

# SQL DOM: Compile Time Checking of Dynamic SQL Statements

Russell A. McClure and Ingolf H. Krüger  
University of California, San Diego  
Department of Computer Science and Engineering  
9500 Gilman Drive, La Jolla, CA 92093-0114, USA  
{rmcclure, ikrueger}@cs.ucsd.edu

## ABSTRACT

Most object oriented applications that involve persistent data interact with a relational database. The most common interaction mechanism is a call level interface (CLI) such as ODBC or JDBC. While there are many advantages to using a CLI – expressive power and performance being two of the most key – there are also drawbacks. Applications communicate through a CLI by constructing strings that contain SQL statements. These SQL statements are only checked for correctness at runtime, tend to be fragile and are vulnerable to SQL injection attacks. To solve these and other problems, we present the SQL DOM: a set of classes that are strongly-typed to a database schema. Instead of string manipulation, these classes are used to generate SQL statements. We show how to extract the SQL DOM automatically from an existing database schema, demonstrate its applicability to solve the mentioned problems, and evaluate its performance.

## Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering.*

D.2.11 [Software Engineering]: Software Architectures – *Data abstraction.*

## General Terms

Algorithms, Reliability, Security.

## Keywords

SQL, SQL DOM, Impedance Mismatch, SQL Strings, SQL Injection, Dynamic SQL

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.  
Copyright 2005 ACM 1-58113-963-2/05/0005...\$5.00.

## 1. INTRODUCTION

The term impedance mismatch [17] refers to the inherent disconnect between databases and programming languages. Many solutions have been proposed, including language extensions [4, 11, 16], call level interfaces [2, 13, 21], object/relational mapping [10, 15, 18] and persistent object systems [3, 7, 20].

Call level interfaces offer the full expressive power of SQL by providing an interface that accepts dynamically generated SQL statements, but they provide no static syntax or type checking. Language extensions, such as SQLJ[4], do provide some static syntax and type checking but they are limited to static SQL statements. Object/relational mapping and persistent object systems allow stored data to be treated as objects but they do not expose the full power of the database engine.

In this paper, we focus on overcoming the problems associated with accessing relational databases through call level interfaces from object oriented programming languages.

### 1.1 Problem Definition

Call level interfaces (CLIs) are powerful because they provide a low level interface to the database engine. Interfacing with a database engine through a CLI involves constructing SQL statements through string concatenation and substitution. This allows the developer to create very flexible and powerful queries. However, the resulting queries cannot be checked for correctness until they are sent to the database engine at runtime.

There are many types of problems and errors that can arise when constructing SQL statements in this way. Some of the more common problems are bad syntax, misspelled column and table names and data type mismatches. SQL strings (strings that contain SQL statements) are also very fragile with respect to changes to the database schema. In medium to large sized projects, the number of SQL strings can become quite large. As an application evolves and the database schema changes, it becomes a difficult task to maintain the SQL strings that are in the code base. SQL strings also pose a security risk. They leave an application extremely vulnerable to SQL injection attacks [14]; this type of attack is typically based on malicious code inserted into a web form, and then – as part of the processing of the form – into a dynamically generated SQL string. The resulting query leads to execution of the malicious code, resulting in adding/removing data from the database.

## 1.2 Solution Proposal

Our goal is to provide a way to have the full expressive power of dynamic SQL statements, without the inherent problems mentioned in the previous section. The end result is increased maintainability, reliability and security for the application.

Our solution consists of an executable, *sqldomgen*, which is executed against a database. The output generated by *sqldomgen* is a Dynamic Link Library (DLL) containing classes that are strongly-typed to a database schema. We refer to these classes as the SQL Domain Object Model (SQL DOM). Using these classes, the application developer is able to construct dynamic SQL statements without manipulating any strings.

Using this method, the compiler is enlisted to eliminate the possibility of SQL syntax, misspelling and data type mismatch problems. Names of tables and columns are incorporated into the SQL DOM through class names or enumeration members. Data types of columns become types of constructor and method parameters. Having the compiler catching bugs that used to show up only occasionally at runtime increases the reliability of the application.

As the database schema changes throughout the life of an application, the support of the compiler also increases the maintainability of the application. If the database schema changes, *sqldomgen* is rerun, generating a modified SQL DOM. When the application is then rebuilt with the modified SQL DOM, compiler errors such as the following would be generated:

- *No such class exists.* This would happen if a table or column was renamed or removed.
- *Data type conversion error.* This would happen if the data type of a column was changed.
- *Missing constructor parameter.* This would happen if a new column was added to a table.

The security of an application using the SQL DOM is increased because all known SQL injection attacks are eliminated (this currently only applies to our test database engine, SQL Server 2000). In an application that uses SQL strings, defending against SQL injection attacks is very difficult. There is no single place where strings are constructed. There is usually no single place through which all database-bound user input passes. The SQL DOM represents a single point of defense. All SQL statements are constructed by the SQL DOM. All database-bound user input passes through constructors of classes in the SQL DOM. These constructors perform the necessary escaping and data type validation to eliminate the threat of SQL injection.

