An Approach to Change Management of Evolving Database Schemas

Mark Grechanik[a], Dewayne Perry[b], and Don Batory[a]
[a]Dept. of Computer Sciences
[b]Department of Electrical and Computer Engineering
UT Center for Advanced Research In Software Engineering (UT ARISE)
University of Texas at Austin
Austin, Texas 78712
{gmark|batory}@cs.utexas.edu,perry@ece.utexas.edu

Abstract. In client/server architectures it is common for software components to have SQL statements embedded in component's source language. The components submit queries to relational databases using such standards as Universal Data Access (UDA) and Open Database Connectivity (ODBC). Such programming practices are widespread and result in poor maintenance and reuse of the components.

We propose an approach that completely eliminates the mix of programming languages and SQL thereby improving software component reuse and automating important architecture software maintenance tasks.

1 Introduction

Client/server architectures are pervasive in today's computing world. The invariants of these architectures are independent of the number of tiers and include clients that may have a front end Graphic User Interface (GUI) and a backend database. The client module interacts with the database to retrieve information for further processing or to modify it in the database. When developers need to interact with these databases they write SQL queries. The queries return tuples that need to be further processed by the client software. In the programmer's view the client software makes the database calls so that it is a natural choice to embed SQL queries into the source code of the client module.

Almost every database system currently offers some kind of embedded SQL API. The generic API consists of four conceptual groups. The first group contains initialization calls: open a database connection, initialize some internal variables and structures, and set properties of the database connectivity. The second group includes calls that accept SQL statements as parameters, do certain preprocessing, and submit the statements to the database engine for execution. The third group deals with processing of the data returned as a result of the execution of the submitted query. Since the database data types and source language types are different the conversion is required between the types. Certain API calls provided bindings between the variables declared in the source code and the returned results. Finally, the fourth group of API calls deals with memory and connection cleanup.

Commonly the client/server architecture is implemented in heterogeneous environment: the backend database runs on a UNIX-flavor server and client software runs on a cheaper PC-compatible computer running a version of Microsoft Windows. Windows has an open standard called Open Database Connectivity that is a part of Universal Data Access standard. Database vendors supply drivers for Windows that conform to ODBC standard. In the first half of 1990s developers wrote client software that used ODBC API directly. However, this approach was hindered by a multiplicity of various problems. The ODBC API is a part of Windows API and carries its problems. The API is complex and hard to master and use [Spin98]. It decreases the safety, robustness, and portability of the applications developed under it. Four levels of con-

formity made it difficult for applications to switch between databases as additional code should be present that verified the conformity level of the ODBC driver supplied by a database vendor.

In the second half of the 1990s and up to this very moment multiple vendors offer their middleware that is ported for major operating systems, databases, and languages. In fact, the database middleware is a virtual machine that hides the peculiarities of ODBC implementations and database driver dependencies. It offers uniform semantics for major development languages and operating systems. A developer does not need to worry about SQL conversion between different databases or data bindings. The positive impact of this approach results in the increased developer productivity and improved software maintenance. Interoperability increased significantly as there is no need to change the source code when a new database is plugged into the architecture. Therefore the trend of introducing database connectivity virtual machine was positive and results in an overall improvement of software processes.

2 The Problem

Let us consider the following example. A developer needs to retrieve the first and last name from a User table located in a database. To do this one writes the following SQL statement: “SELECT fname, lname FROM User” where fname stands for an attribute of User table that keeps first names and lnames stands for an attribute of User table that keeps last names. First, the statement is tried as a stored procedure or entered into the database query executor to verify that it is correct. Next, the developer embeds the statement into an API call from the previously mentioned third group that delivers this statement for execution to the database. Then one adds error checking and writes code that further processes the returned results. Other developers write different modules that embed SQL statements dealing with table User. Everything looks great until a database analyst decides to change the database schema. For example, attribute lname is changed to last_name. Such a small change leads to large consequences. Depending on the level of error checking the software acts as the problem amplifier. It may break right at the place where the API call returns the result code, or it may carry the problem from module to module until it manifests itself in some unexpected place. This problem is compounded by the level of distribution of the embedded SQL statements in the source code and variety of APIs used to embed the statements. While database schema changes are very easy to make, the implications of these changes are very difficult to determine. It requires significant domain knowledge and source code expertise. If software modules are tightly coupled then the regression testing of the whole system must be performed.

This problem is partially addressed by the creation wrappers for embedded SQL statements. For example, the Microsoft Foundation Classes library (MFC) and various publications offer object-oriented semantics for embedded SQL. An example of such semantics is shown in the C++ code below.

