Automating the Development of Data Access Layer

Robert Dollinger
rdolling@uwsp.edu
Mathematics and Computing Department, University of Wisconsin Stevens Point
Stevens Point, WI 54481, USA

Daniel V. Goulet
dgoulet@uwsp.edu
Mathematics and Computing Department, University of Wisconsin Stevens Point
Stevens Point, WI 54481, USA

David Gibbs
dgibbs@uwsp.edu
Mathematics and Computing Department, University of Wisconsin Stevens Point
Stevens Point, WI 54481, USA

Abstract

Modern software environments like XDE .NET provide integrated and powerful tools that designers and developers can use in all stages of the application development from building use case diagrams to automatically generating code. Such tools allow a much more systematic and highly automated approach of the entire development process transforming what not long ago was more like an "art" into a better understood engineering activity. Still, there are many challenges that designers and developers have to deal with in order to make the tools work properly and produce meaningful results. Some of these challenges are faced at the sensitive point of passing from the realm of Object Models to the one of Data Models. There is a missing link between these two, which still has to be coded by the application developers in what is called the Data Access Layer. In order to simplify developers' work we propose a two step approach in automating the creation of a Data Access Layer. The first step consists in factoring out the database specific functionality into a sub-layer which is completely independent of the specifics of the entity types of the given application. The second step consists of using dynamic code generation techniques in order to automatically generate code for the basic database operations associated to the application's entity types. As a result application developers will be able to use a data access class generator instead of having to write the class specific code themselves.

Keywords: Data Access Layer, Entity Classes, Data Access Classes, Data Access Manager, reflection, attributed programming, data access class generator.

1. INTRODUCTION

Based on the Rational Unified Process (RUP) specification, most software engineering manuals present the typical application development activity as a structured and systematic approach that would produce an object model organized in at least three main layers: Presentation Layer, Business Layer and Data Access Layer (Figure 1).