## 1.3 Applications

We believe that any application that is currently using SQL strings could benefit from the SQL DOM. However, there are a few areas we feel would benefit more than others.

Applications being developed with the extreme programming (XP) methodology tend to evolve rapidly through the development process. Iterative user feedback results in iterative changes to the application and database schema. These changes can lead to a large amount of time being spent keeping existing SQL strings consistent with the changing database schema.

Without compiler support the developer can never be sure that the code and database schema remain consistent. Unit tests, even if they exist, are also rarely complete enough to uncover all deviations between code and schema. Using the SQL DOM would eliminate this problem. The compiler would be able to assist in the maintenance process, thereby increasing the reliability of the application. Having the SQL DOM, developers would be more willing to make changes to the database schema to meet the needs of their customers.

Applications that need to make it to market quickly also stand to gain from using the SQL DOM. The SQL DOM eliminates all syntax and data type mismatch bugs, which can easily slip into applications that use SQL strings. The SQL DOM also frees the developer from having to perform many unit tests on data access code. This in turn allows getting the product to market faster.

The SQL DOM can also be used to enforce SQL coding standards. For example, you may want to only allow tables to be joined by a foreign key relationship. Or, only allow filter conditions to be placed on indexed columns. The SQL DOM can be tailored to enforce these standards.

## 2. THE EXISTING METHOD

In this section we will demonstrate by example some of the problems with dynamic SQL strings. In the following section we will show how our SQL DOM alleviates these problems. To illustrate the existing methods and our proposed solution we will use a simple database with four typical tables as shown in Figure 1. The tables store information about customers, products, orders, and order details.

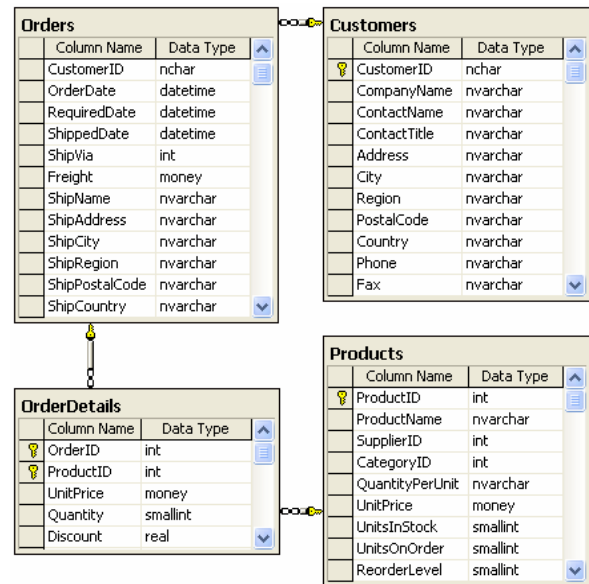


Figure 1. A sample database schema

A common problem that is introduced by using SQL strings is misspelled table and column names as shown in Figure 2. The compiler cannot offer us any aid in catching these kinds of errors. Code reviews would be of limited help because of the difficulty in manually finding misspelled words.

```

16 public string GetAllCustomers()
17 {
18     return "SELECT CustomerID, CompnyName " +
19         "FROM Customrs";
20 }

```

**Figure 2. SQL string with misspelled names**

Misspellings such as this should be caught during unit testing and the example we have shown most likely would be caught at that time. However, SQL strings are often not that simple, as we will see in the next example.

In Figure 3 we see a common example of a function that uses string concatenation to dynamically filter the result set of a select statement. Conditions are added to the where clause of the select statement for each of the function parameters that the caller specifies.

This example also contains a misspelled column name on line 46. In addition, if line 44 is executed, there will be no space between the AND keyword and the column name resulting in an invalid SQL statement. These errors are more difficult to find during unit testing than the errors in the previous example. Calling the function is not enough. Finding these errors depends on particular values being passed in to certain parameters. 100% statement coverage would be required to find these kinds of errors. That may be easy to accomplish with only few tables in the database schema. But if the project contains hundreds of SQL strings to check, 100% statement coverage may be prohibitively time consuming.

```

22 public string GetCustomers( string companyName,
23                             string contactName,
24                             string city,
25                             string region,
26                             string country )
27 {
28     bool firstCondition = true;
29     StringBuilder sql =
30         new StringBuilder("SELECT * FROM Customers ");
31
32     // add a where condition for CompanyName
33     // if the user specified a CompanyName
34     if ((companyName != null)
35         && (companyName.Length > 0))
36     {
37         if (firstCondition)
38         {
39             firstCondition = false;
40             sql.Append(" WHERE ");
41         }
42         else
43         {
44             sql.Append(" AND");
45         }
46         sql.Append("CompnyName = '");
47         sql.Append(companyName);
48         sql.Append("'");
49     }
50
51     // similar code would be placed here for each
52     // of the other five possible conditions
53
54     return sql.ToString();
55 }

```

**Figure 3. SQL string with syntax errors**

Another type of error that can occur in SQL strings is a data type mismatch. This occurs when a data type that developers are using

in their program to hold a column value is of a different type than the column itself. Even 100% statement coverage during testing is not enough to catch a data type mismatch.

