DataBlock

Reference Documentation

Revision 123

© 2004 – 2007 Marius Gheorghe. All rights reserved.

Table of Contents:

About

- About Data Block
- Copyright notice

Overview:

DataBlock overview.

Object oriented application design and development has proven to be a very efficient and productive way in which to develop today's complex business applications. It increases the opportunities for code reuse and gives you more flexibility in the world of ever – changing client requirements.

When creating database driven business applications, however, developers are forced to work with RDBMS objects like datasets, views, sql, stored procedures, tables instead of plain business objects, which, the application ultimately attempts to model. This gap between the RDBMS model and the OOP language presents a significant challenge to developers.

So is there a way in which to map business objects to a relational database without having to deal with tables, columns and views ? With DataBlock you can. DataBlock approaches this problem from a different angle – from the perspective of the business object world. With it, you generate the mapping for your business object from the database schema and the DataBlock persistence framework takes care of the rest on your behalf.

DataBlock is a powerful and high performance object/relational persistence framework for the Microsoft.NET environment. It relieves you from hammering those endless data access routines and let's you concentrate instead on writing business logic code. DataBlock maps the entities of your database and create full features business facades for them. You can easily extend a business façade with business operation specific to you entity.

DataBlock design goals were:

- simple, small and fast persistence framework.
- database independent and database provider independent.
- flexible. We allow you to take advantage of the targeted database server. We allow you to write sql queries and stored procedures.
- quickstart. Ideally you start DataBlock Modeler, generate your entities, include the files in your project and then start writing business code.
- save time by handling hard persistent problems (like handling hierarchical INSERT queries)
- interoperate with existing code. DataBlock allows you to read data as datasets thus preserving and allowing you to extend the already written ADO.NET code.

Related to the flexibility of the framework…the mantra for DataBlock was "work with data the way" you want. We acknowledge the fact that there is NOT a "perfect" way to write data access code which works in every circumstance. That's way DataBlock is very flexible: you can mix and match different ways to write data access code:

- at the very high level you have a OR/M which allows you to work with entities and

business objects. Combined with the QueryCriteria API you have a very powerful, expressive and object oriented way to write **database independent code.**

- you have access to the execution environment which allows you to extend you business objects using query strings or stored procedures and taking advantage of specified database features (like the XML support in SqlServer 2000/2005).

DataBlock comes with DataBlock Modeler, a GUI modeler tool which doesn't need a expensive IDE to run. You just need to run DataBlock Modeler, connect to the database server and generate the mapping files.

Benefits:

How DataBlock can improve and speed up your software development.

DataBlock's purpose is to allow you to work with data in a seamless object oriented way and to **boost your productivity** by providing features that allow you to easily achieve your goals. In a business application the data workflow is usually like this:

- read data from database.

- pass it to the user interface (be it an ASP.NET / Windows form application) and display it (using manual or automatic data binding).

- allow the user to modify the data.

- save the modified data back to the database.

Of course, on most of these levels, the specific business rules come to play. So, how can DataBlock can help you achieve these goals faster and better ? Here is a quick rundown of features :

1. minimum requirements

DataBlock requires only the database schema. Based on that schema it generates the mapping source code. That's all. It's a few minutes process. You use DataBlock Modeler to connect to the database and generate the files. No need to muck around with Visio (or any other NIAM/ORM designer) or to edit xml files manually.
After that you can start writing business code.

2. code organization

- a nice 3 tier architecture. By using DataBlock you are provided with the data access layer and the blueprint for the business layer. So you will **NEVER** have mixed user interface and business logic/data access code.

- inherent use of the domain model pattern. Each application entity will have a "single access" point. For instance if want to interact with the "Order" entity you can do that only by using the OrderPersistentObject class.

2. read data

- because at the user interface level you use data in different ways, DataBlock supports multiple formats of returning data. By default DataBlock supports lazy

loading (you load the data from the child tables only when you need it). If you want to load everything "up front" you can get the data only as a dataset. The supported formats are :

a) **Datasets**/**DataTables** – the dataset is used mostly to display data in tabular format using a DataGrid. But the datasets can be used to "interoperate" with already existing code that doesn't use DataBlock. And last datasets are used in interop scenario (for instance in a web service).

b) Mapping objects – this is the recommended way. It allows you to return directly an instance of the mapped object.

c) lists / collections. - If all you want to do with the data is to populate a combo box then the most natural (and fastest way) is to return the data as a List<T>. DataBlock is using a IDataReader to read the data from database.

DataBlock allows you to filter data in 2 ways:
- by using the QueryCriteria API.
- by using the object's primary key.

To "convert" from one format to the others you have the **DataConvertor** class.

3. update data
Cascading Create/Delete/Update operations running, where is necessary, in transactions.

4. maintain database consistency.
All the operations in which multiples are executed in a transaction. The programmer can specify the transaction granularity.

5. **chain multiple business operations together.**
DataBlock introduce the concept of a "long running transaction" using an object call "Session". A session allows you to "stack" together multiple business operation and run them in a transaction.

Another area of problems that usually appear during the software development process is the necessity of re-coding the parts of the application that are responsible for object-to-relational transformation. For instance, if during the development phase the database schema changes the application will change to reflect those changes. You will have to rewrite/modify the inline SQL/ stored procedures to reflect those changes. With DataBlock you just have to regenerate the mapping classes.

Features:

- **Database independent code**
  The generated code works without modifications on MS Access, Sql Server and MySql.

- **Supports any CLI 2.0 compliant language**
  DataBlock works only with .NET 2.0

- **Full support for relations**
  Supports 1:1, 1:m and m:m relations.

- **GUI code generation tool**
  Easily generate code using the DataBlock Modeler tool.

- **Relational model friendly**
  DataBlock is relation model friendly. It doesn't require any special constructs in your relational model nor extra tables or columns. No locks are hold/set at runtime or during a transaction, other then the locks set by the RDBMS itself. All mapping data is contained in memory at runtime.

- **Supports reading/creating/deleting/updating of single and graphs of objects.**
  CRUD operations for a single (or multiple relation based) objects are included in the PersistentObject derivates.

- **Entities based on custom classes**
  All entity classes are based on lightweight custom classes, not on datasets or datatables.

- **Patterns based generated code**

- **Support for OO query mode**
  The QueryCriteria allows you to model, in an OOP way, SQL queries with criterias.

- **Class Inheritance based code**
  If the generated object is meant to support the common operations you don't even need to generate the PersistentObject class.

- **Powerful data access layer which works in both connected and disconnected mode.**

- **Generic database provider**
  By default the DataBlock Framework is using "stronged referenced" providers but it can be easily changed (thru the configuration file) to use a new provider. On one of our internal project (which is using DataBlock framework) we easely switch back and forth between 3 MySql providers.

- **No external xml mapping files**
  All the generated code is compiled into an assembly.

- **Full support for stored procedures**
  DataBlock allows you to call stored procedures.

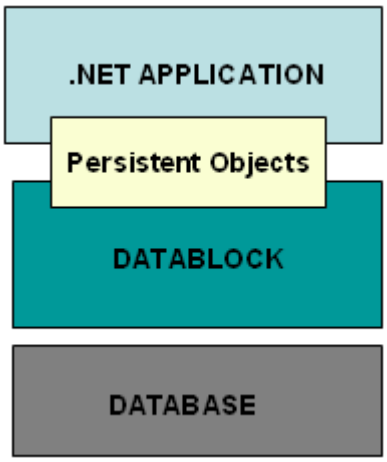
- **Full support for connection pooling**

- **Full support for ADO.NET transactions** (when .NET 2.0 will RTM we'll add suport for multiple database transactions without COM+).
  Also DataBlock introduces a new transaction mode called "long running application transactions". These application transactions allow you to "chain" multiple operations from multiple business facades into one big transaction.

- **Written with performance in mind**

DataBlock Arhitecture :

A **very high level overview** of the DataBlock architecture looks like that :

