CX-Hibernate

Hibernate for C++

REFERENCE MANUAL

Version: 0.9.0
Status: Alpha
Author: ir. W.E. Huisman

Contents

INTRODUCTION

CXHibernate is a database framework to communicate with a persistent object store. Most commonly this is a database, but also a filestore or a vanilla store on the internet are possibilities to persist objects. Because CXHibernate is a C++ framwork, it uses C++ objects. These objects can be stored, retrieved, updated or deleted from SQL databases that are interfaced throug the general ODBC standard. All SQL databases have such a general Open Database Connectivity layer as defined by the Microsoft ODBC standard.

Working with a database can be a difficult and time-consuming task. Not only is their the task of mapping object-oriented classes, but also all the details of programming the low level operations of SELECT-ing, INSERT-ing, UPDATE-ing and DELETE-ing the objects in/from the database.

Hibernate is a paradigm that greatly simplifies the tasks of dealing with a database from the programmers perspective. Although it does not exempt him or her from dealing with database details, the standard workflow of working with persistent objects is quite easy. It acts as a go between layer between your application and the database and it's drivers. CXHibernate does support a number of database platforms, datatypes and Object Relational Mappings. As such, it is a ORM (= Object Relational Mapper).

If you are new tot CXHibernate suggested reading is at least

- Chapter 1: The architecture of Hibernate
- Chapter 2: The configuration files
- Chapter 3: A basic "Hello World" example

After you have familiarized yourself with the basic, suggested reading continues with:

- Chapter 4: The Hibernate basic operations (Load, Insert, Update, Delete)
- Chapter 5: The rules for persistent classes
- Chapter 6: On basic O/R mapping and inheritance
- Chapter 7: Mapping associations in CX-Hibernate
- Chapter 8: Filters and how to use them to load objects
- Chapter 9: Transactions and larger batches
- Chapter 10: Interception events on the objects themselves
- Chapter 11: Fundamental datatypes, and how they are supported by SQLComponents
- Chapter 12: On native SQL and queries through the use of SQLComponents
- Chapter 13: Details of the config.xml mapping files
- Chapter 14: Helpful tools

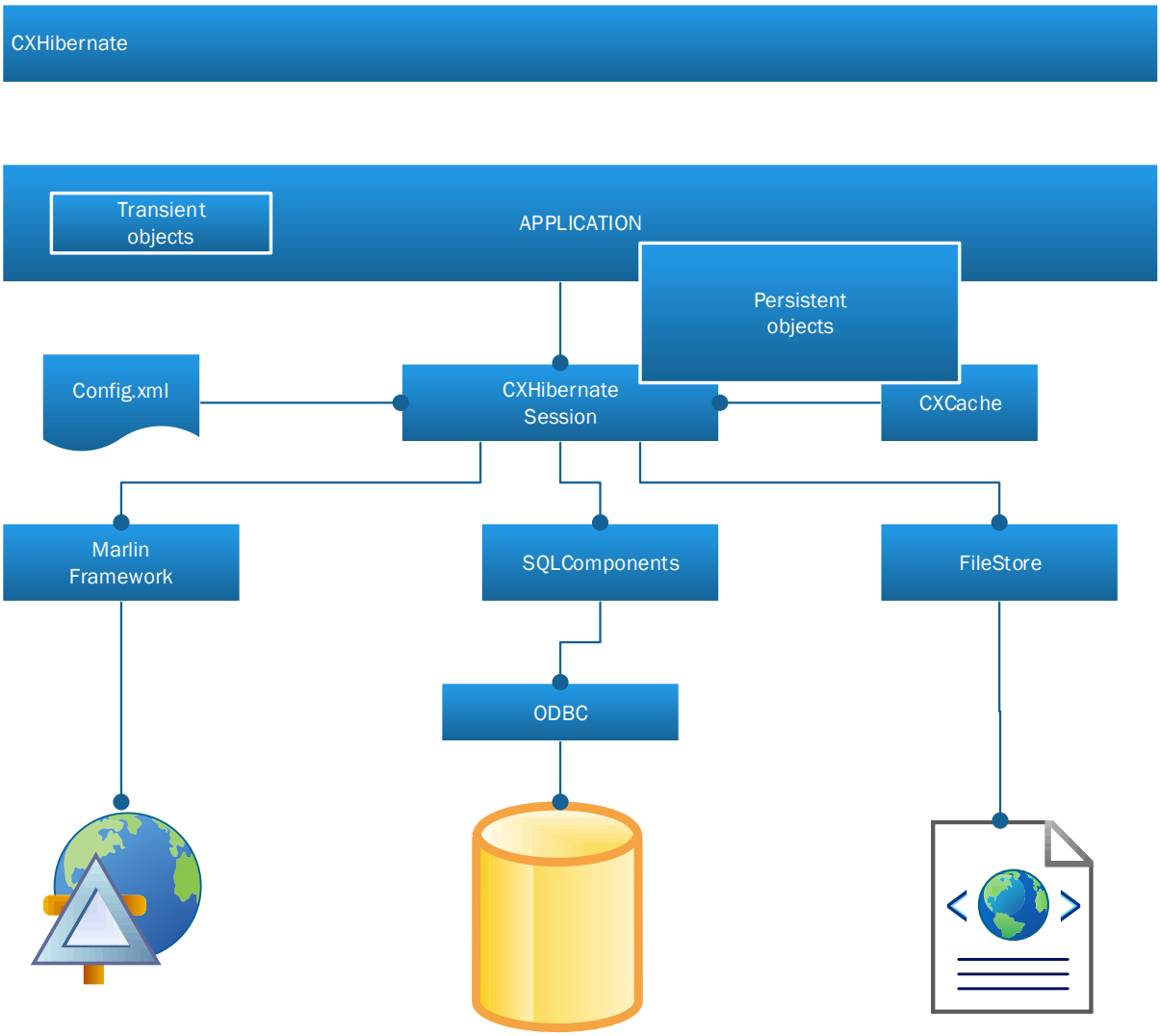
And at last there are appendices:

- Appendix 1: Fun with SQLVariant's
- Appendix 2: Building an IIS application

1. ARCHITECTURE

The central working object of the CXHibernate architecture is the "session". A session is your unit of work that gives you access to the object caches, the database, the filestore and (through the internet) other datastores at a different network location.

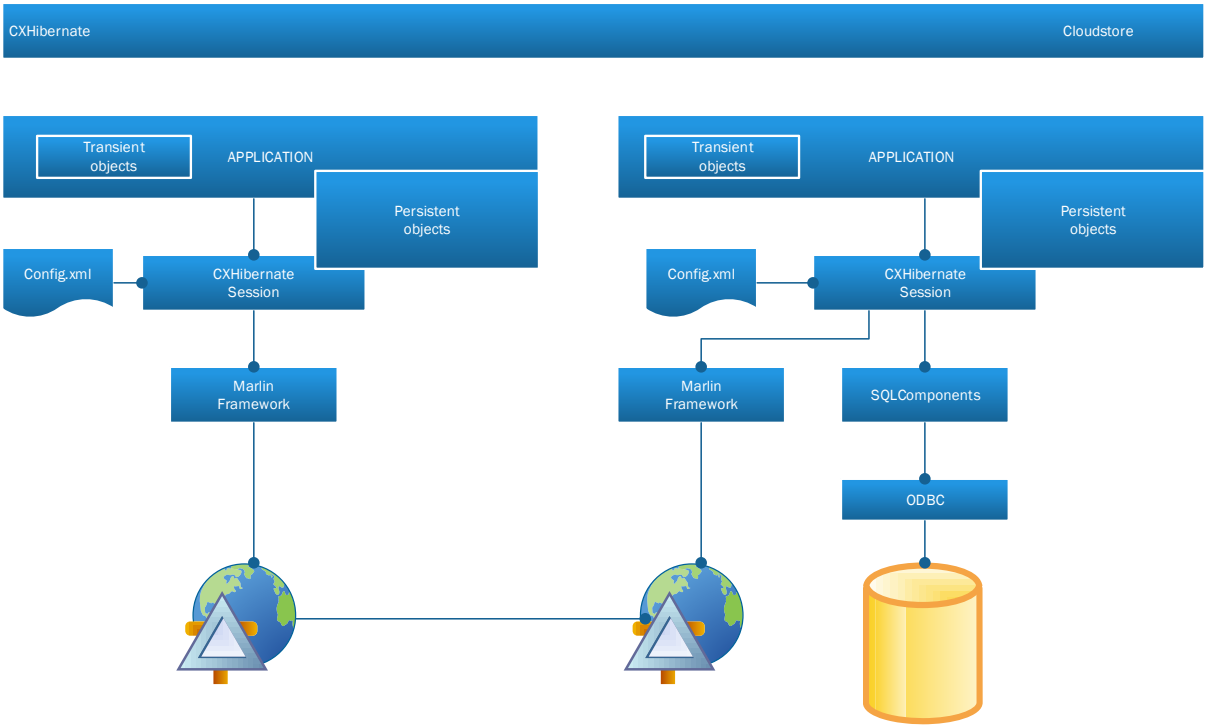
Objects that are made persistent can be handled directly from the application as if the where 'regular' objects. They can be 'found' through the session. The session wil try to find the objects in the cache at first, and in a second attempt at a different stored 'location'.



Objects that are not kept track of are referenced as ‘transient’ objects. Meaning that they will ‘go away’ when the program closes and are not persisted in a database, internet or file storage layer. Handeling the objects is no different in each of these three cases.

The config.xml file (default ‘hibernate.cfg.xml’) describes the data classes in your application and in the storage layers

You can chain two applications together to form a 'cloudstore'. The client side will request objects from the server side that resides 'somewhere-in-the-cloud'. Besides the configuration of the application, there is no difference from storing and retrieving objects from a database. This cloudstore configuration is described in the following image:


CXHibernate
Cloudstore
Transient objects
APPLICATION
Persistent objects
Config.xml
CXHibernate Session
Marlin Framework
Transient objects
APPLICATION
Persistent objects
Config.xml
CXHibernate Session
Marlin Framework
SQLComponents
ODBC

2. CONFIGURATION

The standard configuration of your application is in general contained in the “hibernate.cfg.xml” file in the root directory your application. This is a general XML file with the definition of all of the classes in your application, their attributes and there associations. Loading this file is transparent when you use the default name.

```
CXSession* session = hibernate.GetSession();
```

Any other name can be loaded with the general interface when requesting a new working session. This works by requesting an explicit session from an alternate configuration, as in:

```
CXSession* session = hibernate.LoadConfiguration(“ses”, “C:\Path-to-app\My_config.xml”);
```

Is enough to get you going. Alternatively you may specify a different file as an argument to this call. The *.cxh extension of this file is merely a convention, instead of a requirement. The XML configuration file holds the general para meters for the application and the sessions, and also the definition of all classes. This looks like:

```
<hibernate>
  <strategy>standalone</strategy>
  <logfile>C:\TMP\My_hibernate_logfile.txt</logfile>
  <loglevel>6</loglevel>
  <database_use>use</database_use>
  <class>
    <name>country</name>
    <schema>data</schema>
    <table>country</table>
    <discriminator>cty</discriminator>
    <attributes>
      <attribute name="id" datatype="int" generator="true"isprimary="true" />
      <attribute name="name"  datatype="string" maxlength="100" />
      <attribute name="inhabitants" datatype="int" />
      <attribute name="continent" datatype="string" maxlength="20" />
    </attributes>
    <identity name="pk_country">
      <attribute name="id" />
    </identity>
    <generator name="country_seq" start="1" />
  </class>
</hibernate>
```

In fact: this is al that is needed for the example in the next chapter.

As you can see: the configuration file has a few general settings, and then contains one or more classes and there structure. Whether this be stand-alone classes without any object-oriënted hierarchy or complex hierarchies, class associations, indices and the rest.

Most basic is the fact that the class description names all transient attributes in your class (and thusly in the database table). Of course, your application’s objects can have more data members than just these attributes, but these are the one that will get persisted in the database.

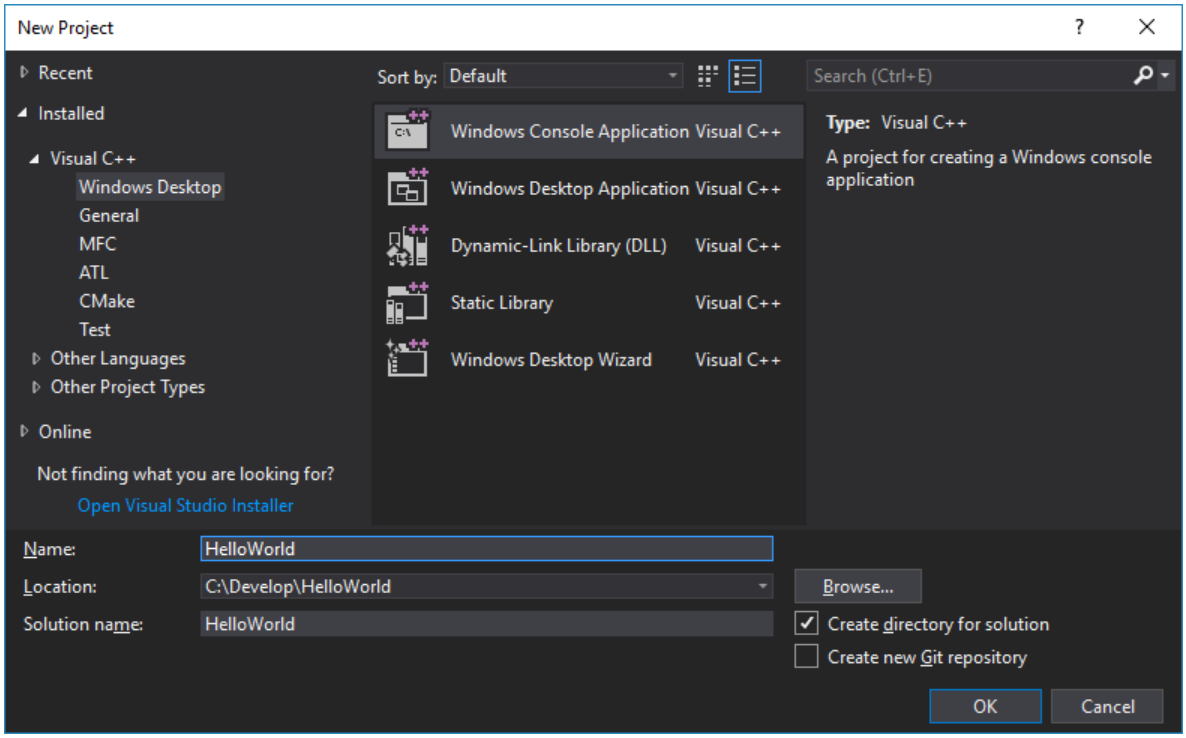
Special care goes to the primary key column (in this case “id”). New instances of objects are created by the generator (starting with the number ‘1’). In the database this column will be part of the primary key, thus forming the identity of the object and of the record in the database table.

