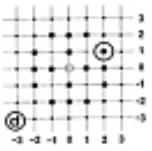
Lattice.DataMapper™
Developer Guide

A Persistence Framework for .NET

Lattice Business Software International, Inc.

www.latticesoft.com



Lattice Business Software International, Inc.

www.latticesoft.com

Table of Contents

Introduction

Overview

This guide provides information on configuring and working with Lattice.DataMapper.

Audience

The intended audience for this guide is .NET and database application developers.

Document organization

This document is divided into the following chapters:

- Chapter 1 provides an overview of the Lattice.DataMapper.
- Chapter 2 provides information on configuring Lattice.DataMapper.
- Chapter 3 provides information on working with Lattice.DataMapper.

Chapter 1. Overview of Lattice.DataMapper

Lattice.DataMapper is a .NET based object persistence library for relational databases. Lattice.DataMapper bridges the gap between objects and data source and allows the developer to work at the object level with little knowledge of the data source.

This chapter provides an overview of Lattice.DataMapper, including:

- What is Lattice.DataMapper?
- What can Lattice.DataMapper do?
- Architecture overview of Lattice.DataMapper.

1. 1 What is the Lattice.DataMapper?

Lattice.DataMapper is a .NET based object persistence library for relational databases. Lattice.DataMapper bridges the gap between objects and data source and allows the developer to work at the object level with little knowledge of the data source. In addition, Lattice.DataMapper shields your business logic from data source schema changes and data source provider changes which benefits maintainability.

Lattice.DataMapper is a tool that maps .NET objects to and from an underlying relational database using external XML configuration and mapping files. The data access is externalized in XML and can include both SQL queries and stored procedure calls. An API is provided to then dynamically store and retrieve .NET objects using the mapping definitions.

Lattice.DataMapper also contains a set of the similar methods in the Enterprise Library Data Access Application Block(DAAB) that save developers from having to write their own methods for common database functions.

Simplicity is the biggest advantage of Lattice.DataMapper over other frameworks and object relational mapping tools. You need only be familiar with XML, stored procedures and SQL.

1. 2 What Lattice.DataMapper Can Do

The Lattice.DataMapper framework will help you to significantly reduce the amount of .net code that you normally need to access a relation database. Simplicity is the biggest advantage of Lattice.DataMapper over other frameworks and object relational mapping (O/R Mapper) tools. To use Lattice.DataMapper you need only be familiar with XML, stored procedures and SQL. Lattice.DataMapper provides the platform and flexibility to build your own architecture and you, the developer, have the full power of both stored procedures and SQL at your fingertips.

Features:

- Support for different data providers and multiple data sources.
- Support object pre-fetch(one-to-one, one-to-many, many-to-many relationship)
- Queries are batched and run as one transaction
- Batch job can be run asynchronously at scheduled time as separate thread
- Cache can be set up per query (supports cache database dependency)
- Support nullable value types using null-value attribute
- Logging and error handling using Log4net framework
- Support distributed transactions without deriving from the ServiceComponent
- Non-intrusive O/R mapping, there is no PersistenceObject base class
- XML file based configuration file to define available data providers and data sources.
- XML based mapping files to define O/R mapping
- XML/XSLT template based tool to auto-generate stored procedures and business entities

Benefits:

- Minimal time spending on software designing, developing and testing that results in profit increase.
- Ease of system functionality modification in ever-changing business environment.
- Easy to use and has a low learning curve
- Enables to easily and fast create multi-tiered applications of corporate scale.
- Full-featured object-oriented approach with all inherent qualities. The developers use real objects and their attributes, and common language terms rather than tables and database entries.
- Possibility to conduct module development of a complex product.
- Possibility to work in heterogeneous distributed systems and interact with legacy systems.

1. 3 Architecture overview of Lattice.DataMapper

1.3.1 N-Tier Application Architecture

Figure 1 shows n-tier application architecture that your programmers should follow when coding their application. The n-tier application architecture is based on the Layer Pattern, the basic idea is that a class within a given layer may interact with other classes in that layer or with classes in an adjacent layer. By layering your source code in this manner you make it easier to maintain and to enhance because the coupling within your applications greatly reduced.

