to be retrieved by using one of the two methods listed in the following table.

Method Name	Description
setFirstResult(int i)	Retrieve the i-th record from the results as the first result.
setMaxResult(int i)	Set the maximum number of retrieved records by specified value, i.

Table 18.38

Loading Retrieved Records

To load the retrieved record(s) to an object or array, use one of the two methods listed below.

Table shows the method summary of the criteria class to be used for retrieving records from database.

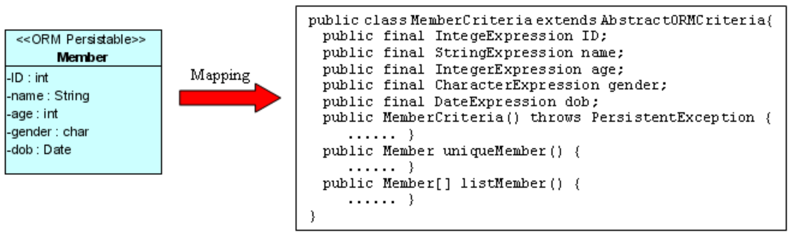
Return Type	Method Name	Description
Class	uniqueClass()	Retrieve a single record matching the specified condition(s) for the criteria; Throw exception if the number of retrieved record is not equal to 1.
Class[]	listClass()	Retrieve the records matched with the specified condition(s) for the criteria.

Table 18.39

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class MemberCriteria extends AbstractORMCriteria{
  public final IntegeExpression ID;
  public final StringExpression name;
  public final IntegerExpression age;
  public final CharacterExpression gender;
  public final DateExpression dob;
  public MemberCriteria() throws PersistentException {
     ...... }
  public Member uniqueMember() {
     ...... }
  public Member[] listMember() {
     ...... }
}
```

Figure 18.76 - Mapping with Criteria Class

Here is an object model of a Member class. After performing the generation of persistent code, an additional Java file is generated, named "MemberCriteria.java".

The following examples illustrate how to query the Member table using the MemberCriteria class:

- By specifying one criterion

  Retrieving a member record whose name is "John":

  ```
  MemberCriteria criteria = new MemberCriteria();
  Criteria.name.eq("John");
  Member member = criteria.uniqueMember();
  System.out.println(member);
  ```

  Retrieving all member records whose date of birth is between 1/1/1970 and 31/12/1985:

  ```
  MemberCriteria criteria = new MemberCriteria();
  criteria.dob.between(new GregorianCalendar(1970, 1, 1).getTime(), new
  GregorianCalendar(1985, 12, 31));
  Member[] members = criteria.listMember();
  for (int i=0; i < members.length; i++){
  	System.out.println(members[i]);
  }
  ```

- By specifying more than one criteria

  Retrieving all male member records whose age is between 18 and 24, ordering by the name:

  ```
  MemberCriteria criteria = new MemberCriteria();
  criteria.age.in(new int[]{18, 24});
  criteria.gender.eq('M');
  criteria.name.order(true); //true = ascending order; false = descending order
  Member[] members = criteria.listMember();
  for (int i=0; i < members.length; i++){
  	System.out.println(members[i]);
  }
  ```

Retrieving all member records whose name starts with "Chan" and age is greater than 30:

```
MemberCriteria criteria = new MemberCriteria();
criteria.name.like("Chan %");
criteria.age.gt(30);
Member[] members = criteria.listMember();
for (int i=0; i < members.length; i++){
       System.out.println(members[i]);
}
```

Using Transactions

A transaction is a sequence of operation. If the operation is performed successfully, the transaction is committed permanently. On the contrary, if the operation is performed unsuccessfully, the transaction is rolled back to the state of the last successfully committed transaction. Since database modifications must follow the "all or nothing" rule, transaction must be manipulated for every access to the database.

To create a transaction for atomicity by the following line of code:

```
PersistentTransaction t = PersistentManager.instance().getSession().beginTransaction();
```

To commit a transaction by the following line of code:

```
t.commit();
```

To rollback a transaction if there is any exception in the program by the following line of code:

```
t.rollback();
```

Using ORM Implementation

The generated persistent code exposes the ability of the Java class to access the database including the basic operations for add, update, delete and search. As the generated persistent code only provides basic operations for manipulating the persistent data, you may find it is insufficient and want to add extra logic to it. However, modifying the generated persistent code may cause your developing program malfunctioned. It is strongly recommended to add the extra logic by using ORM Implementation class to manipulate the persistent data.

An implementation class can be added to the object model. When generating persistent code, the implementation class will also be generated. You are allowed to implement the operation in the generated implementation class. As if the content of an implementation class has been modified, it will not be overwritten when re-generating the persistent code from the object model.

Inserting an ORM Implementation Class

1. Select a class stereotyped as ORM-Persistable that you want to add extra logic for manipulating the persistent data.

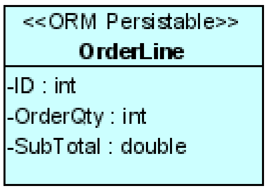


Figure 18.77 - ORM Persistable Class

2. Add a new operation to the class by right clicking the class element, selecting **Add** > **Operation**.

   A new operation is added, type the operation in the form of "operation_name(parameter_name: type) : return_type".

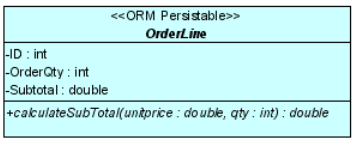
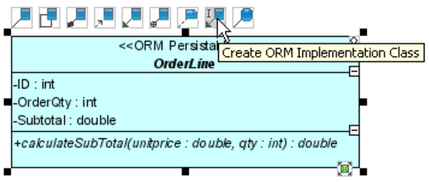


Figure 18.78 - ORM Persistable Class with operation

A new operation can be added by using the keyboard shortcut - ***Alt + Shift + O***. You can also edit the operation by double-clicking the operation name or pressing the ***F2*** button.

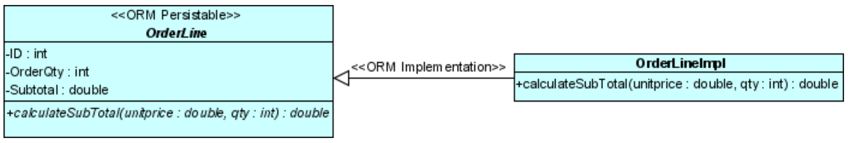
3. Drag the resource of **Create ORM Implementation Class** to the diagram pane.



<<ORM Persista
OrderLine
Create ORM Implementation Class
-ID : int
-OrderQty : int
-Subtotal : double
+calculateSubTotal(unitprice : double, qty : int) : double

*Figure 18.79 - **Create ORM Implementation Class** resource-centric*

A new implementation class is created and connected to the source class with generalization stereotyped as ORM Implementation. The source class becomes an abstract class with an abstract operation.



<<ORM Persistable>>
OrderLine
-ID : int
-OrderQty : int
-Subtotal : double
+calculateSubTotal(unitprice : double, qty : int) : double
<<ORM Implementation>>
OrderLineImpl
+calculateSubTotal(unitprice : double, qty : int) : double

Figure 18.80 - ORM Persistable Class and Implement Class

After transforming the object model to persistent code, an implementation class will be generated. The generated code for the implementation class is shown below:

```
package shoppingcart;

public class OrderLineImpl exteds OrderLine {
	public OrderLineImpl() {
		super();
	}

	public dobule calculateSubTotal(double unitprice, int qty) {
		//TODO: Implement Method
		throw new UnsupportedOperationException();
	}
}
```

The generated implementation class is the same no matter which persistent API is selected for code generation.

You can implement the logic to the method in the generated implementation class.

Code Sample

Here is a sample demonstrating the usage of the implementation class from the above object model.

The implementation class is implemented with the following lines of code:

```
package shoppingcart;

public class OrderLineImpl exteds OrderLine {
	public OrderLineImpl() {
		super();
	}

	public dobule calculateSubTotal(double unitprice, int qty) {
		return unitprice * qty
	}
}
```

The following lines of code demonstrate how to invoke the logic of the implementation class with the persistence class, OrderLine generated using static method persistent API.

```
OrderLine orderline = OrderLine.createOrderLine();
orderline.setID(1);
orderline.setOrderQty(5);
dobule subtotal = orderline.calculateSubTotal(300, 5);
orderline.setSubTotal(subTotal);
```

To invoke the logic of the implementation class by the persistent object, please refer to the Creating a Persistent Object for different types of persistent API.

Running the Sample Code

A set of sample files is generated which guides you how to implement the sample code and run the sample to see how the persistence classes work with the database if you have checked the option for samples before generating the persistent code. For more information on the setting of code generation, refer to Configuring Code Generation Setting for Java in the Implementation chapter.

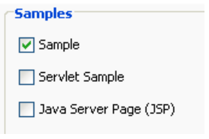

Samples
[✓] Sample
[ ] Servlet Sample
[ ] Java Server Page (JSP)

Figure 18.81 - Generate samples options

The common manipulation to the database includes inserting data, updating data, retrieving data and deleting data. As each type of manipulation can be performed individually, several sample files are generated to demonstrate how the persistence code handles each type of manipulation. The sample files are generated in the ormsamples directory of the specified output path.

There are six sample files demonstrating the database manipulation. They are identified by the type of manipulation and the name of the project.

Type of Manipulation	Sample File
Creating database table	Create%Project_Name%DatabaseSchema
Inserting record	Create%Project_Name%Data
Retrieving and updating record	RetrieveAndUpdate%Project_Name%Data
Deleting record	Delete%Project_Name%Data
Retrieving a number of records	List%Project_Name%Data
Dropping database table	Drop%Project_Name%DatabaseSchema

Table 18.40

Example:

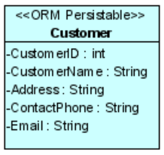


Figure 18.82 - ORM Persistable Class

Here is an object model of a Customer class inside a package of shoppingcart of the ShoppingCart project. By configuring the code generation setting with the factory class persistent API, Sample and Generate Criteria options checked, an ORM-Persistable Java class, a factory class and six sample files are generated.

The following are the examples of manipulating the Customer table in the database by the generated sample files.

Creating Database Table

If the database has not been created by the Generate Database facility, you can generate the database by executing the CreateShoppingCartDatabaseSchema class.

```
package ormsamples;

import org.orm.*;
public class CreateShoppingCartDatabaseSchema {
       public static void main(String args){
              try {
                     ORMDatabasesInitiator.createSchema(shoppingcart.ShoppingCartPersistentMana
                     ger.instance());
              }
              catch (Exception e) {
                     e.printStackTrace();
              }
       }
}
```

After executing the CreateShoppingCartDatabaseSchema class, the database is created with the Customer table.

Inserting Record

As the database has been generated with the Customer table, a new customer record can be added to the table. By implementing the createTestData() method of the CreateShoppingCartData class, a new customer record can be inserted to the table. Insert the following lines of codes to the createTestData() method to initialize the properties of the customer object which will be inserted as a customer record to the database.

```
public void createTestData() throws PersistentException {
       PersistentTransacrion t =
       shoppingcart.ShoppingCartPersistentManager.instance().getSession().beginTransaction();
       try {
              shoppint.Customer shoppingcartCustoemr =
              shoppingcart.CustomerFactory.createCustomer();
              // Initialize the properties of the persistent object
              shoppingcartCustomer.setCustomerName("Joe Cool");
              shoppingcartCustomer.setAddress("1212, Happy
              Building");shoppingcartCustomer.setContactPhone("23453256");
              shoppingcartCustomer.setEmail("joe@cool.com");
              shoppingcartCustomer.save();
              t.commit();
       }
       catch (Exception e){
              t.rollback();
              shoppingcart.ShoppingCartPersistentManager.instance().getSession().close();
       }
}
```

After running the sample code, a customer record is inserted into the Customer table.

Retrieving and Updating Record

In order to update a record, the particular record must be retrieved first. By modifying the following lines of code to the retrieveAndUpdateTestData() method of the RetrieveAndUpdateShoppingCartData class, the customer record can be updated.

```
public void createTestData() throws PersistentException {
       PersistentTransacrion t =
       shoppingcart.ShoppingCartPersistentManager.instance().getSession().beginTransaction();
       try {
              shoppint.Customer shoppingcartCustoemr =
              shoppingcart.CustomerFactory.loadCustomerByQuery("Customer.CustomerName='Joe
              Cool'", "Customer.CustomerName");
              // Update the properties of the persistent object
              shoppingcartCustomer.setContactPhone("28457126");
              shoppingcartCustomer.setEmail("joe.cool@gmial.com");
```

```
            shoppingcartCustomer.save();
            t.commit();
        }
        catch (Exception e){
            t.rollback();
            shoppingcart.ShoppingCartPersistentManager.instance().getSession().close();
        }
}
```

After running the sample code, the contact phone and email of the customer is updated.

Retrieving Record by ORM Qualifier

If the object model is defined with ORM Qualifier(s), the load method(s) for retrieving a particular record by ORM Qualifiers will be generated in the sample code as well.

From the example, the object model of Customer is defined with two ORM Qualifiers, named as Name and Phone, qualified with the attribute, CustomerName and ContactPhone respectively, two methods named as retrieveCustomerByName() and retrieveCustomerByPhone() are generated in the sample class. Modify the two methods as shown below.

```
public void retrieveCustomerByName() {
        System.out.println("Retrieving Customer by CustomerName...");
        // Please uncomment the follow line and fill in parameter(s)
        System.out.println(shoppingcart.CustomerFactory.loadByName("James
        Smith").getCustomerName());
}

public void retrieveCustomerByPhone() {
        System.out.println("Retrieving Customer by ContactPhone...");
        // Please uncomment the follow line and fill in parameter(s)
        System.out.println(shoppingcart.CustomerFactory.loadByPhone("62652610").getCustomerName
        ());
}
```

Retrieving Record by Criteria Class

If you have selected the Generate Criteria option before the generation of code, the Criteria class will be generated accordingly.

From the example, the Criteria class, named as CustomerCriteria is generated for the object model of Customer. A method named as retrieveByCriteria() is generated in the sample class. By following the commented guideline of the method, uncomment the code and modify as follows.

```
public void retrieveByCriteria() {
        System.out.println("Retrieving Customer by CustomerCriteria");
        shoppingcart.CustomerCriteria customerCriteria = new shoppingcart.CustomerCriteria();
        Please uncomment the follow line and fill in parameter(s)
        customerCriteria.CustomerID.eq(3);
        System.out.println(customerCriteria.uniqueCustomer().getCustomerName());
}
```

In order to run the above three methods, uncomment the statements (blue colored in the diagram below) in the main(String[] args) method.

```
public static void main(String[] args){
        try {
            RetrieveAndUpdateShoppingCartData retrieveAndUpdateShoppingCartData = new
            RetrieveAndUpdateShoppingCartData();
            try {
                  retrieveAndUpdateShoppingCartData.retrieveAndUpdateTestData();
                  retrieveAndUpdateShoppingCartData.retrieveCustomerByName();
                  retrieveAndUpdateShoppingCartData.retrieveCustomerByPhone();
                  retrieveAndUpdateShoppingCartData.retrieveByCriteria();
            }
            finally {
                  shoppingcart.ShoppingCartPersistentManager.instance().disposePersistentMan
                  ager();
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
}
```

Having executed the RetrieveAndUpdateShoppingCartData class, the contact phone and email of the customer is updated, and the retrieved customer records are shown in the system output. (Assuming several customer records were inserted to the database.)

Deleting Record

A customer record can be deleted from the database by executing the DeleteShoppingCartData class. Modify the following lines of code in the deleteTestData() method to delete the specified record from the database.

```
public void deleteTestData() throws PersistentException {
        PersistentTransaction t =
        shoppingcart.ShoppingcartPersistenetManager.instance().getSession().beginTrasaction();
        try {
                shoppingcart.Customer shoppingcartCustoemr =
                shoppingcart.CustomerFactory.loadCustomerByQuery("Customer.CustomerName='Harry
                Hong'", "Customer.CusomterName");
                shoppingcartCustomer.delete();
                t.commit();
        }
        catch (Exception e){
                t.rollback();
                shoppingcart.ShoppingCartPersistentManager.instance().getSession().close();
        }
}
```

After running the DeleteShoppingCartData sample, the customer record whose customer name is equals to "Harry Hong" will be deleted.

Retrieving a Number of Records

The list method of the generated persistence class supports retrieving a number of records from the database, By modifying the following lines of code to the listTestData() method of the ListShoppingCartData class, the customer whose name starts with "Chan" will be retrieved and the name will be displayed in the system output.

```
public void listTestData() throws PersistentException {
        System.out.println("Listing Custoemr...");
        shoppingcart.Customer[] shoppingcartCustoemrs =
        shoppingcart.CustomerFactory.listCustomerByQuery("Customer.CustomerName like 'Chan%'",
        "Customer.CusomterName");
        int length = Math.min(shoppingcartCustomers.length, ROW_COUNT);
        for (int i = 0; i < length; i++) {
                System.out.println(shoppingcartCustomers[i].getCustomerName());
        }
        System.out.println(length + " record(s) retrieved.");
}
```

Retrieving a Number of Records by ORM Qualifier

If the object model is defined with ORM Qualifier(s), the list method(s) for retrieving records by ORM Qualifiers will be generated in the sample code as well.

From the example, the object model of Customer is defined with two ORM Qualifiers, named as Name and Phone, qualified with the attribute, CustomerName and ContactPhone respectively, two methods named as listCustomerByName() and listCustomerByPhone() are generated in the sample class. Modify the two methods as shown below.

```
public void listCustomerByName() {
        System.out.println("Listing Customer by CustomerName...");
        // Please uncomment the follow lines and fill in parameter(s)shoppintcart.Customer[]
        customers = shoppingcart.CustomerFactory.listByName("Wong%");
        int length = customers == null ? 0 : Math.min(customers.length, ROW_COUNT);
        for (int i = 0; i < length; i++) {
                System.out.println(customers[i].getCustomerName());
        }
        System.out.println(length + record(s) retrieved.");
}
```

```
public void listCustomerByPhone() {
        System.out.println("Listing Customer by ContactPhone...");
        // Please uncomment the follow lines and fill in parameters
        shoppingcart.Customer[] customers = shoppingcart.CustomerFactory.listByPhone("26%");
        int length = custoemrs == null ? 0 : Math.min(customers.length, ROW_COUNT);
        for (int i =0; i < length; i++) {
                System.out.println(customers[i].getCustomerName());
        }
        System.out.println(length + " record(s) retrieved.");
}
```

Retrieving a Number of Records by Criteria

As the criteria class is generated for the Customer object model, a method named as ListByCriteria() is generated in the sample class. Modify the following lines of code.

```
public void listByCriteria() throws PersistentException {
        System.out.println("Listing Customer by Crieteria...");
        shoppingcart.CustomerCriteria customerCriteria = new shoppingcart.CustomerCriteria();
        //Please uncomment the follow line and fill in parameter(s)
        // customerCriteria.CustomerID.eq();
        customerCriteria.CustomerName.like("Cheung%");
        customerCriteria.setMaxResults(ROW_COUNT);
        shoppingcart.Customer[] customers = customerCriteria.listCustomer();
        int length = customers == null ? 0 : Math.min(customers.length, ROW_COUNT);
        for (int i = 0; i < length; i++) {
                System.out.println(customers[i].getCustomerName());
        }
        System.out.println(length + " Customer record(s) retrieved.");
}
```

In order to run these three methods, uncomment the statements (blue colored in the diagram below) in the main(String[] args) method.

```
public static void main(String[] args) {
        try {
                LsitShoppingCartData lsitShoppingCartData = new LsitShoppingCartData();
                try {
                        lsitShoppingCartData.listTestData();
                        listShoppingCartData.listCustomerByName();
                        listShoppingCartData.listCustomerByPhone();
                }
                finally {
                        shoppingcart.ShoppingCartPersistentManager.instance().disposePersistentMan
                        ager();
                }
        }
        catch (Exception e) {
                e.printStackTrace();
        }
}
```

Having executed the ListShoppingCartData class, the matched records are retrieved and displayed with the customer name according to the condition defined in the listTestData(), listCustomerByName(), listCustomerByPhone() and listByCriteria() methods.

Dropping Database Table

In some cases, you may want to drop the database tables and create the database tables again. By running the DropShoppingCartDatabaseSchema class first, all the database tables will be dropped accordingly.

```
package ormsamples;

import org.orm.*;
public class DropShoppingCartDatabaseSchema {
        public static void main(String[] args){
                try {
                        System.out.println("Are you sure to drop tables(s)? (Y/N)");
                        java.io.BufferReader reader = new java.io.BufferedReader(new
                    java.io.InputStreamReader(System.in));
                    if( (reader.readLine().trim().toUpperCase().equals("Y");){
                          ORMDatabaseInitiator.dropSchema(shoppingcart.ShoppingCartPersistent
                          Manager());
                    }
              }
              catch (Exception e){
                    e.printStackTrace();
              }
        }
}
```

Running the Script File

The script file can be generated for you, which allows you to compile all generated Java classes, both ORM-Persistable classes and sample files and directly execute the sample files to test the sample result. In order to generate the script files, you have to check the option for script files before generating the persistent code.

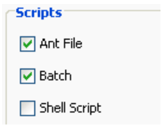

Scripts
[✓] Ant File
[✓] Batch
[ ] Shell Script

Figure 18.83 - Generate scripts options

By running the CompileAll batch file, all Java classes will be compiled at once. After compiled all Java classes, you are able to test the sample file by using the corresponding batch file, such as RunCreateShoppingCartDataSchema.bat, RunCreateShoppingCartData.bat and RunRetrieveShoppingCartData.bat etc.

For Linux user, please execute the CompileAll.sh file to compile all Java classes and test the sample file by the corresponding shell scripts, such as RunCreateShoppingCartDataSchema.sh, RunCreateShoppingCart.sh and RunRetrieveShoppingCartData.sh etc.

For Linux user, please execute the CompileAll.sh file to compile all Java classes and test the sample file by the corresponding shell scripts, such as RunCreateShoppingCartDataSchema.sh, RunCreateShoppingCart.sh and RunRetrieveShoppingCartData.sh etc.

Using Entity Bean

Entity Bean is generated based on the Enterprise JavaBean model defined in the EJB diagram. You are allowed to use the Entity Bean getting from the application server to access the database. In this section, it introduces how to use the Entity Bean to access the database while the Entity Bean is deployed on the JBoss Application Server as an example; please note that you can deploy the Entity Bean to any other supported Application Servers. For more information on deploying beans on application server, refer to Deploying Enterprise JavaBean on Application Servers in the Implementation chapter.

Creating an Entity Bean

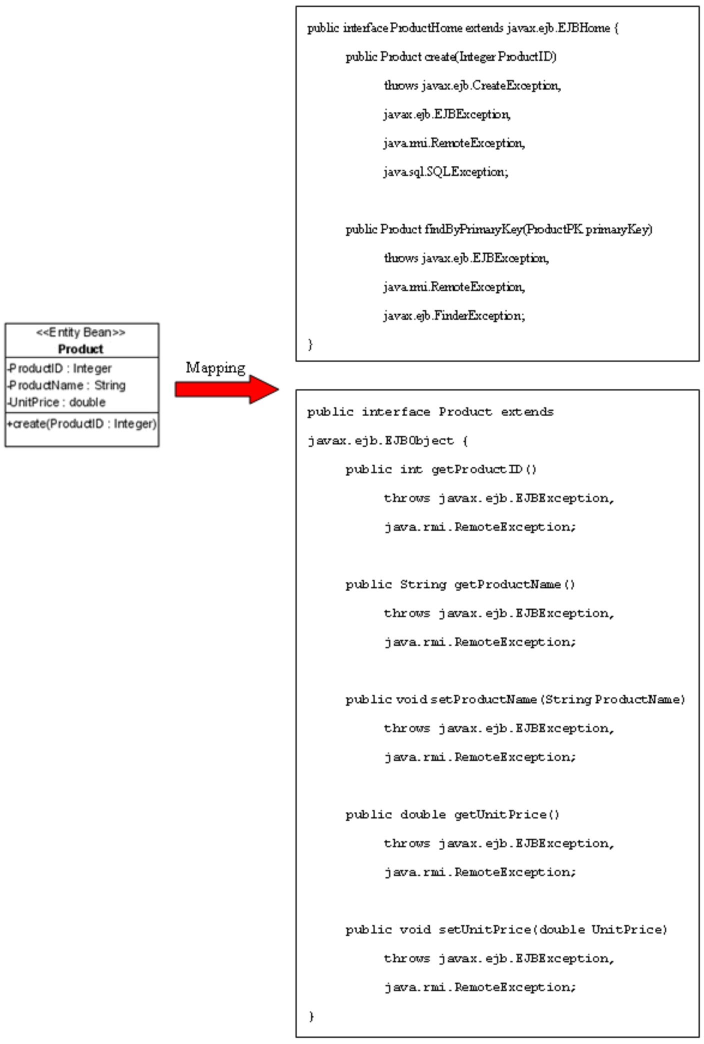
The entity bean is one of Enterprise JavaBeans which represents the persistent data maintained in a database, thus an Entity Bean instance represents a record of the table. The Container Managed Persistent (CMP) Entity Bean is used in the following examples.

In order to insert a new row to the database table, implement the client program with the following steps:

1. Create an Entity Bean model in EJB Diagram, refer to Creating a new Entity Bean to the EJB Diagram in the Object Model chapter for more information.
2. Specify the EJB Class Code Detail for the Entity Bean, refer to Specifying Entity Bean Code Detail in the Implementation chapter for more information.
3. Generate Entity Bean by using Update Code, refer to Using Update Code in the Implementation chapter for more information.
4. Configure Application Server, refer to Configuring Application Servers in the Implementation chapter for more information.
5. Deploy Entity Bean on application server, refer to Deploying Beans on the Application Server in the Implementation chapter for more information.

6. Develop a client program, refer to Developing a Client Program in the Implementation chapter for more information.

Example:



```
public interface ProductHome extends javax.ejb.EJBHome {
    public Product create(Integer ProductID)
        throws javax.ejb.CreateException,
        javax.ejb.EJBException,
        java.rmi.RemoteException,
        java.sql.SQLException;

    public Product findByPrimaryKey(ProductPK primaryKey)
        throws javax.ejb.EJBException,
        java.rmi.RemoteException,
        javax.ejb.FinderException;
}
```

```
public interface Product extends
javax.ejb.EJBObject {
    public int getProductID()
        throws javax.ejb.EJBException,
        java.rmi.RemoteException;

    public String getProductName()
        throws javax.ejb.EJBException,
        java.rmi.RemoteException;

    public void setProductName(String ProductName)
        throws javax.ejb.EJBException,
        java.rmi.RemoteException;

    public double getUnitPrice()
        throws javax.ejb.EJBException,
        java.rmi.RemoteException;

    public void setUnitPrice(double UnitPrice)
        throws javax.ejb.EJBException,
        java.rmi.RemoteException;
}
```

Figure 18.84 - Mapping to Entity Beans

From the above example, ProductHome and Product classes are used to insert the instance as a row of the database table.

To insert a new Product record to the table, Product of the database, the following line of code in client program should be implemented.

```
public class ProductClient{
      public static void main(String[] args){
            Hashtable props = new Hashtable();
            props.put("java.naming.factory.initial",
            "org.jnp.interfaces.NamingContextFactory");
            props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
            props.put("java.naming.provider.url", "jnp://localhost:1099");
            try {
                  Context ctx = new InitialContext(props);
                  ProductHome productHome = (ProductHome)
                  PortableRemoteObject.narrow(ctx.lookup("ProductBean"), ProductHome.class);
                  Product product= productHome.create(new Integer(1));
                  product.setProductName("Introduction on Java");
                  product.setUnitPrice(300);
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }
}
```

After executing the above client program, a row of record is inserted to the database table.

Loading an Entity Bean

As the database table is represented by an Entity Bean, a record of the table can thus be represented by an Entity Bean instance. A record retrieved from the table will be stored as an object.

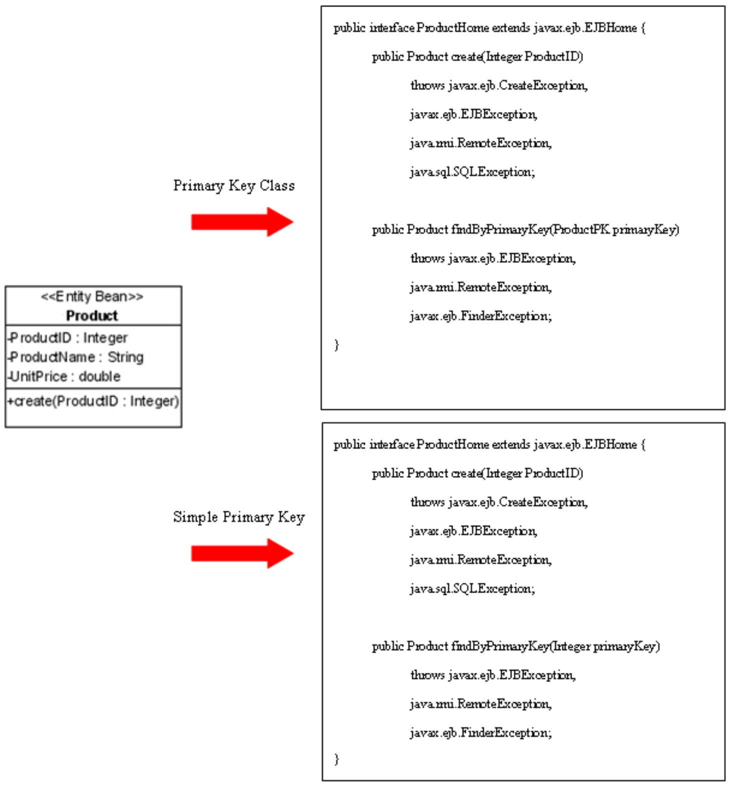
In order to retrieve a record from the database table, you have to specify the condition for finding the record. To retrieve a record, implement the program with the following steps:

1. Specify the primary key value for searching the record by using the findByPrimaryKey() method.

> If you have selected **Simple primary key** option in the **EJB Class Code Detail**, the selected **CMP Field** will be used as the parameter for loading the record from database. On the contrary, if the **Simple primary key** option is not selected, the generated **Primary Key Class** will be used as the parameter for loading the record from database.

2. Load the retrieved record to an object.

Example: The ProductID is the primary key of the Product Entity Bean.



```
public interface ProductHome extends javax.ejb.EJBHome {
        public Product create(Integer ProductID)
                throws javax.ejb.CreateException,
                javax.ejb.EJBException,
                java.rmi.RemoteException,
                java.sql.SQLException;

        public Product findByPrimaryKey(ProductPK primaryKey)
                throws javax.ejb.EJBException,
                java.rmi.RemoteException,
                javax.ejb.FinderException;
}
```

```
public interface ProductHome extends javax.ejb.EJBHome {
        public Product create(Integer ProductID)
                throws javax.ejb.CreateException,
                javax.ejb.EJBException,
                java.rmi.RemoteException,
                java.sql.SQLException;

        public Product findByPrimaryKey(Integer primaryKey)
                throws javax.ejb.EJBException,
                java.rmi.RemoteException,
                javax.ejb.FinderException;
}
```

Figure 18.85 - Mapping Primary Key Class

From the above example, the ProductHome interface contains the **findByPrimaryKey** method to load the record from the database either by a primary key class or a simple primary key.

To retrieve a record from the Product table, the following client program code should be implemented.

Client Program 1: Using Primary Key Class to load the record.