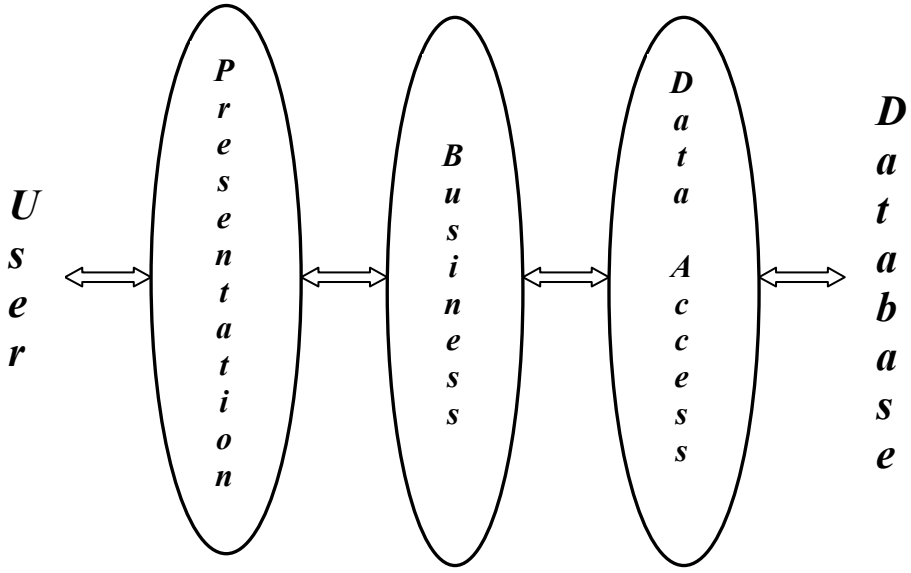

Figure 1: Typical Application Layers

Further on, each of these layers is refined in several class categories or sub-layers. The Presentation Layer is composed of the Boundary Classes, Windows or Web forms with which the users interact directly, and Presentation Logic Control Classes which implement the use cases. In the Business Layer we have the Business Logic Control Classes, implementing the application/enterprise business rules, and the Entity Classes which are the representations of the enterprise entity types. Entity types are common to all applications in the enterprise and thus most of them need to be stored persistently such that they can be shared across applications or for remembering vital enterprise status values. It is the Data Access Layer that makes the connection between the application and the associated persistent storage, typically a database. Its role is two-fold: (1) provide the functionality for transferring data back and forth between the application and the database, and (2) fill the conceptual gap between the representation model used in the application (object oriented) and the one used in the database (relational model). The first function of the Data Access Layer is supported by the Data Layer Classes which implement at least the select, update, insert and delete functions as the minimal set needed for the operation of a typical database. The Data Access Layer lies between the Object Model of the application and the Data Model of the data base thus functioning like an adapter between the two. The second sub-layer of the Data Access Layer consists of the Data Type Classes which are the in memory recipients of the entity classes' instances in the format of the Data Model corresponding to the persistent storage. For example in .NET the Data Type Classes are materialized as XML Data Sets which are memory structures closely mimicking the underlying database. (see Boggs, W. & Boggs M. 2003 and Manassis E. 2004). Modern integrated design and development environments like XDE .NET provide the tools for the creation of the Object Model of an application through a well-defined, systematic and traceable process. Part of the resulted Object Model, consisting of the Entity Classes, forms the Entity Model and usually needs to be stored persistently. One

can further use the XDE .NET tools to convert the Entity Model, consisting of class descriptions, into an equivalent, Database Design Model, consisting of tables, constraints, stored procedures and other. The Database Design Model can then be converted into a database description in the form of a DDL script or into a concrete database. The reverse process is equally easy at any of these stages. One can reverse engineer a database into a Database Design Model, and the later one into an Entity Model. Let us notice that in all these processes the Data Access Layer is completely skipped, which leaves to the application developer the task of filling the gap and make the functional connection between the (objectual) Entity Model and the (relational) Database Design Model. Traditionally, this functionality is implemented in the Data Layer Classes, one class associated to each Entity Class. A Data Layer Class captures both the specifics of its corresponding Entity Class (data members to be saved, constraints etc.) as well as the specifics of the database type where instances of the Entity Class are stored (e.g. Oracle versus Microsoft SQL Server). Fortunately this task can be substantially simplified and almost completely automated by the understanding of the fact that one can separate the two kinds of functions a Data Layer Class implements: functions specific to the Entity Class and functions specific to the database. Thus we propose a two step refinement of the Data Access Layer. The first step consists in factoring out the database specific functionality into a sub-layer which is completely independent of the specifics of the entity types of the given application. The second step consists of using reflection and dynamic code generation techniques in order to automatically generate code for the basic database operations associated to the application's entity types. As a result application developers will be able to use a data access class generator instead of having to write the class specific code themselves.

The advantages of using such an approach are two-fold. As a result of the first step, application developers can get a prefabricated Data Access Manager class capturing the specifics of dealing with the database system used as the persistent storage for the application's data. By simply instantiating this class with the right parameters and calling the right methods, developers would be able to perform the basic database related tasks of connecting to, opening and closing the database, etc. The parameters that need to be provided give the coordinates and type of the database system like server name, database, login, password etc. Much flexibility is gained in this way since one can switch from one database to another by simply changing the parameters without the need of writing new code. The second step is more challenging in that new code is needed in each application to capture the specifics of the Entity Classes. One cannot reuse the same code over several applications as in the case of the Data Access Manager class. The specifics of available database systems can be known ahead of time before any application would be developed, which is not the case with the specifics of the Entity Classes. What we can do is to automate the process of writing the code for these classes, which means that a prefabricated Data Access Class generator would be used to automatically produce the code for saving, deleting, updating or retrieving an Entity Class instance given as parameter.

2. A SIMPLE EXAMPLE

We illustrate the ideas presented in this paper through a simple example inspired by the Bradshaw Marina application from (Doke, E.R. et al. 2003). The class diagram in Figure 2 shows only a slice of the Entity Model from this application.