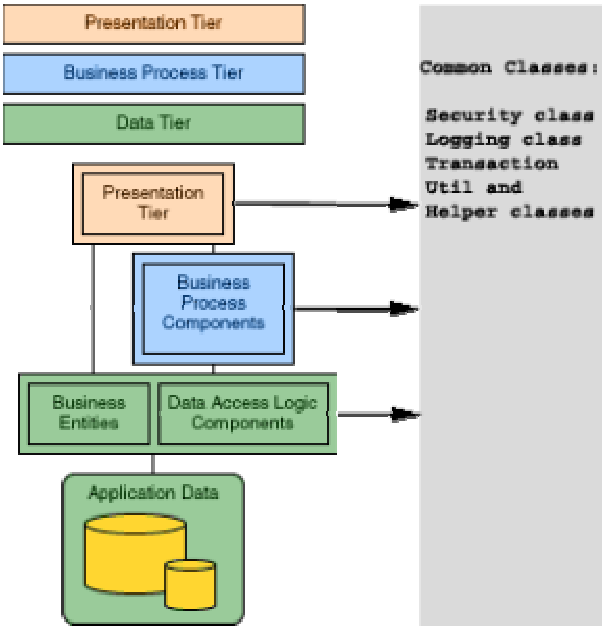


Figure 1.N-tier application architecture

Presentation layer. The layer where the GUI logic of the application is located and generated (if applicable). No business logic functionality is placed in this tier.

Business logic layer. The layer where the logic of the application is placed. This layer is used by the GUI tier to retrieve data to display / for the user to work with, and to store data in the application's database. The GUI tier doesn't know how that data is stored, the business logic tier decides that, by calling services of the data-access tier. As the GUI tier, the business logic tier isn't aware of the concept 'database', it just uses the data-access tier to store/retrieve/update/remove data.

Data Service layer. The Layer, which handles the direct database activity and provides data-access services to the business logic tier. No other tier accesses the database directly and the purpose of the data-access tier is to provide a black-box, database independent interface to the database.

1.3.2 O/R Mapping

Mapping objects to relational database has been done for a long time, but nobody has come up with a perfect solution. The core issue is that the object domain and relational domain are almost orthogonal and is often referred to as the Object – Relational impedance mismatch. Although it is impossible to completely eliminate this impedance mismatch, you can minimize it by using a robust persistence layer. A persistence layer isolates the developer from the details of implementing persistence and protects the application from changes. The persistence layer hides persistence from the developer so that effort can be spent on understanding the domains rather than in making objects persistence.

Reading and writing objects to the database requires basic create, read, update and delete operations. Regardless of the way you implement the persistence layer, it will need to support CRUD (create, read, update, and delete) operations. Basically, there are two ways to implement a persistence layer.

1. Use a PersistenceObject base class. Subclass each domain object from an abstract PersistenceObject. Each object would then inherit how to perform the necessary CRUD operations. This approach is easy to implement and is a good candidates for automatic code generation.

2. Use a Broker that can read or write domain objects to or from the database. The broker must know the format of each domain object, and dynamic generates the SQL to read or write it. This choice separates the database code from the domain object class. It is solution that scales best, though it requires a lot of infrastructure.

These two approaches often create other problems, such as:

- Need a PersistenceObject base class for first approach
- Dynamic generated SQL is not secure and not easy troubleshooting
- Legacy database problems for second approach

Lattice.DataMapper persistence framework takes a simple and flexible approach to map objects to relational database. Lattice.DataMapper use a broker that reads or write objects to or from database without using dynamic generated SQL statement, instead Lattice.DataMapper uses stored procedures or static SQL statement to map objects to relational database.

Lattice.DataMapper provides a very simple framework for using one or more external XML mapping files to map objects to stored procedures or SQL statement. Lattice.DataMapper are configured using a central XML configuration file, which provides configuration details for DataSources, DataProviders and path of mapping file. Figure 2 illustrates how it works:

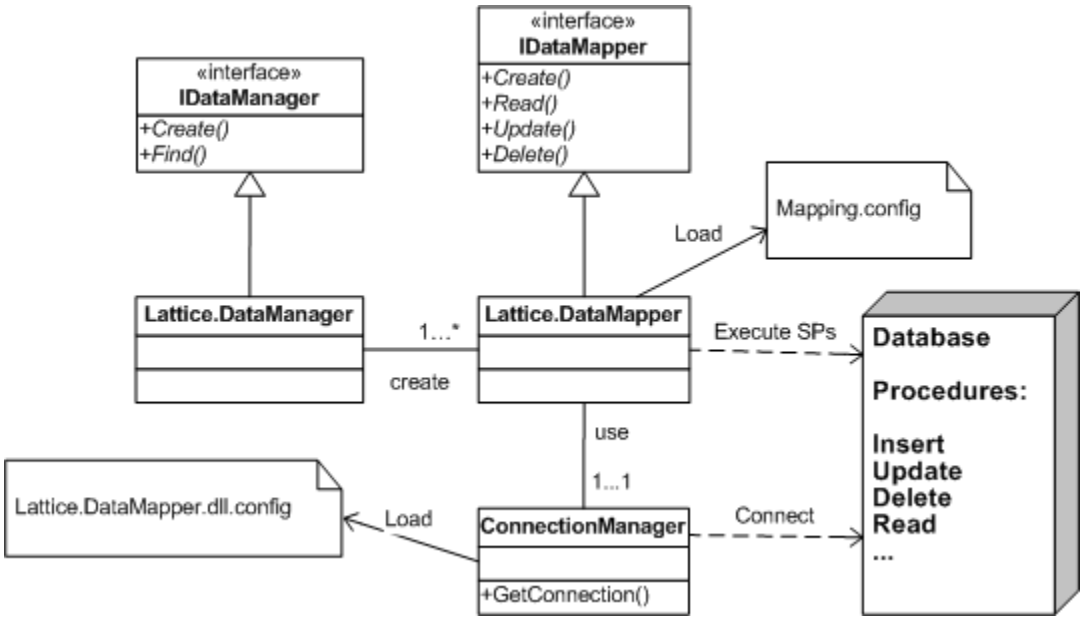


Figure 2.Lattice.DataMapper persistence framework

Chapter 2. Get Started

Before You Begin

Download Lattice.DataMapper persistence framework from www.latticesoft.com.

This chapter provides information on configuring Lattice.DataMapper, including:

- Prerequisites
- Configure the Lattice.DataMapper

2. 1 Prerequisites

Ensure the following prerequisites are met prior to installing the Lattice.DataMapper:

- Microsoft .NET framework 1.1 is required. Download here.
- MDAC 2.7 or later is required. Download here.
- Windows XP Pro or Windows Server 2003 for Lattice.DataMapper Transaction Service
- SqlServer 7/2000/MSDE (using SqlClient)
- Oracle 8i/9i and Oracle client 9.2 (using OracleClient)

The .NET framework and MDAC component both are free download from Microsoft.

2. 2 Configure the Lattice.DataMapper

To get started, you'll need to download the Lattice.DataMapper persistence framework for .NET at http://www.latticesoft.com. Reference the assembly in your project. The next step will be to add the appropriate configuration settings to your application's config file to tell Lattice.DataMapper where and what your data store is as well as which Data Providers will be used.

2.2.1 Lattice.DataMapper.dll.config configuration file

Lattice.DataMapper uses an XML based configuration file, Lattice.DataMapper.dll.config, to define the actual database connections. These definitions include the fairly standard <dataProviders> and <dataSources> elements to define the type of the database, the connection string and the path of mapping file. While any ADO.NET compliant provider can be used, the product ships with provider definitions that include the following providers:

- SqlServer
- OleDb
- Oracle

Lattice.DataMapper.dll.config file shall always be put with Lattice.DataMapper.dll assembly, for example, in your ASP.NET bin folder.

Here's what one might look like:

```
<?xml version="1.0" encoding="utf-8" ?>
<dataSettings>
	<dataProviders>
		<dataProvider name="SqlClient"
			connectionType="System.Data.SqlClient.SqlConnection, System.Data,
Version=1.0.3300.0, Culture=neutr€u108 ?, PublicKeyToken=b77a5c561934e089"
			/>
		<dataProvider name="OleDb"
			connectionType="System.Data.OleDb.OleDbConnection, System.Data,
Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
			/>
		<dataProvider name="OracleClient"
			connectionType="System.Data.OracleClient.OracleConnection,
System.Data.OracleClient, Version=1.0.5000.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
			/>
	</dataProviders>
	<dataSources>
		<dataSource name="SQLPortal" default="true"
			provider="SqlClient"
			connectionString="Server=(local);Database=lattice;User
ID=sa;Password=julie"
			dataMappingDir="/Map/SQLServer/"
			/>
		<dataSource name="OraclePortal"
			provider="OracleClient"
			connectionString="Data Source=lattice; User ID=sa; Password=oracle"
			dataMappingDir="/Map/Oracle/"
			/>
	</dataSources>
</dataSettings>
```

The example above uses either a SqlClient provider or an OracleClient provider, connects to the lattice database on localhost, and uses the supplied username and password. The example above set the path of mapping file in the /Map/SQLServer folder for SQL Server and in the /Map/Oracle/ folder for Oracle under your application root folder.

2.2.2 Lattice.log4net file

Lattice.DataMapper persistence framework uses log4net logging framework.log4net is a tool to help the programmer output log statements to a variety of output targets, such as SQL Server, Oracle, MS Access, file system, application event log, email etc..