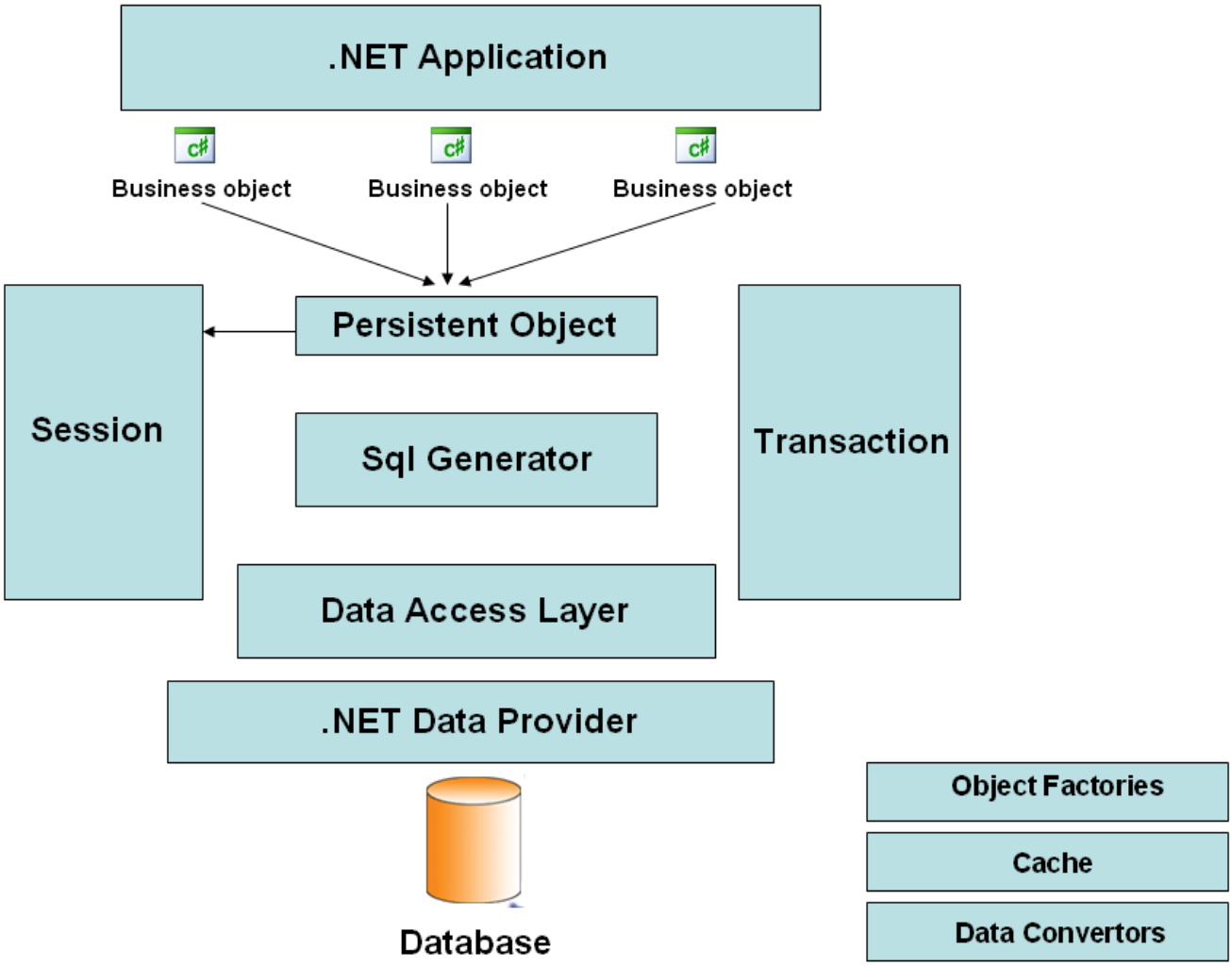


The diagram shows DataBlock using a database to provide persistence and data retriever services to a .NET application.

The .NET application (who can be a Windows application, ASP.NET, Web Service, Windows Service etc) is using DataBlock to retrieve and persist data from the database. DataBlock is accomplishing that by using the defined persistent objects (who are created based on the database schema).

The .NET application developer doesn't have to know anything about the databases, SQL or ADO.NET specific objects (such as IConnection, ISqlParameter etc). He will just use the DataBlock API to persist the object and DataBlock takes care of the rest.

The "**in depth**" DataBlock architecture looks like that:



Here are some definitions of the elements presented above:

- **Business object** : these are the objects with which the programmer is working. A business object is the representation of a database entity.

- **Persistent Objects** : these object encapsulate the business object interaction with the relational database.

- **Sql Generator :** this object creates a **S**tructured **Q**uery **L**anguage query from a Business object representation. This object contains implementations for all the supported database server.

- **Session :** the Session object has 2 purposes :

    1. executes all the operations on the same database connection. (this is used when you want to retrieve/persist multiple objects and you want this on a single database connection).

    2. allows you to "stack" together multiple business operations (which by themselves already run in a existing transaction) together and run them in a massive long running transaction

- **Transaction** : although not a "concrete" object you can use directly, a transaction represents a database transaction initiated in a PersistentObject or by a Session.

- **.NET Data Provider** : this is the ADO.NET data provider used to connect to the database. DataBlock allows you to switch between different providers for the same database server by simply editing a configuration file.

- **Cache:** this represents DataBlock's internal caching mechanism.

- **Data Converters :** this object allows you to convert between different ways of representing data : business objects, datasets, IDictionary etc.

Working with DataBlock.

Introduction to business objects

So let's introduce the abstract notion of business object and let's see how do you use it in your application. A business object is a object which contains services for a database entity. To create a business object for a database entity DataBlock has 2 type of classes :

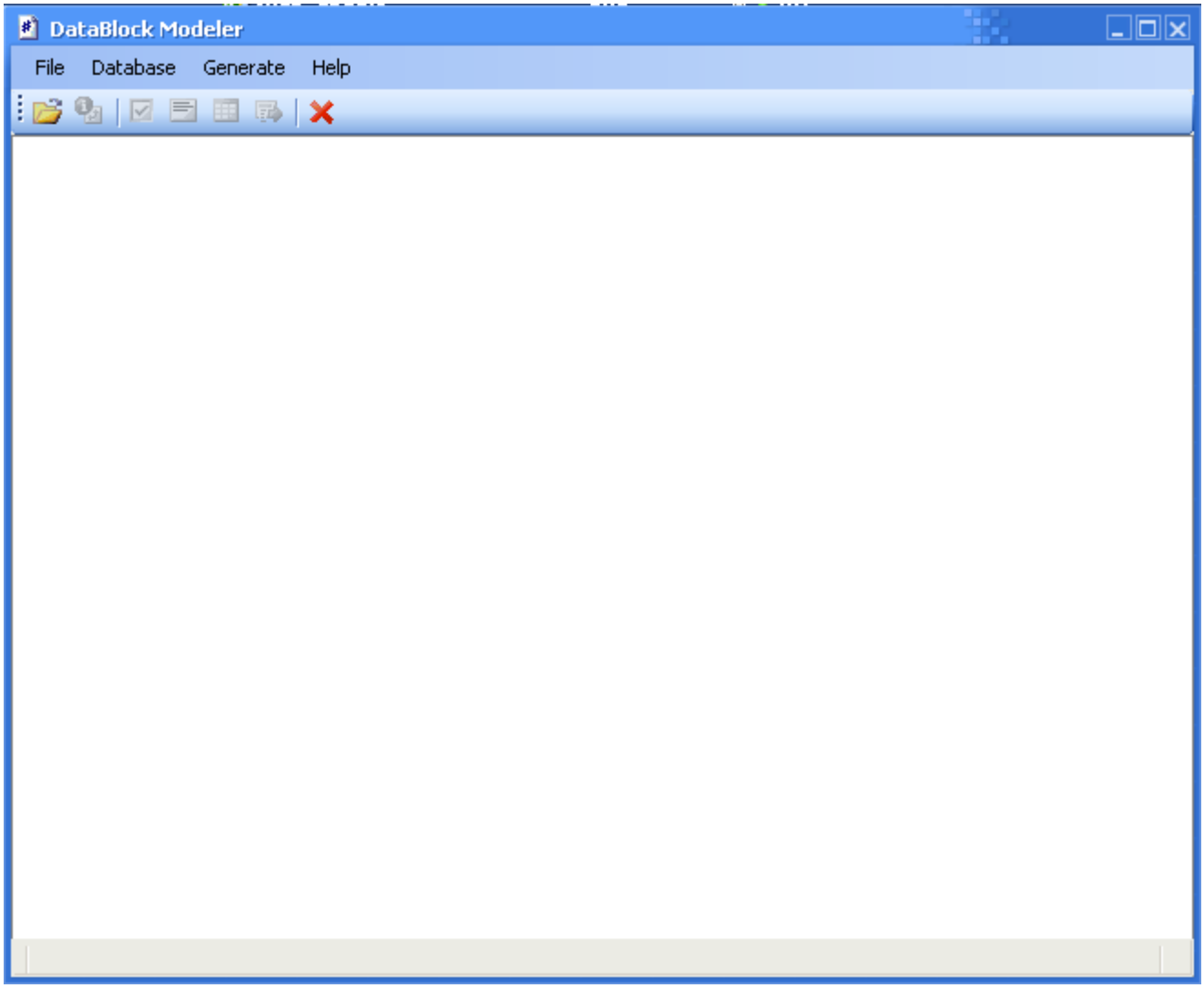
- **TableMetadata** : this is the base class used by the mapping files. It it used to describe the entities (it includes the table fields, table name, entity relations etc). The mapping class for your entity will derive from this class.

- **PersistentObject :** this is the base class for your business object. This class encapsulates all the access with the database. Your entity will be derived from this class.

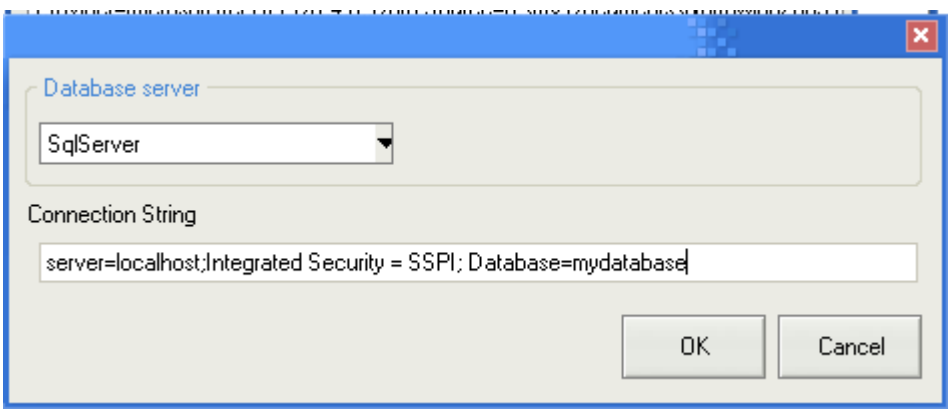
Introduction to DataBlock Modeler :

DataBlock Modeler is a tool which generates mapping files (in C# or VB.NET) from your database entities. This is a tool which encourages incremental generation of entitites. Here is how to use it :

