

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:

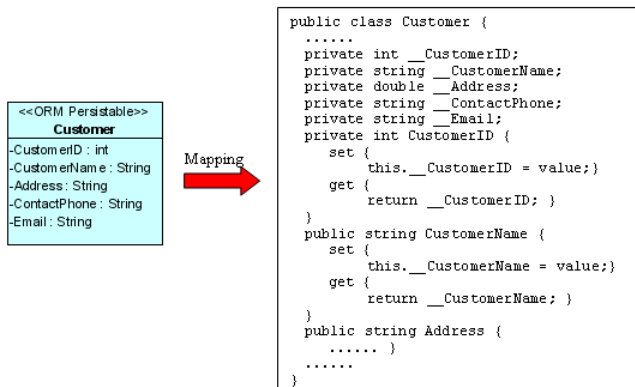


Figure 19.1 - Mapping attributes

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:

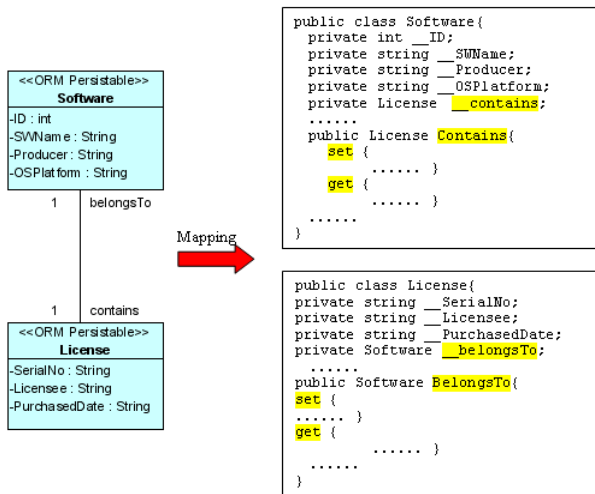


Figure 19.2 - Mapping One-to-one association

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.

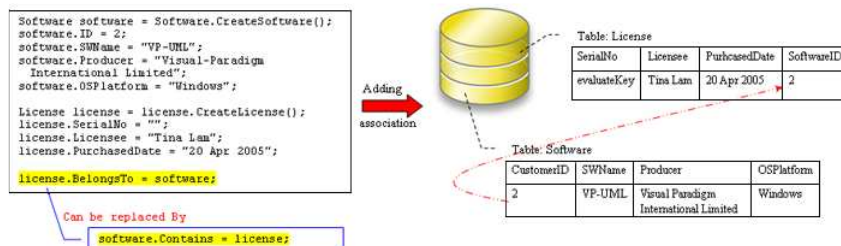


Figure 19.3 - Code for adding one-to-one association

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.

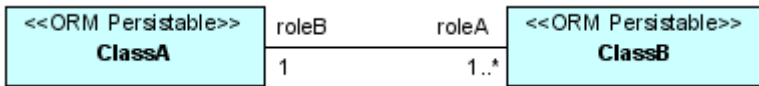


Figure 19.4 - One-to-many association

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:

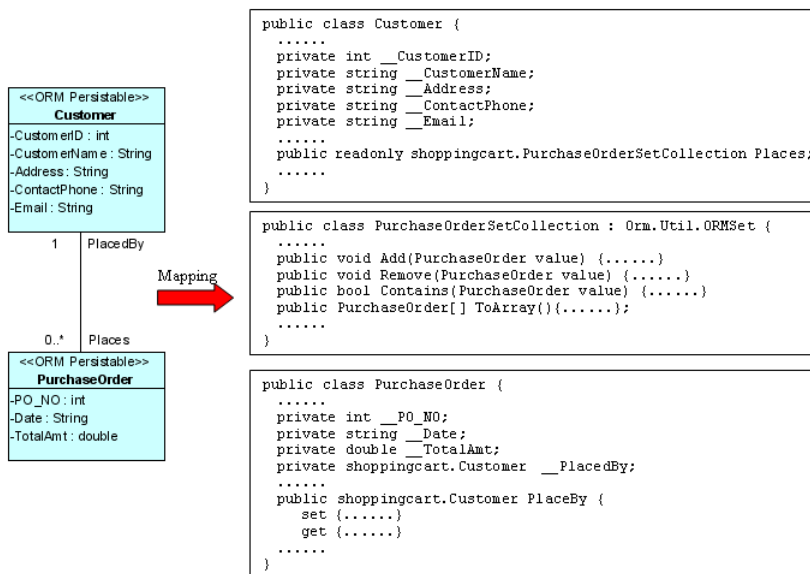



Figure 19.5 - Mapping One-to-many association

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 PurchaseOrder.