```
public class ProductClient {
      public static void main(String[] args) {
            Hashtable props = new Hashtable();
            props.put("java.naming.factory.initial",
            "org.jnp.interfaces.NamingContextFactory");
            props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
            props.put("java.naming.provider.url", "jnp://localhost:1099");
            try {
                  Context ctx = new InitialContext(props);
                  ProductHome productHome = (ProductHome)
                  PortableRemoteObject.narrow(ctx.lookup("ProductBean"), ProductHome.class);
```

```
                        Product product = productHome.findByPrimaryKey(new ProductPK(1));
                        System.out.println("product name = " + product.getProductName());
                        System.out.println("product unit price = " + product.getUnitPrice());
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Client Program 2: Using Simple Primary Key to load the record.

```
public class ProductClient {
        public static void main(String[] args) {
                Hashtable props = new Hashtable();
                props.put("java.naming.factory.initial",
                "org.jnp.interfaces.NamingContextFactory");
                props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
                props.put("java.naming.provider.url", "jnp://localhost:1099");
                try {
                        Context ctx = new InitialContext(props);
                        ProductHome productHome = (ProductHome)
                        PortableRemoteObject.narrow(ctx.lookup("ProductBean"), ProductHome.class);
                        Product product = productHome.findByPrimaryKey(new Integer(1));
                        System.out.println("Product name = " + product.getProductName());
                        System.out.println("Product unit price = "+ product.getUnitPrice());
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Result:

```
Product name = Introduction on Java
Product unit price = 300.0
```

Using Finder Method

A Finder Method is created for retrieving records from the database by the specified rules. Each finder method is associated with a query element in the deployment descriptor while the finder query must be written using a database-independent language specified in the EJB 2.0 specification, namely EJB QL. You are allowed to create the finder method, refer to Creating a Finder Method in the Object Model chapter for more information.

Example:

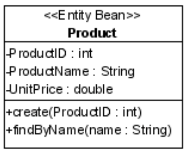


Figure 18.86 - Entity Bean Class

In the above example, a Product Entity Bean model is defined with a finder method named **findByName** which retrieves record from the database according to the specified product name.

The finder query is generated in the deployment descriptor, **ejb-jar.xml** can be found inside the **META-INF** folder of your project by using the **Update Code**.

```
...
<query>
<query-method>
<method-name>findByName</method-name>
<method-params>
<method-param>java.lang.String</method-param>
</method-params>
</query-method>
<ejb-ql>SELECT OBJECT(p) FROM Product p WHERE p.ProductName=?1</ejb-ql>
</query>
...
```

The **findByName** Method is generated in ProductHome class.

```
public interface ProductHome extends javax.ejb.EJBHome {
	public Product create(int ProductID)
		throws javax.ejb.CreateException,
		javax.ejb.EJBException,
		java.rmi.RemoteException,
		java.sql.SQLException;

	public Product findByPrimaryKey(ProductPK primaryKey)
		throws javax.ejb.EJBException,
		java.rmi.RemoteException,
		javax.ejb.FinderException;

	public Product findByName(String name)
		throws javax.ejb.EJBException,
		java.rmi.RemoteException,
		javax.ejb.FinderException;
}
```

Create the Client Program to used **findByName** method.

```
public static void main(String[] args) {
	Hashtable props = new Hashtable();
	props.put("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
	props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
	props.put("java.naming.provider.url", "jnp://localhost:1099");
	try {
		Context ctx = new InitialContext(props);
		ProductHome productHome =
		(ProductHome)PortableRemoteObject.narrow(ctx.lookup("ProductBean"),
		ProductHome.class);
		Product product = productHome.findByName("Introduction on Java");
		System.out.println("Product unit price = " + product.getUnitPrice());
	} catch (Exception e) {
		e.printStackTrace();
	}
}
```

Result:

```
Product unit price = 300.0
```

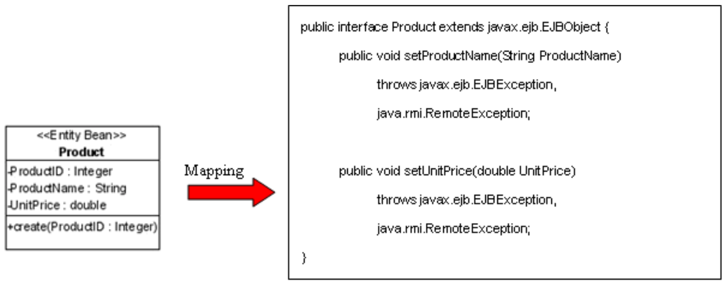
Updating an Entity Bean

As a record can be retrieved from the table and loaded to an instance of the Entity Bean, the record is allowed to update by simply using the setter method of the property.

In order to update a record, you have to retrieve the row being updated, update the value by setting the property to the database. To update the record, implement the program with the following steps:

1. Retrieve a record from database table and load as an Entity Bean instance.
2. Set the updated value to the property of the Entity Bean instance.

Example:



```
public interface Product extends javax.ejb.EJBObject {
	public void setProductName(String ProductName)
		throws javax.ejb.EJBException,
		java.rmi.RemoteException;

	public void setUnitPrice(double UnitPrice)
		throws javax.ejb.EJBException,
		java.rmi.RemoteException;
}
```

Figure 18.86 - Mapping to EJB Object

From the above example, an Entity Bean model, Product maps to a Product interface generated with the methods for setting the properties and updating the row.

To update a Product record, the following line of code should be implemented.

```
ProductHome productHome = (ProductHome) PortableRemoteObject.narrow(ctx.lookup("ProductBean"),
ProductHome.class);
Product product = productHome.findByPrimaryKey(new ProductPK(1));
product.setProductName("OO Technology");
product.setUnitPrice("350");
```

After executing the above lines of code, the product name is updated to "OO Technology" with the unit price of "350" in the database.

Deleting an Entity Bean Instance

As a record can be retrieved from the table and loaded to an entity bean instance, the record can be deleted simply by using the remove method of the entity bean.

In order to delete a record, implement the program with the following steps:

1. Retrieve a record from database table and load as an object.
2. Remove the retrieved record.

Example:

To delete a Product record, the following lines of code should be implemented.

```
ProductHome productHome = (ProductHome) PortableRemoteObject.narrow(ctx.lookup("ProductBean"),
ProductHome.class);
Product product = productHome.findByPrimaryKey(new ProductPK(1));
product.remove();
```

After executing the above code, the specified Product record is removed from the database.

Manipulating Association

When mapping a navigable association to Entity Bean, the role name of the association will become an attribute of the Entity Bean with the data type of its supplier Entity Bean. A pair of getter and setter methods for the role will be generated to the Entity Bean class so as to manipulate its role associated with its supplier Entity Bean.

However, we cannot directly access the getter and setter methods for the role in the client program; Session Bean is used to manipulate the association between the Entity Beans. A Session Bean contains operations which can be called by the client to access database and perform transaction such as calculation.

One-to-One Association

In order to manipulate the directional association, implement the program with the following steps:

1. Set the properties of role name of the object.

Table shows the method summary of the Entity Bean to be used for setting the directional association.

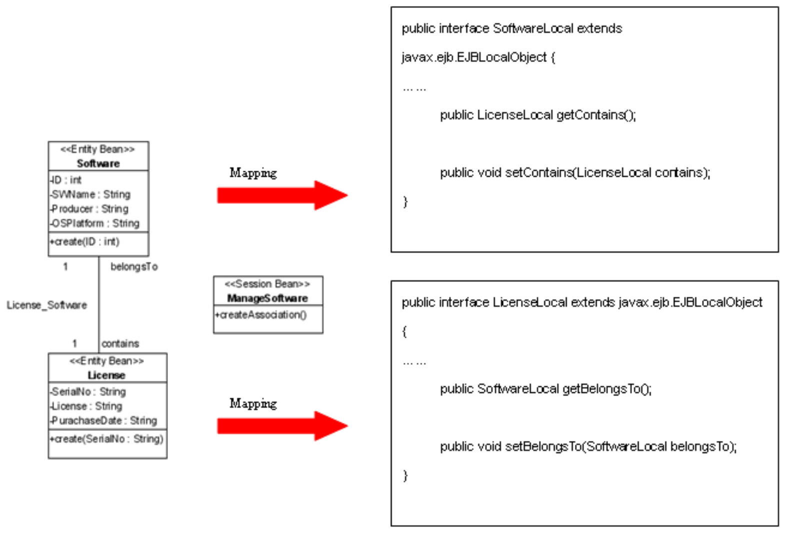
Return Type	Method Name	Description
void	setRoleName(DataType value)	Set the value to the role of an instance.

Table 18.41

Remark:

1. **RoleName** should be replaced by the role of the navigable association.
2. **Data Type** should be replaced by the data type of the supplier local class in the navigable association.

Example:



```
public interface SoftwareLocal extends
javax.ejb.EJBLocalObject {
......
        public LicenseLocal getContains();

        public void setContains(LicenseLocal contains);
}
```

```
public interface LicenseLocal extends javax.ejb.EJBLocalObject
{
......
        public SoftwareLocal getBelongsTo();

        public void setBelongsTo(SoftwareLocal belongsTo);
}
```

Figure 18.87 - Mapping One-to-one relationship for EJB

From the above example, a Software Entity Bean maps to a SoftwareLocal interface with an attribute of role, "contains" in the association typed as its associated interface, LicenseLocal. Meanwhile, the License class maps to a LicenseLocal interface with an attribute of role, "belongsTo" typed as SoftwareLocal. It specifies the association that Software contains a particular License and a License belongs to particular software.

To manipulate the association, ManageSoftware Session Bean should be used and implemented with the following lines of code in the createAssociation() method.

<<Session Bean>>
ManageSoftware
+createAssociation()

Figure 18.88 - Session Bean

```
public void createAssociation() throws javax.ejb.EJBException {
      try {
            Context ctx = new InitialContext();
            SoftwareLocalHome softwareLocalHome = (SoftwareLocalHome)
            PortableRemoteObject.narrow(ctx.lookup("SoftwareLocal"),
            SoftwareLocalHome.class);
            SoftwareLocal softwareLocal = softwareLocalHome.create(1);
            LicenseLocalHome licenseLocalHome = (LicenseLocalHome)
            PortableRemoteObject.narrow(ctx.lookup("LicenseLocal "), LicenseLocalHome.class);
            LicenseLocal licenseLocal = licenseLocalHome.create("key");
            licenseLocal.setBelongsTo(softwareLocal);
      } catch (Exception e) {
            e.printStackTrace();
      }
}
```

In the above sample code, the createAssociation() method implements in a similar way with the previous client program example, it creates the SoftwareLocal and LicenseLocal interfaces to manipulate the association. After deploying the application to the application server, a client program is developed to get the ManageSoftware Session Bean and call the createAssociation() method to establish the association between Software and License.

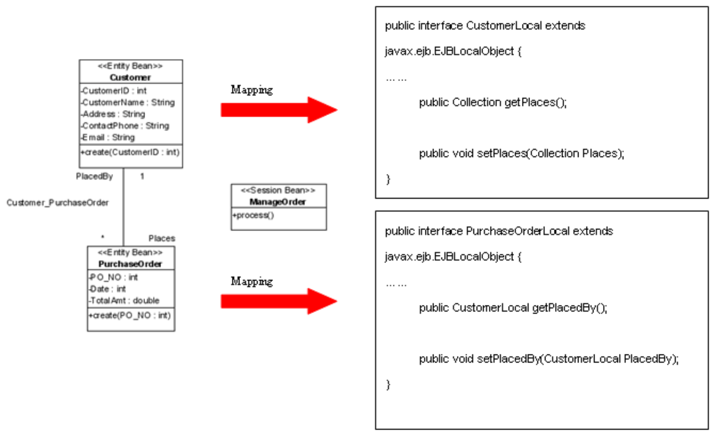
It is a bi-directional association. The association link can be created by setting either the role of Software, or the role of License, i.e. setting one of the roles builds up a bi-directional association automatically. It is not required to set one role after the other. Hence, both license.setBelongsTo(software) and software.contains(license) result in building up the bi-directional association.

One-to-Many Association

In a bi-directional one-to-many association, a class has multiplicity of one while the other has many. If the class has multiplicity of many, a Collection class will be generated for manipulating the objects. When transforming the association to Entity Bean, the role name will map to a Collection class.

1. Set the properties of role name of the object.
2. Create a Collection class to add instance and add the Collection to associated Entity Bean.

Example:



```
public interface CustomerLocal extends
javax.ejb.EJBLocalObject {
… …
        public Collection getPlaces();

        public void setPlaces(Collection Places);
}
```

```
public interface PurchaseOrderLocal extends
javax.ejb.EJBLocalObject {
… …
        public CustomerLocal getPlacedBy();

        public void setPlacedBy(CustomerLocal PlacedBy);
}
```

Figure 18.88 - Mapping Session Bean

From the above example, an Entity Bean object model of PurchaseOrder maps to a PurchaseOrderLocal interface with an attribute of role, "PlacedBy" in the association typed as an instance of CustomerLocal interface, specifying that the particular purchase order is placed by a particular customer. Moreover, the Entity Bean model of Customer maps to a CustomerLocal class with an attribute of role, "Places" in a Collection class, the Collection class which manipulates instances of PurchaseOrder.

To add a PurchaseOrder object to the Collection class, the following lines of code should be implemented in process() method in the ManageOrderBean class.

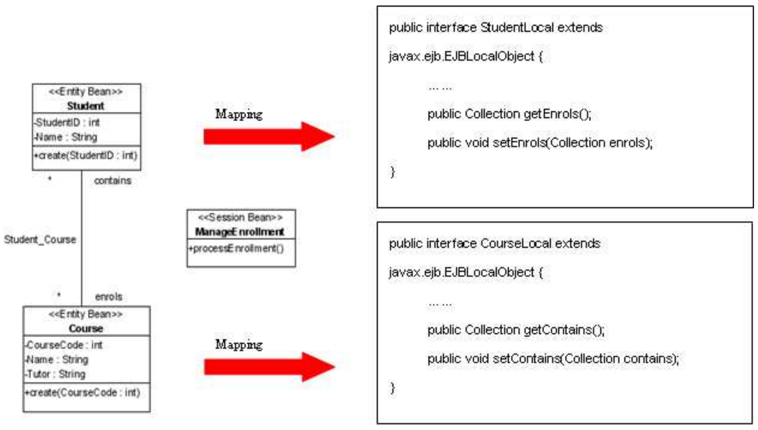
```
public class ManageOrderBean implements javax.ejb.SessionBean {
...
public void process() throws javax.ejb.EJBException {
       try {
              Context ctx = new InitialContext();
              CustomerLocalHome customerLocalHome = (CustomerLocalHome)
              PortableRemoteObject.narrow(ctx.lookup("CustomerLocal"),
              CustomerLocalHome.class);
              CustomerLocal customerLocal = customerLocalHome.create(1);

              PurchaseOrderLocalHome purchaseOrderLocalHome = (PurchaseOrderLocalHome)
              PortableRemoteObject.narrow(ctx.lookup("PurchaseOrderLocal"),
              PurchaseOrderLocalHome.class);
              PurchaseOrderLocal purchaseOrderLocal = purchaseOrderLocalHome.create(1);

              ArrayList purchaseOrders = new ArrayList();
              purchaseOrders.add(purchaseOrderLocal);
              customerLocal.setPlaces(purchaseOrders);

              purchaseOrderLocal.setPlacedBy(customerLocal);
       } catch (Exception e) {
              e.printStackTrace();
       }
}
...
}
```

After executing these lines of code, an ArrayList is created for adding the instance of PurchaseOrder for the Customer. The PurchaseOrders can be retrieved from Customer by using the getPlaces() method to get the collection object of PurchaseOrders.

Many-to-Many Association

When transforming a many-to-many association between two Entity Bean classes, the Collection class is used to manipulate the association class objects. The Entity Bean has getter and setter methods for role name with the parameter of Collection class data type.

Example:



```
public interface StudentLocal extends
javax.ejb.EJBLocalObject {
        ... ...
        public Collection getEnrols();
        public void setEnrols(Collection enrols);
}
```

```
public interface CourseLocal extends
javax.ejb.EJBLocalObject {
        ... ...
        public Collection getContains();
        public void setContains(Collection contains);
}
```

Figure 18.89 - Mapping Many-to-many association for EJB

From the above example, a Student Entity Bean maps to a StudentLocal interface with an attribute of role, "Enrols" in the association typed as Collection class, Collection Class which manipulate instance of the Course. Moreover, the entity bean model of Course maps o a CourseLocal interface with an attribute of role, "Contains" in the association typed as a Collection class, Collection Class which manipulate instance of the Students.

Implement the following code to the processEnrollment() method of the ManageEnrollmentBean:

```
public class ManageEnrollmentBean implements javax.ejb.SessionBean {
...
public void processEnrollment() throws javax.ejb.EJBException {
       try {
              Context ctx = new InitialContext();
              ArrayList students = new ArrayList();
              ArrayList courses = new ArrayList();

              StudentLocalHome studentLocalHome =
              (StudentLocalHome)PortableRemoteObject.narrow(ctx.lookup("StudentLocal"),
              StudentLocalHome.class);
              StudentLocal studentLocal1 = studentLocalHome.create(1);
              studentLocal1.setName("Eric");
              StudentLocal studentLocal2 = studentLocalHome.create(2);
              studentLocal2.setName("Mary");
              students.add(studentLocal1);
              students.add(studentLocal2);

              CourseLocalHome courseLocalHome =
              (CourseLocalHome)PortableRemoteObject.narrow(ctx.lookup("CourseLocal"),
              CourseLocalHome.class);
              CourseLocal courseLocal = courseLocalHome.create(1);
              courseLocal.setName("Math");
              courseLocal.setTutor("Sam");
              courses.add(courseLocal);
              studentLocal.setEnrols(courses);
              courseLocal.setContains(students);
       } catch (Exception e) {
              e.printStackTrace();
       }
}
...
}
```

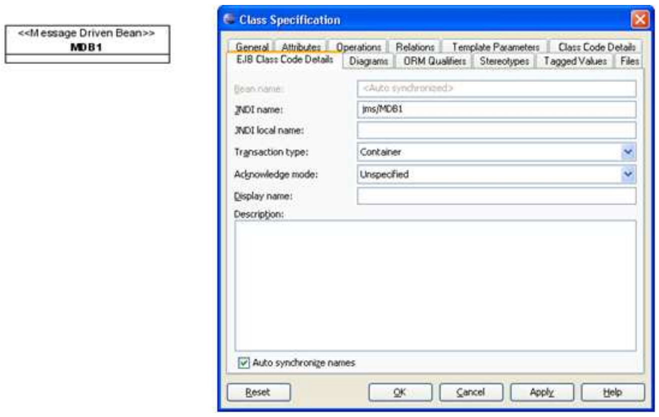
Both the Student table and Course table are inserted with a new record. You can use getter method to get Courses from the Student or get the Students from the Course.

Creating a Message Driven Bean

The message-driven bean allows you to process the JMS message asynchronously. The message may be sent by any J2EE component such as client application, another Enterprise JavaBean, web component or JMS application. The main difference between the message-driven beans, entity beans and session beans is that the client can access the bean directly.

The following example illustrates how to create the client program.

Example:



Bean name: <Auto synchronized>
JNDI name: jms/MDB1
JNDI local name:
Transaction type: Container
Acknowledge mode: Unspecified
Display name:
Description:
[✓] Auto synchronize names

Figure 18.90 - Message Driven Bean

In order to create and deploy the Message Drive Bean Class, implement the client program with the following steps:

1. Create Message Drive Bean Class MDB1 in EJB Diagram, refer to Creating a new Message Driven Bean to the EJB Diagram in the Object Model chapter for more information.
2. Specify the EJB Class Code Detail for the Message Driven Bean, refer to Specifying Message-Driven Bean Code Detail in the Implementation chapter for more information.
3. Generate Message-Driven Bean by using Update Code, refer to Using Update Code in the Implementation chapter for more information
4. Configure Application Server, refer to Configuring Application Servers in the Implementation chapter for more information.
5. Deploy Message-Driven Bean on application server, refer to Deploying Beans on the Application Server in the Implementation chapter for more information.
6. Develop a client program, refer to Developing a Client Program in the Implementation chapter for more information.

The client program for using the Message-Driven Bean should be implemented as follows:

```java
public class Main {
// JNDI name of the Topic
static String TOPIC_NAME = "jms/MDB1";

// JNDI name of the default connection factory
static String CONNECTION_FACTORY_NAME = "JTCF";

public static void main(String[] arg) {
	InitialContext ctx = null;
	TopicConnectionFactory tcf = null;
	Topic topic = null;
	try {
		ctx = new InitialContext();
		// lookup the TopicConnectionFactory through its JNDI name
		tcf = (TopicConnectionFactory) ctx.lookup(CONNECTION_FACTORY_NAME);
		// lookup the Topic through its JNDI name
		topic = (Topic) ctx.lookup(TOPIC_NAME);
	} catch (NamingException e) {
		e.printStackTrace();
		System.exit(1);
	}

	TopicConnection tc = null;
	TopicSession session = null;
	TopicPublisher tp = null;
	try {
```

```
            tc = tcf.createTopicConnection();
            session = tc.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
            tp = session.createPublisher(topic);
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        // publish 5 messages
        try {
            for (int i = 0; i < 5; i++) {
                TextMessage message = session.createTextMessage();
                message.setText("TextMessage " + i);
                tp.publish(message);
            }
            session.close();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
}
}
```

The client program shows that via the JNDI name to access the Message-Driven Bean and get the Topic. After getting the Topic, message can be published in the channel.

Creating a Session Bean

The Session Bean is one of the enterprise beans which are created by a client and it will only exist for the duration of a single client-server session. A Session Bean contains operations which can be called by the client to access the database or perform transaction such as calculation. Session Bean can be stateless or stateful. For a stateful Session Bean, the EJB container will manage the state automatically while stateless Session Bean does not have any state between calls to its methods.

The Session Bean is used to manipulate the associations between Entity Beans, refer to One to One Association , One to Many Association and Many to Many Association of the Using Entity Beans section for more information.

In order to create and deploy the Session Bean Class, implement the client program with the following steps:

1. Create Session Bean Class in EJB Diagram, refer to Creating a new Session Bean to the EJB Diagram.
2. Specify the EJB Class Code Detail for the Message Driven Bean, refer to Specifying Session Bean Code Detail in the Implementation chapter for more information.
3. Generate Session Bean by using Update Code, refer to Using Update Code in the Implementation chapter for more information
4. Configure Application Server, refer to Configuring Application Servers in the Implementation chapter for more information.
5. Deploy the Session Bean on application server, refer to Deploying Beans on the Application Server in the Implementation chapter.
6. Develop a client program, refer to Developing a Client Program in the Implementation chapter for more information.

Example:

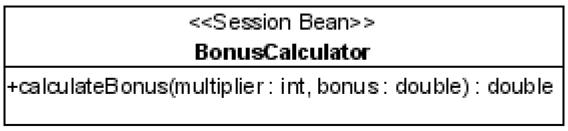


Figure 18.91 - Session Bean

In the above example, a Session Bean is used to calculate the bonus with a calculateBonus() method.

The code implement at the BonusCalculatorBean class.

```
public class BonusCalculatorBean implements javax.ejb.SessionBean {
        private SessionContext ctx;

        public double calculateBonus(int multiplier, double bonus)
        throws javax.ejb.EJBException {
            double calculatedBonus = (multiplier * bonus);
            return calculatedBonus;
        }
...
}
```

To develop the client program to get the Session Bean and help us to calculate the bonus by using the following lines of code:

```
public class ClientProgram {
public static void main(String[] args) {
        Hashtable props = new Hashtable();
        props.put("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
        props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
        props.put("java.naming.provider.url", "jnp://localhost:1099");
        try {
                Context ctx = new InitialContext(props);
                BonusCalculatorHome bonusCalculatorHome = (BonusCalculatorHome)
                PortableRemoteObject.narrow(ctx.lookup("BonusCalculatorBean"),
                BonusCalculatorHome.class);
                BonusCalculator bonusCalculator = bonusCalculatorHome.create();
                System.out.println("The calculate result = " + bonusCalculator.calculateBonus(2,
                100));
        } catch (Exception e) {
                e.printStackTrace();
        }
}
}
```

Result:

The calculate result = 200.0

Transactions

In the J2EE architecture, an EJB application server supports distributed transaction processing. As distributed transaction is supported, a transaction manager is used to control the global transactions. The transaction manager provides the services by using the Java Transaction API (JTA). If a transaction is going to be performed, the client can get access to JTA via the container.

In order to determine how the transactions can be managed by the container, transaction attributes are used. The transaction attribute specifies how the transaction begins and ends within the container (container-managed transactions) or the EJB itself (bean-managed transactions).

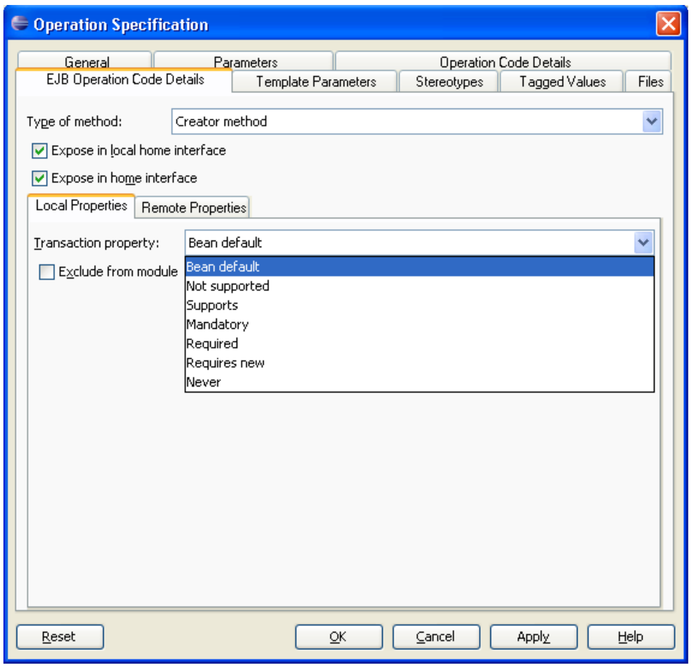
Using Container-Managed Transaction

A transaction which can be managed by the EJB container is called Container-Managed Transaction. For the container-managed transaction, transaction attributes defined in the deployment descriptor specifies how the EJB container manages the transactions at runtime such that the bean does not control the transaction directly. Entity Bean uses container-managed transaction demarcation.

Configuring Transaction Attribute

You can configure the Transaction Attribute of an Entity Bean or a Session Bean by using the **Operation Specification** dialog box.

1. Right-click on an operation of either an Entity Bean or a Session Bean, click **Open Specification...**.
2. Select **EJB Operation Code Details** Tab, select the Transaction Property on the Local Properties and Remote Properties tabs.


Operation Specification

General | Parameters | Operation Code Details | EJB Operation Code Details | Template Parameters | Stereotypes | Tagged Values | Files

Type of method: Creator method

☑ Expose in local home interface

☑ Expose in home interface

Local Properties | Remote Properties

Transaction property: Bean default

☐ Exclude from module

Bean default
Not supported
Supports
Mandatory
Required
Requires new
Never

Reset | OK | Cancel | Apply | Help

Figure 18.92 - The Transaction property

Using Bean-Managed Transaction

A transaction which is managed by the Enterprise JavaBean instead of the EJB container is called Bean-Managed Transaction. Session Bean use bean-managed transaction demarcation which cannot be used by Entity Bean such that Session bean manage its own transactions programmatically by using JDBC transaction system or the Java Transaction API (JTA).

Example of using bean-managed transaction in Session Bean:

1. Create and add an operation for a Session Bean.

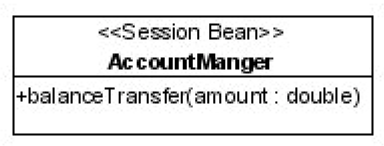
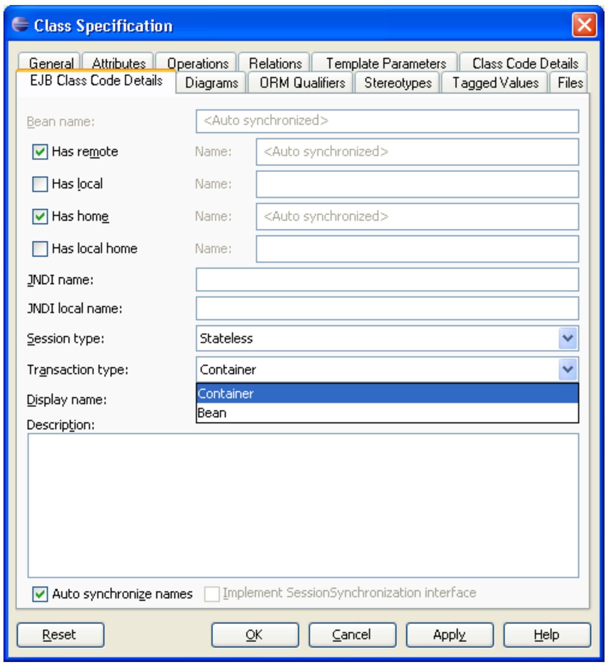


Figure 18.93 - Bean-Managed Transaction

2. Right-click on the Session Bean, click **Open Specification...**.

3. Select **EJB Class Code Details** Tab, select **Bean** from the drop-down menu of the **Transaction Type**.



Bean name: <Auto synchronized>
☑ Has remote Name: <Auto synchronized>
☐ Has local Name:
☑ Has home Name: <Auto synchronized>
☐ Has local home Name:
JNDI name:
JNDI local name:
Session type: Stateless
Transaction type: Container
Container
Bean
Display name:
Description:
☑ Auto synchronize names ☐ Implement SessionSynchronization interface

Figure 18.94 - Set the transaction type

4. Implement the operation with Java Transaction Service (JTS) transaction by using the following lines of code in balanceTransfer() method of a JTA transaction:

```
public class AccountMangerBean implements javax.ejb.SessionBean {
...
public void balanceTransfer(double amount) throws javax.ejb.EJBException {
        UserTransaction userTransaction = ctx.getUserTransaction();
        try {
                userTransaction.begin();
                substractCurrencyAccount1(amount);
                addCurrencyAccount2(amount);
                userTransaction.commit();
        } catch (Exception ex) {
                try {
                        userTransaction.rollback();
                } catch (SystemException syex) {
                        throw new EJBException ("Rollback failed: " + syex.getMessage());
                }
                throw new EJBException ("Transaction failed: " + ex.getMessage());
        }
}
...
}
```

The begin() and commit() invocations delimit the updates to the database. If the update of a transaction is failed, the client program invokes the rollback() method and throws an EJBException.

19 Manipulating Persistent Data with .NET

Chapter 19 - Manipulating Persistent Data with .NET

Java and .NET persistent code can be generated to manipulate persistent data with the database. This chapter shows the use of the generated .NET persistent code by inserting, retrieving, updating and deleting persistent data and demonstrates how to run the generated sample code with C#.

In this chapter:

- Introduction
- Using ORM-Persistable .NET Class to manipulate persistent data
- Using Criteria Class to retrieve persistent data
- Using ORM Implementation
- Applying generated .NET persistence class to different .NET language

Introduction

With the Smart Development Environment Enterprise Edition (SDE EE), you can generate .NET persistence code to manipulate the persistent data of the relational database easily.

In the working environment, you are allowed to configure the database connection for your development project. As the database was configured before the generation of persistent code, not only the ORM persistable .NET classes are generated, but also a set of ORM files which configures the environment for connecting the generated persistence classes and the database. And hence, you are allowed to access the database directly by using the .NET persistence class without using any code for setting up the database connection in your .NET project.

The generated .NET persistence code is applicable to all .NET language such as C#, C++ and VB for .NET project development. C# is used as the sample source code illustrating how to manipulate persistent data with the generated .NET persistence code and ORM files. Using the same set of .NET persistence class and ORM files to develop a .NET project with other .NET languages instead of C# is briefly described at the end of this chapter.

There are several mappings in the SDE environment:

1. Mapping between data model and relational database.
2. Mapping between data model and object model.
3. Mapping between object model and persistent code.

And hence, there is an indirect mapping between persistent code and relational database. Each persistence class represents a table in the database and an instance of the class represents one record of the table. In the generated persistence class, there is not only a pair of getter and setter methods for manipulating the corresponding attribute, but also a set of methods for manipulating records with the database

Using ORM-Persistable .NET Class

ORM-Persistable .NET class is generated based on the object model defined in the class diagram. The generated .NET persistence code can be identified into two categories - Model API and Persistent API. The Model API refers to the manipulation of attributes of models and associations between models while the Persistent API refers to the persistent code used to manipulate the persistent data with relational database.

C# is used to show the manipulation on persistent data throughout this chapter. The generated .NET persistence code is applicable to all .NET language, such as C#, C++ and VB.

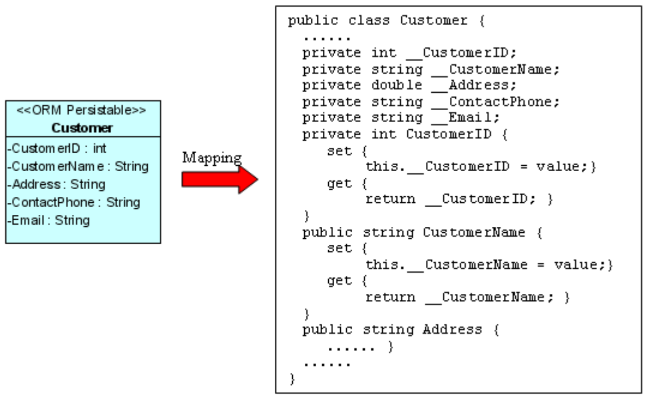
Model API

Model API refers to the generated .NET persistent code which is capable of manipulating the properties of the object model in terms of attributes and associations.

Manipulating Attributes

In order to specify and retrieve the value to the attributes of the ORM-Persistable class, a pair of getter and setter methods for each attribute is generated to the ORM-Persistable .NET class.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
          this.__CustomerID = value;}
     get {
          return __CustomerID; }
  }
  public string CustomerName {
     set {
          this.__CustomerName = value;}
     get {
          return __CustomerName; }
  }
  public string Address {
     ...... }
  ......
}
```

Figure 19.1 - Mapping attributes

From the above example, an ORM-Persistable object model of Customer maps to an ORM-Persistable .NET class generated with the getter and setter methods for each attribute.