Figure 4 contains a function to generate an SQL string to update the UnitsInStock column of the Products table. The SQL data type of the UnitsInStock column is smallint, a 16 bit integer. The developer of the function has mistakenly coded the unitsInStock parameter as an int, a 32 bit integer. The compiler will not complain about this. This bug will lay dormant until a value that is too large for the smallint to hold is passed into this function. The result will be a runtime error generated by the database engine.

```

57 public string SetUnitsInStock( int productID,
58                                 int unitsInStock )
59 {
60     string sql = "UPDATE Products " +
61         " SET UnitsInStock = " +
62         unitsInStock.ToString() +
63         " WHERE ProductID = " +
64         productID.ToString();
65
66     return sql;
67 }

```

**Figure 4. SQL string with a data type mismatch**

The maintenance of SQL strings is also problematic. Changes to the database schema during the life of an application are guaranteed for any non trivial database application. These changes can come either during the initial development of an application or as part of the requirements for a new version. When confronting database schema changes, development teams are faced with difficult questions. How much work will it take to modify the existing body of SQL strings to stay in sync with the changing database schema? How much time will the testing effort require to validate that the SQL strings are in sync with the database schema? How many customer support calls will we receive as a result of runtime errors due to invalid SQL strings?

SQL strings also render an application highly susceptible to SQL injection attacks [14]. A seemingly harmless function such as the one listed in Figure 5 is a major security hole. A malicious user could craft the value of the companyName parameter in such a way, that they could execute arbitrary SQL statements against the database engine. For example, if the value of companyName was set to "Bad Guy"; drop table Customers --", the Customers table would be dropped.

```

69 public string UpdateCustomer( int customerID,
70                                 string companyName )
71 {
72     string sql = "UPDATE Customers " +
73         " SET CompanyName = ' " +
74         companyName +
75         "' WHERE CustomerID = " +
76         customerID.ToString();
77
78     return sql;
79 }

```

**Figure 5. A major security hole**

There is also a lack of developer support during the development of SQL strings. The IDE cannot prompt developers with a list of table names or column names because as far as it knows, the developer is writing a string. This means that developers must have the database schema memorized, or they must look up the

needed information. Having to look up such detail frequently can break their concentration. This, in turn, can lead to decreased productivity. As the size of the database increases, this problem is exacerbated.

### 3. SQL DOM

Our solution consists of two parts. The first is an abstract object model. The second is an executable, *sqldomgen*, which is executed against a database schema to generate a concrete instantiation of the abstract object model. In this section, we will examine our solution in depth.

#### 3.1 Abstract Object Model

Developing the abstract object model was the most challenging problem we faced. We wanted to develop an object model that accomplished two things which were not entirely orthogonal. Our first goal was to construct an object model that could be used to construct every possible valid SQL statement, which would execute at runtime. Our second goal was for our object model to be easy to use. We felt that if the developer had to go through too many contorted and confusing steps, our tool would not be used, and would therefore be useless. When we were forced to choose between the two goals, we always chose the first goal.

One example of the struggle between these two goals occurred during the design of the class that would be used to construct *insert* SQL statements. For an *insert* SQL statement to be valid it has to contain a value for every column in the table that is not auto increment or does not have a default value specified. We had two implementation options. The first was to have the constructor take all of the required values as parameters. This would allow the compiler to ensure that all necessary values were supplied to the insert SQL statement. The other option was to have properties or functions that would be used to associate a value for a column with the insert SQL statement. This would be a little more developer friendly but the compiler would not be able to guarantee the validity of the insert SQL statement. In line with goal number one, we chose option one.

#### 3.2 SQL DOM Generator

We knew that we would need to generate code dynamically for our solution to be viable. The first author was familiar with the classes in the .NET Framework that allow for the dynamic generation and compilation of code. As a result, *sqldomgen* was developed using C# [6] and the .NET Framework [1]. Our approach, however easily generalizes to other development languages and frameworks, including Java.

*sqldomgen* performs three main steps. The first step is to obtain the schema of the database. This is currently accomplished by using methods provided by an OLEDB Provider. The second step is to iterate through the tables and columns contained in the schema and output a number of files containing a strongly-typed instance of the abstract object model. The final step is to compile these source files into a dynamic link library (DLL).

To ensure that changes to the database schema result in compile time errors, we envision the *sqldomgen* being executed as part of the daily build of an application.