Business keys – and so primary keys – can be made up of multiple columns, but one of these columns is assigned to the sequence generator with the ‘generator=”true”’ attribute.

3. A BASIC "HELLO WORLD" EXAMPLE

After a long standing tradition of introducing programmers to a new paradigma, we will program a database version of 'Hello World!" with CXHibernate.

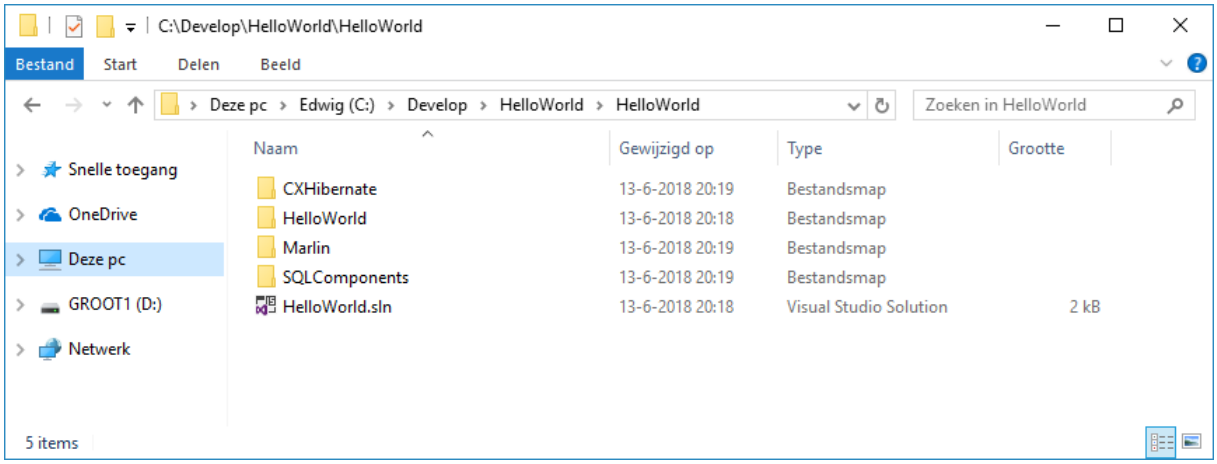
This walk through begins with a new solution directory "HelloWorld" and a solution file in Visual Studio 2017 (any version of Visual Studio will do). We begin with a standard "Windows Console Application". Be sure to 'unselect' the 'Create new Git repository' option.



From github at https://github.com/edwig/cxhibernate we add the following component directories:

- CXHibernate
- SQLComponents
- Marlin

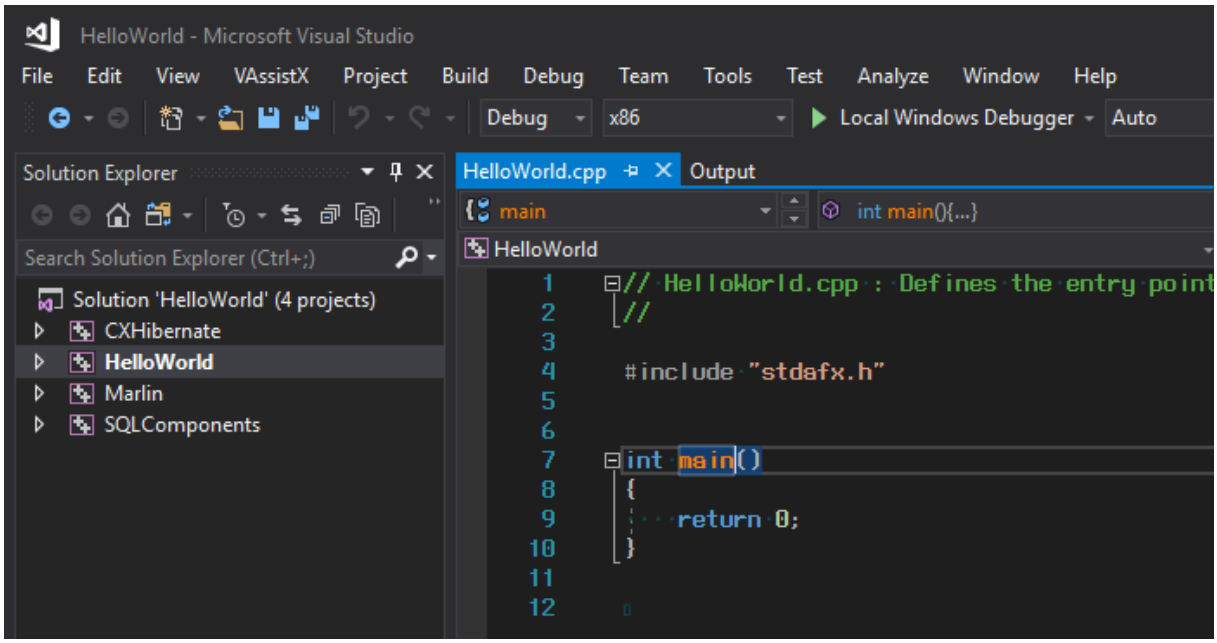
After this inclusions the solution directory should look something like:



(Sorry for the Dutch explorer, and yes 'GROOT' is not a walking tree, it does mean 'BIG')

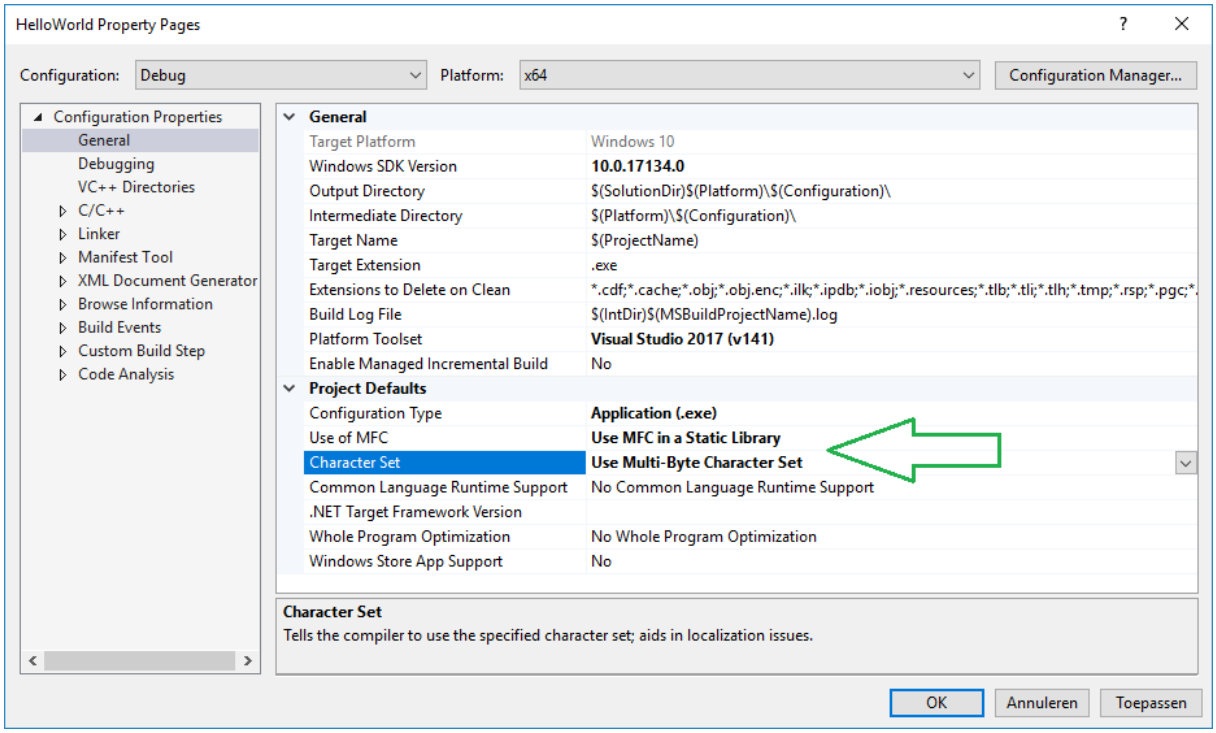
After we have copied the three component directories, we can include the project files of these components in our solution. Just use the "Add…" and "Existing project…" options on the solution level of the "Hello World" solution

After the inclusion of the three project files, your solution should loke something like:



Before we can now begin programming in our “HelloWorld.cpp” file, we need to change some of the project settings, to be able to use the three added components. These are the settings we need to make:

1) Change the “Use of MFC” to “Use MFC in a Static Library”
2) Change “Character Set” to “Use Multi-Byte Character Set”

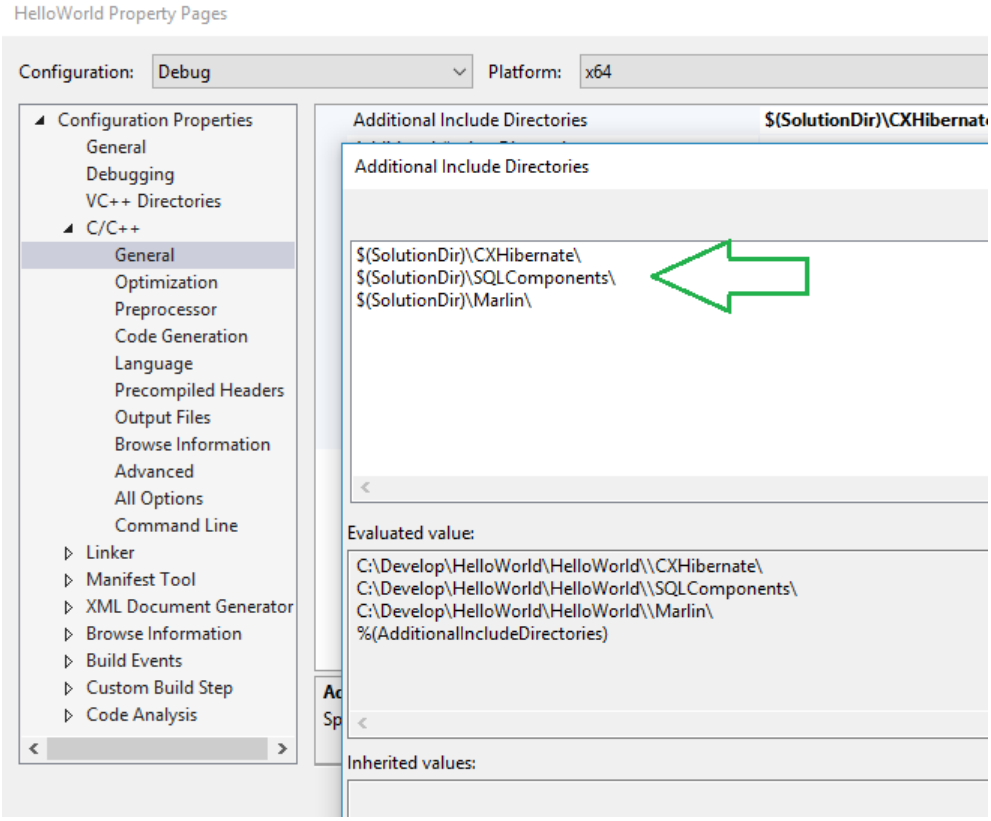


The Hibernate modules are all compiled to be used as static linked libraries. This was done to escape from the ‘DLL Hell’ when installing an application. But you can change that of course at your own leasure if you so please.
Secondly the whole Framework was built in Western Europe with no need or emphasis on Unicode and further Internationalisation. So everything currently only works under the MBCS character set.
A Unicode UTF-8 or UTF-16 version is on the whish list.

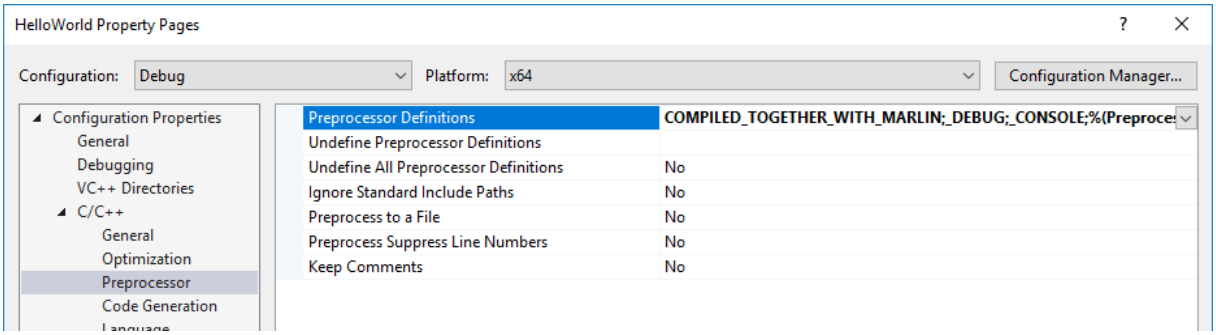
We proceed with the include paths needed for our project. The extra components and their header files need to be found by the compiler so we add the following paths:

- $(SolutionDir)CXHibernate\
- $(SolutionDir)SQLComponents\
- $(SolutionDir)Marlin\

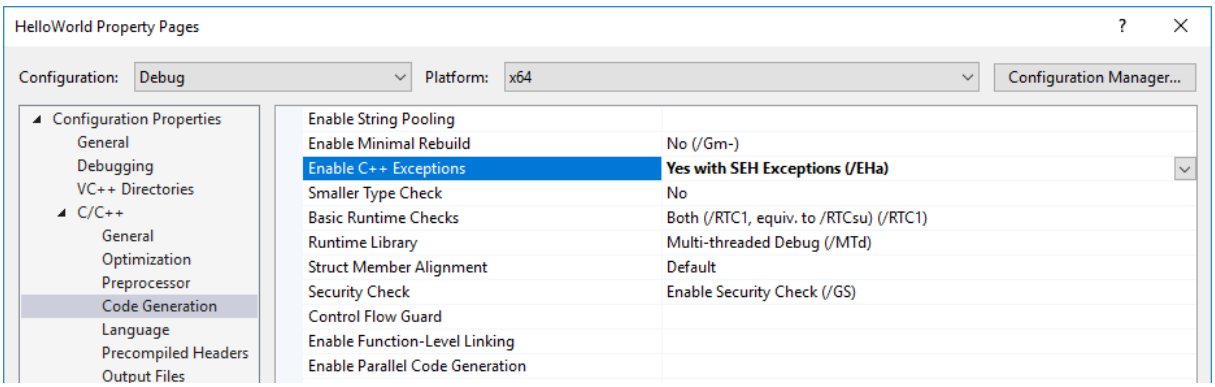
This is done on the C++ General properties page, on the first line “Additional Include Directories”:



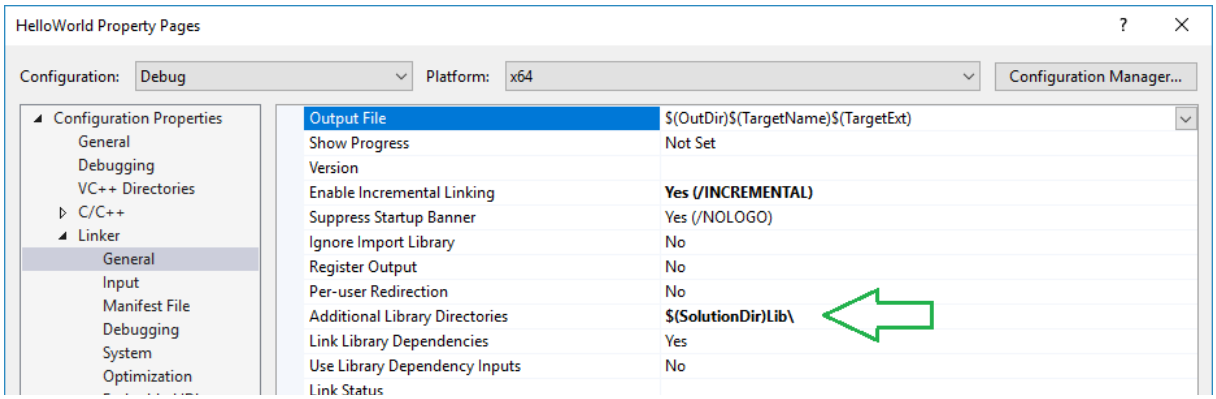
On the “Preprocessor” page we add “COMPILED_WITH_MARLIN” to the preprocessor definitions:



On the “Code Generation” page we enable the asynchroneous exceptions for the SQLComponents and Marlin type exceptions:



A last step is to add the path to the “Lib” directory for linking with the resulting libraries of the “Marlin”, “SQLComponents” and “CXHibernate” modules. Go to the “Linker / General” page and fill in the “$(SolutionDir)Lib\” path at the “Additional Library Directories” setting:



Ok. We’re good to go. You can now in essence compile the application, but first we need to add code to our “main()” function, and add a persistent class called “Country” to a our application.

To create a persistent class real quick, add the configuration file from chapter 2, to our “Hello World” directory and run the “CXH2CPP” utility against it from the command line with the option:

```
CXH2CPP Country
```

This wil generate the “country.h”, “country.cpp”and “country_cxh.cpp” files. Include these files in your “Hello world project”.
Before you can compile them you need to make one more modification, in this case to your “stdafx.h” file. This is what you must add at the end of the file:

```
#include <afx.h>
#include <SQLComponents.h>
#include <CXHibernate.h>
#include <Marlin.h>
```

Not only will this allow you to use MFC, but also CXHibernate. Also the names of the specific libraries to your configuration and platform will be automatically configured in Visual Studio.
Compiling one single file or multiple files will now result in auto linking to the libraries:

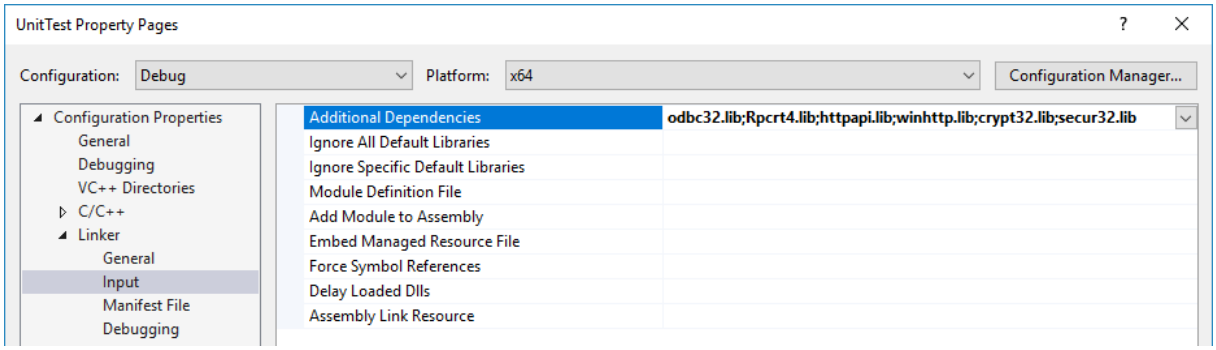
```
1>------ Build started: Project: HelloWorld, Configuration: Debug x64 ------
1>stdafx.cpp
1>Automatically linking with SQLComponents_x64D.lib
1>Automatically linking with CXHibernate_x64D.lib
1>Automatically linking with Marlin_x64D.lib
1>country.cpp
1>country_cxh.cpp
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Still, we cannot build the needed runtimer if we do not specify an extra mandatory set of MS-Windows components that are needed by the Marlin and SQLComponents framework. Otherwise we would get a bunch of “Unresolved external symbol” errors from the system linker.

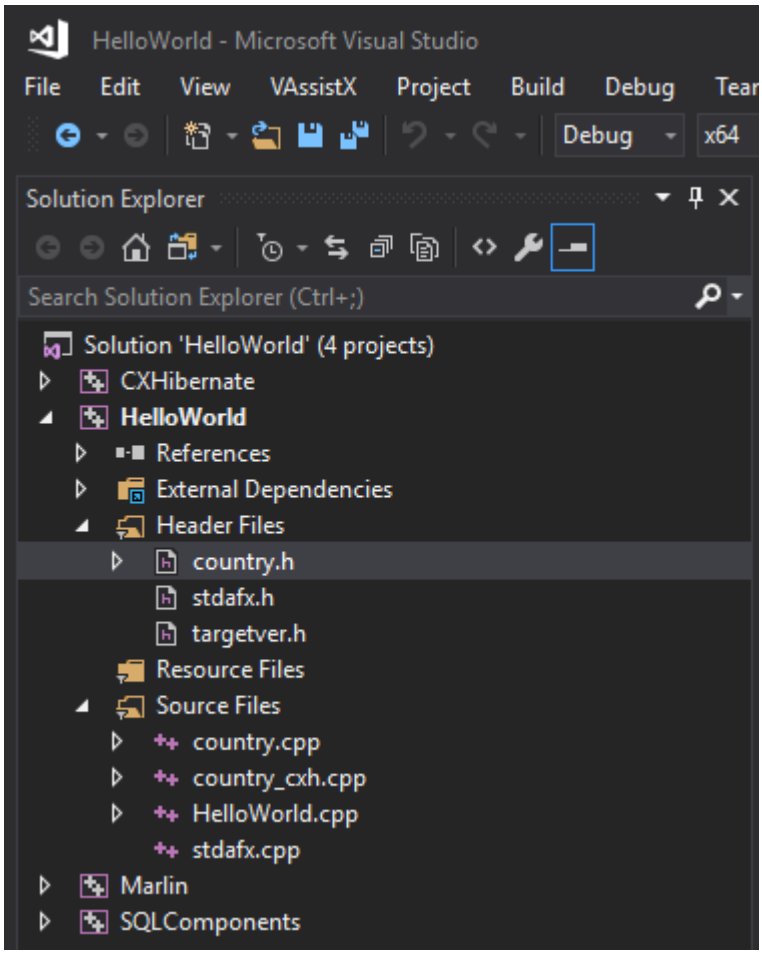
The extra components are:

- odbc32.lib For ODBC and ODBC-Manager functions
- Rpcrt4.lib Needed for the generation of Microsoft GUID’s
- httpapi.lib Needed for the server access to the HTTP service protocol
- winhttp.lib Needed for the client access to the HTTP protocol
- crypt32.lib Needed for encrypted webservices
- secur32.lib Needed for

Add them to the “Linker / Input” page of the project file



OK, now we have everything. Our project should look like:



And everything should compile fine, but for the fact that it does not do anything (yet).

But first let take a peek at the generated files for our “country” class.
This is the implementation *.CPP file

```
// Implementation file for class: Country
// Automatically generated by: CX-Hibernate
//
#include "stdafx.h"
#include "Country.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
// CTOR for class
Country::Country()
{
  // Things to do in the constructor
}

// DTOR for class
Country::~Country()
{
  // Things to do in the destructor
}
```

The interface file (*.H)

```
// Interface definition file for class: Country
// Automatically generated by: CX-Hibernate
// File: country.h
//
#pragma once
#include <CXObject.h>
#include <bcd.h>
#include <SQLDate.h>
#include <SQLTime.h>
#include <SQLTimestamp.h>
#include <SQLInterval.h>
#include <SQLGuid.h>
#include <SQLVariant.h>

class Country : public CXObject
{
public:
  // CTOR of an CXObject derived class
  Country();
  // DTOR of an CXObject derived class
  virtual ~Country();

  // Serialization of our persistent objects
  DECLARE_CXO_SERIALIZATION;

  // GETTERS
  int        GetId()              { return m_id;               };
  CString    GetName()            { return m_name;             };
  int        GetInhabitants()     { return m_inhabitants;      };
  CString    GetContinent()       { return m_continent;        };

protected:
  // Database persistent attributes
  int        m_id                 { 0 };
  CString    m_name              ;
  int        m_inhabitants        { 0 };
  CString    m_continent         ;

private:
  // Transient attributes go here
};
```

And on the following page we find the generated “country_cxh.cpp” file. This is the place where we do our serialization and deserialization. This comes in the place for where other variants of Hibernate can do reflection. C++ has no metadata, so the serialization is done by these macro’s.

```
// (De-)Serializing factories for class: Country
// Generated by CX-Hibernate cfg2cpp tool
//
#include "stdafx.h"
#include "Country.h"
#include <SQLRecord.h>
#include <SOAPMessage.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

BEGIN_XML_SERIALIZE(Country,CXObject)
  CXO_XML_SERIALIZE(int     ,m_id          ,"id"          ,XDT_Integer);
  CXO_XML_SERIALIZE(CString ,m_name        ,"name"        ,XDT_String);
  CXO_XML_SERIALIZE(int     ,m_inhabitants ,"inhabitants" ,XDT_Integer);
  CXO_XML_SERIALIZE(CString ,m_continent   ,"continent"   ,XDT_String);
END_XML_SERIALIZE

BEGIN_XML_DESERIALIZE(Country,CXObject)
  CXO_XML_DESERIALIZE(int     ,m_id          ,"id"          ,XDT_Integer);
  CXO_XML_DESERIALIZE(CString ,m_name        ,"name"        ,XDT_String);
  CXO_XML_DESERIALIZE(int     ,m_inhabitants ,"inhabitants",XDT_Integer);
  CXO_XML_DESERIALIZE(CString ,m_continent   ,"continent"   ,XDT_String);
END_XML_DESERIALIZE

BEGIN_DBS_SERIALIZE(Country,CXObject)
  CXO_DBS_SERIALIZE(int       ,m_id          ,"id"          ,XDT_Integer);
  CXO_DBS_SERIALIZE(CString   ,m_name        ,"name"        ,XDT_String);
  CXO_DBS_SERIALIZE(int       ,m_inhabitants ,"inhabitants",XDT_Integer);
  CXO_DBS_SERIALIZE(CString   ,m_continent   ,"continent"   ,XDT_String);
END_DBS_SERIALIZE

BEGIN_DBS_DESERIALIZE(Country,CXObject)
  CXO_DBS_DESERIALIZE(int     ,m_id          ,"id"          ,XDT_Integer);
  CXO_DBS_DESERIALIZE(CString ,m_name        ,"name"        ,XDT_String);
  CXO_DBS_DESERIALIZE(int     ,m_inhabitants ,"inhabitants",XDT_Integer);
  CXO_DBS_DESERIALIZE(CString ,m_continent   ,"continent"   ,XDT_String);
END_DBS_DESERIALIZE

BEGIN_DESERIALIZE_GENERATOR(Country)
  CXO_DBS_DESERIALIZE(long    ,m_id          ,"id"          ,XDT_Integer);
END_DESERIALIZE_GENERATOR

// Static factory to create a new object if this class
DEFINE_CXO_FACTORY(Country);
```

Now finally we can get some work done. We can now start to fill in our “main()” function of the application. Request a session from the global “hibernate” object and load a first country with the id=1 into memory. If all goes well, we can directly begin calling methods of the object, and print a “Hello world” on the console.
After we have done our work, optionally we can now close the session.

Of course you should also have an ODBC connection named “hibtest” to the test database of the CX-Hibernate project. In this case a Firebird 3.0 database, with the “COUNTRY” table in it.
(Filled from: https://simple.wikipedia.org/wiki/List_of_countries_by_population)

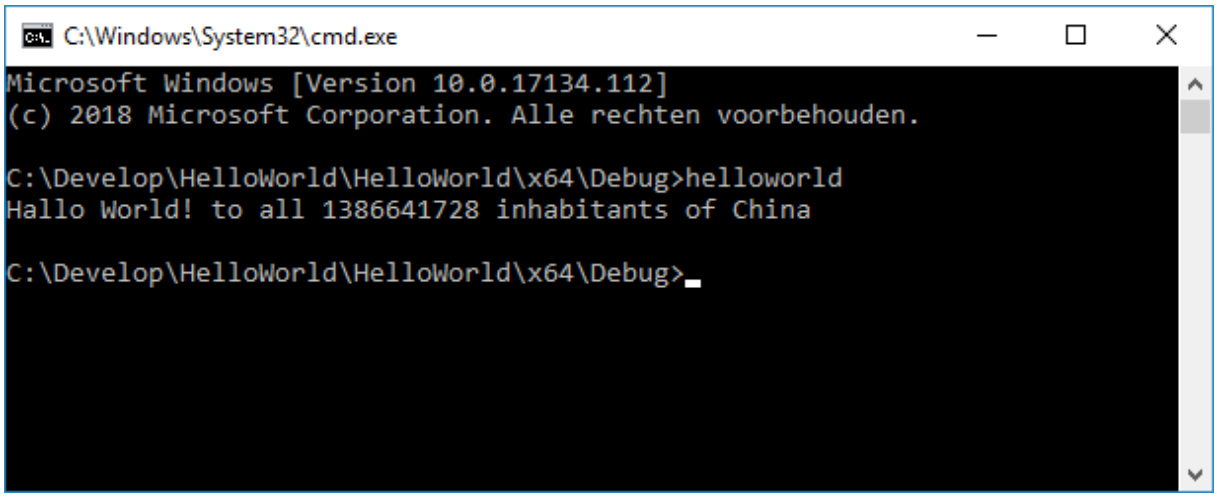
```
// HelloWorld.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "Country.h"
#include <CXHibernate.h>
#include <CXSession.h>

int main()
{
  CXSession * session = hibernate.CreateSession();
  if(session)
  {
    // Set a database session
    session->SetDatabaseConnection("hibtest","sysdba","altijd");