To set the values of properties of the Customer object, the following lines of code should be used.

```
customer.CustomerName = "Joe Cool";
customer.Address = "1212, Happy Building";
customer.ContactPhone ="23453256";
customer.Email = "joe@cool.com";
```

To get the values of properties of the Customer object:

```
String name = customer.CustomerName;
String address = customer.Address;
String phone = customer.ContactPhone;
String email = customer.Email();
```

Manipulating Association

When mapping a navigable association to persistence code, the role name of the association will become an attribute of the class. A pair of getter and setter methods for the role will be generated to the persistence class so as to manipulate its role associated with its supplier class. There are two ways in manipulating association, including Smart Association Handling and Standard Association Handling.

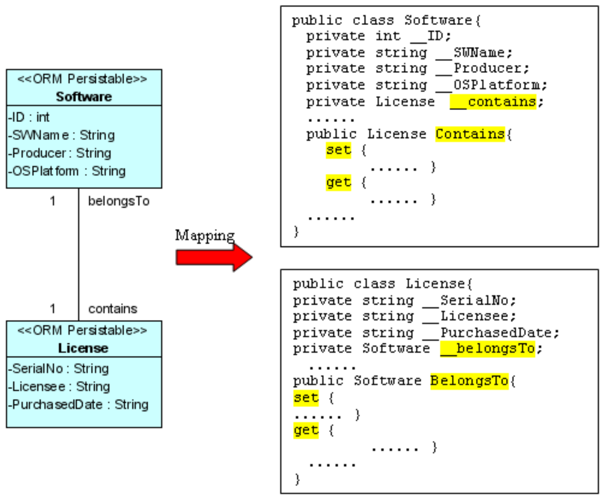
Smart Association Handling

Using smart association handling, the generated persistent code is capable of defining one end of the association which updates the other end of association automatically in a bi-directional association regardless of multiplicity. Examples are given to show how the generated persistent code manipulates the one-to-one, one-to-many and many-to-many associations with smart association handling.

One-to-One Association

In order to manipulate the directional association, implement the program by setting the properties of role name of the object.

Example:



```
public class Software{
  private int __ID;
  private string __SWName;
  private string __Producer;
  private string __OSPlatform;
  private License  __contains;
  ......
  public License Contains{
     set {
          ...... }
     get {
          ...... }
  ......
}
```

```
public class License{
private string __SerialNo;
private string __Licensee;
private string __PurchasedDate;
private Software  __belongsTo;
  ......
public Software BelongsTo{
set {
...... }
get {
           ...... }
  ......
}
```

Figure 19.2 - Mapping One-to-one association

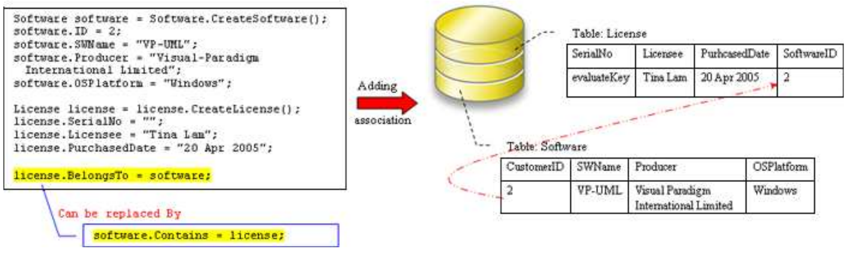
From the above example, an ORM-Persistable object model of Software maps to an ORM-Persistable .NET class with an attribute of role, "__contains" in the association typed as its associated persistence class, License. Meanwhile, the object model of License maps to another .NET class with an attribute of role, "__belongsTo" typed as Software. It specifies the association that Software contains a particular License and a License belongs to particular software.

To manipulate the association, the following lines of code should be implemented.

```
Software software = Software.CreateSoftware();
License license = License.CreateLicense();
```

license.BelongsTo = software;

In the above association, the primary key of the Software table will become a foreign key in License table when transforming to data model. Thus, after executing the lines of code, the primary key of the software will be referenced as a foreign key of the license record.



```
Software software = Software.CreateSoftware();
software.ID = 2;
software.SWName = "VP-UML";
software.Producer = "Visual-Paradigm
  International Limited";
software.OSPlatform = "Windows";

License license = license.CreateLicense();
license.SerialNo = "";
license.Licensee = "Tina Lam";
license.PurchasedDate = "20 Apr 2005";

license.BelongsTo = software;
```

Can be replaced By

```
software.Contains = license;
```

Table: License

SerialNo	Licensee	PurhcasedDate	SoftwareID
evaluateKey	Tina Lam	20 Apr 2005	2

Table: Software

CustomerID	SWName	Producer	OSPlatform
2	VP-UML	Visual Paradigm International Limited	Windows

Figure 19.3 - Code for adding one-to-one association

It is a bi-directional association with smart association handling. The association link can be created by setting either the role of Software, or the role of License, i.e. setting one of the roles builds up a bi-directional association automatically. It is not required to set one role after the other. Hence, both license.BelongsTo = software and software.Contains = license result in building up the bi-directional association.

One-to-Many Association

In a bi-directional one-to-many association, a class has multiplicity of one while the other has many. If the class has multiplicity of many, the corresponding collection class will be automatically generated for manipulating the objects. When transforming the association to persistent code, the role name will map to an attribute with data type of a collection class. For more detailed information on the collection class, refer to the description in Using Collection section.

A bi-directional one-to-many association is shown below.

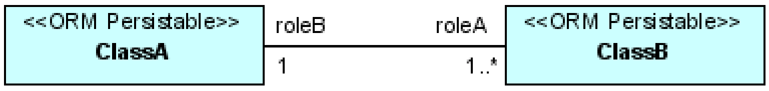


Figure 19.4 - One-to-many association

To manipulate the bi-directional one-to-many association, you can create the association in one of the two ways:

- Set the properties of role with an instance of the associated class; i.e. classB.RoleB = classA
- Add the objects to the collection of the associated class; i.e. classA.RoleA.Add(classB)

where classA is an object of ClassA; classB is an object of ClassB;
RoleA is the collection of ClassB; RoleB is the setter method of property, roleB.

After specifying the association, the value of primary key of the object of ClassA will be referenced as a foreign key of the associated object of ClassB.

Setting the property of role

For information on setting the properties of role, refer to the description in the One-to-One Association section.

Adding objects to the collection

In order to add an object to the collection, implement the program with the following steps:

1. Create a new instance of the class which associates with more than one instance of associated class.
2. Create a new object of the associated class.
3. Add a new object or an existing object of the associated class to the collection belonging to the class.

Table shows the method summary of the collection class to be used for adding a new object to it

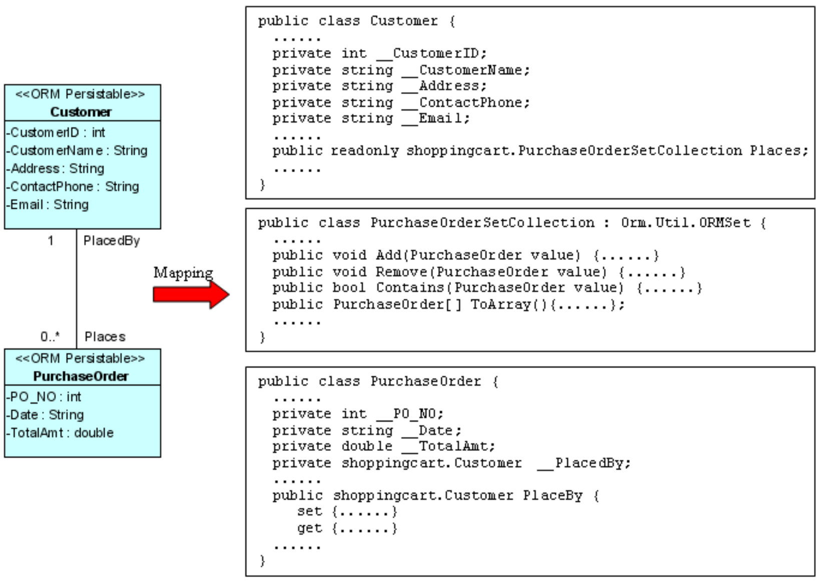
Return Type	Method Name	Description
void	Add(Class value)	Add a new object to the collection of the associated class.

Table 19.1

Remark:

1. **Class** should be replaced by the name of the associated class.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private string __Address;
  private string __ContactPhone;
  private string __Email;
  ......
  public readonly shoppingcart.PurchaseOrderSetCollection Places;
  ......
}
```

```
public class PurchaseOrderSetCollection : Orm.Util.ORMSet {
  ......
  public void Add(PurchaseOrder value) {......}
  public void Remove(PurchaseOrder value) {......}
  public bool Contains(PurchaseOrder value) {......}
  public PurchaseOrder[] ToArray(){......};
  ......
}
```

```
public class PurchaseOrder {
  ......
  private int __PO_NO;
  private string __Date;
  private double __TotalAmt;
  private shoppingcart.Customer  __PlacedBy;
  ......
  public shoppingcart.Customer PlaceBy {
     set {......}
     get {......}
  ......
}
```

Figure 19.5 - Mapping One-to-many association

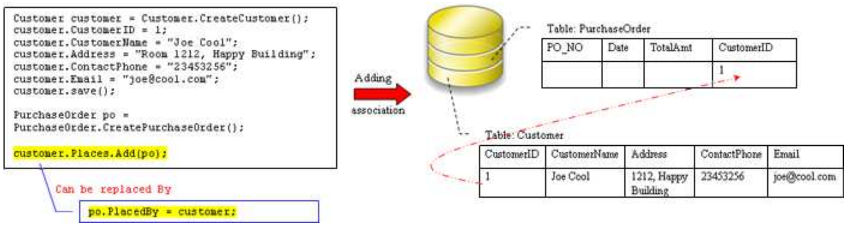
From the above example, an ORM-Persistable object model of PurchaseOrder maps to an ORM-Persistable .NET class with an attribute of role, "__PlacedBy" in the association typed as an instance of Customer, specifying that the particular purchase order is placed by a particular customer. Moreover, the object model of Customer maps to a .NET class with an attribute of role, "__Places" in the association typed as a collection class, PurchaseOrderSetCollection which manipulates instances of PurhcaseOrder.

To add a PurchaseOrder object to the collection of PurchaseOrder, the following lines of code should be implemented.

```
Customer customer = Customer.CreateCustomer();
PurchaseOrder po = PurchaseOrder.CreatePurchaseOrder();
customer.Places.Add(po);
```

After executing these lines of code, an object is added to the collection representing the association. When inserting records to the database tables, the primary key of the customer will be referenced to as the foreign key of the purchase order record.

The alternative way to create the association is using
po.PlacedBy = customer;



```
Customer customer = Customer.CreateCustomer();
customer.CustomerID = 1;
customer.CustomerName = "Joe Cool";
customer.Address = "Room 1212, Happy Building";
customer.ContactPhone = "23453256";
customer.Email = "joe@cool.com";
customer.save();

PurchaseOrder po =
PurchaseOrder.CreatePurchaseOrder();

customer.Places.Add(po);
```

Can be replaced By

```
po.PlacedBy = customer;
```

Table: PurchaseOrder

PO_NO	Date	TotalAmt	CustomerID
			1

Table: Customer

CustomerID	CustomerName	Address	ContactPhone	Email
1	Joe Cool	1212, Happy Building	23453256	joe@cool.com

Figure19.6 - Code for add one-to-many association

Retrieving objects from the collection

In order to retrieve an object from the collection, implement the program with the following steps:

1. Retrieve an instance of the class which associates with more than one instance of associated class.
2. Get the collection from the class, and convert the collection into an array.

Table shows the method summary of the collection class to convert the collection into an array.

Return Type	Method Name	Description
Class[]	ToArray()	Convert the collection into an array which stores the objects of the associated class.

Table 19.2

Remark:

1. **Class** should be replaced by the name of the associated class.

Example:

Refer to the example of the one-to-many association between the ORM-Persistable object models of Customer and PurchaseOrder, implement the following lines of code to retrieve a collection of PurchaseOrder objects from the customer.

```
Customer customer = Customer.LoadByName("Joe Cool");
PurchaseOrder[] orders = customer.Places.ToArray();
```

After executing these lines of code, the purchase order records associated with the customer are retrieved and stored in the array.

Retrieve the customer record by the LoadByName() method which is the method provided by the persistent API. For more information on retrieving persistent object, refer to the Persistent API section.

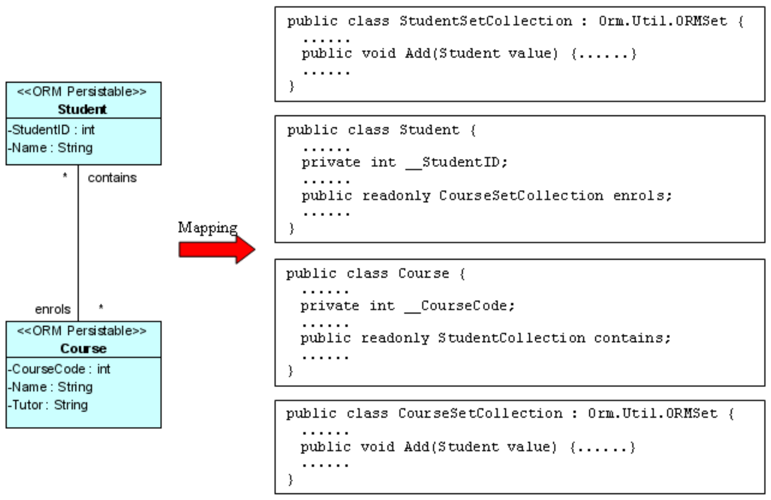
Many-to-Many Association

When transforming a many-to-many association between two ORM-Persistable classes, the corresponding persistent and collection class will be generated simultaneously such that the collection class is able to manipulate its related objects within the collection.

In order to specify the many-to-many association, add the objects to the corresponding collection of the associated class. For information on adding objects to the collection, refer to the description in the One-to-Many Association section.

In addition, a many-to-many association in the object model is transformed to data model by generating a Link Entity to form two one-to-many relationships between two generated entities. The primary keys of the two entities will migrate to the link entity as the primary/foreign keys. When specifying the association by adding objects to the collection, the primary key of the two related instances will be inserted as a pair for a row of the Link Entity automatically.

Example:



```
public class StudentSetCollection : Orm.Util.ORMSet {
  ......
  public void Add(Student value) {......}
  ......
}
```

```
public class Student {
  ......
  private int __StudentID;
  ......
  public readonly CourseSetCollection enrols;
  ......
}
```

```
public class Course {
  ......
  private int __CourseCode;
  ......
  public readonly StudentCollection contains;
  ......
}
```

```
public class CourseSetCollection : Orm.Util.ORMSet {
  ......
  public void Add(Student value) {......}
  ......
}
```

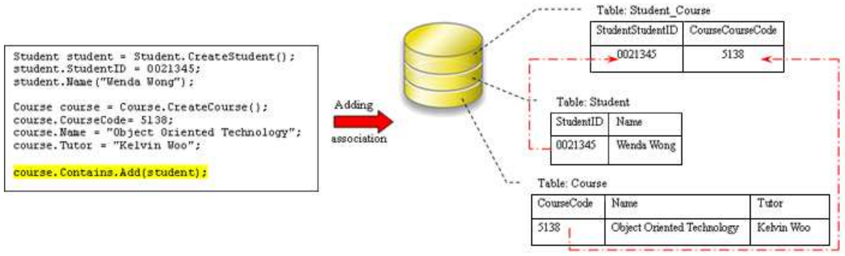
Figure 19.7 - Mapping many-to-many association

There are four classes, including Student, StudentSetCollection, Course and CourseSetCollection generated from the above object model.

By executing the following lines of code:

```
Student student = Student.CreateStudent();
student.StudentID = 0021345;
student.Name = "Wenda Wong";
Course course = Course.CreateCourse();
course.CourseCode = 5138;
course.Name = "Object Oriented Technology";
course.Tutor = "Kelvin Woo";
course.Contains.Add(student);
```

Both the Student table and Course table are inserted with a new record. After specifying the association by course.Contains.Add(student) the corresponding primary keys of student record and course record migrate to the Student_Course Link Entity to form a row of record.



```
Student student = Student.CreateStudent();
student.StudentID = 0021345;
student.Name("Wenda Wong");

Course course = Course.CreateCourse();
course.CourseCode= 5138;
course.Name = "Object Oriented Technology";
course.Tutor = "Kelvin Woo";

course.Contains.Add(student);
```

Table: Student_Course

StudentStudentID	CourseCourseCode
0021345	5138

Table: Student

StudentID	Name
0021345	Wenda Wong

Table: Course

CourseCode	Name	Tutor
5138	Object Oriented Technology	Kelvin Woo

Figure 19.8 - Code for add a many-to-many association

Using Collection

A collection represents a group of objects. Some collections allow duplicate objects and others do not. Some collections are ordered and other unordered. A collection class thus supports the manipulation of the objects within a collection. There are four types of collection supported, including set, bag, list and map.

The type of collection can be specified in advance of generating persistence code. Refer to Specifying Collection Type in the Object Model chapter for more information.

Set

Set is an unordered collection that does not allow duplication of objects. It is the default type for unordered collection.

Table shows the method summary of the collection class typed as Set.

Return Type	Method Name	Description
void	Add(Class value)	Add the specified persistent object to this set if it is not already present.
void	Clear()	Remove all of the persistent objects from this set.
bool	Contains(Class value)	Return true if this set contains the specified persistent object.
Iterator	GetIterator()	Return an iterator over the persistent objects in this set.
bool	IsEmpty()	Return true if this set contains no persistent object.
void	Remove(Class value)	Remove the specified persistent object from this set if it is present.
int	Size()	Return the number of persistent objects in this set.
Class[]	ToArray()	Return an array containing all of the persistent objects in this set.

Table 19.3

Remark:

1. **Class** should be replaced by the persistence class.

Bag

Bag is an unordered collection that may contain duplicate objects.

Table shows the method summary of the collection class typed as Bag.

Return Type	Method Name	Description
void	Add(Class value)	Add the specified persistent object to this bag.
void	Clear()	Remove all of the persistent objects from this bag.
bool	Contains(Class value)	Return true if this bag contains the specified persistent object.
Iterator	GetIterator()	Return an iterator over the persistent objects in this bag.
bool	IsEmpty()	Return true if this bag contains no persistent object.
void	Remove(Class value)	Remove the specified persistent object from this bag.
int	Size()	Return the number of persistent objects in this bag.
Class[]	ToArray()	Return an array containing all of the persistent objects in this bag.

Table 19.4

Remark:

1. **Class** should be replaced by the persistence class.

List

List is an ordered collection that allows duplication of objects. It is the default type for ordered collection.

Table shows the method summary of the collection class typed as List.

Return Type	Method Name	Description
void	Add(Class value)	Append the specified persistent object to the end of this list.
void	Add(int index, Class value)	Insert the specified persistent object at the specified position in this list.
void	Clear()	Remove all of the persistent objects from this list.
bool	Contains(Class value)	Return true if this list contains the specified persistent object.
Object	Get(int index)	Return the persistent object at the specified position in this list.
Iterator	GetIterator()	Return an iterator over the persistent objects in this list in proper sequence.
bool	IsEmpty()	Return true if this list contains no persistent object.
void	Remove(Class value)	Remove the first occurrence in this list of the specified persistent object.

Class	Remove(int index)	Remove the persistent object at the specified position in this list.
Int	Set(int index, Class value)	Replace the persistent object at the specified position in this list with the specified persistent object.
Int	Size()	Return the number of persistent objects in this list.
Class[]	ToArray()	Return an array containing all of the persistent objects in this list in proper sequence.

Table 19.5

Remark:

1. **Class** should be replaced by the persistence class.

Map

Map is an ordered collection which is a set of key-value pairs while duplicate keys are not allowed.

Table shows the method summary of the collection class typed as Map.

Return Type	Method Name	Description
void	Add(Object key, Class value)	Add the specified persistent object with the specified key to this map.
void	Clear()	Remove all mappings from this map.
bool	Contains(Object key)	Return true if this map contains a mapping for the specified key.
Class	Get(Object key)	Return the persistent object to which this map maps the specified key.
Iterator	GetIterator()	Return an iterator over the persistent objects in this map.
Iterator	GetKeyIterator()	Return an iterator over the persistent objects in this map.
bool	IsEmpty()	Return true if this map contains no key-value mappings.
void	Remove(Object key)	Remove the mapping for this key from this map if it is present.
int	Size()	Return the number of key-value mappings in this map.
Class[]	ToArray()	Return an array containing all of the persistent objects in this map.

Table 19.6

Remark:

1. **Class** should be replaced by the persistence class.

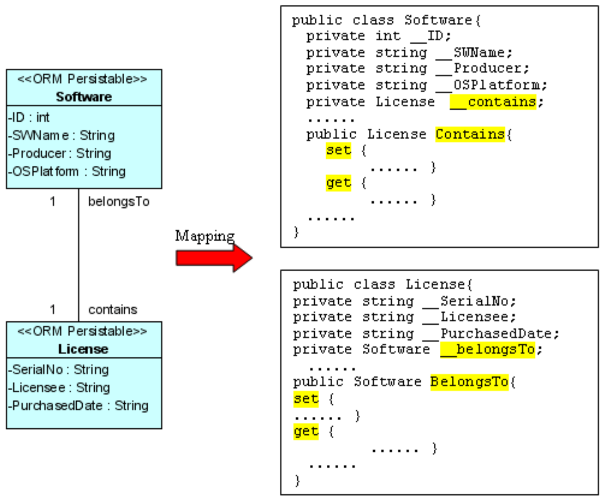
Standard Association Handling

With standard association handling, when updating one end of the association, the generated persistent code will not update the other end of a bi-directional association automatically. Hence, you have to define the two ends of the bi-directional association manually to maintain consistency. Examples are given to show how to manipulate the one-to-one, one-to-many and many-to-many associations with standard association handling.

One-to-One Association

In order to manipulate the directional association, implement the program by setting the properties of role name of the object in the association.

Example:



```
public class Software{
  private int __ID;
  private string __SWName;
  private string __Producer;
  private string __OSPlatform;
  private License __contains;
  ......
  public License Contains{
    set {
          ...... }
    get {
          ...... }
  ......
}
```

```
public class License{
private string __SerialNo;
private string __Licensee;
private string __PurchasedDate;
private Software __belongsTo;
  ......
public Software BelongsTo{
set {
...... }
get {
          ...... }
  ......
}
```

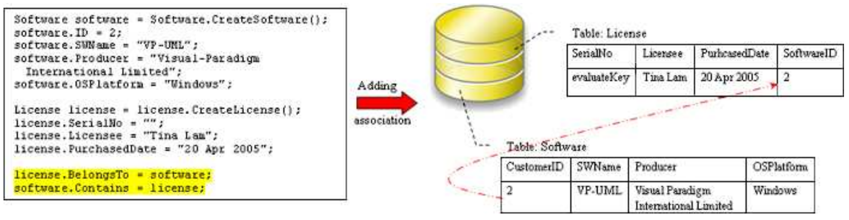
Figure 19.9 - Mapping One-to-one Association

From the above example, an ORM-Persistable object model of Software maps to an ORM-Persistable .NET class with an attribute of role, "__contains" in the association typed as its associated persistence class, License. Meanwhile, the object model of License maps to another .NET class with an attribute of role, "__belongsTo" typed as Software. It specifies the association that Software contains a particular License and a License belongs to particular software.

To manipulate the association, the following lines of code should be implemented.

```
Software software = Software.CreateSoftware();
License license = License.CreateLicense();
license.BelongsTo = software;
software.Contains = license;
```

In the above association, the primary key of the Software table will become a foreign key in License table when transforming to data model. Thus, after executing the lines of code, the primary key of the software will be referenced as a foreign key of the license record.



```
Software software = Software.CreateSoftware();
software.ID = 2;
software.SWName = "VP-UML";
software.Producer = "Visual-Paradigm
  International Limited";
software.OSPlatform = "Windows";

License license = license.CreateLicense();
license.SerialNo = "";
license.Licensee = "Tina Lam";
license.PurchasedDate = "20 Apr 2005";

license.BelongsTo = software;
software.Contains = license;
```

Table: License

SerialNo	Licensee	PurhcasedDate	SoftwareID
evaluateKey	Tina Lam	20 Apr 2005	2

Table: Software

CustomerID	SWName	Producer	OSPlatform
2	VP-UML	Visual Paradigm International Limited	Windows

Figure 19.10 - Code for adding one-to-one association

It is a bi-directional association with standard association handling. The association link can be created by setting both roles of Software and License.

One-to-Many Association

In a bi-directional one-to-many association, a class has multiplicity of one while the other has many. When transforming the association to persistent code, the role name will map to an attribute with data type of a collection class. For more detailed information on the .NET collection class, refer to the description in Using Collection section.

A bi-directional one-to-many association is shown below.

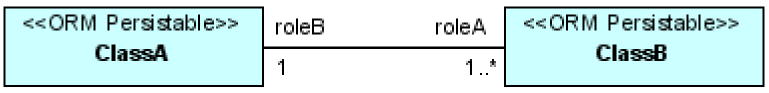


Figure 19.11 - One-to-many Association

With standard association handling, you have to create the association with the following steps in order to manipulate the bi-directional one-to-many association:

- Set the properties of role with an instance of the associated class; i.e. classB.RoleB = classA
- Add the objects to the collection of the associated class; i.e. classA.RoleA.Add(classB)

where classA is an object of ClassA; classB is an object of ClassB;
RoleA is the collection of ClassB; RoleB is the setter method of property, roleB.

After specifying the association, the value of primary key of the object of ClassA will be referenced as a foreign key of the associated object of ClassB.

Setting the property of role

For information on setting the properties of role, refer to the description in the One-to-One Association section.

Adding objects to the collection

In order to add an object to the collection, implement the program with the following steps:

1. Create a new instance of the class which associates with more than one instance of associated class.
2. Create a new object of the associated class.
3. Add a new object or an existing object of the associated class to the collection belonging to the class.

Table shows the method summary of the collection class to be used for adding a new object to it

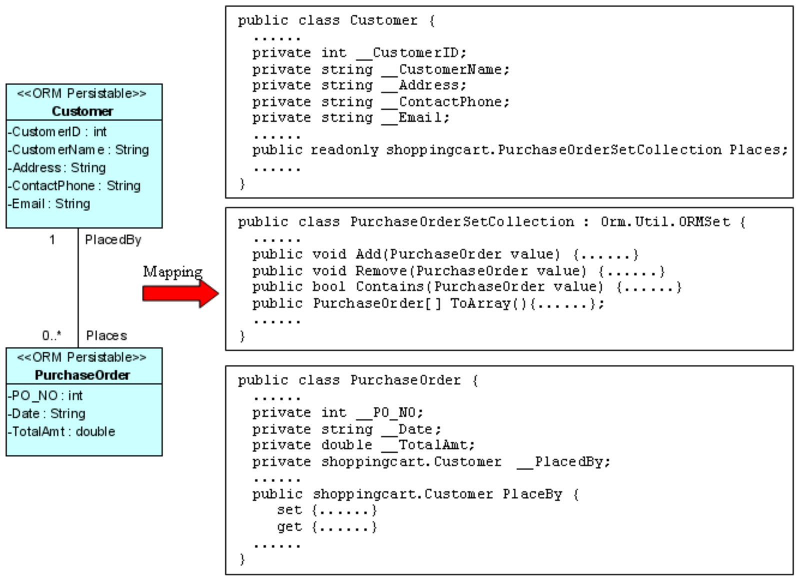
Return Type	Method Name	Description
void	Add(Class value)	Add a new object to the collection of the associated class.

Table 19.7

Remark:

1. Class should be replaced by the name of the associated class.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private string __Address;
  private string __ContactPhone;
  private string __Email;
  ......
  public readonly shoppingcart.PurchaseOrderSetCollection Places;
  ......
}
```

```
public class PurchaseOrderSetCollection : Orm.Util.ORMSet {
  ......
  public void Add(PurchaseOrder value) {......}
  public void Remove(PurchaseOrder value) {......}
  public bool Contains(PurchaseOrder value) {......}
  public PurchaseOrder[] ToArray(){......};
  ......
}
```

```
public class PurchaseOrder {
  ......
  private int __PO_NO;
  private string __Date;
  private double __TotalAmt;
  private shoppingcart.Customer  __PlacedBy;
  ......
  public shoppingcart.Customer PlaceBy {
     set {......}
     get {......}
  ......
}
```

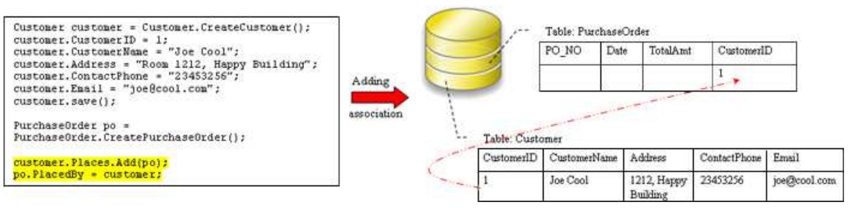
Figure 19.12 - Mapping One-to-many Association using Collection

From the above example, an ORM-Persistable object model of PurchaseOrder maps to an ORM-Persistable .NET class with an attribute of role, "__PlacedBy" in the association typed as an instance of Customer, specifying that the particular purchase order is placed by a particular customer. Moreover, the object model of Customer maps to a .NET class with an attribute of role, "__Places" in the association typed as a .NET collection class, ISet which is the specified type of collection manipulating instances of PurhcaseOrder.