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;
```

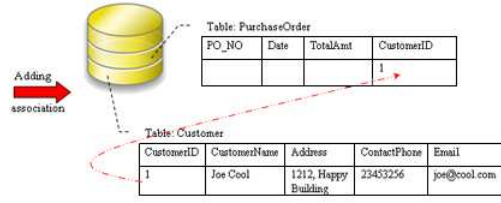


Figure 19.6 - Code for add one-to-many association

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:

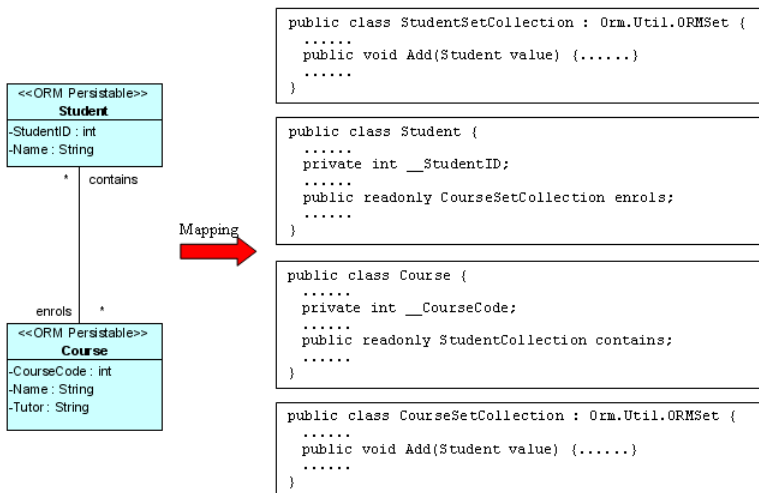


Figure 19.7 - Mapping many-to-many association

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.

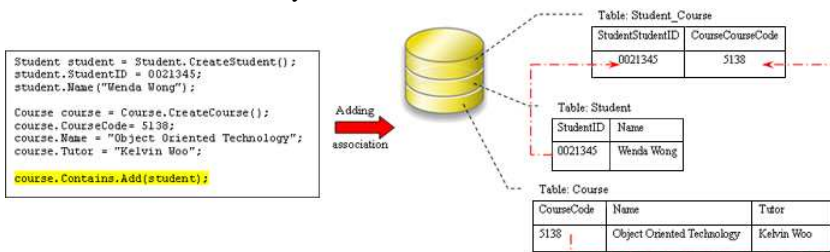


Figure 19.8 - Code for add a many-to-many association

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	GetEnumerator()	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	GetEnumerator()	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	GetEnumerator()	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:

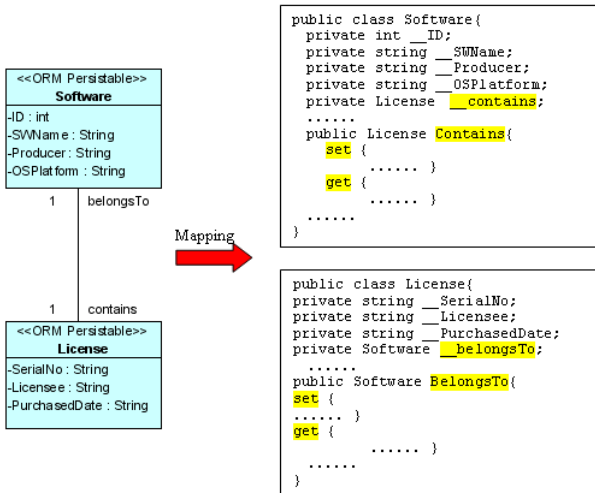


Figure 19.9 - Mapping One-to-one Association

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.

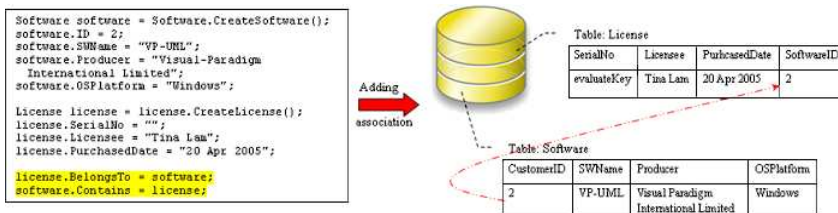



Figure 19.10 - Code for adding one-to-one association

 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.

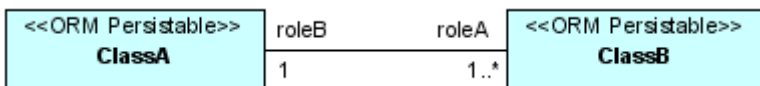


Figure 19.11 - One-to-many Association

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:

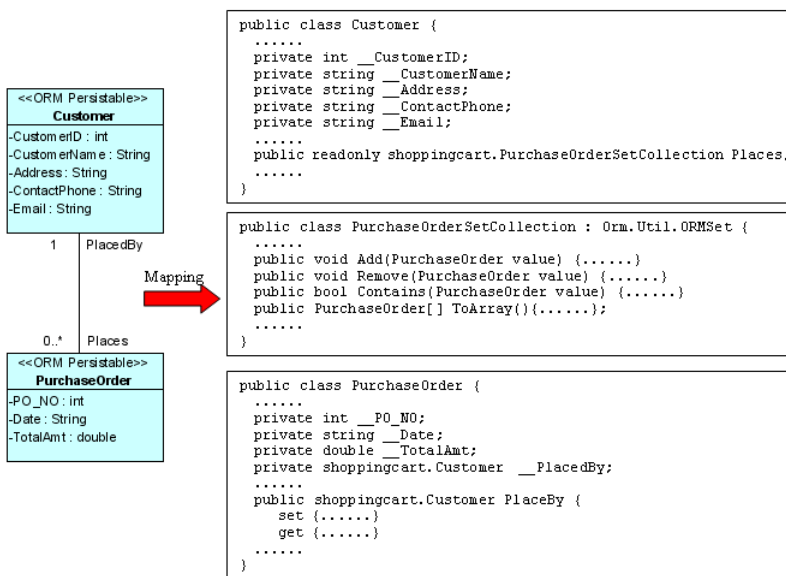


Figure 19.12 - Mapping One-to-many Association using Collection

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 PurchaseOrder.

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.

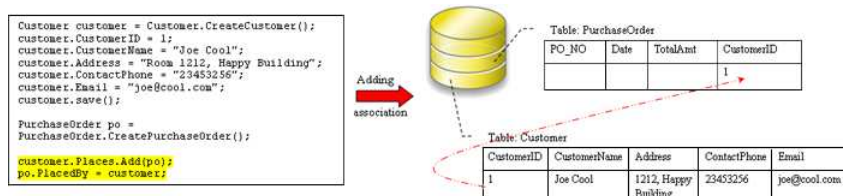


Figure 19.13 - Code for one-to-many association using Collection

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.

 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:

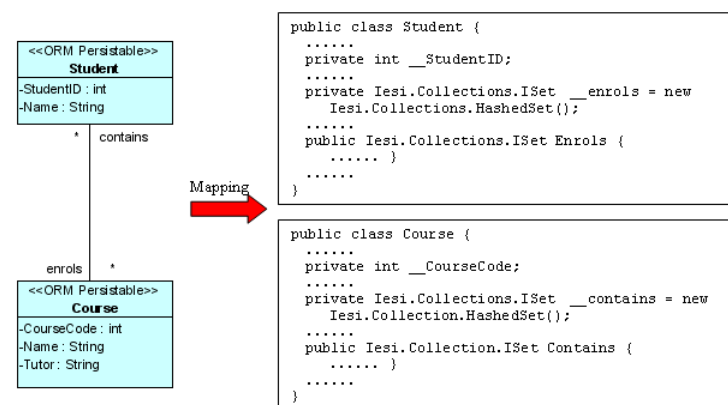


Figure 19.14 - Mapping Many-to-many association using Collection

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.Enrols.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.

