A Static Code Smell Detector for SQL Queries Embedded in Java Code

Csaba Nagy*, Anthony Cleve†
PReCISE Research Center, University of Namur, Belgium
*csaba.nagy@unamur.be, †anthony.cleve@unamur.be

***Abstract*—A database plays a central role in the architecture of an information system, and the way it stores the data delimits its main features. However, it is not just the data that matters. The way it is handled, i.e., how the application communicates with the database is of critical importance too. Therefore the implementation of such a communication layer has to be reliable and efficient. SQL is a popular language to query a database, and modern technologies rely on it (or its dialects) as query strings embedded in the application code. In many languages (e.g. in Java), an embedded query is typically constructed through several string operations that obstruct developers in understanding the statement finally sent to the database. It is a potential source of fault-prone and inefficient database usage, i.e., code smells. In our paper, we present a tool for the identification of code smells in SQL queries embedded in Java code. Our tool implements a combined static analysis of the SQL statements embedded in the source code, the database schema, and the data in the database. We use a lightweight query extraction algorithm to extract SQL code from the Java code and implement smell detectors on the ASG of our fault-tolerant SQL parser. Depending on the context of the smell, its severity is also determined. Developers can examine the identified issues with the help of an Eclipse plug-in or through command line interfaces.**

I. Introduction

In a complex, large-scale information system a database is almost always involved in every use case which gives it a central role in the architecture. A database is also critical as it has to be always readily available and its response time influences the usability of the entire system. Due to its central role, it has been shown that the structure of the database can evolve rapidly reaching hundreds of tables or thousands of database objects [1]. Moreover, because the application code and the database depend on each other, they evolve in parallel [2] resulting in an increased complexity of the source code which implements the database communication. It is vital that this layer remains reliable, robust and efficient.

Although NoSQL solutions are becoming popular, SQL is one of the most widely used languages to interact with a database. According to the 2017 survey of StackOverflow[1], SQL is the second most popular programming language right after JavaScript and ahead of Java. Popular database access technologies also rely on it, e.g., Hibernate (an ORM framework for Java) uses JDBC and internally translates the database accesses to the SQL dialect of the RDBMS. Recently, Goeminne et al. studied 3,707 Java projects on GitHub and found that JDBC, Hibernate, and JPA are the most widely used technologies to communicate with a DB, while JDBC occurs as the only database framework in 56.3% of the projects [3].

Database access technologies intend to help developers in various ways. They make it easier to integrate the communication with the database into the application code, e.g., by providing a link between Java classes and database entities (e.g. ORMs), or merely by supporting to reuse and construct queries (e.g. prepared statements). However, as a drawback, a developer hardly sees the final SQL query that is, in the end, sent to the database. Except for the rather frequent case when the query fails, and it appears somewhere in a log file or a stack trace. In fact, the technology which intended to help obstructs the developer in observing the final SQL statement. It makes harder to understand existing queries and to construct new and efficient ones, which is already a difficult task without the embedded query context [4], [5]. In the worst case, this can lead to a potential source of erroneous, incorrect, or inefficient database usage also known as *code smells* [6]–[8].

Not long ago, Bill Karwin published a book entitled *SQL Antipatterns* [9]. He presents a collection of common issues developers frequently encounter while working with SQL. There are also more books and publications related to SQL code smells [7], [8], but there are only a few tools to identify such code smells. Tools typically designed for database administrators can statically analyze queries and identify common mistakes (e.g., popular ones are *TOAD*[2] and *SQL Enlight*[3]), but they require the SQL code as input.

The analysis of queries embedded in the host code is different, however, and it has several benefits. A developer should not necessarily leave his programming environment to assess existing queries in the application code or to avoid writing a wrong one. To this end, a static analyzer has to analyze (1) the application code to extract queries, then (2) the queries to identify smells. For a precise analysis, this requires two parsing steps for two different languages. Additionally, depending on the success of the query extraction, some statements may remain incomplete because of string fragments which cannot be resolved statically. A simple example is a query to check login credentials at the beginning of a session. A different input means a distinct query, so a static tool either drops the full query because of the unknown part from the user input or has to implement a method to handle it.

