Coupling Detection to Facilitate Maintenance of Database Applications

Paul L. Bergstein and Ashwin Buchipudi
Dept. of Computer and Information Science, Univ. of Massachusetts Dartmouth, Dartmouth, MA, USA

Abstract – *Enterprise applications typically include a relational database layer. Unfortunately, the current generation of IDE's (Integrated Development Environments) do not adequately capture the interaction between the database management system and other layers of the application. For example, current Java IDE's do not evaluate the relationship of classes with the database, or how a particular java method interacts with database tables and columns. We report here our recent progress in developing an Eclipse plug-in that helps the programmer by providing a visual map of interactions between Java code and relational databases. A primary motivation is to facilitate code maintenance in the face of database modifications.*

Keywords: Software maintenance, software visualization tools.

1 Introduction

Modern tools have simplified the development of enterprise applications by bridging gaps across various technologies like file systems, relational databases, messaging, and web services. However, this has also led to challenges in maintenance and enhancement of enterprise applications. An enterprise application usually consists of a web layer, the business logic and relational database, often enhanced with frameworks like Struts and Hibernate for web and persistence. However, the interaction between these various layers is not sufficiently captured by the current generation of IDE (Integrated Development Environment). For example, the Eclipse IDE provides support for syntax and debugging of java classes, but it does not evaluate the relationship with the database, or how a particular java method interacts with database tables and columns. For example, it does not flag a warning where an SQL query might be formed incorrectly. Similarly, the Visual Studio .NET would not flag a warning if an XPath applied on an XML document does not correspond to a valid value according to the schema. This makes it very difficult to maintain and enhance applications written by a third party, since a change in code may break some other layer, and the problem will become known only after extensive testing.

Our goal is to develop a framework that will help programmers in bridging the gap between different technologies used in an enterprise application. However, this is very substantial initiative and we report here our progress in developing an Eclipse plug-in that helps the programmer by providing a visual map of interactions between Java code and relational databases.

The obvious benefit of the mapping is to facilitate code maintenance in the face of database modifications by identifying the code-to-database couplings. In some cases the string search function of the IDE's editor might be useful for finding affected code when changes are made to the database schema. However, this technique is difficult or impossible to use in certain situations. Suppose, for example, that a column name is changed in one table, but other tables have columns with the same name. A search for the column name in the java code may find many instances that are irrelevant. Furthermore, consider that table and column names may be stored in variables, passed as parameters to other methods, or constructed dynamically in code (e.g. by string concatenation). In such cases, the string search technique may fail to detect many areas of affected code. With our tool, the developer only needs to click on a database element to find the code coupled to that element.

A second, and equally important, benefit involves the easy detection of code-to-code couplings that arise when different java methods access the same database elements. Suppose a developer has a Java enterprise application which uses relational database for data persistence, and the Eclipse IDE is being used for development. The programmer wants to make some changes to a method and would like to know the effects of this change on rest of the code. Ordinarily the programmer could use the "Call Hierarchy" feature of the Eclipse IDE to get the dependencies of other methods and classes on this method. But suppose the method uses an SQL statement to store a string in the *address* column of the *customer* table, and the developer wants to change the format of the address. This is not easy because there may be many other methods which are dependent on the address format but are not related to the current method containing this SQL query through the call hierarchy. As noted above, the editor's search function is not a reliable way to find the affected methods. Therefore the programmer has to manually inspect all the classes and check for methods referencing the address column of the customer table, but even this manual process is highly error prone when column and table names are passed as parameters and accessed through parameter or variable names.

2 Background

We have previously reported [1,2] our development of a prototype tool to provide a visual mapping of java code-to-database couplings. In our initial effort we developed a stand alone application to scan java source code and present the coupling information to the user in tabular format. After entering a java source file name and database connection information (database, host, username, and password) the tool would scan the source code for database access and display tables with the couplings that were found.

When we tested the prototype on a simple database application development project, we found it to be quite useful, but several shortcomings were apparent. First, our methodology for identifying the code-to-database couplings depended entirely on static analysis of the java source code, and was not very sophisticated. Second, we had no mechanism for tracking changes to the code and database schema over time, so that the developer would need to identify the relevant couplings *before* making a change. For example, if the developer were to change the name of a database column, it would be necessary to find the couplings of code to that column before the change was made. Afterward, the couplings would no longer exist and our tool had no way to identify the affected code.