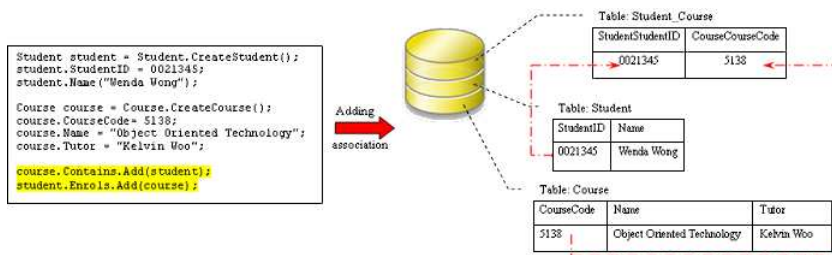


Figure 19.15 - Code for adding Many-to-many association using Collection

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.

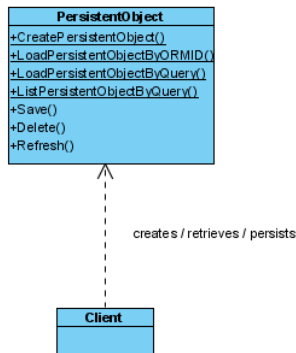



Figure 19.16 - Class diagram with static methods

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:

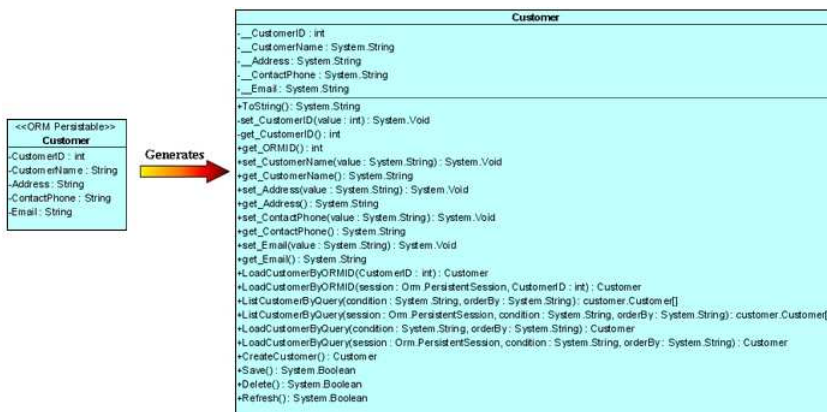


Figure 19.17 - Mapping ORM Persistable Class

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:

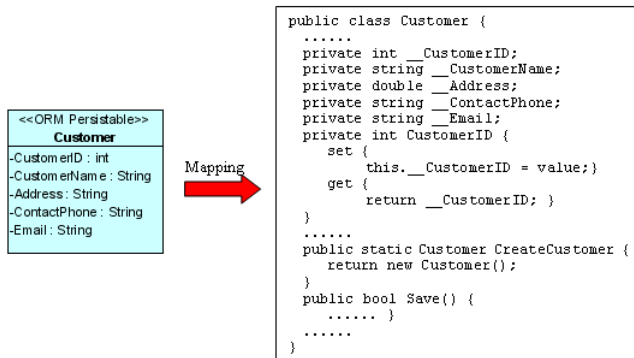


Figure 19.18 - Mapping creator methods

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.

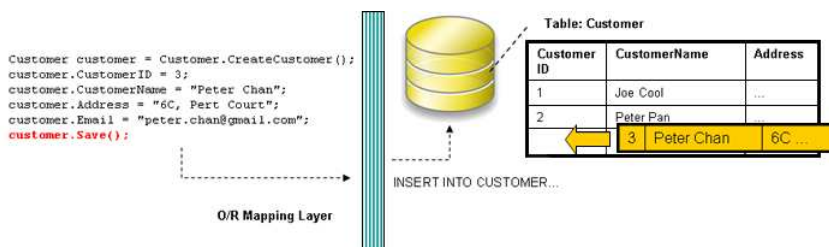


Figure 19.19 - Code for create a records

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:

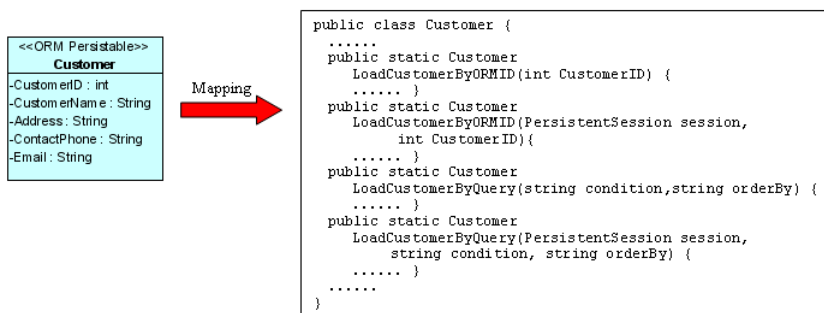


Figure 19.20 - Mapping load methods

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.

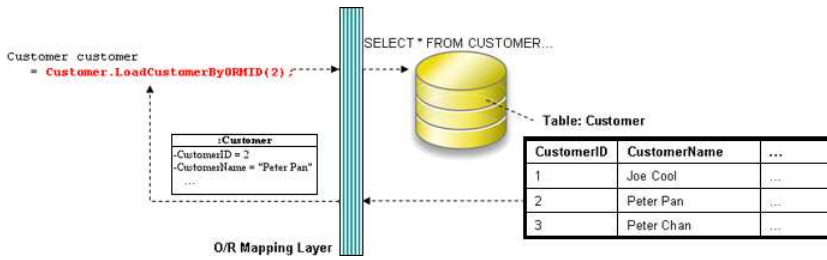


Figure 19.21 - Code for load a record

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:

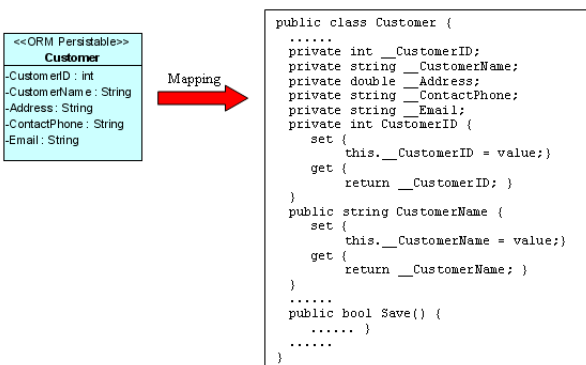


Figure 19.22 - Mapping for update record

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.

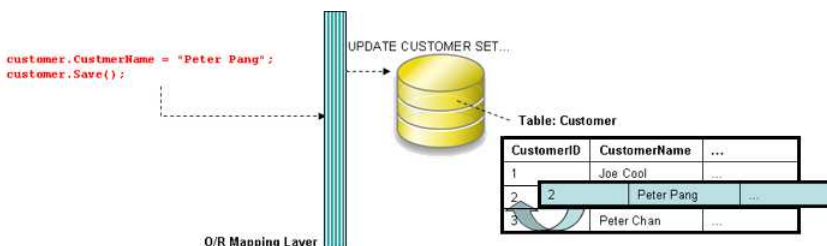


Figure 19.23 - Code for update record

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:

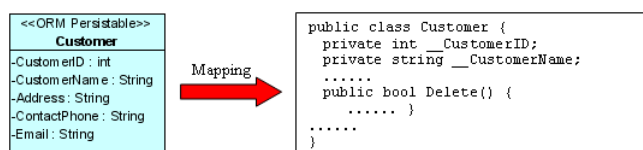


Figure 19.24 - Mapping delete methods

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.

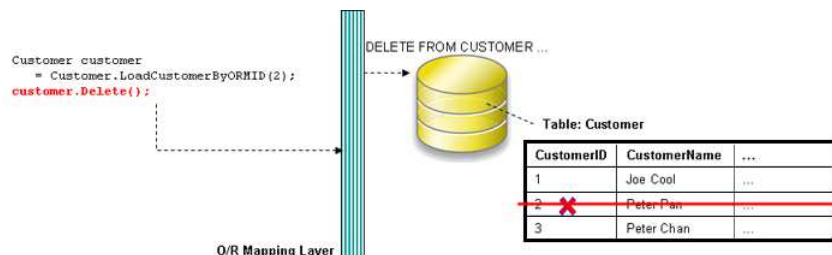


Figure 19.25 - Code for delete a record

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:

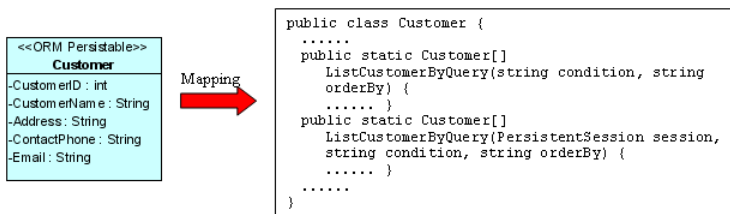


Figure 19.26 - Mapping list methods

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.

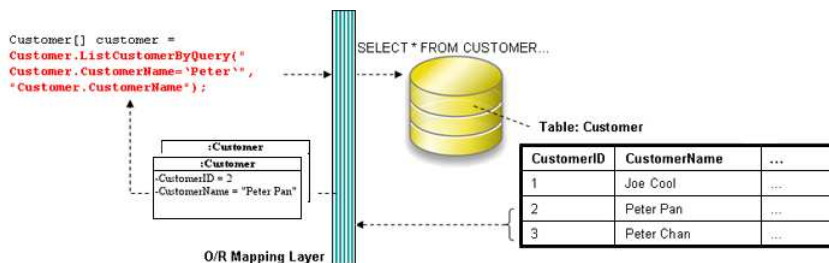


Figure 19.27 - Code for retrieve a list of records

