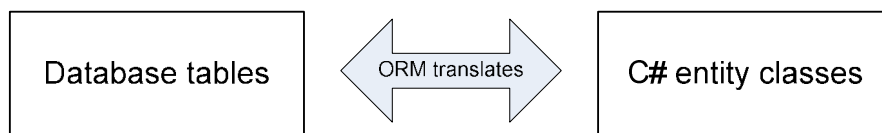


It has always been an interesting discussion how to decrease the time consuming tasks of moving data from a business middle-tier to the database backend. Last year, after looking at different samples, snippets and tools I stumbled upon <http://www.nhibernate.org>.

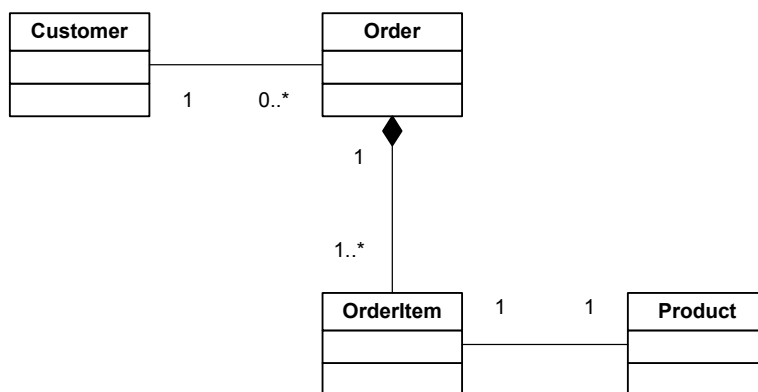
Most of the NHibernate samples I went through on the internet were not complete and were vaguely describing part of the NHibernate framework, many of them providing little to no pieces of code which was not very helpful. My goal is to give a practical example of a business case and the implementation of the business logic and the data access layer using an ORM tool.

The article first describes the business case that will be implemented and how using NHibernate the development effort can be significantly reduced.

Many resources might be found about what ORM is and how it can be applied in commercial applications. The main purpose of ORM tools and technologies is to reduce the development time in moving the information from the business entity classes to the database back and forth. An over simplistic view of ORM is the following diagram:



Let's imagine an e-commerce solution that has several well known business entities.

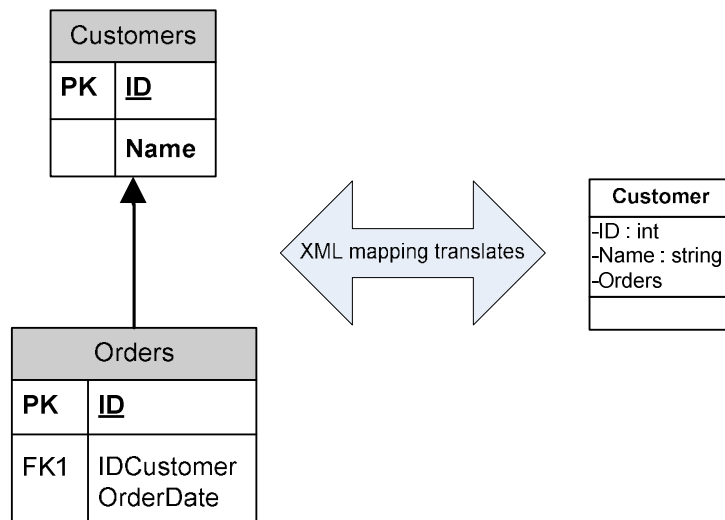


A Customer can have many Orders that can be submitted to the e-commerce site. Every Order can have at least one OrderItem. The OrderItems represent the information in the order with respect to the Products that the Customer has ordered.

No matter which technology we will use for the implementation of the entities, the simple CRUD (create, retrieve, update and delete) operation will take significant part of our development effort. In order to minimize the time needed, we will use the .NET framework, C# and NHibernate. NUnit and Microsoft Enterprise Library will be used for the unit tests.

There are also many discussions lately about TDD (Test Driven Development) and unit tests. While validating the implementation of the CRUD operations in the classes and checking what happens on the database end, we will also reveal the use of the Nunit ([www.nunit.org](http://www.nunit.org)) unit-testing framework.

Let us first start with the basics of ORM. Every such technology needs to make a translation between the database that stores information and the business entities that exists on another tier. For example a table that represents the Customers information will need to be closely designed to the properties which every customer entity has. NHibernate simplifies the mapping of the properties (public members) of a C# class to the columns in a table in the database. Every mapping is described as an XML file with a specific schema. To make things simple we will keep the Customer class with as little properties as possible. Let's have a customer ID and a Name. The database will also need to contain a table called Customers that will have two columns as a start, ID and Name.



The following NHibernate mapping file contains all information that is needed by the framework to provide all CRUD operations on the entity.