Also, our user interface was a bit awkward, not integrated into a development environment, and only allowed the user to process a single source file at a time. Therefore it was not very useful for detecting code-to-code couplings between different source files.

In our more recent implementations we have made improvements to address each of these shortcomings. We have implemented our tool as a fully integrated plug-in to the popular Eclipse development environment with the ability to visualize all the code-to-database and code-to-code (via database) couplings for an entire project. We have continued to improve our static code analysis and added a dynamic (runtime) analysis feature. We have added change tracking ability by storing the coupling information in a database that the tool uses internally.

The current implementation also enhances the user interface by providing the ability to view the database and the project at various levels of granularity. For example, users can choose to view couplings of code to anywhere in the database, to a particular table in the database, or to a specific column in a table. Similarly, they can adjust the granularity of their project view between the project, class, and method levels.

Search facilities have also been enhanced to enable users to easily find the types of coupling they are most interested in. For example, when a method that stores information in the database is modified, it is easy to find all of the methods (or classes or projects) that retrieve the same information and might be affected.

3 Results

Figure 1 shows the overall architecture of the tool. The tool uses both static and dynamic analysis of the java code to find database couplings. The results of both analysis methods are combined in the coupling data repository which is also used to track changes over time. The user interface, implemented as an Eclipse plug-in displays the results to the user and allows easy navigation to code based on its database coupling.