- run the program



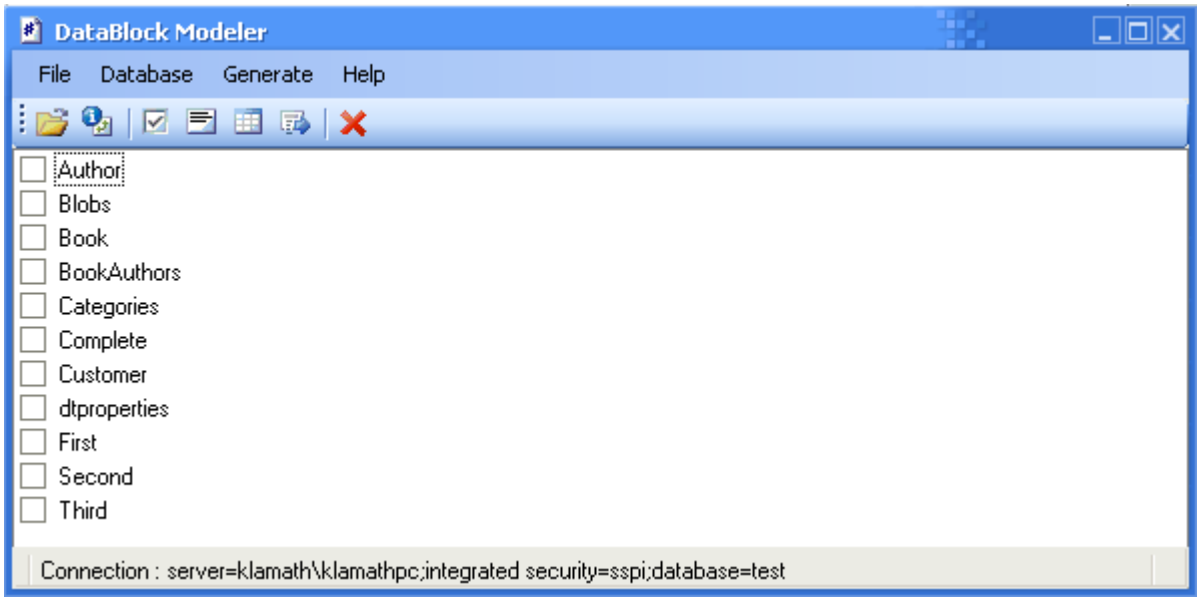
- click on the first icon (or Database/Connect) to connect to the database. If this is the first time we run the program we must add the database connection string. Click on "New" and type the database connection string for your database server (Access, Sql Server or MySql). In this demonstration we will use Sql Server 2000.



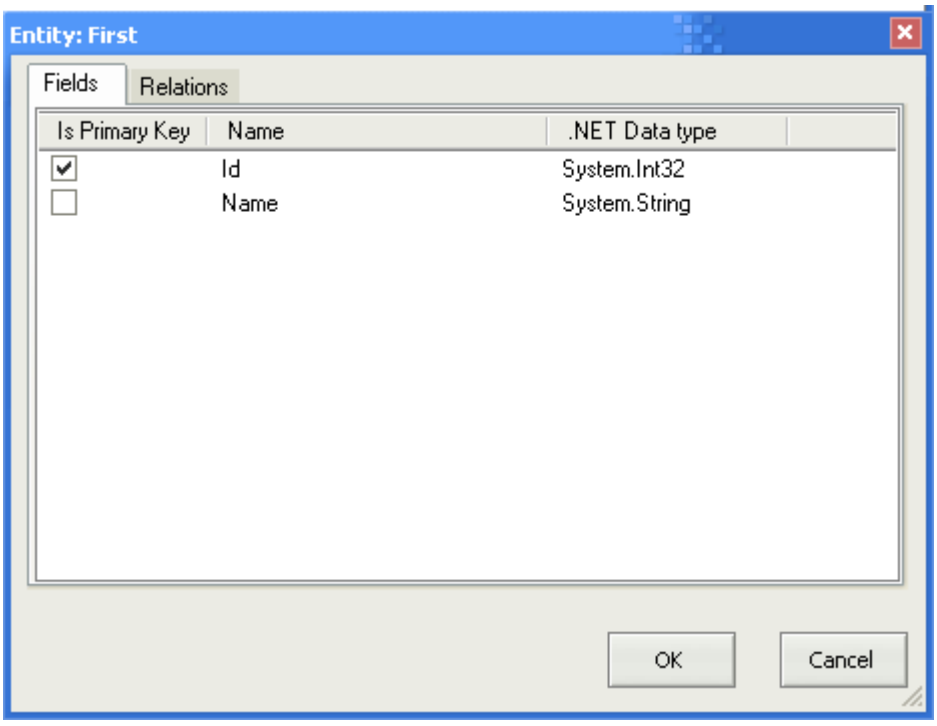
VERY IMPORTANT

To connect to MySql DataBlock is using the ODBC driver so make sure you have the lastest MySql ODBC driver installed before attempting to connect. The connection string looks like : **DRIVER={MySQL ODBC 3.51 Driver};SERVER=localhost;DATABASE=test;USER=sa;password=sa**

Click OK. The connection string is added to the list so double click it to load the database schema. After s short period the database schema is loaded:



Notice the checked list. The schema of the checked items will be used for the generation. But first we must set up the entity relations. Select a entity from the list and press the second button from toolbar (or right click and "Table Properties"). We get the properties of the selected entity :



On the first tab we can see the list of fields along with a few details:

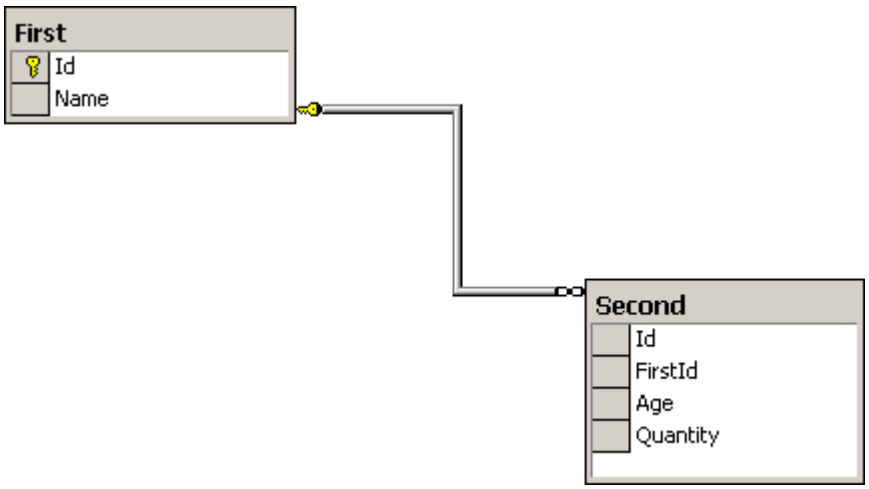
- **Is Primary key** : sets the selected item as being the table's primary key

- **Name:** name of the field.
- **.NET data type**: the data type of the mapped field. A list with the conversion between the database types and .NET types is in the section "Database supported types"**.**

Please note that a table MUST have a primary key.

On the second tab we can edit the table's relations. DataBlock allows you to model the 1:1, 1:m and m:m relations in the following way:

1. **Parent - > Child relation** : this is a type of relation in which our current entity is the Parent and to which we must add a child. Here is a sample of this type of relation and how we model it in DataBlock.



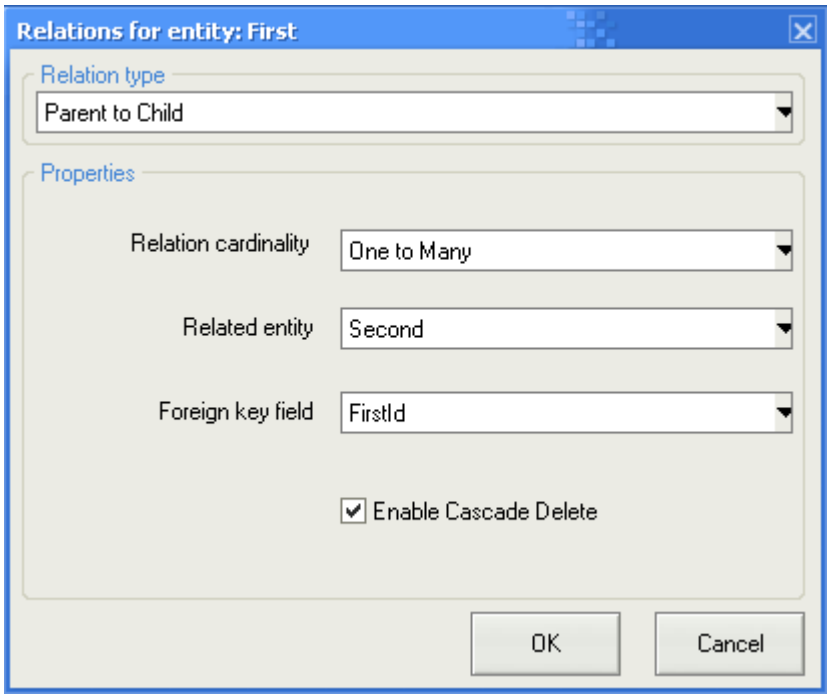
The attributes associated with a Parent -> Child relation are :

- **Cardinality** : we can choose between a One to One and One to Many. In our sample this is "One to Many".