To add a PurchaseOrder object to the collection of PurchaseOrder, the following lines of code should be implemented.

```
Customer customer = Customer.CreateCustomer();
PurchaseOrder po = PurchaseOrder.CreatePurchaseOrder();
customer.Places.Add(po);
po.PlacedBy = customer;
```

After executing these lines of code, an object is added to the collection representing the association. When inserting records to the database tables, the primary key of the customer will be referenced to as the foreign key of the purchase order record.



```
Customer customer = Customer.CreateCustomer();
customer.CustomerID = 1;
customer.CustomerName = "Joe Cool";
customer.Address = "Room 1212, Happy Building";
customer.ContactPhone = "23453256";
customer.Email = "joe@cool.com";
customer.save();

PurchaseOrder po =
PurchaseOrder.CreatePurchaseOrder();

customer.Places.Add(po);
po.PlacedBy = customer;
```

Table: PurchaseOrder

PO_NO	Date	TotalAmt	CustomerID
			1

Table: Customer

CustomerID	CustomerName	Address	ContactPhone	Email
1	Joe Cool	1212, Happy Building	23453256	joe@cool.com

Figure 19.13 - Code for one-to-many association using Collection

Retrieving objects from the collection

In order to retrieve an object from the collection, implement the program with the following steps:

1. Retrieve an instance of the class which associates with more than one instance of associated class.
2. Get the collection from the class, and convert the collection into an object array.

Example:

Refer to the example of the one-to-many association between the ORM-Persistable object models of Customer and PurchaseOrder, implement the following lines of code to retrieve a collection of PurchaseOrder objects from the customer.

```
Customer customer = Customer.LoadByName("Joe Cool");
PurchaseOrder[] orders = customer.Places.ToArray();
```

After executing these lines of code, the purchase order records associated with the customer are retrieved and stored in the object array.

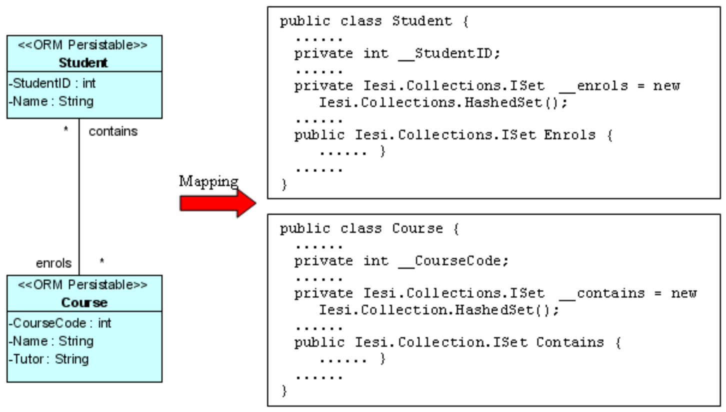
Note Retrieve the customer record by the LoadByName() method which is the method provided by the persistent API. For more information on retrieving persistent object, refer to the Persistent API section.

Many-to-Many Association

When transforming a many-to-many association between two ORM-Persistable classes with standard association handling, the role names will map to one type of collection defined in the object model. In order to specify the many-to-many association, add the objects to the corresponding collection of the associated class. For information on adding objects to the collection, refer to the description in the One-to-Many Association section.

In addition, a many-to-many association in the object model is transformed to data model by generating a Link Entity to form two one-to-many relationships between two generated entities. The primary keys of the two entities will migrate to the link entity as the primary/foreign keys. When specifying the association by adding objects to the collection, the primary key of the two related instances will be inserted as a pair for a row of the Link Entity automatically.

Example:



```
public class Student {
  ......
  private int __StudentID;
  ......
  private Iesi.Collections.ISet  __enrols = new
     Iesi.Collections.HashedSet();
  ......
  public Iesi.Collections.ISet Enrols {
     ...... }
  ......
}
```

```
public class Course {
  ......
  private int __CourseCode;
  ......
  private Iesi.Collections.ISet  __contains = new
     Iesi.Collection.HashedSet();
  ......
  public Iesi.Collection.ISet Contains {
     ...... }
  ......
}
```

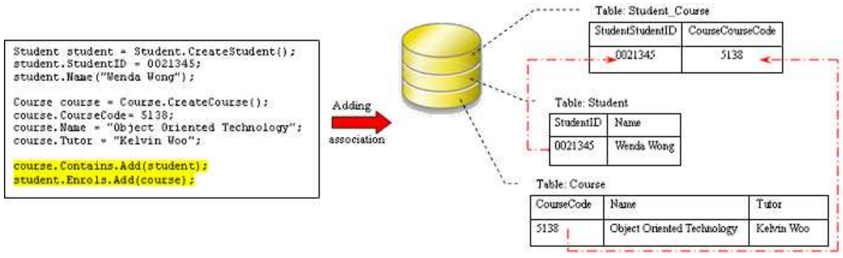
Figure 19.14 - Mapping Many-to-many association using Collection

With standard association handling, two classes, including Student and Course are generated from the above object model.

By executing the following lines of code:

```
Student student = Student.CreateStudent();
student.StudentID = 0021345;
student.Name = "Wenda Wong";
Course course = Course.CreateCourse();
course.CourseCode = 5138;
course.Name = "Object Oriented Technology";
course.Tutor = "Kelvin Woo";
course.Contains.Add(student);
student.Enrols.Add(course);
```

Both the Student table and Course table are inserted with a new record. After specifying the association by course.Contains.Add(student) and student.Entrols.Add(course), the corresponding primary keys of student record and course record migrate to the Student_Course Link Entity to form a row of record.



```
Student student = Student.CreateStudent();
student.StudentID = 0021345;
student.Name("Wenda Wong");

Course course = Course.CreateCourse();
course.CourseCode= 5138;
course.Name = "Object Oriented Technology";
course.Tutor = "Kelvin Woo";

course.Contains.Add(student);
student.Enrols.Add(course);
```

Table: Student_Course

StudentStudentID	CourseCourseCode
0021345	5138

Table: Student

StudentID	Name
0021345	Wenda Wong

Table: Course

CourseCode	Name	Tutor
5138	Object Oriented Technology	Kelvin Woo

Figure 19.15 - Code for adding Many-to-many association using Collection

Using Collection

With standard association handling, the role name which associates with more than one instance of the supplier class is transformed into one type of .NET collection class. The type of collection can be specified before the generation of code, refer to the description of Specifying Collection Type in the Object Model chapter.

The following table shows the mapping between the collection type defined in the association specification and the .NET collection class.

Collection Type	.NET Collection Class
Set	Iesi.Collections.ISet
Bag	System.Collections.IList
List	System.Collections.IList
Map	System.Collections.IDictionary

Table 19.8

Persistent API

Persistent API refers to the persistent code used to manipulate the persistent data. There are four types of persistent API available for generating the .NET persistence code. The four types of persistent API which include Static Method, Factory Class, POJO and Data Access Object (DAO), are capable of manipulating the persistent data with the relational database, i.e., inserting, retrieving, updating and deleting records.

Using Static Method

Generating the persistence code with static methods, the persistence class is generated with the static methods which is capable of creating, retrieving persistent object and persisting data. The following class diagram shows the dependency relationship between the client program and the generated persistence class.

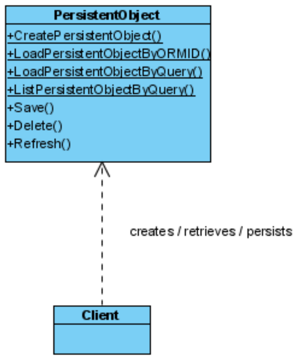

PersistentObject
+CreatePersistentObject()
+LoadPersistentObjectByORMID()
+LoadPersistentObjectByQuery()
+ListPersistentObjectByQuery()
+Save()
+Delete()
+Refresh()
creates / retrieves / persists
Client

Figure 19.16 - Class diagram with static methods

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram. For example, the ORM-Persistable class, Customer persists with the Customer data in the Customer table.

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram. For example, the ORM-Persistable class, Customer persists with the Customer data in the Customer table.

Example:

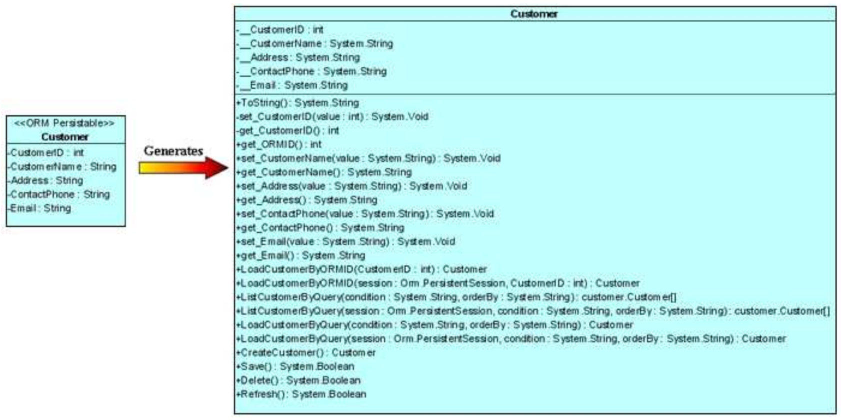

<<ORM Persistable>>
Customer
-CustomerID : int
-CustomerName : String
-Address : String
-ContactPhone : String
-Email : String
Generates
Customer
-__CustomerID : int
-__CustomerName : System.String
-__Address : System.String
-__ContactPhone : System.String
-__Email : System.String
+ToString() : System.String
-set_CustomerID(value : int) : System.Void
-get_CustomerID() : int
+get_ORMID() : int
+set_CustomerName(value : System.String) : System.Void
+get_CustomerName() : System.String
+set_Address(value : System.String) : System.Void
+get_Address() : System.String
+set_ContactPhone(value : System.String) : System.Void
+get_ContactPhone() : System.String
+set_Email(value : System.String) : System.Void
+get_Email() : System.String
+LoadCustomerByORMID(CustomerID : int) : Customer
+LoadCustomerByORMID(session : Orm.PersistentSession, CustomerID : int) : Customer
+ListCustomerByQuery(condition : System.String, orderBy : System.String) : customer.Customer[]
+ListCustomerByQuery(session : Orm.PersistentSession, condition : System.String, orderBy : System.String) : customer.Customer[]
+LoadCustomerByQuery(condition : System.String, orderBy : System.String) : Customer
+LoadCustomerByQuery(session : Orm.PersistentSession, condition : System.String, orderBy : System.String) : Customer
+CreateCustomer() : Customer
+Save() : System.Boolean
+Delete() : System.Boolean
+Refresh() : System.Boolean

Figure 19.17 - Mapping ORM Persistable Class

From the above example, a Customer persistence class is generated with a set of methods supporting the database manipulation.

In this section, it introduces how to use the static methods of the generated persistence classes to manipulate the persistent data with the relational database.

Creating a Persistent Object

As a persistence class represents a table in the database and an instance of the class represents a record of the table, creating a persistent object in the application system is the same as adding a row of new record to the table.

In order to insert a new row to the database table, implement the program with the following steps:

1. Create a new instance of the class.
2. Set the properties of the object.
3. Insert the object as a row to the database.

Table shows the method summary of the persistence class to be used for inserting a row to database.

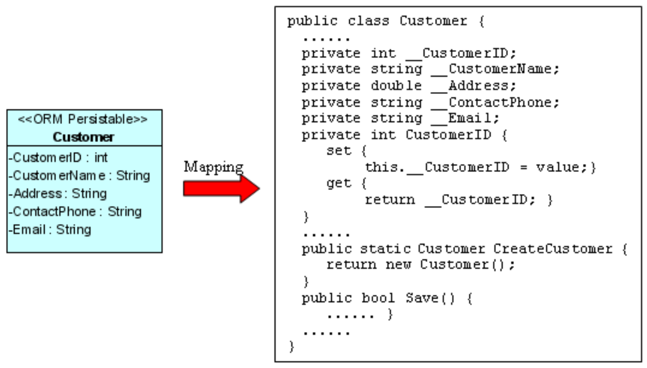
Return Type	Method Name	Description
Object	CreateClass()	Create a new instance of the class.
bool	Save()	Insert the object as a row to the database table.

Table 19.9

Remark:

1. **Object** is the newly created instance of the class.
2. **Class** should be replaced by the name of the generated persistence class.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
        this.__CustomerID = value;}
     get {
        return __CustomerID; }
  }
  ......
  public static Customer CreateCustomer {
     return new Customer();
  }
  public bool Save() {
     ...... }
  ......
}
```

Figure 19.18 - Mapping creator methods

From the above example, an ORM-Persistable object model of Customer maps to an ORM-Persistable .NET class generated with methods for creating a new instance and inserting the instance as a row of the database table.

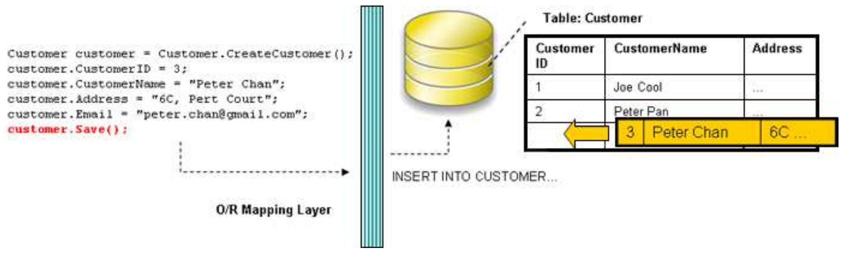
To insert a new Customer record to the table, Customer of the database, the following lines of code should be implemented.

```
Customer customer = Customer.CreateCustomer();
customer.CustomerID = 3;
customer.CustomerName = "Peter Chan";
customer.Address = "6C, Pert Court";
customer.Email = "peter.chan@gmail.com";
customer.Save();
```

After executing these lines of code, a row of record is inserted to the database table.

An alternative way to create a new instance of the class is using the new operator:
Class c = new Class();

From the above example, Customer customer = Customer.CreateCustomer() can be replaced by Customer customer = new Customer() to create a Customer object.



```
Customer customer = Customer.CreateCustomer();
customer.CustomerID = 3;
customer.CustomerName = "Peter Chan";
customer.Address = "6C, Pert Court";
customer.Email = "peter.chan@gmail.com";
customer.Save();
```

Table: Customer

Customer ID	CustomerName	Address
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	6C ...

Figure 19.19 - Code for create a records

Loading a Persistent Object

As the database table is represented by a persistence class, a record of the table can thus be represented by an instance. A record retrieved from the table will be stored as an object.

In order to retrieve a record from the database table, you have to specify the condition for finding the record. To retrieve a record, implement the program with the following steps:

1. Specify the condition for searching the record.
2. Load the retrieved record to an object.

Table shows the method summary of the persistence class

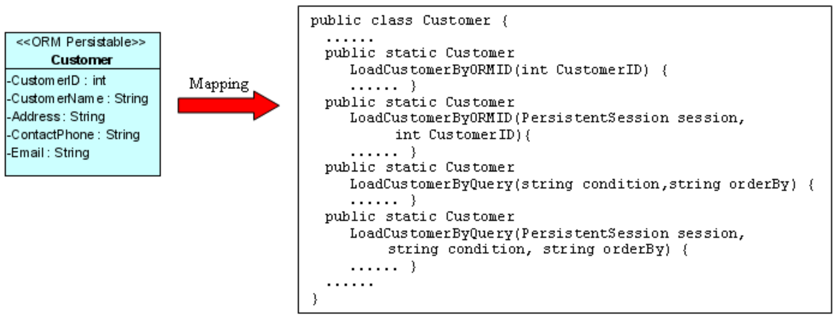
Return Type	Method Name	Description
Class	LoadClassByORMID(DataType PrimaryKey)	Retrieve a record matching with the specified value of primary key.
Class	LoadClassByORMID(PersistentSession session, DataType PrimaryKey)	Retrieve a record matching with the specified value of primary key and specified session.
Class	LoadClassByQuery(string condition, string orderBy)	Retrieve the first record matching the user defined condition while the matched records are ordered by a specified attribute.
Class	LoadClassByQuery(PersistentSession session, string condition, string orderBy)	Retrieve the first record matching the user defined condition and specified session while the matched records are ordered by a specified attribute.

Table 19.10

Remark:

1. **Class** should be replaced by the name of the persistence class.
2. DataType should be replaced by the data type of the attribute defined in the object model.

Example:



```
public class Customer {
  ......
  public static Customer
     LoadCustomerByORMID(int CustomerID) {
     ...... }
  public static Customer
     LoadCustomerByORMID(PersistentSession session,
           int CustomerID){
     ...... }
  public static Customer
     LoadCustomerByQuery(string condition,string orderBy) {
     ...... }
  public static Customer
     LoadCustomerByQuery(PersistentSession session,
          string condition, string orderBy) {
     ...... }
  ......
}
```

Figure 19.20 - Mapping load methods

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable .NET class generated with methods for retrieving a matched record.

To retrieve a record from the Customer table, the following line of code should be implemented.

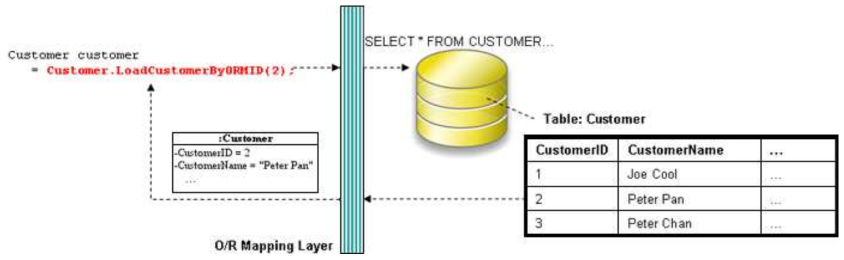
Loading an object by passing the value of primary key:

```
Customer customer = Customer.LoadCustomerByORMID(2);
```

Loading an object by specifying a user defined condition:

```
Customer customer = Customer.LoadCustomerByQuery("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

After executing the code, a matched row is retrieved and loaded to a Customer object.



```
Customer customer
   = Customer.LoadCustomerByORMID(2);
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 19.21 - Code for load a record

Updating a Persistent Object

As a record can be retrieved from the table and loaded to an object of the persistence class, the record is allowed to update by simply using the setter method of the property.

In order to update a record, you have to retrieve the row being updated, update the value by setting the property to the database. To update the record, implement the program with the following steps:

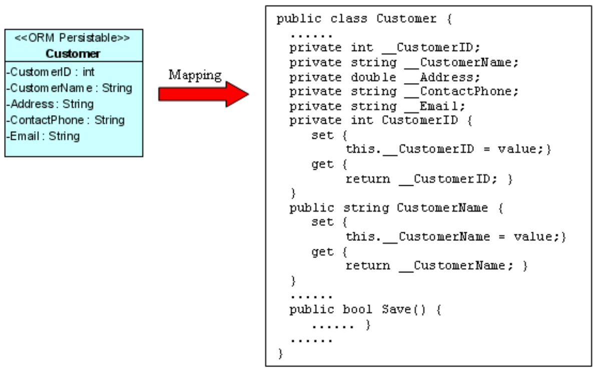
1. Retrieve a record from database table and load as an object.
2. Set the updated value to the property of the object.
3. Save the updated record to the database.

Table shows the method summary of the persistence class to be used for retrieving a record from database.

Return Type	Method Name	Description
bool	Save()	Update the value to database.

Table 19.11

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
          this.__CustomerID = value;}
     get {
          return __CustomerID; }
  }
  public string CustomerName {
     set {
          this.__CustomerName = value;}
     get {
          return __CustomerName; }
  }
  ......
  public bool Save() {
     ...... }
  ......
}
```

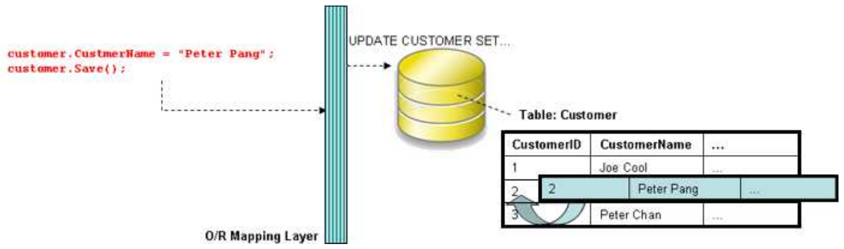
Figure 19.22 - Mapping for update record

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable .NET class generated with the methods for setting the properties and updating the row.

To update a Customer record, the following lines of code should be implemented.

```
customer.CustomerName = "Peter Pang";
customer.Save();
```

After executing the above lines of code, the customer name is updated to "Peter Pang" in the database.



```
customer.CustmerName = "Peter Pang";
customer.Save();
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pang	...
3	Peter Chan	...

Figure 19.23 - Code for update record

Deleting a Persistent Object

As a record can be retrieved from the table and loaded to an object of the persistence class, the record can be deleted by simply using the delete method of the persistence class.

In order to delete a record, implement the program with the following steps:

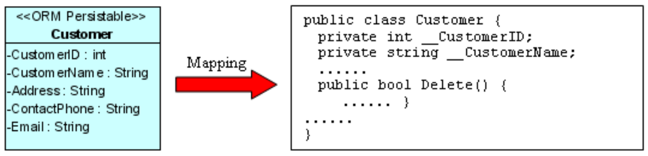
1. Retrieve a record from database table and load as an object.
2. Delete the retrieved record.

Table shows the method summary of the persistence class to be used for

Return Type	Method Name	Description
bool	Delete()	Delete the current instance.

Table 19.12

Example:



```
public class Customer {
  private int __CustomerID;
  private string __CustomerName;
  ......
  public bool Delete() {
     ...... }
......
}
```

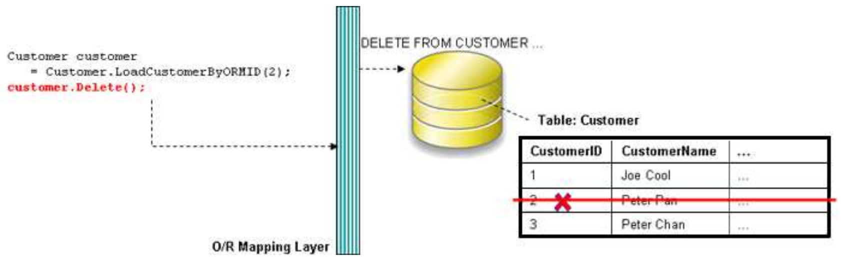
Figure 19.24 - Mapping delete methods

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable .NET class generated with the methods for deleting the specified record from the database.

To delete a Customer record, the following lines of code should be implemented.

```
Customer customer = Customer.LoadCustomerByORMID(2);
customer.Delete();
```

After executing the above code, the specified customer record is deleted from the database.



```
Customer customer
   = Customer.LoadCustomerByORMID(2);
customer.Delete();
```

CustomerID	CustomerName	...
1	Joe Cool	...
~~2~~	~~Peter Pan~~	~~...~~
3	Peter Chan	...

Figure 19.25 - Code for delete a record

Querying

For most of the database application, the database is enriched with information. Information may be requested from the database so as to performing an application function, such as retrieving the unit price of products for calculating the total amount for a purchase order.

As the persistence class represents a table, the ORM-Persistable .NET class is generated with methods for retrieving information from the database.

Loading a Collection of Persistent Objects

As the database table usually contains many records, you may want to query the tables by a specified condition. The generated persistence class supports querying the database, the matched records will be retrieved and loaded as an object array.

In order to retrieve records from the table, you have to specify the condition for querying. To retrieve a number of records, implement the program with the following steps:

1. Specify the condition for searching the record.
2. Load the retrieved records as an object array.

Table shows the method summary of the persistence class to be used for retrieving records from database table.

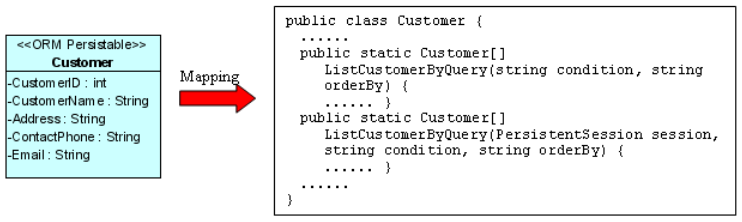
Return Type	Method Name	Description
Class[]	ListClassByQuery(string condition, string orderBy)	Retrieve the records matched with the user defined condition and ordered by a specified attribute.
Class[]	ListClassByQuery(PersistentSession session, string condition, string orderBy)	Retrieve the records matched with the user defined condition and specified session and ordered by a specified attribute.

Table 19.13

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  ......
  public static Customer[]
     ListCustomerByQuery(string condition, string
     orderBy) {
     ...... }
  public static Customer[]
     ListCustomerByQuery(PersistentSession session,
     string condition, string orderBy) {
     ...... }
  ......
}
```

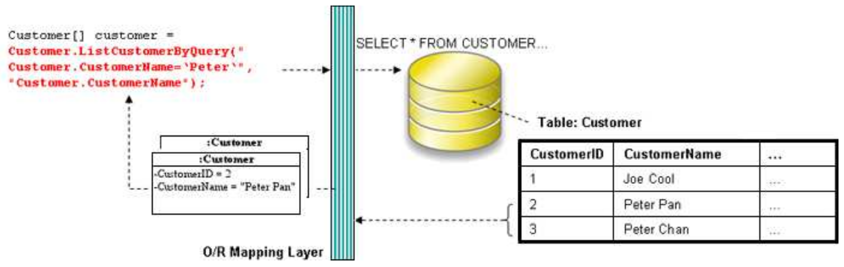
Figure 19.26 - Mapping list methods

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable .NET class generated with methods for retrieving records.

To retrieve records from the Customer table, the following line of code should be implemented.

```
Customer[] customer = Customer.ListCustomerByQuery("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

After executing the code, the matched rows are retrieved and loaded to an object array of Customer.



```
Customer[] customer =
Customer.ListCustomerByQuery("
Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 19.27 - Code for retrieve a list of records

Using ORM Qualifier

ORM Qualifier is an additional feature which allows you to specify the extra data retrieval rules apart from the system pre-defined rules. The ORM Qualifier can be defined in order to generate persistence code. Refer to Defining ORM Qualifier in the Object Model chapter for more information.

By defining the ORM Qualifier in a class, the persistence class will be generated with additional data retrieval methods, load and list methods.

Table shows the method summary generated by defining the ORM Qualifier.

Return Type	Method Name	Description
Class	LoadByORMQualifier(DataType attribute)	Retrieve the first record that matches the specified value with the attribute defined in the ORM Qualifier.
Class	LoadByORMQualifier (PersistentSession session, DataType attribute)	Retrieve the first record that matches the specified value with the attribute defined in the ORM Qualifier and specified session.

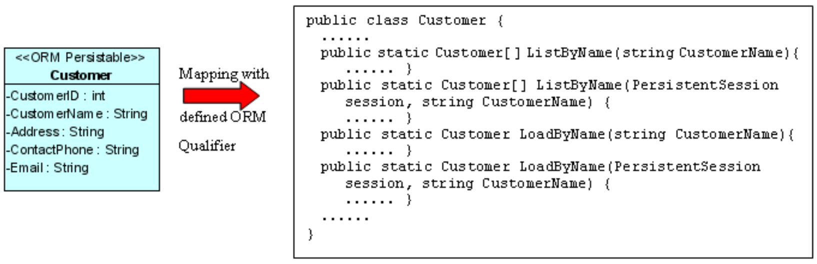
Class[]	ListByORMQualifier (DataType attribute)	Retrieve the records matched that matches the specified value with the attribute defined in the ORM Qualifier.
Class[]	ListByORMQualifier (PersistentSession session, DataType attribute)	Retrieve the records that match the specified value with the attribute defined in the ORM Qualifier and specified session.

Table 19.14

Remark:

1. **Class** should be replaced by the name of the persistence class.
2. **ORMQualifier** should be replaced by the Name defined in the ORM Qualifier.
3. **DataType** should be replaced by the data type of the attribute which associated with the ORM Qualifier.
4. **attribute** is the specified value to be used for querying the table.

Example:



```
public class Customer {
  ......
  public static Customer[] ListByName(string CustomerName){
     ...... }
  public static Customer[] ListByName(PersistentSession
     session, string CustomerName) {
     ...... }
  public static Customer LoadByName(string CustomerName){
     ...... }
  public static Customer LoadByName(PersistentSession
     session, string CustomerName) {
     ...... }
  ......
}
```

Figure 19.28 - Mapping load and list method with qualifier

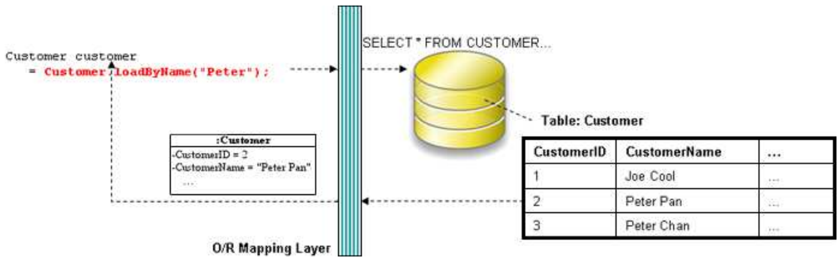
In the above example, a customer object model is defined with an ORM Qualifier named as Name and qualified with the attribute, CustomerName.

