AppDetector: A Tool Prototype for Visualizing Java Code Dependencies on Relational Databases

Sai Ravindran and Paul L. Bergstein

Dept. of Computer and Information Science
University of Massachusetts Dartmouth
Dartmouth, MA

Abstract - *Modern tools have simplified the development of the enterprise applications by bridging gaps across various technologies like file systems, relational databases, messaging, and web services. However, the interaction between these various layers is not sufficiently captured by the current generation of IDE (Integrated Development Environment). Our idea 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 a simple prototype, called AppDetector that we have developed for the Java language and relational database interactions as a proof of concept.*

Keywords: Software maintenance, software visualization tools.

1 Introduction

Modern tools have simplified the development of the 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 idea 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 a simple prototype, called AppDetector, that we have developed for the Java language and relational database interactions as a proof of concept. AppDetector helps the programmer to visually map the Java code blocks and database interactions. The technique of parser generators is used for code analysis since it may be easily adopted for other languages. The AppDetector scans java source code, API's, schemas, and other information and identifies application components such as functions, variables, procedures, database tables & columns, and more. Components are organized in a centralized, standards-based catalog. Our prototype currently supports only the Java programming language and relational databases. Enterprise applications written using the Java language are quite common and they access databases through the JDBC API. This API is a simple layer that directly sends the SQL queries (Select, Insert, Delete and Update) to the database. Hence we have chosen these technologies for building our proof of concept. This will enable us to test this tool with real-world programmers.

Future extensions may enhance the application for different technologies and then it can be developed as a framework with bindings for various languages, like C#, PHP and Ruby. AppDetector then would analyze each component to map dependencies within applications, between applications and their underlying databases, and across multiple applications, including integration middleware to uncover dependencies and relationships that would be nearly impossible to find manually. AppDetector would then work like a search engine constantly gathering and updating information to create a living, searchable map of the complex inner-workings of your applications and databases.

2 Background and Motivation

Our AppDetector is based on the following scenario. Suppose a programmer 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. So the programmer uses the "Call Hierarchy" feature of the Eclipse IDE to get the dependencies of other methods and classes on this method.

Now the programmer wants to change an SQL query being used in the program, for example, a column 'date of birth' of Table 'Student' to a column 'place of birth'. This is not easy because there may be methods which are dependent on this column but are not related to the current method containing this SQL query through the call hierarchy. Therefore the programmer has to manually inspect all the classes and check for methods referencing the column 'date of birth'. Sometimes the string search of an editor may be useful but it is not helpful in operations like removing the column or when a query is dynamically constructed inside the code, e.g. if the query is constructed by concatenation of strings like "select date_"+" of_birth from person". In this case using the editor to search for column 'date_of_birth' will fail. To solve this problem the programmer needs a tool to easily visualize the intra-application dependencies just like the Call Hierarchy feature, i.e. an application which helps the programmer to find out the table or columns in a relational database referenced from methods in a programming language.

3 Results

Our AppDetector is built using ANTLR [1] and a public domain java grammar [2] for ANTLR to parse

the java source code and build an abstract syntax tree. The JDBC API is used to extract metadata from the

database accessed by the java code we are analyzing.

The AppDetector then attempts to figure out the dependencies between the two different technologies, using the public domain ZQL parser [3] to help analyze java strings potentially containing SQL. The high-level architecture of the AppDetector protoype is shown in Figure 1.

The Java Analyzer first parses the artifacts (classes and methods) of the programming language and stores information about these artifacts in its database. It also detects all the static strings inside corresponding methods and maps these strings to those particular methods. At this point, AppDetector has a list of classes, methods related to them, and static strings related to every method.

The Database Detector uses the JDBC API to extract all the tables and column names from the database used in the enterprise application. The input for this component is the JDBC URL of the database and credentials required to extract its metadata.