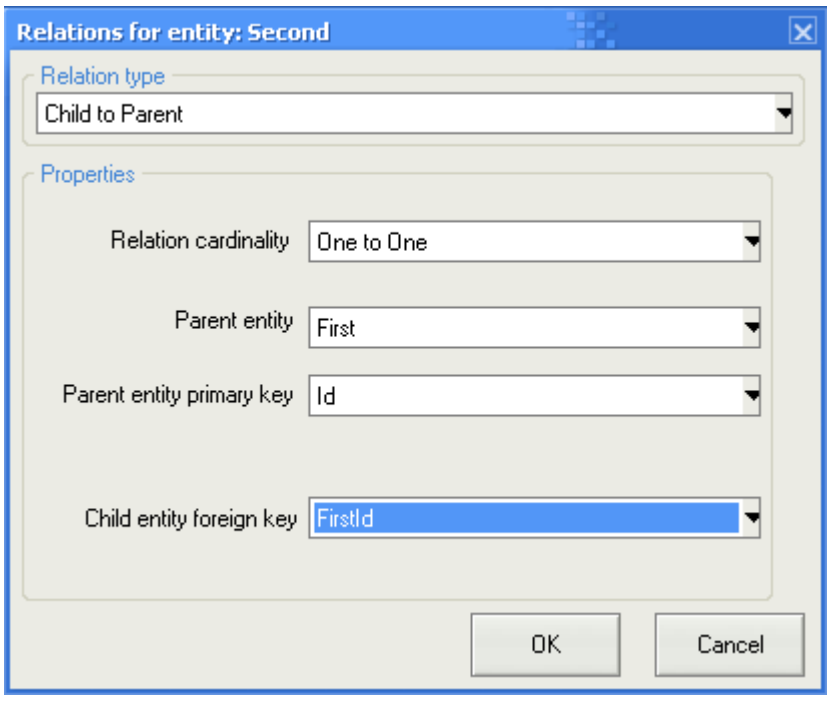
- **Related entity** : we choose the name of the related entity. In our sample this is "Second".

- **Foreign Key field** : we choose the name of the foreign key from the related entity. Note that the type of this field must be the same with the table of the primary key from our current entity. In our sample this is "FirstId".

- **Enable Cascade delete** : we can enable the "Cascade delete" feature. In this case, when the parent record will be deleted we will also delete the associated records form the child table.



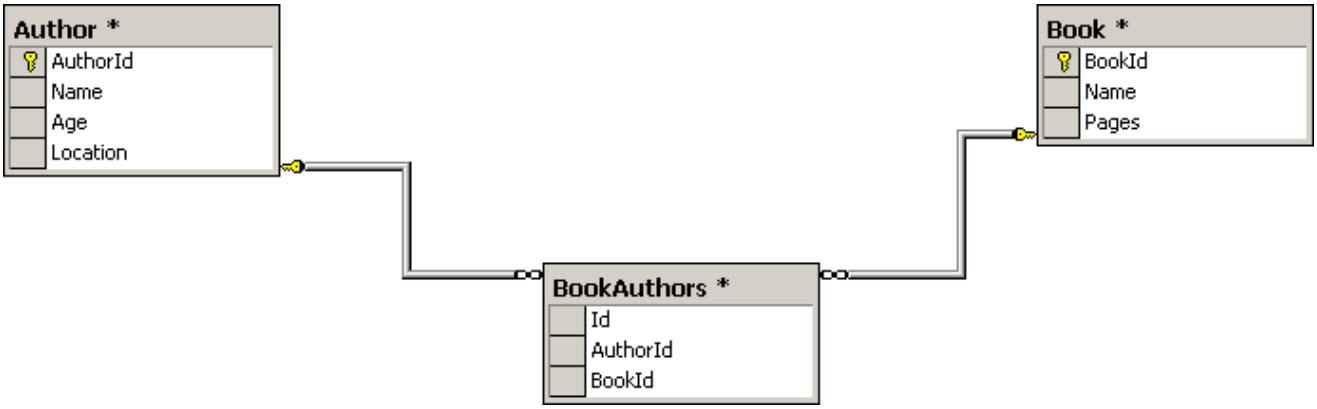
2. **Child -> Parent** : this is the type of relation in which our entity is the child and we must select the related parent entity. For a sample we take the above relation and we model now a Parent -> Child relation between the "Second" and "First" entities.



This type of relation has the following attributes :

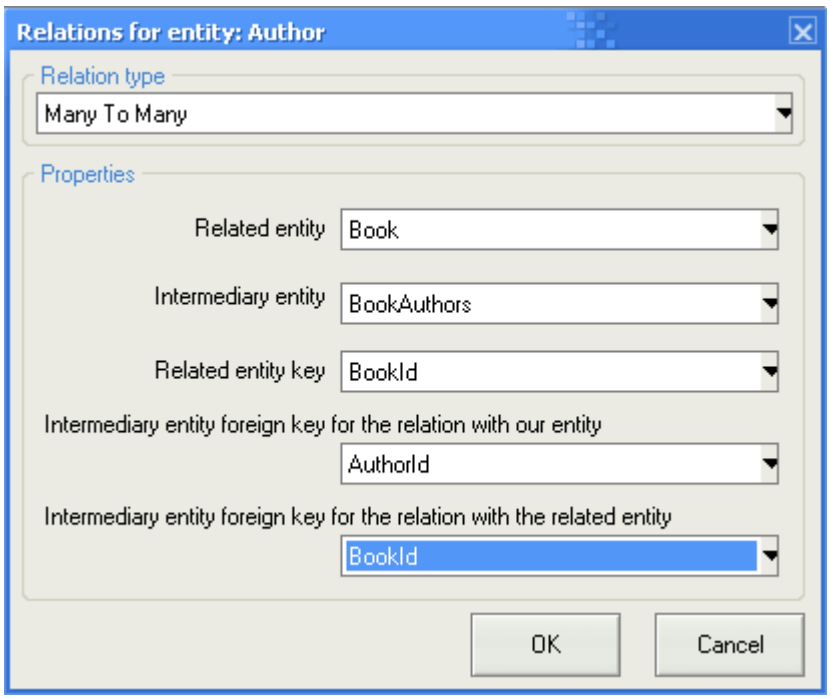
- **Cardinality** : which in this case can be only be One to One.
- **Parent related entity** :the name of the related entity.
- **Parent entity primary key** : choose the primary key of the parent entity.
- **Foreign key** : choose the foreign key from our entity

3. **Many -> Many** : represents a m:m relation. Here is a sample of a m:m relation and how we model this particular exemple in DataBlock.



In our example our current entity is "Author" and we want to add a m:m relation with the "Book" entity.

To model this type of relation we must set the following attributes :



- **Related entity** : this is the name of the related entity. In our sample this is "Book".

- **Intermediary entity** : this is the name of the entity thru which the current and related entity are relating. In our sample this is "BookAuthors".

- **Related entity primary key** : the name of the related entity primary key. In our sample this is the primary key of the "Book" entity, "BookId".

- **Intermediary entity foreign key with our table** : this is the foreign key from the intermediary entity which relates to our current entity. In our sample this is "AuthorId".

- **Intermediary entity foreign key with the related table** : this is the foreign key from the intermediary entity which relates to our related entity. In our sample this is "BookId".

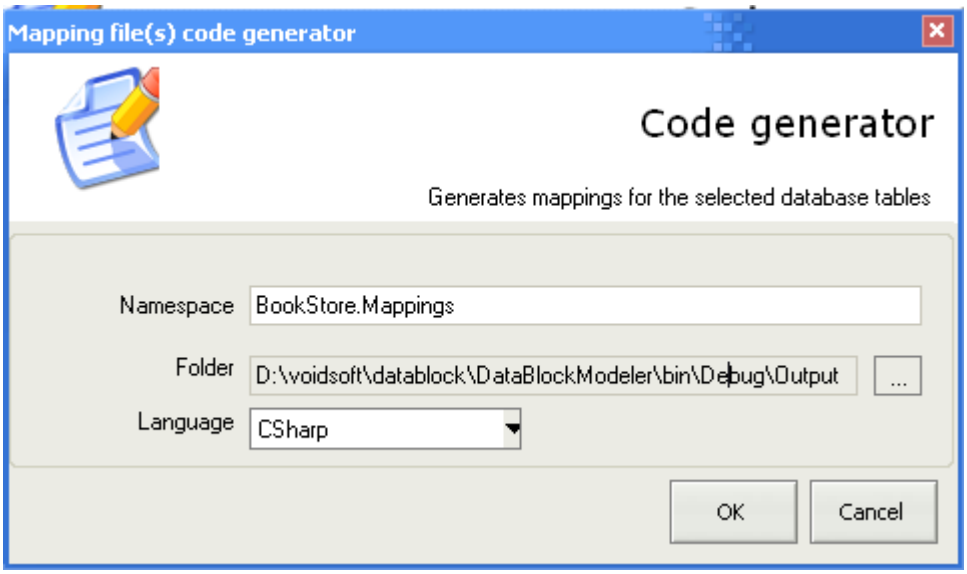
Ok. So after adding relations to the entities it's time to generate some code. We can generate 2 types of files(classes) :

- **TableMetadata** (the mapping files) which are generated FOR EACH entity.

- **PersistentObject** (your business façade for a specific entity). These should be generated only when needed. For instance if we have a Parent –> Child (One to Many cardinality) relation between 2 entities and the child entity is only a placeholder data (and it doesn't have relations with other entities) for the parent entity then a business façade for the child entity is pointless because all the data manipulation will be done from the parent entity.

The DataBlock Modeler is pretty flexibile and allows you to generate multiple/single mapping files, multiple/single persistent object or these 2 combined.

Let's show a simple mapping generation : please check the name of a entity from the list and then Generate/ Generate mapping files. You'll see a screen like this :



You can set the following attributes:

VERY IMPORTANT

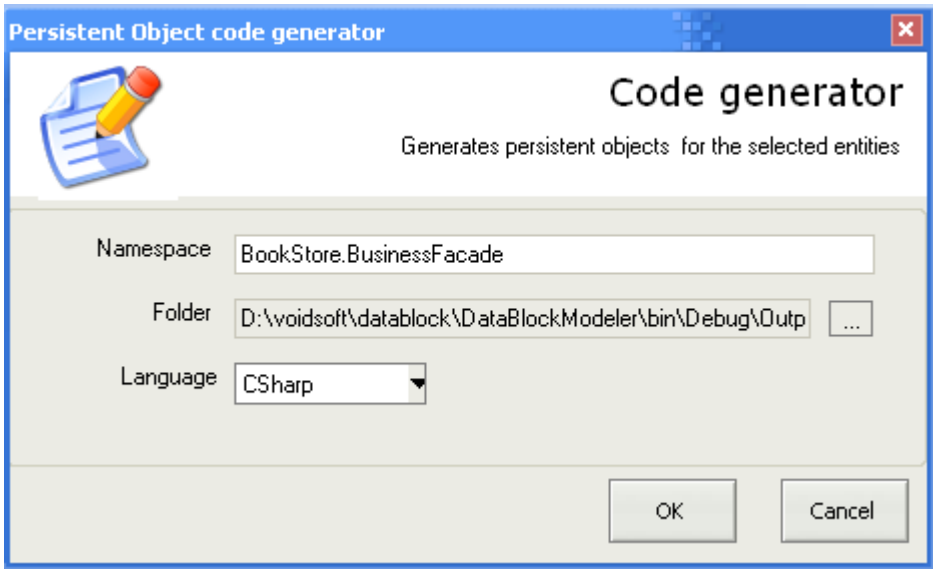
The namespace must be the same for all the mapping files. Also we recommend you to

compile all the mapping classes in a single library.