To query the Customer table with the ORM Qualifier in one of the two way

- By Load method

  ```
  Customer customer = Customer.LoadByName("Peter");
  ```

  After executing the code, the first occurrence of "Peter" in the CustomerName column in the Customer table will be loaded to the object identified as customer.



```
Customer customer
   = Customer.LoadByName("Peter");
```

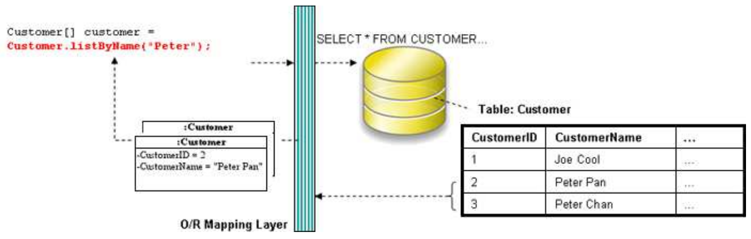
CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 19.29 - Retrieve a record by qualifier

- By List method

```
Customer[] customer = Cudtomer.ListByName("Peter");
```

After executing the code, all rows which contain "Peter" in the CustomerName column in the Customer table will be retrieved and stored in an array of Customer object.



```
Customer[] customer =
Customer.listByName("Peter");
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 19.30 - Retrieve a list of records by qualifier

Using Criteria Class

When generating the persistence class for each ORM-Persistable class defined in the object model, the corresponding criteria class can also be generated. Criteria class is a helper class which provides an additional way to specify the condition to retrieve records from the database. Refer to Using Criteria Class section for more detailed information on how to specify the conditions to query the database by the criteria class.

You can get the retrieved records from the criteria class in one of the two ways:

- Use the retrieval methods of the criteria class.

  For information on the retrieval methods provided by the criteria class, refer to the description of Loading Retrieved Records section.

- Use the retrieval by criteria methods of the persistence class.

Table shows the method summary generated to the persistence class

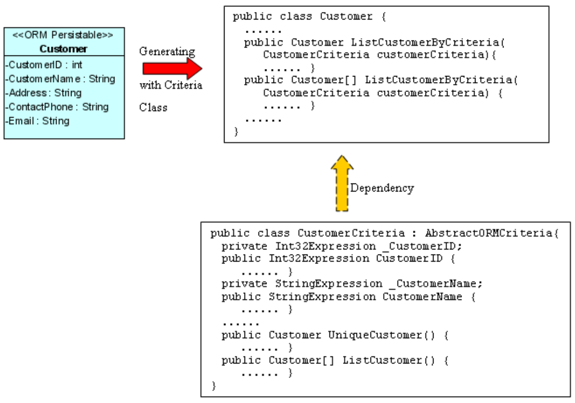
Return Type	Method Name	Description
Class	LoadClassByCriteria(ClassCriteria value)	Retrieve the single record that matches the specified conditions applied to the criteria class.
Class[]	ListClassByCriteria(ClassCriteria value)	Retrieve the records that match the specified conditions applied to the criteria class.

Table 19.15

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  ......
  public Customer ListCustomerByCriteria(
     CustomerCriteria customerCriteria){
     ...... }
  public Customer[] ListCustomerByCriteria(
     CustomerCriteria customerCriteria) {
     ...... }
  ......
}
```

```
public class CustomerCriteria : AbstractORMCriteria{
  private Int32Expression _CustomerID;
  public Int32Expression CustomerID {
     ...... }
  private StringExpression _CustomerName;
  public StringExpression CustomerName {
     ...... }
  ......
  public Customer UniqueCustomer() {
     ...... }
  public Customer[] ListCustomer() {
     ...... }
}
```

Figure 19.31 - Mapping with Criteria Class

To retrieve records from the Criteria Class in one of the two ways:

- By Load method

```
CustomerCriteria customerCriteria = new customerCriteria.CustomerName.Like("%Peter%");
Customer customer = Customer.LoadCustomerByCriteria(customerCriteria);
```

After executing the code, the first occurrence of "Peter" in the CustomerName column in the Customer table will be loaded to the object identified as customer.

- By List method

```
CustomerCriteria customerCriteria = new CustomerCriteria.CustomerName.Like("%Peter%");
Customer[] customer = Customer.ListCustomerByCriteria(customerCriteria);
```

After executing the code, all rows which contain "Peter" in the CustomerName column in the Customer table will be retrieved and stored in an array of Customer object.

Using Factory Class

Generating the persistence code with factory class, not only the persistence class will be generated, but also the corresponding factory class for each ORM-Persistable class.

The generated factory class is capable of creating a new instance of the persistence class, which assists in inserting a new record to the database, and retrieving record(s) from the database by specifying the condition for query. After an instance is created by the factory class, the persistence class allows setting the values of its property and updating into the database. The persistence class also supports the deletion of records.

The following class diagram shows the relationship between the client program, persistent object and factory object.

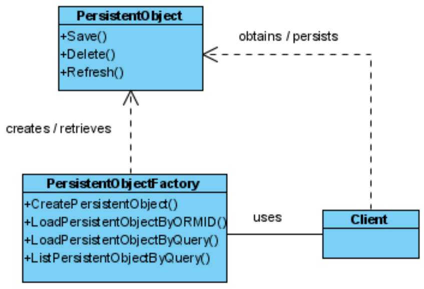

PersistentObject
+Save()
+Delete()
+Refresh()
obtains / persists
creates / retrieves
PersistentObjectFactory
+CreatePersistentObject()
+LoadPersistentObjectByORMID()
+LoadPersistentObjectByQuery()
+ListPersistentObjectByQuery()
uses
Client

Figure 19.32 - Class Diagram for Factory Class

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram while the PersistentObjectFactory refers to the generated factory class of the ORM-Persistable class. For example, the CustomerFactory class creates and retrieves Customer persistent object while the ORM-Persistable class, Customer persists with the Customer data in the Customer table.

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram while the PersistentObjectFactory refers to the generated factory class of the ORM-Persistable class. For example, the CustomerFactory class creates and retrieves Customer persistent object while the ORM-Persistable class, Customer persists with the Customer data in the Customer table.

Example:

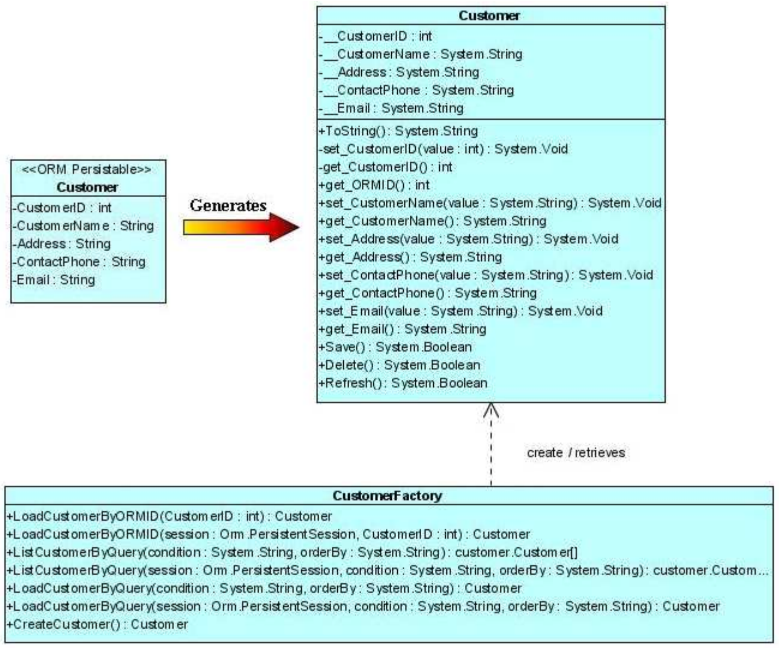

<<ORM Persistable>>
Customer
-CustomerID : int
-CustomerName : String
-Address : String
-ContactPhone : String
-Email : String
Generates
Customer
-__CustomerID : int
-__CustomerName : System.String
-__Address : System.String
-__ContactPhone : System.String
-__Email : System.String
+ToString() : System.String
-set_CustomerID(value : int) : System.Void
-get_CustomerID() : int
+get_ORMID() : int
+set_CustomerName(value : System.String) : System.Void
+get_CustomerName() : System.String
+set_Address(value : System.String) : System.Void
+get_Address() : System.String
+set_ContactPhone(value : System.String) : System.Void
+get_ContactPhone() : System.String
+set_Email(value : System.String) : System.Void
+get_Email() : System.String
+Save() : System.Boolean
+Delete() : System.Boolean
+Refresh() : System.Boolean
create / retrieves
CustomerFactory
+LoadCustomerByORMID(CustomerID : int) : Customer
+LoadCustomerByORMID(session : Orm.PersistentSession, CustomerID : int) : Customer
+ListCustomerByQuery(condition : System.String, orderBy : System.String) : customer.Customer[]
+ListCustomerByQuery(session : Orm.PersistentSession, condition : System.String, orderBy : System.String) : customer.Custom...
+LoadCustomerByQuery(condition : System.String, orderBy : System.String) : Customer
+LoadCustomerByQuery(session : Orm.PersistentSession, condition : System.String, orderBy : System.String) : Customer
+CreateCustomer() : Customer

Figure 19.33 - Class Diagram with Factory Class

From the above example, the CustomerFactory class supports the creation of Customer persistent object, the retrieval of Customer records from the Customer table while the Customer persistence class supports the update and deletion of customer record.

Creating a Persistent Object

An instance of a persistence class represents a record of the corresponding table, creating a persistent object corresponds to inserting a new record to the table.

In order to insert a new row to the database table, implement the program with the following steps:

1. Create a new instance of the class by the factory class.
2. Set the properties of the object by the persistence class.
3. Insert the object as a row to the database by the persistence class.

Table shows the method summary of the factory and persistence classes to be used for inserting a row to database.

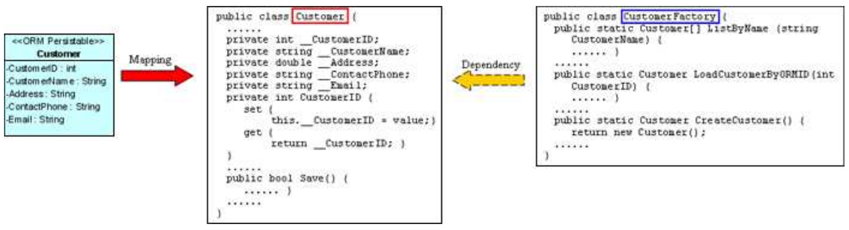
Class Type	Return Type	Method Name	Description
Factory Class	Class	CreateClass()	Create a new instance of the class.
Persistence Class	bool	Save()	Insert the object as a row to the database table.

Table 19.16

Remark:

1. **Class** should be replaced by the name of the generated persistence class.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
         this.__CustomerID = value;}
     get {
         return __CustomerID; }
  }
  ......
  public bool Save() {
     ...... }
  ......
}
```

```
public class CustomerFactory {
  public static Customer[] ListByName (string
     CustomerName) {
     ...... }
  ......
  public static Customer LoadCustomerByORMID(int
     CustomerID) {
     ...... }
  ......
  public static Customer CreateCustomer() {
     return new Customer();
  ......
}
```

Figure 19.34 - Mapping with Factory Class

From the above example, an ORM-Persistable object model of Customer maps to an ORM-Persistable .NET class with methods for setting the properties. An ORM-Persistable factory class is generated with method for creating a new instance and; and thus these methods allow inserting the instance as a row of the database table.

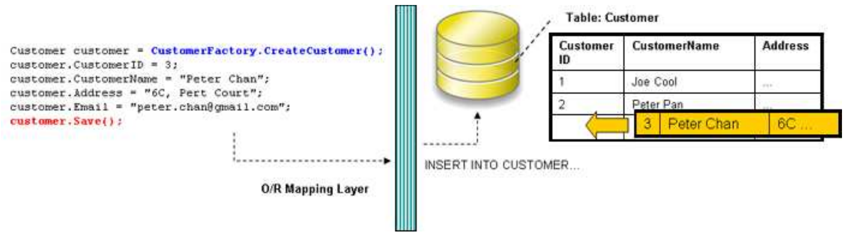
To insert a new Customer record to the table, Customer of the database, the following lines of code should be implemented.

```
Customer customer = CustomerFactory.CreateCustomer();
customer.CustomerID = 3;
customer.CustomerName = "Peter Chan";
customer.Address = "6C, Pert Court";
customer.Email = "peter.chan@gmail.com";
customer.Save();
```

After executing these lines of code, a row of record is inserted to the database table.

An alternative way to create a new instance of the class is using the new operator:
Class c = new Class();

From the above example, Customer customer = CustomerFactory.CreateCustomer() can be replaced by Customer customer = new Customer() to create a Customer object.



```
Customer customer = CustomerFactory.CreateCustomer();
customer.CustomerID = 3;
customer.CustomerName = "Peter Chan";
customer.Address = "6C, Pert Court";
customer.Email = "peter.chan@gmail.com";
customer.Save();
```

Table: Customer

Customer ID	CustomerName	Address
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	6C ...

Figure 19.35 - Code for create records using factory class

Loading a Persistent Object

As an instance of a persistence class represents a record of the corresponding table, a record retrieved from the table will be stored as an object.

In order to retrieve a record from the database table, you have to specify the condition for finding the record. To retrieve a record, implement the program with the following steps:

1. Specify the condition for searching the record by the factory class.
2. Load the retrieved record to an object.

Table shows the method summary of the factory class to be used for retrieving a record from database.

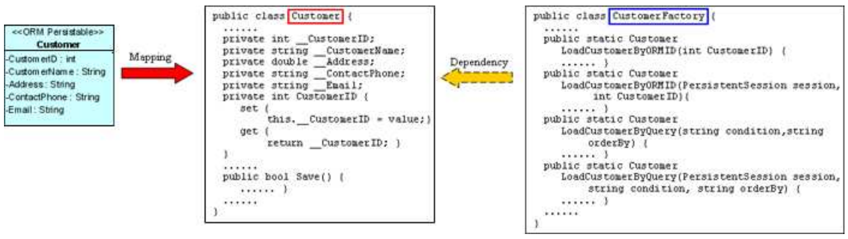
Class Type	Return Type	Method Name	Description
Factory Class	Class	LoadClassByORMID(DataType PrimaryKey)	Retrieve a record matching with the specified value of primary key.
Factory Class	Class	LoadClassByORMID(PersistentSession session, DataType PrimaryKey)	Retrieve a record matching with the specified value of primary key and specified session.
Factory Class	Class	LoadClassByQuery(string condition, string orderBy)	Retrieve the first record matching the user defined condition while the matched records are ordered by a specified attribute.
Factory Class	Class	LoadClassByQuery(PersistentSession session, string condition, string orderBy)	Retrieve the first record matching the user defined condition and specified session while the matched records are ordered by a specified attribute.

Table 19.17

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
         this.__CustomerID = value;}
     get {
         return __CustomerID; }
  }
  ......
  public bool Save() {
     ...... }
  ......
}
```

```
public class CustomerFactory {
  ......
  public static Customer
     LoadCustomerByORMID(int CustomerID) {
     ...... }
  public static Customer
     LoadCustomerByORMID(PersistentSession session,
         int CustomerID){
     ...... }
  public static Customer
     LoadCustomerByQuery(string condition,string
         orderBy) {
     ...... }
  public static Customer
     LoadCustomerByQuery(PersistentSession session,
         string condition, string orderBy) {
     ...... }
  ......
}
```

Figure 19.36 - Mapping load methods in factory class

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable .NET class. A factory class is generated with methods for retrieving a matched record.

To retrieve a record from the Customer table, the following line of code should be implemented.

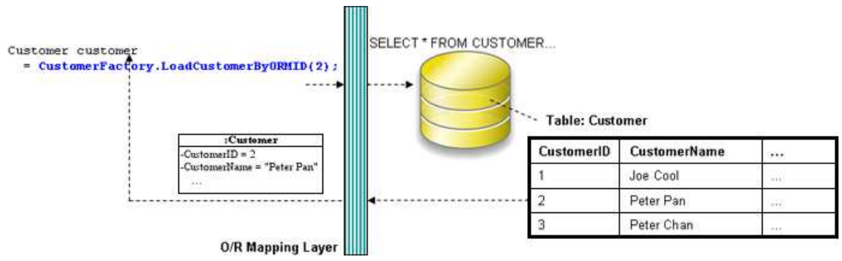
Loading an object by passing the value of primary key:

```
Customer customer = CustomerFactory.LoadCustomerByORMID(2);
```

Loading an object by specifying a user defined condition:

```
Customer customer = CustomerFactory.LoadCustomerByQuery("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

After executing the code, a matched row is retrieved and loaded to a Customer object.



```
Customer customer
  = CustomerFactory.LoadCustomerByORMID(2);
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 19.37 - Code for load a records

Updating a Persistent Object

As a record can be retrieved from the table and loaded as an instance of the persistence class, setting a new value to the attribute by its setter method supports the update of record.

In order to update a record, you have to first retrieve the row to be updated, and then set a new value to the property, finally update to the database. To update the record, implement the program with the following steps:

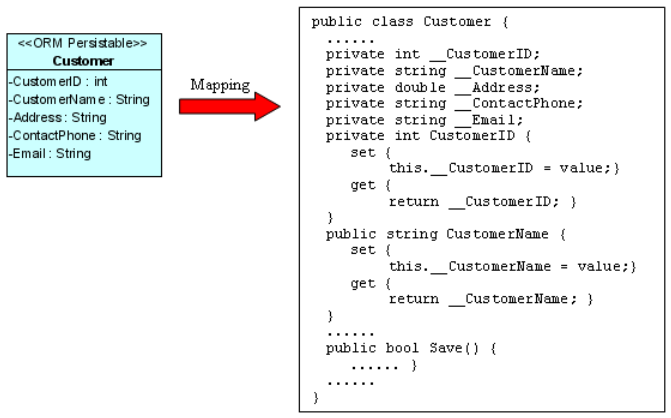
1. Retrieve a record from database table and load as an object by the factory class.
2. Set the updated value to the property of the object by the persistence class.
3. Save the updated record to the database by the persistence class.

Table shows the method summary of the persistence class to be used for updating a record.

Class Type	Return Type	Method Name	Description
Persistence Class	bool	Save()	Update the value to database.

Table 19.18

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
          this.__CustomerID = value;}
     get {
          return __CustomerID; }
  }
  public string CustomerName {
     set {
          this.__CustomerName = value;}
     get {
          return __CustomerName; }
  }
  ......
  public bool Save() {
     ...... }
  ......
}
```

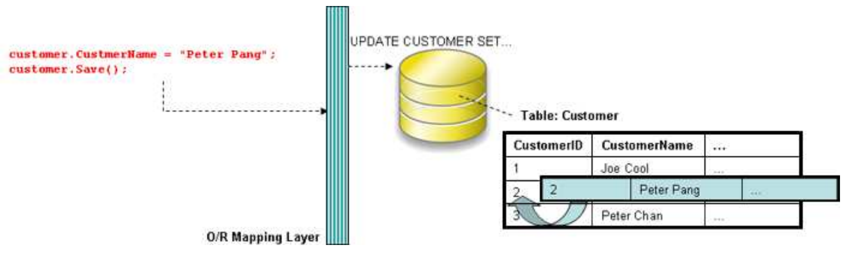
Figure 19.38 - Mapping save method

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable .NET class generated with the methods for setting the properties and updating the row.

To update a Customer record, the following lines of code should be implemented.

```
customer.CustomerName = "Peter Pang";
customer.Save();
```

After executing the above lines of code, the customer name is updated to "Peter Pang" in the database.



```
customer.CustmerName = "Peter Pang";
customer.Save();
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pang	...
3	Peter Chan	...

Figure 19.39 - Code for insert a record

Deleting a Persistent Object

As a record can be retrieved from the table and loaded to an object of the persistence class, the record can be deleted by simply using the delete method of the persistence class.

In order to delete a record, implement the program with the following steps:

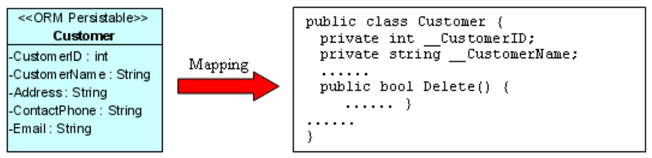
1. Retrieve a record from database table and load as an object by the factory class.
2. Delete the retrieved record by the persistence class.

Table shows the method summary of the persistence class to be used for deleting a record from database.

Class Type	Return Type	Method Name	Description
Persistence Class	bool	Delete()	Delete the current instance.

Table 19.19

Example:



```
public class Customer {
  private int __CustomerID;
  private string __CustomerName;
  ......
  public bool Delete() {
     ...... }
......
}
```

Figure 19.40 - Mapping delete methods

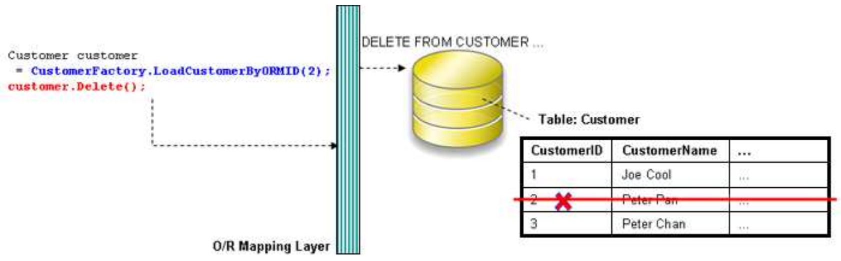
From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable .NET class generated with the methods for deleting the specified record from the database.

To delete a Customer record, the following lines of code should be implemented.

```
Customer customer = CustomerFactory.LoadCustomerByORMID(2);
customer.Delete();
```

>

After executing the above code, the specified customer record is deleted from the database.



```
Customer customer
  = CustomerFactory.LoadCustomerByORMID(2);
customer.Delete();
```

CustomerID	CustomerName	...
1	Joe Cool	...
~~2~~	~~Peter Pan~~	~~...~~
3	Peter Chan	...

Figure 19.41 - Code for delete record

Querying

It is well known that the database is enriched with information. In some cases, information may be retrieved from the database to assist another function of the application, such as retrieving the unit price of products for calculating the total amount for a purchase order.

As the persistence factory class supports the retrieval of records, using the methods of the factory class can retrieve records from the database according to the specified condition.

Loading a Collection of Persistent Objects

As the database table usually contains many records, you may want to query the tables by a specified condition. The generated factory class supports querying the database, the matched records will be retrieved and loaded as an object array.

In order to retrieve records from the table, you have to specify the condition for querying. To retrieve a number of records, implement the program with the following steps:

1. Specify the condition for searching the record by the factory class.
2. Load the retrieved records as an object array by the factory class.

Table shows the method summary of the factory class to be used for

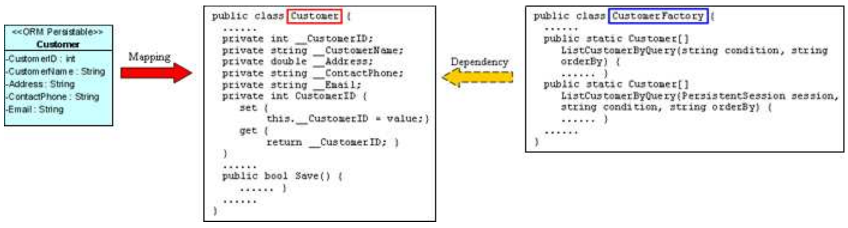
Class Type	Return Type	Method Name	Description
Factory Class	Class[]	ListClassByQuery(string condition, string orderBy)	Retrieve the records matched with the user defined condition and ordered by a specified attribute.
Factory Class	Class[]	ListClassByQuery(PersistentSession session, string condition, string orderBy)	Retrieve the records matched with the user defined condition and specified session and ordered by a specified attribute.

Table 19.20

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
         this.__CustomerID = value;}
     get {
         return __CustomerID; }
  }
  ......
  public bool Save() {
     ...... }
  ......
}
```

```
public class CustomerFactory {
  ......
  public static Customer[]
     ListCustomerByQuery(string condition, string
     orderBy) {
     ...... }
  public static Customer[]
     ListCustomerByQuery(PersistentSession session,
     string condition, string orderBy) {
     ...... }
  ......
}
```

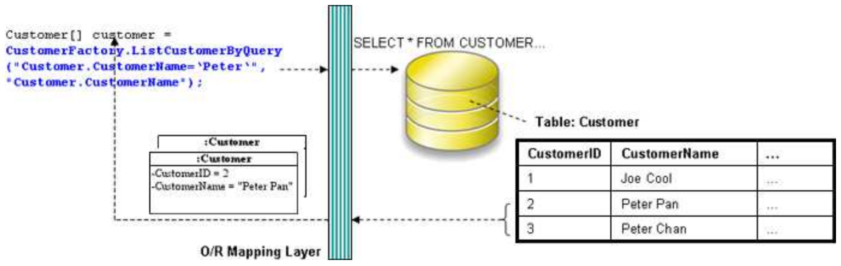
Figure 19.42 - Mapping list method in Factory

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable .NET factory class generated with methods for retrieving records.

To retrieve records from the Customer table, the following line of code should be implemented.

```
Customer[] customer = CustomerFactory.ListCustomerByQuery("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

After executing the code, the matched rows are retrieved and loaded to an object array of Customer.



```
Customer[] customer =
CustomerFactory.ListCustomerByQuery
("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 19.43 - Retrieve a list of records by using Factory Class

Using ORM Qualifier

ORM Qualifier is an additional feature which allows you to specify the extra data retrieval rules apart from the system pre-defined rules. The ORM Qualifier can be defined in order to generate persistence code. Refer to Defining ORM Qualifier in the Object Model chapter for more information.

By defining the ORM Qualifier in a class, the factory class will be generated with additional data retrieval methods, load and list methods.

Table shows the method summary generated to the factory class by defining the ORM Qualifier.

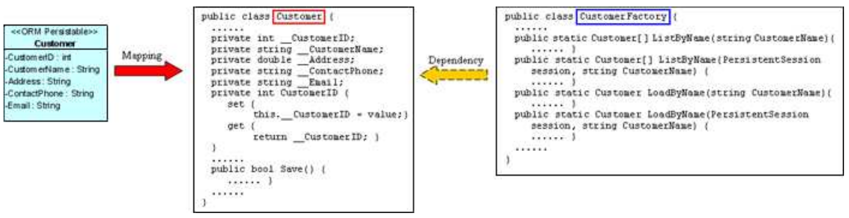
Class Type	Return Type	Method Name	Description
Factory Class	Class	LoadByORMQualifier(DataType attribute)	Retrieve the first record that matches the specified value with the attribute defined in the ORM Qualifier.
Factory Class	Class	LoadByORMQualifier (PersistentSession session, DataType attribute)	Retrieve the first record that matches the specified value with the attribute defined in the ORM Qualifier and specified session.
Factory Class	Class[]	ListByORMQualifier (DataType attribute)	Retrieve the records matched that matches the specified value with the attribute defined in the ORM Qualifier.
Factory Class	Class[]	ListByORMQualifier (PersistentSession session, DataType attribute)	Retrieve the records that match the specified value with the attribute defined in the ORM Qualifier and specified session.

Table 19.21

Remark:

1. **Class** should be replaced by the name of the persistence class.
2. **ORMQualifier** should be replaced by the Name defined in the ORM Qualifier.
3. **DataType** should be replaced by the data type of the attribute which associated with the ORM Qualifier.
4. **attribute** is the specified value to be used for querying the table.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
          this.__CustomerID = value;}
     get {
          return __CustomerID; }
  }
  ......
  public bool Save() {
     ...... }
  ......
}
```

```
public class CustomerFactory {
  ......
  public static Customer[] ListByName(string CustomerName){
     ...... }
  public static Customer[] ListByName(PersistentSession
     session, string CustomerName) {
     ...... }
  public static Customer LoadByName(string CustomerName){
     ...... }
  public static Customer LoadByName(PersistentSession
     session, string CustomerName) {
     ...... }
  ......
}
```

Figure 19.44 - Mapping list and load by qualifier methods

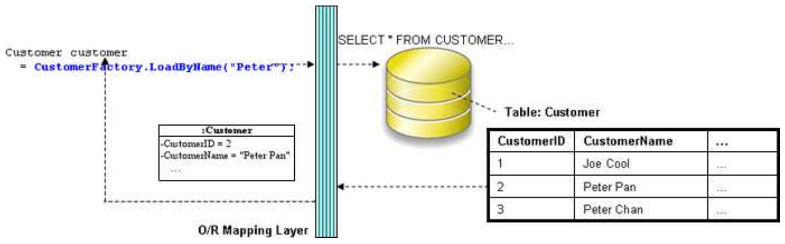
In the above example, a customer object model is defined with an ORM Qualifier named as Name and qualified with the attribute, CustomerName.

To query the Customer table with the ORM Qualifier in one of the two way

- By Load method

```
Customer customer = CustomerFactory.LoadByName("Peter");
```

After executing the code, the first occurrence of "Peter" in the CustomerName column in the Customer table will be loaded to the object identified as customer.



```
Customer customer
  = CustomerFactory.LoadByName("Peter");
```

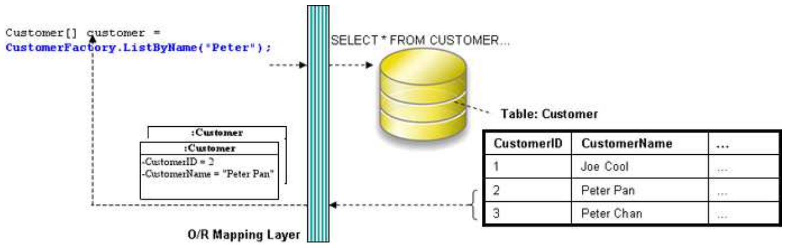
CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 19.45 - Code for load a record by qualifier

- By List method

```
Customer[] customer = CustomerFactory.listByName("Peter")
```

After executing the code, all rows which contain "Peter" in the CustomerName column in the Customer table will be retrieved and stored in an array of Customer object.



```
Customer[] customer =
CustomerFactory.ListByName("Peter");
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 19.46 - Code for retrieve a list of record by qualifier

Using Criteria Class

When generating the persistence class for each ORM-Persistable class defined in the object model, the corresponding criteria class can also be generated. Criteria class is a helper class which provides an additional way to specify the condition to retrieve records from the database. Refer to Using Criteria Class section for more detailed information on how to specify the conditions to query the database by the criteria class.

You can get the retrieved records from the criteria class in one of the two ways:

- Use the retrieval methods of the criteria class.

  For information on the retrieval methods provided by the criteria class, refer to the description of Loading Retrieved Records section.

- Use the retrieval by criteria methods of the factory class.

Table shows the method summary generated to the persistence class to retrieve records from the criteria class.

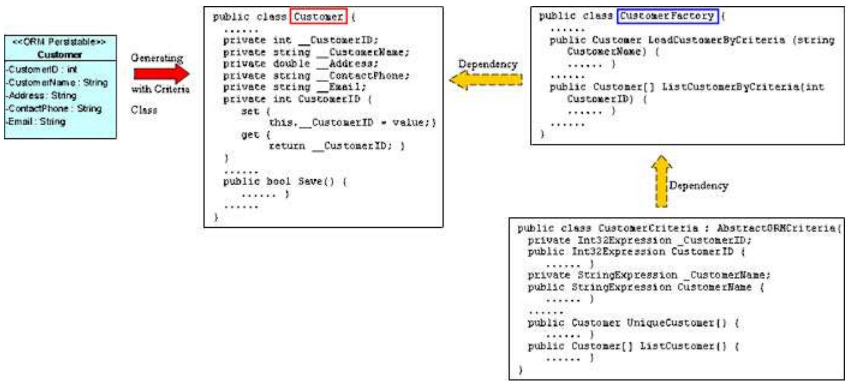
Class Type	Return Type	Method Name	Description
Factory Class	Class	LoadClassByCriteria(ClassCriteria value)	Retrieve the single record that matches the specified conditions applied to the criteria class.
Factory Class	Class[]	ListClassByCriteria(ClassCriteria value)	Retrieve the records that match the specified conditions applied to the criteria class.

Table 19.22

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
         this.__CustomerID = value;}
     get {
         return __CustomerID; }
  }
  ......
  public bool Save() {
     ...... }
  ......
}
```

```
public class CustomerFactory {
  ......
  public Customer LoadCustomerByCriteria (string
     CustomerName) {
     ...... }
  ......
  public Customer[] ListCustomerByCriteria(int
     CustomerID) {
     ...... }
  ......
}
```

```
public class CustomerCriteria : AbstractORMCriteria{
  private Int32Expression _CustomerID;
  public Int32Expression CustomerID {
     ...... }
  private StringExpression _CustomerName;
  public StringExpression CustomerName {
     ...... }
  ......
  public Customer UniqueCustomer() {
     ...... }
  public Customer[] ListCustomer() {
     ...... }
}
```

Figure 19.47 - Mapping with Factory and Criteria Class

To retrieve records from the Criteria Class in one of the two ways:

- By Load method

```
CustomerCriteria customerCriteria = new customerCriteria.CustomerName.Like("%Peter%");
Customer customer = CustomerFactory.LoadCustomerByCriteria(customerCriteria);
```

After executing the code, the first occurrence of "Peter" in the CustomerName column in the Customer table will be loaded to the object identified as customer.