The static strings collected while parsing the source code may be SQL syntax containing table and column

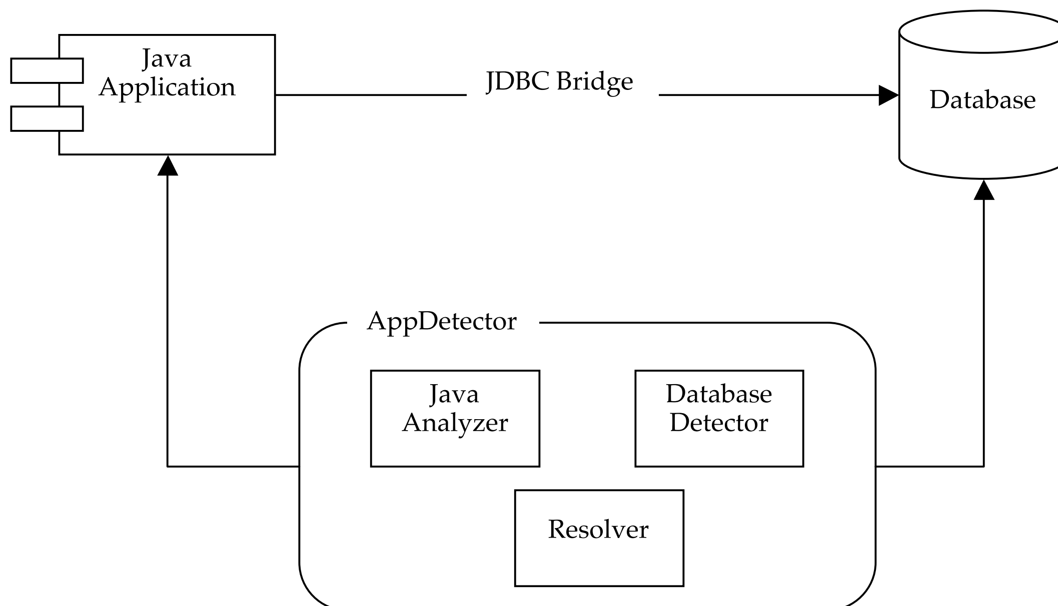


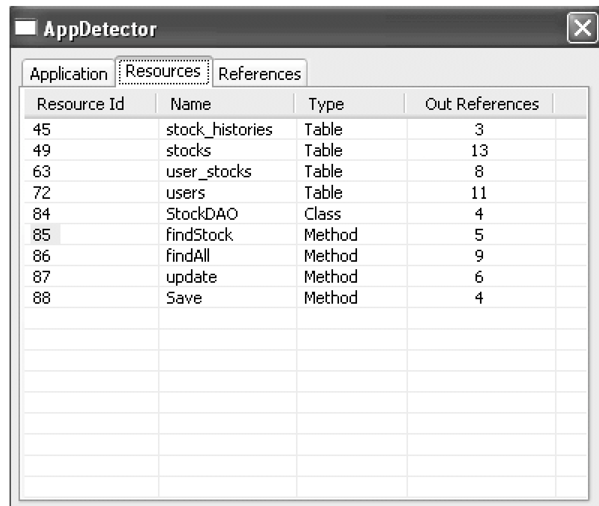
Figure 1: High-level Architecture of AppDetector Application.

names or they may be regular strings. So the Resolver component runs an intelligent matcher that tries to map the static strings found in the methods against database tables and column names. The Resolver verifies which strings belong to SQL statements and maps the database tables and columns contained in the SQL statement to the methods containing these strings. Any such matches indicate the relationship between the database table/column and the method, which is visually shown by the AppDetector tool in a similar way as the Call Hierarchy. This enables the programmer to immediately see which methods are dependent on a particular database table or column.

The user interface component displays the relationships between tables/columns and the methods detected by Application Detector. It also provides a feature to view the relationships so that the programmer can easily find methods depending on a particular column. We have tested our prototype on several student written sample applications and found it to be effective. Figures 2-4 show screen shots from the analysis of a sample stock trading application.

Figure 2: Input Data Screen.