- **Namespace:** The namespace's name.
- **Folder:** the output folder. Make sure you have write right there.
- **Language:** The output language. Can be C# or VB.NET

Note that when choosing to generate mapping files for multiple entities, these will be generated each in his own file. Also the class (and file) naming scheme is this : name of the entity stripped by the empty string (so the table Author will have the file Author.cs/vb in which the class Author is declared, the table Line Items will have a file LineItems.cs/vb in which the class LineItems will be declared).

Generating PersistentObjects :



The attributes are the same as above. There is no restriction on placement of the persistent object classes.

After the generation process is finished you must compile the generated files :

- the mapping files (TableMetdata) must be all compiled in a library (and all have the same namespace).

- the persistent object files (PersistentObject) can be compiled however you like (each in his own library or all in a single library). Make sure that this library references the library of the TableMetadata files.

VERY IMPORTANT

If the database schema is changed we recommend you to regenerate the mapping files instead of hacking manually the existing ones.

So let's see now what methods were added to a mapping file which has relations. Basically you can manipulate related entities from the base entity in 3 ways :

- **retrieve related entities** : this is done with the "Get" method. Note that for each relations the entity will have a "Get" and the related entity name (for instance in the many to many relation above the entity "Author" will have a GetBooks() method which will return a array of TableMetadata with related books). If the relation's cardinality is many to many or one to many the function will return an array. Otherwise it will return a single TableMetadata object.

VERY IMPORTANT
Please notice that the "Get" methods return a TableMetadata or an array of TableMetadata so you must cast the result into the desired mapping class.

- **add entities :** this is done with the "AddTableMetadata" method. We can "Add" related entities to the parent entity in a Parent -> Child (One to Many cardinality) and then cascade insert data using the related PersistentObject "Create" or "Update" method.

- **remove entities** : this is done with the "RemoveTableMetadata" method. We can remove related entities from the parent entity in a Parent -> Child (One to Many cardinality) and then delete the "removed" rows using the related PersistentObject "Update" method. Please note that, if "Cascade delete" is enabled in a Parent -> Child relation and you delete the parent the child entities are automatically deleted.

Configuration :

a) **Configuring DataBlock from the application config file**.

The configurable properties of DataBlock are in voidsoft.DataBlock.Configuration class. These properties can be set programmatically at runtime or they can be read from the application configuration file. These are :

- **Connection String** : Is useful to set this when working with a single database in your application. However note that is risky to put user names and passwords in clear text in the config file.

- **Isolation Level** – this is the transaction isolation level used when multiple queries are running in a transaction. Suported values are :

  "1" -> Chaos
  "2" -> ReadCommited
  "3" -> ReadUncommited
  "4" -> RepeatableRead
  "5" -> Serializable
  "6" -> Unspecified

- **Database Server Type** – type of the database server. Valid values are :
  - "Access"
  - "SqlServer"
  - "MySql"

- **Command timeout** - a values which specifies the timeout for a command. Default value is 50 seconds.

- **LogEnabled** – Boolean flag to enable/disable the DataBlock logging. Valid values are "true" or "false". The default value is "false".

- **LogFilePath** – allows you to set the path of the log file. If you don't set this and logging is enabled then the file used for logging will be "datablocklog.txt" located in the root of your application.

To configure DataBlock from the config file you must add these attributes in your application config file and call voidsoft.Configuration.ReadDataFromConfigFile() at application startup.

b) Programatic configuration

You can configure DataBlock programmatically by setting the properties from the Configuration class. If you choose to do this make sure you set the values right at the application start before attempting to initialize the business objects.

c) Configuring multiple database providers

- configuring support for a native provider

Alternate database providers can be configured from the config file. You must add 2 simple entries. Here is a example for setting MySql to run with its .NET Connector

```
<add key = "ProviderMySql" value = "MySql.Data.dll"/>
<add key = "ProviderMySqlParameterChar" value="?" />
```

Please note that, by default, for Access and SqlServer DataBlock is using the managed providers (OleDb for Access and SqlClient for SqlServer) that are part of the base class library. For other supported databases the provider must be configured as shown above.

- configuring support for the ODBC driver

The configuration entry looks like this :

```
<add key = "ProviderMySql" value = "odbc"/>
<add key = "ProviderMySqlParameterChar" value="?" />
```

Mapping objects

The structure of a generated mapped object look like this :

```
[C#]
public class Category : TableMetadata
{
 //generated implementation
}


[VB.NET]
Public Class Category
  Inherits TableMetadata

 'generated implementation

End Class
```

The mapped object inherit from a class called TableMetadata. This base class exposes informations to the persistent object about the state of the mapped object.

NOTE : The generated class **should not** be modified by hand. In case the database schema changes the mapping class should be regenerated by the DataBlock Modeler.

Database supported types:
DataBlock supports the types defined in System.Data.DbType enumeration. These are :

AnsiString	A variable-length stream of non-Unicode characters ranging between 1 and 8,000 characters.
AnsiStringFixedLength	A fixed-length stream of non-Unicode characters.
Binary	A variable-length stream of binary data ranging between 1 and 8,000 bytes.
Boolean	A simple type representing Boolean values of **true** or **false**.
Byte	An 8-bit unsigned integer ranging in value from 0 to 255.
Currency	A currency value ranging from -2^{63} (or -922,337,203,685,477.5808) to $2^{63}-1$ (or +922,337,203,685,477.5807) with an accuracy to a ten-thousandth of a currency unit.
Date	Date and time data ranging in value from January 1, 1753 to December 31, 9999 to an accuracy of 3.33 milliseconds.
DateTime	A type representing a date and time value.
Decimal	A simple type representing values ranging from 1.0×10^{-28} to approximately 7.9×10^{28} with 28-29 significant digits.
Double	A floating point type representing values ranging from approximately 5.0×10^{-324} to 1.7×10^{308}

	with a precision of 15-16 digits.
Guid	A globally unique identifier (or GUID).
Int16	An integral type representing signed 16-bit integers with values between -32768 and 32767.
Int32	An integral type representing signed 32-bit integers with values between -2147483648 and 2147483647.
Int64	An integral type representing signed 64-bit integers with values between -9223372036854775808 and 9223372036854775807.
Object	A general type representing any reference or value type not explicitly represented by another **DbType** value.
SByte	An integral type representing signed 8-bit integers with values between -128 and 127.
Single	A floating point type representing values ranging from approximately 1.5×10^{-45} to 3.4×10^{38} with a precision of 7 digits.
String	A type representing Unicode character strings.
StringFixedLength	A fixed-length stream of Unicode characters.
Time	Date and time data ranging in value from January 1, 1753 to December 31, 9999 to an accuracy of 3.33 milliseconds.
UInt16	An integral type representing unsigned 16-bit integers with values between 0 and 65535.
UInt32	An integral type representing unsigned 32-bit integers with values between 0 and 4294967295.
UInt64	An integral type representing unsigned 64-bit integers with values between 0 and 18446744073709551615.
VarNumeric	A variable-length numeric value.

PersistentObjects

The DataBlock framework creates a business façade for your entities using the **domain model pattern.** Let's take a look at this from the conceptual point of view. You have a entity in your applications called Category. You application must process data about categories. So we create a "Category" table in our database. DataBlock creates 2 types of object for your database entity:

- **CategoryTableMetadata** : which derives from TableMetadata and represents the mapping class for your entity.

- **CategoryPersistentObject :** which represents the business façade for your object and interacts with the database.

So let's see how we can use the business façade in our application:

Initializing a PersistentObject in different contexts

The PersistentObject can be used in 3 different contexts:

a) with a specified database server

In this case the constructor looks like :

```
[C#]
public CategoriesPersistentObject(DatabaseServer database,
                                  string connectionString,
                                  TableMetadata mainTable) : base (database,
                              connectionString, mainTable)


[VB.NET]
Public Sub New (database as DatabaseServer,
                connectionString as String,
                  mainTable as TableMetadata)
              MyBase.New(database, connectionString, mainTable)
End Sub
```

The arguments which are passed to the constructor are :

- DatabaseServer type : the type of the database server with which this instance will communicate. This argument is necessary because you can access multiple database servers in your application.
- Connection string : the connection string used to connect to the database.
- TableMetadata : the mapping object with which the PersistentObject is associated.

b) with the generic database server

```
[C#]
public CategoriesPersistentObject( TableMetadata mainTable) : base
(mainTable){
}

[VB.NET]
Public Sub New (mainTable as TableMetadata)
              MyBase.New(mainTable)
 End Sub
```

This context is just as the above context with the difference that now DataBlock will use the DatabaseServer and connection string specified in the Configuration file. This is very useful when you have to access a single database in your application.

c) in the context of a Session

```
[C#]
public CategoriesPersistentObject(Session session,
                                  TableMetadata mainTable) : base(session,
                                  mainTable)
{
}

[VB.NET]
Public Sub New (s as Session, mainTable as TableMetadata)
    MyBase.New(s, mainTable)
End Sub
```

You can initialize the PersistentObject to run in the context of a session thus enabling transactional support for the PersistentObject methods.