One of the most useful features we found when logging in services is the ability to set logging levels, the following levels are defined in order of increasing priority:

- ALL
- DEBUG
- INFO
- WARN
- ERROR
- FATAL
- OFF

Another very useful feature is the ability to very easily create a rolling-log logging system, where logs are "rolled over" when the reach a configured size. The number of log files that are retained is also configurable, making this is an easy way to log quite a bit of information out to files without being concerned about filling up the disk.

You can put Lattice.Log4Net configuration file either in the bin folder with Lattice.DataMapper.dll assembly or you can configure where to find it in your web.config file.

Here is layout of Lattice.log4net configuration file:

```
<?xml version="1.0" encoding="utf-8" ?>
<log4net>
  <appender name="FileAppender" type="log4net.Appender.RollingFileAppender">
            <param name="File" value="Lattice.log" />
            <param name="AppendToFile" value="true" />
            <layout type="log4net.Layout.PatternLayout">
                        <param name="ConversionPattern" value="%d [%t] %-5p %c [%x] - %m%n" />
            </layout>
  </appender>

  <!-- Set root logger level to DEBUG  -->
  <root>
            <level value="DEBUG" />
            <appender-ref ref="FileAppender" />
  </root>
</log4net>
```

This particular configuration sets the RollingFileAppender to write to a file called Lattice.log. The configuration above prints a line that is similar to the one shows here in the log file:

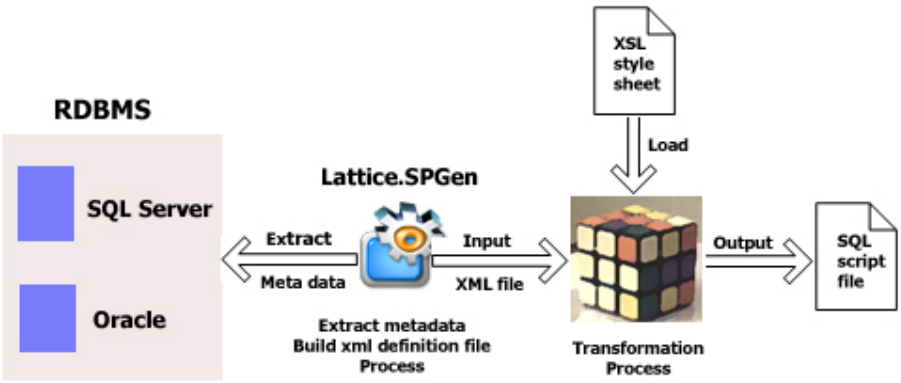
```
2004-08-18 20:37:49,653 [1704] INFO  Lattice.Data.ConnectionManager [] - Application
[ConnectionManager] Start
```


2.3 Lattice.SPGen Code Generator

Lattice.SPGen is a XML/XSLT template based code generator that supports Microsoft SQL Server, Oracle and IBM DB2 UDB. Lattice.SPGen will process each table and view in the database and generate CRUD stored procedures, business entities(c# and vb.net) and mapping files per table. Download now!

Below is how it works:

- Collect metadata from different datasource(Oracle, DB2, SQL Server)
- Build xml metadata input definition file
- Generate stored procedures, data transfer objects, and mapping files using XSLT with different XSL style sheet.



Chapter 3. Working with Lattice.DataMapper

Lattice.DataMapper persistence framework takes a simple and flexible approach to map objects to relational database. Lattice.DataMapper use a broker that reads or write objects to or from database using stored procedures or static SQL statement to map objects to relational database.

Lattice.DataMapper also contains a set of the similar methods in the Enterprise Library Data Access Application Block(DAAB) that save developers from having to write their own methods for common database functions.

This chapter provides information on working with Lattice.DataMapper, including:

- Mapping file
- Classes
- Object Persistence
- Object Query
- Batch
- Cache
- Nullable Value Types
- Transaction Service
- Logging
- Unit test
- Enterprise Library Data Access Application Block

3.1 Mapping File

Here is layout of Mapping.config file:

```
<?xml version="1.0" encoding="utf-8" ?>
<commands parameterPrefix="@">
        <command name="getEmployees" type="storedprocedure" returnType="Collection">
                <commandtext>ReadAll_Employees</commandtext>
                <class name="Lattice.BLL.Employee" assemblyName="Lattice.DataMapper.BLL" >
                        <property name="EmployeeID" column = "EmployeeID" />
                        <property name="Name" column="Name" />
                        <property name="Title" column = "Title" />
                        <property name="HireDate" column = "HireDate" />
                        <property name="Notes" column = "Notes" />
                </class>
        </command>
        <command name="deleteCustomer" type="storedprocedure"
                 operation="Delete" objectType="Lattice.BLL.Customer"
                 parameterName= "@CustomerID" >

                <commandtext>Delete_Employees</commandtext>
        </command>
</commands>
```