The main entity types are **Customer**, **Boat** and **Slip**. There is a many to many relationship between **Customer** and **Slip** which is captured by the association class called **Lease**. **Boat** is an abstract class specialized in two subclasses **Powerboat** and **Sailboat**. The diagram in Figure 2 also captures some significant business rules of the Bradshaw Marina enterprise. First as shown by the association relationships between **Boat** and **Customer**, each **Boat** belongs to exactly one **Customer,** while a **Customer** may have at most one **Boat** (or none!). Second, the relationships between **Slip** and **Boat** specify that a **Slip** may or may not be occupied by at most one **Boat,** and a **Boat** may have assigned a **Slip** (or none!).

Current modeling tools like XDE .NET are capable of automatically converting the above class diagram into a corresponding database schema which can then be further

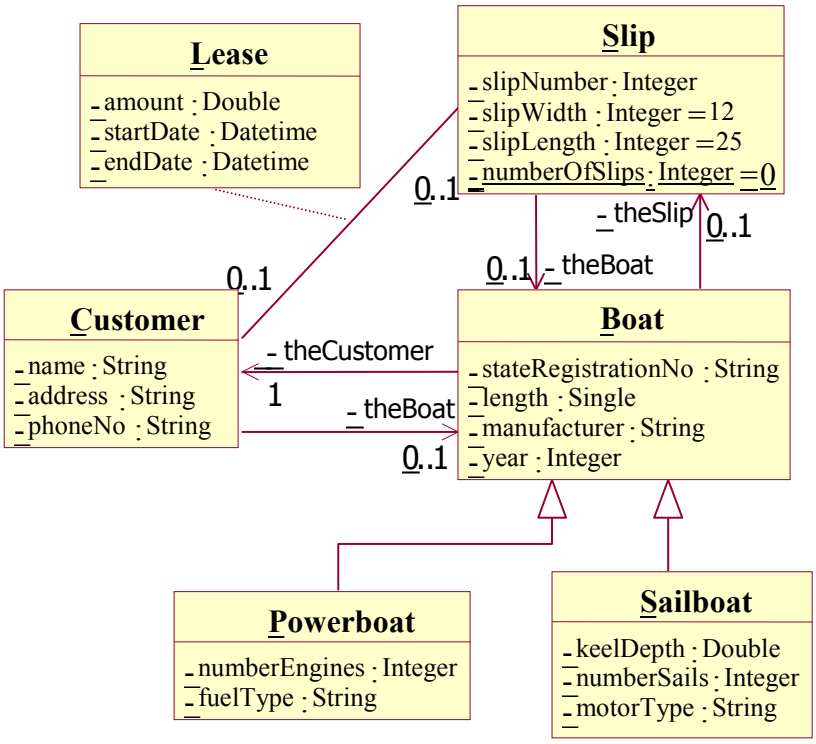

Figure 2: Sample class diagram (Bradshaw Marina partial model)

used to create a DDL script or to directly create the tables into a target database. In XDE .NET the classes in the diagram of Figure 2 would be packaged into a Logical Data Model. The classes from such a model can be transformed into corresponding tables of a Data Model associated to a target DBMS type like SQL Server, Oracle or Sybase. From this point on a simple forward engineering step would create either the DDL script or database schema itself.

The result of transforming the Bradshaw Marina Entity Model into a database schema is shown in the E/R Diagram of Figure 3. First thing to notice in the E/R Diagram is that primary key and foreign key columns have been added into the tables, e.g. the **Boat** table added the Boat_ID column as primary key and Customer_ID and Slip_ID columns as foreign keys referencing tables **Customer** and **Slip**. The association class **Lease** is converted to the **Lease** table representing the many to many relationship between **Customer** and **Slip**. The diagram also reflects some of the business rules mentioned above, e.g. each boat is owned by exactly one customer which is realized by enforcing a unique constraint on the Customer_ID foreign key in the **Boat** table.