    Country* land = (Country*)session->Load(Country::ClassName(),1);
    if(land)
    {
      printf("Hallo World! to all %d inhabitants of %s\n"
            ,land->GetInhabitants()
            ,land->GetName().GetString());
    }
    else
    {
      printf("Cannot find a country with id = %d\n",1);
    }
    // And close our session
    session->CloseSession();
  }
  return 0;
}
```

Now compile, copy the “hibernate.cfg.xml” to the runtime directory and run it!
And low and behold!



This concludes our “Hello World” first example. Of course you can go off now and create all kinds of extra test to this simple first program. Suggestions are that you try to:

- Add an extra parameter to the program to request a different country
- Add a load with a filter to request a set of countries and print them all
- Count all the inhabitants in the world and say “Hello” to all of them, spreading ‘peace and happiness’ to the world ☺

4. BASIC OPERATIONS

Now that we have introduced you to CX-Hibernate in the last chapter with a “Hello World” example, lets look at the basic Hibernate operations and their derivates. The basic operations are:

- Load: Get an object from an external store (database, webservice, filestore);
- Insert: Place a new object in hibernation in an external store;
- Update: Change the object in the hibernation store, so that it reflects the one in our app;
- Save: Inserts new objects, or updates existing loaded ones;
- Delete: Delete the object from the hibernation store, and most likely from our application too.

The basic operations are methods of the general CXSession class. Each application’s user must have an active session, in order to be able to perform basic operations on the external store.
Per basic operation, a number of variants and requirements are described in the paragraphs below.

4.1 Loading an object

Loading an object has various overloads in the session. This is done to make it easy on the programmer to take the shortest route to an easy load of the object. The load methods go from simple one parameter loads to a load with a complex set of filters returning a set of objects. These are the load methods of the CXSession:

```
CXObject*   Load(CString p_className,int           p_primary);
CXObject*   Load(CString p_className,CString       p_primary);
CXObject*   Load(CString p_className,SQLVariant*   p_primary);
CXObject*   Load(CString p_className,VariantSet&   p_primary);
CXResultSet Load(CString p_className,SQLFilter*    p_filter);
CXResultSet Load(CString p_className,SQLFilterSet& p_filters);
```

There are so many “Load” operations, as to accomodate the situations that the database objects have primary keys that exist out of more than one database column. In many cases where we have just one “id” column, the first “Load” operation with an “integer” primary key is quite sufficient. In cases where we have compound primary keys, the variants with a “VariantSet” and a “SQLFilterSet” the way to go. And in fact, these variants get called internally by the other “Load” operations.

A second thing to notice is the fact that all the “Load” operations have the name of the class as a first parameter. This parameter can be filled with a static string like e.g. “classname” (case-insensitve!!), but it’s easier and more natural to use the “ClassName” from the object factory of the class. We have already seen such an example in the HelloWorld program of the previous chapter as in:

```
...
Country* land = (Country*)session->Load(Country::ClassName(),1);
...
```

Here are all the load operations:

4.1.1 Load from an integer identifier

The most common method to identify an object in the database – and mostly the fastest – is by using an integer “id” column. If you – your application – happen to now the id, the load is most easily done by using the “Load” variant with the integer primary key as a second parameter, as in the example above.
In general, a network database with all de-personalized “id” numbered identifiers is by far the most performant way of putting a database application together.
That’s why this type of load is recommended;

```
CXObject* Load(CString p_classname,int p_primaryKey);
```

4.1.2 Load from a string identifier

In the case where you have a single string column as the primary key of a class of objects, the optimized form with a CString as second parameter is the quickest way to load an object:

```
CXObject* Load(CString p_classname,CString p_primaryKey);
```

Notice that code tables with ("string identifier", "string tekst_info") pairs often have this type of format in the database. For such code tables this type of load is the easiest.

4.1.3 Load from a single identifier (not being integer or string)

In cases where the primary key column has another datatype than "integer" or "string", you can load an object through a SQLVariant. SQLVariants can hold any datatype that can be put into an ODBC database or can be retrieved from it. For a more elaborate discussion about SQLVariants, please lookup the chapter on "Datatypes" and the "Fun with Variants" appendix.

```
CXObject* Load(CString p_classname,SQLVariant* p_primaryKey);
```

In general this works by first assigning a value to a variant, and then load the object through the variant. For instance, if we supply a GUID in the form of a "p_guid" parameter, we can:

```
...
SQLVariant myGuid(p_guid)
Hardware* pc = (Hardware*)session->Load(Hardware::ClassName(),&myGuid);
...
```

Load the object through the use of a SQLVariant, supplied as the primary key of the object.

4.1.4 Load from a compound set of values

CX-Hibernate is especially not restricted to objects with a single identifier primary key. In fact, most relational database have tables with compound primary keys. That's why it's made easy to work with those objects. If you want to load an object with a primary key, you shall have to supply a compound set of values to load the object. This is done through the VariantSet. In fact it's a simple "std::vector" of SQLVariant's. Loading is done by

```
CXObject* Load(CString p_classname,VariantSet& p_primaryKey);
```

This makes it easy to refer to a compound primary key. In fact, internally CX-Hibernate works always on VariantSet's of values. In the CXObject's primary key, in search functions and in all Load functions. We can directly load an object from a compound key:

```
...
SQLVariant myCompany(p_company);
SQLVariant myLedger(p_ledger);
SQLVariant myTransaction(p_transID);
VariantSet key;
Key.push_back(myCompany);
Key.push_back(myLedger);
Key.push_back(myTransaction);

Payment* paymentToday = (Payment*)session->Load(Hardware::ClassName(),key);
...
```

This looks akward, but in a real application, we already will have a loaded "VariantSet p_key" somewhere around, and the load will only take one line of code.

4.1.5 Load of a set of objects from a single condition filter

In many cases we will want to look for objects, not by their primary key, but by some candidate key. A search form where we fill in one or more filter values is a perfect example of such a load. That's why we can load objects by one (this paragraph) or by a set of filters (next paragraph). Filters can be defined through the "SQLFilter" object and used as the second parameter in a "Load" operation.
This is the form of a "Load" with one filter:

```
CXResultSet* Load(CString p_classname,SQLFilter* p_filter);
```

A few things to keep in mind:

- A “CXResultSet” is in fact a “std:vector<CXObject*>” container. It gives you a bunch of pointers to objects as a result of the load from the datastore.
- A “SQLFilter” has in general an attribute, an operator and a value. For more information on filters, please refer to the chapter “Filters”.

We can specify a filter and then load a set of objects. For instance we could in the “HelloWorld” application load all the countries with more than 10 million inhabitants:

```
...
SQLFilter filter("inhabitants",OP_Greater,10000000)
CXResultSet set = session->Load(Country::ClassName(),&filter);
...
```

Once we loaded the objects, we can simply step through them with a for loop. Because we know the data class beforehand, we can simply cast the CXObject’s to the desired class, as in:

```
...
  for(auto& object : set)
  {
    Country* country = reinterpret_cast<Country*>(object)
    .. do something interessing with the country object ...
  }
...
```

4.1.6 Load of a set of objects from multiple condition filters

An extension of loading objects through a filter, is loading a set of objects through a set of filters. This works exactly as loading through one (1) filter. But instead a “SQLFilterSet” is used.

```
CXResultSet* Load(CString p_classname,SQLFilterSet& p_filter);
```

This makes it possible to provide multiple search conditions at once to the load operation. In fact: this overload of the “Load” operation is more in use then the “Load” from just one filter. We can e.g.:

```
...
SQLFilter filter1("inhabitants",OP_Greater,10000000);
SQLFilter filter2("continent",  OP_Equal, "Asia");
SQLFilterSet filters;
filters.push_back(filter1);
filters.push_back(filter2);

CXResultSet set = session->Load(Country::ClassName(),filters);
...
```

… load all the countries on the continent of “Asia” having more inhabitants than 10 million.

4.2 Inserting an object

This is a two step procedure. Inserting an object requires the creation of an object of a certain persistent class first. After filling the object with data, and perhaps a partial primary key, we can ask Hibernate to insert the object into the external store.

Here is an example of such a two fase insertion:

```
...
CXSession* session = hibernate.CreateSession();
...
Cat* pussy = (Cat*) session->CreateObject(Cat::ClassName());

// Fill in our object with reasonable values
pussy->SetAnimalName("Silvester");
pussy->SetHas_claws(true);
pussy->SetLikesBirds("Tweety");

// Go save it in the database in two tables (Animal and Cat)
bool result = session->Insert(pussy);
...
```

Instead of “Insert” you may also use “Save” as an equivalent.

The “CreateObject” method of the session is called instead of “new”-ing the object. This assures us that the object was created by the object factory and that all necessary operations for the Hibernate framework have been taken care off. **Never “new” your object!**
A side effect of inserting the object into an external store is that the primary key in the base class CXObject will get filled in, which in it’s turn changes the state of the object from “transient” to “persistent”.

Another side effect is that the object is referenced in the object cache. You need not keep track of it in your application for free-ing. The hibernate framework does that for you.

4.3 Updating an object

Updating a single object is quite simple. Just call the session’s “Update” method. That’s it.

Here is an example:

```
...
CXSession* session = hibernate.CreateSession();
...
Cat* pussy = (Cat*) session->Load(Cat::ClassName(),42);

// change the state of the object
pussy->SetColor("black-and-white");

// Go update the external store
bool result = session->Update(pussy);
...
```

4.4 Deleting an object

Deleting a single object is quite simple. Just call the session's "Delete" method. That's it.

Here is an example:

```
...
CXSession* session = hibernate.CreateSession();
...
// Load a certain object
Cat* pussy = (Cat*) session->Load(Cat::ClassName(),13);

// Go delete from the external store
bool result = session->Delete(pussy);
...
```

Please remember: if the deletion goes well (resulting in a 'true' return value), the object is removed from the hibernate caches AND it is destroyed by a "delete" action of the C++ language. Your pointer to the object is then no longer valid!
Only when the "Delete" operation returns a 'false' (whether the object gets not deleted from the database or from the cache) the pointer is still intact. After optionally logging this fact the caller (you!) must call "delete" on the object.

5. PERSISTENT CLASSES

In Hibernate 'Persistent classes' are referred to as those classes that consists of objects that reside in an external store and 'do hibernate' there when the application is not running. In general the objects are stored in one or more – related – database tables. In CX-Hibernate the external store can also be a filestore or an internet webservice server that is internally serviced by CX-Hibernate as well.

In order for CX-Hibernate to cooperate with these persistent classes, they must obey to a set of special rules in general. Some of these rules are found in other Hibernate products as well. Other rules are native to CX-Hibernate, for no other reason that the C++ language does not support meta data and a reflection mechanism like Java and C#. Advocates of the C++ language however do not recon that to be a drawback, but a winning point, as reflection is quite slow.

5.1 Rules

Classes must obey to the following rules:

1. Each class must be derived from the general "CXObject" class.
   Reason for this rule, is the fact that a lot of general functionality is already in the CXObject base class, and deriving from this general 'object' type of class makes 90 % of reflection unnecessary. Creating, loading, tracking changes, inserting, updating and deleting all works through the general CXObject class interface.

2. Each class must have a basic constructor without any parameters.
   The parameter-less constructor is used by Hibernate to create instances of your class. This works through a generalized object factory mechanism that is created at startup link time of your application.

3. Each class must either have no destructor, or a "virtual" destructor.
   A lot of functionality in the CXObject class are overridable by your own class. So there exist already a good number of virtual methods. This creates the necessity of a virtual destructor. Any other destructor will not be called by the C++ language.

4. It is highly recommended to declare the "DECLARE_CXO_SERIALIZATION" macro in the interface declaration in the "<class>.h" file.
   This macro defines a lot of virtual overrides for your class for the object factory, the generator mechanism and the serialization and de-serialization of information out of your database, or out of XML/SOAP messages. It's an easy way of declaring everything at once without forgetting everything.

5. Each class must implement an "object factory"
   The object factory is an small automated class does the "new" of a new object of the class. The object factories are put by the linker in a special loading segment and started by the "hibernate" static object at program startup time, before the "main" is called. As such the programmer is assured that the object factory can be called.
   Also: Please be very aware that only classes that define an object factory can be named in the "hibernate.cfg.xml" configuration file

6. Each class must implement a database and/or a soap serializator and deserializator.
   In order to assign fields from a database record to data member attributes of a class object, you must implement deserializators. See below under the "BEGIN_DBS_DESERIALIZE" and the "BEGIN_XML_DESERIALIZE" macros.
   In order to bring data from class object members to the database you must implement serializators. See below under the "BEGIN_DBS_SERIALIZE" and "BEGIN_XML_SERIALIZE" macros.

7. Classes that have auto generating primary keys in the database must implement a "generator deserializator"
   In order to accept the value of a database sequence or generator, the class must have a specialized de-serializator to upate it's view of the business key with the new value of a generator after the insert operation in the database. Otherwise following updates and deletes on the object will not work.

5.2 The default implementation

When you are following the default implementation. E.g. by generating the class implementation with the CFG2CPP tool, you will get:

- A header file with all data members and default setters and getters for all the data members. The class declaration then contains one (1) extra macro for the declaration of the serialization factory ("DECLARE_CXO_SERIALLIZATION")

- A factory implementation file, containing the serialization and de-serialization of the object for both the database and SOAP messages. It also contains the create-object factory that the hibernate framework uses to create a new object of this class.
  Mostly, you can leave this implementation file alone. Only when you add new attributes to your class (and columns to the database) need you go here and add those attributes. You can either do one of the following:
  a) Create a new set of files with the “cfg2cpp” tool, after you changed the config file and pluck out the *.cxh.cpp file;
  b) Add the attribute by adding lines to the file. Great care has been taken to make the four implementations alike, so that it’s easy to add the lines for a new attribute.

- An implementation file with just a constructor and a destructor. You can now start to begin the implementation of your class right away in this file. No visible overhead in your way.

5.3 Overridable virtual functionality

Apart from the default implementation, there are a number of overridable methods in CXObject, that you can use and add to your class. Here is a list of all overridable methods:

5.3.1 Compare

The standard “**`int CXObject::Compare(CXObject* p_other)`**” compares two objects on basis of their primary keys. You can think of it as if you where comparing two strings by comparing there “**`const char *`**” with the “**`strcmp`**” function. The function returns an ‘int’ that is zero (0), smaller than zero, or greater than zero, depending on the fact whether the object is ‘before’, ‘the same’ or ‘after’ the object pointed to by the p_other parameter.
This compare function obviously fails if the objects are not persistent but transient. The reason for the failure is the fact that transient objects have no filled in primary key. For this reason alone, you can override the “Compare” method of your object.

5.3.2 Hashcode

The hascode is a single value that is calculated from the primary key. This is standard the “**`CString CXObject::Hascode()`**” method. The hascode is used to store the object in the first and second line caches. And of course an attempt is made to find the object by the hashcode before each “Load” operation, bypassing the datastore. Especially for compound keys it’s handy that we have a single hashcode. This makes storing and retrieving objects with compound keys just as fast a objects with a single ‘id’ key.
Care has been taken to guarantee that the hashcodes are unique. In the very unlikely case that they are not, your free to overload this method.

5.3.2 OnLoad / OnInsert / OnUpdate / OnDelete

These “On*” methods are interception events. (Read more about interception events in “Chapter 10. Interception Events”). They fire as triggers just before their respective operations (Load, Insert, Update and Delete). The OnLoad operation returns void, but the other three return a bool. Returning ‘false’ from your overload will cancel the respective operation.

5.3.3 PreSerialize (Database store)

Very shorty before an object is stored into a database the ‘PreSerialize’ is called. It is a ‘protected’ method and thus only visible for other objects in the class hierarchy. This method is used to fill in the parts of the primary key that are not dependent on a generator. In general this is done in the root class of the hierarchy and that is the reason why you must ALWAYS remember to call the PreSerialize of the direct parent class of your object as in:

```
...
void MyClass::PreSerialize(SQLRecord& p_record)
{
   // Call the PreSerialize of our parent (fills in the primary key!)
   MyParent::PreSerialize(p_record);
   // Go do our overloaded functionality
   ...
}
```

5.3.4 PreSerialize (Internet/Filestore)

Very shorty before an object is stored into a filestore or on the internet through a webservice, the 'PreSerialize' is called. It is a 'protected' method and thus only visible for other objects in the class hierarchy. In general you must ALWAYS remember to call the PreSerialize of the direct parent class of your object as in:

```
...
void MyClass::PreSerialize(SOAPMessage& p_message,XMLElement* p_entity)
{
   // Call the PreSerialize of our parent
   MyParent::PreSerialize(p_message,p_entity);
   // Go do our overloaded functionality
   ...
}
```

5.3.5 PostSerialize (Database store)

Directly after the object has been serialized in the database record, the 'PostSerialize' method is called for the object It is a 'protected' method and thus only visible for other objects in the class hierarchy. When this method is called, the SQLRecord of the database is now filled in with the data from the persistent data attributes of the class. Any operation you would want to do on the database record before letting it continue to the database can now be done.
This is also the fase where the CXObject stores the record as the place where to store any changes. And that is the reason why you must ALWAYS remember to call the PostSerialize of the direct parent class of your object as in:

```
...
void MyClass::PostSerialize(SQLRecord& p_record)
{
   // Call the PostSerialize of our parent (remembers our record!!)
   MyParent::PostSerialize(p_record);
   // Go do our overloaded functionality
   ...
}
```

5.3.6 PostSerialize (Internet/Filestore)

Directly after the object has been serialized in the SOAPMessage, the 'PostSerialize' method is called for the object It is a 'protected' method and thus only visible for other objects in the class hierarchy. Hibernate uses this method to de-serialize the primary key of the object, as it was gotten from an external source. This is done in the root class (CXObject). And that is the reason why you must ALWAYS remember to call the PostSerialize of the direct parent class of your object as in:

```
...
void MyClass::PostSerialize(SOAPMessage& p_message,XMLElement* p_entity)
{
   // Call the PostSerialize of our parent (fills in the primary key!)
   MyParent::PostSerialize(p_message,p_entity);
   // Go do our overloaded functionality
   ...
}
```

5.3.7 PreDeSerialize (Database store)

Very shorty after an object is loaded from a database the 'PreDeSerialize' method is called. It is a 'protected' method and thus only visible for other objects in the class hierarchy. This method is used to fill in the primary key and store the SQLRecord where the object originated from. In general this is done in the root class of the hierarchy and that is the reason why you must ALWAYS remember to call

the PreDeSerialize of the direct parent class of your object as in:

```
...
void MyClass::PreDeSerialize(SQLRecord& p_record)
{
   // Call the PreDeSerialize of our parent (fills in the primary key!)
   MyParent::PreDeSerialize(p_record);
   // Go do our overloaded functionality
   ...
}
```

5.3.8 PreDeSerialize (Internet/Filestore)

Very shorty after an object is loaded from a an external datastore or from an internet webservice, the 'PreDeSerialize' method is called. It is a 'protected' method and thus only visible for other objects in the class hierarchy. This method is used to fill in the primary key. In general this is done in the root class of the hierarchy and that is the reason why you must ALWAYS remember to call the PreDeSerialize of the direct parent class of your object as in:

```
...
void MyClass::PreDeSerialize(SOAPMessage& p_message,XMLElement* p_entity)
{
   // Call the PreDeSerialize of our parent (fills in the primary key!)
   MyParent::PreDeSerialize(p_message,p_entity);
   // Go do our overloaded functionality
   ...
}
```

5.3.9 PostDeSerialize (Database store)

The 'PostDeSerialize' method has presently no function in the CXObject class. It's there just for the sake of isomorphism. But when you override this method, do not formet to call the same method in the direct parent class. So when the framework will start to use this method in a future version, everything keeps working as intended. Just as in:

```
...
void MyClass::PostDeSerialize(SQLRecord& p_record)
{
   // Call the PostDeSerialize of our parent (does nothing for now!)
   MyParent::PostDeSerialize(p_record);
   // Go do our overloaded functionality
   ...
}
```

5.3.10 PostDeSerialize (Internet/Filestore)

The 'PostDeSerialize' method has presently no function in the CXObject class. It's there just for the sake of isomorphism. But when you override this method, do not formet to call the same method in the direct parent class. So when the framework will start to use this method in a future version, everything keeps working as intended. Just as in:

```
...
void MyClass::PostDeSerialize(SOAPMessage& p_message,XMLElement* p_entity)
{
   // Call the PostDeSerialize of our parent (does nothing for now!)
   MyParent::PostDeSerialize(p_message,p_entity);
   // Go do our overloaded functionality
   ...
}
```

6. BASIC O/R MAPPING

CXHibernate knows about three different types of object-relational mapping. The current mapping can be found by querying the “hibernate.GetStrategy()” interface. The mappings are:

- MapStrategy::Strategy_standalone. This is the default strategy, where every object class has exactly one database table, and no class inheritance takes place. All object transactions are always carried out on a 1:1 basis as standard SELECT, INSERT, UPDATE and DELETE actions against the database

- MapStrategy::Strategy_one_table. This is the strategy where you can have linear class inheritance. All attributes of the super class and all derived classes of an object are stored in one record of one database table. This strategy is also known as the “table-per-class-hierarchy” mapping.
  A drawback of this strategy is that it wastes some database space, and that the attributes of the subclasses cannot have a NOT-NULL constraint in the database.
  The advantage of this strategy is that all database operations are against one record of one table and thus gain in performance.

- MapStrategy::Strategy_sub_table. This is the strategy where every class (super class and subclass alike) have each there own table. This strategy is also kwown as the “joined-table-strategy”.
  The advantage of this strategy is that it does not waste any database space, and that mandatory attributes can have a NOT-NULL constraint in the database.
  The drawback on the other hand is that SELECT statements require “LEFT OUTER JOIN” links to the tables of the subclasses, and that the other statements (insert, update, delete) have to be repeated against multiple database tables.

  Coming in a later version:
- *MapStrategy::Strategy_classtable. This is the strategy where every class have there own table. Even for super- and subclasses. This strategy is also kwown as the “union-table-strategy”, because it takes a “SELECT union SELECT” construct, to get all the relevant records from the database when querying for an object.*
  The advantage of this strategy is that it wastes no space, and that every attribute in every class can have the full swing of the database help like NOT-NULL constraints and such.
  The drawback on the other hand is that it takes quite complex multiple SELECT statements chained together with a UNION construct. These are inherently slower than the select statements in the other strategies.

The mapping strategy can be set by your application by calling “hibernate.SetStrategy()”, but **ONLY** before all classes and configurations are loaded, either by loading the configuration.cxh file, or by loading the table definitions. As soon as there are sessions and classes defined, the strategy is fixed and cannot be changed again. This also means that the strategy is currently the same for all classes in the application!

Planned for a later version is the configuration where each class hierarchy can have it’s own mapping strategy. For now the strategy is fixed for the complete application

7. ASSOCIATION MAPPINGS

Data classes in “Entity Relation Diagrams” (ERD’s) are held together with associations. In the theory of datadesign there are in fact more possible association types than exist in relational databases today. This is one of the strengths of ORM (Object Relation Mappers) like Hibernate: translating those association mappings from theory to practical database implementations.

CX-Hibernate differs from other Hibernate implementations in the way we work with associations between classes. Instead of trying to lazely load as much data ‘around’ an object, we just ‘load’ the object itself, without the associated objects around it. This choice comes from the daily observation that much goes wrong with the loading of just to much data. Which in turn takes to much performance away from applications in unintended ways.

In this chapter the associations that are supported are described, after that the methods to follow associations from one object, leading into others are explained.

7.1 Association types

The association types from the relelation data theory are described in this paragraph. Not all types have an implementation in CX-Hibernate that differs from all others. They are described here from theory. *In future versions they may receive there own separate implementation however.*

7.1.1 The one-to-many assocation

One-to-many is an association as a master sees it’s details. Like the ‘invoice’ sees it’s ‘invoice-lines’. Surely the primary key (the one) sees the candidate key’s in the detailed class, which are in their turn are implemented by the database as a foreign key constraint on an not-null column and an index.

7.1.2 The many-to-one association

The ‘many-to-one’ is an association as a detail object sees it’s master. Like the ‘invoice-line’ sees the ‘invoice’. Practically implemented as a foreign key constraint on a not-null column to a primary key in the master record.

7.1.3 The zero-or-one-to-many association

The 'zero-or-one-to-many' association is in fact technically a 'one-to-many' association with a column that allows NULL's to be inserted into the database. In an application there is no distinction between an attribute with a zero (0) or a NULL (empty) value. And that's why it's hard to implement. Unless we make the agreement that an integer "id" with the value of zero (0) counts as a NULL indication in the database.
CX-Hibernate follows that agreement in two way's:

- You can have an SQLVariant as the association attribute. And SQLVariants can register NULL status, so we can stop the search for associated objects before we go to the database;
- Integral and ordinal types with the zero (0) value count also as a zero association, so no database search is made for associated objects.

7.1.4 The many-to-one-or-zero association

The 'many-to-one-or-zero' assication followed from the master side can only find those detailed records/objects that have a filled in association. Essentially no zero associations can be found.

7.1.5 The many-to-many association

The 'many-to-many' association cannot be implemented in a database in one-go. Under water an extra table with two associations one-to-many is made to implement such an association.
In a future version of CX-Hibernate the automatic implementation of this association will be made. For now we need to handcraft the extra association table with two foreign keys ourselves.

7.1.6 The one-to-one association

Theoretically this association should be avoided. If a 'one-to-one' association pops-up, the first question we must ask ourselves is: "why are these two objects, and not just one?" and "Why are we not refactoring our application model?". That said, we implement this association in a database with a many-to-one association and a unique index on the many-side of the association.

7.1.7 The one-to-oneself association

The 'one-to-oneself' association needs always some extra care because we must be sure which side of the association we are looking to. Technically it's implemented with a foreign key to the primary key of the same table. Withing the logical datamodel we need two names to be able to follow the association within the same class.

7.2 Following an association

All associated data in CX-Hibernate ***MUST*** be loaded explicitly by your application. You the programmer must take the discission to go load the associated data.
The data is collected in a "CXResultSet" vector, and you can then e.g. store these objects in a data member of your class if you like. Regardless whether you expect one or more-than-one result, the results are always returned to you in a set.
And again, you have three overloads. One for a single integer value, one for a SQLVariant and thus a value of any datatype, and one for a set of SQLVariants, so that you can easily follow a compound association key. The are the prototypes of CXSession to follow an association look like:

```
CXResultSet   FollowAssociation(CXObject*  p_object
                               ,CString    p_toClass
                               ,int        p_value
                               ,CString    p_associationName = "");
```

Because it is quite possible and most certainly likely that there are multiple assocaitions between classes, the last (optional !) parameter is the name of the association to follow. And this is how we distinguish between associations, and how we find the right side of the association in the case of a 'one-to-onself' association.
If you do not specify the exact association's name, Hibernate will search for an unique combination of the two classes. If it finds exactly one association, that one will be used. If not an exception will be thrown, and you must re-program your code to use the exact association.

Here is an example:

```
...
Master* master = (Master*)m_session->Load(Master::ClassName(),2);
CalculateWithMaster(master);

// Getting the details of the master
CXResultSet set;
set = m_session->FollowAssociation(master,Detail::ClassName(),master->GetID());
for(auto& object : set)
{
  Detail* detail = (Detail*)object;
  CString message;
  message.Format("Detail: %d Master: %d\n",detail->GetID(),master->GetID());
  CalculateDetail(detail,message);
}

// Go do something interesting with the master and all the details...
CalculateTotals(master,set);
...
```

7.3 Update and delete actions and associations

What happens when a record gets deleted (the master) with its associated detail records? In general the following implementations exist:

- RESTRICT: if detailed data (one-to-many) exist, the object cannot be deleted and the primary key cannot be updated. The database will fail, and thus the “Update” action in Hibernate will fail;
- CASCADE: if detailed data (one-to-many) exist, the detailed objects will get deleted with the master object. Any updates of the primary key will result in an update of the foreign key fields of the detail records;
- NO_ACTION: The name says it all: when one side of the association is changed, the other side is left alone, and no action is taken to change anything there. Of course, this can create inconsistencies in the object model, but that was clearly intentional.

- SET_NULL: If a master gets updated or deleted, the foreign key side of the association gets reset to NULL (empty). This prevents the inconsistencies that arise from the “no-action” optione. No pointers are left dangling in your datastore.
- SET_DEFAULT: If a master gets updated or deleted, the foreign key side of the association gets set to the default value as defined for the attribute. This also prevents the inconsistencies that arise from the “no-action” optione. No pointers are left dangling in your datastore.

7.4 Deferabillity of associations

Future paragraph about the deferability of associations. Deferabillity is needed for databases that by default check all constraints directly and not at ‘commit time’. Currently only needed for Informix databases.