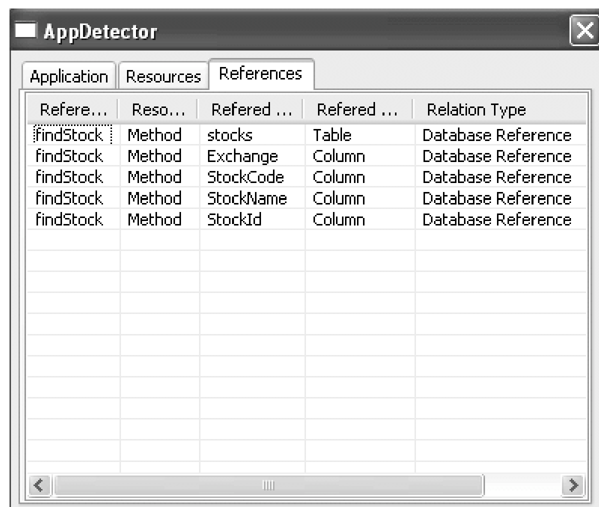
On the Input Data Screen, the user enters a java source file name and database properties which need to be processed and clicks on the 'Start' button to start processing. The user can exit from the application by clicking on 'Exit' button.



Resource Id	Name	Type	Out References
45	stock_histories	Table	3
49	stocks	Table	13
63	user_stocks	Table	8
72	users	Table	11
84	StockDAO	Class	4
85	findStock	Method	5
86	findAll	Method	9
87	update	Method	6
88	Save	Method	4

Figure 3: Resources Screen.

After processing the source code and database, resources are shown to the user. The screen displays the resource Id, resource name, resource type and number of references detected by the AppDetector business logic. User can find out the details of references by clicking on Resource Id.



Refere...	Reso...	Refered ...	Refered ...	Relation Type
findStock	Method	stocks	Table	Database Reference
findStock	Method	Exchange	Column	Database Reference
findStock	Method	StockCode	Column	Database Reference
findStock	Method	StockName	Column	Database Reference
findStock	Method	StockId	Column	Database Reference

Figure 4: References Screen.

The References Screen displays all the references of the selected resource. The screen displays Referrer name, Resource type of referrer, Referred name, Resource Type of the Referred and relationship between referrer and referred.

As shown in the above screen, AppDetector detects the mapping between Java method '*findStock (Stock)*' in our Sample class StockDAO.java and the table '*Stocks*' and columns '*StockName', 'StockCode', 'Exchange*' and '*StockId*'.

4 Related Work

There is a large body of work on software visualization 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.

5 Conclusion and Future Work

Many researchers have investigated to resolve the dependencies between different technologies involved in an enterprise application. The AppDetector utility significantly enhances visibility between java and relational databases. The principal benefits are the ability to detect the dependencies between program code and databases used in enterprise applications. Since AppDetector stores the dependency information in its own database, it would be easy to detect dependency changes and flag them for the user.

We have implemented AppDetector as a utility using Grammar Parsing technology (ANTLR), SQL Parser (ZQL) and JDBC driver. For this project, the sample application is written to extract out the dependencies between java and relational database as proof of concept. A complete application implementation is left for the future work. We have tested our implementation on a sample application and found it to be highly effective. However, we are still working on improvements in several areas.

There are many enhancements required to make AppDetector a really effective tool. The AppDetector can be designed as a web application and can be setup to analyze all code-bases that an organization has in its various repositories. This search engine can be made accessible via a web browser interface to access enterprise applications. Thus it will be easy for programmers across the organization to view the dependencies between various applications and it provides various opportunities of reuse of code.

This prototype can be extended to various technologies so the programmer can view the dependencies between various technologies. For example, it can analyze the java snippets used in the JSP and then find the JSP pages related to the code. In a typical J2EE environment, programmer can detect the dependencies between various layers like presentation (JSP), business logic code and relational database. If all these dependencies can be shown through a visual tool, the task of maintaining and enhancing applications can be made very easy.

6 References

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

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

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

[4] Sun Microsystems, *"Getting Started with the JDBC API,"* Sun Developer Network (SDN) Products and Technologies, Technical Topics, September 1999. http://java.sun.com/j2se/1.3/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html

[5] Sun Microsystems, *"JDBC Overview,"* Sun Developer Network (SDN) Products and Technologies, Technical Topics. http://java.sun.com/products/jdbc/overview.html

[6] SWT: The Standard Widget Toolkit http://www.eclipse.org/swt/