### 3.3 Concrete Object Model

The object model consists of three main types of classes. They are SQL statements, columns and where conditions.

Figure 6 shows a few of the SQL statement classes generated for our sample database. For each of the four types of SQL statements (*select*, *insert*, *update* and *delete*), a class is created for each table in the database schema. These classes are used to construct SQL statements. To construct a *select* SQL statement for the customers table, you would use an instance of the CustomersTblSelect-SQLStmt class. To construct an *update* SQL statement for the Orders table, you would use an instance of the OrdersTblUpdateSQLStmt class. Each class is associated with a single table. The constructors of each class are typed to take only parameters representing columns of the table with which the class is associated.

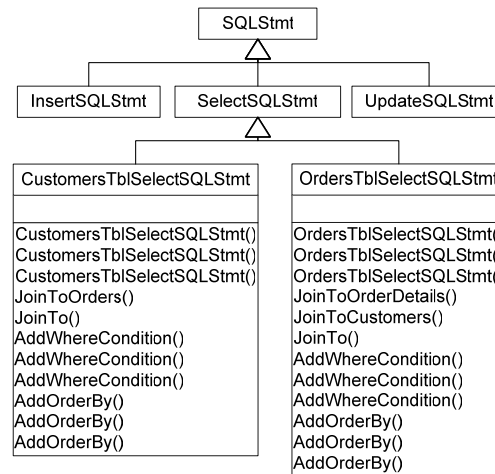


Figure 6. SQL statement classes

*Select* SQL statements have a *JoinTo<TableName>* method for each table with which they share a foreign key relationship. This precludes the developer from having to remember or look up the names of the foreign and primary key columns.

Column classes, shown in Figure 7, are used as parameters to the constructors and methods of the SQL statement classes. They are used to specify which columns are to be selected, updated or inserted. Column classes hold data in variables of the same type as the column with which it is associated. Since some tables have columns with identical names, namespaces are used to prevent name collisions.

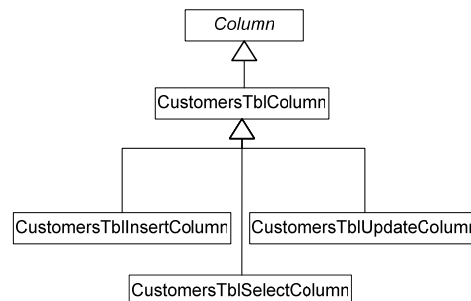


Figure 7. Column classes

It is within the column classes that SQL injection attacks are mitigated. Column classes, whose data type is string, parse and possibly modify the values that are passed to them. For example, a single quote in a string value would be escaped to two single quotes to prevent an SQL injection attack. Other data types besides strings do not require escaping to be performed because of the strongly-typed nature of the classes. A class that represents a column whose data type is int only accepts values that are valid ints. It does not accept string values that could possibly contain an SQL injection attack. In this way the compiler is leveraged to assist us in thwarting such attacks.

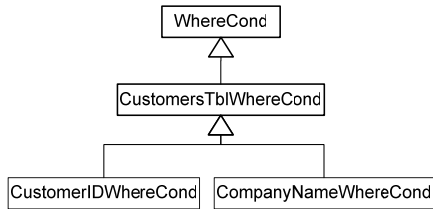


Figure 8. Where condition classes

The final type of class that we will look at in the SQL DOM are where condition classes. These classes, shown in Figure 8, are used to specify conditions in the where clauses of *select*, *update* and *delete* SQL statements. Where condition objects are added to SQL statements through the *AddWhereCondition* function. Where condition classes can also be grouped together and arbitrarily nested to create complex conditions.

### 3.4. SQL DOM in Action

In this section we demonstrate how the SQL DOM overcomes the problems associated with SQL strings that were described in Section 2.

Figure 9 shows a rewritten *GetAllCustomers* function. The previous version of this function had some misspelled table and column names. In the new version, there are no strings at all. Misspelling a column or table name in the new function would result in a compiler error message.

```

17 public string GetAllCustomers()
18 {
19     CustomersTblSelectSQLStmnt sql
20     = new CustomersTblSelectSQLStmnt (
21         ECustomersTblColumns.CustomerID,
22         ECustomersTblColumns.CompanyName );
23
24     return sql.GetSQL();
25 }
  
```

Figure 9. GetAllCustomers using SQL DOM

Figure 10 shows a rewritten *GetCustomers* function. The old version suffered from an SQL syntax bug as well as some misspelling problems. In the new version, all of the SQL syntax is being generated by the SQL DOM thereby eliminating those errors. The new version is also much easier to read and comprehend which tends to result in increased maintainability.

```

29 public string GetCustomers( string companyName,
30                             string contactName,
31                             string city,
32                             string region,
33                             string country )
34 {
35     CustomersTblSelectSQLStmnt sql =
36         new CustomersTblSelectSQLStmnt ();
37
38     // add a where condition for CompanyName
39     // if the user specified a CompanyName
40     if ( (companyName != null)
41         && (companyName.Length > 0) )
42     {
43         sql.AddWhereCondition(
44             new CompanyNameWhereCond(companyName) );
45     }
46
47     // similar code would be placed here for each
48     // of the other five possible conditions
49
50     return sql.GetSQL();
51 }
  