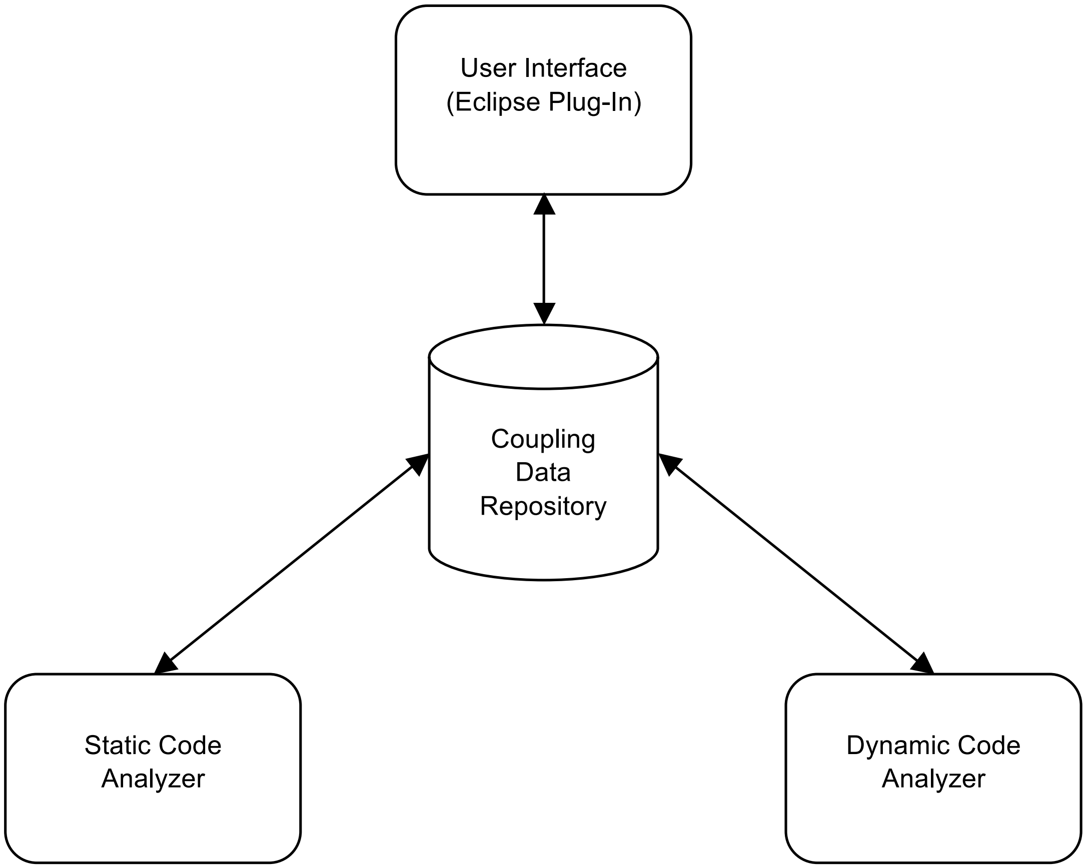

Figure 1

3.1 Static Code Analysis

The static code analyzer uses the Sun java compiler API [11] and the Compiler Tree API [12] to parse the java source and walk the abstract syntax tree. The static code analyzer performs its own parsing, so that it can identify incomplete SQL fragments as well as complete valid SQL statements. In particular, it looks for string literals that are included either directly or after assignment to String variables in calls to the *execute*, *executeQuery*, and *executeUpdate* methods of the JDBC *Statement* class. The analyzer attempts to identify column and table names occurring in select, from and where clauses and record these dependencies in the coupling repository.

Although this ability is still somewhat primitive, the code analyzer is able to detect simple cases of dynamic SQL generation in the code. The analyzer considers certain string concatenations including some concatenations that are built from a combination of string literals and variables.

However, static analysis in general is a hard problem and it will never be possible to detect all couplings to the database that may occur at runtime, possibly dependent on user input. We have therefore focused most of our recent code analysis work on dynamic techniques.

3.2 Dynamic Code Analysis

In order to overcome some of the difficulties of static code analysis, we have implemented dynamic code analysis in our system. The main component of the dynamic analyzer is a JDBC bridge driver that logs the database accesses to the coupling data repository. Our driver acts as a bridge between the application and the "real" driver that communicates with the user's database. The implementation is conceptually simple. Most of the methods in our driver classes simply pass requests on to the underlying "real" driver and return whatever data is returned from the real driver. The main exception is in the Statement class methods (e.g. execute, executeQuery, executeUpdate) that take SQL statements as arguments. These methods receive only complete, valid, fully formed SQL statements as arguments (unless there are errors in the application) even if they have been built dynamically.

The SQL statements processed in the JDBC driver are easily parsed with the ZQL [10] SQL parser to determine the database elements that are being accessed. The driver methods that process the SQL statements create (but don't throw) Exceptions and use the Exception object to obtain a stack trace. Using methods in the StackTraceElement class, the driver can determine the class, method, file, and line number from which it was called. The coupling information is recorded in the coupling data repository.

In order to ensure that all JDBC database access goes through our bridge driver, we also supply a replacement for the DriverManager class. Installation of the dynamic analyzer requires installation of the bridge driver and replacing the standard DriverManager.

3.3 Coupling Database

The coupling data repository is implemented as a database that is used internally by our tool. For every code-to-database coupling that is detected by either the static or dynamic code analyzer, there is an entry in the repository. Each coupling entry in the repository includes the code location (class, method, file, and line number), the database element (database, table, and column), the SQL statement type (select, insert, update, etc.) and the type of access (read, write, or read/write). The statement type does not necessarily determine the access type. For example, a field occurring in the *set* clause of an update statement indicates a write access, but a field occurring in the *where* clause of the same statement indicates a read access.

In order to detect changes over time, the repository also records the first time and last time that a coupling is detected. Also, each time the tool is run, the structure of the database is checked using the JDBC metadata API, and any structural changes are recorded in the repository.

3.4 User Interface

The screenshot in Figure 2 shows the tool interface in an Eclipse pane. On the Database View tab the database structure is shown as a tree with database, table, and column information arranged in a hierarchal structure. Selecting an element from this tree brings the associated couplings into view along with the controls to select sorting options. In the example in Figure 2, the tree is collapsed to a single node (which is selected), and all couplings to the database are displayed. Selecting a code reference from the coupling list brings the corresponding java source into view in an editor pane with the appropriate line of code highlighted.

When two or more methods are coupled to the same database element, there is a suggestion that these methods may be coupled through the database. The nature of the coupling can be seen from the statement type and access type information. For example, one method might read data that is written to the database by another method.