- By List method

```
CustomerCriteria customerCriteria = new CustomerCriteria.CustomerName.Like("%Peter%");
Customer[] customer = CustomerFactory.ListCustomerByCriteria(customerCriteria);
```

After executing the code, all rows which contain "Peter" in the CustomerName column in the Customer table will be retrieved and stored in an array of Customer object.

Using POJO

Generating the persistence code with POJO, the persistence class will be generated only with the attributes and a pair of getter and setter methods for each attribute. As the persistence class is generated without the methods for database manipulation, the generated PersistentManager and PersistentSession classes are responsible for manipulating the database.

By also generating the corresponding Data Access Object (DAO) class for each ORM-Persistable class inside the defined package of orm package, you are allowed to use the generated DAO class as same as the DAO class generated from the DAO persistent API. For information on using the DAO class to manipulate the database, refer to the description in Using DAO section.

When using the DAO class generated with POJO persistent API, manipulate the database with the same persistent code of DAO class generated with DAO persistent API by replacing the DAO class with the DAO class inside the orm package.

When using the DAO class generated with POJO persistent API, manipulate the database with the same persistent code of DAO class generated with DAO persistent API by replacing the DAO class with the DAO class inside the orm package.

The following class diagram shows the relationship between the client program, persistent object, persistent session and persistent manager.

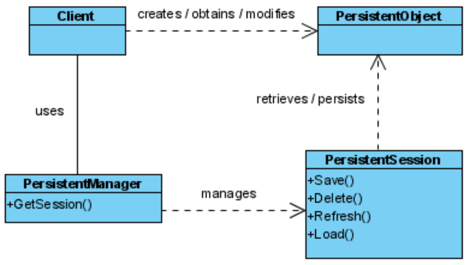


Figure 19.48 - Relation between Client and POJO Classes

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram.

Example:

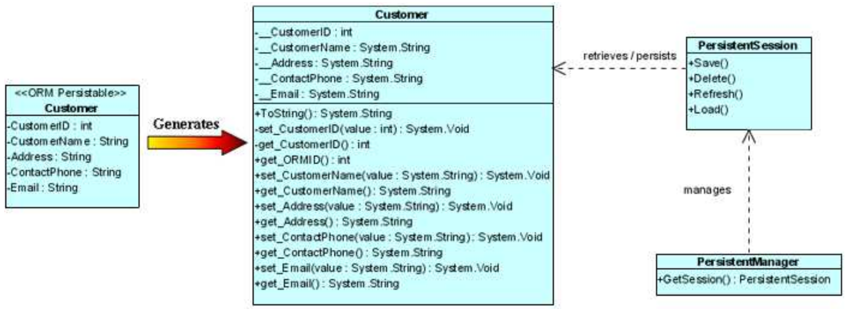

<<ORM Persistable>>
Customer
-CustomerID : int
-CustomerName : String
-Address : String
-ContactPhone : String
-Email : String
Generates
Customer
-__CustomerID : int
-__CustomerName : System.String
-__Address : System.String
-__ContactPhone : System.String
-__Email : System.String
+ToString() : System.String
-set_CustomerID(value : int) : System.Void
-get_CustomerID() : int
+get_ORMID() : int
+set_CustomerName(value : System.String) : System.Void
+get_CustomerName() : System.String
+set_Address(value : System.String) : System.Void
+get_Address() : System.String
+set_ContactPhone(value : System.String) : System.Void
+get_ContactPhone() : System.String
+set_Email(value : System.String) : System.Void
+get_Email() : System.String
retrieves / persists
PersistentSession
+Save()
+Delete()
+Refresh()
+Load()
manages
PersistentManager
+GetSession() : PersistentSession

Figure 19.49 - Class Diagram with POJO

From the above example, the Customer persistence class is generated with only the getter and setter methods for each attribute while the Customer persistent object is managed by the generated PersistentManager and PersistentSession.

Creating a Persistent Object

As a persistence class represents a table, an instance of a persistence class represents a record of the corresponding table. Creating a persistent object represents a creation of new record. With the support of the PersistentManager class, the newly created object can be saved as a new record to the database.

In order to insert a new row to the database table, implement the program with the following steps:

1. Create a new instance of the class by using the new operator.
2. Set the properties of the object by the persistence class.
3. Insert the object as a row to the database by the PersistentSession class.

Table shows the method summary of the PersistentSession class to be used for inserting a row of database.

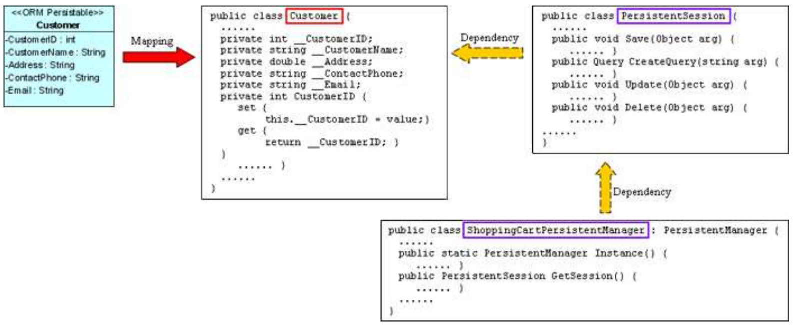
Class Type	Return Type	Method Name	Description
PersistentSession Class	Serializable	Save(Object value)	Insert the object as a row to the database table.

Table19.23

Remark:

1. **Object** represents the newly created instance of the persistence class to be added to the database.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
          this.__CustomerID = value;}
     get {
          return __CustomerID; }
  }
     ...... }
  ......
}
```

```
public class PersistentSession {
  ......
  public void Save(Object arg) {
     ...... }
  public Query CreateQuery(string arg) {
     ...... }
  public void Update(Object arg) {
     ...... }
  public void Delete(Object arg) {
     ...... }
......
}
```

```
public class ShoppingCartPersistentManager : PersistentManager {
  ......
  public static PersistentManager Instance() {
     ...... }
  public PersistentSession GetSession() {
     ...... }
  ......
}
```

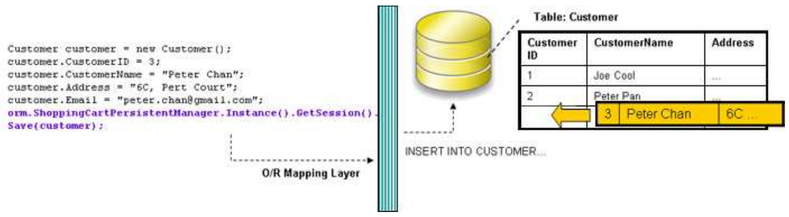
Figure 19.50 - Mapping with POJO

From the above example, an ORM-Persistable object model of Customer maps to an ORM-Persistable class with a pair of getter and setter methods for each attributes. With the PersistentManager class, the PersistentSession object can assist in inserting the instance as a row of the database table.

To insert a new Customer record to the table, Customer of the database, the following lines of code should be implemented.

```
Customer customer = new Customer();
customer.CustomerID = 3;
customer.CustomerName = "Peter Chan";
customer.Address = "6C, Pert Court";
customer.Email = "peter.chan@gmail.com";
orm.ShoppingCartPersistentManager.Instance().GetSession().Save(customer);
```

After executing these lines of code, a row of record is inserted to the database table.



```
Customer customer = new Customer();
customer.CustomerID = 3;
customer.CustomerName = "Peter Chan";
customer.Address = "6C, Pert Court";
customer.Email = "peter.chan@gmail.com";
orm.ShoppingCartPersistentManager.Instance().GetSession().
Save(customer);
```

Customer ID	CustomerName	Address
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	6C ...

Figure 19.51 - Code for add a record with using POJO

Loading a Persistent Object

As an instance of a persistence class represents a record of a table, a record retrieved from the table will be stored as an object. By specifying the query-like condition to the PersistentManager class, records can be retrieved from the database.

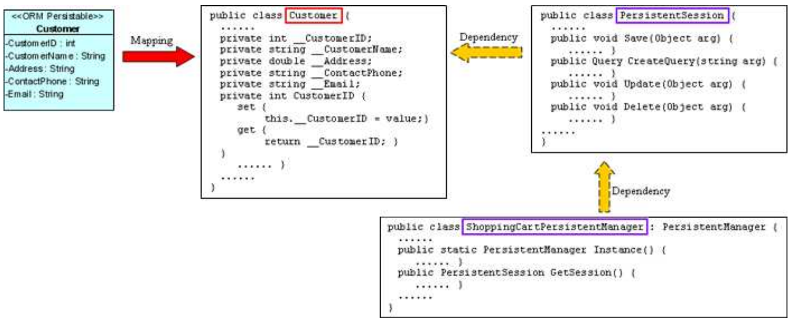
To retrieve a record, simply get the session by PersistentManager class and retrieve the records by defining the condition for query to the PersistentSession object and specify to return a single unique record.

Table shows the method summary of the PersistentSession class to be used for managing the retrieval of records

Class Type	Return Type	Method Name	Description
PersistentSession Class	Query	CreateQuery(string arg)	Retrieve the records matched with the user defined condition and ordered by a specified attribute.

Table 19.24

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
          this.__CustomerID = value;}
     get {
          return __CustomerID; }
  }
     ...... }
  ......
}
```

```
public class PersistentSession {
  ......
  public void Save(Object arg) {
     ...... }
  public Query CreateQuery(string arg) {
     ...... }
  public void Update(Object arg) {
     ...... }
  public void Delete(Object arg) {
     ...... }
......
}
```

```
public class ShoppingCartPersistentManager : PersistentManager {
  ......
  public static PersistentManager Instance() {
     ...... }
  public PersistentSession GetSession() {
     ...... }
  ......
}
```

Figure 19.52 - Mapping for create query

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable class with a pair of getter and setter methods for each attribute. The PersistentManager class gets the PersistentSession object which helps to query the database by specifying a user defined condition.

To retrieve a record from the Customer table, the following line of code should be implemented.

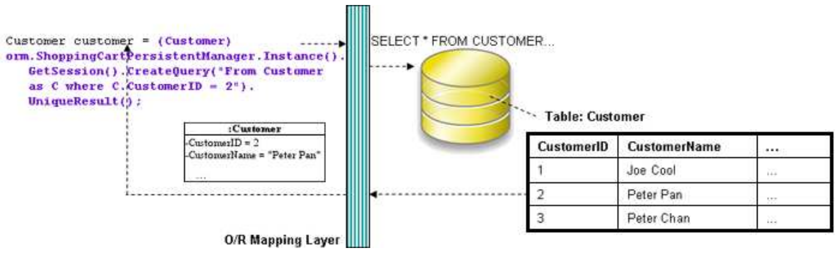
Loading an object by specifying a user defined condition with a passing value typed as "int":

```
Customer customer = (Customer)
orm.ShoppingCartPersistentManager.Instance().GetSession().CreateQuery("From Customer as C
where C.CustomerID = 2").UniqueResult();
```

Loading an object by specifying a user defined condition with a passing value typed as "String":

```
Customer customer = (Customer)
orm.ShoppingCartPersistentManager.Instance().GetSession().CreateQuery("From Customer as C
where C.CustomerName = 'Peter Pan'").UniqueResult();
```

After executing the code, a matched row is retrieved and loaded to a Customer object.



```
Customer customer = (Customer)
orm.ShoppingCartPersistentManager.Instance().
    GetSession().CreateQuery("From Customer
    as C where C.CustomerID = 2").
    UniqueResult();
```

Table: Customer

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 19.53 - Code for load a record by query

Updating a Persistent Object

As a record can be retrieved from the database table and loaded as an instance of the persistence class, updating a record can be achieved by setting a new value to the attribute by its setter method and saving the new value to the database.

In order to update a record, you have to retrieve the row which will be updated, and then set a new value to the property, finally update the database record. To update the record, implement the program with the following steps:

1. Retrieve a record from database table and load as an object by the PersistentManager class.
2. Set the updated value to the property of the object by the persistence class.
3. Save the updated record to the database by the PersistentManager class.

Table shows the method summary of the PersistentSession class to be used for updating a record.

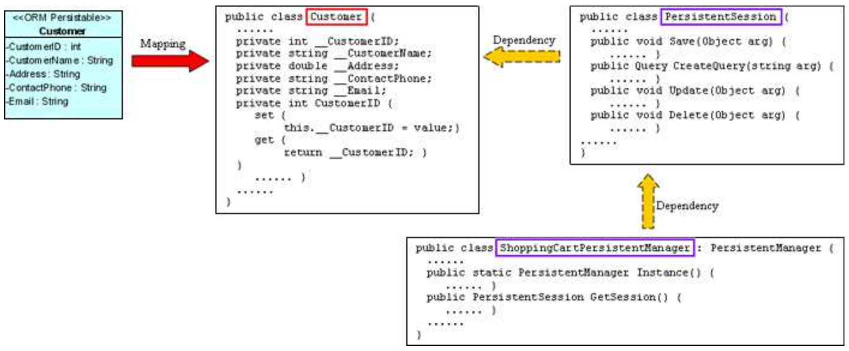
Class Type	Return Type	Method Name	Description
PersistentSession Class	void	Update(Object arg)	Update the value to database.

Table 19.25

Remark:

1. **Object** represents the instance of the persistence class to be updated to the corresponding database record.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
        this.__CustomerID = value;}
     get {
        return __CustomerID; }
  }
     ...... }
  ......
}
```

```
public class PersistentSession {
  ......
  public void Save(Object arg) {
    ...... }
  public Query CreateQuery(string arg) {
    ...... }
  public void Update(Object arg) {
    ...... }
  public void Delete(Object arg) {
    ...... }
......
}
```

```
public class ShoppingCartPersistentManager : PersistentManager {
  ......
  public static PersistentManager Instance() {
    ...... }
  public PersistentSession GetSession() {
    ...... }
  ......
}
```

Figure 19.54 - Mapping for update record

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable class with a pair of getter and setter methods for each attributes. The PersistentManager class gets the PersistentSession object which helps to save the updated record to database.

To update a Customer record, the following line of code should be implemented.

```
customer.CustomerName = "Peter Pang";
orm.ShoppingCartPersistentManager.Instance().GetSession().Update(customer);
```

After executing the above lines of code, the customer name is updated to "Peter Pang" in the database.

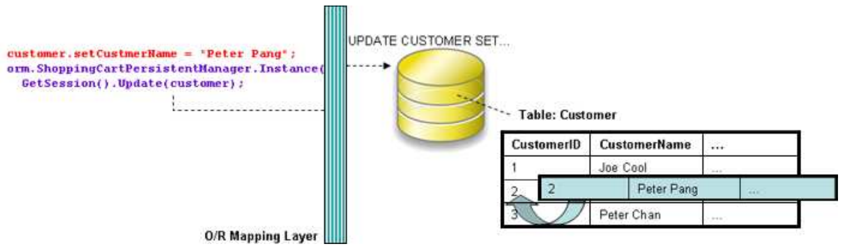

customer.setCustmerName = "Peter Pang";
orm.ShoppingCartPersistentManager.Instance(
GetSession().Update(customer);
UPDATE CUSTOMER SET...
Table: Customer
CustomerID
CustomerName
...
1
Joe Cool
2
Peter Pang
3
Peter Chan
O/R Mapping Layer

Figure 19.55 - Code for update record

Deleting a Persistent Object

As a record can be retrieved from the table and loaded as an object of the persistence class, the record can be deleted with the help of the PersistentManager class.

In order to delete a record, implement the program with the following steps:

1. Retrieve a record from database table and load as an object by the PersistentManager class.
2. Delete the retrieved record by the PersistentManager class.

Table shows the method summary of the PersistentSession class to be used for deleting a record from database.

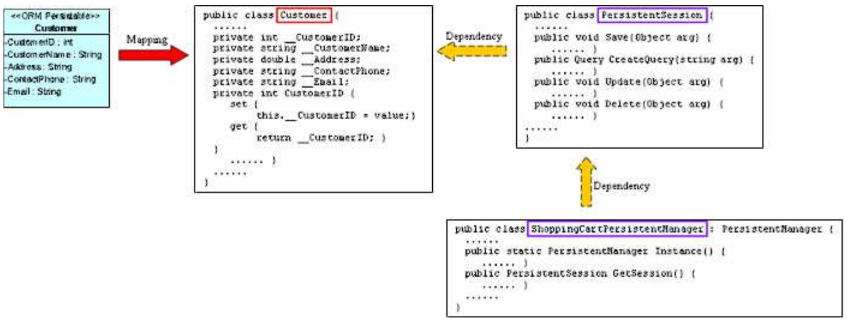
Class Type	Return Type	Method Name	Description
PersistentSession Class	void	Delete(Object arg)	Delete the current instance.

Table 19.26

Remark:

1. **Object** represents the instance of the persistence class corresponding to a record that will be deleted from the database.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
          this.__CustomerID = value;}
     get {
          return __CustomerID; }
  }
     ...... }
  ......
}
```

```
public class PersistentSession {
  ......
  public void Save(Object arg) {
     ...... }
  public Query CreateQuery(string arg) {
     ...... }
  public void Update(Object arg) {
     ...... }
  public void Delete(Object arg) {
     ...... }
......
}
```

```
public class ShoppingCartPersistentManager : PersistentManager {
  ......
  public static PersistentManager Instance() {
     ...... }
  public PersistentSession GetSession() {
     ...... }
  ......
}
```

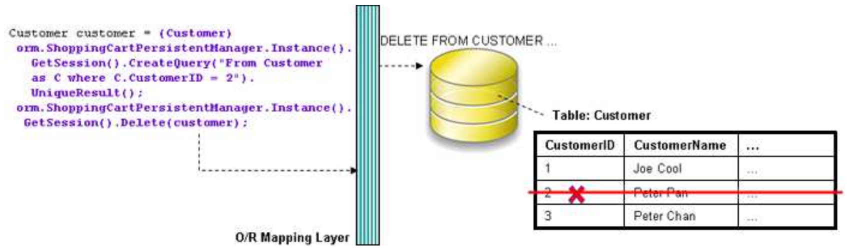
Figure 19.56 - Mapping for delete record

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable class with a pair of getter and setter methods for each attributes. The PersistentManager class gets the PersistentSession object which helps to delete the record from database.

To delete a Customer record, the following line of code should be implemented.

```
Customer customer = (Customer)
orm.ShoppingCartPersistentManager.Instance().GetSession().CreateQuery("From Customer as C
where C.CustomerID = 2").UniqueResult();
orm.ShoppingCartPersistentManager.Instance().GetSession().Delete(customer);
```

After executing the above lines of code, the specified customer record is deleted from the database.



```
Customer customer = (Customer)
 orm.ShoppingCartPersistentManager.Instance().
   GetSession().CreateQuery("From Customer
   as C where C.CustomerID = 2").
   UniqueResult();
 orm.ShoppingCartPersistentManager.Instance().
  GetSession().Delete(customer);
```

CustomerID	CustomerName	...
1	Joe Cool	...
~~2~~	~~Peter Pan~~	~~...~~
3	Peter Chan	...

Figure 19.57 - Code for delete record

Querying

It is well known that the database is enriched with information. In some cases, information may be retrieved from the database to assist another function of the application, such as retrieving the unit price of products for calculating the total amount for a purchase order.

As the persistence class is not capable of retrieving records from the database, the PersistentManager class makes use of the PersistentSession object to retrieve records from the database.

Loading a Collection of Persistent Objects

As the database table usually contains many records, you may want to query the tables by a specified condition. The PersistentManager and PersistentSession supports querying the database, the matched records will be retrieved and loaded to a list of objects.

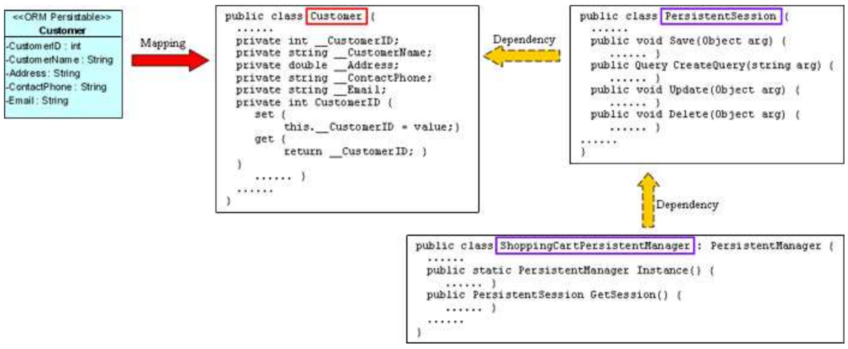
In order to retrieve records from the table, simply get the session by PersistentManager class and retrieve the records by defining the query condition to the PersistentSession object and specify to return a list of matched records.

Table shows the method summary of the PersistentSession class to be used for managing the retrieval of records.

Class Type	Return Type	Method Name	Description
PersistentSession Class	Query	CreateQuery(string arg)	Retrieve the records matched with the user defined condition and ordered by a specified attribute.

Table 19.27

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
          this.__CustomerID = value;}
     get {
          return __CustomerID; }
  }
     ...... }
  ......
}
```

```
public class PersistentSession {
  ......
  public void Save(Object arg) {
     ...... }
  public Query CreateQuery(string arg) {
     ...... }
  public void Update(Object arg) {
     ...... }
  public void Delete(Object arg) {
     ...... }
......
}
```

```
public class ShoppingCartPersistentManager : PersistentManager {
  ......
  public static PersistentManager Instance() {
     ...... }
  public PersistentSession GetSession() {
     ...... }
  ......
}
```

Figure 19.58 - Mapping create query methods

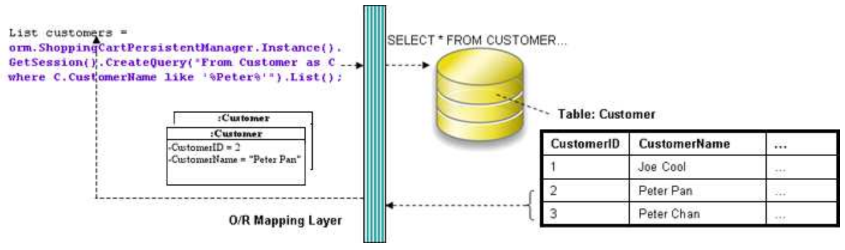
From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable class generated with a pair of getter and setter methods for each attribute. The PersistentManager class gets the PersistentSession object which helps to query the database by giving a user defined condition.

To retrieve records from the Customer table, the following line of code should be implemented.

Loading objects by specifying a user defined condition with a passing value typed as "String":

```
List customers = orm.ShoppingCartPersistentManager.Instance().GetSession().CreateQuery("From
Customer as C where C.CustomerName like '%Peter'").List();
```

After executing the code, the matched rows are retrieved and loaded to a list containing the Customer objects.



```
List customers =
orm.ShoppingCartPersistentManager.Instance().
GetSession().CreateQuery("From Customer as C
where C.CustomerName like '%Peter%'").List();
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

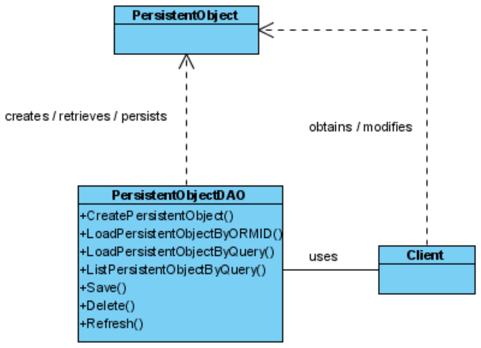
Figure 19.59 - Code for retrieve a list of records

Using DAO

Generating the persistence code with data access object (DAO), not only the persistence class will be generated, but also the corresponding DAO class for each ORM-Persistable class.

The generated persistence class using DAO persistent API is as same as that using POJO; that is, the persistence class contains attributes and a pair of getter and setter methods for each attribute only. Instead of using the persistence class to manipulate the database, the DAO class supports creating a new instance for the addition of a new record to the database, and retrieving record(s) from the database, updating and deleting the records.

The following class diagram shows the relationship between the client program, persistent object and DAO object.



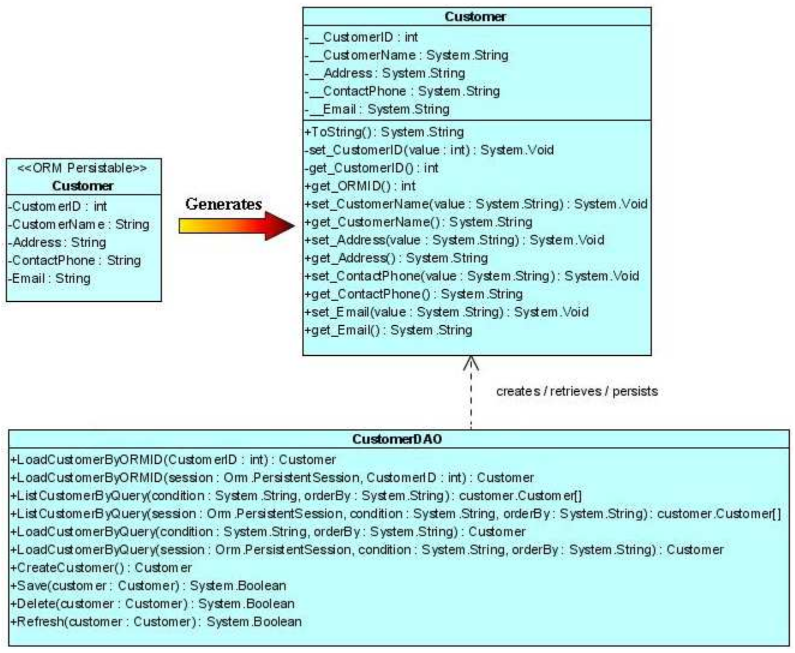
PersistentObject
creates / retrieves / persists
obtains / modifies
PersistentObjectDAO
+CreatePersistentObject()
+LoadPersistentObjectByORMID()
+LoadPersistentObjectByQuery()
+ListPersistentObjectByQuery()
+Save()
+Delete()
+Refresh()
uses
Client

Figure 19.60 - Class Diagram for DAO

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram while the PersistentObjectDAO refers to the generated DAO class of the ORM-Persistable class. For example, the CustomerDAO class persists with the Customer persistent object and the database.

In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram while the PersistentObjectDAO refers to the generated DAO class of the ORM-Persistable class. For example, the CustomerDAO class persists with the Customer persistent object and the database.

Example:



<<ORM Persistable>>
Customer
-CustomerID : int
-CustomerName : String
-Address : String
-ContactPhone : String
-Email : String
Generates
Customer
-__CustomerID : int
-__CustomerName : System.String
-__Address : System.String
-__ContactPhone : System.String
-__Email : System.String
+ToString() : System.String
-set_CustomerID(value : int) : System.Void
-get_CustomerID() : int
+get_ORMID() : int
+set_CustomerName(value : System.String) : System.Void
+get_CustomerName() : System.String
+set_Address(value : System.String) : System.Void
+get_Address() : System.String
+set_ContactPhone(value : System.String) : System.Void
+get_ContactPhone() : System.String
+set_Email(value : System.String) : System.Void
+get_Email() : System.String
creates / retrieves / persists
CustomerDAO
+LoadCustomerByORMID(CustomerID : int) : Customer
+LoadCustomerByORMID(session : Orm.PersistentSession, CustomerID : int) : Customer
+ListCustomerByQuery(condition : System.String, orderBy : System.String) : customer.Customer[]
+ListCustomerByQuery(session : Orm.PersistentSession, condition : System.String, orderBy : System.String) : customer.Customer[]
+LoadCustomerByQuery(condition : System.String, orderBy : System.String) : Customer
+LoadCustomerByQuery(session : Orm.PersistentSession, condition : System.String, orderBy : System.String) : Customer
+CreateCustomer() : Customer
+Save(customer : Customer) : System.Boolean
+Delete(customer : Customer) : System.Boolean
+Refresh(customer : Customer) : System.Boolean

Figure 19.61 - Class Diagram for using DAO

From the above example, the CustomerDAO class supports the creation of Customer persistent object, the retrieval of Customer records from the Customer table while the Customer persistence class supports the update and deletion of customer record.

Creating a Persistent Object

An instance of a persistence class represents a record of the corresponding table, creating a persistent object supports the addition of new record to the table.

In order to add a new row to the database table, implement the program with the following steps:

1. Create a new instance of the class by the DAO class.
2. Set the properties of the object by the persistence class.
3. Insert the object as a row to the database by the DAO class.

Table shows the method summary of the DAO class to be used for inserting a new row in database

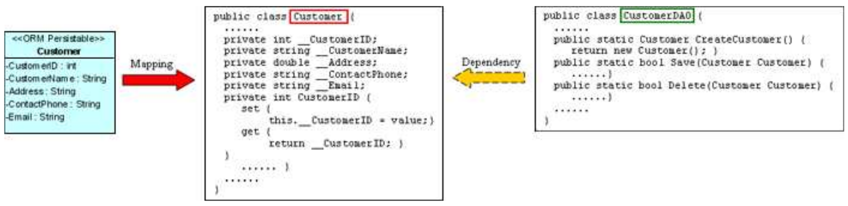
Class Type	Return Type	Method Name	Description
DAO Class	Class	CreateClass()	Create a new instance of the class.
DAO Class	bool	Save(Object value)	Insert the object as a row to the database table.

Table 19.28

Remark:

1. **Class** should be replaced by the name of the generated persistence class.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
          this.__CustomerID = value;}
     get {
          return __CustomerID; }
  }
     ...... }
  ......
}
```

```
public class CustomerDAO {
  ......
  public static Customer CreateCustomer() {
     return new Customer(); }
  public static bool Save(Customer Customer) {
     ......}
  public static bool Delete(Customer Customer) {
     ......}
  ......
}
```

Figure 19.62 - Mapping with DAO

From the above example, an ORM-Persistable object model of Customer maps to an ORM-Persistable lass with methods for setting the properties. An ORM-Persistable DAO class is generated supporting the creation of persistent object and adding it into the database.

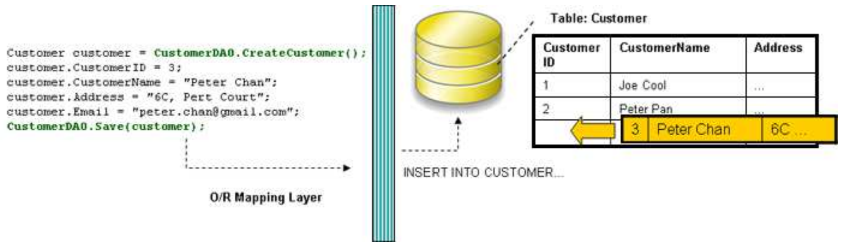
To add a new Customer record to the table, Customer of the database, the following lines of code should be implemented.