```

Figure 10. GetCustomers using SQL DOM

Figure 11 shows a rewritten *SetUnitsInStock* function. The old version of the *SetUnitsInStock* function suffered from a hard to find data type mismatch bug. If a developer accidentally did the same thing using the SQL DOM, a compiler error message would be generated like the one shown as a tool tip in Figure 11. *sqldomgen* generated a *UnitsInStock* property whose type matches the type of the *UnitsInStock* column. Internally, the *UnitsInStock* property instantiates the *UnitsInStockUpdateColumn* class to do all of the work. The type of the *UnitsInStock* column is *shortint*, which is equivalent to a short in C#. When an *int* is assigned to the property instead of a short, the compiler generates an error. The developer would be reminded that the data type of the *UnitsInStock* column is in fact short and the appropriate code changes could be made.

```

50 public string SetUnitsInStock( int productID,
51                                 int unitsInStock )
52 {
53     ProductsTblUpdateSQLStmnt sql =
54         new ProductsTblUpdateSQLStmnt ();
55
56     sql.UnitsInStock = unitsInStock;
57     // Cannot implicitly convert type 'int' to 'short'
58     sql.AddWhereCondition(
59         new ProductIDWhereCond( productID ) );
60
61     return sql.GetSQL();
62 }
  
```

Figure 11. SetUnitsInStock using SQL DOM

Our final function rewrite is shown in Figure 12. The previous version of this function was a gaping security hole. The SQL DOM has now plugged the hole. If a malicious user attempts to perform an SQL injection attack [14] by submitting a specially crafted value for the *companyName* parameter, it will be mitigated by the *CompanyNameUpdateColumn* class (created internally by the *CompanyName* property). In the case where the value of the *companyName* parameter was set to "Bad Guy"; drop table Customers -, the *CompanyNameUpdateColumn* would modify it to be "Bad Guy"; drop table Customers -. The attack would have failed.

```

64 public string UpdateCustomer( int customerID,
65 string companyName )
66 {
67     CustomersTblUpdateSQLStmnt sql =
68     new CustomersTblUpdateSQLStmnt ();
69
70     sql.CompanyName = companyName;
71
72     return sql.GetSQL ();
73 }

```

Figure 12. A plugged security hole

The SQL DOM encapsulates the entire database schema. Because of this, the IDE now knows the database schema. Therefore, developers can rely on the IDE to prompt them for, say, column names, as in Figure 13, or the data type of a column, as in Figure 14. This relieves developers of the burden of memorizing the database schema or having to look it up somewhere.

```

17 public string GetAllCustomers()
18 {
19     CustomersTblSelectSQLStmnt sql
20     = new CustomersTblSelectSQLStmnt (
21     ECustomersTblColumns.CustomerID,
22     ECustomersTblColumns. );
23
24     return sql.GetSQL ();
25 }

```

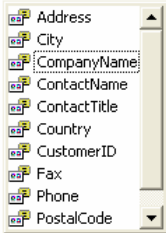


Figure 13. The IDE prompting the developer

```

50 public string SetUnitsInStock( int productID,
51 int unitsInStock )
52 {
53     ProductsTblUpdateSQLStmnt sql =
54     new ProductsTblUpdateSQLStmnt ();
55
56     sql.UnitsInStock = |
57     short ProductsTblUpdateSQLStmnt.UnitsInStock

```

Figure 14. The IDE prompting the developer

## 4. EVALUATION

### 4.1 Benefits

Before we subject our approach to a quantitative evaluation using performance measurements in Section 4.2, we first summarize its qualitative benefits.

As shown in the preceding section, the SQL DOM together with the supporting tool, *sqldomgen*, allows detecting errors in code that accesses the database during compile-time instead of at runtime. This has a direct impact on the reliability of the running system: runtime error messages and exceptions pertaining to the errors addressed above can be avoided.

The SQL DOM also impacts testability and maintainability of the overall code-base: unit tests no longer have to address typos and data type mismatches, and can focus on functionality instead. Maintenance is improved because the transition from one database schema to another is supported by the compiler and corresponding IDE.

The readability of code accessing the database is increased, because class names make explicit use of the “language” created by the database schema; simple syntactic errors, such as string concatenation errors are avoided entirely. This may come at the expense of a slight increase in code length, caused by length of class names in the SQL DOM.