The interface also allows the user to browse in the opposite direction, i.e. from code to database. The project view tab provides a hierarchal view of the projects where the user can select a project, class, or method to find the database elements that it accesses. A developer can find methods that are coupled through the database to a method that has been added or modified by using both views. First, the database elements accessed by the new or modified method can be found in the project view tab. Then, other methods that access the same database elements can be found in the database view tab. In future versions, we plan to improve the interface to automate this task.

4 Related Work

There is a large body of work on software visualization [3-7] and also on database visualization. There is also a good deal of work on reverse engineering of databases and CASE tools that support reverse engineering with visualization techniques. However, we are not aware of any other system designed to support the development and maintenance of software through the visualization of program code dependencies on the database.

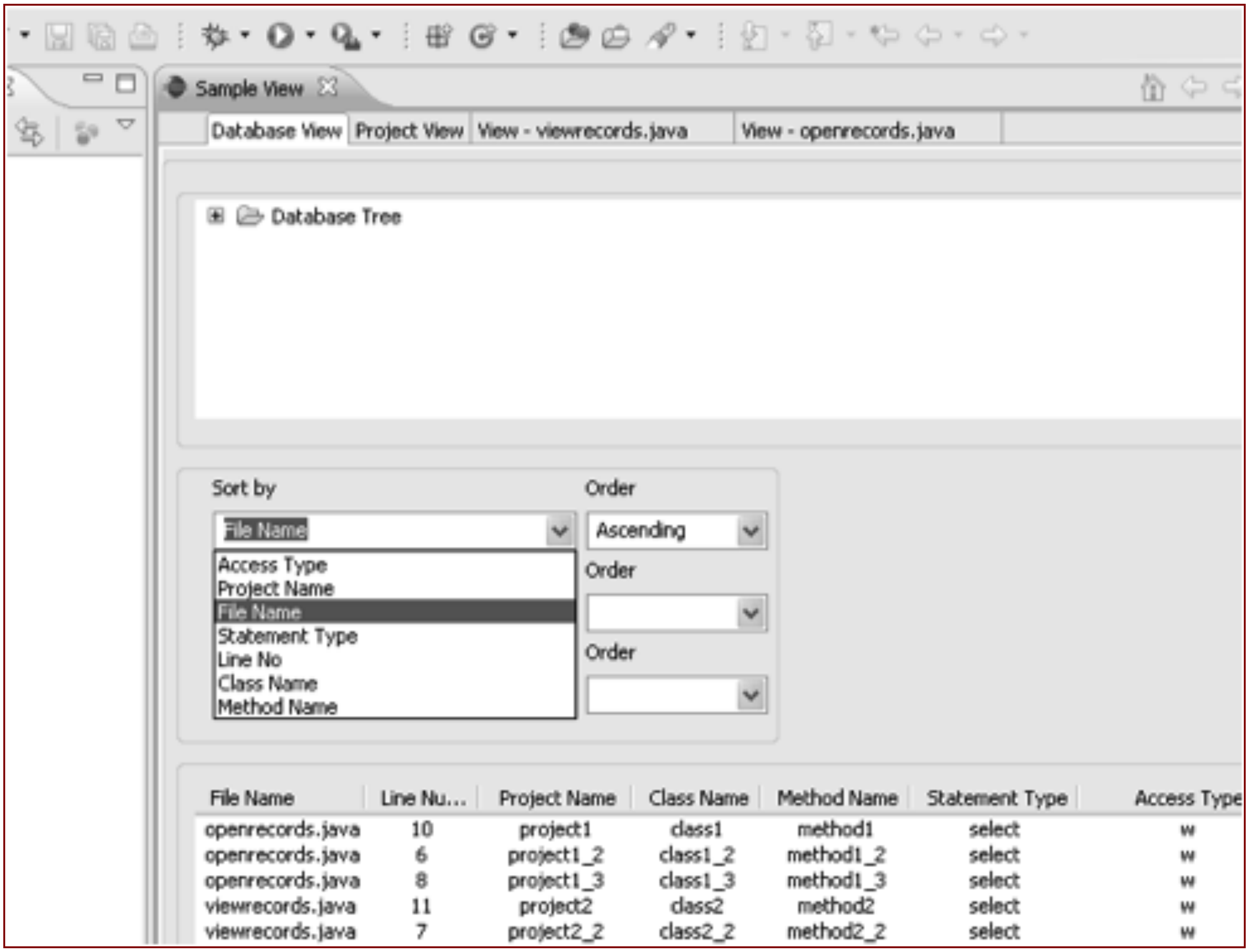

Figure 2

5 Conclusions

Many researchers have investigated to resolve the dependencies between different technologies involved in an enterprise application. Our tool significantly enhances visibility between java and relational databases. The principal benefit is the ability to easily detect the code-to-database couplings and couplings of code-to-code via the database. This ability makes it easy to maintain application code in the face of structural changes to the database, or changes in the format of data stored in the database.

Static and dynamic analysis of java code to discover database couplings each have their advantages and disadvantages. Dynamic analysis is easier to implement and will find all couplings that occur during testing. Static analysis is harder to implement and cannot identify couplings that only occur dynamically (e.g. based on user input). However, static analysis may identify couplings that are missed during the testing phase. By combining the results of static and dynamic analysis in a coupling data repository, we get the combined benefits of each. The repository also allows for tracking of changes over time so that areas of code that may be affected by a change could be flagged for the developer.

6 Future Work

We test our tool on medium sized applications that are developed as team based student projects in a database course. So far, our users have found the tool to be very useful. However, we are actively working on improvements to the user interface based on their feedback. In particular, our users are most interested in automating the discovery of code-to-code couplings through the database, and in automatic flagging of potentially affected code when structural changes to the database occur. We are also working on improving the static code analyzer to reduce the number of couplings that are only detected dynamically.

