DIPLOMAMUNKA

Papp Zoltán

Debrecen
2008

Debreceni Egyetem
Informatikai Kar

NHibernate
ORM solution for .NET platform

Témavezető:
Kollár Lajos
Egyetemi tanársegéd

Készítette:
Papp Zoltán
Programtervező matematikus

Debrecen
2008

Köszönetnyilvánítás

Ezúton szeretném megköszönni tanáraimnak az egyetemi éveim alatt nyújtott segítségét, szakmai támogatását, állandó tanácsait, építő jellegű kritikáit, melyek nélkül diplomamunka dolgozatom valószínűleg nem készült volna el. A munka során jelentkező szakmai kérdésekben segítségemre volt témavezető tanárom Kollár Lajos, Esa Salmikangas és Juha Peltomäki, a Jyväskylä University of Applied Sciences oktatói, továbbá Petteri Weckström, a JAMK LASSO projekt menedzsere.

Abstract

This publication covers the description of NHibernate, an object-relational mapping framework for .NET platform. It describes how the tool was used to make the persistence layer of the back-end applications of Jyväskylä University of Applied Sciences - LASSO M2M Gateway project.

The thesis discusses briefly the project itself, its goals and basic requirements; how the system architecture was planned to solve the requirements the best; which factors were considered to choose NHibernate while designing the system. Different options are shown how the system can be configured to use the framework.

Several elements of the mapping documents, various relations between domain objects are also discussed. The publication covers a detailed description of the implementation of the project, preparing the implemented domain object classes and the persistence layer working together with NHibernate. The two different queries that NHibernate offers for developers are demonstrated and compared shortly.

Finally, this work discusses a simple example from the LASSO M2M Gateway application how the with NHibernate developed persistence layer can be used by other applications or layers of the same application; how easily work with the implemented libraries and how the transparent database access works in practise.

CONTENTS

FIGURES

TABLES

1 PREFACE

NHibernate is an open-software, a port of the Hibernate Core for Java to the .NET Framework. Basically, it is a middle-tier between the application and the database management system. It ensures that the programmer does not have to write any SQL command while persisting plain .NET objects to and from an underlying relational database, thus a database independent application can be developed. It automatically handles the problems which come from the differences between relational and object-oriented aspects. Thus the programmer can focus on the task to be solved, the developing time can short, more efficient code can be written.

This work describes the ORM framework that was used to make the persistence layer of the back-end application of Jyväskylä University of Applied Sciences - LASSO M2M Gateway project. It will be discussed which factors were considered to choose NHibernate while designing the system; how and what must be configured, how to use the framework, and at the end of the thesis an example will be shown how the developed libraries can be used by other applications or layers of the application.

2 INTRODUCTION

2.1 The project

LASSO is a project of Jyväskylä University of Applied Sciences School of Information Technology focusing on the development environment for wireless applications. One of its sub-projects is the Machine-to-machine communication (M2M) Gateway project, within the frameworks of which this thesis was made. The main objective of the project was to develop a measurement system using by the IEEE 802.15.4 standard communication protocol between the physical devices.

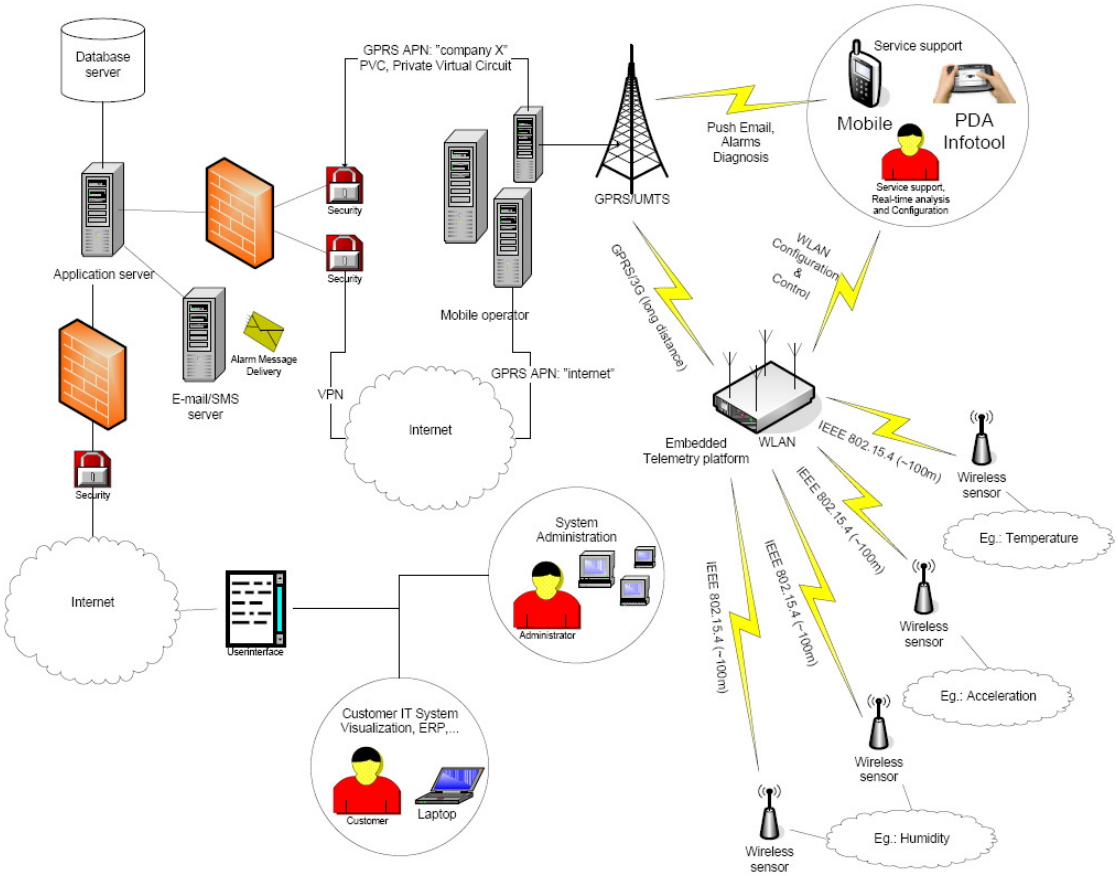

FIGURE 1. LASSO M2M Gateway system

The sensors take measurements about their close environment; the unit can be temperature, moisture, acceleration, etc. The values are stored temporarily in the memory of the device, and collected till the report interval expires.

The logger module is the heart of the system, it keeps contact with the web server and the sensor modules, controls the measurements, collects the taken values from the sensors and forwards them to the web server. The module synchronizes the configuration file between the server and its local drive. It informs the system about the problems and errors that occur during the measurements, and sends reports about its own status and GPS location as well.

Because the sensor and logger devices use wireless protocol for communication, there is no need to build a fixed topology; ad hoc network can be installed or the topology can be changed easily. The sensors can be located even on moving platforms. The system is perfect for temporary use or places/situations where building a wire-based measurement system is not employable. It is also energy efficient, because the modules use battery supply with at least a one-year long lifetime, but they can operate from main-power as well.

The web server receives and analyses the messages, and sends alarm messages via SMS or e-mail to the service department about the problems. It archives the measurement messages and logger reports to the database for later issues. The web server informs the logger module to synchronize if new configuration file was made using by the web user interface, or it saves the changes of the configurations to the database if they come from the logger.

There is a web user interface with which the user can manage the database entities, add, edit or delete devices. It authorizes the users, filters their possibilities to use the system depending upon their role and their company. It gives an interface to make charts to monitor the measurements.

Furthermore, there is a possibility to connect directly to the logger module via WLAN or Ethernet cable with a handheld computer. With its user interface the member of the service department can set the configurations on the device's local drive, and force the synchronization with the web server if the changes have been saved. The user can see the results of the measurements on-the-fly when he/she looks for better performance settings or reasons of errors; these measurements can be deleted or be sent immediately to the web

server. This user interface is not part of the first release of the LASSO M2M Gateway project's application, but when designing the system it had to be taken consideration.

2.2 Requirements

2.2.1 Functional requirements

Before the planning and the implementation, information has to be collected about the requirements and demands of the system. The best technique for gathering ideas is the brainstorming with the costumer, making use case diagrams which illustrates all the features that the application must provide in the future.

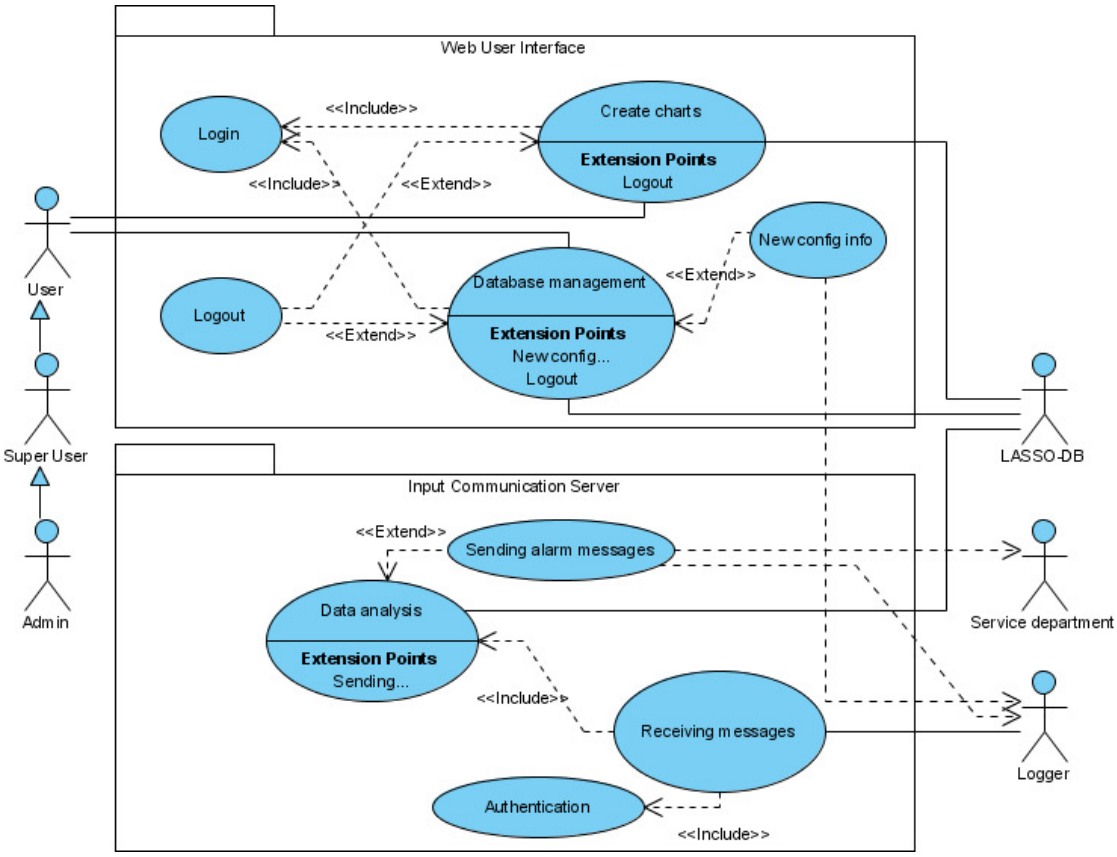

FIGURE 2. The use case diagram of the system

After the use case diagram seen on Figure 2. was made, the listed below functional requirements were identified and analysed. The following table consists of the titles of the requirements and their short descriptions with explanations of expectations which were found during the requirement analysis.

TABLE 1. Functional requirements

Code	*Title*	*Description*
1	Authentication (Login/Logout)	The users and the logger devices are authenticated on a secure channel using account name and password.
2	Managing database entities	All the database entities are managed by the users, CRUD operations are supported, the functions are available due to user roles.
3	Creating charts	Graphical reports can be made to monitor the measurements.
4	Receiving logger messages	The application receives the three different kind of messages from the logger module. These are logger report, sensor measurements and configuration file.
5	Data analysis	All the incoming messages are analysed by the system: message syntax and content of data are checked before saving into the database.
6	Sending info/alarm messages	When new configuration file is created by the user, the logger module is informed about it and the module is called to synchronize; if error is detected during data analysis then the service department and the logger module is informed.
7	Making log about problems	Information about exceptions and errors is stored for application debugging purposes.