[1]https://stackoverflow.com/insights/survey/2017

[2]https://www.quest.com/toad

[3]http://ubitsoft.com

In our paper, we present a prototype tool, which was designed to statically analyze SQL queries embedded in Java code and to identify typical coding mistakes among them, i.e., *code smells*. To the best of our knowledge, there is no other tool available which can statically extract SQL queries from the Java code, then perform a *combined analysis* of the database schema and the database to identify code smells in the queries. With our prototype implementation, we targeted JDBC as a common database access technology and MySQL as a popular RDBMS. *The tool is publicly available*[4].

In the rest of the paper, we discuss related work and tools which are available for similar purposes. Then we introduce our tool, and the code smells we implemented. We also present example usage of the tool and finally we discuss future directions for possible improvements.

II. Related Work

Common mistakes in SQL has been already in the interest of researchers before the appearance of the ISO SQL-92 standard [10]. In 1985, Welty studied how human factors can affect users in using SQL and found that user performance could be significantly improved [11]. Later, Brass et al. started working on the automatic detection of logical errors in SQL queries [12] and extended their work with the recognition of common semantic mistakes [13]. They implemented the *SQLLint* tool which was able to automatically identify these errors in (syntactically correct) SQL statements [14]. The tool seems to be unsupported today. There is another online tool named *SQLLint*[5], but it is a SQL beautifier.

There are also books in this area. The *Art of SQL* [8] and *Refactoring SQL Applications* [15] provide guidelines to write efficient queries, while the book of Bill Karwin [9] collects antipatterns that should be avoided. In a paper, Ahadi et al. presented a large-scale analysis of students' semantic mistakes in writing SQL `SELECT` statements [16]. They collected data from over 2,300 students across nine years and summarized typical mistakes of the students. They found that most of the mistakes were made in queries which require a `JOIN`, a subquery or a `GROUP BY` operator. We argue that queries typically use more complex syntax in practice compared to student projects. Hence, the situation can be even worse.

In the realm of *embedded SQL*, Christensen et al. proposed a technique and a tool (*JSA*, *Java String Analyzer*) to extract string expressions from Java code statically [17]. As a potential application of their approach, they check the syntax of dynamically generated SQL strings. They limit their approach to the syntactic validation of the queries. Wassermann et al. propose a static string analysis technique to identify possible errors in dynamically generated SQL code [18]. With the implementation of a CFL-reachability algorithm they detect *type errors* (e.g., concatenating a character to an integer value). Their approach works with extracted query strings of *valid* SQL syntax. In a tool demo paper, they present their prototype tool called *JDBC Checker* [19]. Recently, Anderson and Hills studied query construction patterns in PHP [20]. They analyzed query strings embedded in PHP code with the help of the PHP AiR framework.

Quality assessment of embedded SQL was proposed by Brink et al. in 2007 [21]. They analyzed embedded query strings in PL/SQL, Cobol, and Visual Basic programs while they propose a generic approach which could be applied to Java too [22]. They investigate relationships which could be detected through embedded queries (e.g., access, duplication, control dependencies) and they propose quantitative query measures for quality assessment. Many static techniques which try to deal with embedded query strings do it with the purpose of *SQL injection detection* [23]. Yeole and Meshram published a survey of these techniques [24]. SQL injection detection is different as the goal is specifically to determine whether a query could be affected by user input. Some papers also tackle *SQL fault localization techniques*. A dynamic approach was proposed by Clark et al. to localize SQL faults in database applications [25]. They provide command-SQL tuples to show the SQL statements executed at database-interaction points.