Reading entities :

A PersistentObject allows you to read data from database and return the values in various formats : TableMetadata arrays, datasets, single objects and ArrayList / SortedLists.

1. GetTableMetadata – allows you to read data as a TableMetadata array.

This returns a CategoryTableMetadata array which contains all the categories from the database. So you have a array of Category objects which can be "consumed" from the presentation layer : the CategoryTableMetadata can be binded to data grids, listboxes, textboxes etc.

- you can retrieve a single object by specifing the primary key value as in :

```
[C#]
Category categories = (Category) categ.GetTableMetadata(5);
[VB.NET]
Dim category As Category =  categ.GetTableMetadata(5)
```

- you can retrieve all the data like this :

```
[C#]
Category[] categories = (Category[]) categ.GetTableMetadata();
[VB.NET]
Dim category As Category =  categ.GetTableMetadata(5)
```

- there is also an overloads which returns a subset of data by using a QueryCriteria.

An important note here : make sure you are selecting all the DatabaseFields of an entity (by initializing the Query criteria with the TableMetadata overload).

```
[C#]
QueryCriteria qc = new QueryCriteria(ctg);
qc.Add(CriteriaOperator.Higher, ctg.GetField("Id"), 40);
Category[] categories = (Category[]) categ.GetTableMetadata(qc);
[VB.NET]
Dim qc As New QueryCriteria(ctg);
qc.Add(CriteriaOperator.Higher, ctg.GetField("Id"), 40);
Dim category As Category =  categ.GetTableMetadata(5)
```

2. GetDataSet - allows you to return data as a dataset.

To retrieve all the domain objects data from database you can use:

```
[C#]
DataSet ds = categ.GetDataSet();

[VB.NET]
Dim ds As DataSet = categ.GetDataSet
```

This will return a DataSet with all the categories from the database. To return only certain fields you can use :

```
DataSet ds = categ.GetDataSet(categ.GetFieldByName("Description"));

[VB.NET]
Dim ds As DataSet = categ.GetFieldList(category.GetFieldByName("Description"))
```

You can also use a QueryCriteria (for a in depth reference to QueryCriteria please check out the "Writing object oriented queries" section) to specify only the data you need.

This will return a dataset containing only the categories which primary key has a value bigger then 100.

```
[C#]
QueryCriteria qc = new QueryCriteria(category);
qc.Add(CriteriaOperator.Higher, category.TableFields[0], 100);
DataSet ds = categ.GetDataSet(qc);

[VB.NET]
Dim qc As QueryCriteria = New QueryCriteria(category)
qc.Add(CriteriaOperator.Higher, category.TableFields(0), 100)
Dim ds As DataSet = categ.GetDataSet(qc)
```

The last overload allows you to "lazy read" related data to a certain item and return the result as a dataset.

3. Get ArrayList / SortedList - this should be used to read data directly from the database without the overhead of creating datasets or table metadata instances.

To return data as StringCollection/StringDictionary you can use :

```
[C#]
ArrayList list = categ.GetFieldList(category.GetFieldByName("Description"));

[VB.NET]
Dim list As ArrayList =
categ.GetFieldList(category.GetFieldByName("Description"))
```

4.Get a single value - to return a single value from the database you can use :

```
[C#]
QueryCriteria qc = new QueryCriteria(category.TableName,
category.TableFields[0]);
qc.Add(CriteriaOperator.Equality, category.TableFields[0], 20);
object result = categ.GetValue(qc);

[VB.NET]
Dim qc As QueryCriteria = New QueryCriteria(category.TableName,
category.TableFields(0))
qc.Add(CriteriaOperator.Equality, category.TableFields(0), 20)
Dim result As Object = categ.GetValue(qc)
```

Deleting entities:

The objects can be deleted using the Delete method of the PersistentObject. You can delete entities in 2 ways :

- delete a single entity

```
[C#]
Category cat = categ.GetTableMetadata(5);
categ.Delete(cat);

[VB.NET]
Dim cat As Category =  categ.GetTableMetadata(5)
categ.Delete(cat)
```

Note, in the above code, that if your entity is the Parent in one or more Parent -> Child relations and cascade delete is enabled the child entities will also be deleted.

- delete multiple entities using a QueryCriteria

```
[C#]
QueryCriteria qc = new QueryCriteria(category);
qc.Add(CriteriaOperator.Higher, category.TableFields[0], 100);
categ.Delete(qc);

[VB.NET]
Dim qc as new QueryCriteria(category);
qc.Add(CriteriaOperator.Higher, category.TableFields[0], 100)
categ.Delete(qc)
```

Updating entities :

- updating a single "table based" entity

```
[C#]
CategoryTableMetadata cat = categ.GetTableMetadata(5);
cat.Description = "My New Category";
categ.Update(cat);

[VB.NET]
Dim cat As CategoryTableMetadata =  categ.GetTableMetadata(5)
cat.Description = "My New Category"
categ.Update(cat)
```

This shows you how to update a single entity

- updating a entity with relations

Lets suppose that out Category entites has a Parent -> Child relationship (with the OneToMany cardinality) with another entity called CategoryDetails.

Here is how we can add, update and delete the parent and child entities.

```
[C#]
//first we load the related entity data
CategoryDetails[] details = categ.GetCategoryDetails();

//let's suppose we have 2 CategoryDetails returned.
//we edit one
details[0].Name = "Name";
categ.AddCategoryDetail(details[0]);
//we remove one
categ.Remove(details[1]);
//and we add a new one
CategoryDetails newCategory = new CategoryDetails();
newCategory.Name = "New";
categ.AddCategoryDetail(newCategory);

//and we update
categPersistent.Update(categ);

[VB.NET]
//first we load the related entity data
Dim details as CategoryDetails() = categ.GetCategoryDetails();

//let's suppose we have 2 CategoryDetails returned.
//we edit one
details(0).Name = "Name";
categ.AddCategoryDetail(details(0));
//we remove one
categ.Remove(details(1));
//and we add a new one
Dim newCategory as New CategoryDetails();
newCategory.Name = "New";
categ.AddCategoryDetail(newCategory);

//and we update
categPersistent.Update(categ);
```

- updating multiple entities with a QueryCriteria. In the exemple below the names of all Categories which have the Id higher than 100 will be updated to "Test".

```
[C#]
category.GetField("Name").FieldValue = "Test";
QueryCriteria qc = new QueryCriteria(category.TableName,
category.GetField("Name"));
qc.Add(CriteriaOperator.Higher, category.GetField("Id"), 100);
categ.Update(qc);

[VB.NET]
Category.GetField("Name").FieldValue = "Test";
Dim qc as new QueryCriteria(category);
qc.Add(CriteriaOperator.Higher, category.TableFields[0], 100)
categ.Delete(qc)
```

Creating entities :

- creating a simple entity. Here is an example which creates a new entity. In this example we suppose that the primary key of the table is an autoincremented integer.

```
  [C#]
Category cat = new Category ();
cat.Description = "Fish";
cat.CategoryName = "Sea stuffs";
categ.Create(cat);

[VB.NET]
Dim cat As Category New Category ()
cat.Description = "Fish"
cat.CategoryName = "Sea stuffs"
categ.Create(cat)
```

If the primary key is not auto incremented then you can set its value from the code :

```
  [C#]
Category cat = new Category ();
cat.Id = Guid.New();
cat.Description = "Fish";
cat.CategoryName = "Sea stuffs";
categ.Create(cat);

[VB.NET]
Dim cat As Category New Category ()
Cat.Id = Guid.New()
cat.Description = "Fish"
cat.CategoryName = "Sea stuffs"
categ.Create(cat)
```

- creating recursive multiple entities. For example let's suppose that the Category entity has a Parent -> Child relation with the CategoryDetails entity. Also the CategoryDetails has a Parent -> Child relationship with another entity called Dealers. At his turn Dealers itself has a Parent -> Child relationship with a entity called DealerDetails. So here we are with a nice 4 levels hierarchy. Here is how simple it is to create a new Category.

```
[C#]
Category ca = new Category();
ca.Name = "New Category";

CategoryDetails ctg = new CategoryDetails();
ctg.Name = "NewCategoryDetails";

ca.AddCategoryDetails(ctg);

Dealers d = new Dealers();
d.Name = "New Dealer IN Town";

ctg.AddDealer(d);

DealerDetails ddetails = new DealerDetails();
d.AddDealerDetails(ddetails);

categ.Update(ca);

[VB.NET]
Dim ca As New Category()
ca.Name = "New Category"

Dim ctg As New CategoryDetails();
ctg.Name = "NewCategoryDetails";

ca.AddCategoryDetails(ctg);

Dim d As New Dealers();
d.Name = "New Dealer IN Town";

ctg.AddDealer(d);

Dim ddetails As New DealerDetails();
d.AddDealerDetails(ddetails);

categ.Update(ca);
```

Accessing executing queries at runtime

DataBlock allows you to access the queries at runtime before execution. This is very useful if you want to extend the operations (for instance if you want to do SQL logging). To do this you must override the Before ExecutionQueries method like this :

```
[C#]
public override void BeforeExecutingQueries(Operation operation,
                                            ref List<ExecutionQuery> list,
                                            params object[] args)
{
}

[VB.NET]
Public Overrides  Sub BeforeExecutingQueries(ByVal operation As Operation,
                                          ByRef List(Of ExecutionQuery) as list,
ByVal ParamArray As Object())

End Sub
```

The Operation enumeration shows you what type of operation is execution now. The Custom field is for your own methods when you're extending the PersistentObject.

