OBJECT ORIENTED PLATFORM TO RDBMS STORED PROCEDURES

Óscar Narciso Mortágua Pereira
DET / IEETA – University of Aveiro, 3810-193 Aveiro, Portugal
oscar@ieeta.pt

Joaquim de Sousa Pinto
DET / IEETA – University of Aveiro, 3810-193 Aveiro, Portugal
jsp@ieeta.pt

António José Batel Anjo
Dep. of Mathmatics – University of Aveiro, 3810-193 Aveiro, Portugal
batel@ua.pt

ABSTRACT

Object-oriented programming (OOP) is the most successful paradigm used in programming environments. Some difficulties arise when it is necessary to deal with data stored in a relational database because relational databases do not provide an object-oriented interface to their entities. The most common solution consists in developing a specific interface that guarantees some specific requirements. Here, we explain a methodology to implement an application-side object-oriented platform in order to access stored procedures in relational databases.

KEYWORDS

database, object-oriented programming, interface, platform, stored procedures

1. INTRODUCTION

As Thomas Connolly said [CONN, pag 14], a database is "*a shared collection of logically related data (and a description of this data), designed to meet the information needs of an organization.*". Using a database as storage information component is of an unquestioned utility, providing not only the information storage mechanisms, but also, among others, security, integrity and maintenance mechanisms. The first approach to a database system was made through file-based systems. File-based systems were an attempt to computerize the manual filing system and revealed many problems based on data integrity, data consistency, fixed queries, etc. The appearance of the DBMS – Database Management Systems – overcome the file-based systems weaknesses by providing features as security, integrity, recovery, query flexibility, etc. Several models of DBMS have been developed since then, as the relational model and the object DBMS. The goal of the object DBMS it to close the gap between the relational DBMS and the object-oriented paradigm. A mixed DBMS, known as object relational DBMS, keeps the fundamental of the relational DBMS and incorporates some features of the object DBMS. Object relational DBMS promises to be the future of the DBMS.

This paper explains a methodology to implement an object-oriented interface to stored procedures in a relational DBMS. A stored procedure is a collection of code statements stored in a database server, compiled into a single execution plan. A stored procedure implements an interface used by the application for data interchange and it is made by four elements: stored procedure name, stored procedure parameters, the returned value and the returned selected data. We are focused on stored procedures related with database data interchange, through the execution of SQL commands such as Select, Insert, Delete and Update. In these cases, either the parameters or the returned selected data are usually mainly related with tables of the database: parameters refer to table columns and the selected data refers to tables and their columns. This paper presents a methodology to implement a client–side object-oriented interface to a relational DBMS, electing the stored procedures as the main access gate to the database. This methodology is very efficient for small and medium applications when it is not convenient to use one of the existing solutions in the market.

2. DESCRIPTION

In this chapter we will describe the methodology explaining not only some requirements that must be followed but also the necessary classes to implement the interface.

2.1 Tables

Tables are the main entities in a database. A table is a storage container for a single collection of information organized as rows and columns. Each column identifies one entity attribute and it is characterized, among other properties, by its name, its data type, etc. As one includes data in a table, the number of rows gets growing. In our methodology, table and column naming follows some rules in order to avoid future conflicts, namely when two or more columns from different tables share the same designation. Before that we must introduce the concept of derivative column as a column that references another column, known as parent column, in another table: an example of a derivative column is a foreign key. To avoid future naming conflicts, the following requirements may be implemented: 1) there cannot be two columns in different tables with the same name except 2) the derivative ones that must have the same name as their parent columns.

Our implementation follows the next steps:

- each table name has a *tb* prefix as in *tbUser*
- each column name has 2 components - the table name and its description, separated by an '_' as *tbUser.tbUser_username*.
- a derivative column name must be equal to the parent column as *tbProfile.tbProfile_id* (parent column) and *tbUser.tbProfile_id* (derivative column).

Any other solution may be applied since it guarantees the requirements. The solution here presented is the one we use in this paper.

2.2 Table mapping

An object-oriented platform to a relational DBMS must have some knowledge about each table description in order to deal with the information within it. To accomplish this goal, each database table is mapped into an object-oriented class, called the mapping class. A mapping class provides all the necessary information to describe its correspondent table such as table name, column names, columns data type, columns default value, etc.

2.3 Stored procedures

A stored procedure is a group of SQL statements that form a logical unit and perform a particular task. They are stored in the database server and are used to encapsulate a set of operations or queries to be executed in a database server. Some of the most important stored procedures features are: 1) they may be executed with different parameters and results, and they may have any combinations of input, output, and input/output parameters; 2) once compiled, they can be used over and over improving server performance; 3) they reduce network traffic; 4) they can help ensure database integrity, etc.

In our platform, stored procedures must match the following requirements: parameters matching and unique relation.