A recent work of Delplanque et al. targets the database to *assess the quality of the schema* and to detect *design smells* in it [26]. They implement a tool called *DBCritics* which can analyze PostgreSQL schema dumps and identify design smells such as missing primary keys or foreign key references. A tool which also has to be mentioned here is the Eclipse plug-in called *Alvor* [27]. Just like *JDBCChecker* [19] and *JSA* [17], this plug-in analyzes the string expressions in Java code. What is more, *Alvor* checks syntax correctness, semantics correctness, and object availability by comparing the extracted queries against its internal SQL grammar and by checking SQL statements against an actual database.

III. Overview

Our work was inspired by the antipattern catalog of Bill Karwin [9]. As he says, "*SQL Antipatterns describes the most frequently made missteps I've seen people naively make while using SQL.*" The book categorizes antipatterns into Logical Database Design, Physical Database Design, Query, and Application Development categories. Our purpose was to support working with queries embedded in Java code. Hence, we implemented a prototype tool for the identification of Query antipatterns. We refer to these as *SQL code smells* since they are specific to SQL queries and indicate 'smelly' code, i.e. there might be a bug or an issue nearby [6]. The rest of the categories mostly relate to database design.

Figure 1 presents a high-level overview of the main components of our tools. A primary component is the *SQL Extractor* which extracts SQL queries embedded in the Java code. The input of this component is the Java source code, and the output is a list of SQL statements with some additional meta information such as the call chain through which the statement is constructed or the location in the source code where it is sent to the database. By default, the tool executes our lightweight SQL extractor implementation which relies on

[4]http://perso.unamur.be/~cnagy/scam2017-eng

[5]http://www.sqllint.com

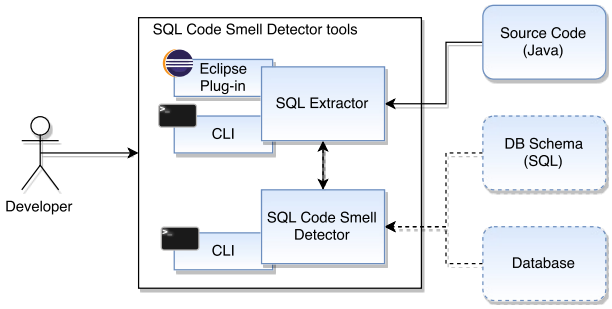

Figure 1. An overview of the SQL code smell detector tools.

an intra-procedural string resolution working with the AST provided by Eclipse JDT. Since the tool comes with an *Eclipse plug-in* as a user interface, this solution makes available an easy way to integrate the analyzer into the IDE and perform a fast, initial analysis that can quickly present results to the developer. Preliminary results indicate that this engine can identify the locations in the source code where SQL statements are sent to the database with great success, and able to extract a query string when its main part is constructed within a method. In a sample open source project management application, which uses JDBC and has about 130 kLOC in Java, *Plandora*[6], the extractor identified 424 unique SQL statements in 62 DAO classes. The design of the tool allows an easy replacement of this component with other SQL extraction engines through the implementation of a wrapper if needed. Example SQL extraction techniques which could be used here are the solution of Meurice et al. [1] or the JSA of Christensen et al. [17]. A more accurate engine could perform better in the extraction of more SQL statements resulting in a more precise analysis but may have an adverse impact on the execution time, which may also affect usability.

The *SQL Code Smell Detector* takes as input the SQL statements extracted and optionally the schema description of the database along with the table content. It parses the statements, constructs an AST then an ASG and runs the smell detectors. The description of the schema can also be provided as data definition statements (e.g. create table statements) in an external file, or when access is given to the database, the tool can fetch it by itself. For parsing, we rely on a newer version of our SQL parser [1], [28], [29]. This version is implemented in Java with an ANTLR4 grammar. The grammar covers data definition and CRUD statements of MySQL with about 400 syntactic rules in 4800 lines of code. It was designed to handle embedded SQL statements even if some of their parts cannot be successfully extracted. For an '*unrecognized code fragment*,' the parser inserts a special node into the AST which indicates the incompleteness of the statement but helps to keep its original structure.