```
SQL_INITIALLY_DEFERRED
SQL_INITIALLY_IMMEDIATE
SQL_NOT_DEFERRABLE
```

7.5 Partly matched associations

Future paragraph about the partly matched associations. Not widly in use on all types of databases.

```
SQL_MATCH_FULL(0)
SQL_MATCH_PARTIAL(1)
SQL_MATCH_SIMPLE(2)
```

8. FILTERS

Filters play an essential role in CX-Hibernate. Not only are they used to find and load objects, they are also used to follow associations. You can think of a filter as a conditional clause in a “WHERE” part of the SQL statement. Filters play an important role in completing SQL statements, without having to write SQL yourself.

SQLFilters have in essence three parts:

1) A column (attribute)
2) An operator
3) Zero, one or more values in the form of a SQLVariant

Examples of filters can be:

- “WHERE amount > 10000”
- “WHERE id = 312”
- “WHERE status IS NOT NULL”
- “WHERE age BETWEEN ’01/01/2012’ AND ’12/31/2012’ “
- “WHERE type IN (‘A’, ‘B’, ‘C’, ‘X’,’Z’)”

8.1 Operators

The following operators are defined for the SQLFilter object:

Operator	Function	Remarks on values
OP_Equal	Attribute equals the value	One value only can be set
OP_NotEqual	Attribute is not equal to the value	One value only can be set
OP_Greater	Attribute is greater than the value	One value only can be set
OP_GreaterEqual	Attribute is greater or equal to the value	One value only can be set
OP_Smaller	Attribute is smaller than the value	One value only can be set
OP_SmallerEqual	Attribute is smaller than or equal to the value	One value only can be set
OP_IsNULL	Attribute is NULL (empty)	No values can be set!
OP_IsNotNULL	Attribute is NOT NULL (filled)	No values can be set!
OP_LikeBegin	Attribute matches value with LIKE with only a ‘%’ at the end (index optimized!)	One value only can be set
OP_Like_Middle	Attribute matches value with LIKE with a ‘%’ at the beginning and at the end. Cannot be searched index optimized!	One value only can be set
OP_Between	Attribute’s value is between the two values (inclusive!)	One (1) extra value can be set with “AddValue”
OP_IN	Attribute equals one of the values	More than 1 value can be set with “AddValue”
OP_Exists	Do NOT specify an attribute	Use “AddExpression()” instead
OP_OR	Filter acts as an ‘OR’ condition connection instead of ‘AND’. See paragraph about the difference between ‘AND’ an ‘OR’	

The operator can be negated in the filter by calling the “Negate()” method. This comes in handy for the ‘BETWEEN’ and ‘IN’ operators, that have no logical negated counterpart.

8.2 More than one value

Generally you will need one value on a filter. In case of the ‘IN’ and ‘BETWEEN’ operators, you need more than one value. This can easily be done by adding more values to the filter with the method:

```
void AddValue(SQLVariant* p_value);
```

You can call this method once (BETWEEN) or multiple times (IN). The SQLFilter object will keep a stack of SQLVariant values to build the required SQL condition. In case of doubt you can retrieve the values again with

```
SQLVariant* GetValue(int p_number);
```

8.3 AND and OR

When filters are strung together in SQLFilterSet's, you must interpret the sets in first instance as a set of conditions strung together by the 'AND' keyword.

```
...
SQLFilter filt1("amount",OP_Greater,10000);
SQLFilter filt2("status",OP_IsNotNull);
SQLFilterSet set;
set.AddFilter(filt1);
set.AddFilter(filt2);
...
```

Put together in a SQLFilterSet this will result in a SQL condition like

`SQL => "WHERE amount > 10000 AND status IS NOT NULL"`

This will work well if we just want to add extra filters. All filters will then be strung together with 'AND's and it will constraint the search results further (filters the available objects).

Specifying an "OR" relationship between two condition filters can be done by creating a filter with just an "OP_OR" operator in the set. Here is an example:

```
...
SQLFilter filt1("amount",OP_Greater,30000);
SQLFilter filt2(OP_OR);
SQLFilter filt3("status",OP_Equal,12);
SQLFilterSet set;
set.AddFilter(filt1);
set.AddFilter(filt2);
set.AddFilter(filt3);
...
```

This wil result in exactly what we expect:

`SQL => "WHERE amount > 30000 OR status = 12"`

This works for simple cases, but what happens when we create complex conditions with sets of sub-conditions with and's and or's? This can not be specified by simply adding more filters. Consider the next example condition:

SQL -> "WHERE (type = 'A' and amount < 450) OR (type = 'B' and amount < 978)"

This is somewhat harder to put together. Luckily the "OpenParenthesis()" and "CloseParenthesis()" methods of the SQLFilter comes here to the rescue:

```
...
SQLFilter filt1("type",  OP_Equal, "A");
SQLFilter filt2("amount",OP_Smaller,450);
SQLFilter filt3(OP_OR);
SQLFilter filt4("type",  OP_Equal, "B");
SQLFilter filt5("amount",OP_Smaller,978);
filt1.OpenParenthesis();
filt2.CloseParenthesis();
filt3.OpenParenthesis();
filt4.CloseParenthesis();
SQLFilterSet set;
set.AddFilter(filt1);
set.AddFilter(filt2);
set.AddFilter(filt3);
set.AddFilter(filt4);
set.AddFilter(filt5);
...
```

8.4 Free expression filters

In some cases a single value is not enough. Maybe you will want to compare an attribute to a sub-select or a different construction than here supported. In those cases you can add a free expression filter. The free expression however does *NOT* replace the filter, but it replaces the value!
In this way it is possible to write a filter with a subselect.
You can add the expression to the filter with:

```
void AddExpression(CString p_expression);
```

The 'IN (sub-select)' construction is an example where the adding of the free expression can do things that you cannot with just a bunch of values, as in:

```
...
SQLFilter filt("status",OP_IN);
filt.AddExpression("(SELECT status FROM general_ledger WHERE sales = 'done')");
...
```

```
SQL => WHERE status IN
              ( SELECT status
                  FROM general_ledger
                 WHERE sales = 'done')
```

We can even do this with a filter with no attribute. The "EXISTS" clause is an example where this has a meaningfull implementation:

```
...
SQLFilter filt(OP_Exists);
filt.AddExpression("(SELECT 1 FROM general_ledger WHERE object = var.id)");
...
```

This will result in the following SQL condition:

```
SQL => "WHERE EXISTS (SELECT 1 FROM general_ledger WHERE object = var.id");"
```

9. TRANSACTIONS AND BATCHES

When performing single operations against a CXSession, that single operation will always be wrapped into a transaction of the database. The reason therefore is twofold:

1) Various databases only work from within transactions, even for a single SELECT, and;
2) A single operation (e.g. an Update) could need multiple database tables to be changed, depending on the complexity of the data model and the number of subclasses and the chosen hibernate mapping strategy.

It is the latter reason which force us to always use a transaction, even though we do not 'see' this transaction as programmers: we simple call the 'Update' method of the session.

When modifying a larger amount of objects, we want some way of ensuring the correctness and stability of the database, in case we encounter a problem halfway through the change.

9.1 Starting and committing transactions

We can start a new transaction by calling the sessions "StartTransaction()". This method will return a transaction number. The transaction number is also visible in the hibernation logfiles for the basic operations.
Committing the transaction – in case we do not encounter any problems – is just as simple. Just call "CommitTransaction" on the session and we're done.

To create more resilient programs, hibernate provides us with a "CXTransaction" class with auto-pointer capabilities. In case we make a programming mistake, or encounter an exception somewhere halfway the transaction, this autopointer class ensures the rollback of the transaction. In general we shall only call the "Commit()" method of this class. Here is an example:

```
...
try
{
  // Start our transaction from here
  CXTransaction trans(session);

  // Start our transaction from here
  session.Update(subject);
  session.Insert(new-things);
  session.Update(journal1);
  ... // other things to do
  session.Update(journal2);
  session.Delete(journal3);

  // Ready with all the basic operations
  trans.Commit();
}
catch(StdException& ex)
{
  ...
}
```

So the key point is: We program the start of the transaction and the success line, where we commit the total transaction. Any form of error will throw us to the catch handler, and that also removes the CXTransaction from the stack, which in it's turn will do a "Rollback" in the destructor. Performing an automatic rollback for us, in case we fuck things up!

9.2 Mutation stacks and mutation numbers

Internally the transaction number is used in the SQLDataSet as a mutation number on the mutation stack of a database field. This makes it possible to keep track of 'who-done-what'. Changes to the fields of records in the SQLDataSet are stacked by mutation number. If the last mutation is from the same source (mutation number), the mutation gets overwritten. If the last chage is for a new number, it always creates an extra mutation on the field.

When the system starts to change a record in the database (update, delete) it checks whether all changes to the record came from the same source (mutation number). It's a non blocking, non intrusive way to make multi-session applications possible. Only when two sessions are getting in each

others way, the update or delete must fail. Otherwise we can mix the mutations of multiple sessions on the caches of the class objects.

For applications: you can request the current mutation number from the CXTransaction object with the “Mutation()” method.

9.3 Subtransactions

Calling “StartTransaction()” a second time e.g. from a deeper level of methods or subroutines does NOT start a new transaction, but increments a “subtransaction” counter within the session. Committing likewise first decrements this subtransaction counter, and only the last outer call to ‘commit’ will perform the ‘real’ database commit.

9.4 Rolling back a transaction

Oops: we encountered an exception and rolled back our transaction. What now? Clearly the internal state of our objects are no longer in line with the contents of the external store (the database). Even so: we could have rolled back halfway along the line, the first object operations could have gone well. Through the database rollback, the database now has it original state back, but the objects in our caches **DO NOT!**

The next action to take leads to considerable discussion. Depending on your own views we could:

- Start over with our application (ouch!);
- Close the database and our hibernation session, and start a new one, and retrieve our objects again. Very safe, but cumbersome as users lose there changes;
- Remove only the affected objects from the caches with “`RemoveObject(object)`”. Nasty to program as we should have an inverse operation for every transaction that we program.

Whichever road you take, it’s up to you. CX-Hibernate does not inflict any of these choices on you.

9.5 Commiting on other data stores

Here will come a section in the manual how to ‘Start’, ‘Commit’ and ‘Rollback’ for other data stores like the internet store. For now – this version – it’s not yet possible to start an transaction on the internet store.

10. INTERCEPTION EVENTS

In application development – and especially in large applications – it's always handy to have a mechanism to intercept the base operations. Even if it's in the very last stages of a transaction. Multiple programmers may work in a large scale project with different knowledge of the lay of the land, or you just want to have a way to do a quick and dirty solution for the duration of a quick patch.

Precisly because of these reasons CX-Hibernate has four classes of interception events on the base operations of an object. These events are declared in "CXObject.h" and the are:

- **`void OnLoad()`**: Just after CX-Hibernate loads an object or a set of objects, the virtual overridable "OnLoad" of that object is called. This event cannot stop the application in loading this object, but it can perform extra actions on the just loaded object. The return type is 'void', because it cannot stop the load;
- **`bool OnInsert()`**: Before inserting an object in the datastore, this trigger gets called. Be aware that the trigger gets called before the generator value can be assigned to the designated attribute / property of the object. The primary key / business key of this object is bound to be incomplete upon calling this trigger. But as with the other triggers, we can perform extra actions or even stop the insert by returning 'false' from the trigger;
- **`bool OnUpdate()`**: Before updating an object in the datastore, this trigger gets called. Here you can add extra actions to perform before every update, or you can even stop the update by returning 'false' from this trigger. Be sure to return 'true' if you want the update to continue;
- **`bool OnDelete()`**: Before deleting an object from the datastore, this trigger gets called. Here you can add extra actions to perform before every deletion, or you can even stop the deletion by return 'false' from this trigger. Be sure to return 'true' if you want the deletion to continue.

Here are a few more facts to keep in mind about all of the above triggers:

- Triggers on single objects are carried out in relation to the action, but triggers on sets of objects are carried out in one bunch. So if all objects are loaded from the datastore, then the triggers for all objects are fired in a tight loop, all at once;
- Triggers are called for all datastores. It does not matter that the object goes to and from a database, a filestore or a webservice on the internet;
- Triggers are generally called before the logging of the object, so that if the trigger changes the object, the changes are reflected in the logfile. Exception to this rule is the OnDelete trigger, where we log the object to delete before we do anything else;
- Triggers from the framework to the application are called from within a try … catch() loop. The loop will intercept the "StdException" exception only and log the fact that the trigger misbehaved in the logfile at the 'error-level';
- Overloaded events are NOT generated by the "CXH2CPP" utility. In general it's unlikely that you will need these interception events right at the start of an application;
- It is a good custom to keep the code of the events in the <classname>.cpp file and not in the accompanying "<classname>_cxh.cpp" file. Reasons for this rule / custom is that we can later on re-generate the *_cxh.cpp" file from scratch – e.g. with the CXH2CPP utility – and then replace it on the file system level, without touching the trigger code.

11. DATATYPES

All standard fundamental C++ datatypes can be used in the CXHibernate framwork. Apart from these a number of extensions are created that are needed to communicate with the ODBC interface of the database. All datatypes that can be streamed from a to a database are so encapsulated in our own datatypes.

The following datatypes can be used and are supported by CXHibernate:

Datatype	ODBC type	Explanation
int	SQL_LONG	Standard 32 bits integer in the range from -2,147,483,648 to 2,147,483,647
long	SQL_LONG	Standard 32 bits integer in the range from -2,147,483,648 to 2,147,483,647
bigint	SQL_BIGINT	64 bits integer in the range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
short	SQL_SHORT	16 bits integer from -32768 to 32767
char	SQL_TINYINT	8 bits integer from -128 to 127
bool	SQL_BIT	True or false (0 or 1)
uint	SQL_ULONG	32 bits unsigned integer in the range from 0 (zero) to 4,294,967,295
ulong	SQL_ULONG	32 bits unsigned integer in the range from 0 (zero) to 4,294,967,295
ubigint	SQL_UBIGINT	64 bits integer in the range from 0 (zero) to 18,446,744,073,709,551,615
ushort	SQL_USHORT	16 bits integer from 0 to 65535
uchar	SQL_UTINYINT	8 bits integer from 0 (zero) to 255
float	SQL_FLOAT	Approximate floating point number as defined by the IEEE. Upto 3.4E +/- 38 (7 digits)
double	SQL_DOUBLE	Approximate floating point number as defined by the IEEE. Upto 1.7E +/- 308 (15 digits)
CString	SQL_CHAR	MFC String type
void* + size	SQL_BINARY	Pointer to a binary buffer. Can be used with database types as BLOB and CLOB
bcd	SQL_NUMERIC	High level exact floating point number upto 40 decimal places + math functions + operators
SQLDate	SQL_DATE	High level Gregorian date + subfunctions + operators on dates, time(stamp) and intervals
SQLTime	SQL_TIME	High level time type + subfunctions + operators
SQLTimestamp	SQL_TIMESTAMP	High level timestamp type + subfunctions + operators on dates, time(stamps) and intervals
SQLInterval	SQL_INTERVAL	High level interval type for all 13 types
SQLGuid	SQL_GUID	Microsoft GUID compatible type
SQLVariant	All above !!	Variant class, encapsulating all other types in this list!!
var		Same as the SQLVariant class

The following should be taken into account:

- Some of these datatypes are elementary C++ types, but the typedef defines (ushort for unsigned short) should be used instead of the elementary types. The reason for this rule is that the typenames are used in the serialization code for objects;
- SQLVariant (var) is an encapsulation of all other datatypes, and is used in the communication with the ODBC driver. Data must be encapsulated in this datatype to be carried to- and from the database driver.

12. QUERY LANGUAGE, NATIVE SQL and ODBC-SQL

One of the components CX-Hibernate was built upon is “SQLComponents”. If the hibernate framework does not provide a certain functionality, or the framework is too great a constraint, you can always break away to using standard ODBC query language or even native SQL.

In general the way to access the database is by way of the “SQLQuery” object. You can think of this object as the executor of your SQL statement and the holder of the resulting data tuple(s). After you execute the SQL statement, you can scroll through the results and access all columns of the data set.

12.1 Quick example of a native SQL query

Suppose we want to count the amount in the “orderline” table for all records where the “order_id” has a certain key. We could simply write:

SQL => “SELECT amount FROM orderline WHERE order_id = 314”

Here is how we do it in code:

```
...
try
{
  int totalAmount = 0;
  CString sql(”SELECT amount FROM orderline WHERE order_id = 314”);

  // Get a query object for this database
  SQLQuery query(session->GetDatabase());

  // Send SQL query to the database
  query.DoSQLStatement(sql);
  // Loop through our results
  while(query.GetRecord())
  {
    int amount = query.GetColumn(1)->GetAsSLong());
    totalAmount += amount;
  }
}
catch(StdException& ex)
{
  ...
}
```

In essence we create an SQL query string and a SQLQuery object. Then we execute the query by calling the “DoSQLStatement” worker method. Next we enter into a ‘while-loop’ that keeps going as long as we can get a record from the database in this result set. Per record we find the first result column (remind: the SQL language is ‘1’ based and so is the SQLQuery record that’s fetched from the database. We read the first column as a signed long and add it to the ‘totalAmount’ variable.

12.2 Fun with SQLQuery (binding)

The “SQLQuery” has a ton of functionality hidden inside. To give you a feel for one of these, the example of 12.1 is now repeated with parameter binding and result binding. These two are the most used features of the SQLQuery object.

First we re-write the SQL statement and replace all the values with parameter bindings. In ODBC the parameter binding character is a “?”. We use this for all the supported database types, so we do not have to worry if the parameter binding is different for different types of databases.

Secondly we set the parameters beforehand in the SQLQuery object with the “SetParameter” method. More than one parameter can be bound, and there exist a “ResetParameters” method, so we can use the SQLQuery object more than once for different sets of parameters.

As a third optimization, we use the result set binding. Here we get the result from SQLQuery with the array overload. It works quite natural, as e.g. the “[4]” binding will get you the result of the 4th column of the result set.

So here is the re-written example:

```
...
try
{
  int totalAmount = 0;
  CString sql(”SELECT amount FROM orderline WHERE order_id = ?”);

  // Get a query object for this database
  SQLQuery query(session->GetDatabase());

  // Send SQL query to the database
  query.SetParameter(1,314);
  query.DoSQLStatement(sql);
  // Loop through our results
  while(query.GetRecord())
  {
    totalAmount += query[1];
  }
}
catch(StdException& ex)
{
  ...
}
```

Keep in mind that not only are the result columns ‘1-based’ but also the parameters!

12.3 Native transactions

We can now expand on our example and do mutations on the database in a native transaction. To do so the SQLComponents has a “SQLTransaction” object to guard the transaction, do the “Commit()” or the “Rollback()”. In fact, it its this class that CX-Hibernate is building it’s transactions on. Here is how we can update the amount in the order table:

```
...
try
{
  int totalAmount = GetTotalAmount();
  CString sql(”UPDATE orders SET amount = ? WHERE id = ?”);

  // Get a query object for this database
  SQLQuery       query(session->GetDatabase());
  SQLTransaction trans(session->GetDatabase(),”UpdateOrders”);

  // Send update query to the database
  query.SetParameter(1,totalAmount);
  query.SetParameter(2,314);
  query.DoSQLStatementNonQuery(sql);
  // Commit the result
  trans.Commit();
}
catch(StdException& ex)
{
  ...
}
```

What happens here? There is an extra parameter binding in the SQL statement: one for the amount and one for the primary key (“id”). Then there is an extra “SQLTransaction” object for this statement. Keep in mind that each transaction needs a transaction name for those databases that do main- and subtransactions.
Next we set two parameter values with “SetParameter”, and then call “DoSQLStatementNonQuery”. It works just the same as the “DoSQLStatement” with the difference that the return type is the number of records affected in the database.
The transaction object either does the “Commit()” after a successful update, or it will automatically rollback in case of an error, by way of its destructor method.

12.3 ODBC SQL Functions

ODBC not only gives you the opportunity to write the same parameter bindings, it also has the possibility to write the SQL functions in the same way. The standard SELECT / INSERT / UPDATE / DELETE statements are now today quite standard. But the functions that we use in the select statements can be quit different, depending on the database that you are using.
Luckily ODBC has a kind of escape sequence to write database dependent constructions such as functions. The ODBC driver will translate those escaped parts of your SQL to the native SQL!

Instead of:

```
SQL => "SELECT stringlength(name) FROM table …"
```

Now: not every database has a "stringlength" function, so in ODBC you can write:

```
SQL => "SELECT { fn CHAR_LENGTH(name) } FROM table …"
```

Here the "{ fn" and "}" delimiters serve as a reminder that what's in between is a function (fn).
There exist a greate number of function escapes. The appendix 3 lists them all

12.4 Database dependent SQL language

Although SQL is a standardized language (ISO:9045) and almost every database vendor claims to adhere to that standard, a great many variations in sub-commands exist. The way to deal with al these variable implementations was to have a class (SQLInfoDB) with a bunch of methods that return a specified string or procedure. By connecting to a database we do not get the main SQLInfoDB class, but a derived subclass per database vendor. Within the SQLComponents module there exist:

- SQLInfoAccess
- SQLInfoFirebird
- SQLinfoGenericODBC
- SQLInfoInformix
- SQLInfoMySQL
- SQLInfoOracle
- SQLInfoPostgreSQL
- SQLInfoSQLServer

There are over 110 methods in these classes. Most of them return a single string for a keyword or command and some of them return complete SQL statements to query the database catalog for a specific item. You can use these methods to query the database catalog, or put together your own database independent SQL statements.

To get one of these, request an "SQLInfoDB" object from your logged in SQLDatabase. The reason that you must be logged in, lies in the fact that only when you log in to an actual database, the system can decipher which type of database you are on.

13. XML MAPPINGS

This chapter contains all information about the configuration and mapping file. By default the name of this file is 'hibernate.cfg.xml'. But this is only by convention. You can name the file anything you want, as long as you specify it when requesting a new session.

This is the format of the table in this chapter

LEGEND	
nodename	Nodenames are XML elements that contain a complex set of other child XML nodes. <class> and <attributes> are examples of these.
super -> nodename	Nodename as a direct child of "super" as in <super> <nodename /> </super>
Element	Elements contain a definition name. e.g. <attribute> If no supernode is given, it's a child of the first complex node above it!!
Attribute	Attributes of an element. E.g. "name" of an <attribute>

In the next table, all XML nodes in the file are described by name and contents.

ELEMENT	DESCRIPTION
hibernate	Main rootnode of the configuration file
default_catalog	Default database catalog to connect to (name of the datasource)
default_schema	Default schema within the database catalog. If a table of a class does NOT specify otherwise, this schema is used.
strategy	Selector that defines the O/R mapping mode for the application. Values can be: standalone Each class is standalone: No inheritance allowed one_table All classes of a class-hierarchy are in one database table sub_table Each sub-class is in it's own subclass table classtable Each class + superclasses together in one table
logfile	Name of the logfile used by hibernate
loglevel	Level of logging between -1 (off) and 6 (highest loglevel)
session_role	Default session role. Roles are: database_role Data is stored in an ODBC database internet_role Data is stored through webservices on the internet filestore_role Data is stored in separate files on a filesystem
database_use	How to treat the database at 'startup' and 'shutdown' time use Use an existing database create Create database tables from the config file create_drop Create database and drop tables at 'shutdown' time
hibernate -> resources	List of resources to include. Read each resource as if it was in this file.
resource	Filename to read as part of this definition file. Convention is that each class is in it's own definition file.
hibernate -> class	Definition of a data class containing objects
name	Name of the dataclass
schema	Name of the table schema (if not the default_schema!)
table	Name of the table (if not the same as the class)
super	Name of the superclass of this table. Superclass must have been defined previous in this file!
discriminator	Value of the discriminator for this class. In fact this is the discriminator value. *In this version it's restricted to a string of max 5 characters.*
class -> subclasses	List of subclasses of this class
subclass	Subclass of this class. Definition must follow later on in this file!
class -> attributes	List of persistent data attributes in the class
attribute	Persistent data attribute of the enclosing class

name	Name of the attribute. Max length of the name is determined by the underlying database engine.
datatype	Datatype of the attribute. Can be one of: string String of characters. Implemented as “VARCHAR” int Signed ordinal integer (32 bits) bool Boolean logic (true or false) bcd Exact numeric / decimal of max 40 decimal places uint Unsigned ordinal integer (32 bits) short Signed ordinal integer (16 bits) ushort Unsigned ordinal integer (16 bits) long Signed ordinal integer (32 bits) ulong Unsigned ordinal integer (32 bits) tinyint Signed ordinal integer (8 bits) utinyint Unsigned ordinal integer (8 bits) bigint Signed ordinal integer (64 bits) ubigint Unsigned ordinal integer (64 bits) float Approximate floating point (7 decimal places) double Approximate floating point (15 decimal places) date Gregorian date (day-month-year) time Clocktime on a day (0:00:00 upto 23:59:59 hours) timestamp Timestamp (year-month-day hour:minute:second) guid Microsoft compatible GUID var SQLVariant (all of the above in one container) For database systems that understand the ISO-8059 intervals, there are the following data types: interval_day_to_second interval_year_to_month interval_hour_to_second interval_hour_to_minute interval_day_to_minute interval_day_to_hour interval_minute_to_second interval_year interval_month interval_day interval_hour interval_minute interval_second
maxlength	For certain datatypes the maximum length of the data. Currently only used for string length.
generator	“true” or “false”. This attribute contains the value of the generator. One attribute can be named the generator.
isprimary	“true” or “false”.
isforeign	“true” or “false”.
not-null	“true” or “false”.
dbs_column	Name of the column in the database, if not the attribute name
default	Default value of the database column, when not given by the application.
class -> identity	Identity definition of the class. In general you can think of the identity as the business key or primary key. Not restricted to the “id” column!
name	Name of the identity. By convention it’s named “pk_<classname>”
deferrable	When the check of the identity is performed: initially_deferred Check can be performed at “commit” time initially_immediate Immediately checked if filled in not_deferrable Immediate checked and cannot be deferred
initially_deferred	Not filled in or contains the text “deferred”.
identity -> attribute	This element can be added more than once to the identity node. All attributes together form the ‘identity’ of the class.

name	Name of one of the attributes from the attributes list of “class -> attributes”
class -> associations	List of all associations of this class
association	One of the associations of the class. Can occur more than once.
name	Name of the association as seen from this class in the direction of the association_class (see below)
Type	Type and form of the association many-to-one one-to-many many-to-many
association_class	Name of the associated dataclass
attribute	One of the attributes of the candidate key to the association class. Can be added more than once, but must conform to the identity of the associated class *(full / partial matching may be added in a future version)*
name	Name of one of the attributes from the attributes list of “class -> attributes”
class -> indices	List of indeces for this dataclass
Index	Index. Can occur more than once
Name	Name of the index. Must be unique within a schema
Unique	‘true’ or ‘false’. When true this index contains unique values The default is ‘false’
Ascending	‘true’ or ‘false’. When true this index is fysically sorted in ascending order. The default is ‘true’
Filter	Index is not for attributes, but for this filter expression (Not supported on all databases!)
attribute	One of the attributes in the index. Fysical limitations on how much attributes can occur in an index and how much data they can contain exist on all databases.
name	Name of the attribute. Must be present in the attributes list of the class
class -> generator	Generator for unique values of the identity of an object
name	Name of the generator. In version 1.0 **must** be “<classname>_seq”
start	Starting value (ordinal integer value > 0) of the generator
class -> access	Access rights on the dataclass
user	Database user that’s granted access to this class
name	Name of the user that’s granted access (grantee)
rights	List of rights (SELECT,INSERT etc)
grantable	‘true’ or ‘false’. If true, the user can grant this rights as a grantor. The default is ‘false’

14. TOOLS

Currently there are two tools in the toolset. These tools are:

14.1 CFG2CPP

Generates a wizard like beginning for your class implementation out of the hibernate configuration file (Config-2-Cplusplus). Creates a *.H, *.CPP and a *_CXH.CPP.
Usage of the tool is simple: place a copy of the tool in the directory where you keep your hibernation configuration file. If you use the default “hibernate.cfg.xml” file, you will only need to call:

cfg2cpp classname

from the command line, where “classname” is one of the named classes in your configuration file. The tool will then spit out “classname.h”, “classname.cpp”and “classname_cxh.cpp”.

When you use a different name for the confuration file, you will have to specify the name on the command line like:

```
cfg2cpp /config:application.cfg.xml classname
```

So the option with the configuration file comes first, and after that we specify the classname we want to extract from it.

14.2 CFG2DDL

Generates a DDL (Data Definition Language) SQL script for the creation of the database that stores your objects. Generation reads the hibernate configuration file and spits out a SQL file.
The command line syntax of this tool is:

```
cfg2ddl /config:application.cfg.xml /datasource:odbc-datasource /user:username
        /password:secret [outputfile.sql]
```

If the /config parameter is not given, the default ‘hibernate.cfg.xml’ is used.
If the ‘outputfile.sql’ is not a given argument, the generated SQL is shown on the standard output.
A database (odbc-datasource name), user and password must be supplied so that the tool can log in. This is only used to determine the type of the database. The database could be devoid of tables or any other objects.

APPENDIX 1: FUN WITH SQLVariant's

The datatype object "SQLVariant" plays an important role in the SQLComponents library. It is an encapsulation of all used data types in the ODBC standard. This is a fast and efficient way to let data flow from your application to the ODBC datasource and vice versa. The SQLVariant datatype is internally thus structured that a direct binding to the ODBC driver can be done for data columns, parameters and resulting data sets.

APPENDIX 2: BUILDING AN IIS SERVER APP

APPENDIX 3: ODBC ESCAPE FUNCTIONS

This appendix list all string, numeric, date and time, and system functions that can be supported with ODBC escapes. The "{ fn" and "}" delimiters are left out of the tables, as they must be used on all functions.

A3.1 ODBC String functions

Here are all the string and character functions.

Function	Explanation
ASCII(*string_exp*)	Returns the ASCII code value of the leftmost character of *string_exp* as an integer.
BIT_LENGTH(string_exp)	Returns the length in bits of the string expression.
CHAR(*code*)	Returns the character that has the ASCII code value specified by *code*. The value of *code* should be between 0 and 255; otherwise, the return value is data source – dependent.
CHAR_LENGTH(string_exp)	Returns the length in characters of the string expression, if the string expression is of a character data type; otherwise, returns the length in bytes of the string expression (the smallest integer not less than the number of bits divided by 8). (This function is the same as the CHARACTER_LENGTH function.)
CHARACTER_LENGTH (*string_exp*)	Returns the length in characters of the string expression, if the string expression is of a character data type; otherwise, returns the length in bytes of the string expression (the smallest integer not less than the number of bits divided by 8). (This function is the same as the CHAR_LENGTH function.)
DIFFERENCE(*string_exp1*, *string_exp2*)	Returns an integer value that indicates the difference between the values returned by the SOUNDEX function for *string_exp1* and *string_exp2*
INSERT(*string_exp1*, *start*, *length*, *string_exp2*)	Returns a character string where *length* characters have been deleted from *string_exp1* beginning at *start* and where *string_exp2* has been inserted into *string_exp,* beginning at *start.*
LCASE(*string_exp*)	Returns a string equal to that in *string_exp* with all uppercase characters converted to lowercase.
LEFT(*string_exp*, *count*)	Returns the leftmost *count* characters of *string_exp*.
LENGTH(*string_exp*)	Returns the number of characters in *string_exp,* excluding trailing blanks.
LOCATE(*string_exp1*, *string_exp2*[, *start*])	Returns the starting position of the first occurrence of *string_exp1* within *string_exp2*. The search for the first occurrence of *string_exp1* begins with the first character position in *string_exp2* unless the optional argument, *start*, is specified. If *start* is specified, the search begins with the character position indicated by the value of *start*. The first character position in *string_exp2* is indicated by the value 1. If *string_exp1* is not found within *string_exp2,* the value 0 is returned.
LTRIM(*string_exp*)	Returns the characters of *string_exp,* with leading blanks removed
OCTET_LENGTH *(string_exp)*	Returns the length in bytes of the string expression. The result is the smallest integer not less than the number of bits divided by 8.
POSITION(*character_exp* IN *character_exp*)	Returns the position of the first character expression in the second character expression. The result is an exact numeric with an implementation-defined precision and a scale of 0.
REPEAT(*string_exp*, *count*)	Returns a character string composed of *string_exp* repeated *count* times.
REPLACE(*string_exp1*, *string_exp2*, *string_exp3*)	Search *string_exp1* for occurrences of *string_exp2* and replace with *string_exp3*.
RIGHT(*string_exp*, *count*)	Returns the rightmost *count* characters of *string_exp*.
RTRIM(*string_exp*)	Returns the characters of *string_exp* with trailing blanks removed.
SOUNDEX(*string_exp*)	Returns a data source – dependent character string representing the sound of the words in *string_exp*. For example, SQL Server returns a 4-digit SOUNDEX code; Oracle returns a phonetic representation of each word.
SPACE(*count*)	Returns a character string consisting of *count* spaces.

SUBSTRING(*string_exp*, *start*, *length*)	Returns a character string that is derived from *string_exp* beginning at the character position specified by *start* for *length* characters.
UCASE(*string_exp*)	Returns a string equal to that in *string_exp* with all lowercase characters converted to uppercase.

A3.2 ODBC Numeric functions

The following functions are defined for integer, float, double, numeric and decimal datatypes.

Function	Explanation
ABS(*numeric_exp*)	Returns the absolute value of *numeric_exp*.
ACOS(*float_exp*)	Returns the arccosine of *float_exp* as an angle, expressed in radians.
ASIN(*float_exp*)	Returns the arcsine of *float_exp* as an angle, expressed in radians.
ATAN(*float_exp*)	Returns the arctangent of *float_exp* as an angle, expressed in radians.
ATAN2(*float_exp1*, *float_exp2*)	Returns the arctangent of the x and y coordinates, specified by *float_exp1* and *float_exp2*, respectively, as an angle, expressed in radians
CEILING(*numeric_exp*)	Returns the smallest integer greater than or equal to *numeric_exp*. The return value is of the same data type as the input parameter.
COS(*float_exp*)	Returns the cosine of *float_exp,* where *float_exp* is an angle expressed in radians.
COT(*float_exp*)	Returns the cotangent of *float_exp,* where *float_exp* is an angle expressed in radians.
DEGREES(*numeric_exp*)	Returns the number of degrees converted from *numeric_exp* radians
EXP(*float_exp*)	Returns the exponential value of *float_exp*.
FLOOR(*numeric_exp*)	Returns the largest integer less than or equal to *numeric_exp*. The return value is of the same data type as the input parameter.
LOG(*float_exp*)	Returns the natural logarithm of *float_exp*.
LOG10(*float_exp*)	Returns the base 10 logarithm of *float_exp*.
MOD(*integer_exp1*, *integer_exp2*)	Returns the remainder (modulus) of *integer_exp1* divided by *integer_exp2*.
PI( )	Returns the constant value of pi as a floating point value.
POWER(*numeric_exp*, *integer_exp*)	Returns the value of *numeric_exp* to the power of *integer_exp*.
RADIANS(*numeric_exp*)	Returns the number of radians converted from *numeric_exp* degrees.
RAND([*integer_exp*])	Returns a random floating point value using *integer_exp* as the optional seed value.
ROUND(*numeric_exp*, *integer_exp*)	Returns *numeric_exp* rounded to *integer_exp* places right of the decimal point. If *integer_exp* is negative, *numeric_exp* is rounded to |*integer_exp*| places to the left of the decimal point.
SIGN(*numeric_exp*)	Returns an indicator of the sign of *numeric_exp*. If *numeric_exp* is less than zero, –1 is returned. If *numeric_exp* equals zero, 0 is returned. If *numeric_exp* is greater than zero, 1 is returned.
SIN(*float_exp*)	Returns the sine of *float_exp,* where *float_exp* is an angle expressed in radians.
SQRT(*float_exp*)	Returns the square root of *float_exp*.
TAN(*float_exp*)	Returns the tangent of *float_exp,* where *float_exp* is an angle expressed in radians.
TRUNCATE(*numeric_exp*, *integer_exp*)	Returns *numeric_exp* truncated to *integer_exp* places right of the decimal point. If *integer_exp* is negative, *numeric_exp* is truncated to |*integer_exp*| places to the left of the decimal point.

A3.3 ODBC Time, date and interval functions

Some date / time functions have duplicates, depending on the ODBC version, such as “CURTIME()” and “CURRENT_TIME()”. Interval functions are only well supported for those database engines that support the INTERVAL data type.

Function	Explanation
CURRENT_DATE( )	Returns the current date.
CURRENT_TIME[(*time-precision*)]	Returns the current local time. The *time-precision* argument determines the seconds precision of the returned value.

CURRENT_TIMESTAMP [(*timestamp-precision*)]	Returns the current local date and local time as a timestamp value. The *timestamp-precision* argument determines the seconds precision of the returned timestamp.
CURDATE( )	Returns the current date.
CURTIME( )	Returns the current local time.
DAYNAME(*date_exp*)	Returns a character string containing the data source – specific name of the day (for example, Sunday through Saturday or Sun. through Sat. for a data source that uses English, or Sonntag through Samstag for a data source that uses German) for the day portion of *date_exp*.
DAYOFMONTH(*date_exp*)	Returns the day of the month based on the month field in *date_exp* as an integer value in the range of 1 – 31.
DAYOFWEEK(*date_exp*)	Returns the day of the week based on the week field in *date_exp* as an integer value in the range of 1 – 7, where 1 represents Sunday.
DAYOFYEAR(*date_exp*)	Returns the day of the year based on the year field in *date_exp* as an integer value in the range of 1 – 366.
EXTRACT(*extract-field FROM extract-source*)	Returns the *extract-field* portion of the *extract-source*. The *extract-source* argument is a datetime or interval expression. The *extract-field* argument can be one of the following keywords: YEAR MONTH DAY HOUR MINUTE SECOND The precision of the returned value is implementation-defined. The scale is 0 unless SECOND is specified, in which case the scale is not less than the fractional seconds precision of the *extract-source* field.
HOUR(*time_exp*)	Returns the hour based on the hour field in *time_exp* as an integer value in the range of 0 – 23.
MINUTE(*time_exp*)	Returns the minute based on the minute field in *time_exp* as an integer value in the range of 0 – 59.
MONTH(*date_exp*)	Returns the month based on the month field in *date_exp* as an integer value in the range of 1 – 12.
MONTHNAME(*date_exp*)	Returns a character string containing the data source – specific name of the month (for example, January through December or Jan. through Dec. for a data source that uses English, or Januar through Dezember for a data source that uses German) for the month portion of *date_exp*.
NOW( )	Returns current date and time as a timestamp value.
QUARTER(*date_exp*)	Returns the quarter in *date_exp* as an integer value in the range of 1 – 4, where 1 represents January 1 through March 31.
SECOND(*time_exp*)	Returns the second based on the second field in *time_exp* as an integer value in the range of 0 – 59.
TIMESTAMPADD(interval, integer_exp, timestamp_exp)	Returns the timestamp calculated by adding *integer_exp* intervals of type *interval* to *timestamp_exp*. Valid values of *interval* are the following keywords: SQL_TSI_FRAC_SECOND SQL_TSI_SECOND SQL_TSI_MINUTE SQL_TSI_HOUR SQL_TSI_DAY SQL_TSI_WEEK SQL_TSI_MONTH SQL_TSI_QUARTER SQL_TSI_YEAR where fractional seconds are expressed in billionths of a second. For example, the following SQL statement returns the name of each employee and his or her one-year anniversary date:

	SELECT NAME, {fn TIMESTAMPADD(SQL_TSI_YEAR, 1, HIRE_DATE)} FROM EMPLOYEES If *timestamp_exp* is a time value and *interval* specifies days, weeks, months, quarters, or years, the date portion of *timestamp_exp* is set to the current date before calculating the resulting timestamp. If *timestamp_exp* is a date value and *interval* specifies fractional seconds, seconds, minutes, or hours, the time portion of *timestamp_exp* is set to 0 before calculating the resulting timestamp.
TIMESTAMPDIFF(*interval*, *timestamp_exp1*, *timestamp_exp2*)	Returns the integer number of intervals of type *interval* by which *timestamp_exp2* is greater than *timestamp_exp1*. Valid values of *interval* are the following keywords: SQL_TSI_FRAC_SECOND SQL_TSI_SECOND SQL_TSI_MINUTE SQL_TSI_HOUR SQL_TSI_DAY SQL_TSI_WEEK SQL_TSI_MONTH SQL_TSI_QUARTER SQL_TSI_YEAR where fractional seconds are expressed in billionths of a second. For example, the following SQL statement returns the name of each employee and the number of years he or she has been employed: SELECT NAME, {fn TIMESTAMPDIFF(SQL_TSI_YEAR, {fn CURDATE()}, HIRE_DATE)} FROM EMPLOYEES If either timestamp expression is a time value and *interval* specifies days, weeks, months, quarters, or years, the date portion of that timestamp is set to the current date before calculating the difference between the timestamps. If either timestamp expression is a date value and *interval* specifies fractional seconds, seconds, minutes, or hours, the time portion of that timestamp is set to 0 before calculating the difference between the timestamps.
WEEK(*date_exp*)	Returns the week of the year based on the week field in *date_exp* as an integer value in the range of 1 – 53.
YEAR(*date_exp*)	Returns the year based on the year field in *date_exp* as an integer value. The range is data source – dependent.

A3.4 ODBC System functions

The following are system functions available since ODBC 1.0, so most database engines do support them.

Function	Explanation
DATABASE( )	Returns the name of the database corresponding to the connection handle. (The name of the database is also available by calling **SQLGetConnectOption** with the SQL_CURRENT_QUALIFIER connection option.)
IFNULL(*exp*, *value*)	If *exp* is null, *value* is returned. If *exp* is not null, *exp* is returned. The possible data type or types of *value* must be compatible with the data type of *exp*.
USER( )	Returns the user name in the DBMS. (The user name is also available by way of **SQLGetInfo** by specifying the information type: SQL_USER_NAME.) This may be different than the login name.