Explanation of attributes:

Commands node:

parameterPrefix: Prefix used for parameters. @ is for SQL Server

Command node

name: stored procedure's alias name
type: either "storedprocedure" or "text"
returnType: this is an option attribute, can be one of the below four types:

- Collection: return an ArrayList collection object which hold business entity object you define in class node
- Single: return a single business entity object you define in class node
- Output: return a single business entity object using value of stored procedure's output parameter

operation: Insert/Update/Delete/Load operation type

CommandText node: stored procedure's real name or static SQL statement

Parameter node:

name: parameter's name (case sensitive)
fieldname: matched business entity property name
direction: stored procedure parameter direction.
dbType: parameter DbType

Class node:

name: returned business entity name
assemblyName: Assembly which contains business entity

Property node:

name: business entity's property name
column: matched column's alias name in stored procedure's or SQL statement select clause.

3.2 Lattice.DataMapper Classes

A system of Lattice.DataMapper classes is quite easy and convenient to use. In the Lattice.DataMapper based application only three basic classes are operated:

1. *DataManager* class: A factory class to create a DataMapper instance
2. *Criteria class:* A class handle ADO.NET command's parameters
3. *DataMapper* class: A core class responsible for read and write objects to or from database

DataManager Class

DataManager class is a factory class associating DataMapper class with its corresponding XML mapping file. DataManager class follows the Singleton design pattern in that there is only one instance of it in the object space. A DataManager is used to get a DataMapper instance. There is a one-to-many relationship between the DataManager and the DataMapper: A DataManager can create and manage many DataMapper instances and even implement pooling of DataMapper instances.

```
IDataMapper mapper = DataManager.Instance.Create("Mapping.config");
```

Criteria Class

Criteria class is a helper class for ADO.NET Command; it can be used to configure CommandType, CommandText and IDataParameter.

```
Criteria criteria = new Criteria();
criteria.CommandName = "getCustomersByCity";
criteria.AddParameter("@City", "Chicago" );

IList customers = (IList)mapper.Find( criteria );
```


DataMapper Class

In many ways the DataMapper class is the key to the persistence layer. This class implements the CRUD(create, read, update, delete) functionality and map objects to relational database using stored procedures or static SQL statement based on information in the mapping file. It acquires the database connection through the ConnectionManager class.

Below is public methods provided by DataMapper class:

```
void Save( string commandName, object o);
void Create( object o );
void Update( object o );
void Delete( Type t, object value );
void Delete( Type t, Criteria criteria );
object Read( string commandName, object o);
object Find( Criteria criteria);
object Load( Type t, object value);
object Load( Type t, Criteria criteria );
object Load( Criteria criteria );

IDataReader ExecuteReader( string commandText, CommandType commandType );
IDataReader ExecuteReader( Criteria criteria );
DataSet ExecuteDataSet( string commandText, CommandType commandType );
DataSet ExecuteDataSet( Criteria criteria );
object ExecuteScalar( Criteria criteria );
object ExecuteScalar( string commandText, CommandType commandType );
object ExecuteNonQuery( string commandText, CommandType commandType );
object ExecuteNonQuery( Criteria criteria );

void Persist();
void Persist( DateTime dueTime );
void Persist( TimeSpan timeSpan );
```


3.3 Object Persistence

Creating an Object

Here is layout of xml mapping file to create a new employee record in database:

```
<command name="addEmployee" type="storedprocedure"
         operation="Insert" objectType="UnitTest.Employee">

        <commandtext>Insert_Employee</commandtext>
        <parameter name="FirstName" fieldName="FirstName" direction="Input" />
        <parameter name="LastName" fieldName="LastName" direction="Input" />
        <parameter name="Title" fieldName="Title" direction="Input" />
        <parameter name="HireDate" fieldName="HireDate" direction="Input" />
        <parameter name="Notes" fieldName="Notes" direction="Input" />
        <parameter name="EmployeeID" fieldName="EmployeeID" direction="OutPut"
                  direction="OutPut" identity="true"/>
</command>
```

Based on above mapping file, here is the code snippet to insert a new employee record into database:

```
IDataMapper mapper = DataManager.Instance.Create("Mapping.config", "Northwind");

Employee employee = new Employee();

employee.FirstName = “firstName”;
employee.LastName = “lastName”;
employee.Title = “title”;
employee.hireDate = “1/1/2000”;
employee.notes = “this is a test”;

mapper.Create( employee );
mapper.Persist();

int employeeId = employee.EmployeeId;
```