Finally, the code smell detector algorithms are executed to identify different smell types. Each smell detector implements a different algorithm to search for patterns in the ASG during its traversal. Our goal was also to provide a list of smells with a severity categorization which can help the developer to first concentrate on the more serious issues. Hence, a detector does not just point to the source code but also provides severity information based on the type and context of the smell. For some smells, this may require access to the database too. The results are reported to the standard output or in a report file in XML format.

Each analyzer has a *Command-line user interface* to execute the analyses. The main UI, however, is the *Eclipse plug-in* which was developed on top of the SQL extractor. We present some use cases of these interfaces in Section V.

IV. Query Smells

The *SQL Antipatterns* book [9] lists six query antipatterns out of which we implemented four in our prototype. Two of the antipatterns remain unimplemented as we found them too general for static analysis. For the sake of completeness, we introduce all the smells here using the original names from the catalog. We explain the problems and outline our implementation to identify the issues.

A. Fear of the Unknown

The reason behind this antipattern is the special meaning of `NULL` in relational databases that can easily confuse developers. `NULL` is a special marker to indicate that data does not exist in the database, i.e. it is 'unknown'. `NULL` is not the same as `0`. A number 'ten times greater than an unknown' is still 'unknown'. `NULL` is not the same as `FALSE`, either. Hence, a boolean expression with `AND`, `OR`, and `NOT` can easily produce a result that someone may find confusing. Also, `NULL` is not the same as an empty string. A string combined with `NULL` is also `NULL` in standard SQL, but to make things more complicated, different RDBMSs handle it differently, e.g., in Oracle and Sybase the concatenation of `NULL` to a string will result in the original string.

```
SELECT * FROM rentals WHERE customer_id = 456;
-- will not return records where customer_id is null
SELECT * FROM rentals WHERE NOT (customer_id = 456);
```

Listing 1. Fear of the Unknown example.

A common confusion can be seen in Listing 1. Both the first and the second queries are syntactically and semantically correct. The problem is that it is tempting to assume that the result of the second query is the complement of the first query. However, this is not always the case. For records in the table with a `NULL` value for `customer_id`, the expression in the `WHERE` clause evaluates to 'unknown,' which do not satisfy the search criteria, so these records will not appear in the output.

Other typical mistakes can be seen in Listing 2. None of the last two queries will return records where `customer_id` is `NULL`. The proper way to test for `NULL` is to use the `IS NULL` operator in SQL. Both queries are syntactically and semantically correct and will run without error, however.

We implemented the recognition of the antipattern as follows. We check if a column is used in one of the following expressions: `column = NULL` or `column <> NULL`. In such cases, we report a smell with `HIGH_CERTAINTY`. We

[6]http://www.plandora.org/index.html

```
-- will return NULL if age is NULL
SELECT age + 10 FROM customers;
-- will return NULL if e.g. middle_initial is NULL
SELECT first_name || ' ' || middle_initial || ' '
       || last_name AS full_name FROM customers;

-- should use IS NULL or IS NOT NULL
SELECT * FROM rentals WHERE customer_id = NULL;
SELECT * FROM rentals WHERE customer_id <> NULL;
```

Listing 2. Fear of the Unknown example (cont.)

also check whether a column has `NOT NULL` constraint in the database schema if it is used in an expression of a where clause. If the column has no such constraint, and it appears in arithmetic, string or boolean expression, we report a smell with `NORMAL_CERTAINTY`; or with `HIGH_CERTAINTY` if we find a record with a `NULL` field in the table. In any other cases, the problem is reported with `LOW_CERTAINTY`.