…

```
string strFirstName, strLastName;
SQLWrapper Sql;

Sql.Type( CSQLWrapper::SELECT );
Sql.Table( “User” );
Sql.AddAttribute( “fname” );
Sql.AddAttribute( “lname” );
Sql.Execute();

while( Sql.GetRow() )
```

```
{
strFirstName = Sql.GetAttribute( “fname” );
strLastName = Sql.GetAttribute( “lname” );
...
}

…
```

SQLWrapper is a wrapper class that embeds methods that build SQL statements dynamically. Sql is an instance of this class. Example methods that are semantically linked to the SQL grammar take table and attribute names as parameters. The Execute() method builds SQL statement like “SELECT fname, lname FROM User” and sends it to the database to execute.

This idea is a very weak abstraction over the existing SQL syntax. It hides the actual semantics of SQL and the embedded API while not resolving the problem of schema changes. Object-oriented design helps very little because it is not intended to resolve problems linked to encapsulating one language into the other.

3 The Solution

3.1 What is a Stored Procedure

Stored procedures are database objects that consists of SQL statements and some fourth-generation language statements designed to work with SQL. One may think of a stored procedure as a function or subroutine in a high-level language. There are no programming limitations to using stored procedures. Since they reside inside the database they may be precompiled and optimized before they are used. This makes them much faster than other ways of executing SQL queries at the database server. Most importantly, stored procedured can be viewed as consolidation nodes for database related code. Once created a stored procedure can be reused by many applications therefore improving the reuse metrics. The ability of a stored procedure to take parameters and return results in various forms as well as to use a high-order languages provided by specific database implementations, makes it very attractive for use with distributed applications to decrease their complexity.

Despite all the benefits, many companies experience problems with stored procedures similar to those of support and maintenance of software in general. There are a number of books written that outline the guidelines for creation and maintenance of stored procedures [http://www.acs.ncsu.edu/Sybase/NCSU/sp_guidelines/]. Such guidelines do not go farther than outlining rules for documenting stored procedures. For example, it is recommended that a developer creates a maintenance wrapper that is a documentation header containing the names of tables and attributes used in a stored procedure. If the database schema changes later, then someone needs to go through all procedures and analyze the maintenance wrappers with the intent of correcting the code to reflect the schema changes.

3.2 Our Approach

Our solution removes the embedded SQL statements from the source code ans is based on the ability to receive the information from the database server about its objects. Recall that a developer tests the SQL statements initially in a stored procedure (SP). After testing the SQL code the developer leaves the stored procedure inside the database system. At this point it becomes a database object linked to other objects inside the database server. Let us introduce an operator called WG that stands for Wrapper Generator. When applied to any database object O_i, WG produces the following tuple defined recursively

<<O_j, Type(O_j)>, <WG(O_j), Type(O_j)>, <IN_k, Type(IN_k)>, <OUT_n, Type(OUT_n)>> = WG(O_i),

where O_j is a set of other database objects, for example, tables that O_i uses, Type() is an operator that return the database specific type of an object, IN_k is a set of input parameters that O_i may require, and OUT_n is a set of results that O_i may return. For example, let us consider the following simple procedure using T-SQL syntax.

```
CREATE PROCEDURE get_user_name_by_id @UserID varchar(10), @User_First_Name varchar(20) OUTPUT, @User_Last_Name varchar(50) OUTPUT
AS
SELECT @User_First_Name, @User_Last_Name FROM User WHERE UserId = @UserID
```

When applying WG to *get_user_name_by_id* object we get the following result

<<User, table>, <User.fname, varchar(20)>, <User.lname, varchar(50)>, <User.UserId, varchar(10)>, <UserID, varchar(10)>, <User_First_Name, varchar(20)>, <User_Last_Name, varchar(50)>>

Based on this information a rooted dependency graph can be created. The root node of this graph describes the parameter object to the top level WG operator. The graph edges describe the dependencies between the database objects.

Once created, such graph can be analyzed to determine all dependencies. Later, type conversion information can be added to map the database specific types to the programming language types from which this procedure may be called. At this point we have enough information to generate a class wrapper for an external language in which the client software is written. An example of a class wrapper definition in C++ for *get_user_name_by_id* is shown below.

```
class SP_Wrapper_get_user_name_by_id
{
//constructor and destructor
public:
SP_Wrapper_get_user_name_by_id( void );
~SP_Wrapper_get_user_name_by_id( void );

//operations
void Set_UserId( string strUserId );
string Get_User_First_Name( void );
string Get_User_Last_Name( void );
virtual void Execute( void );

//attributes
protected:
string User_First_Name, User_Last_Name;
string Database_Object_Name;
};
```

A developer can use this class without having a single SQL statement embedded in the source code. This class can be reused multiple times without retesting since the database object does not change. Let us consider a situation when a database schema is changed.

When traversing the graph, each node is validated against the current database schema. For example, attribute fname in table User is changed to first_name. Then the validation fails when checking fname node against the database schema. At this point a developer is notified that WG needs to be reapplied to build or change the wrapper class. Once the developer specifies the mapping between the original and changed names the task of automated maintenance will be done without the developer knowing the details of the conversion.