2.2.2 Non-functional requirements

There was a basic requirement in the project that the developed application must be *scalable* and *modifiable* in the future. Adding new features, reusing or adapting existing code parts must be done easily. The whole system should be developed using the *same programming language* to ensure the mentioned demands. On the target hand-held devices there is Windows CE or Windows Mobile 5.0 operating system, to which applications can be written simply by using the Microsoft .NET Compact Framework. Thus, it was a rational choice to use .NET/C# and ASP.NET to implement the server side application and the web user interface.

The components of the measurement system are installed around the database management system. The devices send the measurement messages and status reports into it, the charts are made based on the values received from it. Because the database was not chosen at the beginning of the project and *it could be changed any time* during the

development or in even the lifetime of the application, while designing the system it had to be guaranteed that the code will be *independent from the dialect*. A persistence layer must be used to hide it from the application, and a special framework or technique which ensures this transparency.

Another important requirement of the LASSO M2M Gateway project was connected to the charts. These graphical reports must be *complex* enough to present the values of the different terms and searching criteria. They must be *easy to create and visualize* on the screen, the size of the chart file must be *small* to avoid the problems when the user uses low-bandwidth Internet connection. After thinking over the demands and the possible alternatives, the Scalable Vector Graphic (SVG) was decided to use. Because of its XML structure, the creation can be easily automated and even modified by hand. It is supported in the popular web browsers, such as Mozilla Firefox, Opera and Microsoft Internet Explorer. However, the last one needs the Adobe SVG Viewer plug-in to download, but the animations can be shown using it, with which the charts become interactive.

3 DESIGNING THE APPLICATION

Nowadays most of the business applications use object-oriented technology. In a large- or medium-sized application, just like the LASSO M2M Gateway project, it makes sense to organize classes by concern. These are the persistence, business logic and presentation. Of course, there are also the so-called “cross-cutting” concerns, like logging, authorization, etc., which may be implemented generically. A typical object-oriented architecture includes layers that represent the concerns. These kind of architectures are called n-tier or layered system architectures.

In the n-tier architecture each layer interacts with only the layer directly below, and has specific function that it is responsible for. Its advantage is that each layer can be located on physically different servers with only minor code changes, thus they handle more server load. Besides, what each layer does inside is completely hidden to others and this makes it possible to change or update one layer without recompiling or modifying other ones. This is a very powerful feature of layered system architecture, that adding new features or changing a layer can be done without redeploying the whole application. For example, by separating data access code from the business logic code, when the database server changes then just the data access code has to be changed. Because business logic code stays the same, it does not need to be modified or recompiled. (Yang 2002)

It is a normal and certainly the best practice to group all classes and components responsible for persistence into a separate persistence layer in the n-tier architecture. The main objective of the persistence layer is to transform the objects to or from relational entities when retrieving, creating, editing or deleting them. The programmer has to take care of solving the problems caused by the differences between the relational and object oriented aspects, which can take lots of development time. Fortunately, there are a lot of existing techniques and frameworks which can handle the problems automatically, here the object-relational mapping (ORM) comes to the picture.

An ORM implementation must systematically and predictably choose which tables to use for the relevant object type, and generate the necessary SQL-commands automatically. The system creates a “virtual database”, which consists of persistent objects that can be modified directly by the program code. The objects are created on a usual manner with

constructors, filled with data using their properties, after which they automatically end up in the database. The database access is fully transparent for the application. (Wikipedia 2007)

After feeling the weight of pros and cons of the usage of object relational mapping, the decision was made to use it; namely one of the most popular object-relational mapping tool, the .NET alternative of in the Java world well known and widely used Hibernate was decided to use, which is NHibernate.

Advantages of using a separated persistence layer with ORM framework are that the application code and the SQL commands do not mix, the code will be more clear-out and will have a better structure. The modifications in the database do not affect the code of other layers of the application, the developers do not have to know anything how the objects are represented in the data tables, they do not have to write any SQL commands. The definition of the database can be found in typed-classes, thus most of the runtime errors can be avoided in compile time. With using an ORM tool a lot of time can be saved, especially during the development, performance and scalability can be increased.

Advantages of using NHibernate are, for example, that it does not need any special interface implementation or extra framework usage to work fine with the domain model objects. The database can be changed and modification of database implementation can be done easily. It does not just handle the mapping but gives an efficient query language for the application developers. It is popular, very well documented and widely supported, thus a beginner programmer can find help for his/her problems quite fast from many sources.

The LASSO application can be divided into two basic components depending upon the two distinct functionalities it has to execute. One of the components, called `Web User Interface` on the Figure 3. keeps contact with the user through a web browser interface. It handles the user's interactions, manages the database entities, informs the logger module if new configuration file was made after modifying the modules' settings, and makes graphical reports to monitor the measurements based on different criteria. The other component, called `Input Communication Server` is responsible for archiving the messages that come from the devices. It analyses the input information, and if it is necessary informs the service department unit and the device about the found problem(s);otherwise, it saves everything into the database. The communication between

the the logger modules and the `Input Communication Server`, the message- and the configuration file-transfer, goes through the FTP-server, the data is stored there temporarily until one party does not process them.

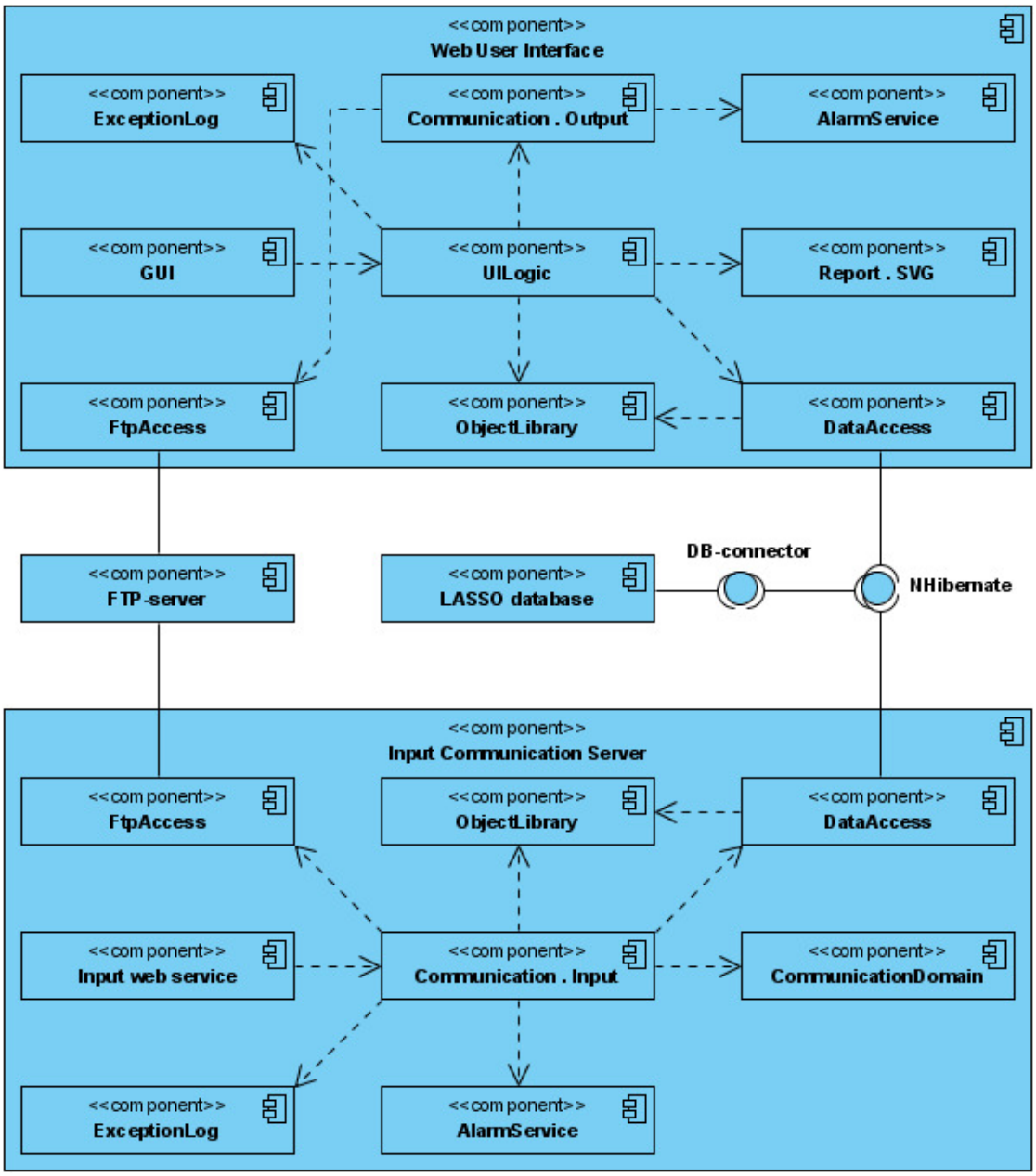

FIGURE 3. The component diagram of the system

As can be seen, the two components have some common features, such as data base and FTP server access, sending alarm messages. Both of them work on the same domain model, just the operations that they can do on it are not the exactly the same. While

designing the applications, these factors had to be taken into consideration. Both of the back-end applications could be divided into further components. These parts were implemented separately, and the applications were built from this parts later.

This work focuses on introducing just two components, namely the `ObjectLibrary` and the `DataAccess`. The first one consists the classes of the domain model; by almost all of the other components of the system; these objects are saved into the database and are retrieved from it. All of these operations are handled by the second one, which is the persistence layer of the n-tier applications. It uses the NHibernate to access the database and execute the necessary functions.

4 BASIC CONFIGURATIONS

In this chapter will be illustrated what has to be configured and how, in the application to use NHibernate for the object-relational mapping and manipulating persistent data. These configurations may be used in separated layers of the architecture, the programmer will meet them in different periods of the development but all of them are necessary sooner or later.

The basic configurations will be discussed now, before the specific chapters about implementation, because they are necessary if the data access tier has not been implemented but the mapping needs to be tested with a console application, for example. During the development of the LASSO M2M Gateway project, a simple windows forms application was created, and used to show the test results in listboxes and gridviews. While checking the mapping documents it contained the whole data access code, later it was just the user interface that used all the functionality of the persistence layer.

The main source of this and the other chapters about implementation is the NHibernate Reference Documentation (see NHibernate 2007) which was also the base material during implementation of the namespaces which will be next discussed below.

4.1 Start using NHibernate

The most important thing to consider before using NHibernate is what kind of database will be used. NHibernate is designed to operate in many different environments, the number of the supported databases is quite big. It works fine with most of the database systems on the market in default, so it depends mostly on the developer's taste what will be used. There was a basic demand in the LASSO requirement specification that an open database must be used, and the decision was made to use MySQL.

After installation and configuration of the database management system, Visual Studio can be opened to prepare the application to use NHibernate. In the distributed package which can be downloaded from the official website everything can be found that is needed. In the project of the data access layer reference has to be added to `NHibernate.dll`, Visual Studio will automatically copy the library and all the

dependencies to the project output directory. (There are two different versions of this library in the installed package, one for .NET 1.1 and one for .NET 2.0.)

4.2 Programmatic configuration

Because NHibernate needs explanation what kind of objects it will persist, in what kind of database environment it will work, an instance of **Nhibernate.Cfg.Configuration** is needed which represents an entire set of mappings of the .NET types of the application to a relational database. The metadata of mappings is compiled from various XML mapping files, which will be discussed later in Chapter 5.2. A **Configuration** is meant to be a configuration-time object which is discarded after an immutable **ISessionFactory** is built.