```
Customer customer = CustomerDAO.CreateCustomer();
customer.CustomerID = 3;
customer.CustomerName = "Peter Chan";
customer.Address = "6C, Pert Court";
customer.Email = "peter.chan@gmail.com";
CustomerDAO.Save(customer);
```

After executing these lines of code, a row of record is inserted to the database table.

An alternative way to create a new instance of the class is using the new operator:
Class c = new Class();

From the above example, Customer customer = CustomerDAO.CreateCustomer() can be replaced by Customer customer = new Customer() to create a Customer object.



```
Customer customer = CustomerDAO.CreateCustomer();
customer.CustomerID = 3;
customer.CustomerName = "Peter Chan";
customer.Address = "6C, Pert Court";
customer.Email = "peter.chan@gmail.com";
CustomerDAO.Save(customer);
```

Table: Customer

Customer ID	CustomerName	Address
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	6C ...

Figure 19.63 - Code for create records by using DAO

Loading a Persistent Object

As an instance of a persistence class represents a record of the table, a record retrieved from the table will be stored as an object.

In order to retrieve a record from the database table, you have to specify the condition for finding the record. To retrieve a record, implement the program with the following steps:

1. Specify the condition for searching the record by the DAO class.
2. Load the retrieved record to an object.

Table shows the method summary of the DAO class to be used for retrieving record from database.

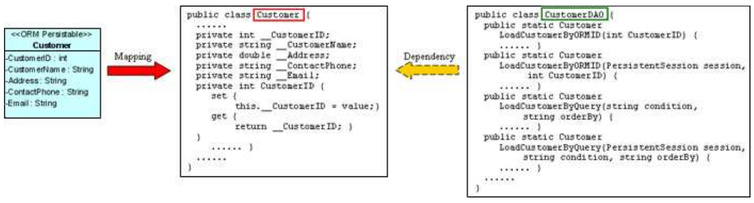
Class Type	Return Type	Method Name	Description
DAO Class	Class	LoadClassByORMID(DataType PrimaryKey)	Retrieve a record matching with the specified value of primary key.
DAO Class	Class	LoadClassByORMID(PersistentSession session, DataType PrimaryKey)	Retrieve a record matching with the specified value of primary key and specified session.
DAO Class	Class	LoadClassByQuery(string condition, string orderBy)	Retrieve the first record matching the user defined condition while the matched records are ordered by a specified attribute.
DAO Class	Class	LoadClassByQuery(PersistentSession session, string condition, string orderBy)	Retrieve the first record matching the user defined condition and specified session while the matched records are ordered by a specified attribute.

Table 19.29

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
        this.__CustomerID = value;}
     get {
        return __CustomerID; }
  }
     ...... }
  ......
}
```

```
public class CustomerDAO {
  public static Customer
     LoadCustomerByORMID(int CustomerID) {
     ...... }
  public static Customer
     LoadCustomerByORMID(PersistentSession session,
        int CustomerID) {
     ...... }
  public static Customer
     LoadCustomerByQuery(string condition,
        string orderBy) {
     ...... }
  public static Customer
     LoadCustomerByQuery(PersistentSession session,
        string condition, string orderBy) {
     ...... }
  ......
}
```

Figure 19.64 - Mapping load and list methods in DAO

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable .NET class. A DAO class for the customer class is generated with methods for retrieving a matched record.

To retrieve a record from the Customer table, the following line of code should be implemented.

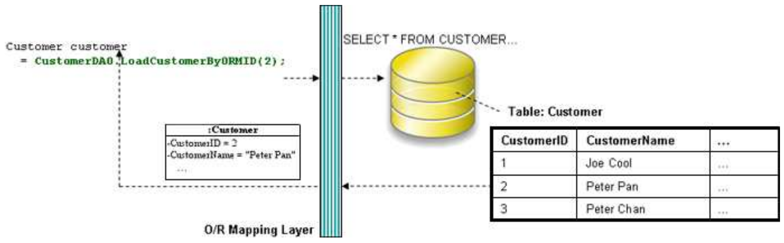
Loading an object by passing the value of primary key:

```
Customer customer = CustomerDAO.LoadCustomerByORMID(2);
```

Loading an object by specifying a user defined condition:

```
Customer customer = CustomerDAO.LoadCustomerByQuery("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

After executing the code, a matched row is retrieved and loaded to a Customer object.



Customer customer
= CustomerDAO.LoadCustomerByORMID(2);
SELECT * FROM CUSTOMER...
Table: Customer
:Customer
-CustomerID = 2
-CustomerName = "Peter Pan"
...
O/R Mapping Layer

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 19.65 - Code for load a records from database

Updating a Persistent Object

The DAO class not only supports the retrieval of record, but also the update on the record with the assistance of the setter method of the persistence class.

In order to update a record, you have to retrieve the row to be updated first, and then set a new value to the property, and finally save the updated record to the database. To update the record, implement the program with the following steps:

1. Retrieve a record from database table and load as an object by the DAO class.
2. Set the updated value to the property of the object by the persistence class.
3. Save the updated record to the database by the DAO class.

Table shows the method summary of the DAO class to be used for updating a record.

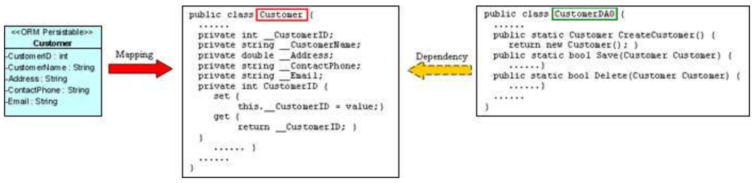
Class Type	Return Type	Method Name	Description
DAO Class	bool	Save(Object value)	Update the value to database.

Table 19.30

Remark:

1. **Object** represents the instance of the persistence class to be updated to the corresponding database record.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
        this.__CustomerID = value;}
     get {
        return __CustomerID; }
  }
     ...... }
  ......
}
```

```
public class CustomerDAO {
  ......
  public static Customer CreateCustomer() {
     return new Customer(); }
  public static bool Save(Customer Customer) {
     ......}
  public static bool Delete(Customer Customer) {
     ......}
  ......
}
```

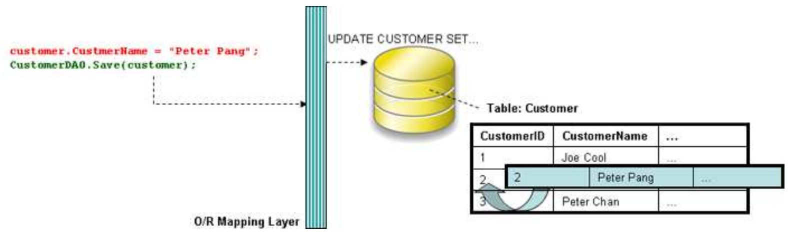
Figure 19.66 - Mapping for update record

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable .NET class with a pair of getter and setter methods for each attributes. The generated DAO class provides method for updating the record.

To update a Customer record, the following lines of code should be implemented.

```
customer.CustomerName = "Peter Pang";
CustomerDAO.Save(customer);
```

After executing the above lines of code, the customer name is updated to "Peter Pang" in the database.



```
customer.CustmerName = "Peter Pang";
CustomerDAO.Save(customer);
```

Figure 19.67 - Code for update record

Deleting a Persistent Object

The DAO class also supports deleting a record from the database. In order to delete a record, implement the program with the following steps:

1. Retrieve a record from database table and load as an object by the DAO class.
2. Delete the retrieved record by the DAO class.

Table shows the method summary of the DAO class to be used for deleting a record from database.

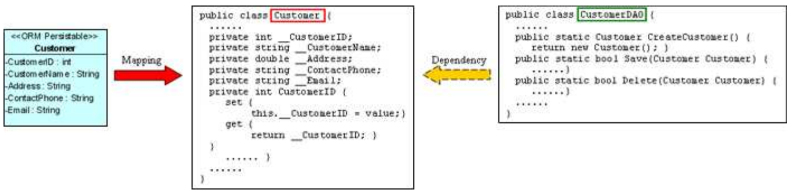
Class Type	Return Type	Method Name	Description
DAO Class	bool	Delete(Class value)	Delete the current instance.

Table 19.31

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
          this.__CustomerID = value;}
     get {
          return __CustomerID; }
  }
     ...... }
  ......
}
```

```
public class CustomerDAO {
  ......
  public static Customer CreateCustomer() {
     return new Customer(); }
  public static bool Save(Customer Customer) {
     ......}
  public static bool Delete(Customer Customer) {
     ......}
  ......
}
```

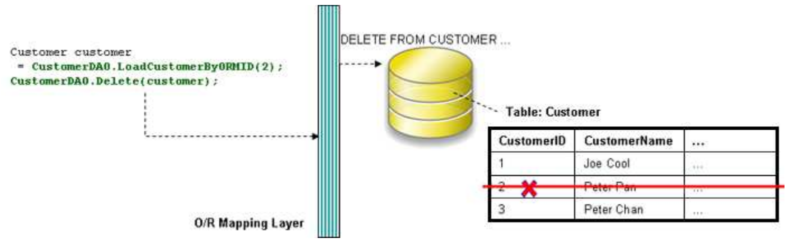
Figure 19.68 - Mapping for delete record

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable .NET class generated with the methods for deleting the specified record from the database.

To delete a Customer record, the following lines of code should be implemented.

```
Customer customer = CustomerDAO.LoadCustomerByORMID(2);
customer.Delete();
```

After executing the above code, the specified customer record is deleted from the database.



```
Customer customer
 = CustomerDAO.LoadCustomerByORMID(2);
CustomerDAO.Delete(customer);
```

Figure 19.69 - Code for delete record by using DAO

Querying

It is well known that the database is enriched with information. In some cases, information may be retrieved from the database to assist another function of the application, such as retrieving the unit price of products for calculating the total amount for a purchase order.

As the DAO class is capable of querying the database, records can be retrieved by specifying the searching condition.

Loading a Collection of Persistent Objects

As the database table usually contains many records, you may want to query the tables by a specified condition. The generated DAO class supports querying the database, the matched records will be retrieved and loaded as an object array.

In order to retrieve records from the table, you have to specify the condition for querying. To retrieve a number of records, implement the program with the following steps:

1. Specify the condition for searching the record by the DAO class.
2. Load the retrieved records as an object array by the DAO class.

Table shows the method summary of the DAO class to be used for retrieving record from database table

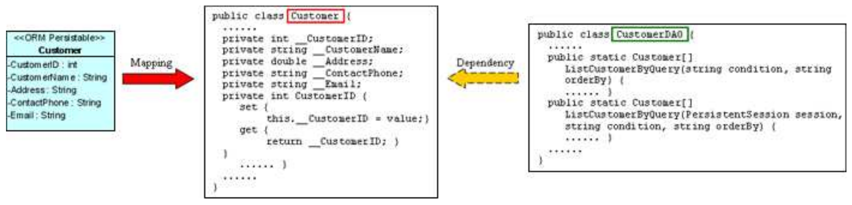
Class Type	Return Type	Method Name	Description
DAO Class	Class[]	ListClassByQuery(string condition, string orderBy)	Retrieve the records matched with the user defined condition and ordered by a specified attribute.
DAO Class	Class[]	ListClassByQuery(PersistentSession session, string condition, string orderBy)	Retrieve the records matched with the user defined condition and specified session and ordered by a specified attribute.

Table 19.32

Remark:

1. Class should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
          this.__CustomerID = value;}
     get {
          return __CustomerID; }
  }
     ...... }
  ......
}
```

```
public class CustomerDAO {
  ......
  public static Customer[]
     ListCustomerByQuery(string condition, string
     orderBy) {
     ...... }
  public static Customer[]
     ListCustomerByQuery(PersistentSession session,
     string condition, string orderBy) {
     ...... }
  ......
}
```

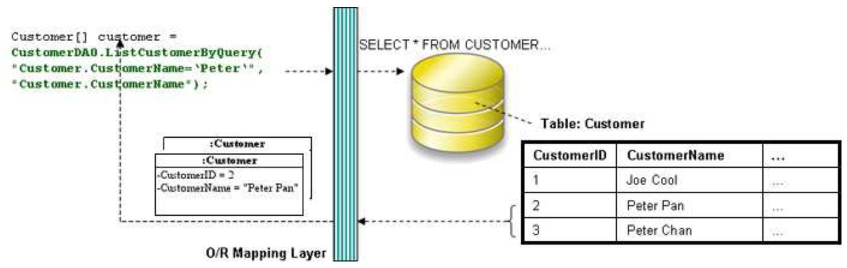
Figure 19.70 - Mapping for list method in DAO

From the above example, an ORM-Persistable object model, Customer maps to an ORM-Persistable .NET class. The DAO class for the customer is generated with methods for retrieving records.

To retrieve records from the Customer table, the following line of code should be implemented.

```
Customer[] customer = CustomerDAO.ListCustomerByQuery("Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

After executing the code, the matched rows are retrieved and loaded to an object array of Customer.



```
Customer[] customer =
CustomerDAO.ListCustomerByQuery(
"Customer.CustomerName='Peter'",
"Customer.CustomerName");
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 19.71 - Code for retrieve a list of records by using DAO

Using ORM Qualifier

ORM Qualifier is an additional feature which allows you to specify the extra data retrieval rules apart from the system pre-defined rules. The ORM Qualifier can be defined in order to generate persistence code. Refer to Defining ORM Qualifier in the Object Model chapter for more information.

By defining the ORM Qualifier in a class, the persistence DAO class will be generated with additional data retrieval methods, load and list methods.

Table shows the method summary generated to the DAO class by defining the ORM Qualifier.

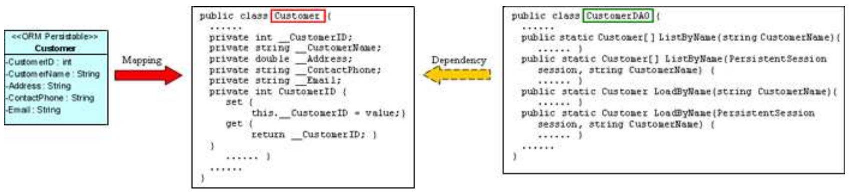
Class Type	Return Type	Method Name	Description
DAO Class	Class	LoadByORMQualifier(DataType attribute)	Retrieve the first record that matches the specified value with the attribute defined in the ORM Qualifier.
DAO Class	Class	LoadByORMQualifier (PersistentSession session, DataType attribute)	Retrieve the first record that matches the specified value with the attribute defined in the ORM Qualifier and specified session.
DAO Class	Class[]	ListByORMQualifier (DataType attribute)	Retrieve the records matched that matches the specified value with the attribute defined in the ORM Qualifier.
DAO Class	Class[]	ListByORMQualifier (PersistentSession session, DataType attribute)	Retrieve the records that match the specified value with the attribute defined in the ORM Qualifier and specified session.

Table 19.33

Remark:

1. **Class** should be replaced by the name of the persistence class.
2. **ORMQualifier** should be replaced by the Name defined in the ORM Qualifier.
3. **DataType** should be replaced by the data type of the attribute which associated with the ORM Qualifier.
4. **attribute** is the specified value to be used for querying the table.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
          this.__CustomerID = value;}
     get {
          return __CustomerID; }
  }
     ...... }
  ......
}
```

```
public class CustomerDAO {
  ......
  public static Customer[] ListByName(string CustomerName){
     ...... }
  public static Customer[] ListByName(PersistentSession
     session, string CustomerName) {
     ...... }
  public static Customer LoadByName(string CustomerName){
     ...... }
  public static Customer LoadByName(PersistentSession
     session, string CustomerName) {
     ...... }
  ......
}
```

Figure 19.72 - Mapping load and list methods for qualifier

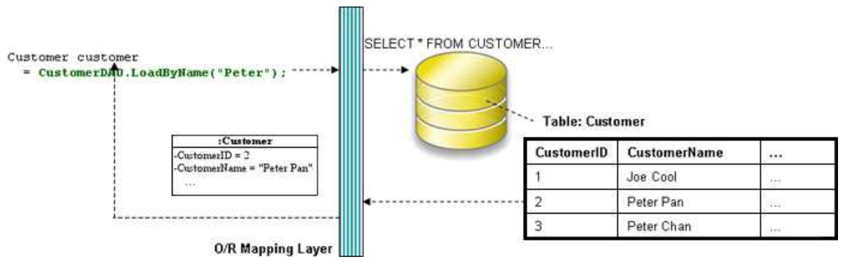
In the above example, a customer object model is defined with an ORM Qualifier named as Name and qualified with the attribute, CustomerName.

To query the Customer table with the ORM Qualifier in one of the two way

- By Load method

```
Customer customer = CustomerDAO.LoadByName("Peter");
```

After executing the code, the first occurrence of "Peter" in the CustomerName column in the Customer table will be loaded to the object identified as customer.



```
Customer customer
  = CustomerDAO.LoadByName("Peter");
```

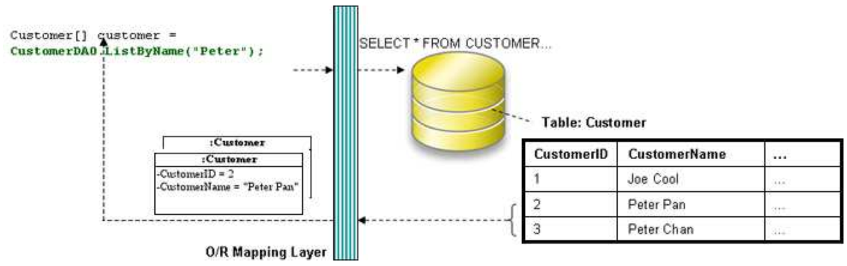
CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 19.73 - Code for loading a records by qualifier

- By List method

```
Customer[] customer = CustomerDAO.ListByName("Peter");
```

After executing the code, all rows which contain "Peter" in the CustomerName column in the Customer table will be retrieved and stored in an array of Customer object.



```
Customer[] customer =
CustomerDAO.ListByName("Peter");
```

CustomerID	CustomerName	...
1	Joe Cool	...
2	Peter Pan	...
3	Peter Chan	...

Figure 19.74 - Code for retrieving a list of record by qualifier

Using Criteria Class

When generating the persistence class for each ORM-Persistable class defined in the object model, the corresponding criteria class can also be generated. Criteria class is a helper class which provides an additional way to specify the condition to retrieve records from the database. Refer to Using Criteria Class section for more detailed information on how to specify the conditions to query the database by the criteria class.

You can get the retrieved records from the criteria class in one of the two ways:

- Use the retrieval methods of the criteria class.

  For information on the retrieval methods provided by the criteria class, refer to the description of Loading Retrieved Records section.

- Use the retrieval by criteria methods of the DAO class.

Table shows the method summary generated to the persistence class to retrieve records from the criteria class.

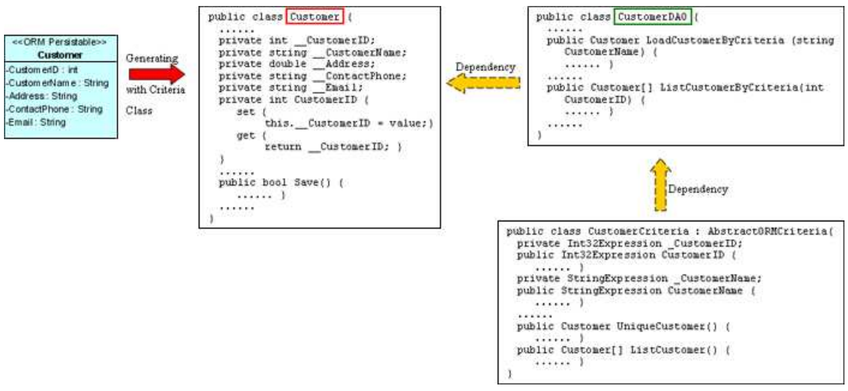
Class Type	Return Type	Method Name	Description
Factory Class	Class	LoadClassByCriteria(ClassCriteria value)	Retrieve the single record that matches the specified conditions applied to the criteria class.
Factory Class	Class[]	ListClassByCriteria(ClassCriteria value)	Retrieve the records that match the specified conditions applied to the criteria class.

Table 19.34

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class Customer {
  ......
  private int __CustomerID;
  private string __CustomerName;
  private double __Address;
  private string __ContactPhone;
  private string __Email;
  private int CustomerID {
     set {
          this.__CustomerID = value;}
     get {
          return __CustomerID; }
  }
  ......
  public bool Save() {
     ...... }
  ......
}
```

```
public class CustomerDAO {
  ......
  public Customer LoadCustomerByCriteria (string
     CustomerName) {
     ...... }
  ......
  public Customer[] ListCustomerByCriteria(int
     CustomerID) {
     ...... }
  ......
}
```

```
public class CustomerCriteria : AbstractORMCriteria{
  private Int32Expression _CustomerID;
  public Int32Expression CustomerID {
     ...... }
  private StringExpression _CustomerName;
  public StringExpression CustomerName {
     ...... }
  ......
  public Customer UniqueCustomer() {
     ...... }
  public Customer[] ListCustomer() {
     ...... }
}
```

Figure 19.75 - Mapping with criteria class

To retrieve records from the Criteria Class in one of the two ways:

- By Load method

  ```
  CustomerCriteria customerCriteria = new customerCriteria.CustomerName.Like("%Peter%");
  Customer customer = CustomerDAO.LoadCustomerByCriteria(customerCriteria);
  ```

  After executing the code, the first occurrence of "Peter" in the CustomerName column in the Customer table will be loaded to the object identified as customer.

- By List method

  ```
  CustomerCriteria customerCriteria = new CustomerCriteria.CustomerName.Like("%Peter%");
  Customer[] customer = CustomerDAO.ListCustomerByCriteria(customerCriteria);
  ```

  After executing the code, all rows which contain "Peter" in the CustomerName column in the Customer table will be retrieved and stored in an array of Customer object.

Using Criteria Class

As a database is normally enriched with information, there are two ways to retrieve records from the database. Firstly, the list and load methods to the persistence code are generated which retrieve matched records from the database with respect to the user defined condition. Secondly, a criteria class can be generated for each ORM-Persistable class which supports searching the records from the corresponding table.

By using the criteria class, it supports querying the database with multiple criteria. To generate the criteria class for query, simply check the Generate Criteria option before the generation of code. For more information on the setting of code generation, refer to Configuring Code Generation Setting for C# in the Implementation chapter.

Figure 19.76 - Generate criteria checkbox

Having selected the Generate Criteria option for the code generation, a criteria class is generated in addition to the classes generated with respect to the persistent API. The generated criteria class is named as "ClassCriteria" in which Class is the name of the ORM Persistable class accordingly. The criteria class is generated with attributes, which are defined in the object model, with type of one of Expression with respect to the type of attribute defined in the object model and two operations for specifying the type of record retrieval.

In order to query the database by the criteria class, implement the program with the following steps:

1. Create an instance of the criteria class.
2. Apply restriction to the property of the class which specifies the condition for query.
3. Apply ordering to the property if it is necessary.
4. Set the range of the number of records to be retrieved if it is necessary.
5. Load the retrieved record(s) to an object or array.

Applying Restriction to Property

To apply the restriction to the property, implement with the following code template:

```
criteria.property.expression(parameter);
```

where criteria is the instance of the criteria class; property is the property of the criteria;
expression is the expression to be applied on the property; parameter is the parameter(s) of the expression.

Table shows the expression to be used for specifying the condition for query.

Expression	Description
Eq(value)	The value of the property is equal to the specified value.
Ne(value)	The value of the property is not equal to the specified value.
Gt(value)	The value of the property is greater than to the specified value.
Ge(value)	The value of the property is greater than or equal to the specified value.
Lt(value)	The value of the property is less than the specified value.
Le(value)	The value of the property is less than or equal to the specified value.

IsEmpty()	The value of the property is empty.
IsNotEmpty()	The value of the property is not empty.
IsNull()	The value of the property is NULL.
IsNotNull()	The value of the property is not NULL.
In(values)	The value of the property contains the specified values in the array.
Between(value1, value2)	The value of the property is between the two specified values, value1 and value2.
Like(value)	The value of the property matches the string pattern of value; use % in value for wildcard.
Ilike(value)	The value of the property matches the string pattern of value, ignoring case differences.

Table 19.35

There are two types of ordering to sort the retrieved records, that is, ascending and descending order.

Sorting Retrieved Records

There are two types of ordering to sort the retrieved records, that is, ascending and descending order.

To sort the retrieved records with respect to the property, implement the following code template:

```
criteria.property.Order(ascending_order);
```

where the value of ascending_order is either true or false. True refers to sort the property in ascending order while false refers to sort the property in descending order.

Setting the Number of Retrieved Records

To set the range of the number of records to be retrieved by using one of the two methods listed in the following table.

Method Name	Description
setFirstResult(int i)	Retrieve the i-th record from the results as the first result.
setMaxResult(int i)	Set the maximum number of retrieved records by specified value, i.

Table 19.36

Loading Retrieved Records

To load the retrieved record(s) to an object or array, use one of the two methods listed below.

Table shows the method summary of the criteria class to be used for retrieving records from database.

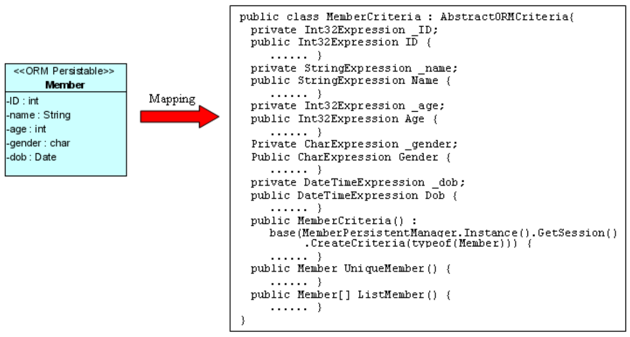
Return Type	Method Name	Description
Class	UniqueClass()	Retrieve the first record matching the specified condition(s) for the criteria.
Class[]	ListClass()	Retrieve the records matched with the specified condition(s) for the criteria.

Table 19.37

Remark:

1. **Class** should be replaced by the name of the persistence class.

Example:



```
public class MemberCriteria : AbstractORMCriteria{
  private Int32Expression _ID;
  public Int32Expression ID {
     ...... }
  private StringExpression _name;
  public StringExpression Name {
     ...... }
  private Int32Expression _age;
  public Int32Expression Age {
     ...... }
  Private CharExpression _gender;
  Public CharExpression Gender {
     ...... }
  private DateTimeExpression _dob;
  public DateTimeExpression Dob {
     ...... }
  public MemberCriteria() :
     base(MemberPersistentManager.Instance().GetSession()
          .CreateCriteria(typeof(Member))) {
     ...... }
  public Member UniqueMember() {
     ...... }
  public Member[] ListMember() {
     ...... }
}
```

Figure 19.77 - Mapping with Criteria Class

The generated attributes is typed as one type of Expression with respect to the type of attribute defined in the object model. The attribute defined as primary key will not be generated as an attribute of the criteria class.

Here is an object model of a Member class. After performing the generation of persistent code, an additional .NET file is generated, named "MemberCriteria.cs".

The following examples illustrate how to query the Member table using the MemberCriteria class:

- By specifying one criterion

  Retrieving a member record whose name is "John":

```
MemberCriteria criteria = new MemberCriteria();
criteria.Name.Eq("John");
Member member= criteria.UniqueMember();
Console.WriteLine(member);
```

  Retrieving all member records whose date of birth is between 1/1/1970 and 31/12/1985:

```
MemberCriteria criteria = new MemberCriteria();
criteria.Dob.Between(new GregorianCalendar(1970, 1, 1).GetTime(), new
GregorianCalendar(1985, 12, 31).GetTime());
Member[] members = criteria.ListMember();
foreach (int i in members) {
       Console.WriteLine(members[i]);
}
```

- By specifying more than one criteria

  Retrieving all male member records whose age is between 18 and 24, ordering by the name:

```
MemberCriteria criteria = new MemberCriteria();
criteria.Age.In(new int[] {18, 24});
criteria.Gender.Eq('M');
criteria.Name.Order(true); //true = ascending order; flase = descending order
Member[] members = criteria.ListMember();
foreach (int i in members) {
       Console.WriteLine(members[i]);
}
```

  Retrieving all member records whose name starts with "Chan" and age is greater than 30:

```
MemberCriteria criteria = new MemberCriteria();
criteria.Name.Like("Chan %");
criteria.Age.Gt(30);
Member[] members = criteria.ListMember();
foreach (int i in members) {
       Console.WriteLine(members[i]);
}
```

Using Transactions

A transaction is a sequence of operation. If the operation is performed successfully, the transaction is committed permanently. On the contrary, if the operation is performed unsuccessfully, the transaction is rolled back to the state of the last successfully committed transaction. Since database modifications must follow the "all or nothing" rule, transaction must be manipulated for every access to the database.

To create a transaction for atomicity by the following line of code:

```
PersistentTransaction t = PersistentManager.Instance().GetSession().BeginTransaction();
```

To commit a transaction by the following line of code:

```
t.Commit();
```

To rollback a transaction if there is any exception in the program by the following line of code:

```
t.RollBack();
```

Using ORM Implementation

The generated persistent code exposes the ability of the .NET class to access the database including the basic operations for add, update, delete and search. As the generated persistent code only provides basic operations for manipulating the persistent data, you may find it is insufficient and want to add extra logic to it. However, modifying the generated persistent code may cause your developing program malfunctioned. It is strongly recommended to add the extra logic by using ORM Implementation class to manipulate the persistent data.

An implementation class can be added to the object model. When generating persistent code, the implementation class will also be generated. You are allowed to implement the operation in the generated implementation class. As if the content of an implementation class has been modified, it will not be overwritten when re-generating the persistent code from the object model.

Inserting an ORM Implementation Class

1. Select a class stereotyped as ORM-Persistable that you want to add extra logic for manipulating the persistent data.

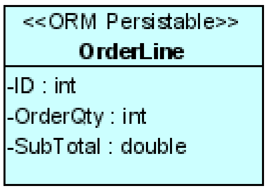


Figure 19.78 - ORM Persistable Class

2. Add a new operation to the class by right clicking the class element, selecting **Add** > **Operation**.

   A new operation is added, type the operation in the form of "operation_name(parameter_name: type) : return_type".

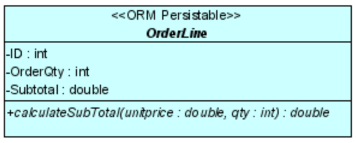

<<ORM Persistable>>
OrderLine
-ID : int
-OrderQty : int
-Subtotal : double
+calculateSubTotal(unitprice : double, qty : int) : double

Figure 19.79 - ORM Persistable Class with operation

A new operation can be added by using the keyboard shortcut - ***Alt + Shift + O***. You can also edit the operation by double-clicking the operation name or pressing the ***F2*** button.

3. Drag the resource of **Create ORM Implementation Class** to the diagram pane.

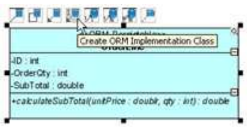

Create ORM Implementation Class
-ID : int
-OrderQty : int
-SubTotal : double
+calculateSubTotal(unitPrice : double, qty : int) : double

*Figure 19.80 - **Create ORM Implementation Class** resource-centric*

A new implementation class is created and connected to the source class with generalization stereotyped as ORM Implementation. The source class becomes an abstract class with an abstract operation.