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:

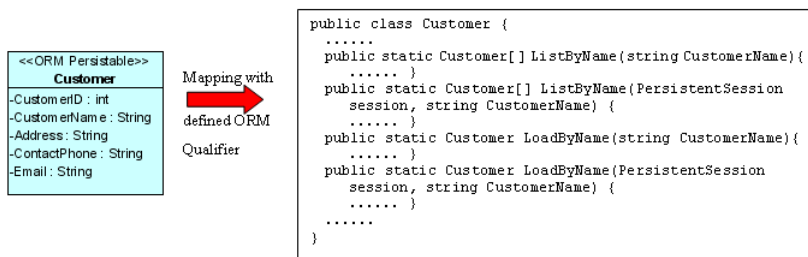


Figure 19.28 - Mapping load and list method with qualifier

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.

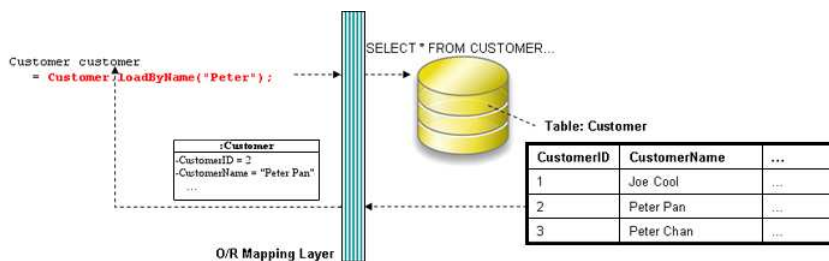


Figure 19.29 - Retrieve a record by qualifier

- 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.

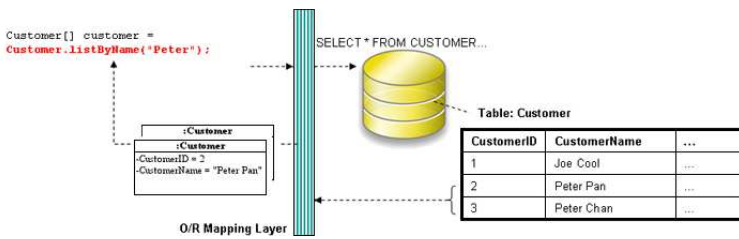


Figure 19.30 - Retrieve a list of records by qualifier

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:

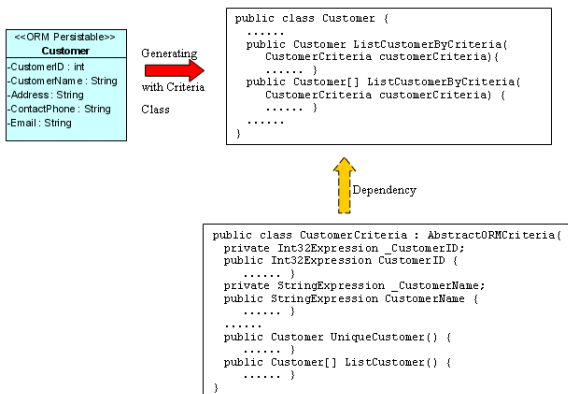


Figure 19.31 - Mapping with Criteria Class

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.

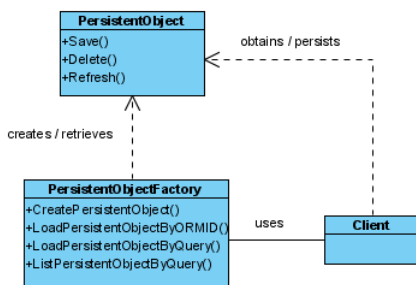


Figure 19.32 - Class Diagram for Factory Class

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:

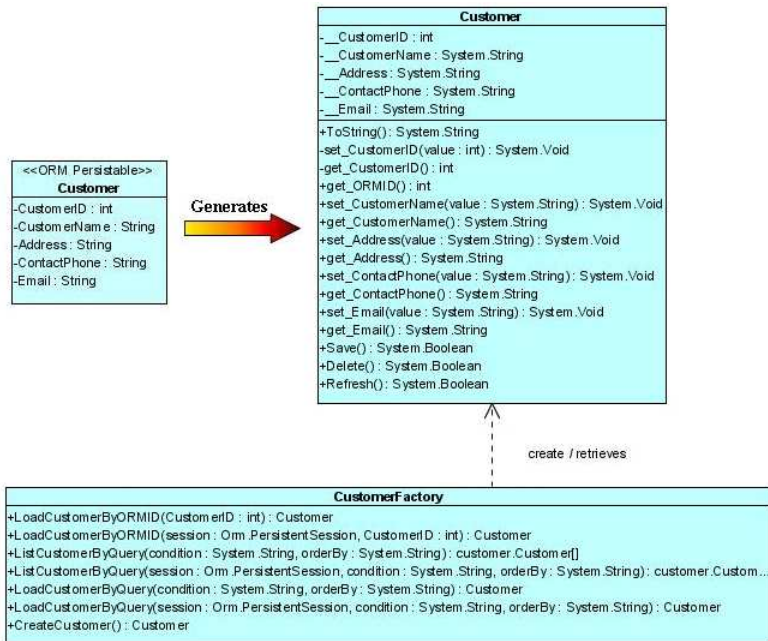


Figure 19.33 - Class Diagram with Factory Class

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:

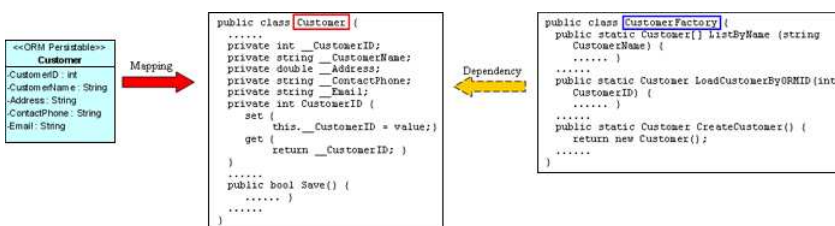



Figure 19.34 - Mapping with Factory Class

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.

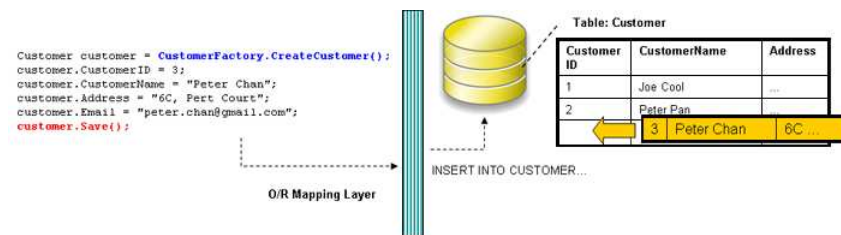


Figure 19.35 - Code for create records using factory class

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:

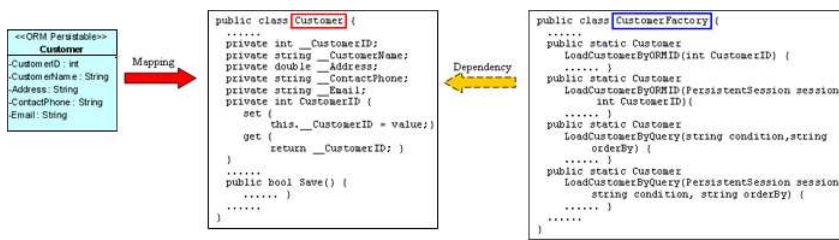


Figure 19.36 - Mapping load methods in factory class

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.

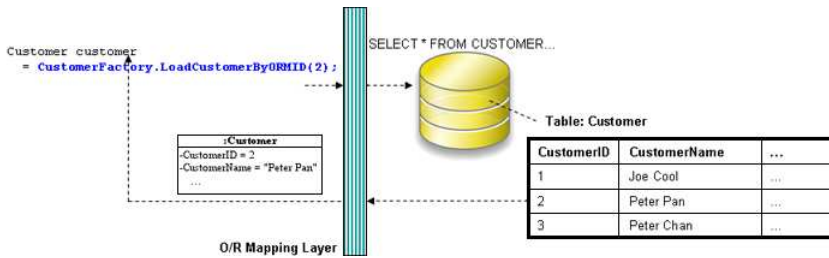


Figure 19.37 - Code for load a records

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:

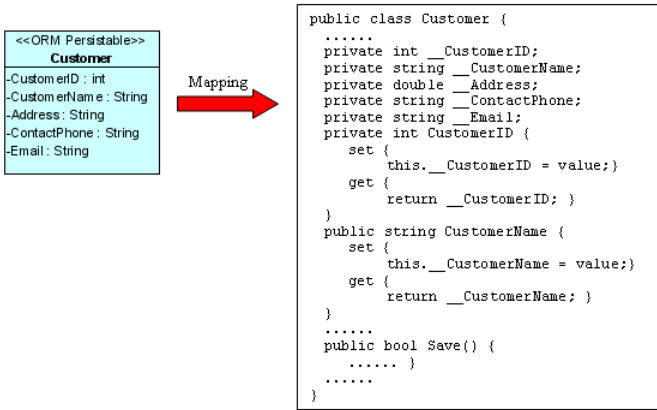


Figure 19.38 - Mapping save method

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.

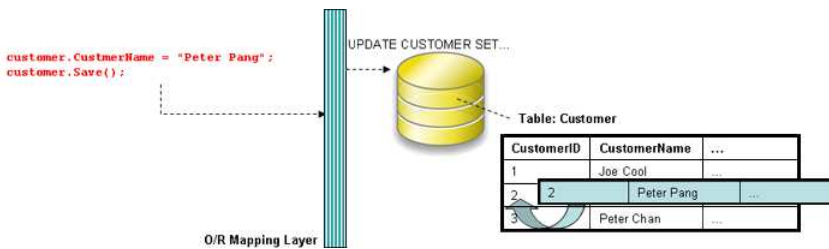


Figure 19.39 - Code for insert a record