Updating an Object

Here is layout of xml mapping file to update an employee record in database:

```
<command name="updateEmployee" type="storedprocedure"
         operation="Update" objectType="UnitTest.Employee">

       <commandtext>Update_Employee</commandtext>
       <parameter name="FirstName" fieldName="FirstName" direction="Input" />
       <parameter name="LastName" fieldName="LastName" direction="Input" />
       <parameter name="Title" fieldName="Title" direction="Input" />
       <parameter name="HireDate" fieldName="HireDate" direction="Input" />
       <parameter name="Notes" fieldName="Notes" direction="Input" />
       <parameter name="EmployeeID" fieldName="EmployeeID" direction="Input" />
</command>
```

Here is the code snippet to update an employee record in database based on EmployeeID :

```
IDataMapper mapper = DataManager.Instance.Create("Mapping.config", "Northwind");

Employee employee = new Employee();

employee.EmployeeID = employeeID;
employee.FirstName = “firstName”;
employee.LastName = “lastName”;
employee.Title = “title”;
employee.hireDate = “1/1/2000”;
employee.notes = “this is a test”;

mapper.update(employee);
mapper.Persist();
```


Deleting an Object

Here is layout of xml mapping file to delete an employee record from database:

```
<command name="deleteCustomer" type="storedprocedures"
        operation="Delete" objectType="UnitTest.Employee"
        parameterName= "@EmployeeID" >

       <commandtext>Delete_Employee</commandtext>

</command>
```

Here is the code snippet to delete an employee record in database based on EmployeeID:

```
IDataMapper mapper = DataManager.Instance.Create("Mapping.config", "Northwind");
mapper.delete( typeof(Employee), employeeId );
mapper.Persist();
```

3.4 Object Query

Loading single object

```
<command name="loadEmployee" type="storedprocedure"
         operation="load" objectType="UnitTest.Employee"
         parameterName= "@EmployeeID" returnType="Single">

        <commandtext>Read_Employee</commandtext>

        <class name="Lattice.BLL.Employee" assemblyName="Lattice.DataMapper.BLL" >
                <property name="EmployeeID" column = "EmployeeID" />
                <property name="Name" column="Name" />
                <property name="Title" column = "Title" />
                <property name="HireDate" column = "HireDate" />
                <property name="Notes" column = "Notes" />
        </class>
</command>
```

Below is the code snippet to get a single employee from database based on above xml mapping file:

```
Employee employee = (Employee)mapper.Load( typeof(Employee), employeeId ) ;
```

Finding objects

```
<command name="getEmployees" type="storedprocedure" returnType="Collection">
        <commandtext>ReadAll_Employees</commandtext>
        <class name="Lattice.BLL.Employee" assemblyName="Lattice.DataMapper.BLL" >
                <property name="EmployeeID" column = "EmployeeID" />
                <property name="Name" column="Name" />
                <property name="Title" column = "Title" />
                <property name="HireDate" column = "HireDate" />
                <property name="Notes" column = "Notes" />
        </class>
</command>
```

Below is the code snippet to get all employees from database and return as ArrayList collection which contains Employee object based on above xml mapping file:

```
Criteria criteria = new Criteria();
criteria.CommandName = "getEmployees";

IList employees = (IList)mapper.Find( criteria );
```

You can navigate the employees collection as below:

```
IEnumerator E = employees.GetEnumerator();

while (E.MoveNext())
{
        Employee employee = (Employee)E.Current;
        string name = employee.Name;
        string title = employee.Title;
        ...
```



```
}
```

Association Mapping

Association mappings are the often most difficult thing to get right. Lattice.DataMapper supports one-to-one, one-to-many and many-to-many relationship. In this section we'll go through each case one by one, starting with one-to-one, one-to-many mappings, and then considering the many-to-many cases.

One- to-One Mapping

A one-to-one association to another persistent class is declared using a one-to-one attribute.

one-to-one association on a foreign key.

```
<property name="ProductItem" subclass="UnitTest.ProductItem" fk="ProductID"
          relationship="One-To-One" type="subclass" />

<subclass name="UnitTest.ProductItem" assemblyName="UnitTest" >
       <property name="ProductId" column="ProductID" type="subclass" />
       <property name="ProductName" column="ProductName" type="subclass" />
</subclass>
```

one-to-one association on a primary key.

```
<property name="ProductItem" subclass="UnitTest.ProductItem" pk="ProductID"
          relationship="One-To-One" type="subclass" />
```

One to many

An ordinary association to another persistent class is declared using a one-to-many attribute.
A *one to many association* links the tables of two classes via a foreign key.

```
<property name="LineItems" subclass="UnitTest.OrderDetail" fk="OrderID"
          relationship="One-To-Many" type="subclass"/>

<subclass name="UnitTest.OrderDetail" assemblyName="UnitTest" >
       <property name="UnitPrice" column="UnitPrice" type="subclass" />
       <property name="Quantity" column="Quantity" type="subclass" />
</subclass>
```