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:NHibernate-mapping-2.0" schema="dbo">
  <class name="DemoNHibernate.Entities.Customer, DemoNHibernate"
table="Customers">
    <id name="ID" type="Int32" column="ID" unsaved-value="0">
      <generator class="identity" />
    </id>
    <property name="Name" column="CustomerName" type="String" />

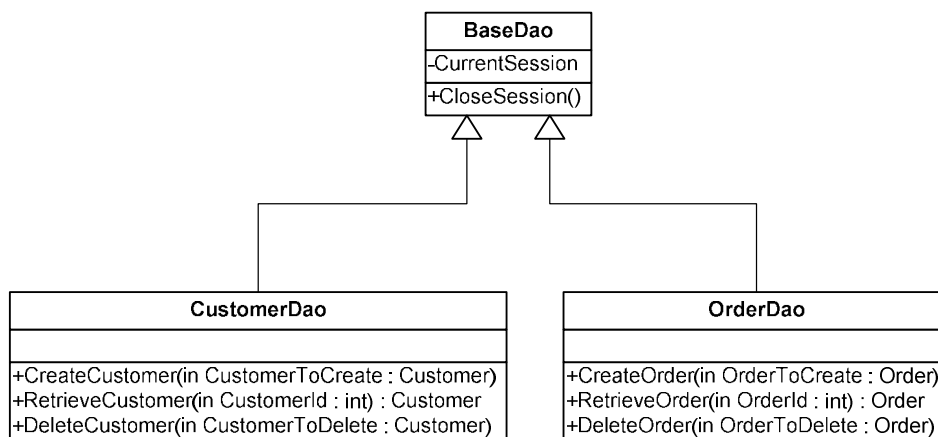
    <bag name="Orders" inverse="true" cascade="all" lazy="true">
      <key column="IDCustomer" />
      <one-to-many class="DemoNHibernate.Entities.Order,
DemoNHibernate" />
    </bag>
  </class>
</hibernate-mapping>
```

```
</class>  
</hibernate-mapping>
```

The mapping starts with the relation between the class that has to be persisted (DemoNHibernate.Entities.Customer from the DemoNHibernate assembly) and the Customers table in the database. It then continues with the mapping between the class properties and the table columns. For example the ID property from the class maps to the ID column in the table. Note the bag tag that describes the one-to-many relationship between the Customer and the Order entities.

One common task will be to add or update class properties. What we will need to do is add/update the column in the table, the XML file and the corresponding property in the C# class. It is a great time redeemer.

Once we have all the classes, the tables in the database and the mappings between the classes and the tables, we will also need to implement the data access layer using NHibernate functionality. Only mapping the property will not do that much. It will not transfer the business information to the database. For that purpose we will use a very common design approach. For every business class that we have we will introduce a corresponding DAO (data access object) class. The implementation of the DAO classes will include NHibernate specific parts that will utilize the ORM framework and will make things really working from end to end.



The set of DAO classes have a common part, which is the handling of the connections to the database using the ISession interface from NHibernate. The BaseDao class manages the connection to the database. All DAO classes are direct inheritors of BaseDao and have the connection to the database already available. Their only purpose is to implement the standard CRUD operations.

Some caveats exists, for example that every XML mapping file has to be compiled into the assembly as an embedded resource. NHibernate uses reflection to extract the resources from the assembly once loaded.

The solution has the following structure:

DemoNHibernate – contains the main console application  
Dao – contains the implementation of the DAO classes  
Entities – contains the implementation of the business entities  
Mappings – contains the NHibernate mapping XML files  
scripts – contains the db script that describes all db objects  
Tests – contains the unit tests for the critical components  
App.config – describes the mandatory NHibernate settings  
dataconfiguration.config – used only by the nunit tests

In order to build the solution you will need to follow the steps:

1. Download and install [NHibernate 1.0](#);
2. Download and install [Microsoft Enterprise Library June 2005](#);
3. Download and install [NUnit 2.2.4](#);
4. Create a new database <database name>;
5. Run the db\_script.sql to create all database tables;
6. Open the solution and rebuild, referencing to the appropriate assemblies;
7. Reconfigure the App.config file, changing the NHibernate settings for the connection to the database;
8. Reconfigure the dataconfiguration.config file, changing the settings for the connection to the database from the unit tests;
9. Rebuild and run the unit tests.
10. Explore the implementations of the entity classes and the DAO classes, review the XML files with the NHibernate mappings.

As a conclusion, the article uses a pretty nice set of tools and technologies. It demonstrates a complete set of business entities and data access layer using ORM, as well as unit testing. It also uses Enterprise Library in the unit tests to connect to the database and check what are the results of the ORM operations and how the changes in the business layer are reflected to the database.

Although reducing the time to implement stored procedures and calling them using standard ADO.NET or Enterprise Library, we are still required to implement the DAO classes which adds a little bit of overhead. Here the experience of the developer will take place in deciding if an ORM tool will be applicable to the current situation and if it will save time or not. For small and less complex projects the approach that I have described is pretty much everything that is needed. For heavier business logic, lots of relations and a complex database, an ORM tool might not be the best choice. I still haven't evaluated the performance impact of using NHibernate, but that will follow in an additional article.

You can get the complete solution from [here](#). The article in PDF format you can get from [here](#).

I would like to thank Peter Skelin, for the review of the article prior to its publishing.