2.3.1 Parameters matching

Stored procedures parameters are used to send and receive information from the application. From the database point of view, parameters can be seen as free parameters when they are not strictly related to a column table or derivative parameters if they are strictly related to a column table. Free parameters may have any name and any data type; derivative parameters name and data type must match the correspondent column name and data type. This way, it is possible to benefit from the information available in the mapping classes, such as the parameters names, parameters data type and the parameters preparation for query execution.

2.3.2 Unique relation

An ordinary stored procedure may return none, one or more relations. In our methodology, in case it returns any relation, it must return only one relation and always with the same schema. The returned relation attributes may also be derivative or free. The derivative attributes must also match the correspondent column name and data type. The free attributes may have any name and data type. Through the relation schema it will be possible to implement an object-oriented interface to deal with the data included in the relation. In this case, the mapping classes will also play an important role, such as providing the attributes names.

2.4 Invocation classes

The object oriented interface is provided by a group of classes called the invocation classes. Each invocation class is associated with only one stored procedure. Invocation classes are responsible not only for providing the methods to call the stored procedures but also the necessary methods, when necessary, to deal with the relation returned by the stored procedure. The provided interface to deal with the returned relation is also object-oriented. The invocation classes provide 2 sort of public interfaces: one for calling the stored procedures and another for indexing the returned relation. The interface dedicated to call the stored procedures is designated as the access interface and the interface dedicated to indexing the returned relation is designated as the indexing interface. Figure 1 presents the invocation classes architecture and interfaces. The required steps are: 1) instantiation of the invocation class, 2) invocation of one of its access methods, 3) iteration of the returned relation through the indexing interface.

2.4.1 Access interface

The access interface is implemented through the access methods. Each access interface may have one or more access methods. Each access method structure is organized in 3 blocks: the first block, known as initialization block, is responsible, if necessary, for all the initialization necessary to establish a successful connection with the database; the second block, known as parameters block, is responsible for the setting of all the needed information for preparing the parameters passing to the stored procedure; the third block, known as execution block, is responsible for the query execution. The first and the third blocks are identical in all access methods in all invocations classes, and can be implemented in a super-class. This means that the specific code of an access method is reduced to the parameters block. Parameters blocks deal with information related to the stored procedure parameters and its implementation is also made easier through the mapping classes. Mapping classes have all the required information to the parameter passing except for their values which are defined in the application and passed to the access methods via their arguments.

2.4.2 Indexing interface

The indexing interface is only required when the access interface calls stored procedures that return a relation. The indexing interface is implemented through the indexing method, the count property and the class of attributes.

- The count property provides information about the number of records in the relation. This information is necessary to let the application control the iteration along the relation.
- The indexing method returns an instance of a class of attributes and has an argument that identifies the index of the record to be returned
- The class of attributes is a class that implements an interface equal to the returned relation schema. Its constructor receives a relation record as argument and provides each relation attribute name in its interface. The class of attributes processes each derivative attribute through the mapping classes, which play again an important role. If *rel* is a relation record, than *rel [tbUser.tbUser_username]* is the value of the attribute tbUser.tbUser_username.

2.4.3 Invocation classes architecture

Invocation classes behave as an interface between the application and the database stored procedures. The interface is implemented through two sub-interfaces: 1) access interface offers an object-oriented interface to the application to call the stored procedures; 2) the indexing interface offers an object-oriented interface to the application in order to process the data contained in the returned relation. Figure 1 shows a general invocation class architecture and their interfaces .

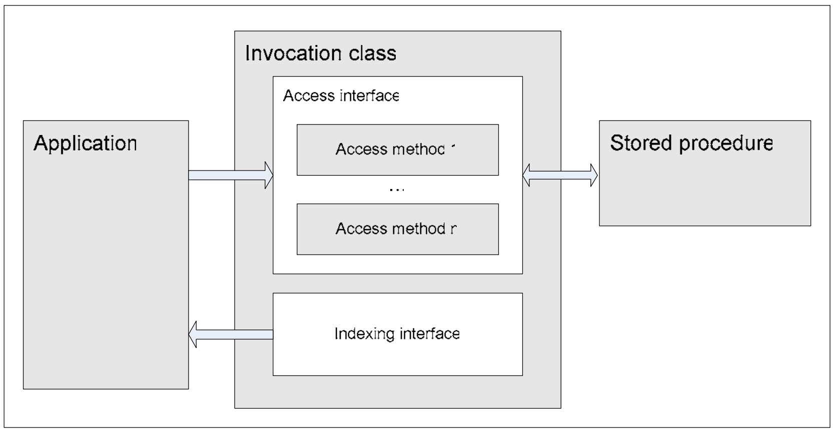

Figure 1. Invocation classes architecture and interfaces

3. EXAMPLE

The methodology is explained through a case, using Microsoft.Net technologies as C# and SQL Server, but it can be used with any other ordinary technology.