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:

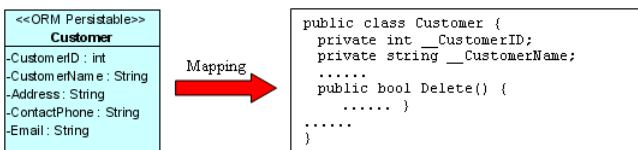


Figure 19.40 - Mapping delete methods

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.

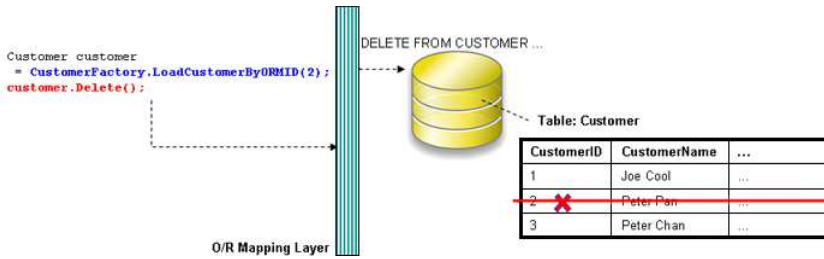


Figure 19.41 - Code for delete record

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:

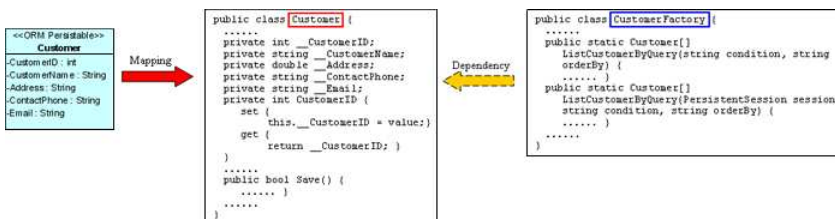


Figure 19.42 - Mapping list method in Factory

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.

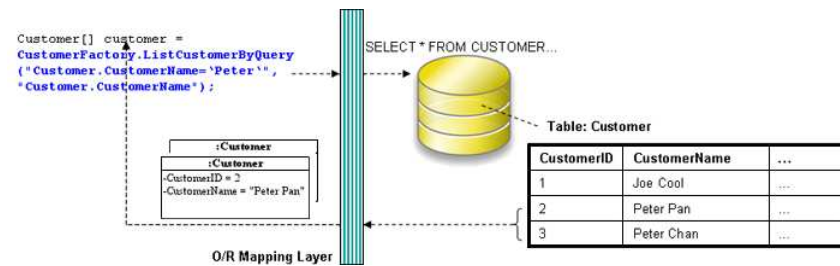


Figure 19.43 - Retrieve a list of records by using Factory Class

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:

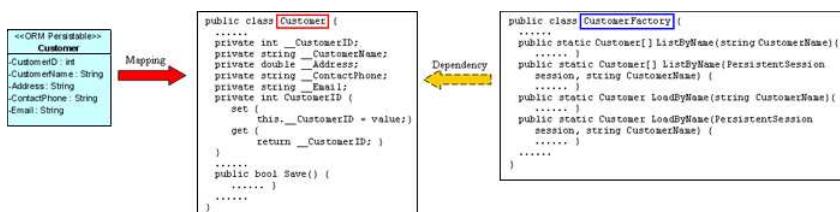


Figure 19.44 - Mapping list and load by qualifier methods

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.

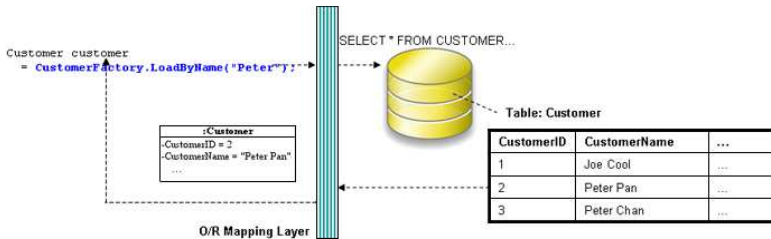


Figure 19.45 - Code for load a record by qualifier

- 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.

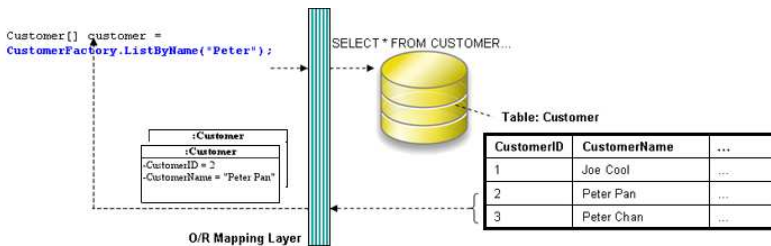


Figure 19.46 - Code for retrieve a list of record by qualifier

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:

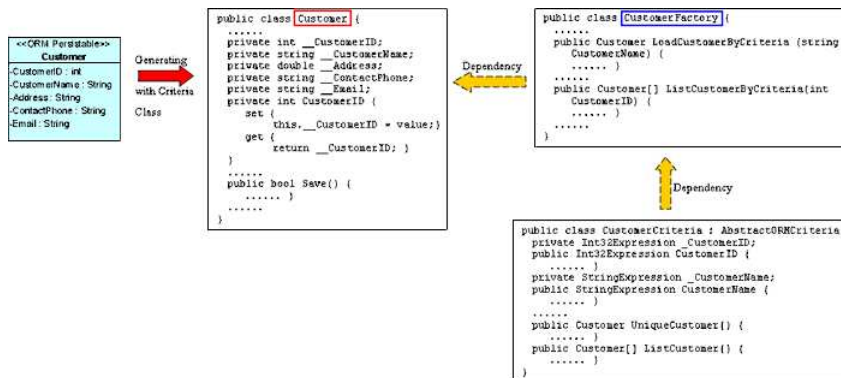


Figure 19.47 - Mapping with Factory and Criteria Class

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.

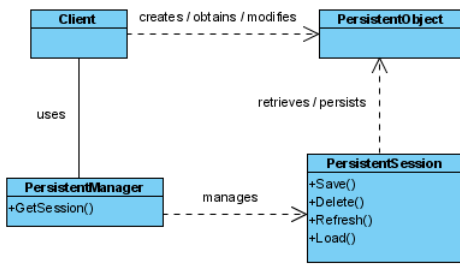



Figure 19.48 - Relation between Client and POJO Classes

 In the above class diagram, the PersistentObject refers to the ORM-Persistable class defined in the class diagram.

Example:

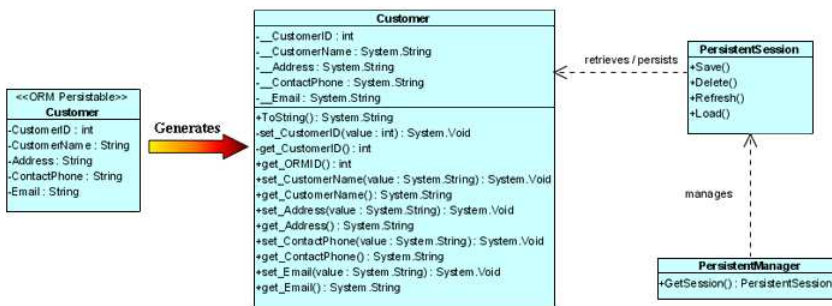


Figure 19.49 - Class Diagram with POJO

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:

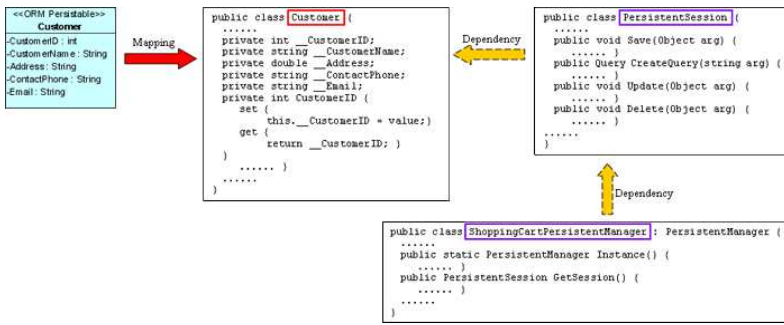


Figure 19.50 - Mapping with POJO

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.

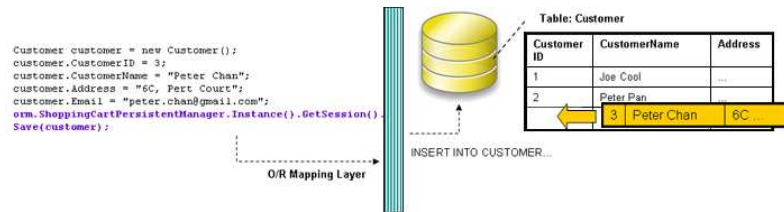