There are three different ways to create **Configuration** instance:

- The first and probably not so often used way is when the names of the mapping files are specified in the application code directly. This approach is uncomfortable if there are a lot of mapping documents, because they have to be specified one by one. This solution should be avoided.

- The second and better option to make a **Configuration** instance is to let NHibernate load mapping files from embedded resources. Then NHibernate will look for mapping documents named `FullyQualifiedClassName.hbm.xml` embedded as resources in the assembly that the types are contained in. This approach eliminates any hardcoded filenames.

- The last and probably best alternative is to load all of the mapping files contained in an Assembly. This solution is comfortable, because the changes in numbers or names of the domain classes and their documents do not need to be considered during the development, NHibernate will look through the assembly for any resources that end with `.hbm.xml`. The only task is to create the mapping files with this extension and compile them into the assembly as Embedded Resources.

```
Configuration config = new Configuration()
   .AddAssembly( "Lasso.ObjectLibrary" );
```

In the LASSO M2M Gateway project the third way of creating **Configuration** is used.

4.3 Obtaining an ISessionFactory

When all mappings have been parsed by the **Configuration**, the application must obtain a factory for **ISession** instances. An **ISessionFactory** is an expensive-to-create, threadsafe object intended to be shared by all application threads.

```
ISessionFactory factory = config.BuildSessionFactory();
```

4.4 ADO.NET connection

One of the supported and in the project also used alternative to make connection with the database is the NHibernate provided ADO.NET connection. The **ISessionFactory** must be provided with ADO.NET connection properties which must be specified in the application configuration file, in the `hibernate.cfg.xml` file or added to the **Configuration** instance by using its **SetProperties()** method.

Opening an **ISession** is as simple as:

```
ISession session = factory.OpenSession(); // open a new Session
// do some data access work
```

The example below shows how to specify the database connection properties in the application configuration file. From NHibernate 1.2 the **Assembly.Load()** is used instead of **Assembly.LoadWithPartialName()** to load driver assemblies. This means that NHibernate will no longer look for the highest-versioned assembly in the Global Assembly Cache (GAC), which was sometimes undesirable. Instead, it is now the programmer's responsibility to either put the provider assembly into the application directory, or add a **qualifyAssembly** element to the application configuration file, specifying the full name of the assembly.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="nhibernate"
             type="System.Configuration.NameValueSectionHandler,
```

```
                    System, Version=1.0.5000.0, Culture=neutral,
                    PublicKeyToken=b77a5c561934e089" />
  </configSections>
  <nhibernate>
    <add key="hibernate.connection.provider"
         value="NHibernate.Connection.DriverConnectionProvider" />
    <add key="hibernate.dialect"
         value="NHibernate.Dialect.MySQLDialect" />
    <add key="hibernate.connection.driver_class"
         value="NHibernate.Driver.MySqlDataDriver" />
    <add key="hibernate.connection.connection_string"
         value="Server=localhost;Database=cat;
                User ID=root;Password=pass;CharSet=utf8" />
  </nhibernate>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <qualifyAssembly
          partialName="MySql.Data"
          fullName="MySql.Data, Version=5.1.2.2, Culture=neutral,
                    PublicKeyToken=C5687FC88969C44D"/>
    </assemblyBinding>
  </runtime>
  <!-- other application specific configuration here -->
</configuration>
```

4.5 Configuration properties

There are a number of optional properties that control the behaviour of NHibernate at runtime, but their default values are suitable for most of the developers. Of course, there are exceptions, for example the `hibernate.dialect` which always must be set to the correct class of the **NHibernate.Dialect** namespace for the database.

The system-level properties can only be set manually by setting static properties of **Nhibernate.Cfg.Environment** class or be defined in the `<nhibernate>` section of the application configuration file, they cannot be set using **Configuration.SetProperties()** or be defined in the `<hibernate-configuration>` section of the application configuration file.

5 THE `Lasso.ObjectLibrary` NAMESPACE

Once the framework is configured to recognize the data store, the next step is to create the domain model and database representation. It is entirely plausible to carry out those steps in either order.

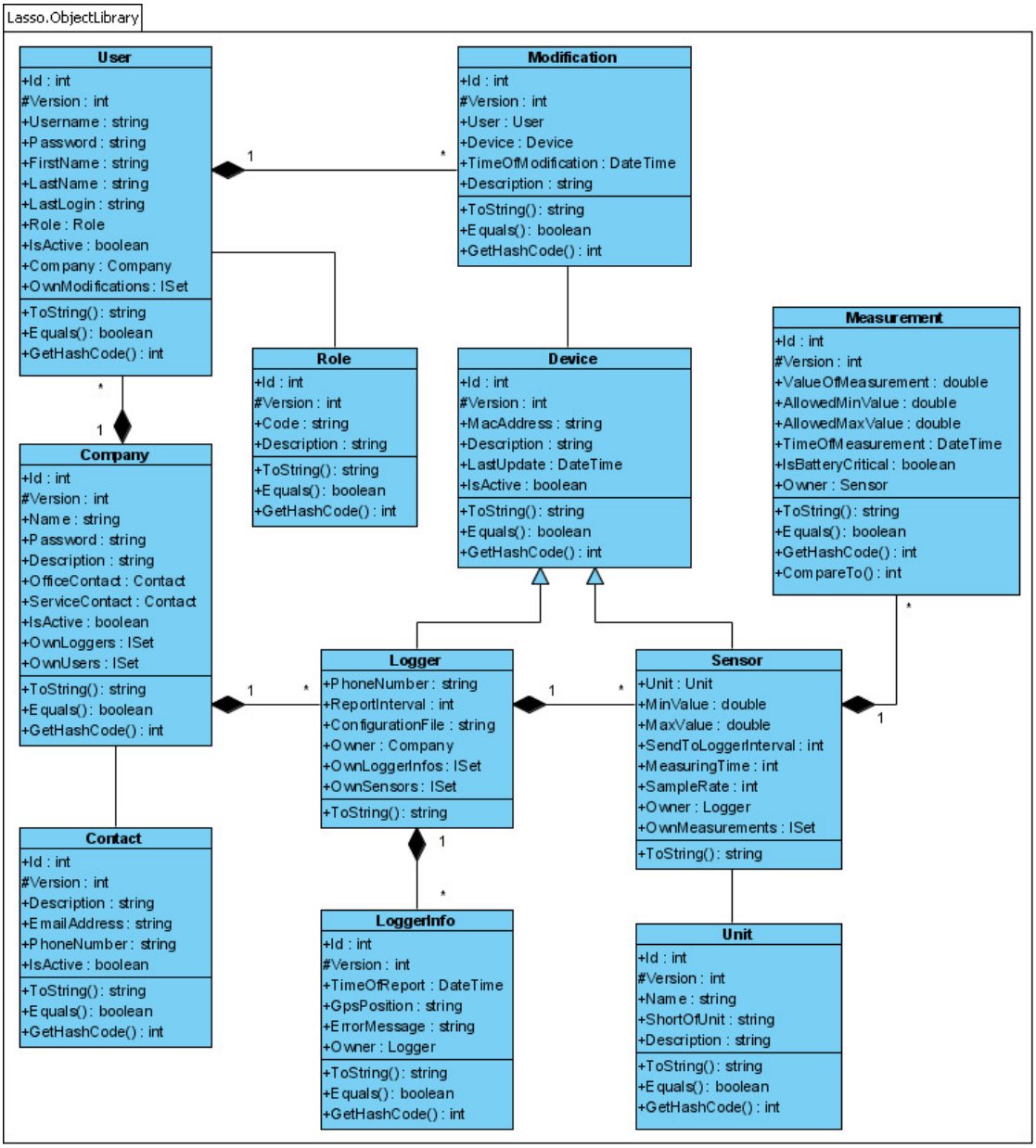

FIGURE 4. Lasso.ObjectLibrary namespace

If the application depends on a highly efficient storage schema, perhaps starting with the database representation is appropriate. However, if the database is just a place to store the

object state, then it makes more sense to start in the domain classes. There is a third option to start with the mapping files that will describe the relations between the classes and data tables. (Gethland, 2004a)

In this case, the implementation of the domain classes was made first (the class diagram can be seen on the Figure 4.), because this side is more flexible for changes, which were common at the beginning of the project. The database schema was designed just after the database management system was selected and the domain model was more or less fixed.

Through examples of the LASSO project is illustrated more in detail how the implementation of a persistent class should be done, what rules must be followed, how a mapping document looks like and how it is created.

5.1 Restrictions in persistent classes

The implementation of the domain model is fairly straightforward. No framework is needed nor it is necessary to extend some "DAOSupport" superclass from any library. All that is needed to do is to keep the transaction boundary (begin and commit) as well as any **ISession** handling code outside of the domain model implementation. After the data fields are coded, some simple rules should be followed so that NHibernate works fine with the classes.

A closer look at the **Lasso.ObjectLibrary.User** class, as follows:

```
using System;
using Iesi.Collections;

namespace Lasso.ObjectLibrary {
   public class User {
      private int _id;
      private int _version;
      private string _username;
      private string _password;
      private string _firstName;
      private string _lastName;
      private DateTime _lastLogin;
      private bool _isActive;
```

```
        private Role _role;
        private Company _company;
        private ISet _ownModifications;
    }
}
```

As against many other ORM tools, NHibernate persists properties using their data fields, their getter and setter methods. So accessors have to be declared for all the fields. Fortunately, the visibility of these properties can be internal, protected, protected internal or even private, but all the public methods, properties and events have to be declared as virtual , as the following string of code illustrates.

```
public class User {
    // ...
    public virtual int Id {
        get { return _id; }
        protected set { _id = value; }
    }
    protected int Version {
        get { return _version; }
        set { _version = value; }
    }
    public virtual string Username {
        get { return _username; }
        set { _username = value; }
    }
    // ...
}
```

Having a property that holds the primary key column of the database table (just like the `Id` here) is a good and popular design decision for many applications. This property can be called anything and its type can be any primitive type, **string, System.DateTime** or even user-defined class type.

While using NHibernate the identifier property is optional. However, some functionality, such as cascaded updates and **ISession.SaveOrUpdate**(), is available only to classes which declare it. Thus declaration of consistently-named identifier properties is recommended.

The next thing NHibernate needs is a parameterless constructor. All the persistent classes must have one with any of the mentioned visibility, so that the ORM tool instantiates the objects using **Activator.CreateInstante()**. Other constructors can be declared with business logic meaning, but attention should be paid: if a non-default constructor is implemented, the parameterless one has to be declared explicitly.