Many to many

A many-to-many association between persistent classes is declared using a many-to-many attribute.

```
<class name="UnitTest.User" assemblyName="UnitTest" pk="UserID">
       <property name="UserId" column="UserID" />
       <property name="Name" column="Name" />
       <property name="Roles" subclass="UnitTest.Role" pk="RoleID"
                 relationship="Many-To-Many" lazyload="false" type="root" />
</class>

<subclass name="UnitTest.Role" assemblyName="UnitTest" >
       <property name="RoleId" column="RoleID" type="subclass" />
       <property name="Name" column="RoleName" type="subclass" />
       <property name="Users" subclass="UnitTest.User" pk="UserID"
                 relationship="Many-To-Many" lazyload ="false" type="subclass" />
</subclass>
```


3.5 Batch

Lattice.DataMapper persistence framework builds up all sql statements as a “Batch Command”, and then executes them in one final call at the end of process, and the end method can open the connection, call line up methods in one transaction. You even can run this batch job asynchronously using separate threads at certain time you scheduled.

Below code snippet show how to run jobs as batch:

```
Customer customer = new Customer();
customer.CustomerID = "AAAAA";
customer.CompanyName = "Lattice Business Software";

mapper.Create( customer );

customer.CompanyName = "Just For Fun, Inc.";

mapper.Update( customer );

mapper.Delete( typeof(Customer), "AAAAA" );

mapper.Persist();
```

And below code snippet run batch asynchronously after 30 seconds:

```
Customer customer = new Customer();
customer.CustomerID = "AAAAA";
customer.CompanyName = "Lattice Business Software";

mapper.Create( customer );

customer.CompanyName = "Lattice Software";

mapper.Update( customer );

mapper.Delete( typeof(Customer), "AAAAA" );

//30 seconds later
TimeSpan span = new TimeSpan(0,0,30);
mapper.Persist( span );
```


3.6 Cache

Lattice.DataMapper supports cache per query, and you can turn on/off cache for each stored procedure call at the mapping file.

Below show you how to configure cache at the mapping file:

```
<!-- Get All Customers and Cache it -->
<command name="getCustomers" type="storedprocedures" returnType="Collection"
        cache="true,-1,h" cacheDependency="C:\\cache_customers.txt">
<commandtext>Customers_List</commandtext>
```

cache attribute:

- true/false: turn on/off cache
- -1: cache never expire
- 10,h: cache expire after 10 hours(s:second; m:minute; h:hour; d:day;)

cacheDependency attribute:

Set up database dependency on trigger update file (C:\cache_customers.txt)

You need create a trigger sending notification to above file whenever table records changed.

```
CREATE TRIGGER [UpdateCacheDependency] ON [dbo].[Customers]
FOR INSERT, UPDATE, DELETE
AS
EXEC sp_makewebtask @outputfile = 'C:\cache_customers.txt',
@query = 'SELECT count(*) FROM customers'
```

Lattice.Common assesmbly also provides thread-safe Cache class you can use at your applications, below is methods Cache class provides:

```
public static object GetCache( string cacheKey )
public static void SetCache( string cacheKey, object obj )
public static void SetCache( string cacheKey, object obj,
System.Web.Caching.CacheDependency dependency )
public static void SetCache( string cacheKey, object obj,
System.Web.Caching.CacheDependency dependency, DateTime absoluteExpiration, TimeSpan
slidingExpiration )
public static object RemoveCache( string cacheKey )
```

3.7 Nullable Value Type

Lattice.DataMapper supports nullable value types using null-value attribute at the mapping file, and you can configure nullable value for each stored procedure call, for float/double value type, you do not need set up nullable value at the mapping file, just assign “NaN” to the property.

```
<command name="createCustomer" type="storedprocedures"
         operation="Insert" objectType="UnitTest.Customer" >
        <commandtext>Customers_Insert</commandtext>

        <parameter name="NumberOfOrders" fieldName="NumberOfOrders"
                   nullValue="-1" direction="input" />

</command>
```

At your code you need assign -1 to NumberOfOrder property:

```
order.NumberOfOrders = -1;
```

If Lattice.DataMapper encounters -1 of property NumberOfOrders, it will set parameter value to DBNull then inserting null value into database.