In the long term, we plan to extend this tool to handle additional languages and technologies. For example, we plan to extend our java code analyzers to support JSP by analyzing the java snippets embedded in JSP pages, so that we can show couplings of JSP pages to the database. This would also allow the visualization of couplings between the presentation layer (JSP) and business logic code that occur through the database in a typical J2EE environment. If all these dependencies between the various layers of a J2EE application can be shown through a visual tool, the task of maintaining and enhancing such applications would be greatly facilitated. Eventually, we would also like to support additional programming languages such as C# and C++ and add support for ODBC applications.

7 References

[1] Sai Ravindran and Paul L. Bergstein. AppDetector: A Tool Prototype for Visualizing Java Code Dependencies on Relational Databases. In *Proceedings of the 2007 International Conference on Software Engineering Research and Practice (SERP'07)*, Pages 497-500, June 25-28, 2007, Las Vegas, Nevada. CSREA Press, ISBN 1-60132-034-5.

[2] Paul L. Bergstein, Priyanka Gariba, Vaibhavi Pisolkar, and Sheetal Subbanwad. An Eclipse Plug-In for Visualizing Java Code Dependencies on Relational Databases. In *Proceedings of the 2009 International Conference on Software Engineering Research and Practice (SERP'09)*, Pages 64-69, July 13-16, 2009, Las Vegas, Nevada. CSREA Press ISBN 1-60132-129-5.

[3] G. C. Roman and K. C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, Vol. 26(12), Pages 11-24, 1993.

[4] Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, Vol. 4, Pages 211-266, 1993.

[5] Jonathan I. Maletic, Andrian Marcus, and Michael L. Collard. A task oriented view of software visualization. In *Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, Pages 32-40, 2002.

[6] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM symposium on Software visualization*, Pages 77-86, 2003. ACM Press.

[7] M. D. Storey, K. Wong, F. D. Fracchia, and H. A. Müller. On Integrating Visualization Techniques for Effective Software Exploration. In *Proceedings of IEEE Symposium on Information Visualization*, Pages 38-45, 1997.

[8] Terence Parr. An Introduction To ANTLR. http://www.cs.usfca.edu/~parrt/course/652/lectures/antlr.html

[9] Java Grammar http://www.antlr.org/grammar/java

[10] Zql: a Java SQL parser http://www.experlog.com/gibello/zql/

[11] Java 6 Compiler API http://today.java.net/pub/a/today/2008/04/10/source-code-analysis-using-java-6-compiler-apis.html

[12] Compiler Tree API http://java.sun.com/javase/6/docs/jdk/api/javac/tree/index.html