B. Ambiguous Groups

`GROUP BY` is a feature of SQL which can make the construction and comprehension of a query more complicated. Ahadi et al. found that `GROUP BY` is one of the most common reasons for students' mistakes in queries [16]. Common confusions are misplaced conditions in `WHERE` clause instead of `HAVING` clause, missing `ORDER BY`, or too many unnecessary columns in the `GROUP BY` clause.

```
-- the reference to 'address' causes unpredictable
-- behaivour in MySQL and SQLite
SELECT name, address, AVG(age) FROM customers
  GROUP BY name;
```

Listing 3. Ambiguous Groups example.

Another source of errors is the handling of columns in the `SELECT` list of a query with a `GROUP BY` clause. The SQL-92 standard is straightforward and it "*does not permit queries for which the select list, HAVING condition, or ORDER BY list refer to nonaggregated columns that are not named in the GROUP BY clause.*" Therefore, the reference to the `address` column in Listing 3 makes the query invalid according to the standard. Later standards, however, such as SQL-99 "*permits such non-aggregates [. . .] if they are functionally dependent on GROUP BY columns*." So, if `address` functionally depends on `name`, the query is acceptable. This relaxation of the rule leads to various implementations in RDBMSs. For instance, Oracle follows the strict standard[7]. While the default behavior of MySQL is that the "*use of GROUP BY permits the select list, HAVING condition, or ORDER BY list to refer to nonaggregated columns even if the columns are not functionally dependent on GROUP BY columns.*"[8] Moreover, "*the server is free to choose any value from each group, so unless they are the same, the values chosen are indeterminate,* ***which is probably not what you want.***" SQLite, as another popular RDBMS implements a similar behavior: "*if the expression is an aggregate expression, it is evaluated across all rows in the group. Otherwise, it is evaluated against a single* ***arbitrarily chosen*** *row from within the group.*"[9] Needless to say that this is a reason for queries which sometimes work as the developer wants it, but sometime may also result in unexpected behavior.

Our implementation detects queries where the `SELECT` list refers to columns which are not listed in the `GROUP BY` clause and not used in an aggregate function.

C. Random Selection

It is relatively common that one needs to select a random value from the database. A standard solution can be seen in Listing 4. The problem with such a solution is that it performs a full table scan and an expensive sort operation, which sounds unnecessary for the selection of a single record. The possibilities in SQL are limited, and even popular StackOverflow posts come to the same solution[10]. However, if one considers the embedded context too, there are additional possibilities to avoid the performance drop back. The *SQL Antipatterns* book gives useful advice to overcome the issue [9]. For example, some servers provide ready solutions like the `TABLESAMPLE` clause in SQL Server or the `SAMPLE` clause in Oracle.

```
SELECT * FROM customers ORDER BY RAND() LIMIT 1;
```

Listing 4. Random Selection example.

We implemented this antipattern by searching for the invocation of the `RAND` method in the `ORDER BY` clause.

D. Implicit Columns

The usage of $\star$ in a `SELECT` list makes it easy to construct a query (see Listing 5), but it is not recommended in batch or embedded context. If the query is embedded, e.g. in Java, a change in the order of the columns in the returned result set may cause problems for the code which processes it. Whenever the structure of the table is modified (e.g., a column is added/deleted, or an existing one is renamed), the query will return records according to the modified structure. Also, the query may return irrelevant columns which may generate unnecessary network traffic. The recommended way to avoid the issue is to give up using $\star$ and name the columns in the select list explicitly.

```
SELECT * FROM customers c JOIN rentals r
  ON c.customer_id = r.customer_id;

INSERT INTO customers VALUES
  (DEFAULT, 'Brown', 'Peter', NULL, NULL);
```

Listing 5. Implicit Columns example.

Our implementation looks for the usage of $\star$ in the `SELECT` lists except if it is used in the form of `tablename.*` which indicates that the developer selected all the columns on purpose.