Figure 19.51 - Code for add a record with using POJO

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:

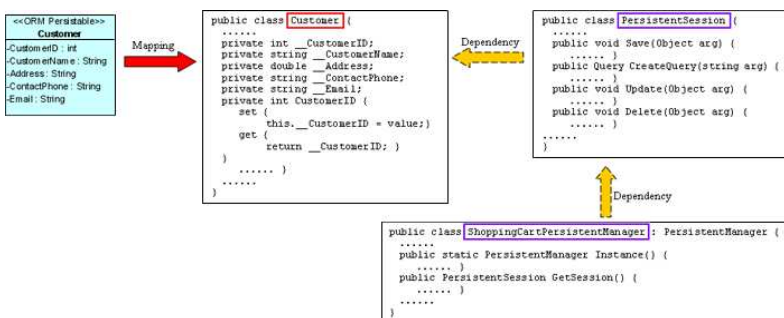


Figure 19.52 - Mapping for create query

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.

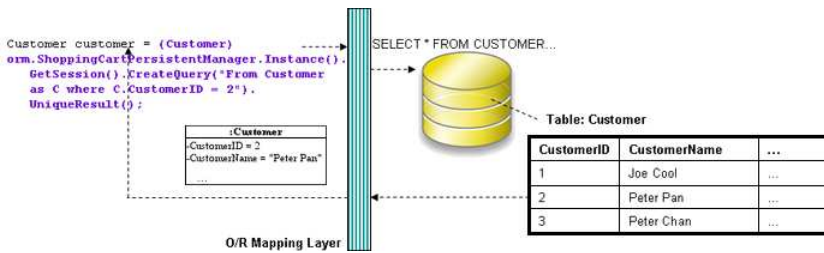


Figure 19.53 - Code for load a record by query

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:

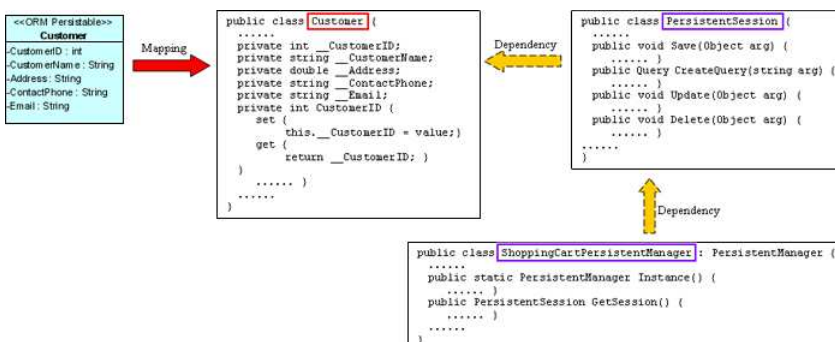


Figure 19.54 - Mapping for update record

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.

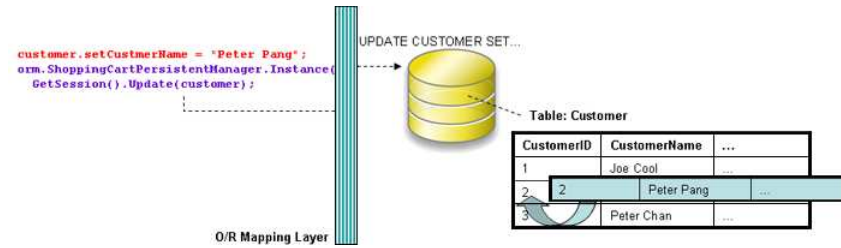


Figure 19.55 - Code for update record

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:

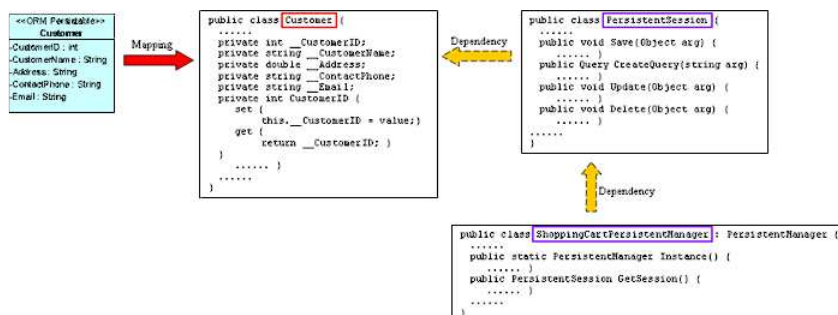


Figure 19.56 - Mapping for delete record

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.

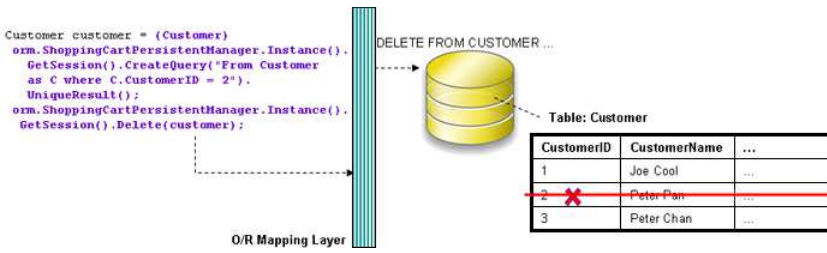


Figure 19.57 - Code for delete record

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:

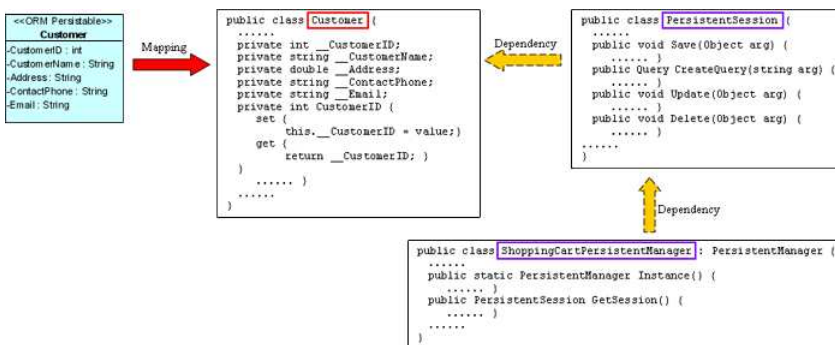


Figure 19.58 - Mapping create query methods

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.

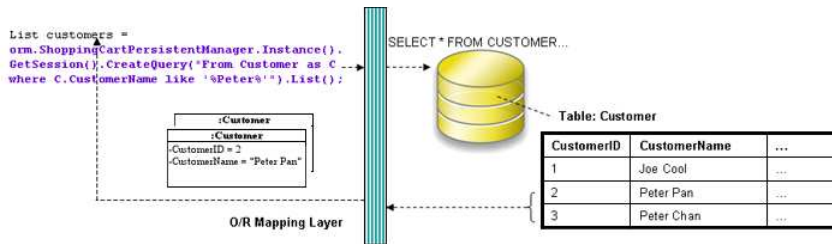


Figure 19.59 - Code for retrieve a list of records

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.

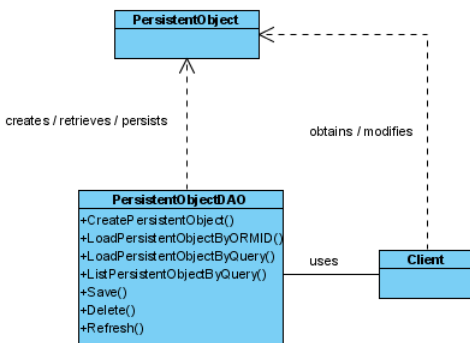



Figure 19.60 - Class Diagram for DAO

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:

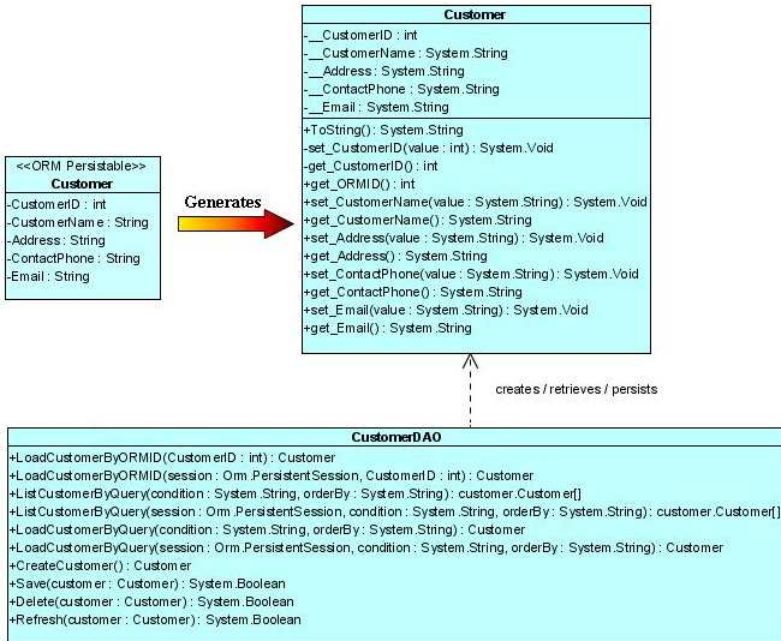


Figure 19.61 - Class Diagram for using DAO

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:

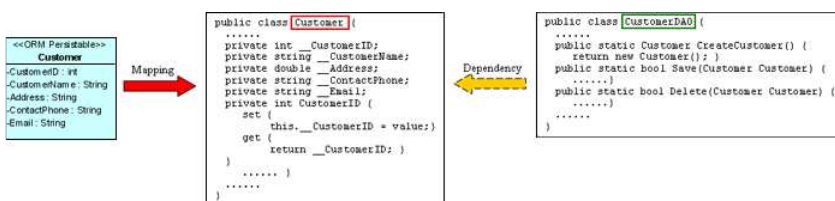



Figure 19.62 - Mapping with DAO

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.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.

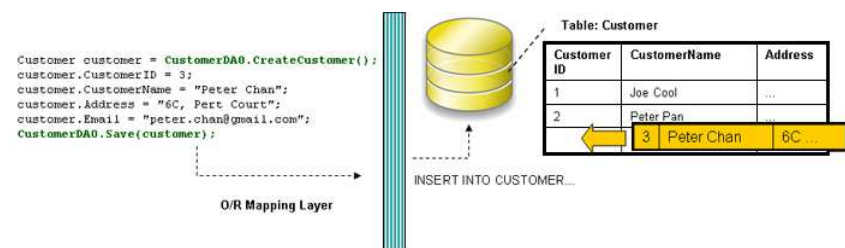


Figure 19.63 - Code for create records by using DAO