3.8 Logging

Lattice.DataMapper using Log4Net logging framework, and build a common logging class on top of the log4net, so you do not need to configure log4net in each class. Below is code snippet how to log below five logging levels:

- DEBUG
- INFO
- WARN
- ERROR
- FATAL

```
LatticeLogging.Instance.Log( "Log Info", LatticeLogging.LoggingType.INFO );
LatticeLogging.Instance.Log( "Log Debug", LatticeLogging.LoggingType.DEBUG );
LatticeLogging.Instance.Log( "Log Warn", LatticeLogging.LoggingType.WARN );
LatticeLogging.Instance.Log( "Log Error", LatticeLogging.LoggingType.ERROR );
LatticeLogging.Instance.Log( "Log Fatal", LatticeLogging.LoggingType.FATAL );
```


3.9 Distributed Transaction Service

When saving objects, it is important to allow for objects to be stored in such a way that if some values are not stored properly, previously objects can be rolled back. COM+ Services, referred to as Enterprise Services, provide .NET developers with a powerful set of services for developing robust and scalable server applications. The most used feature of System.EnterpriseServices or COM+ is the distributed transaction support. And the automatic transaction programming model in ES using attributes ([Transaction] and [AutoComplete]) is great and nice but there are problems with the Service Component Approach, if you wanted to use COM+ services from a .NET application, you had to go through the following steps:

- Derive your class from the ServiceComponent class
- Create a strong key and associate that with the assembly
- Register the component with the component service explorer either manually or using the automatic registration feature.

If doing this seems overkill to you, because all you need is a distributed transaction to protect your code/actions and you don't care of any of the others ES features (which are great ones nevertheless) then there is a solution for you: Lattice.DataMapper transaction framework. There are two basic classes:

1. TransactionContextFactory: a class to create a TransactionContext instance
2. TransactionContext: a class provides support for beginning transactions, committing transactions, and rolling back transactions.

Here is an example to use Transaction model:

```
TransactionContext ctx = TransactionContextFactory.GetContext(TransactionLevel.RequiresNew);

try
{
        ctx.BeginTransaction();
        //Call bll or dal methods(db operations)
        //methodA();
        //methodB();
        ctx.Commit();
}
catch ( Exception e )
{
        ctx.Rollback();
        log.Error("Error Transaction: " + e.Message);
}
finally
{
        ctx.Exit();
}
```


3.10 Enterprise Library Data Access Application Block

Lattice.DataMapper also contains a set of the similar methods in the Enterprise Library Data Access Application Block (DAAB) that save developers from having to write their own methods for common database functions.

You need create an IDataMapper object first:

```
IDataMapper mapper = DataManager.Instance.Create();
```

Below code snippet call stored procedure "Customers_GetSingle" and return a DataReader:

```
Criteria criteria = new Criteria();
criteria.CommandName = "Customers_GetSingle";
criteria.CommandType = CommandType.StoredProcedure;
criteria.AddParameter("@CustomerID", "ALFKI" );

IDataReader reader = mapper.ExecuteReader( criteria );
```

Below code snippet call stored procedure "Customers_GetSingle" and return a DataSet:

```
Criteria criteria = new Criteria();
criteria.CommandName = "Customers_GetSingle";
criteria.CommandType = CommandType.StoredProcedure;
criteria.AddParameter("@CustomerID", "ALFKI" );

DataSet ds = mapper.ExecuteDataSet( criteria );
```


3.11 Unit Test

You have a wide variety of tools and techniques to test your source. We use NUnit testing framework for our unit test. NUnit is an open-source testing framework for .NET modeled after similar testing frameworks on the xUnit family, such as JUnit for Java.

NUnit provides a simple way for developers to write unit tests of the .NET classes. The current version is written in C# and relies on attributes rather than inheritance and /or naming conventions to define tests and test suites. The main attributes involved are **TestFixture**, which applies to a class containing test, **SetUp** and **TearDown**, which are run before and after each test, and **Test**.

NUnit also provides a simple graphical user interface that lets you select which assembly you want to test and which set of tests within that assembly you want to run. It then runs all of the tests in the assembly (or namespace or class)selected, displaying a green bar if everything passes and a red bar if any tests failed. Details of each failed test are also displayed, making it very easy to locate the cause of the failure.

Following is the code snippet that establishes a database connection as a test:

```
[Test]
public void isConnectionOpen()
{
        IDbConnection connection = ConnectionManager.GetInstance().GetConnection();
        Assert.IsNotNull(connection);

        connection.Open();
        Assert.AreEqual( ConnectionState.Open, connection.State );
        connection.Close();
}
```

First, we get the default connection based on information in Lattice.DataMapper.dll.config configuration file. Second, the database connection is not considered opened when it is constructed and need to be opened explicitly. After the connection has been opened, we can verify its state by accessing the State property on the connection object; the ConnectionState enumeration defines the common connection states. When we finish with the connection, we use the Close method to close the connection.