3.3 Architecture

We propose an architecture that represents how our approach works. A block schema of the architectural

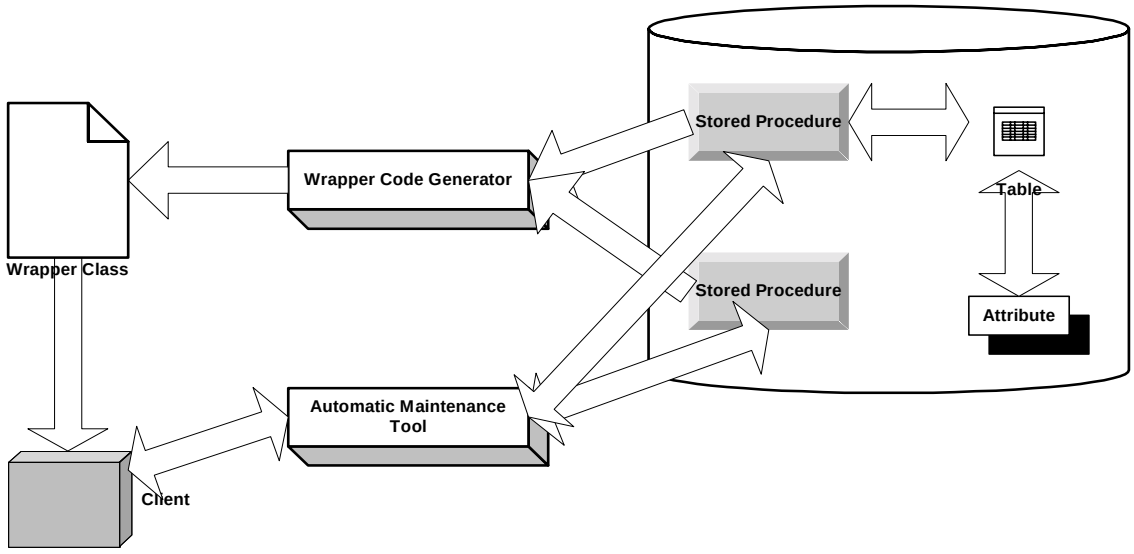

Figure 1: The Architecture of Stored Procedure Wrapper Generator and Maintenance System

solution is shown in Figure 1. A database is shown in the right half of the figure that contains stored procedures and other schema elements. A relationship between a stored procedure and some table is shown with the arrow connecting them. The wrapper code generator reads and analyzes a stored procedure and generates a class wrapper in a common object-oriented language. The class wrapper becomes an integral part of the Client. All accesses to the stored procedures from other classes in the Client occur via the wrapper class objects. The automatic maintenance tool does reverse engineering of the client software and locates all references to the wrapper class. It keeps all references in a local registry. When a database analyst needs to change the database schema the maintenance tool is invoked that presents all database schema elements. By clicking on an element the analyst causes the tool to retrieve the dependency graphs that show what database elements are affected by the change. The tool then searches the registry to determine whether any of the affected elements are stored prcedures that are used in the client software via the generated wrapper. It then presents the report about all required changes and their cost.

4 Conclusion and Future Work

Stored procedures occupy a very important niche in today's database world. Besides purely database related aspects they also play an important role in overall software engineering. We have shown how the use of stored procedures reduces the complexity of the software, improves software reuse, and simplifies change management.

We believe that once stored procedures are recognized not just as database objects but as important components of the overall software design it will lead to introducing new software tools that simplify software development and maintenance. To show the usefulness of our approach, we plan to develop the Wrapper Code Generator and Automatic Maintenance Tool and apply them to the development of multitier software systems that extensively uses databases. We also plan experiments that measure the quantitative characteristics of software metrics and improvement of overall software design and maintenance.

5 References

[Sar00] C. Saracco. "Leveraging DBMS Stored Procedures Through Enterprise JavaBeans", *IBM Report*, San Jose, CA, August 2000.

[Risch97]N. Rische, A. Vaschillo, D. Vasilevsky, A. Shaposhnikov, S. Chen. "The Architecture for Semantic Data Access to Heterogeneous Information Sources". *1st Int. Workshop on Cooperative Information Agents*, 1997

[Spin98] D. Spinellis. "A Critique of the Windows Application Programming Interface". *Computer Standards & Interfaces*, 20:1-8, November 1998

[Wells] G. Wells. "Code-Centric: T-SQL Programming with Stored Procedures and Triggers". *Apress/Springer-Verlag* , 2001

[sql00] D. Sunderic and T. Woodhead. "SQL Server 2000 Stored Procedure Programming". *Osborne* , 2001

[wood] C. Wood. "OLE DB and ODBC". *M&T Books*, 1999