3. THE MISSING LINK: DATA ACCESS LAYER

The Entity Classes mainly represent the business entities of an enterprise and are the part of the Object Model most likely to be saved in a persistent storage. The Entity Classes are related to a given enterprise and are more or less application independent. Most often one would organize Entity Classes into a separate package in the form of an Entity Model which is common to all applications related to an enterprise. Applications need to perform current functions related to object persistency of Entity Class instances like saving or retrieving an object. This functionality has to be provided in a uniform and storage independent way, while maintaining all the advantages of the object oriented approach. This means that the applications would refer to some completely generic and polymorphicaly available methods like "Save" or "Read" object, which provide the same interface for each possible Entity

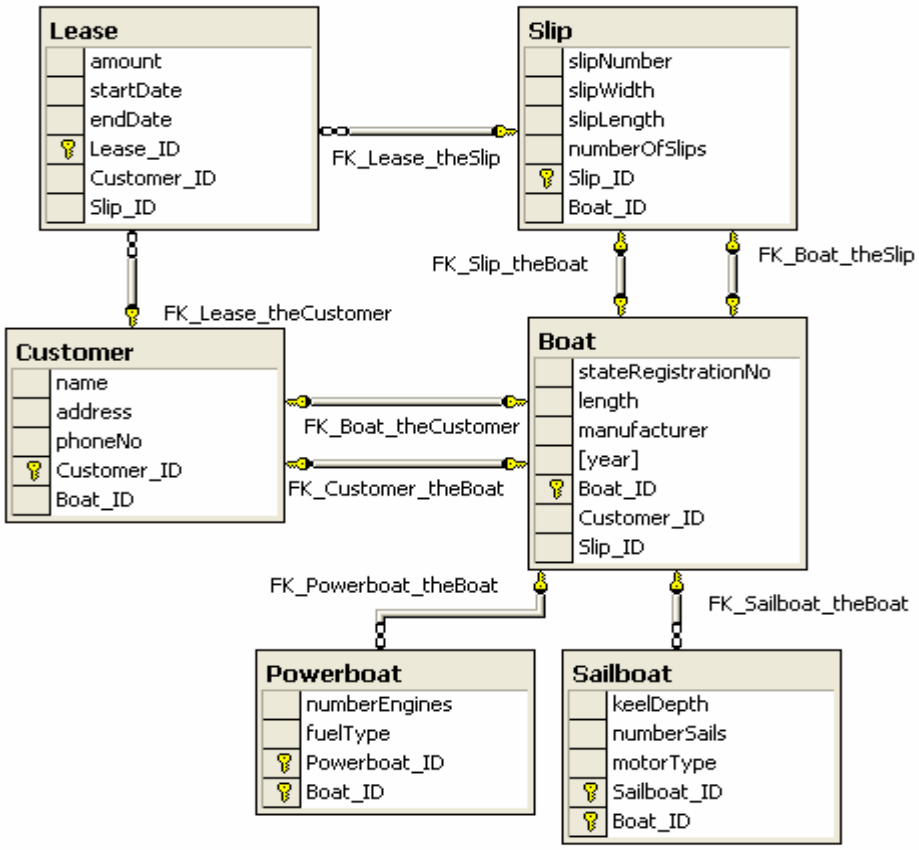

Figure 3: Database E/R Diagram Corresponding to the partial Bradshaw Marina Class Diagram

Class. For example, applications will invoke the "Save" method in the same way for each of the elements of a polymorphicaly processed array of mixed typed objects.

On the other hand, the Entity Classes need to be decoupled from the details of persistently storing or retrieving objects. It is the Data Access Layer through which the Entity Model gets decoupled from the specifics of the persistent storage. The traditional technique for achieving this is to interpose a layer of Data Access Classes where the specifics of persistently saving, retrieving or updating a particular type of object are dealt with. One would have a Data Access Class corresponding to each Entity Class.