Writing object oriented queries with QueryCriteria :

The QueryCriteria is an API which allows you to "model" database independent queries in a object oriented way. Here is how you can use it:

- you specify what DatabaseFields you want to select in the constructor. You can select all the fields of a entity like this :

```
[C#]
Category categ = new Category();
QueryCriteria qc = new QueryCriteria(categ);

[V.NET]
Dim categ As New Category();
Dim qc As New QueryCriteria(categ)
```

Or select a subset of fields :

```
[C#]
QueryCriteria qc = new QueryCriteria(categ.TableName, categ.GetField("Id",
categ.GetField("Description"));

[V.NET]
Dim qc As New QueryCriteria(categ.TableName, categ.GetField("Id"),
categ.GetField("Description")
```

After specifying what fields you want to select you can add criterias to the fields.

List of criteria operators :

You can add criterias to the queries by using the supported operators:

- **OrderBy** : orders the result set in ascending or descending mode.

```
[C#]
qc.Add(CriteriaOperator.OrderBy, cat.TableFields[0], "asc");

[VB.NET]
qc.Add(Criteria.Operator.OrderBy, cat.TableFields(0), "asc")
```

- **Distinct** : filters the result set by selecting only distinct values.

```
[C#]
qc.Add(CriteriaOperator.Distinct, cat.TableFields[0]);

[VB.NET]
qc.Add(Criteria.Operator.Distinct, cat.TableFields(0))
```

- **Between** : filters the result set by selecting only the data between 2 given values

```
[C#]
qc.Add(CriteriaOperator.Between, cat.TableFields[0], 5, 70 );

[VB.NET]
qc.Add(Criteria.Operator.Between, cat.TableFields(0), 5, 50)
```

- **Not** : filters the result set by adding a negation.

```
[C#]
qc.Add(CriteriaOperator.Not, cat.TableFields[0], 5);

[VB.NET]
qc.Add(Criteria.Operator.Not, cat.TableFields(0), 5)
```

- **Like** : filters the result set by adding a "like" condition.

Note : do NOT add the "%" to the string because it will be appended by DataBlock.

```
[C#]
qc.Add(CriteriaOperator.Like, cat.TableFields[1], "Roches");

[VB.NET]
qc.Add(Criteria.Operator.Like, cat.TableFields(1), "Roches")
```

- **Equality** : filters the result set by adding a equality operator

```
[C#]
qc.Add(CriteriaOperator.Equality, cat.TableFields[0], 5);

[VB.NET]
qc.Add(Criteria.Operator.Equality, cat.TableFields(0), 5)
```

- **IsNull** : filters the result set by included the values which are null

```
[C#]
qc.Add(CriteriaOperator.IsNull, cat.TableFields[0]);

[VB.NET]
qc.Add(Criteria.Operator.IsNull, cat.TableFields(0))
```

- **IsNotNull** : filters the result set by including the values which are not null

```
[C#]
qc.Add(CriteriaOperator.IsNotNull, cat.TableFields[0]);

[VB.NET]
qc.Add(Criteria.Operator.IsNotNull, cat.TableFields(0))
```

- **Or** : adds a simple "OR" operator between 2 conditions

```
[C#]
qc.Add(CriteriaOperator.Equality, cat.TableFields[0], 5);
qc.Add(CriteriaOperator.Or, cat.TableFields[0]);
qc.Add(CriteriaOperator.Equality, cat.TableFields[1],26);

[VB.NET]
qc.Add(Criteria.Operator.Equality, cat.TableFields(0), 5)
qc.Add(CriteriaOperator.Or, cat.TableFields[0]);
qc.Add(CriteriaOperator.Equality, cat.TableFields[1], 26);
```

- **Smaller** : filters the result set by adding a "smaller then" condition

```
[C#]
qc.Add(CriteriaOperator.Smaller, cat.TableFields[0], 67);

[VB.NET]
qc.Add(Criteria.Operator.Smaller, cat.TableFields(0), 67)
```

- **SmallerOrEqual** : filters the result set by adding a "smaller then" condition

```
[C#]
qc.Add(CriteriaOperator.SmallerOrEqual, cat.TableFields[0], 67);

[VB.NET]
qc.Add(Criteria.Operator.SmallerOrEqual, cat.TableFields(0), 67)
```

- **Higher** : filters the result set by adding a "higher then" condition

```
[C#]
qc.Add(CriteriaOperator.Higher, cat.TableFields[0], 67);

[VB.NET]
```

```
qc.Add(Criteria.Operator.Higher, cat.TableFields(0), 67)
```

- **HigherOrEqual** : filters the result set by adding a "higher or equal then" condition

```
[C#]
qc.Add(CriteriaOperator.HigherOrEqual, cat.TableFields[0], 67);

[VB.NET]
qc.Add(Criteria.Operator.HigherOrEqual, cat.TableFields(0), 67)
```

- **Max** : filters the result set by adding a "max" condition

```
[C#]
qc.Add(CriteriaOperator.Max, cat.TableFields[0]);

[VB.NET]
qc.Add(Criteria.Operator.Max, cat.TableFields(0))
```

- **Min** : filters the result set by adding a "min" condition

```
[C#]
qc.Add(CriteriaOperator.Min, cat.TableFields[0]);

[VB.NET]
qc.Add(Criteria.Operator.Min, cat.TableFields(0))
```

- **Count** : filters the result set by adding a "count" condition

```
[C#]
qc.Add(CriteriaOperator.Count, cat.TableFields[0]);

[VB.NET]
qc.Add(Criteria.Operator.Count, cat.TableFields(0))
```

Writing Joins

The QueryCriteria API allows you to write joins between 2 or more tables. You must add the query criteria of the child table to the query criteria of the parent table.

Example :
Let's suppose there is a relationship between Customer and Client (and Customer is the parent table). An inner join query can be expressed like this :

```
[C#]
Customer cust = new Customer();
Client ct = new Client();
QueryCriteria qcCustomer = new QueryCriteria(cust);
QueryCriteria qcClient = new QueryCriteria(client);
qcCust.AddJoin(JoinType.Inner, cust.GetField("CustomerId"),
client.GetField("CustomerId"));

[VB.NET]
Dim cust As New Customer()
Dim ct As New Client()
Dim qcCustomer As New QueryCriteria(cust)
Dim qcClient As New QueryCriteria(client)
qcCust.AddJoin(JoinType.Inner,cust.GetField("CustomerId"),client.GetField("CustomerId"))
```

Let's suppose we have to write a join between 3 entities : Order, OrderDetails and Customer and also add criterias. And some field aliases. Here is how we can write this

```
[C#]
Order o = new Order();
QueryCriteria qc = new QueryCriteria(o);

//add criterias to order
o.Add(CriteriaOperator.Higher, o.TableFields[0], 50);

//add OrderDetails
OrderDetails ood = new OrderDetails();
QueryCriteria qcDetails = new QueryCriteria(ood.TableName, ood.TableFields[4]);
//add criterias to order details
qcDetails.Add(CriteriaOperator.Equality, ood.TableFields[0], 100);

//add Customer
Customer cst = new Customer();
QueryCriteria qcCust = new QueryCriteria(cst.TableName, cst.TableFields[4]);
//add criterias to customer
qcCust.Add(CriteriaOperator.Equality, cst.TableFields[0], 100);


//add now the joins
qc.AddJoin(JoinType.Inner, o.TableName, o.GetField("Id"), ood.TableName,
ood.GetField("Id"));
qc.AddJoin(JoinType.Inner, ood.TableName, ood.GetField("CustomerId"), cst.TableName,
cst.GetField("Id"));

[VB.NET]
Dim o As New Order()
Dim qc As New QueryCriteria(o)

'add criterias to order
o.Add(CriteriaOperator.Higher, o.TableFields[0], 50)

'add OrderDetails
Dim ood As New OrderDetails()
Dim qcDetails As New QueryCriteria(ood.TableName, ood.TableFields[4])
'add criterias to order details
qcDetails.Add(CriteriaOperator.Equality, ood.TableFields[0], 100)

'add Customer
Dim cst As New Customer()
Dim qcCust As New QueryCriteria(cst.TableName, cst.TableFields[4])
'add criterias to customer
qcCust.Add(CriteriaOperator.Equality, cst.TableFields[0], 100)


'add now the joins
qc.AddJoin(JoinType.Inner, o.TableName, o.GetField("Id"), ood.TableName,
ood.GetField("Id"))
qc.AddJoin(JoinType.Inner, ood.TableName, ood.GetField("CustomerId"), cst.TableName,
cst.GetField("Id"))
```

Please note that you add the join between the last 2 table to the "main" QueryCriteria (in the above example is "qc").

Adding field aliases to queries:
The QueryCriteria also support field aliases to prevent name clashing in the results. Here is a sample on how to use this :

```
[C#]
QueryCriteria qc = new QueryCriteria(categ);
qc.AddAlias("Id", "NewId");

[VB.NET]
Dim qc As New QueryCriteria(categ);
  qc.AddAlias("Id", "NewId")
```

Make sure that the name of the field is in the list of selected fields.

Session :

A session is a high level object which allows you to :

- run multiple persistent object operations on the same database connection.

- run multiple unrelated persistent object operations inside a transaction. This enabled what we call "long running transactionable operations". These types of operations are very useful. The Session allows you to quicky chain multiple business operations to run in a single transaction .

