Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 1. pp. 353–359.

SQL Code Complexity Analysis

Melinda Simon, Norbert Pataki

Dept. of Programming Languages and Compilers,
Fac. of Informatics, Eötvös Loránd University, Budapest
e-mail:{melinda, patakino}@elte.hu

Abstract

Structured Query Language (SQL) is a database computer language designed for managing data in relational database management systems, and originally based upon Relational Algebra. Its scope includes data query and update, schema creation and modification, and data access control. Since 1970 SQL became the most widely used language for relational databases, and based upon the declarative paradigm.

In software development lifecycle testing, maintenance and bug-fixing depends on the complexity of code, this is the reason why we would like to have an accurate estimate. In the past two decades a software product is usually designed with the use of a database because it makes the solution easier. For this reason the host language (eg Java, C#, C++, etc.) contains embedded SQL commands, usually as a string which can be constant or can be created at runtime. Most of the used methods consider these code fragments having a constant complexity and calculates with it or use a very simple and imprecise metrics as LOC. (Eg OxyProjectMetrics 1.8.1)

In this paper we argue for a multiparadigm software metric to measure precisely source code of SQL-supported applications. We present an overview about the used techniques, concentrating on the strengths, weaknesses, opportunities, threats.

Keywords: embedded SQL, complexity, measuring

MSC: 68N30 Mathematical aspects of software engineering

1. Introduction

As the major part of the cost connected to software systems is arisen at the testing and maintenance phase, metrics play increasingly important role in software engineering. The cost of testing and the maintenance of the system strongly correlate to the complexity of the software. Metrics for procedural, object-oriented and functional programs are well-known and extensively used today. Unfortunatelly, when

paradigms are mixed, these metrics distort. A typical example is embedded SQL, where declarative SQL statements are embedded into procedural or object-oriented code. Moreover, SQL queries are often sheltered in string objects without being recognized by static analysis tools.

Most modern programs are written by using more paradigms. Object-oriented programs have large procedural components in implementations of methods. AOP implementations highly rely on OOP principles. AspectJ essentially integrates tools for modularizing crosscutting concerns into object-oriented programs. Moreover multiparadigm programs [5] appear in C++, Java, on the .NET platform, and others.

Databases are essential auxiliary tools creating large programs. They relieve querying, deleting and modifying data. SQL is a widely-used language for managing databases. However, SQL is not an imperative language but a set-based *declarative* one. Inasmuch as we use SQL as an embedded language in an imperative host language (for example C#, Java, C++, etc.) we have multiparadigm programs.

Metrics applied to different paradigms than the one were designed for, might report false results [15]. Therefore an adequate measure applied to multiparadigm programs should not be based on special features of only one paradigm. A multiparadigm metric has to be based on basic language elements and construction rules applied to different paradigms. A paradigm-independent software metric is applicable to programs using different paradigms or in a multiparadigm environment. Paradigm-independent metrics should be based on general programming language features which are paradigm- and language independent. Multiparadigm metrics can be used in many various environments for many different purposes [11, 13].

This paper is organized as follows. In section 2 we present a motivating example. After, we present a multiparadigm software metric to measure complexity if SQL code in section 3. We give an overview about the method for extracting SQL codes from a runtime program in section 4. Finally, this paper concludes in section 5.

2. Motivating example

Nowadays SQL often appears as an *embedded language*, therefore we can write SQL statements in a high-level host language [14, 2].

Let us consider the following C# code fragment that can be found in [14]

```
SqlConnection connection = new SqlConnection(
  "server=localhost;database=Northwind;uid=sa;pwd=sa);

SqlCommand command = connection.CreateCommand();

command.CommandText =
  "SELECT TOP 5 CustomerID, CompanyName, ContactName, Address " +
  "FROM Customers" +
  "ORDER BY CustomerID";
```

```
connection.Open();

SqlDataReader reader = command.ExecuteReader();

while(reader.Read())
{
  Console.Writeline("reader[\"CustormerID\"] = " +
    reader["CustomerID"];
  ...
}

reader.Close();

connection.Close();
```

The complexity of this code fragment includes the complexity of the SQL query. But the query appears as a string literal object in the previous code. Usually metrics measure string objects without any semantical examination - maybe has constant complexity or zero complexity at all. Sometimes strings are generated in runtime with string operations (concatenation, replace, etc.). Therefore hard to measure complexity of embedded SQL statements [2].

Multiparadigm metrics can measure the complexity of the previous code snippet, but some strings should be measured as SQL queries and some of them should be measured as normal strings. We would like to measure embedded SQL queries and the host language code with the same metric because of consistency.

AV-graph is a multiparadigm metric that can be applied in this situation and it measures in a sophisticated way. But AV-graph metric should be refined to support SQL queries.

Not all host languages support database connections in a standard way. For example, the C++ standard library does not include standard database handler classes. Most SQL distributions offer tailor-made extensibles for C++. Therefore SQL queries' complexity cannot be measured via standard SQL connection classes. This is a problem because we cannot distinguish between strings according to their context of usages.