While a thorough evaluation in production environments is an element of future work, we have already witnessed these benefits in a large enterprise integration effort; in this project, quality prototypes have to be produced at high frequency, while concurrent changes to multiple database schemata require flexibility in the code accessing the database. The SQL DOM provides this flexibility and increases database access security at the same time.

### 4.2 Execution Time

The benefits of the SQL DOM come at a cost. The price we pay is increased time to construct SQL statements. Without the SQL DOM we simply manipulate strings to generate SQL statements. With the SQL DOM we are instantiating and manipulating a number of objects to generate SQL statements. To get an idea of the magnitude of the cost we incur, we ran three tests. In each of the three tests we compare the SQL DOM execution times with that of SQL string execution times, in the generation of identical SQL statements. To see how this cost changes with the size of the resulting SQL statement, each test contains five increasingly bigger versions of the same type of SQL statement.

First we tested the time it takes to generate a simple *select* statement. Only the number of columns being selected was increased from 1 to 13. The results can be seen in Figure 15.

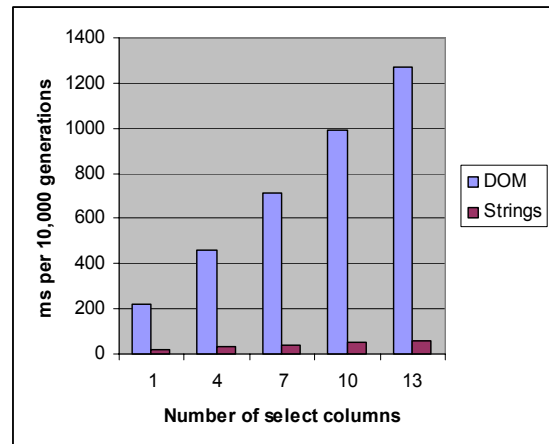


Figure 15. Performance test #1

Our second test also involved the generation of a *select* SQL statement. However, this time the number of columns selected was constant and the number of where conditions were increased from 1 to 13. The results can be seen in Figure 16.

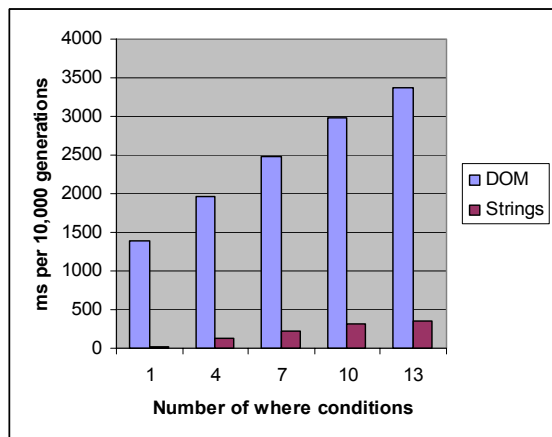


Figure 16. Performance test #2

The final test generated an *update* SQL statement. The number of columns being updated was increased from 1 to 13. The results can be seen in Figure 17.

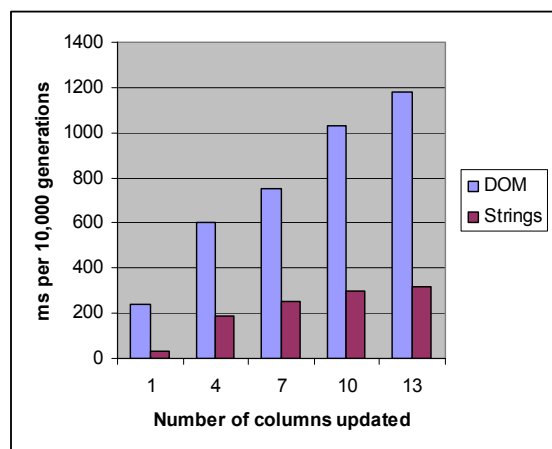


Figure 17. Performance test #3

The results of all of the tests were in line with what we were expecting to see. While in some cases the increase is a factor of 10 it is important to realize that the number being increased by this factor is incredibly small. For example, consider the last data point in Figure 16. The time to generate a single SQL statement is increased from roughly .035 ms to .35 ms. This increase of .31 ms would be unnoticeable when the cost of accessing the database is taken into consideration. On our test machine, the cost of executing simple, static, SQL statements against a local database averaged 180 ms. .35 ms is approximately .2% of 180 ms. So, while the increase in SQL generation times may look expensive, when viewed in the context of database access it is virtually unnoticeable. In our estimation, the performance cost does not come close to outweighing the benefits achieved by the SQL DOM.

As yet, we have not spent any time optimizing the SQL DOM. We believe that the performance of the SQL DOM can be substantially improved.