```
public class User {
   // ...
   protected User() {
      _ownModifications = new HashedSet();
   }
   public User(string username, string password, string firstName,
               string lastName, Company company, Role role,
               bool isActive) : this() {
      _username = username;
      _password = password;
      _firstName = firstName;
      _lastName = lastName;
      _company = company;
      _role = role;
      _isActive = isActive;
      _lastLogin = DateTime.MinValue;
   }
}
```

Except for the implementation of the associations and inheritance that will discussed later in Chapter 5.2.7, the classes with the data fields, prepared properties and constructors are ready to work together with NHibernate. It is time to provide the mapping files that fill the domain model from the data tables.

5.2 Mapping declaration

The object relational mapping is defined in XML documents. The mapping language is object-centric, it means the mapping files are used to define the persistent objects and contain information about an object's persistent fields, associations and subclasses, they are constructed around the class declaration, not table declaration. They are compiled at

application start-up to provide this information. One mapping document is created for each class, their filenames have to follow the pattern `ClassName.hbm.xml`.

During the LASSO development the mapping documents were created and edited by hand, no automatism was used, even though there are several possibilities to create these documents automatically with code generators or decorating the classes with the `NHibernate.Mapping.Attributes` library. In this case the validation of the XML-files was made with help of Visual Studio and `nhibernate-mapping.xsd` schema definition file.

Next, the content of a mapping document will be discussed, focusing on the most important elements and attributes are used by NHibernate at runtime. If one is interested in other elements or attributes not-mentioned here or more information is needed on the presented ones, the NHibernate Reference Documentation can be checked.

For example, the mapping file of the **Lasso.ObjectLibrary.User** class can be seen as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
                   namespace="Lasso.ObjectLibrary"
                   assembly="Lasso.ObjectLibrary">
  <class name="User" table="User">
    <id name="Id" column="Id"
        access="nosetter.camelcase-underscore" unsaved-value="0" >
      <generator class="native" />
    </id>
    <version name="Version" column="Version"
          access="field.camelcase-underscore" unsaved-value="0" />
    <property name="Username" column="Username"
           access="nosetter.camelcase-underscore" unique="true" />
    <property name="Password" column="Password" not-null="true" />
    <property name="FirstName" column="FirstName" />
    <property name="LastName" column="LastName" />
    <property name="LastLogin" column="LastLogin" type="DateTime"/>
    <property name="IsActive" column="IsActive" />
    <many-to-one name="Role" class="Role" column="RoleId"
                 not-null="true" cascade="none" lazy="false" />
```

```
    <many-to-one name="Company" class="Company" column="CompanyId"
                 access="nosetter.camelcase-underscore"
                 not-null="true" cascade="none" lazy="false" />
    <set name="OwnModifications" table="Modification"
         inverse="true" lazy="true" cascade="all">
      <key column="UserId" />
      <one-to-many class="Modification" />
    </set>
  </class>
</hibernate-mapping>
```

5.2.1 The `hibernate-mapping` element

As can be seen in the code snippet above, the file starts with a standard XML declaration.

The root element of the mapping file is the `<hibernate-mapping>` element with the namespace declaration. The schema definition `nhibernate-mapping.xsd` file can be found in the distributed package of the ORM framework.

```
<hibernate-mapping
     schema="schemaName"                (1)
     default-cascade="none|save-update"
     auto-import="true|false"
     assembly="assemblyName"            (2)
     namespace="namespaceName"          (3)
/>
```

The optional `schema` (1) attribute specifies that tables referred to by this mapping belong to the named schema. If specified, table names will be qualified by the given schema name. If missing, table names will be unqualified. The value of the attribute can be overridden by similar named attribute of the class element. The optional `assembly` (1) and `namespace` (2) attributes free the user to use the fully-qualified class names in the mapping document, including the name of the assembly. Because the name of the classes may occur several times in the mapping documents, use of these attributes is recommended.

5.2.2 The `class` element

The next step is to tell NHibernate which class type is mapped, using the `<class>` element.

```
<class
     name="ClassName"                              (1)
     table="tableName"                             (2)
     discriminator-value="discriminator_value"     (3)
     mutable="true|false"                          (4)
     schema="schemaName"
     proxy="ProxyInterfaceName"
     dynamic-update="true|false"
     dynamic-insert="true|false"
     select-before-update="true|false"
     polymorphism="implicit|explicit"
     where="arbitrary sql where condition"
     persister="PersisterClassName"
     batch-size="N"
     optimistic-lock="none|version|dirty|all"
     lazy="true|false"
/>
```

When a class or interface type is passed to NHibernate, the fully qualified `name` (1) including all namespaces and the name of the assembly containing it must be given. For the **User** class, this is `"Lasso.ObjectLibrary.User, Lasso.ObjectLibrary"`. The `table` (2) is the name of the data table that holds the data also has to be specified.

If there is a class or interface type, which contains constant data, the `mutable` (4) attribute with `"false"` value may be specified. The immutable classes cannot be updated or deleted by the application, this allows NHibernate to make some performance optimizations.

The `discriminator-value` (3) attribute is optional, it plays important role when inheritance is implemented with table-per-class hierarchy strategy. More can be read about this topic in subchapter “Inheritance mapping**”**.

5.2.3 Property mapping

All property mappings share some common features: the name of the field on the class, the column in the table that contains the field's data and the type of the field that is being persisted. Any standard properties of a class will look largely the same.

The user's classes can have properties that are not persistent. This is good in the transparent data layer: if the domain calls for runtime-calculated properties, the classes can have them mixed in with the persistent ones. The mapping files can just ignore the non-persistent ones.

5.2.4 System type properties

All the classes have some data fields, which have .NET basic type (e.g. **System.String**, **System.Int32**), enumeration, serializable .NET type or custom type. These fields hold the data of the persistent object, and they can be accessed through properties. In the NHibernate mapping these fields and/or properties are defined with the `<property>` element. Its syntax is the following:

```
<property
     name="PropertyName"                 (1)
     column="column_name"                (2)
     type="typename"                     (3)
     update="true|false"
     insert="true|false"
     formula="arbitrary SQL expression"
     access="field|property|ClassName"   (4)
     optimistic-lock="true|false"
     generated="never|insert|always"
/>
```

The `name` (1) - name of the property - is a mandatory, but the `column` (2) - the name of the mapped table column - is an optional attribute. If the second is not specified, NHibernate will look for a column with the name of the property.

The `type` (3) which indicates the NHibernate type does not have to be specified, either. The value of this attribute can be the name of a NHibernate type, a basic .NET type, an

enumeration, a serializable .NET type or a custom class type. If no type is given, NHibernate will use reflection upon the named property to take a guess at the correct NHibernate type. However, in some cases the type attribute is necessary.

NHibernate supports .NET 2.0 **Nullable** types. These are mostly treated the same as plain non-nullable types inside. For example, a property of type **Nullable<Int32>** can be mapped using `type="Int32"` or `type="System.Int32"`. The difference between them is that the first uses the **Int32** basic NHibernate, the second the **System.Int32** .NET type.

In Chapter 5.1 was mentioned that all the fields have to have accessor methods, because NHibernate persists properties using their getter and setter methods. This it is not totally true. NHibernate uses properties to manipulate the object's data, but `access` (4) attribute gives an opportunity not to write getter and/or setter accessor for all the fields or not to use them while mapping persistent data from the database; but work just with “virtual properties”.

The `access` attribute lets the programmer control how NHibernate accesses the value of the property at runtime. The value of the attribute should be text formatted as `access-strategy.naming-strategy`. With the default value (`"property"`) NHibernate uses the get/set accessors, no naming strategy should be used, because the value of the name attribute is the name of the property.

The more interesting values are the `"nosetter"` and the `"field"`, both of them need naming strategy. With the first one, the field will be accessed directly when the value is set and the property will be used when the value is got. With the second one NHibernate can use the value of the `name` attribute as the name of the field. This can be used when a property's getter and setter contain extra actions that should not occurre when NHibernate is populating or reading the object.

If the access attribute in the mapping file is not specified, the unnecessary get/set properties can be ignored from the implementation like in this thesis work. After the modification the **User** class looks like this:

```
public class User {
   private int _id;
   private int _version;
   private string _username;
```

```
   private string _company;
   // rest of the fields
   public virtual int Id {
      get { return _id; }
   }
   // no Version property at all !
   public virtual string Username {
      get { return _username; }
   }
   public virtual string Company {
      get { return _company; }
   }
   // rest of the properties
   // constructors
}
```

5.2.5 The `id` element

A more interesting case is dealing with the field of the class that identifies the persistent instance. Every class should have a field like this which contains the value of the primary key column in the database table. This is known as the `Id` property; it is defined by the `<id>` element in the mapping file. When this property is mapped, the standard set of attributes has to be used as could be seen in the previous chapter, and two new ones: the `unsaved-value` attribute and the `<generator>` child element.

The syntax is the following:

```
<id
     name="PropertyName”
     column="column_name"
     type="typename"
     unsaved-value="any|none|null|id_value"          (1)
     access="field|property|nosetter|ClassName”>

     <generator class="generatorClass"/>             (2)
</id>
```

The `generator` (2) of the `Id` field lets NHibernate know how these unique identifiers are created: by the programmer, by NHibernate or by the underlying persistence store. Different applications will have different rules about identifiers, and different databases offer unique services for managing those values, thus the decision has to be made carefully based on the requirements and infrastructure.

All the generator classes implement the interface **NHibernate.Id.IIdentifierGenerator**. However, NHibernate provides a range of built-in implementations, some applications may choose to provide their own specialized ones. There are shortcut names for the built-in generators, the most common ones are the following:

- `identity`: supports identity column types. The identifier returned by the database is converted to the property type using **Convert.ChangeType**. It supports any integral property type.
- `sequence`: uses a sequence in DB2, PostgreSQL, Oracle or a generator in Firebird. The identifier returned by the database is converted to the property type using **Convert.ChangeType**. It supports any integral property type.
- `hilo`: uses a hi/lo algorithm to generate identifiers of any integral type, but these are unique only for a particular database. It is not recommended to use this generator with a user provided connection.
- `native`: chooses one of the previous three values depending upon the capabilities of the underlying database.

Recommended to use the native generatorand the `unsaved-value` (1) attribute that specifies a default value for the `Id` property when an object is created and not yet persisted.

5.2.6 The `version` element

When designing the application the question of transactions and concurrency must be considered. Many business processes require a whole series of interactions with the user, but in web and enterprise applications it is not acceptable for a database transaction to wait for interaction. In this case, if long application transactions are planned to be use, the only approach that is consistent with high concurrency and high scalability is optimistic concurrency control with versioning. Here the `<version>` element comes to the picture.

```
<version
     column="version_column"
     name="PropertyName"
     type="typename"
     access="field|property|nosetter|ClassName"
     unsaved-value="null|negative|undefined|value"
     generated="never|always"
/>
```

NHibernate can automatically handle the version number defined with this special property which is used for automatic or application handled version checking in optimistic concurrency control.

5.2.7 Relations between objects

Now, new application can be created in which all the domain model classes are independent from each other and have just common **System** type data fields. Unfortunately, this is not enough for any application.

While designing a system of a real life situation, connections between objects are always looked for, problems are generalized, instances are organized to hierarchy based on common properties and behaviour; this is the human nature. But the relational model and object-oriented paradigm define the solution for these questions differently. For example, in the database there are relationships between tables as against in the world of the object-oriented approach, relations between objects are represented by associations. Managing these differences between the two worlds is the soul of object relational mapping.

Associations

Association defines a relationship between classes of objects which allows one object instance to cause another to perform an action on its behalf. This relationship is structural, because it specifies that objects of one kind are connected to objects of another. In describing and classifying associations, almost always their multiplicity is used. (Bauer, King 2005)

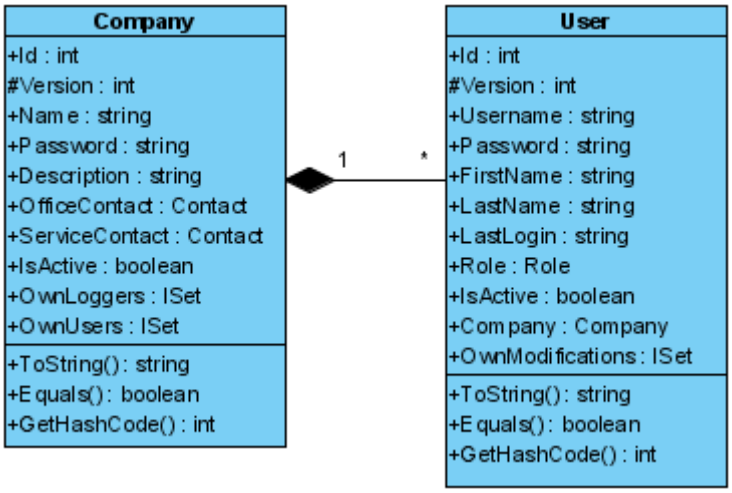

FIGURE 5. Association between the Company and User classes

Next the relation between the **User** and **Company** domain classes is discussed more in detail.

After taking a look at the object model, it can be concluded that the association from **User** to **Company** is a many-to-one association. Reminding that associations are directional, the inverse association from **Company** to **User** is also called a one-to-many association. (Clearly there are two more possibilities, the one-to-one and the many-to-many, but they are not used in the LASSO domain, thus they are not discussed here.)