Consider two tables, one for user login identification called *tbUser* and other for user profiles definitions called *tbProfile*, as shown in Figure 2:

Figure 2 a) Tables and columns naming obeys to the announced requirements. A closer look to the table *tbUser* shows that it has 3 columns, one of which is a derivative column (tbProfile_id). Now let us suppose that the application specification requires two different criteria for data selection, both with the same final schema. The criteria selection are: 1) users by username and password and 2) all users by profile identification. These criteria selections may be implemented by the same stored procedure as shown in

Figure 2 b), because the final relation schema is the same in both cases. The stored procedure has 3 derivative parameters, 2 of them are from *tbUser* and the third comes from table *tbProfile*. Both the table and the stored procedure agree with the standard requirements.

a) tables	b) stored procedure

tbUser
    tbUser_username varchar(15)
    tbUser_password varchar(15)
    tbProfile_id int
 	
tbProfile
    tbProfile_id int,
    tbProfile_ref varchar(15)
	create procedure spUser
	    (@tbUser_username varchar(12) = ‘’,
	      @tbUser_password varchar(12) = ‘’,
	      @tbProfile_id int)
	as
	  if (@tbProfile_id = -1)
	    select * from tbUser u
	    where(u.tbUser_name = @tbUser_name) and
	        (u.tbUser_password = @tbUser_password)
	  else
	    select * from tbUser
	    where (u.tbProfileId = @tbProfile_id)
	return;

Figure 2. Example of two tables a) and a stored procedure b)

In the next step it is necessary to map the 2 tables. Figure 3 shows the table *tbUser* mapping class, called *mapTbUser*, where it presents not only the mapped columns names but also the preparation of the derivative parameters and the default values for their non-derivative columns. The default values for the derivative columns are defined in the mapping class associated to the table owning the parent columns. The mapping class for the table *tbProfile* is not presented but it follows exactly the same steps.

```
// columns names
public static string TableName {
    get { return “tbUser”; }
}
public static string Username {
    get { return TableName + “_username”; }
}
public static string Password {
    get { return TableName + “_password”; }
}
public static string ProfileId {
    get { return mapTbProfile.ProfileId; }
}
// derivative parameters preparing
public static void Prepare_username( SqlCommand cmd, string username ){
    cmd.Parameters.Add(new SqlParameter("@" + username, SqlDbType.VarChar, 15 ) );
    cmd.Parameters[ “@” + Username ].Value = username;
}
public static void Prepare_password( SqlCommand cmd, string password ){
    cmd.Parameters.Add(new SqlParameter("@" + password, SqlDbType.VarChar, 15 ) );
    cmd.Parameters[ “@” + Password ].Value = password;
}
// default values
public static string Default_username {
    get { return “”; }
}
Public static string Default_password {
    get { return “”; }
}
```

Figure 3. Table tbUser mapping class.

Now we may create the class of attributes for the relation returned by the stored procedure. Figure 4 presents the class main interface. The constructor receives as argument a row of the returned relation and keeps a reference to it. The remaining class interface implements all the attributes of the returned relation which are built by using class properties. Each class property returns the correspondent attribute current value by indexing the row through the mapping class. This is possible because: 1) the stored procedure returns only one relation always with the same schema and 2) the relation attributes are all derivative attributes.

```
public attUser( dataRow row ) {
    dr = row;
}
public string Username {
    get { return (string) dr[ mapTbUser.Username ]; }
}
public string Password {
    get { return (string) dr[ mapTbUser.Passord; }
}
public int ProfileId {
    get { return (int) dr[ mapTbProfile.ProfileId ]; }
}
```

Figure 4. Class of attributtes

The next and final step is to write the invocation class. The invocation class has 5 main blocks:

- the superclass whose interface is accessed by the base keyword [MS-BASE]
- the constructor with an argument of type SqlConnection [MS-CONN]
- the indexing method implemented by the C# indexer [MS-INDX]
- two access methods, one for each selection criterion
- a private method (param) which is responsible for parameter preparation for all the access methods

```
// constructor
public invocUser( SqlConnection conn ): base( conn ) {
}
// indexing method
public attUser this[ int index ]
{
    get {
        attUser AttUser = new attUser( table.Rows[ index ] );
        return AttUser;
    }
}
// access method 1
public void Select( string username, string password )
{
    base.initialization( “spUser” );
    param( command, username, password, mapTbProfile.Default_profileId );
    base.execute();
}
// access method 2
public void Select( int profileId )
{
    base.initialization( “spUser” );
    param( command, mapTbUser.Default_username, mapTbUser.Default_password, profileId );
    base.execute();
}
private void param( sqlCommand cmd, string username, string password, int profileId )
{
    mapTbUser.Prepare_username( cmd, username );
    mapTbUser.Prepare_userPassword( cmd, password );
    mapTbProfile.Prepare_profileId( cmd, profileId );
}
```