## 5. RELATED WORK

SQLJ [4] and Embedded SQL [11] provide language extensions for Java and C respectively. SQL statements are written using these extensions in the same file as regular source code. Before being compiled by the regular language compiler, the file is preprocessed. Among other steps, the preprocessor checks the SQL statements against a database schema for syntax and type mismatch errors. The main disadvantage to these approaches is that they do not support dynamic SQL statements. Also, these approaches were never widely adopted. In fact, according to Oracle's SQLJ Roadmap [19], future versions of their development tools and database will no longer support SQLJ due to lack of use.

Brant and Yoder [5] present a collection of patterns for developing applications that need to support user-configurable reports. Applications such as this would contain a great number of dynamic SQL statements. Their patterns, Report Objects, Query Objects, Formula Objects, Composable Query Objects and Constraint Observers, provide a way to organize dynamic SQL statements so that they are easily customizable to support user-configurable reports. Their work is similar to ours in that they propose using objects to construct and output SQL strings. However, they do not propose any sort of compile time checking of the validity of the generated SQL statements.

Object/relational mapping [10, 15, 18] and persistent object systems [3, 7, 20] provide an object-oriented view of data stored in a relational database. Instead of querying the customers table and having a set of rows returned, you would have a collection of customer objects returned instead. The properties on the customer object would correspond to the columns on the customers table. This is very useful and has a number of advantages including a familiar programming model and type safety. Although abstracting away relational data behind a set of strongly-typed objects is useful, it also results in the full power of the database engine being obscured. This leads to the two main disadvantages of this approach, namely the loss of expressive power and performance associated with directly accessing the database engine through a call level interface.

Cook and Rai [9] introduce Safe Query Objects. Safe Query Objects allow query behavior to be defined using strongly-typed objects and methods. In addition, safe query objects support query shipping by automatically generating code to execute queries remotely in a relational database. Java Data Objects (JDO) is used to provide the strongly-typed objects. This approach provides a way to efficiently execute strongly-typed queries that can be expressed in terms of the properties of strongly-typed objects. This is definitely a useful tool for those applications that will be using object/relational mapping or a persistent object system. However, these approaches share the shortcomings we discussed in the previous paragraph.

Gould, Su and Devanbu [12] present a different solution to the same problem we discuss in this paper: overcoming the problems associated with dynamically generating SQL statements. Their approach is to statically analyze source code to locate where SQL strings are being constructed. Each of the different possible resulting strings is then analyzed for syntax and type mismatch errors. Their approach has two notable advantages over our approach. Since all the analysis is performed statically, no

performance penalty is incurred. Their solution can also be used on existing applications whereas our approach would require existing applications to rewrite their SQL statement generation code using classes from the SQL DOM. However, performing static checking leaves the developer powerless to defend against SQL injection attacks. This would require dynamically analyzing and possibly altering user input like we do in the SQL DOM. Another advantage the SQL DOM has is that it offers developers development-time-support in the construction of SQL statements via intellisense™. Also, mistakes made by the developer are caught by the compiler, not by a separate analyzer. This means developers never need to leave the confines of their favorite IDE to find errors in their SQL statements.

Haskell/DB [16] is an effort to embed SQL in the functional programming language, Haskell. HaskellDB is similar to our approach in two ways. The first is that it provides an executable to generate the necessary Haskell types which are used to create SQL strings. This is similar to our *sqldomgen*. The other similarity is that the end result of their language extensions are SQL statements which are then passed to the database through a call level interface. HaskellDB is also similar to the language extensions mentioned above and therefore lacks support for dynamic SQL statements. Another notable difference is that HaskellDB is designed to work with functional programming languages whereas the SQL DOM is designed to be used from object oriented programming languages.

## 6. CONCLUSION

### 6.1 Summary

In this paper we have looked at the problems inherent in accessing relational databases from object oriented programming languages. In particular, we have investigated the shortcomings of communicating with database engines through call level interfaces. This communication requires the construction of dynamic SQL statements through string concatenation and string replacement. Creating SQL statements in this way can lead to a number of problems, including run time errors, SQL injection attacks and code that is not easily maintainable.

To overcome these problems, we have introduced an object model, which we refer to as the SQL DOM. The SQL DOM allows SQL statements to be constructed through the manipulation of objects, which are strongly-typed to the database schema.

### 6.2 Future Work

There are a number of ways in which the SQL DOM can be improved.

It has been our experience that many applications use object/relational mapping and call level interfaces in different parts of the code. Integrating the SQL DOM with an object/relational mapper can provide a unified solution.

Currently, the classes that make up the SQL DOM are output as C# source code using classes from the System.CodeDom namespace of the .NET Framework. At the time we were developing *sqldomgen*, this was the most familiar way to perform dynamic code generation. We have since come across a tool called CodeSmith [22]. CodeSmith allows the developer to write

templates in any language that can then be manipulated programmatically. Rewriting *sqldomgen* to use CodeSmith would make it much easier to provide the SQL DOM to other object oriented languages such as Java.