Unidirectional associations

The association from **User** to **Company** is an example of the simplest possible kind of association in ORM. The object reference returned by the get of the `Company` property in the **User** class is easily mapped to a foreign key column in the `User` table to the primary key of the `Company` table. In NHibernate this association is declared using the `<many-to-one>` element.

```
<many-to-one
     name="PropertyName"                                    (1)
     column="column_name"
     class="ClassName"
```

```
    cascade="all|none|save-update|delete"
    fetch="join|select"
    update="true|false"
    insert="true|false"
    property-ref="PropertyNameFromAssociatedClass"
    access="field|property|nosetter|ClassName"
    unique="true|false"
    optimistic-lock="true|false"
    not-found="ignore|exception"                          (2)
/>
```

In the mapping document the `name` (1) of the class that the association refers to can be specified explicitly, but it is usually optional, since NHibernate can determine this using reflection. With the `not-found` (2) attribute it can be specified how foreign keys that the reference missing rows should be handled: they can be ignored with creation of a null-asspciation, or exception can be thrown.

At this point the unidirectional many-to-one association that could be seen on Figure 4. between the **User** and **Role** classes, for example, is ready to map. The word 'unidirectional' means that the **User** object knows about its role, but the other side of the relation, the instance of the **Role** does not know anything about who uses it. This is the simplest case can exist.

However, the relation between the **User** and **Company** classes is not so simple. Every **User** object has a reference to the instance of the **Company** class, but fetching all the users for a particular company also must be implemented, because the company has to know its employees. The following chapter describes how to make the association bidirectional between these classes.

Bidirectional associations

A bidirectional association allows navigation from both ends of the association. As mentioned, the **Company** object has to know which users belong to it, so that a collection of **User** objects is necessary in the **Company** class to collect its employees. NHibernate requires that persistent collection-valued fields must be declared as an

interface type which can be any of the common collection interfaces or an own one can be written by implementing the **NHibernate.UserType.IUserCollectionType** interface. However, in the two supported bidirectional associations, one-to-many and many-to-many, these collections can be just set or bag valued.

However, the **System.Collections** namespace does not provide any set valued collection, but fortunately, a good implementation can be found in the `Iesi.Collections.dll` library. This is based on the Java **Set** interface definition, so if the developer is also a Java programmer, this may seem familiar. A bag is an unordered, not indexed collection which is either provided by the .NET Framework, but NHibernate lets user map properties of type **IList** or **ICollection** with the `<bag>` element. However, that bag semantics are not really part of the **ICollection** contract and they actually conflict with the semantics of the **IList** contract.

There are a number of advantages why it is worth using the set valued collection in the bidirectional association instead of a bag valued: **List**s allow to add new objects easily, but there can be numerous duplicate elements, which sometimes can be a problem. Searching for elements in **Array**s or **List**s is slow for large data sets. With **Set**s, adding-removing elements and checking for existed ones is fast and simple. The elements can be mixed and matched in different sets using the supported mathematical set operators: union, intersection, exclusive-or, and minus.

After feeling the weight of pros and cons, the set valued collections were decided to be used in the LASSO implementation. The interesting part of the **Company**'s code is the following:

```
using System;
using Iesi.Collections;

namespace Lasso.ObjectLibrary {
   public class Company {
      // ... members
      private ISet _ownUsers;
      // ... constructors
```

```
      protected Company()    {
         _ownUsers = new HashedSet;
      }
      // ... properties
      public ISet OwnUsers {
         get { return _ownUsers; }
         set { _ownUsers = vaule; }
      }
      // ... methods
   }
}
```

It should be noticed how the instance variable with an instance of **HashedSet** is initialized. When the persistent instance is made (by calling **ISession.Save()**, for example) NHibernate will actually replace the **HashedSet** with an instance of NHibernate's own implementation of **ISet**.

Collection instances have the usual behaviour of value types. They are automatically persisted when referenced by a persistent object and automatically deleted when unreferenced. Due to the underlying relational model, collection-valued properties do not support null values; NHibernate does not distinguish between a null collection reference and an empty collection.

Next is illustrated how the set valued collection should be defined in the mapping file. The syntax is the following:

```
<set
     name="PropertyName"
     table="table_name"
     schema="schema_name"
     lazy="true|false"                                          (1)
     inverse="true|false"                                       (2)
     cascade="all|none|save-update|delete|all-delete-orphan"    (3)
     sort="unsorted|natural|comparatorClass"
     order-by="column_name asc|desc"
     where="arbitrary sql where condition"
```

```
      fetch="select|join"
      batch-size="N"
      access="field|property|ClassName"
      optimistic-lock="true|false"
      generic="true|false"
>
      <key column=”column_name” />                                    (4)
      <one-to-many                                                    (5)
            class=”ClassName”
            not-found=”ignore|exception”
      />
</set>
```

Collection instances are distinguished in the database by a foreign key to the owning entity. This foreign key is referred to as the collection key, that is mapped by the `column` (4) of the `<key>` element. The type of the collected class is determined by the `<one-to-many>` (5) element.

By default, NHibernate does not navigate an association when searching for transient or detached objects, so saving, deleting, or reattaching an object on one side will not affect the ones on the other side; but it allows to specify a `cascade` (3) style for each association mapping with the listed values:

- `none`, the default, tells NHibernate to ignore the association.
- `save-update` tells NHibernate to navigate the association when the transaction is committed and when an object is passed to **ISession.Save()** or **ISession.Update()** and save newly instantiated transient instances and persist changes to detached instances.
- `delete` tells NHibernate to navigate the association and delete persistent instances when an object is passed to **Isession.Delete()**.
- `all` means to cascade both save-update and delete, as well as calls to evict and lock.

In the **User** class there is the **Company** type property as it was discussed before. This is the link from the **User** to the **Company** defined with the `<many-to-one>` element in the `User.hbm.xml` mapping document. In the **Company**'s mapping it has to be

marked that this relation, the one-to-many, is the other side of the same link, actually this is its inverse. For this purpose the `inverse` (2) attribute must be specified with `"true"` value. Mapping one end of an association with `inverse="true"` does not affect the operation of cascades, both are different concepts.

Collections and objects (see the specification of the `<many-to-one>` element in the previous chapter) may be initialized lazily, it means they load their state from the database only when the application needs to access them. Initialization takes place transparently so the application would not normally need to worry about this. In fact, transparent lazy initialization is the main reason why NHibernate needs its own collection implementations.

Using lazy collections can rise the application's performance; but there is a problem, the lazily initialized proxy object or collection can load its state from the database just before the **ISession** instance was committed. The fix is to move the line that reads from the collection just before the commit. A solution for this problem will be shown in subchapter “Problem of lazy initialization”. Non-laziness is the default, but non-lazy initialization can be used with declaring the `lazy="false"` (1) in the mapping file.

At the end of the chapter about associations the relevant part of the `Company.hbm.xml` mapping file is as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
                   namespace="Lasso.ObjectLibrary"
                   assembly="Lasso.ObjectLibrary">
  <class name="Company" table="Company">
  ...
    <set name="OwnUsers" table="User" inverse="true" lazy="true"
         cascade="all">
      <key column="CompanyId" />
      <one-to-many class="User" />
    </set>
  ...
```

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

Inheritance mapping

Till now each of the classes were mapped into separated database tables. All the relations between objects were implemented with references or collections in the application code, and with foreign keys in the database schema. There is just one thing missing from the mapping files, which is the implementation of the inheritance.

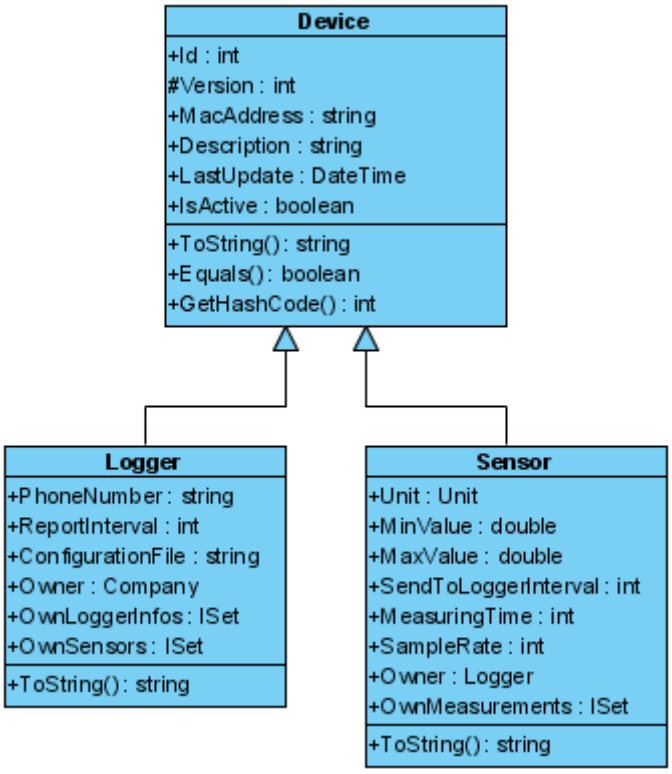

FIGURE 6. A class hierarchy in the LASSO domain

In the object oriented paradigm the classes can be generalized with emphasizing common properties and/or behaviours to a common forefather. The derived class inherits some attributes and methods from its parent. Parent-child relationship can be defined easily but the problem is that the phrase of inheritance is missing from the relational world, thus it has to be originated from the entities and the relations between them. There are three different alternative ways to do it. It is possible to use different mapping strategies for

different branches of the same inheritance hierarchy, but NHibernate does not support mixing table-per-hierarchy and table-per-subclass mappings inside the same `<class>` element. Now the possibilities will be demonstrated one after another, through the example of the Device – Logger – Sensor hierarchy implementation.

Table-per-concrete class

Having one table per concrete class and none for the **Device** abstract superclass. Every table consists of all the own and derived attributes of the represented classes. An advantage is that the ad hoc queries can be executed easily, because all the data which is needed for one class is in the table. The **Device** class is nowhere can be found in the mapping files, but it still exists in the class hierarchy and can be used as a superclass in queries. The biggest disadvantage of this implementation is the redundancy. If one attribute has to be changed in a superclass, all the tables have to be modified, which is quite a big problem if the hierarchy is large. A possible database implementation of this strategy is can be seen on Figure 7.

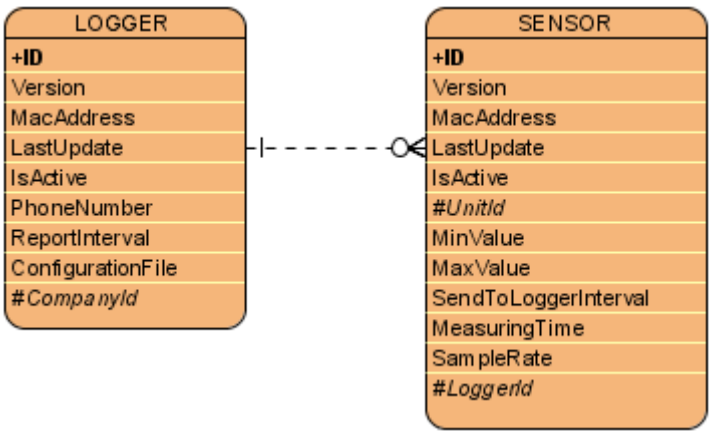

FIGURE 7. Table-per-concrete class implementation

It is important not to have identical `ID` values shared between the two tables. If they are there, NHibernate will return several different objects for the same identifier. This may prove confusing for Hibernate and the user as well.

Table-per-subclass

The data model is very close to the class model. A different table is used for each class in the hierarchy, all these tables have just the specific attributes, but they must share the same primary key. The creation of new subclasses or modification of the superclasses is very simple, because just one table has to be added or modified. However, it has some disadvantages. There can be a lot of tables in the database – one for each class – writing and reading the records take more time, because more tables are involved. This problem can be eliminated by good design of the database schema. Unfortunately, without using view, the realization of the ad hoc queries is quite difficult. Datatables for this solution are on Figure 8.

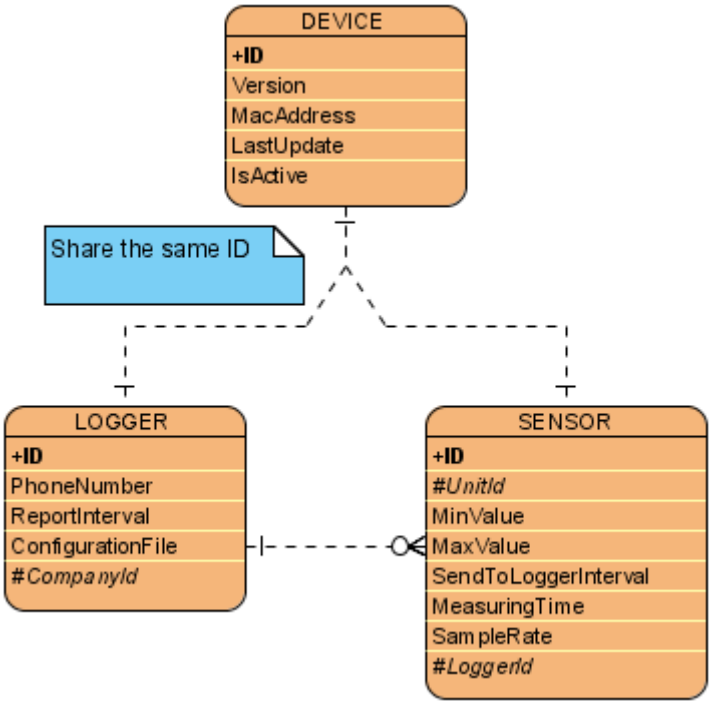

FIGURE 8. Table-per-subclass implementation

NHibernate will read multiple tables each time an object is instantiated and populated. That operation can produce good results if the indexes are well defined and the hierarchy is not too deep. (Coulon, Brousseau 2004).

Table-per-class hierarchy

The whole class hierarchy is mapped into one database table, all the attributes of all the classes are stored here. The objects from different classes have to be distinguished somehow; that is why a special column is needed, which defines the type of the objects. The advantages of the solution are: it is simple; the performance of the queries is good, because there is no need for SQL joins between the tables, and while inserting or updating data just into one table is written. A disadvantage is that every time when a new attribute is added to any of the classes or a new class is defined, then the whole table has to be modified; if a table contains a large amount of data, this process can take a lot of time. In addition, it wastes too much disk space in vain. The data model of this solution as follows on Figure 9. Important, in the database schema all non-shared columns must be set to `NULLABLE`.

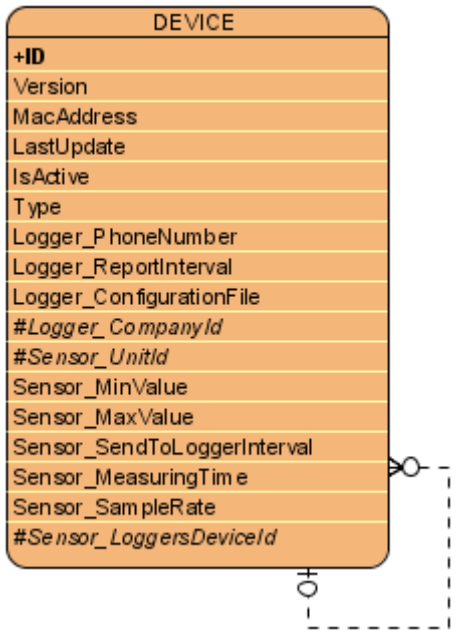

FIGURE 9. Table-per-hierarchy implementation

None of the strategies is perfect. After feeling the weight of advantages and drawbacks the last way was decided to use to map the class hierarchy as follows below:

In the superclass' declaration NHibernate has to be informed which column is the discriminator `column` (1) that contains marker values that tell the persistence layer what subclass to instantiate for a particular row. Popular design decision to chose **String** `type`

(2) for this column which is simply a technical column shared between NHibernate and the database, it is not mapped. The `<discriminator>` defines it which has to stay right after the `<id>` element in the mapping document.