One can identify two types of functionality the Data Access Classes need to implement. First, each Data Access Class has to deal with the specifics of the corresponding Entity Class; the second is to deal with the specifics of the persistent storage where the object is saved. For example, as part of the first kind of functionality, the Data Access Class would take care of building the particular SQL string such that the relevant fields of an object would be inserted in a database table. For another Entity Class the SQL string will be different and it is a different Data Access Class building it. Besides building the SQL strings, which is a task specific to each Entity Class, the Data Access Classes need to deal with database specific tasks as well, like connecting to the database server, opening/closing the database, and so on. This makes the Data Access Classes dependent on the specific kind of persistent storage used, which means that one would have to rewrite or have a different set of Data Access Classes for each type of persistent storage.

The approach described above results in a relatively simple and easy to understand structure, but it offers little flexibility in the organization of code which, at least in its database related part, repeats itself from application to application. Based on the idea of separating the two types of functionality

Data Access Classes are required to implement, we propose a two step approach towards refining and automating the creation of the Data Access Layer. The first step consists of moving out the storage specific functionality from the Data Access Classes into a (set of) separate classe(s) called Data Access Manager(s). One could have a Data Access Manager class for each type of persistent storage used by the enterprise's applications: MSSQL database, Oracle database or even flat files, or one single Data Access Manager Class serving all types of data storages. What matters is that these classes are completely unaware of the specifics of any Entity Class and provide exclusively storage specific functions like connecting to the database server, opening/closing the database or execute an SQL string given as parameter by the Data Access Class.

```
namespace ProblemDomain.EntityModel
{
      public class Customer
      {
            private string name;
            private string address;
            private string phoneNo;
            ...
            //saving a customer to the database
            public virtual void Save()
            {
                  DataAccess.CustomerDA.Save(this);
            }
      }
}
namespace DataAccess
{
      public class CustomerDA
      {
            //build SQL string to save customer
            //and call the Data Access Manager to execute it
            public void Save(Customer customerToSave)
            {
               string sqlString="INSERT Customer VALUES (";
               sqlString+="'"+customerToSave.name+"',";
               sqlString+="'"+customerToSave.address+"',";
               sqlString+="'"+customerToSave.phoneNo+"')";

            DAManagers.DAManager.currentDAManager.
                              ExecuteSQLString(sqlString);
            }
            //other methods here
            ...
      }
}
```

Figure 4: Implementation layout of the Customer and CustomerDA Classes

4. IMPLEMENTING THE ACCESS CLASSES AND THE DATA ACCESS MANAGERS

In this section we detail the first step of our approach and provide a possible C# implementation layout that allows decoupling of Entity Classes from the details of persistent storage functionality, as well as independence of Data Access Classes from the database server used as persistent storage. Figure 4 shows the layouts for the Customer Entity Class and its corresponding Data Access Class. Notice that all the Customer class has to do in its Save() method is to call the Save() of the CustomerDA giving itself as parameter. This later method constructs the SQL string for the specific INSERT statement of a Customer object into the database and calls the ExecuteSQLString() method from the currently active Data Access Manager. Note that no database specific code can be found in the CustomerDA class. Other methods for building the SQL strings needed for delete, update and retrieve operations of a customer would be included.

All database specific code is located in the DAManager class, which is illustrated in Figure 5. One would create an instance of this class for each concrete database to which an application connects.

The constructor of DAManager takes as parameters the type of database as a value of the ConnectionType enumeration, that is: ODBC, OleDB, Oracle or SQL which are the types supported by this implementation. The second parameter is the database connection string specifying the concrete database to which this DAManager will connect. An alternative overloaded constructor may take instead the individual elements of the connection string identifying the target database: database provider, server name, database name, user name, password, etc. In both versions, what the DAManager constructor does is to create the corresponding connection and command objects which would be subsequently used to execute the SQL statements. A switch-case statement controlled by the type of the database, given as the first parameter, determines the specific kind of connection and command objects to create.

The ExecuteSQLString() method is called by the Data Access Classes whenever a non-query operation is to be requested from the database. This method opens the database connection, sets the command object to execute the SQL string given as parameter, executes the command and closes the connection.