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:

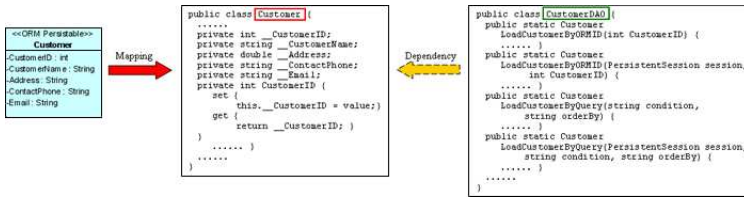


Figure 19.64 - Mapping load and list methods in DAO

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.

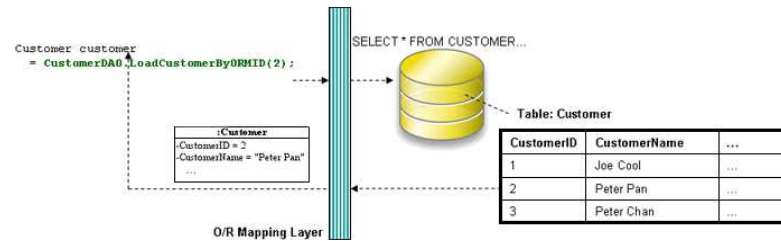


Figure 19.65 - Code for load a records from database

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:

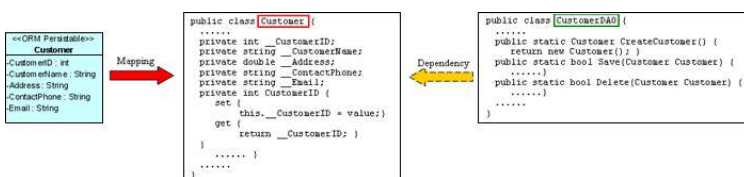


Figure 19.66 - Mapping for update record

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.

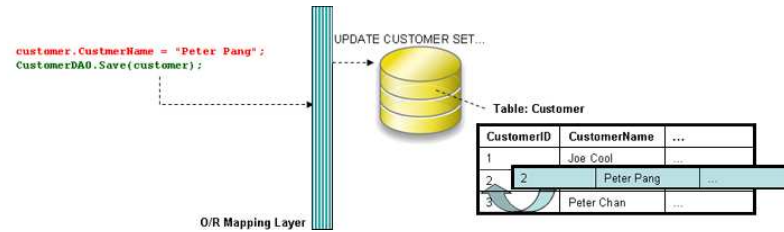


Figure 19.67 - Code for update record

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:

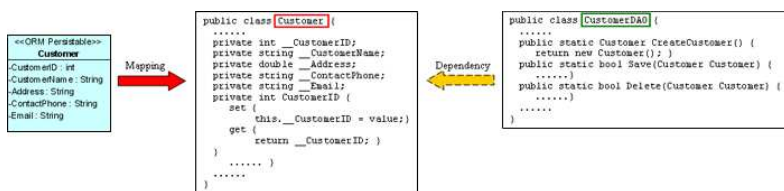


Figure 19.68 - Mapping for delete record

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.

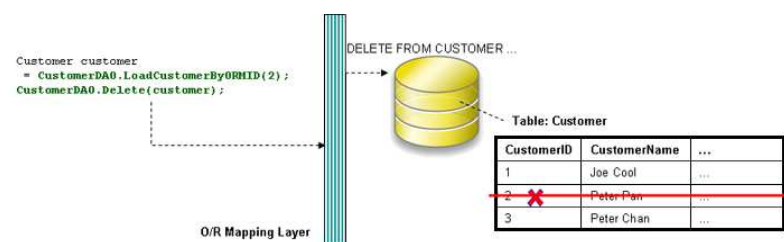


Figure 19.69 - Code for delete record by using DAO

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:

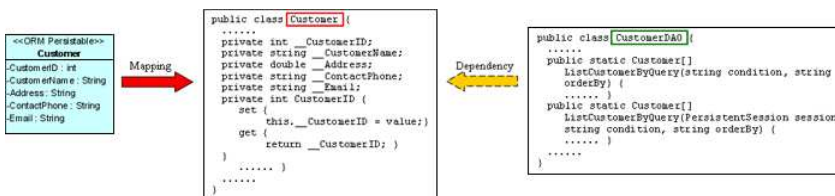


Figure 19.70 - Mapping for list method in DAO

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.

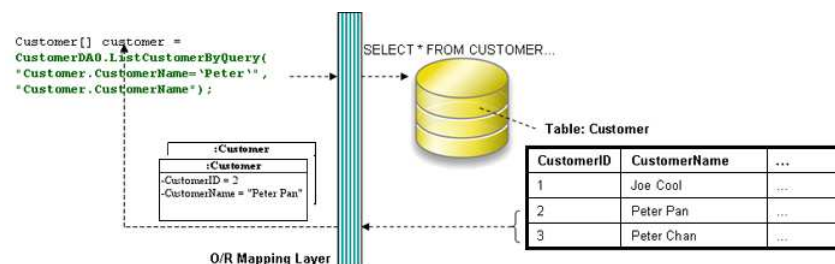


Figure 19.71 - Code for retrieve a list of records by using DAO

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:

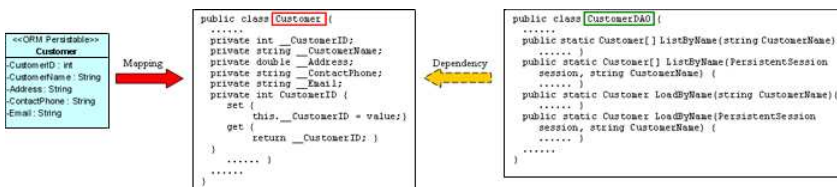


Figure 19.72 - Mapping load and list methods for qualifier

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.

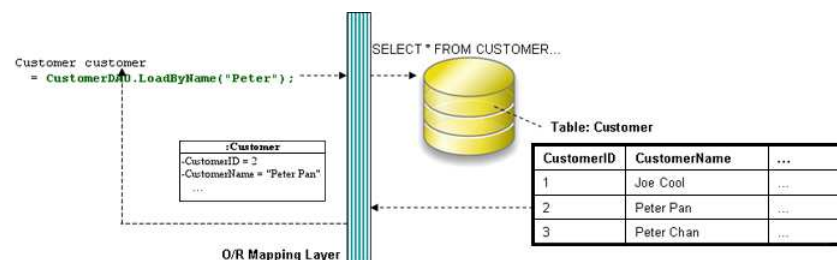


Figure 19.73 - Code for loading a records by qualifier

- 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.

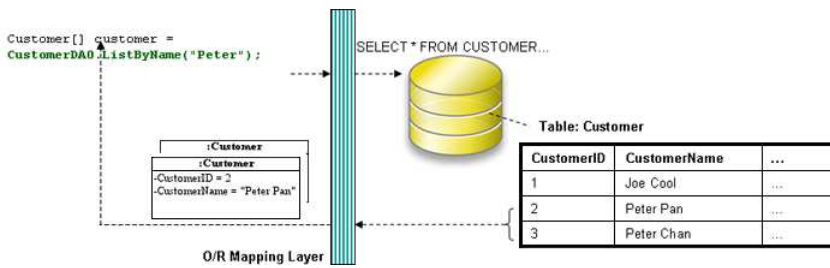


Figure 19.74 - Code for retrieving a list of record by qualifier

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:

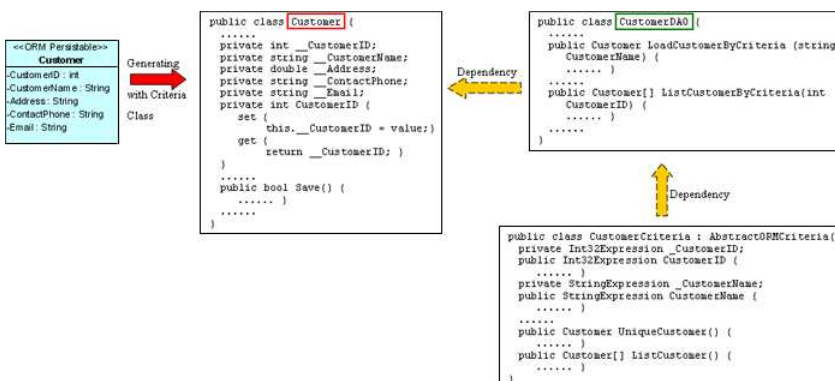


Figure 19.75 - Mapping with criteria class

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.