```
<discriminator
     column="discriminator_column"            (1)
     type="discriminator_type"                (2)
     force="true|false"
     insert="true|false"
     formula="arbitrary SQL expression"
/>
```

The table-per-hierarchy strategy requires the declaration of each subclass in the mapping document of the root persistent class. Each subclass should declare its own persistent properties and subclasses, but the `<version>` and `<id>` properties are always inherited from the root class. Each subclass in a hierarchy must define a unique `discriminator-value` (4). If none is specified, the fully qualified .NET class name is used.

```
<subclass
     name="ClassName"
     extends=”SuperClassName”                 (3)
     discriminator-value="discriminator_value" (4)
     proxy="ProxyInterface"
     lazy="true|false"
     dynamic-update="true|false"
     dynamic-insert="true|false">
     <property .... />
     ...
</subclass>
```

It is possible to define subclass mappings in separate mapping documents, directly beneath `<hibernate-mapping>`. This allows to extend a class hierarchy just by adding a new mapping file. An `extend` (3) attribute must be specified in the subclass mapping, naming a previously mapped superclass. Use of this feature makes the ordering of the mapping documents important.

For completeness, here are the relevant parts of the `Device.hbm.xml` document:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
                   namespace="Lasso.ObjectLibrary"
                   assembly="Lasso.ObjectLibrary">
  <class name="Device" table="Device">
    <id name="Id" column="Id"
        access="nosetter.camelcase-underscore" unsaved-value="0" >
      <generator class="native" />
    </id>
    <discriminator column="Type" type="string"/>
    <version name="Version" column="Version"
          access="field.camelcase-underscore" unsaved-value="0" />
    <property name="MacAddress" column="MacAddress"
          access="nosetter.camelcase-underscore" not-null="true"
          unique="true" />
    ...
    <subclass name="Logger" discriminator-value="LOGGER">
      ...
    </subclass>

    <subclass name="Sensor" discriminator-value="SENSOR">
      ...
    </subclass>
  </class>
</hibernate-mapping>
```

Once all of the mapping declarations were introduced, an own set of mapping files can be created to create, save and load instances in the domain model.

5.3 Implementing methods

Before showing how to deal with NHibernate to manipulate persistent data, some information must be given about the implementation of methods in the domain classes.

The methods for the classes can freely be written, but the lazy initialization never should be forgotten if it is used in the mapping documents. Uninitialized objects or collections

should never be used outside of an open **ISession**'s context, because **NHibernate.LazyInitializationException** will occur. Additionally, it is hardly recommended to override the **Equals()** and **GetHashCode()** methods if objects of persistent classes are intended to be mixed, for example in an **ISet** type collection; and these objects are loaded in two different sessions..

The most obvious way is to implement **Equals()** or **GetHashCode()** by comparing the identifier value of both objects. If the value is the same, both must be the same database row, they are therefore equal (if both are added to an **ISet**, we will only have one element in the **ISet**). Unfortunately, this approach cannot be used, because the value of the identifier is not assigned before the object is persisted, the newly initialized instances do not have any identifier value. That is why it is recommended to implement **Equals()** and **GetHashCode()** using business key equality which means that the **Equals()** method compares only the properties that form the business key that would identify our instance in the real world.

The implementation of these methods in the **User** class is the following:

```
public class User {
   public override int GetHashCode() {
      return _username.GetHashCode();
   }
   public override bool Equals(object other) {
      if (this == other) {
         return true;
      }
      User u = other as User;
      if (u == null || !_username.Equals(u.Username)) {
         return false;
      }
      return true;
   }
}
```

The domain model is fully implemented, all the mapping documents are ready. The next chapter defines classes in the data access layer of the application.

6 THE `Lasso.DataAccess` NAMESPACE

NHibernate works like a separated layer in the n-tier architecture: it hides the database from the application, makes manipulating persistent data transparent. The application cannot feel the difference using different database systems; nothing has to be changed in the application code if the decision is made to change the database from MySQL to PostgreSQL, for example. The only thing that has to be modified is the application configuration file was discussed before**.** Representing persistence layer with NHibernate is easier than anyone could imagine. In this chapter through the code examples of the Lasso.DataAccess namespace (class diagram on Figure 10.) will be demonstrated how the basic features of a persistence layer can be implemented.

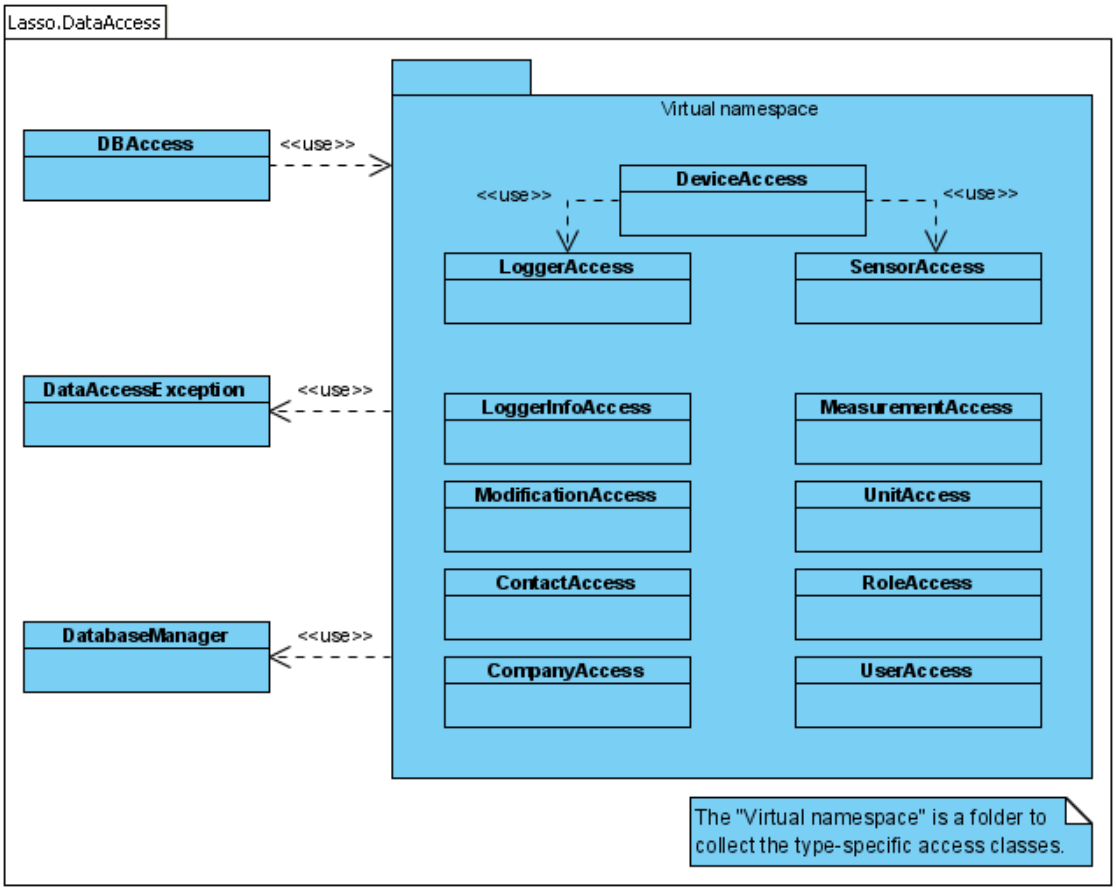

FIGURE 10. Lasso.DataAccess namespace

The classes of this namespace are used to access the database both in the back end of the web user interface and the input communication server. These members of the namespace use and have direct contact with NHibernate.

6.1 Object states

Using NHibernate the objects can be in one of the three different kind of states: transient, persistent, detached.