Here is a sample of how this works. Let's suppose we want to run in a transaction operation from 2 different entities:

[C#]

```
//create the session using the default database
Session ss = Session.CreateNewSession();

//or specify the database server
Session ss = Session.CreateNewSession(DatabaseServer.Access, connectionString);

//now init the entities in the context of the entities
CustomerPersistentObject cpo = new CustomerPersistObject(ss, cust);
OrderPersistentObject ops = new OrderPersistentObject(ss, order);

ss.BeginTransaction()

try
{
      //these 2 separate operations will be running in a transaction
      cpo.Delete(customer);
      ops.Create(order);

     //commit the session's transaction
     ss.Commit();
}
catch
{
     ss.Rollback();
}

//now we should also dispose the session (underlying the session also closes the
database connection)
ss.Dispose();
```

[VB.NET]

```
'create the session using the default database
Dim ss As Session =  Session.CreateNewSession()

'or specify the database server
Dim ss As Session =
Session.CreateNewSession(DatabaseServer.Access,connectionString)

'now init the entities in the context of the entities
Dim cpo As CustomerPersistentObject =  New CustomerPersistObject(ss,cust)
Dim ops As OrderPersistentObject =  New OrderPersistentObject(ss,order)

ss.BeginTransaction()

Try
'these 2 separate operations will be running in a transaction
cpo.Delete(customer)
ops.Create(order)

     'commit the session's transaction
     ss.Commit()
Catch
     ss.Rollback()
End Try

'now we should also dispose the session (underlying the session also closes the
database connection)
```

```
ss.Dispose()
```

Transactions :

Multiples queries are always executed in transactions. DataBlock allows you to set a default IsolationLevel for transactions. The PersistentObject operations (Create, Delete, Update) are always executed with this isolation level. When extending the PersistentObject you have access to the underlying ExecutionEngine to whom you can pass a custom IsolationLevel. Also in the context of a session you can pass a custom isolation level.

Concurrency :

For delete and update operations DataBlock generates queries with optimistic concurrency (the WHERE clause is based **ONLY** on the primary key). If you would like to change that for the built in methods (Delete, Update) then you need to override the implementation. Please look at "**Extending the business façade"** section.

Exception handling :

DataBlock exposes 3 types of exceptions:

- **PersistentObjectException** : this one is the high level exception and it's being thrown by the business façade. You are advised to throw this exception in your business façade too. Usually this type of Exception wraps the SqlGeneratorException or the ExecutionEngineException.
- **SqlGeneratorException** : this is thrown by the SqlGenerator.
- **ExecutionEngineException :** this exception is thrown by the Execution engine.

Troubleshooting :

The best way to troubleshoot DataBlock is to enable the logging feature (please check the Configuration sections on how to do that). All the generated queries and exception + stack traces will be dumped in the log.

Extending the business façade :

DataBlock internals :

- ExecutionEngine
- SqlGenerator
- DataFactory

ExecutionEngine

The ExecutionEngine is used to run the queries. It operates in two ways: connected and disconnected mode. To operate in connected mode you must create a instance of the ExecutionEngine class :

```
[C#]
ExecutionEngine dal = ExecutionEngine.CreateNewInstance(EDatabase.SqlServer,
"Server=localhost;Integrated Security=SSPI;Database=Nothwind");

[VB.NET]
Dim dal As ExecutionEngine  =
DataAccessLayer.CreateNewInstance(EDatabase.SqlServer,"Server =
localhostIntegrated Security = SSPIDatabase = Nothwind")
```

If the connection with the database server is successful we can execute operations:

```
[C#]
dal.ExecuteNonQuery(insertQuery);
DataSet ds =  dal.ExecuteDataset(selectQuery);
IdataReader iread = dal.ExecuteReader(selectQuery);

[VB.NET]
dal.ExecuteNonQuery(insertQuery)
Dim ds As DataSet =  dal.ExecuteDataset(selectQuery)
Dim iread As IdataReader =  dal.ExecuteReader(selectQuery)
```

After running the required operation the ExecutionEngine instance **must** be disposed. Disposing the object also closes the underlying database connection.

```
[C#]
dal.Dispose();

[VB.NET]
dal.Dispose
```

The difference between connected and disconnected mode is that , in connected mode, the database connection remains always opened (ADO style) while in disconnected mode a database connection is opened at the beginning of a operation and closed after. To operate in "disconnected mode" you can use the static methods of the ExecutionEngine class. In this mode the database connection is opened and closed before and after each operation.

```
[C#]
DataSet ds = ExecutionEngine.ExecuteDataSet(EDatabase.SqlServer,
"Server=localhost;Integrated  Security=SSPI;Database=Nothwind", selectQuery);

[VB.NET]
Dim ds As DataSet =
ExecutionEngine.ExecuteDataSet(EDataMyBase.SqlServer,"Server=localhost;Integrated
Security=SSPI;Database=Nothwind",selectQuery)
```

The operations supported by the ExecutionEngine (in both connected and disconnected modes) are :

1.ExecuteNonQuery – allows you to execute a SQL statement against the specified database server. Returns the number of affected rows.
2.ExecuteReader – allows you to get a IdbReader. The reader is created with the

CommandBehaviour.CloseConnection, so when the reader is closed the undelying database connection is also closed.

3.ExecuteDataSet – allows you to fill a DataSet.

4.ExecuteScalar - executes the query, and returns the first column of the first row in the resultset returned by the query.

5.ExecuteXmlReader – this is a Microsoft Sql Server 2000 exclusive operation which returns the result of the query as an XmlDocument.

SqlGenerator :

The SqlGenerator's job is, as the name implies, to generate SQL queries based on the TableMetadata format. This **isn't** a standard SQL 2003 query generator as the the queries are generated independently for each of the supported database server : Access, Sql Server and MySql. The SqlGenerator can generate select, update, insert and delete queries for a entity but also it can generate hierarchical list of queries based upon the relations between entitites.

DataFactory :

The data factory is a factory pattern class used to initialize datababase + ADO.NET provider objects based on the specified database server. A special particularity of the DataFactory is that it supports a "plug in" provider model. Thus you can use different providers with the same database server. These providers can be configured by editing the config file. (please have a look at the Configuration section for more details).

Extending the business façade :

Let's see how you can extend the business façade:

1. **Overriding the current implementation :**

   Note that all methods from the PersisteObject class are virtual. You can override them and write your own implementation for those operations. For instance DataBlock uses optimistic concurrency for updates and deletes. If you would like to use pessimistic concurrency you can simple override the update implementation like this :

```
[C#]
public override void Update(TableMetadata table)
{
    //here you generate the SQL update code based on the
}

[VB.NET]
Public Overrides  Sub Update(ByVal table As TableMetadata)
    'here you generate the SQL update code based on the
End Sub
```

2. **Extending the business façade with entity specific operations.**

   Before showing you how to extend the PersistentObject you must be aware of the execution context. There can be 3 execution contexts inside a PersistentObject:

   - running in a session without transaction
   - running in the context of a session with transaction
   - normal context

Let's suppose we want to extend the business façade of out Category entity with a Here is how we do it. Please check the inline comments

```
[C#]
public class CategoryPersistentObject : PersistentObject
{
      public void DeleteForeignCustomers(Customer cst)
      {
            List<ExecutionQuery> listQueries = null;

            try
           {
                 //init the list of queries
               listQueries = new List<ExecutionQuery>();

               //we get the execution query for out customer
               listQueries.Add(SqlGenerator.GenerateDelete(cst));

              //at the same time up update the stock
              Stock st = new Stock();
              st.Price = cst.Price * 40;

              //generate update for the stock
               listQueries.Add(SqlGenerator.GenerateUpdate(st,
               st.GetField("Price")));

        //now we have 2 queries which must be runned in a transaction


               //check the execution context
               if(this.sessionContext != null)
               {
                   //run in the context of a session

                   //check for transaction
                   if(this.sessionContext.IsInTransaction)
                   {
                       //we are running in a transaction
                       foreach(ExecutionQuery q in listQueries)
                       {
                           this.contextSession.Queries.Add(q);
                       }
                   }
                   else
                 {
                        //not in transaction
                        this.dal.ExecuteNonQuery(listQueries);
                    }
               }
               else
               {
    //execute the list of queries with the configured transaction
    isolation level
                ExecutionEngine.ExecuteNonQuery(this.database,
              this.connectionString, listQueries,
              Configuration.DefaultTransactionIsolationLevel);
                   }
```

```
                }
                catch (Exception ex)
                {
                    throw new PersistentException(ex.Message, ex;
                }
            }
        }
```

[VB.NET]

```
Public Class CategoryPersistentObject
     Inherits PersistentObject


Public  Sub DeleteForeignCustomers(ByVal cst As Customer)


Try

    'we get the execution query for out customer
     listQueries.Add(SqlGenerator.GenerateDelete(cst))

                          'at the same time up update the stock
     Dim st As Stock New Stock()
     st.Price = cst.Price * 40

    'generate update for the stock
     listQueries.Add(SqlGenerator.GenerateUpdate(st,
st.GetField("Price")))

    'now we have 2 queries which must be runned in a transaction
    'check the execution context

   If Not Me.sessionContext Is Nothing Then
                    'run in the context of a session

                    'check for transaction
                    If Me.sessionContext.IsInTransaction Then
                        'we are running in a transaction
                        Dim q As ExecutionQuery
                        For Each q In listQueries
                            Me.contextSession.Queries.Add(q)
                        Next
                    Else
                         'not in transaction
                         Me.dal.ExecuteNonQuery(listQueries)
                    End If
                Else
'execute the list of queries with the configured transaction
isolation level
ExecutionEngine.ExecuteNonQuery(Me.database, Me.connectionString,
listQueries, Configuration.DefaultTransactionIsolationLevel)
   End If
Catch ex As Exception
       Throw New PersistentException(ex.Message, ex)
End Try
 End Sub
End Class
```

Requirements:

DataBlock needs a CLI 2.0 execution environment. For the moment only Microsoft.NET 2.0 meets the criteria. I haven't tested yet DataBlock with Mono but there is no reason the library won't work when Mono will be CLI 2.0 compliant.

DataBlock ships with support for the following databases:

-Microsoft Access 97 or higher.
-Microsoft SqlServer 7/2000/MSDE
-MySql 4 or higher
-PostgreSQL 8.x

About and contact

DataBlock © 2004 - 2005 Marius Gheorghe. All rights reserved.

Email : gmarius@gmail.com
Web : www.voidsoft.ro
Phone : +(40) 0741118274