3. Proposed software metric

In this section we describe our multiparadigm metric in an informal way. The formal definition of the AV-graph can be found in [13].

The AV-graph complexity based on programs' three different characteristics. Complexity of dataflow, control structure, and datatypes are taken into account. We can transform the source code into a graph in a multiparadigm way because the

previous characteristics cannot be binded to a specific paradigm and they appears in almost all kind of program.

The main concept behind the definition of AV-graph is that the complexity of a certain code element – either data or control – is heavily depends on its environment. The execution of a control node and the possible value of a data node depends on the predicates dominating it. Thus understanding a node depends on its nesting depth.

There is an other possible way to get these results. We can map our AV-graph model with control and data nodes to the Howatt's model [6] without data nodes and data edges. Hence we replace data edges with special control nodes: "reader" and/or "writer". These control nodes only send and receive information. They will be inserted just before and after the real control nodes which read and/or write data. The nesting depth and complexity value we get with this model is the same as the AV-graph complexity.

This definition reflects our experience properly. For example, if we take a component out of the graph which does not contain a predicate node to form a procedure, (i.e. a basic block, or a part of it – this means a single node), then we increase the complexity of the whole program according to our definition. This is a direct consequence of the fact that in our measures so far we contracted the statement-sequences that are reasonable according to this view of complexity. If we create procedures from sequences, the program becomes slightly difficult to follow. Since we cannot read the program linearly, we have to "jump" from the procedures back and forth. The reason for this is that a sequence of statements can always be viewed as a single transformation. This could of course be refined by counting the different transformations as being of different weight, but this approach would transgress the competence of the model used. The model mirrors these considerations since if we form a procedure from a sub- graph containing no predicate nodes, then the complexity increases according to the complexity of the new procedure subgraph, (i.e. by 1).

On the other hand, if the procedure does contain predicate node(s), then by modularization we decrease the complexity of the whole program depending on the nesting level of the outlifted procedure. If we take a procedure out of the flowgraph, creating a new subgraph out of it, the measure of its complexity becomes independent of its nesting level. On the place of the call we may consider it as an elementary statement (as a basic block, or part of it).

As a matter of fact, we can decrease the complexity of a program in connection with data if we build abstract data types hiding the representation. In this case the references to data elements will be replaced by control nodes since data can only be handled through its operations. While computing the complexity of the whole program, we have to take into account not only the decreasing of the complexity, but also its increase by the added complexity determined by the implementation of the abstract data type. Nevertheless, this will only be an additive factor instead of the previous multiplicative factor.

That is the most important complexity-decreasing consequence of the object-

oriented view of programming: the class hides its representation (both data structure and algorithm) from the predicates (decisions) supervising the use of the object of class. We can naturally apply our model to object-oriented programs. The central notion of the object-oriented paradigm is the class. Therefore we describe how we measure the complexity of a class first. We can see the class definition as a set of (local) data and a set of methods accessing them.

A data member of a class is marked with a single data node regardless of its internal complexity. If it represents a complex data type, its definition should be included in the program and its complexity is counted there. Up to the point, where we handle this data as an atomic entity, its effect to the complexity of the handler code does not differ from the effect of the most simple (built-in) types.

The complexity of a class is the sum of the complexity of the methods and the data members (attributes). As the control nodes (nodes belonging to the control structure of one of the member graphs) were unique, there is no path from one member graph to another one. However, there could be attributes (data nodes) which are used by more than one member graph. These attributes have data reference edges to different member graphs.

This model reflects the fact that a class is a coherent set of attributes (data) and the methods working on the attributes. Here the methods (member functions) are procedures represented by individual AV-graphs (the member graphs). Every member graph has its own start node and terminal node, as they are individually callable functions. What makes this set of procedures more than an ordinary library is the common set of attributes used by the member procedures. Here the attributes are not local to one procedure but local to the object, and can be accessed by several procedures.

Let us consider that the definition of the AV-graph permits the empty set of control nodes. In that case we get a classical data structure. The complexity of a classical data structure is the sum of the data nodes. The opposite situation is also possible. When a "class" contains disjunct methods – there is no common data shared between them –, we compute the complexity of the class as the sum of the complexities of the disjunct functions. We can identify this construct as an ordinary function library.

These examples also point to the fact that we use paradigm-independent notions, so we can apply our measure to procedural, object-oriented, aspect-oriented or even mixed-style programs. The AV-graph also analyzed in the declarative programming paradigm [10]. It has been evaluated in a real-time application [9].

4. Extracting methods

In this section we give a brief overview about the extracting methods. The following methods are used to retrieve SQL codes from a runtime program:

- static code analysis
- dynamic code analysis

- sampling

Static code analysis is a method that works with high-level source code. It can be executed independently from the running application. However, no source code modification is necessary. Nevertheless, there is no overhead at runtime. On the other hand, in many cases it is not precise or not all runtime information is available and it does not reflect runtime behaviour. As a technical issue, but it is hard to define if a string literal is an SQL code.