An object is transient, if it is not associated with any database row. Their state is lost when the reference is lost, they will be collected by the .NET garbage collector. If an object is referenced only by another transient object, by default it is also transient. To persist a transient object, the **ISession.Save()** must be called or a reference from a persistent instance must be created.

The persistent objects have database identity, they are represented by database rows. These objects can be loaded or deleted from and can be updated in the database. Persistent objects are always transactional. Their state is synchronized with the database at the end of a transaction automatically, this process is not visible for the application, it is called transparent transaction-level write-behind.

After closing an **ISession**, the persistent objects lose their association with the persistence manager, and they become detached objects. Modifications are no more guaranteed to be synchronized with the database. The references are still in the application memory but they are no longer managed by NHibernate till they are reused in a new transaction by reassociation with a new persistence manager instance. All objects retrieved in a transaction become detached when the **ISession** is closed or they are serialized.

6.2 Session management

The interface between the application and database is the **ISession**, which is an inexpensive, non-threadsafe object that should be used once, for a single business process, and then discarded. This persistence manager can be obtained from the application's **ISessionFactory**.

An **ISessionFactory** is meant to model the entirety of a conversation between the application and a single database. The only reason to have more than one instance of **ISessionFactory** is if your application communicates with more than one database. In this

case just one database is used; hence, saving the application from a lot of unnecessary overhead all the objects that are related from the **ISessionFactory** instance were made static and their configuration was moved into a static constructor.

```
using System;
using System.Collections;
using NHibernate;
using NHibernate.Cfg;
using Lasso.ObjectLibrary;

namespace Lasso.DataAccess {
   class DatabaseManager {
      private static ISessionFactory _factory;

      static DatabaseManager() {
         try {
            _factory = new Configuration()
               .AddAssembly("Lasso.ObjectLibrary")
               .BuildSessionFactory();
         }
         catch (Exception e) {
            // handle exception
         }
      }
      public static ISession OpenSession() {
         return _factory.OpenSession();
      }
      // ...
   }
}
```

Now, all instances of the **DatabaseManager** class and other classes of the namespace share the same object of the **ISessionFactory** class and use it to create **ISession** instances to open connection with the database.

6.3 Manipulating persistent data

6.3.1 Create

Saving, in other word persisting transient instance, consists of the following steps. Initialization of the object by using constructor with business logic meaning. Newly instantiated objects are, of course, transient. Setting of the properties; if user-defined identifier is used it has to be initialized also. After these the **ISession.Save()** method has to be called in the scope of an open transaction, NHibernate will execute the proper SQL commands.

```
User johnDoe = new User("username", "password", hisEmployer);
// 'hisEmployer' is a Company type instance
johnDoe.FirstName = "John";
johnDoe.LastName = "Doe";

ISession session = DatabaseManager.OpenSession();
Itransaction transaction = session.BeginTransaction();
session.Save(johnDoe);
transaction.Commit();
session.Close();
```

Of course, this solution is just a simple example how to use NHibernate to save domain objects. If layers are separated depending upon their functionality in the application, then this code is not appropriate, because it mixes business logic and data access. During the Lasso development separated classes were used to manipulate the various domain classes. The needed methods were collected and the common features were generalized. There is no way to say to NHibernate what kind of domain object wants to be saved, updated or deleted. It will automatically detect the type of the instance, and create the appropriate SQL commands. Nothing needs to be specified, just called, for example, the **ISession.Save()** method with a domain object as a parameter. Saving an **User** instance in the LASSO code looks like this:

```
// somewhere in the business logic code
User johnDoe = new User("username", "password", hisEmployer);
// 'hisEmployer' is a Company type object
johnDoe.FirstName = "John";
```

```
johnDoe.LastName = "Doe";
UserAccess.Instance.Create(johnDoe);
```

And the definition of the classes and methods which are called while the application is running is the following:

```
public class UserAccess {
   private static UserAccess _instance = null;

   protected UserAccess() { }

   public static UserAccess Instance {
      get {
         if (_instance == null) {
            return new UserAccess();
         }
         return _instance;
      }
   }
   public void Create(User u) {
      DatabaseManager.Create(u);
   }
   // ...
}

class DatabaseManager {
   // ...
   public static void Create(object o) {
      ISession session = _factory.OpenSession();
      ITransaction transaction = null;

      try {
         transaction = session.BeginTransaction();
         session.Save(o);
         session.Flush();
         transaction.Commit();
      }
      catch (Exception e) {
         if (transaction != null) {
```

```
            transaction.Rollback();
         }
         // handle exception
      }
      finally {
         session.Close();
      }
   }
}
```

While using NHibernate errors, usually **NHibernateException** can occur. If the **ISession** throws an exception the transaction immediately should be rollbacked, **ISession.Close()** be called and the **ISession** instance be discarded. Certain methods of **ISession** will not leave the session in a consistent state.

For exceptions thrown by the data provider while interacting with the database, NHibernate will wrap the error in an instance of **ADOException**. The underlying exception is accessible by calling **ADOException.InnerException**.

6.3.2 Retrieving data

There are two techniques to retrieve objects from the database: for loading one single instance the **ISession.Load()** method can be used, for loading a set of suitable instances a search criterium can be specified. There are two ways of selecting a list of objects. One is using the **IQuery** interface which lets the programmer build a query using a SQL like syntax. The other is using the **ICriteria** interface, which is a more object oriented way of creating a query.

ISession.Load() and ISession.Get()

If the **ISession.Load()** is used then the identifier of the desired object has to be known. This method returns a single instance if there is row with the specific identifier; otherwise, throws a **Nhibernate.ObjectNotFoundException.** If the identifier is unknown the **ISession.Get()** should be used which returns with **null** instead of throwing an exception.

The following example illustrates how the retrieving data by identified is implemented in the LASSO M2M Gateway project:

```
public class UserAccess {
   // ...
   public User Get(int id) {
      Company result = null;
      ISession session = DatabaseManager.OpenSession();

      try {
         result = session.Get(typeof(User), id);
         // or the generic version of the method also can be used:
         // result = session.Get<User>(id);
      }
      catch (Exception e) {
         // handle exception
      }
      finally {
         session.Close();
      }
      return result;
   }
}
```

The IQuery interface

NHibernate is equipped with an powerful object-oriented query language which looks like SQL. Although it is possible to use native SQL queries directly with a NHibernate-based persistence layer, it is more efficient to use HQL (Hibernate Query Language) instead which is designed as a minimal object-oriented extension to SQL. It supports many SQL-like features, such as aggregate functions, sum() and max(), for example; and clauses such as group by and order by. The queries, except the names of classes and properties used in them, are not case sensitive.

The **ISession.CreateQuery()** method returns an **IQuery** object, the parameter of the method is the HQL query string. From the **IQuery** object, the result set can be fetched by calling the **IQuery.List()** method.

```
public class UserAccess {
   // ...
   public IList<User> Find() {
      IList<User> result = null;
      ISession session = DatabaseManager.OpenSession();

      try {
         IQuery query = session.CreateQuery(“from User”);
         result = query.List<User>();
      }
      catch (Exception e) {
         // handle exception
      }
      finally {
         session.Close();
      }
      return result;
   }
}
```

The query above returns all the **User** instances. The result set can be narrowed by adding the where clause, it can be ordered, the number of the returned rows can be limited, etc. as can be seen on the examples.

```
// using alias and ordering
IQuery query = session.CreateQuery("from User as u order by u.Id
desc, u.FirstName asc");
// limiting the number of returned rows
query.SetMaxResults(25);

// using where clause
// parameter values in the query string
IQuery query = session.CreateQuery("from User as u where c.Id =
1");

// parameter values added later based on their position in the
string
IQuery query = session.CreateQuery("from User as u where u.Id > ?
and u.LastName like ?");
```

```
query.AddParameter(0, 5);
query.AddParameter(1, "Mc%");

// parameter values added later based on their name in the string
IQuery query = session.CreateQuery("from User as u where u.Id >
:id and u.LastName like :lastName");
query.AddParameter("id", 5);
query.AddParameter("lastName", "Mc%");
```

Using **IQuery** interface is not even a weaker solution than using its relational equivalent, because the most complex queries in SQL can be mapped to HQL. In HQL the domain objects can be worked with, they can be used when the query string is defined, their property values are compared in conditions. This object-oriented approach makes HQL easy to learn and use; on the other hand, it looks better in the .NET code than the native SQL.

The ICriteria interface

However, HQL is extremely powerful, its disadvantage is that the user has to know the syntax of SQL. If he/she is uncomfortable with this, he/she may prefer to build queries dynamically, using an object oriented API, the **ICriteria** query API. To get a reference to an **ICriteria** instance, the **ISession.CreateCriteria()** method is used.

The implementation of the **Lasso.DataAccess.UserAccess.Find()** by using **ICriteria** looks like this:

```
public class UserAccess {
   // ...
   public IList<User> Find() {
      IList<Company> result = null;
      ISession session = DatabaseManager.OpenSession();

      try {
         ICriteria criteria = session
            .CreateCriteria(typeof(User));
         result = criteria.List<User>();
      }
```

```
      catch (Exception e) {
         // handle exception
      }
      finally {
         session.Close();
      }
      return result;
   }
}
```

Of course, there is not much difference at the first look. Below is the illustration of how the examples in the previous chapter look like with the **ICriteria** API.

```
// ordering
ICriteria criteria = session.CreateCriteria(typeof(User));
criteria.AddOrder(Order.Desc("Id"));
criteria.AddOrder(Order.Asc("FirstName"));
// limiting the number of returned rows
criteria.SetMaxResults(25);

// using where clause
ICriteria criteria = session.CreateCriteria(typeof(User));
criteria.Add(Expression.Lt("Id", 5));
criteria.Add(Expression.Like("LastName", "Mc%"));
```

It depends on the user which interface is used to write and execute queries with, or if the programmer wants he/she can deal even with native SQL. However, in the LASSO project using database independent persistence layer implementation was a basic requirement, thus this possibility was not used.

Problem of lazy initialization

In NHibernate 1.2, fetching collections and single-valued associations is lazy in default. It means that the elements of the collection or the proxy objects will be initialized when they are used for the first time. This makes sense for almost all associations in almost all applications. However, lazy fetching poses one problem that the programmer must be aware of: access to a lazy association outside of the context of an open session will result

in an exception, **Nhibernate.LazyInitializationException.** To solve this problem all the later-needed associations have to be accessed before the transaction is commited or non-lazy collection/association should be used by specifying the `lazy="false"` in the mapping files. If too many non-lazy associations are defined in the domain model, then the entire database will be fetched into the memory in every transaction.

Sometimes it is necessary to ensure that a proxy or collection is initialized before closing the **ISession**. It is possible to force initialization by calling the association or collection objects directly, but it is confusing. The static **NHibernateUtil.Initialize()** and **NHibernateUtil.IsInitialized()** methods are suitable for generic purposes:

```
// in an open session
User u = session.Get<User>(5);
// forcing the initialization
// instead of
u.Company.Name; // confusing the meaning
u.OwnModifications.Count; // for readers of code
// use the convinient alternatives
NhibernateUtil.Initialize(u.Company); // more convenient
NhibernateUtil.Initialize(u.OwnModifications);
```

Another option is to keep the **ISession** open until all needed collections and proxies have been loaded. In some application architectures, where the code that accesses data using NHibernate, and the code that uses it are in different application layers, to ensure that the **ISession** is open when a collection is initialized can be a problem.

6.3.3 Modify

Objects may be modified by the application which can be done in two ways. First, when the persistent instance is loaded from the database, the users manipulate it and than the changes are saved into the database using the **ISession.Flush()** method in the same **ISession** in which the object was retrieved. This is mostly not efficient.