The last method shown in Figure 5, illustrates a remarkably useful feature of C# .NET, which is the key element for building database server independent Data Access Classes. Namely, .NET defines a set of interfaces containing a unique interface for each of the most important objects used in database connectivity operations: IDBConnection for connection objects, IDBCommand for command objects, IDataReader for data reader objects and so on. The GetReader() method in our implementation creates a data reader object for the SQL SELECT statement given as parameter and returns it to the Data Access Class. Actually, what this method generates can be any of ODBCDataReader, OleDBDataReader, OracleDataReader or SQLDataReader objects depending on the database type to which the current DAManager is connected. In all cases the data reader is returned as an IDataReader allowing for uniform treatment in the Data Access Class.

5. IMPLEMENTATION OF A GENERIC DATA ACCESS CLASS

As a result of the first refinement step of the Data Access Layer, application developers are relieved from the burden of dealing with the database specific functionality in each and every application. Once a Data Access Manager class is properly developed and tested, its functionality will be available for reuse in any application. All that needs to be done is to import the class into the current application. So far, as one can see in Figure 4, the object persistency specifics of each Entity Class are dealt with on a class by class basis. For example, there is a CustomerDA class associated to the Customer class and it incorporates specific code to generate the SQL strings for operations like saving, updating, deleting or retrieving customer objects.

The second step of our refinement process is to automate the creation of the functionality in the Data Access Classes that is specific to the Entity Types. The approach here has to be different from the one used in factoring out database functionality. One cannot have

a precompiled class that would deal with the specific properties of various Entity Classes. Instead, by using reflection and dynamic code generation techniques, it is possible to

```
namespace DAManagers
{
   public enum ConnectionType {Odbc,OleDb,Oracle,Sql}
   public class DAManager
   {
      public static DAManager currentDAManager;
      System.Data.IDbConnection connection;
      System.Data.IDbCommand command;
      System.Data.IDataReader dataReader;
      System.Data.IDbDataAdapter dataAdapter;

      public DAManager(ConnectionType connectionType,
                       string connectionString)
      {
        switch(this.connectionType){
            case ConnectionType.Odbc:
              {
               this.connection=
                        new OdbcConnection(connectionString);
               this.command=new OdbcCommand("",
                        (OdbcConnection)this.connection);
               break;
               }
               ...
            }
      }
      public void ExecuteSQLString(String SqlString)
      {
            this.connection.Open();
            this.command.CommandText=SqlString;
            try{
                  this.command.ExecuteNonQuery();
            }catch(Exception ex)
                  this.errorMessage=ex.Message;
            this.connection.Close();
      }
      public IDataReader GetReader(String SqlString)
      {
            this.connection.Open();
            this.command.CommandText=SqlString;
            return this.dataReader=
                              this.command.ExecuteReader();
      }
      ...
   }
}
```

Figure 5: Implementation layout of DAManager Class

automatically generate the code we need during program execution. With this second step one would not have to develop a Data Access Class for each Entity Class in the application. There will be one single generic class that would generate the code for saving, updating, deleting or retrieving any instance of an Entity Class given as parameter whenever such an operation is requested during application execution.

A sample layout of the code for this generic class, called here GenericDA, is given in Figure 6. Like in the case of the Data Access Manager class, the code for the GenericDA class is the same for all applications and can be reused by simply importing the class.

For simplicity, only the method for saving an object into the database is presented in the sample code from Figure 6, but this is just enough to illustrate the main ideas underlying our approach. The method gets as a parameter an object which is an instance of an Entity Class and generates an SQL statement of the form:

INSERT *table_name*(*list_of_columns*)
VALUES (*list_of_values*)

where *table_name*, *list_of_columns* and *list_of_values* are unknown at the time of compilation and have to be revealed during execution from the description of the Entity Class. The class name will give us the database table name, the Entity Class properties correspond to the column names and the values of these properties are saved in the columns with the same names. The key for the entire process is called reflection (see Liberty, J. 2003 and Troelsen, A. 2003), a very powerful feature of modern OO languages like C#, by which, given an object, one can identify, during runtime, the most important features of the class it instantiates, such as: class name, class fields, methods, properties and so on. All this is made available by setting a reference to the System.Reflection library of the .NET environment.