Dynamic code analysis works during runtime. In this case a trace is generated and analyzed at runtime. The trace generation can be on the server side, as well as, on the client side. This method reflects precise program behaviour. With this method we can trace the entire command text, which usually created at runtime. Many disadvantages this method has. The most important problem is, that dynamic code analysis results in a significant runtime overhead. Another problem is also occured. The original source code has to be modified to generate trace at runtime. It can generate huge amount of data.

Sampling is a statistical method. It is perodically sampling the application to collect data. This results in a minimal runtime overhead. Fortunately, we do not need change the souce code of application. In this case important information can be missed. Another problem is that rare actions could be under-represented.

5. Conclusion

This paper describes a widely-used pattern that requires multiparadigm software. We detail a multiparadigm software metric that is able to measure declarative SQL code. We evaluated different methods to collect SQL queries.

Work in progress to fine-tune AV-graph. Especially database features, like constraints, special built-in types (like timestamp) and built-in functions should be evaluated.

Another possible research area is to extend our metrics to stored procedures. Stored procedures has a certain complexity, but their usage may simplify the SQL queries, therefore may reduce general complexity of the system.

References

[1] van den Brink, H., van der Leek, R.: Quality metrics for SQL queries embedded in host languages, in Proc. of Eleventh European Conference on Software Maintenance and Reengineering, 2007

[2] van den Brink, H., van der Leek, R.,Visser, J.: *Quality Assessment for Embedded SQL*, in proc. of Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2007, pp. 163–170.

[3] Calero, C., Sahraoui, H., Piattini, M.: An Empirical Study with Metrics for Object-Relational Databases, In the proc. of the 7th European Conference on Software Quality (ECSQ.02), 2002.

[4] Calero, C., Sahraoui, H., Piattini, M., Lounis, H.: Estimating Object-Relational Database Understandability Using Structural Metrics, In Proc. of the 11th International Conference on Database and Expert Systems Applications (DEXA.01), Munich, 2001.

[5] Coplien, J. O.: *Multi-Paradigm Design for C++*, Addison-Wesley, 1998.

[6] Howatt, J.W., Baker, A.L.: *Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting*, The Journal of Systems and Software **10**, 1989, pp.139–150.

[7] McCabe, T.J.: *A Complexity Measure*, IEEE Trans. Software Engineering, SE-**2(4)**, 1976, pp. 308–320.

[8] Pataki, N., Porkoláb, Z., Csizmás, E.: *Why Code Complexity Metrics Fail on the C++ Standard Template Library*, in Proc. of the 7th International Conference on Applied Informatics (ICAI 2007), Eger, Hungary, Vol. **2**, pp. 271–276.

[9] Pataki, N., Porkoláb, Z., Simon, M., Vincellér, Z.: *Measuring Complexity of Embedded SQL Code*, in Local Proceedings of Advances in Databases and Information Systems (ADBIS 2009), pp. 117–130.

[10] Pataki, N., Simon, M., Porkoláb, Z.: *The AV-graph in SQL-Based Environment*, Proc. of 12th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering QAOOSE Workshop, ECOOP 2008, Paphos, Cyprus, pp. 11–20.

[11] Pataki, N., Sipos, Á., Porkoláb, Z.: *Measuring the Complexity of Aspect-Oriented Programs with Multiparadigm Metric*, In Proc. of 10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering QAOOSE Workshop, ECOOP 2006, Nantes, pp. 1–11.

[12] Piattini, M., Calero, C., Sahraoui, H., Lounis, H.: Object-relational database metrics, L'Objet, Vol. 17, No. 4, Edition Hermès Sciences, 2001, pp. 477–498.

[13] Porkoláb, Z., Sillye, Á.: *Towards a multiparadigm complexity measure*, 9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering QAOOSE Workshop, ECOOP 2005, Glasgow, pp. 134–142.

[14] Price, J.: *Mastering C# Database Programming*, Sybex, 2003.

[15] Seront, G., Lopez, M., Paulus, V., Habra, N.: *On the Relationship between Cyclomatic Complexity and the Degree of Object Orientation*, 9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering QAOOSE Workshop, ECOOP 2005, Glasgow, pp. 109–117.

[16] Sipos, Á., Pataki, N., Porkoláb, Z.: *On Multiparadigm Software Complexity Metrics*, Pure Mathematics and Applications (PU.M.A) vol. **17**, pp. 469–482.

[17] Weyuker, E.J.: Evaluating software complexity measures, IEEE Trans. Software Engineering, 1988, vol.**14**, pp. 1357–1365.

Melinda Simon, Norbert Pataki
Pázmány Péter sétány 1/c., H-1117 Budapest, Hungary