Another approach is to load the persistent instance in one transaction, close the **ISession** (the instances bacome detached), send it to the user interface, and after the interaction save the modifications in an other transaction. Applications that use this kind of approach in a high-concurrency environment usually use versioned data to ensure transaction

isolation. NHibernate supports this model by providing the method **Isession.Update()** or the **ISession.SaveOrUpdate()**, which either saves a transient instance by generating a new identifier or updates the persistent state associated with its current identifier.

```
class DatabaseManager {
   // ...
   public static void Modify(object o) {
      ISession session = _factory.OpenSession();
      ITransaction transaction = null;

      try {
         transaction = session.BeginTransaction();
         session.Update(o);
         session.Flush();
         transaction.Commit();
      }
      catch (Exception e) {
         if (transaction != null) {
            transaction.Rollback();
         }
         // handle exception
      }
      finally {
         session.Close();
      }
   }
}
```

Versioning

Chapter 5.2.6 showed how to define the `<version>` element in the mapping document, that describes a property to be used for versioning in optimistic concurrency control. If it is specified and **ISession.Update()** or **ISession.SaveOrUpdate()** methods are used, the ORM framework will take care about the concurrency. If the version number of the detached and persistent instance is different, then the database row was deleted or modified by an other thread or application, thus the instances cannot be reassociated any more, the detached object has to be discarded. NHibernate throws an exception.

If the programmer wants he/she can check the version values manually in the following way that can be seen in the example below. It is useful to know that NHibernate will still update version numbers.

```
// user1 is an instance loaded by a previous ISession
session = _factory.OpenSession();
transaction = session.BeginTransaction();
int oldVersion = user1.Version;
session.Load(user1, user1.Id);
if (oldVersion != user1.Version) {
   throw new StaleObjectStaleException();
}
// modify user1 instance
// ...
session.Flush();
transaction.Commit();
session.Close();
```

Of course, if the programmer operates in a low-data-concurrency environment and does not require version checking, it may be skipped.

6.3.4 Delete

ISession.Delete() will remove an object's state from the database. This method makes the persistent instance transient; the application still can hold a reference to the object.

```
class DatabaseManager {
   // ...
   public static void Delete(object o) {
      ISession session = _factory.OpenSession();
      ITransaction transaction = null;
      try {
         transaction = session.BeginTransaction();
         session.Delete(o);
         session.Flush();
         transaction.Commit();
      }
```

```
      catch (Exception e) {
         if (transaction != null) {
            transaction.Rollback();
         }
         // handle exception
      }
      finally {
         session.Close();
      }
   }
}
```

Many objects can also be deleted at once by passing a NHibernate query string to **ISession.Delete()**, for example:

```
string queryString = string.Format("from User as u where
u.Username like '{0}' or u.Id <= {1}", "use%", 4);
session.Delete(queryString);
```

7 USE OF THE IMPLEMENTED LIBRARIES

After the domain classes are implemented to work together with NHibernate and the persistence layer is also ready, the development of the business logic layer and presentation layer of the application can be started. Because NHibernate hides the whole data access process, the upper layers can see just the “virtual database”, and they just use the offered methods of the **Lasso.DataAccess** namespace's classes.

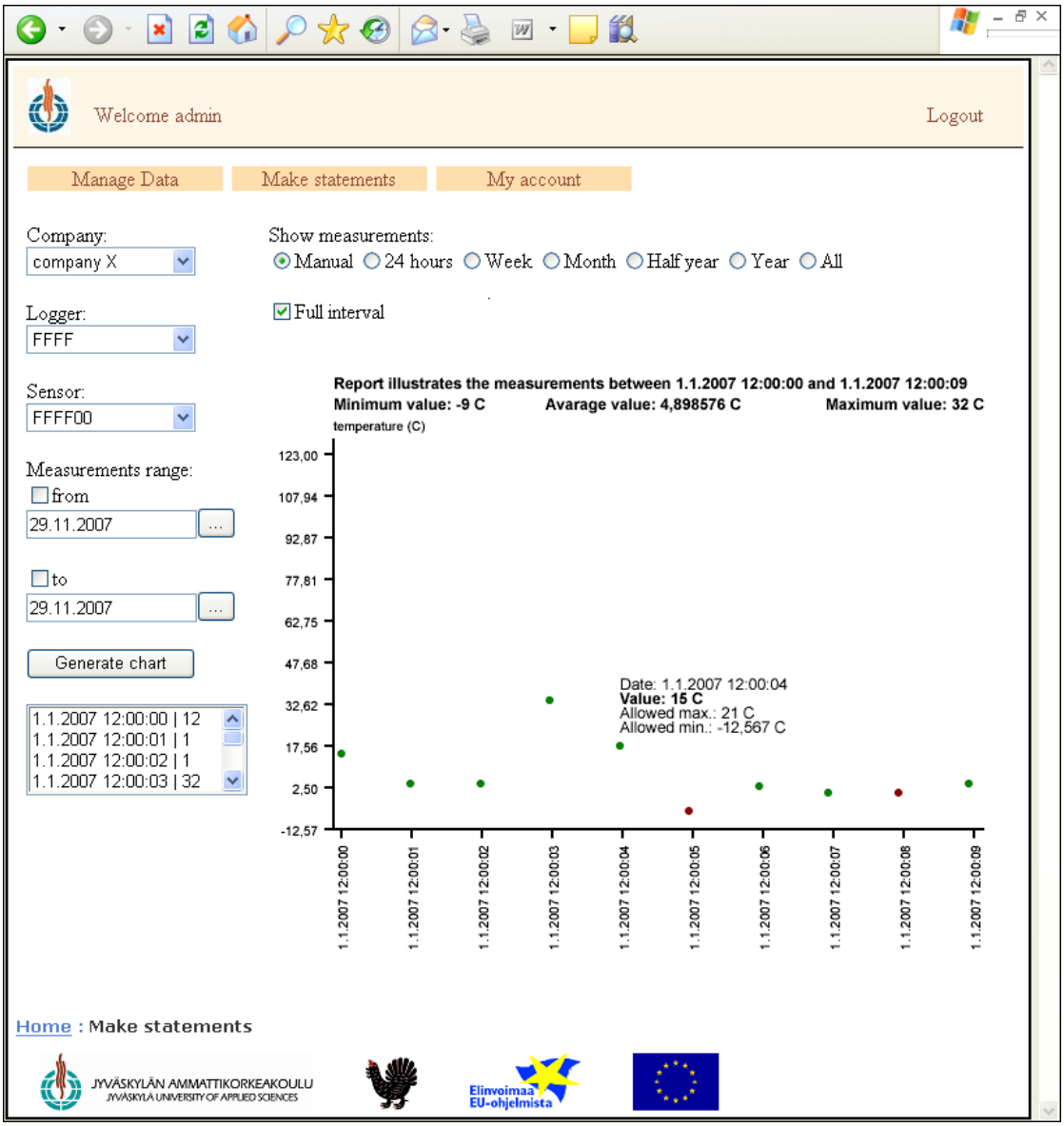

FIGURE 11. The web user interface of LASSO M2M Gateway application

On the Figure 11. the web user interface of the LASSO project can be seen. This screen shot shows an example chart which was made by the report module of the application.

In the background the business logic selects the requested **Measurement** instances using by the **Find()** method of the **Lasso.DataAccess.MeasurementAccess** class. The method executes NHibernate queries (see Chapter 6.3.2) to select the objects of the different search criteria from the database. After the objects are retrieved into a **System.Collection.Generic.IList<Measurement>** collection, their data is processed and used by the appropriate class of **Lasso.Report.SVG** namespace, which creates the graphical report. After all the report file (`*.svg`) is displayed in a frame on the proper ASP.NET page.

8 SUMMARY

NHibernate is an object-relational mapping tool for Microsoft .NET platform, it is an open source software that is distributed under the GNU Lesser General Public License which allows to use it in open-source or even commercial projects. It is widely used and actively developed, well supported on developer's forums especially by Java programmers, but fortunately more and more .NET specialists start using it.

NHibernate supports a large number of database management systems, it ensures that the programmer does not have to write any SQL commands. Changing the DBMS can be done easily, because it does not have effect on the application code. The domain model can be developed independently from the underlying relational database, the classes are flexible for changes in the requirements of the system. NHibernate supports collections, composite types, relations between objects of classes, such as inheritance or associations. The methods were illustrated and used in this work just scratched the surface of the set of opportunities that are offered by NHibernate, however they were enough to show how multiple and professional tool this framework is.

8.1 Results

In the LASSO M2M Gateway project NHibernate gave freedom for the programmers during the development. Even though the domain model and implementation of the database schema was changed due to new requirements, developers did not need to take care of it manually. Reimplementation of the affected code parts was absolutely unnecessary, because the mapping was automatically handled by NHibernate; the programmers could focus on their tasks. Thus the development became more flexible, and the time needed for the implementation was shortened.

Because of the work with NHibernate the goals of the project were reached, all the requirements were implemented and fulfilled.

8.2 Further improvements

There are some open questions on the field of lazy initialization and loading performance, which do not affect the first release of the application, but can be solved in the future. The

problem is when an object is retrieved from the database, its associations are initialized, but not the user-defined type collections of the associated instances. Thus navigation on the references is quite limited, on the other hand the performance of loading is better than it would be with fulfilled collections. After tests focusing on this area of the application some changes could be done, but then all of the upper layers which use these associations and collections should be reimplemented to use the benefits of the new initialization technique.

An other problem was found during the implementation that the difference between application transactions and database transactions is large. Because of the waiting time for user interaction, the database transactions are closed right after they did their tasks, there is no way to handle more then one single database task in an application transaction. Hence, the business logic layer of the application has to take care of handling coherent tasks. In the future, the problem could be solved using a middle layer or framework between the business tier and persistence layer, for example Spring.NET, or breaking the n-layered system architecture and locating the implementation of handling coherent business tasks in the persistence layer.

8.3 Personal experience

My personal experience with NHibernate is good. I found the framework easy to use, I got a lot of ideas, help and support on the official forum and on non-official ones as well. After collecting the requirements of the project and drafting the goals, the implementation part was quite convenient. Even though many of the requirements were changed during the development, the code did not change much, the applications could not feel these modifications and could be finished in the available period of time.

8.4 Conclusion

After all, my impression of NHibernate is positive. The implementation of the n-tier architecture application went well, some minor open questions left, but they did not affect reaching the goals. My opinion is that for all the professional applications using persistence layer with object-relational mapping, NHibernate is a wise choice.

REFERENCES

C# Language Specification, Version 3.0, 2007. http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/csharp%20language%20specification.doc [referenced to 28.11.2007]

Wikipedia, the free encyclopedia, 2007, http://en.wikipedia.org/wiki/Object-relational_mapping [referenced to 28.11.2007]

Yang, J. 2002. "Boosting Your .NET Application Performance", http://www.developerfusion.co.uk/show/3058/2/ [referenced to 28.11.2007]

Hibernate – Relational Persistence for Java and .NET, http://www.hibernate.org/ [referenced to 28.11.2007]

NHibernate Reference Documentation, 2007. http://www.hibernate.org/hib_docs/nhibernate/1.2/reference/en/pdf/nhibernate_reference.pdf [referenced to 28.11.2007]

Bauer, Ch., King, G. 2005 "Hibernate in Action", Manning

Gethland, J. 2004a "NHibernate", http://www.theserverside.net/tt/articles/showarticle.tss?id=NHibernate [referenced to 28.11.2007]

Gethland, J. 2004b, "NHibernate – Part 2", http://www.theserverside.net/tt/articles/showarticle.tss?id=NHibernateP2 [referenced to 28.11.2007]

Coulon, X., Brousseau, Ch. 2004 "Hibernate Simplifies Inheritance Mapping", http://www.ibm.com/developerworks/java/library/j-hibernate/ [referenced to 28.11.2007]

Rajshekhar, A. P. 2006a "Hibernate: HQL in Depth", http://www.devarticles.com/c/a/Java/Hibernate-HQL-in-Depth/ [referenced to 28.11.2007]

Rajshekhar, A. P. 2006b "Hibernate: Criteria Queries in Depth", http://www.devarticles.com/c/a/Java/Hibernate-Criteria-Queries-in-Depth/ [referenced to 28.11.2007]