Figure 5. The invocation class - *invocUser*

All the invocation classes share the same structure and are easy to understand and implement. The access methods, as shown in Figure 5, always call the same 3 methods, 2 of them from the superclass.

```
private void getAllUsersByProfileId( int profileId )
{
    invocUser InvocUser = new invocUser( conn );
    InvocUser.Select( profileId );
    for ( int n = 0; n > InvocUser.count; n++ ) {
        attUser AttUser = InvocUser[ n ];
        username = AttUser.Username;
        password = AttUser.Password;
        profileId = AttUser.ProfileId;
        // … process user
    }
}
```

Figure 6. Example of using the interface

Now lets present the interface from the application side as shown in Figure 6. The application must:

- create an instance of the invocation class
- select an access method to call the stored procedure
- iterate through all the users through the indexer interface, which returns an instance of the attribute class
- access each relation attribute through the attribute class instance interface

In this example, the relation returned by the stored procedure only includes columns from the table *tbUser*. If necessary, the stored procedure could also return information related to the user profile as shown in Figure 7, joining the 2 tables.

```
create procedure spUser
       ( @tbUser_username   varchar(12) = ‘’,
         @tbUser_password    varchar(12) = ‘’,
         @tbProfile_id            int )
as
  if ( @tbProfile_id = -1 )
     select * from tbUser u, tbProfile p
     where( u.tbUser_name = @tbUser_name ) and
            ( u.tbUser_password = @tbUser_password ) and
            ( u.tbProfile_id = p.tbProfile_id )
  else
     select * from tbUser u, tbProfile p
     where ( u.tbProfile_id = @tbProfile_id ) and
             ( u.tbProfile_id = p.tbProfile_id )
return;
```

Figure 7. Extended stored procedure

```
public attUser( dataRow row ) {
    dr = row;
}
public string Username {
    get { return (string) dr[ mapTbUser.Username ]; }
}
public string Password {
    get { return (string) dr[ mapTbUser.Passord; }
}
public int ProfileId {
    get { return (int) dr[ mapTbProfile.ProfileId ]; }
}
// new property
public string ProfileRef{
   get { return (string) dr[ mapTbProfile.ProfileRef ]; }
}
```

Figure 8. Extended attribute class

In order to accomplish the new requirements some changes must be made. These changes are only required in the attribute class, which must reflect the new schema of the returned relation. This new relation includes only one new attribute which is the *tbProfile_ref*. Figure 8 shows the new version for the extended attribute class.

4. CONCLUSION

This platform implements an object-oriented interface to database stored procedures. In spite of being an object-oriented interface to a database, some rules must be followed both in database design and in the stored procedure design. From the interface design point of view, 3 classes must be built: mapping classes, the attributes classes and the invocation classes. This platform presents the following advantages:

- writing a new interface for a stored procedure may seem a lot of work, but the experience has shown that after having built some of the mapping, invocation and attributes classes, the work can be sharply reduced if certain procedures are followed; from the example shown, one can see that within each type (mapping / invocation / attributes) the classes are very similar, and one can take advantage of the existing classes to build the new ones;
- most of the invocation class code is defined in a superclass;
- defining a new access method in an existing invocation class, takes no more than 2 minutes to be accomplished; in most cases it is enough to create the method with the 3 known methods inside of it: initialization, param and execute.
- some changes in the database may have impact on the platform interface; the changes may occur by modifying column names, column data type, or by defining new columns, etc; the changes to

be made in the platform are always easily localized and can be executed in a very short period; as an example, changing a column name always implies changes in the same classes and places.

This platform presents the following disadvantages:

- it has some restrictions at the table and column naming level
- it has some restrictions at the stored procedure level
- some additional classes are necessary to map the non-derivative parameters and attributes

Future work will be focused not only in the disadvantages presented but also in the implementation of automatic code generation, namely for the mapping classes.

REFERENCES

[CONN] Connolly, Thomas et al, 1999. *Database Systems, A Pratical Approach to Design, Implementaion and Management*. Addison Wesley Limited, Essex, England. ISBN: 0-201-34287-1.

[MS-BASE] Microsoft, (2004), “Inheritance fron a Base Class in Microsoft.NET”. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/classinherit.asp. [accessed on 27/09/2004].

[MS-CONN] Microsoft, (2004), “.Net Framework Class Library – SqlConnection Class”. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemDataSqlClientSqlConnectionClassTopic.asp [acessed on 27/09/2004].

[MS-INDX] Microsoft, (2004), “C# Programmers Reference – Indexer Declaration”. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vclrfindexedpropertiespg.asp [accessed on 27/09/2004].