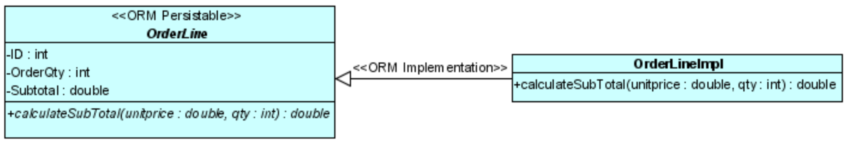

<<ORM Persistable>>
OrderLine
-ID : int
-OrderQty : int
-Subtotal : double
+calculateSubTotal(unitprice : double, qty : int) : double
<<ORM Implementation>>
OrderLineImpl
+calculateSubTotal(unitprice : double, qty : int) : double

Figure 19.81 - ORM Persistable and Implementation Class

After transforming the object model to persistent code, an implementation class will be generated. The generated code for the implementation class is shown below:

```
using System;
namespace shoppingcart {
public class OrderLineIml : OrderLine {
        public OrderLineImpl() : base() {
        }

        public override double calculateSubTotal(double unitprice, int qty){
                // TODO: Implement Method
                throw new System.Exception("Not implemented");
        }
}
}
```

The generated implementation class is the same no matter which persistent API is selected for code generation.

You can implement the logic to the method in the generated implementation class.

Code Sample

Here is a sample demonstrating the usage of the implementation class from the above object model.

The implementation class is implemented with the following lines of code:

```
using System;
namespace shoppingcart {
public class OrderLineIml : OrderLine {
        public OrderLineImpl() : base() {
        }

        public override double calculateSubTotal(double unitprice, int qty){
                return unitprice * qty;
        }
}
}
```

The following lines of code demonstrate how to invoke the logic of the implementation class with the persistence class, OrderLine generated using static method persistent API.

```
OrderLine orderline = OrderLine.CreateOrderLine();
orderline.ID = 1;
orderline.OrderQty = 5;
double subtotal = orderline.calculateSubTotal(300, 5);
orderlin.SubTotal = subtotal;
```

To invoke the logic of the implementation class by the persistent object, please refer to the Creating a Persistent Object for different types of persistent API.

Running the Sample Code

A set of C# sample files is generated which guides you how to implement the sample code and run the sample to see how the persistence classes work with the database if you have checked the option for samples before generating the persistent code. For more information on the setting of code generation, refer to Configuring Code Generation Setting for C# in the Implementation chapter.

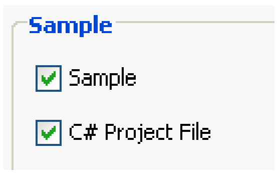


Figure 19.82 - Generate sample options

The common manipulation to the database includes inserting data, updating data, retrieving data and deleting data. As each type of manipulation can be performed individually, several sample files are generated to demonstrate how the persistence code handles each type of manipulation. The sample files are generated in the ormsamples directory of the specified output path.

There are six sample files demonstrating the database manipulation. They are identified by the type of manipulation and the name of the project.

Type of Manipulation	Sample File
Creating database table	Create%Project_Name%DatabaseSchema
Inserting record	Create%Project_Name%Data
Retrieving and updating record	RetrieveAndUpdate%Project_Name%Data
Deleting record	Delete%Project_Name%Data
Retrieving a number of records	List%Project_Name%Data
Dropping database table	Drop%Project_Name%DatabaseSchema

Table 19.38

Example:

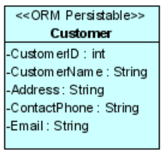


Figure 19.83 - ORM Persistable Class

Here is an object model of a Customer class inside a package of shoppingcart of the ShoppingCart project. By configuring the code generation setting with the factory class persistent API and Sample options checked, an ORM-Persistable .NET class, a factory class and six sample files are generated.

The following are the examples of manipulating the Customer table in the database by the generated sample files.

Creating Database Table

If the database has not been created by the Generate Database facility, you can generate the database by executing the CreateShoppingCartDatabaseSchema class.

```
using system;
using Orm;

namespace ormsamples {
public class CreateShoppingCartDatabaseSchema {
       [STAThread]
       public static void Main(string[] args){
            ORMDatabaseInitiator.CreateSchema(shoppingcart.ShoppingCartPersistentManager.Inst
            ance());
        }
}
}
```

After executing the CreateShoppingCartDatabaseSchema class, the database is created with the Customer table.

Inserting Record

As the database has been generated with the Customer table, a new customer record can be added to the table. By implementing the CreateTestData() method of the CreateShoppingCartData class, a new customer record can be inserted to the table. Insert the following lines of codes to the CreateTestData() method to initialize the properties of the customer object which will be inserted as a customer record to the database.

```
public void CreateTestData() {
        PersistentTransaction t =
        shoppingcart.ShoppingCartPersistentManager.Instance().GetSession().BeginTransaction();
        try {
            shoppingcart.Customer shoppingcartCustomer =
            shoppingcart.CustomerFactory.CreateCustomer();
            // Initialize the properties of the persistent object

            shoppingcartCustomer.CustomerName = "Joe Cool";
            shoppingcartCustomer.Address = "1212, Happy Building";
            shoppingcartCustomer.ContactPhone = "23453256";
            shoppingcartCustomer.Email = "joe@cool.com";

            shoppingcartCuetomer.Save();
            t.Commit();
        }
        catch (Exception e) {
            t.RollBack(0);
            shoppingcart.ShoppingCartPersistentManager.Instance().GetSession().Close();
            Console.WriteLine(e);
        }
}
```

After running the sample code, a customer record is inserted into the Customer table.

Retrieving and Updating Record

In order to update a record, the particular record must be retrieved first. By modifying the following lines of code to the RetrieveAndUpdateTestData() method of the RetrieveAndUpdateShoppingCartData class, the customer record can be updated.

```
private void RetrieveAndUpdateTestData() {
        PersistentTransaction t =
        shoppingcart.ShoppingCartPersistentManager.Instance().GetSession().BeginTransaction();
        try {
            shoppingcart.Customer shoppingcartCustomer =
            shoppingcart.CustomerFactory.loadCustomerByQuery("Customer.CustomerName='Joe
            Cool'", "Customer.CustomerName");
            // Update the properties of the persistent object

            shoppingcartCustomer.ContactPhone = "28457126";
            shoppingcartCustomer.Email = "joe.cool@gmail.com";

            shoppingcartCustomer.Save();
            t.Commit();
        }
        catch (Exception e) {
            t.RollBack(0);
            shoppingcart.ShoppingCartPersistentManager.Instance().GetSession().Close();
            Console.WriteLine(e);
        }
}
```

After running the sample code, the contact phone and email of the customer is updated.

Retrieving Record by ORM Qualifier

If the object model is defined with ORM Qualifier(s), the load method(s) for retrieving a particular record by ORM Qualifiers will be generated in the sample code as well.

From the example, the object model of Customer is defined with two ORM Qualifiers, named as Name and Phone, qualified with the attribute, CustomerName and ContactPhone respectively, two methods named as RetrieveCustomerByName() and RetrieveCustomerByPhone() are generated in the sample class. Modify the two methods as shown below.

```
public void RetrieveCustomerByName(){
        System.Console.WriteLine("Retrieving Customer by CustomerName...");
        // Please uncomment the follow line and fill in parameter(s)
        System.Console.WriteLine(shoppingcart.CustomerFactory.LoadByName("James Smith");
}

public void RetriveCustomerByPhone() {
        System.Console.WriteLine("Retrieving Customer by ContactPhone...");
        // Please uncomment the follow line and fill in parameter(s)
        System.Console.WriteLine(shoppingcart.CustomerFactory.LoadByPhone("62652610");
}
```

Retrieving Record by Criteria Class

If you have selected the Generate Criteria option before the generation of persistent code, the Criteria class will be generatedaacordingly.

For the example, the Criteria class, named as CustomerCritera is generated for the object model of Customer. A method named as RetrieveByCriteria() is generated in the sample class. By following the commented guideline of the method, uncomment the code and modify as follows.

```
pbulic void RetrieveByCriteria() {
        System.Console.WriteLine("Retrieving Customer y CustomerCriteria");
        shoppingcart.CustomerCriteria customerCriteria = new shoppingcart.CustomerCriteria();
        //Please uncomment the follow line and fill in parameter(s)
        customerCriteria.CustomerID.Eq(3);
        System.Console.WriteLine(customerCriteria.UniqueCustomer().CustomerName);
}
```

In order to run these two methods, uncomment the statements (blue colored in the diagram below) in the Main(string[] args) method.

```
public static void Main(string[] args){
        RetrieveShoppingCartData retrieveShoppingCartData = new RetrieveShoppingCartData();
        try {
                retrieveShoppingCartData.RetrieveAndUpdateTestData();

                retrieveShoppingCartData.RetrieveCustomerByName();
                retrieveShoppingCartData.RetrieveCustomerByPhone();
                retrieveAndUpdateShoppingCartData.RetrieveByCriteria();

        }
        finally {
                shoppingcart.ShoppingCartPersistentManager.Instance().DisposePersistManger();
        }
}
```

Having executed the RetrieveAndUpdateShoppingCartData class, the contact phone and email of the customer is updated, and the retrieved customer records are shown in the system output. (Assuming several customer records are inserted to the database.)

Deleting Record

A customer record can be deleted from the database by executing the DeleteShoppingCartData class. Modify the following lines of code in the DeleteTestData() method to delete the specified record from the database.

```
private void DeletetestData() {
        PersistentTransaction t =
        shoppingcartPersistentManager.Instance().GetSession.BeginTransaction();
        try {
                shoppingcart.Customer shoppingcartCustomer =
                shoppingcart.CustomerFactory.LoadCustomerByQuery("Customer.CustomerName='Harry
                Hong'", "Customer.CustomerName");
```

```
            shoppingcartCustomer.Delete();
            t.Commit();
        }
        catch (Exception e) {
            t.RollBack();
            shoppingcart.ShoppingCartPersistentManager.Instance().GetSession().Close();
            Console.WriteLine(e);
        }
}
```

After running the DeleteShoppingCartData sample, the customer record whose customer name is equals to "Harry Hong" will be deleted.

Retrieving a Number of Records

The list method of the generated persistence class supports retrieving a number of records from the database, By modifying the following lines of code to the ListTestData() method of the ListShoppingCartData class, the customer whose name starts with "Chan" will be retrieved and the name will be displayed in the system output.

```
public void LsitTestData(){
        System.Console.WriteLine("Listing Custoer...");
        shoppingcart.Customer[] shoppingcartCustomers =
        shopping.CustomerFactory.ListCustomerByQuery("Customer.CustomerName like 'Chan%'",
        "Customer.CustomerName");
        int length = Math.Min(shoppingcartCustomers.Length, ROW_COUNT);
        for (int i=0; i<length; i++){
            System.Console.WroteLine(shoppingcartCustomers[i].CustomerName);
        }
        System.Console.WriteLine(length + " record(s) retrieved.");
}
```

Retrieving a Number of Records by ORM Qualifier

If the object model is defined with ORM Qualifier(s), the list method(s) for retrieving records by ORM Qualifiers will be generated in the sample code as well.

From the example, the object model of Customer is defined with two ORM Qualifiers, named as Name and Phone, qualified with the attribute, CustomerName and ContactPhone respectively, two methods named as ListCustomerByName() and ListCustomerByPhone() are generated in the sample class. Modify the two methods as shown below.

```
public void LsitCustomerByName(){
        System.Console.WriteLine("Listing Customer by CustomerName...");
        //Please uncomment the follow lines and fill in parameters
        shoppingcart.Customer[] customers = shoppingcart.CustomerFactory.ListByName("Wong%");
        int length = customers == null ? 0 : Math.Min(customers.length, ROW_COUNT);
        for (int i=0; i<length; i++){
            System.Console.WriteLine(customer[i].CustomerName)
        }
        System.Console.WriteLine(length + " record(s) retrieved.");
}

public void ListCustomerByPhone(){
        System.Console.WriteLine("Listing Customer by ContactPhone...");
        // Please uncomment the follow lines and fill in parameters
        shoppingcart.Customer[] customers = shoppingcart.CustomerFactory.ListByPhone("26%");
        int length = customer == null ? 0 : Math.Min(customers.length, ROW_COUNT);
        for (int i=0; i<length; i++){
            System.Console.WriteLine(customers[i].CustomerName);
        }
        System.Console.WriteLine(length + " record{s} retrieved.");
}
```

Retrieving a Number of Records by Criteria

As the criteria class is generated for the Customer object model, a method named as ListByCriteria() is generated in the sample class. Modify the following lines of code.

```
public void LsitByCriteria(){
        System.Console.WriteLine("Listing Customer by Criteria...");
        shoppingcart.CustomerCriteria customerCriteria = new shoppingcart.CustomerCriteria();
        // Please uncomment the follow line and fill in parameter(s)
        // customerCriteria.CustomerID.Eq();
        customerCriteria.CustomerName.Like("Cheung%");
        customerCriteria.SetMaxResults(ROW_COUNT);
        shoppingcart.Customer[] customers = customerCriteria.ListCustomer();
        int length = customer == null ? 0 : Math.Min(customers.Length, ROW_COUNT);
        for (int i=0; i<length; i++){
                System.Console.WriteLine(customers[i].CustomerName);
        }
        System.Console.WriteLine(length + " Customer record(s) retrieved.");
}
```

In order to run these three methods, uncomment the statements (blue colored in the diagram below) in the Main(string[] args) method.

```
public static void Main(string[] args){
        ListShoppingCartData listShoppingCartData = new ListShoppingCartData();
        try{
                listShoppingCartData.ListTestData();

                listShoppingCartData.ListCustomerByName();
                listShoppingCartData.ListCustomerByPhone();
                listShoppingCartData.ListByCriteria();

        }
        finally {
                shoppingcart.ShoppingCartPersistenetManager.Instance().DisposePersitentManager();
        }
}
```

Having executed the ListShoppingCartData class, the matched records are retrieved and displayed with the customer name according to the condition defined in the ListTestData(), ListCustomerByName(), ListCustomerByPhone() and ListByCriteria()methods.

Dropping Database Table

In some cases, you may want to drop the database tables and create the database tables again. By running the DropShoppingCartDatabaseSchema class first, all the database tables will be dropped accordingly.

```
namespace ormsamples {
public class DropShoppingCartDatabasesSchema {
        [STAThread]
        public static void Main(string[] args) {
                System.Console.WriteLine("Are you sure to drop table(s)?(Y/N)");
                if (System.Console.ReadLine().Trim().Topper.Equals("Y")) {
                        ORMDatabaseInitiator.DropSchema(shoppingcart.ShoppingCartPersistentmanager
                        .Instance());
                }
        }
}
}
```

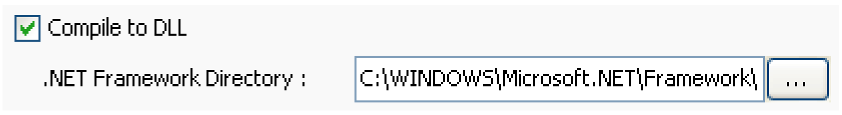
Applying .NET Persistence Class to different .NET Language

As the generated .NET persistence class which provides methods to manipulate the database, you can directly implement these methods to manipulate the database. Since the generated .NET persistence class is applicable to all .NET language, such as C#, C++ and VB.NET, the C# source is generated for demonstrating the usage of .NET persistence class.

You can apply the generated .NET persistence class to different .NET language by one of the two ways:

- By compiling C# source to DLL

  SDE can compile the C# source to DLL files which can be referenced by other projects of other .NET language. To compile the C# source to DLL files, simply check the option of Compile to DLL on the Database Code Generation dialog box before generating the persistence code. For more information, refer to the description of Configuring Code Generation Setting for C# in the Implementation chapter.



☑ Compile to DLL

.NET Framework Directory : C:\WINDOWS\Microsoft.NET\Framework\ ...

Figure 19.84 - Compile to DLL options

- By referencing C# project

  Having generated the .NET persistence class, you can first create the C# project with the generated source, and then create another project with other .NET language with referencing the created C# project.

E

JDBC and .NET Drivers

Appendix E - JDBC and .NET Drivers

This chapter lists the download link of the JDBC and .NET Drivers for database supported by SDE.

JDBC Drivers

Table shows the download link of JDBC Drivers with respect to the database supported in SDE.

Database name	Download link
Cloudscape/Derby 10	http://www-306.ibm.com/software/data/cloudscape
HSQLDB	http://hsqldb.sourceforge.net/
MS SQL Server (DataDirect SequeLink Driver)	http://www.datadirect.com/download/index.ssp
MS SQL Server (jTDS Driver)	http://jtds.sourceforge.net/
MS SQL Server (Microsoft Driver)	http://www.microsoft.com/downloads/details.aspxFamilyID=ee91ad1a-1ee4-49e1-95ea-e3f0e39114a9&DisplayLang=en
MS SQL Server (WebSphere Connect Driver)	http://www-1.ibm.com/support/docview.wssrs=180&org=SW&doc=4001312
MS SQL Server (JSQL Driver)	http://www.jnetdirect.com/products.phpop=jsqlconnect
MS SQL Server (JTURBO Driver)	http://www.newatlanta.com/c/products/jturbo/download/home
MS SQL Server (WebLogic Connect Driver)	WebLogic Type 4 JDBC drivers are installed with WebLogic Server in the WL_HOME\server\lib folder, where WL_HOME is the directory in which you installed WebLogic Server. Driver class files are included in the manifest classpath in weblogic.jar, so the drivers are automatically added to your classpath on the server. The WebLogic Type 4 JDBC drivers are not included in the manifest classpath of the WebLogic client jar files (e.g., wlclient.jar). To use the drivers with a WebLogic client, you must copy the following files to the client and add them to the classpath on the client: wlbase.jar wlutil.jar The DBMS-specific JAR file: For DB2: wldb2.jar For Informix: wlinformix.jar For MS SQL Server: wlsqlserver.jar For Oracle: wloracle.jar For Sybase: wlsybase.jar
MySQL (Connector/J Driver)	http://www.mysql.com/products/connector/j/
Oracle8i (THIN JDBC Driver)	http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html
Oracle9i	http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html
Oracle (DataDirect SequeLink Driver)	http://www.datadirect.com/download/index.ssp
PostgreSQL	http://jdbc.postgresql.org/download.html

Sybase Adaptive Server Enterprise (jConnect)	http://www.sybase.com/products/informationmanagement/softwaredeveloperkit
Sybase Adaptive Server Enterprise (jTDS)	http://jtds.sourceforge.net/
Sybase SQL Anywhere (jConnect)	http://www.sybase.com/products/middleware/jconnectforjdbc

Table E.1

.NET Drivers

Table shows the download link of .NET Drivers with respect to the database supported in SDE.

Database Name	Download Link
DB2 7/8	http://www-03.ibm.com/servers/eserver/iseries/access/dotnet/
MS SQL Server 2000	http://jtds.sourceforge.net/
MySQL 3/4	http://www.mysql.com/products/connector/net/
Oracle	http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html
PostgreSQL	http://jdbc.postgresql.org/ http://gborg.postgresql.org/project/npgsql/projdisplay.php

Table E.2

F

Glossary

Appendix F - Glossary

A	
Activity diagram	Activity diagrams are an amalgamation of a number of techniques: Jim Odell's event diagrams, SDL state modeling techniques, workflow modeling and petri-nets. They can also be considered as variants of state diagrams. Activity diagrams are organized according to actions and mainly target towards representing the internal behaviors of a method or a use case. They describe the sequencing of activities, with support for both conditional and parallel behaviors.
Actor input	In editing flow of event, an actor input is the input from an actor to the system.
Align to grid	Whether diagram elements should align to grid when being moved.
Anti-aliasing	A method which handles the staircase pixels of slanted lines and curves to make them look smoother.
Application Options	The global options in SDE.
Auto save	SDE provides an auto save feature that saves a copy of the current project periodically. If the application terminates abnormally, with this feature turned on, when you start SDE again the system will check if an auto save file exists. If so, it will ask you whether you want to recover the project.
Automatic containment rule detection	A facility to automatically detect the containment rule for a container. For example, an Actor will not be contained in the System Boundary even if they are moved into the container's region.
B	
Backup files	Every time you save a project a backup file will be created. The backup file name is determined by the original project file name, followed by a "~" and the version number. A backup file with a larger version number means that it is more recent than those with smaller version numbers.
Button group	The diagram toolbar groups some of the diagram elements that are similar in nature. For example, Package and Subsystem are grouped into a single button group. Buttons that are grouped are indicated by a small triangle on the lower-right-hand corner. To view the list of items under the group click on the small triangle, or click on the button and hold until the selection list appears.
C	
Candidate Class Pane	The candidate class pane, located at the upper-right corner of the textual analysis pane, displays the candidate classes as rectangle shapes.
Candidate class view	In performing textual analysis, the Candidate Class View hides the Problem Statement Editor and only displays the Candidate Class Pane and the Data Dictionary Table. It allows you to concentrate on further editing of the identified candidate classes, such as specifying the candidate class type or creating models.
Cascade	Arrange the opened windows diagonally, with the active window on top.
Class diagram	Class diagrams are the most common diagram found for modeling object-oriented systems. They are used to describe the types of objects and their relationships by providing a static, structural view of a system. They are important not only for visualizing, specifying, and documenting structural models, but also for constructing executable systems through forward and reverse engineering.
Class repository	A project may contain many classes. The Class Repository View lists all the classes within the current project.
Collaboration diagram	Collaboration diagrams emphasize the organization of objects that are involved in an interaction. Collaboration is a mechanism composed of both structural and behavioral elements. Two important features - the concept of a path and the sequence number - distinguish collaboration diagrams from sequence diagrams.
Component diagram	Component diagrams show the various components (physical modules of code) in a system and their dependencies. A component can often be the same as a package.
Copy as image	To copy the selected diagram elements to the system clipboard as image. This feature is supported in both the Windows platform and the Linux platform.
Copy to system clipboard	To copy the selected diagram elements to the system clipboard as OLE objects, so that the copied content can be pasted to OLE containers like Microsoft Word/Excel/PowerPoint. And you can directly edit the OLE object inside the document. This feature is supported in Windows platform only.

Copy within SDE	To copy the selected diagram elements to the application clipboard. You can then paste the diagram elements to other SDE diagrams.
D	
Data dictionary table	The data dictionary table, which located at the lower-right area of the textual analysis pane, provides a table view for the candidate classes. It displays all the information of a candidate class. You can edit its name and type, as well as adding description to the candidate class.
Data dictionary view	In performing textual analysis, the Data Dictionary View displays only the Data Dictionary Table. It allows you to concentrate on filling the candidate class information in the data dictionary.
Deployment diagram	Deployment diagrams show the physical layout and relationships among software and hardware components in the implemented system. It shows how components and objects are routed and moved around a distributed system.
Diagram base layout	In the print preview pane, if the Fit to Pages option is selected, and there are multiple pages in the printout, selecting Diagram Base Layout will cause the distribution of pages to be diagram-oriented. Note that this option affects the preview only, the order of the printout remains unchanged.
Diagram element	A diagram element is a shape or a connector that represent the view of its underlying model element.
Diagram exporter	The diagram exporter allows you to export selected diagrams as images in JPG, PNG or SVG format.
Diagram pane	The diagram pane contains the opened diagrams; it allows you edit multiple diagrams at the same time.
Diagram toolbar	The diagram toolbar contains the buttons of the diagram elements available for developing the active diagram.
Diagram navigator	A project may consist of many diagrams. The Diagram Navigator lists all the diagrams within the project. Through the use of a folding tree structure, you can browse the names of these diagrams by expanding or collapsing the folders and perform sorting by diagram type.
Document info	When generating HTML/PDF reports, the document info (such as title, author, keywords) you specified becomes the meta data of the report. Users can open the HTML source/PDF document summary to view this information.
Documentation pane	The Documentation pane allows you to enter a description about a diagram or a diagram element.
E	
Extra Resource-Centric	By default, the resource-centric interface displays the most commonly used resources of a diagram element. The least commonly used resources are hidden by default, and they are called the extra resources.
F	
Flow of event	A section in the use case description for editing the base paths and the alternative paths in a use case.
H	
HTML report generation	To generate report for the SDE project in HTML format.
J	
Java-enabled platforms	Any platforms that have Java runtime installed and thus able to run Java programs.
L	
Layout diagram	A feature to layout the shapes so that they do not overlap, and to layout the connectors so that they do not cross with one another.
License key	The license key is a file that you import using the License Key Manager so that you can start using SDE.
License Key Manager	The License Key Manager allows you to manage the license key files of Visual Paradigm products.
Logical View	The Logical View refers to a user's view of the way project is organized. It provides another view of creating, structuring and sharing the UML diagrams and models apart from the traditional Diagram Navigator, Model Tree View and Class Repository.
Look and Feel	The appearance of SDE user interface.

M	
Message pane	The message pane logs the messages for the operations that you performed. For example, Java language syntax checking, model validation, report generation, etc.
Model element	A model element stores the model data. A diagram element associates a model element, and a model element may be associated with more than one diagram element (multiple views).
Model repository	The repository where the model elements are stored.
Model tree view	The Model Tree View lists all the model elements within the current project. Model elements can be dragged to appropriate diagrams to create a new diagram element.
Model validation	A process to validate the models against UML syntax.
O	
OLE	An object that supports the OLE protocol for object linking and embedding.
Open specification dialog	The open specification dialog of a diagram allows you to configure the diagram settings, such as the diagram name and grid settings; while the open specification dialog of a model element allows you to configure its model data.
ORM Pane	Display a list of classes and database tables from the specified classpath (s) and database (s). You can click **Refresh** to update the content under Class View and DataBase View whenever there are changes to source code or database. You can drag classes or entities onto diagrams and generate source code/database from them when necessary.
P	
Paper base layout	If the Fit to Pages option is selected, and there are multiple pages in the printout, selecting Paper Base Layout will cause the distribution of pages to be paper-oriented (the diagram size is ignored in arranging the preview). Note that this option affects the preview only; the order of the printout remains unchanged.
Paper place style	To change the order of the printout. Consider a large diagram is divided into many pages, selecting From left to right will arrange the printout order from the pages on the left to the pages on the right, while selecting From top to bottom will arrange the print order from the pages on the top to the pages on the bottom.
Pattern watermark	The watermark that repeats the product name diagonally in the printout, exported image or copied content.
PDF report generation	To generate report for the SDE project in PDF format.
Preview pane	The Preview pane, also known as the Diagram Monitor, shows an overall view of the diagram. The Diagram Monitor allows you to navigate the whole diagram pane when the diagram is larger than the display area of the diagram pane.
Print preview pane	The print preview pane allows you to configure various print settings, preview the printout and print the diagrams.
Problem statement	A description about the problem to investigate.
Problem statement editor	The problem statement editor is the text editor located on the left of the text analysis pane, which allows you to view and edit the problem statement.
Problem statement view	The Problem Statement View displays the Problem Statement Editor, the Candidate Class Pane and the Data Dictionary Table; it allows you to concentrate on editing the problem statement.
Project explorer	The project explorer pane contains three views: the Diagram Navigator, the Model Tree View, and the Class Repository View. Each view shows different perspectives of the project.
Properties pane	There are four pages associated with the Properties Pane: the Property page, the Preview page, the Documentation page and the Element Viewer page.
Property pane	Every diagram and diagram element has its own properties. The Property pane in the Properties Pane allows you to view and edit various its properties.
Q	
Quick Print	To print diagrams without previewing them hence speeds the print job.

R	
Realistic containment interaction	A specific effect to indicate a diagram element moving in/out of a container.
Reference shape for alignment	When there are multiple shapes selected, the last selected shape will be used as the referenced shape for alignment. That is, the alignment methods will be performed based on the position/size of the referenced shape. The referenced shape will be rendered with its resize handles surrounded by black rectangles.
Report Writer	A feature for performing agile report creation.
Resource-centric	A user interface based on the Resource-Centric approach is adopted in SDE to enable UML diagrams to be constructed intuitively with minimal efforts. With the novel interface, only valid editing resources are grouped around a graphical entity, totally eliminating invalid operations during diagram construction.
Rose importer	The Rose importer allows you to import a Rational Rose project file and convert it as the diagrams and models into your SDE project.
Round trip engineering	Round trip engineering is the process to convert from diagram to code, and to convert from code to diagram.
S	
Scrollable toolbar	If you have resized the diagram pane to the extent that some of the buttons on the diagram toolbar are not visible, an "Up" button and a "Down" button will appear. You can click on these buttons to scroll up or down to the desired buttons on the toolbar.
Sequence diagram	Sequence diagram captures the behavior of a single use case and displays a number of example objects, as well as the messages that are passed between these objects within the use case from a temporal standpoint. There are two important features, which are the object lifeline and the focus of control that distinguish them from collaborative diagrams.
Single line watermark	The watermark that prints a single line of the product name in the printout, exported image or copied content.
State diagram	State diagrams, sometimes referred to as state chart diagrams, are a common technique to describe the dynamic behavior of a system. They represent state machines from the perspective of states and transitions, describing all the possible states that a particular object can get into and how the object's state changes as a result of events that affect the object. In most Object-Oriented techniques, state diagrams are drawn for a single class to show the lifetime behaviors of a single object.
Stencil Pane	Although the original UML notations are rich, but still may not expressive enough to present your idea. The stencil in SDE provides a large variety of shapes apart from the ordinary UML notations, and you can place the stencil in UML diagram to present your own idea. Stencil Pane is a repository where the imported those shapes are stored.
Stereotype	The stereotype concept provides a way of classifying (marking) elements so that they behave in some respects as if they were instances of new "virtual" metamodel constructs.
Sub-diagrams	A facility to associate a diagram with other lower level UML diagrams to facilitate levels of abstraction and increase the traceability among UML diagrams.
System response	In editing flow of event, a system response is the response from the system (to an actor input).
T	
Textual analysis	Textual analysis is a process to analyze the system domain. It helps to identify the candidate classes in a problem statement.
Tile	Arrange the opened windows so that all windows are visible at the diagram pane.
Tile horizontally	Arrange the opened windows horizontally. The windows are resized to share the available workspace height, without overlapping each other.
Tile vertically	Arrange the opened windows vertically. The windows are resized to share the available workspace width, without overlapping each other.

U	
UML	The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of the best engineering practices that have proven successful in the modeling of large and complex systems.
Use case description	A use case description describes the use case, including the preconditions, post-conditions, flow of events, etc.
Use case detail	A use case detail holds one or more use case description.
Use case diagram	Use case diagrams, together with activity diagrams, state diagrams, sequence diagrams and collaboration diagrams, are the five diagrams in UML for modeling the dynamic aspects of a system. Invented by Ivar Jacobson, use case diagrams are central to modeling the behaviors of the system, a sub-system or a class, providing a means to visualize, specify and document the behaviors of an element. They describe the behavior of a system from a user's perspective by using actions and reactions. A use case shows the relationships between actors and objects, and between the system and its environment.
Use case scheduling	To schedule the use cases by assigning priorities.
V	
Visio integration	SDE allows you to create Visio drawing in UML diagrams. Besides, you can also import Visio stencil into SDE and use the Visio shape in UML diagrams.
Visual Paradigm Suite	Abbreviated as VP-Suite, Visual Paradigm Suite allows you to install all Visual Paradigm leading CASE Tools.
X	
XMI importer	The XMI importer imports the models from an XMI file into a SDE project.