The Save(Object entityInstance) method, gets as a parameter the object to be saved in the database through the entityInstance variable and will save it in the database table with the same name as the class from which the object is instantiated. The class name is part of the class description returned as a Type object by the GetType() method (every .NET object comes with a GetType() method) and will be saved in the local variable called classType:

```
Type classType=entityInstance.GetType();
```

The name of the class (alias database table) is stored in the Name property of the classType object and will be inserted into the generated SQL string after the keyword "INSERT" and before the list of columns:

```
string sqlString="INSERT ";
sqlString+=classType.Name;
sqlString+="(";
```

The next line of code:

```
PropertyInfo[] pInfo=
                classType.GetProperties();
```

returns the list of get/set properties defined in the class under the form of an array of PropertyInfo objects. The Name properties of these objects give us the names of the get/set properties defined in our class, which allows building the list of corresponding table columns by inserting them one by one in the generated SQL statement.

After closing the list of property names (alias table columns) and adding the "VALUES" keyword, the last element to be added to the generated SQL statement are the values the get/set properties have in the particular object that was given as parameter. In order to properly build the list of property values, one needs both the value of the property, returned by the GetValue() method, as well as the data type of each property (integer, double, character or else), which is returned by the GetType() method. This is necessary because values get inserted in the SQL string in different ways for different types. In our sample code we differentiate only between character types, identified as sqlString.GetType(), and all other types. Values of character types have to be delimited by simple quotes, while numeric values and values of other types are inserted as such. More sophisticated processing to take into account other data types can be easily implemented.

As in the case of the CustomerDA class from Figure 4, the Save method ends by a call to the ExecuteSQLString() method of the currently active Data Access Manager which takes care of all the related tasks like: opening the database, executing the generated SQL string, producing an exception in case

of failure and properly closing the database when done.

```
using System;
using System.Reflection;

namespace DataAccess
{
   public class GenericDA
   {
      public void Save(Object entityInstance)
      {
            Type classType=entityInstance.GetType();
            string sqlString="INSERT ";
            sqlString+=classType.Name;
            sqlString+="(";
            //generate comma separated list of property names
            PropertyInfo[] pInfo=classType.GetProperties();
            int i;
            for(i=0;i<pInfo.Length-1;i++)
                  sqlString+=pInfo[i].Name+",";
            sqlString+=pInfo[i].Name+")";

            sqlString+=" VALUES (";
            //generate comma separated list of property values
            for(i=0;i<pInfo.Length-1;i++)
               if(pInfo[i].PropertyType==sqlString.GetType())
                  sqlString+="'"+pInfo[i].GetValue(entityInstance,
                                          new Object[]{})+"',";
               else
                   sqlString+=pInfo[i].GetValue(entityInstance,
                                          new Object[]{})+",";
            if(pInfo[i].PropertyType==sqlString.GetType())
               sqlString+="'"+pInfo[i].GetValue(entityInstance,
                                       new Object[]{})+"')";
            else
               sqlString+=pInfo[i].GetValue(entityInstance,
                                       new Object[]{})+")";
            DAManagers.DAManager.currentDAManager.
                                 ExecuteSQLString(sqlString);
            }
            //other methods here
            ...
      }
}
```

Figure 6: Implementation layout of the GenericDA Classes

6. BASIC ASSUMPTIONS, LIMITATIONS AND POSSIBLE REFINEMENTS

The approach presented above heavily relies on some assumptions in order to make code like the one in Figure 6 work properly. These assumptions can be summarized as follows:

- the Entity Classes and the corresponding database tables have exactly the same names;
- every table column referred in a generated SQL statement corresponds to the same name get/set property of the corresponding Entity Class;
- every relevant (that is one that needs to be stored persistently) get/set property of an Entity Class corresponds to a table column with the same name;
- corresponding get/set properties and table columns have compatible data types.
- all other table columns accept null values, unless part of a primary key.