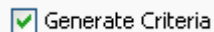


Figure 19.76 - Generate criteria checkbox

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:

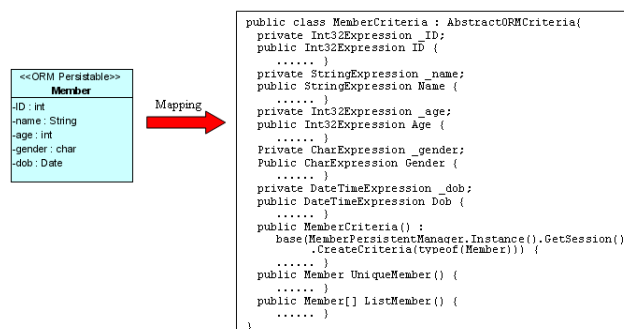


Figure 19.77 - Mapping with Criteria Class



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; false = 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.

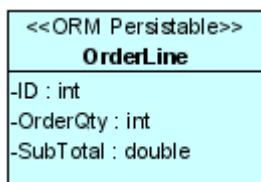


Figure 19.78 - ORM Persistable Class

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".

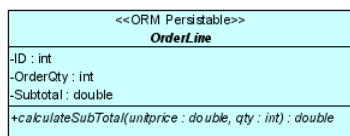


Figure 19.79 - ORM Persistable Class with operation



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.

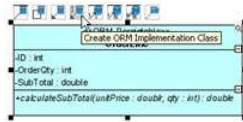


Figure 19.80 - Create ORM Implementation Class resource-centric

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.




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 OrderLineImpl : 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 OrderLineImpl : 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.

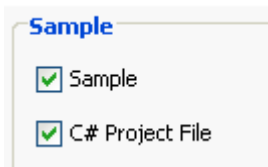


Figure 19.82 - Generate sample options

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:

```
<<ORM Persistable>>
Customer
-CustomerId : int
-CustomerName : String
-Address : String
-ContactPhone : String
-Email : String
```

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.Instance());
    }
}

```

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 generated accordingly.

For 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.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.ShoppingCartPersistenetmanager
                .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.

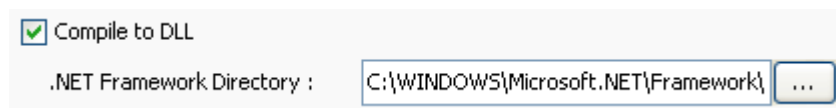


Figure 19.84 - Compile to DLL options

- 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.