An Approach to Evolving Database Dependent Systems (Extended Abstract)

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 AustinAustin, Texas 78712

{gmark|batory}@cs.utexas.edu,perry@ece.utexas.edu

Abstract. It is common in client/server architectures for components to have SQL statements embedded in their source code. Components submit queries to relational databases using such standards as *Universal Data Access (UDA)* and *Open Database Connectivity (ODBC)*. The API that implements these standards is complex and requires the embedding of SQL statements in the language that is used to write the components. Such programming practices are widespread and result in increased complexity in maintaining systems.

We propose an approach that eliminates the embedding of SQL in programming languages, thereby enabling the automation of important software maintenance tasks.

1 Introduction

Client/server architectures are pervasive in today's computing world. Common to these architectures are multiple tiers, *Graphic User Interfaces (GUIs)*, and a backend database. Developers interact with databases by writing SQL queries. Queries return tuples that need to be further processed by the client software. In the programmer's view the client software makes database calls so that it is a natural choice to embed SQL queries into the source code of client modules.

The idea of embedding SQL code into software to allow it to communicate with databases was introduced in 1980s. Databases did not provide internal storage for SQL queries that could be invoked by external programs. Embedded SQL was introduced as database-specific extension to high-level languages like C/C++ in order to allow programs to query and manipulate the content of databases. Although this method has been used for some time, we will show that it creates many problems with type checking that leads to inefficient and slow evolution of database schemas and software that uses the databases.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IWPSE 2002 Orlando Florida

Copyright ACM 2002 1-58113-545 -9/02/05...$5.00

Client/server architecture is commonly 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 (ODBC)* that is a part of Universal Data Access standard [4][6]. Database vendors supply drivers for Windows that conform to ODBC. In the early 1990s, developers wrote client software that used ODBC API directly. However, this approach was hindered by a multiplicity of problems. The ODBC API is complex and hard to master and use [3]. It decreases the safety, robustness, and portability of applications developed using it. Four levels of conformity 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 database vendors.

Today multiple vendors offer their middleware via major operating systems, databases, and languages [2]. In fact, database middleware is a virtual machine that hides the peculiarities of ODBC implementations and database driver dependencies. The trend of introducing database connectivity virtual machine was positive and results in an overall improvement of software processes.

2 The Problem

Suppose 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 `lname` for an attribute 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. 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 `User`. The problem appears when 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. Because SQL com-

mands are submitted as *strings*, these commands cannot be statically type-checked with the database schema; errors will only be discovered at run-time. And of course, these errors will be discovered at an in-opportune time. They might arise 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 (MFC)* library 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 [1]. 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 for execution. Because all of this is done at program run-time, it is very difficult to assess the impact of changes made to database schemas in client software.

3 A Solution

3.1 Stored Procedures

Stored procedures are database objects that consist of SQL statements and some fourth-generation language statements designed to work with SQL [5]. 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 procedures can be viewed as consolidation points for database related code. 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 their creation and maintenance [`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 and 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

$$WG(O_i) = \langle\langle O_j, Type(O_j)\rangle, \langle WG(O_j), Type(O_j)\rangle, \langle IN_k, Type(IN_k)\rangle, \langle OUT_n, Type(OUT_n)\rangle\rangle,$$

where O_j is a set of database objects, for example, tables that O_i uses, Type() is an operator that returns 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, 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 obtain 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)>>
```

A rooted dependency graph can be created given this information. The graph edges describe the dependencies between the database objects. Once created, such graph can be analyzed to determine 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 his/her source code. This class can be reused multiple times without retesting since the database object does not change. Now consider a situation when a database schema is changed.

When traversing a dependency graph, each node is validated against the current database schema. Suppose attribute `fname` in table `User` is changed to `first_name`. Validation fails when checking `fname` 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. *Client source code is unaffected by these changes.*

3.3 Architecture

A block schema of the architectural 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. 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 a change. The tool then searches the registry to determine whether any of the affected elements are stored procedures 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 simultaneously reduces software complexity and simplifies change management.

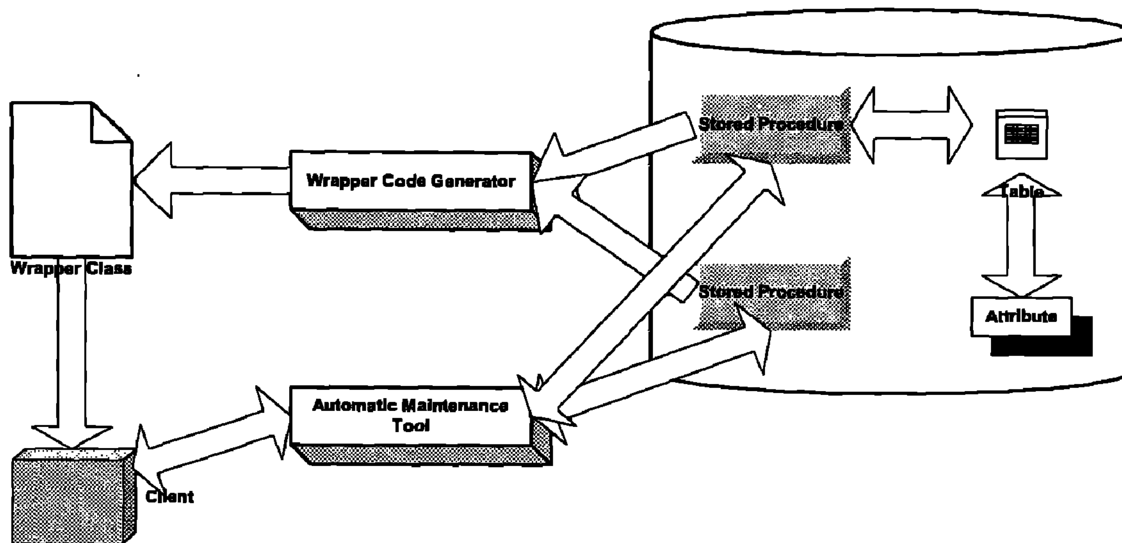

Figure 1: The Architecture of Stored Procedure Wrapper Generator and Maintenance System

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 a Wrapper Code Generator and Automatic Maintenance Tool and apply it to the development of multi-tier 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

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

[2] 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.

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

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

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

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