The first four assumptions are satisfied by default when we use XDE .NET to either automatically convert a class diagram into a database schema or the other way around to produce a class diagram from an existing database as illustrated in section 2. This makes the entire procedure described in sections 4 and 5 fairly viable as long as the guidelines described in this work are followed. In the case where, for some reason, there is no one to one correspondence between class get/set properties and table columns one can still use an automated approach in a more elaborate and sophisticated way by using attributed programming. Through attributes one can specify all sorts of options like: which get/set property is going to be saved in the database, the name of the corresponding column, explicit data type conversion etc.

The last assumption requires that if there are columns in the database tables other then those corresponding to the get/set properties, they should allow null values if not part of a primary key.

As we have seen in section 2, primary and foreign key columns may be automatically added to tables. Single column primary keys will not pose a problem for the operations described in section 5. When saving an object, the primary key value would be typically generated by the database system as a sequence number. The primary key column can be easily ignored when retrieving the database row into an Entity Class instance and/or handled separately if needed. XDE .NET generates single key primary keys by default which means that our approach would work unless other database design options are considered.

The problem is a little bit more complicated in the case of foreign keys. One can imagine several workarounds for this problem, but there will be some specific coding in each of these. One simple approach would be to add a second parameter to the Save method in Figure 6 (and to all other methods in the GenericDA class far that matter). This second parameter would be a list of foreign key name-value pairs to be included in the generated SQL statement representing the links the object is saved has to other objects in the database. Of course, building this list would be the job of the application developer which is one of the limitations of the approach considered here. As a possible refinement and topic for further research the idea of automatically dealing with several related objects at once, thus controlling generation of primary and foreign key values seems to be appealing to the resolution of this problem.

7. THEORY AND PRACTICE

Obviously, the two step approach to the Data Access Layer automation we emphasized in this paper has some inherent limitations, which raises the question of how much its use can be generally applied in the current software development practice. More research will be needed in order to test and asses the viability of the proposed approach. Also, there are several alternative refinements to explore (e.g. using attributed programming), as well possible requirements and challenges that may yet to have to be discovered (e.g. dealing with transactions, storing very large objects like images, etc.). On the other hand the ideas captured in the two step approach seem to be productive by themselves and proved to be useful in practice, not only when developing applications, but also in the classroom. Separation of the database specifics into a specialized sub-

layer which is independent of the application specifics makes it easier to understand the role and functions of the Data Access Layer in general. The use of reflection as a basis for the automatic generation of Entity Class specific code is a generous one and deserves further exploration of its potential.

8. CONCLUSIONS

Current design tools offer powerful support in almost all stages of the application development process. However, there are still important issues remaining to be solved by proper code structuring. In this paper we illustrated a two step approach as a way to automate the development of the Data Access Layer. The first step consist of refining the Data Access Layer by factoring out database server specific elements into a layer of Data Access Manager classes by which effective decoupling between Data Access Classes and database servers is achieved. The second step is based on the dynamic generation of SQL code dealing with the specifics of Entity Classes, thus eliminating the need to develop specialized Data Access Classes, which are replaced by one single application independent generic class. The techniques presented in this work can be further refined in order to eliminate some limitations like those related to the proper handling of foreign keys. Several approaches can be emphasized which will be the subject of future research.

REFERENCES

Boggs, W., & Boggs M. (2003). Mastering Rational XDE, Alameda, CA: SYBEX Inc.

Doke, E.R. et al. (2003) Object Oriented Application Development Using Microsoft Visual Basic .Net, Thomson Course Technology.

Liberty, J. (2003) Programming C#, Third Edition, O'Reilly

Manassis E. (2004). Practical Software engineering, Analysis and Design for the .NET Platform, Addison-Wesley, The Addison-WesleyObject Technology Series.

Troelsen, A. (2003) C# and the .NET Platform, Second Edition, Apress.