*sqldomgen* is run against a database schema. One of the inputs to *sqldomgen* is a connection string which is used to connect to the database. *sqldomgen* connects to the database to discover the schema. Allowing the schema to be specified in other ways, such as an XSD file, will increase the flexibility of our approach further.

The SQL DOM classes are manipulated to generate strings containing SQL statements. These strings are then passed by the developer through a call level interface to the database engine. We believe that a significant performance boost can be achieved by integrating the SQL DOM more closely with a call level interface. Providing a way to take advantage of the parameterized query support of the call level interface is an interesting alternative to just generating strings.

The final piece of future work we suggest pertains to other query languages. XPATH [8] is another query language where queries are often constructed using string manipulations. Given an XSD for an XML document, one could generate a set of strongly-typed objects. These objects would then be manipulated to obtain an XPATH query. This approach would give the same benefits to dynamic XPATH queries as the SQL DOM gives to dynamic SQL statements.

## 7. ACKNOWLEDGEMENTS

Our work was partially supported by the UC Discovery Grant and the Industry-University Cooperative Research Program, as well as by funds from the California Institute for Telecommunications and Information Technology (Calit2). We are grateful to Alin Deutsch and the anonymous reviewers for insightful comments.

## 8. REFERENCES

- [1] .NET Framework. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/netfxanchor.asp>, 2004.
- [2] ADO.NET. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconaccessingdatawithadonet.asp>, 2004.
- [3] Atkinson, M. P., and Morrison, R. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319-401, 1995.
- [4] American National Standard for Information Technology. Database languages – SQLJ – Part 1: SQL routines using the Java programming language. Technical Report ANSI/INCITS 331.1-1999, InterNational Committee for Information Technology Standards (formerly NCITS), 1999.
- [5] Brant, J., and Yoder, J. W. Creating reports with query objects. In Harrison, N., Foote, B., and Rohnert, H., editors, *Pattern Languages of Program Design 4*. Addison Wesley, 2000.
- [6] C#. <http://msdn.microsoft.com/vcsharp/>, 2004.
- [7] Cengija, D. Hibernate your data. *onJava.com*, 2004.



- [8] Clark, J., and DeRose, S. XML Path Language (XPath) Version 1.0. Technical report, W3C, 1999.
- [9] Cook, W., and Rai, S. Safe Query Objects: Statically-typed objects as remotely-executable queries. [http://www.cs.utexas.edu/users/wcook/Drafts/SafeQuery\\_CookRai.pdf](http://www.cs.utexas.edu/users/wcook/Drafts/SafeQuery_CookRai.pdf), 2004.
- [10] Dub, J. A., Sapir, R., and Purich, P. Oracle Application Server TopLink application developers guide, 10g (9.0.4). Oracle Corporation, 2003.
- [11] Embedded SQL for C. [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/esqlforc/ec\\_6\\_epr\\_01\\_3m03.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/esqlforc/ec_6_epr_01_3m03.asp), 2004.
- [12] Gould, C., Su, Z., and Devanbu, P. Static checking of dynamically generated queries in database applications. In *Proceedings, 26<sup>th</sup> International Conference on Software Engineering (ICSE)*. IEEE Press, 2004.
- [13] Hamilton, G., and Cattell, R. JDBC™ patterns. Sun Microsystems, 2003.
- [14] Howard, M., and LeBlanc, D. *Writing Secure Code, Second Edition*, Microsoft Press, ch. 12, 2003.
- [15] Keller, W. Mapping objects to tables – a pattern language. In *Proceedings of the 1997 European Pattern Languages of Programming Conference*, number 120/SW1/FB in Siemens Technical Report, Irsee, Germany, X. EA Generali, Vienna, Austria.
- [16] Leijen, D., and Meijer, E., Domain specific embedded compilers. In *Proceedings of the 2<sup>nd</sup> conference on Domain-specific languages*, pages 109-122. ACM Press, 1999.
- [17] Maier, D. Representing database programs as objects. In Bancelhon, F., and Buneman, P., editors, *Advances in Database Programming Languages, Papers from DBPL-1, September 1987, Roscoff, France*, pages 377-386. ACM Press / Addison Wesley, 1987.
- [18] Matena, V., and Hapner, M. Enterprise Java Beans Specification 1.0. Sun Microsystems, 1998.
- [19] Oracle SQLJ Roadmap, [http://www.oracle.com/technology/tech/java/sqlj\\_jdbc/pdf/oracle\\_sqlj\\_roadmap.pdf](http://www.oracle.com/technology/tech/java/sqlj_jdbc/pdf/oracle_sqlj_roadmap.pdf), 2004.
- [20] Russell, C. Java Data Objects (JDO) Specification JSR-12. Sun Microsystems, 1998
- [21] Sanders, R. E. *ODBC 3.5 Developer's Guide*. M&T Books, 1998.
- [22] Smith, E. J. CodeSmith. <http://www.ericjsmith.net/codesmith/>, 2004.