[7]Restriction on the select list: https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_10002.htm#i2182483

[8]https://dev.mysql.com/doc/refman/5.7/en/group-by-handling.html

[9]https://sqlite.org/lang_select.html

[10]https://stackoverflow.com/questions/580639/how-to-randomly-select-rows-in-sql

E. Spaghetti Query (not implemented)

We tend to construct huge queries when we need to join multiple tables and specify more search criteria spiced with some subqueries. The maintenance of such a query can be a tedious task as it becomes hard to rediscover what and why we (or someone else) did to make it work. The problem can often be broken down into smaller ones which could be put together with the help of views, stored procedures or multiple queries embedded in the client code. Sometimes, however, a single query has benefits: e.g., the SQL engine can better optimize it, or it is easier to integrate one in a reporting tool.

There is no objective definition to tell whether a query is too large with more disadvantages than advantages, so it 'smells' like a spaghetti query. For instance, one can say that a query which joins more than X tables, views or subqueries is too large; or a query over Y lines or characters is a spaghetti query. We found such a rule too subjective to implement in our prototype tool.

F. Poor Man's Search Engine (not implemented)

```
SELECT * FROM customers WHERE name LIKE '%brown%';
SELECT * FROM customers WHERE name REGEXP 'brown';
```

Listing 6. Poor Man's Search Engine example.

We often need to use pattern-matching predicates to compare strings in fields (see Listing 6). Such operations may have poor performance because they cannot always benefit from indexes. The recommendation is to avoid these predicates and take advantage of full-text indexes or text search features of the underlying database, e.g., use `FULLTEXT INDEX` and `MATCH AGAINST` in MySQL. Another suggestion is to use third-party search engines like *Sphinx*[11] or *Apache Lucene*[12]. These solutions are rather problem or RDBMS dependent, so we did not implement the recognition of this code smell.

V. Usage Examples

The tool has three interfaces to interact with users. The primary UI is an Eclipse plug-in, which integrates the analyzer into the Eclipse IDE (currently supported version is Eclipse Neon, 4.3.6). Using the plug-in, one can execute the analysis of a project and then examine the results through a queries view, and markers of the code smells.

An example screenshot of Eclipse with a DAO class opened from the Plandora project can be seen in Figure V. Among the views at the bottom, there is a list of queries in the actual source file including their execution points. In the middle of the editor, there is an example of a prepared statement with a marker showing the actual query sent to the database.

Another interface is the command line interface of the SQL extractor. This command line tool can be fully parameterized to run the analysis of the Java code. It requires a directory of the source files and invokes the Query Extractor. Its analysis is based on the `ASTParser` of the Eclipse JDT API. Configuration options such as the classpath for the analysis can be passed to it through command line parameters. An example usage of this command line interface can be seen in Listing 7. Also, Listing 8 shows an XML output of the tool with a short list of code smells in Plandora.

There is a command line interface for the SQL Smell Detector too. It mainly serves testing purposes but also enables a user to analyze text files containing SQL statements directly. One can get different debug outputs from the phases of analyses, e.g. dumps of the AST.

A detailed description of the usage of different interfaces, the internal working mechanisms, and the supported smells can be found at the website of the tool set (see Section I).

[11] http://www.sphinxsearch.com

[12] http://lucene.apache.org

VI. Conclusions

During our earlier research work, we observed a lack of code smell detector tools which can detect potential coding errors in SQL statements embedded in application code, particularly in Java. Due to the increasing popularity of these technologies, there is a high demand for such a tool, which is also supported by many StackOverflow questions and blog posts. Although there have been some attempts to fill this gap, to the best of our knowledge, there is no publicly available tool that is readily usable for developers. Our goal is to provide a solution and help developers who struggle with complex SQL statements deeply buried in their source code. In this paper, we introduced our prototype implementation which includes a lightweight query extraction and can detect query antipatterns from Bill Karwins *SQL Antipatterns* catalog [9]. It was tested on open source projects with various sizes, and as a prototype tool, it does not just serve well as a proof concept but also demonstrates the potential of the approach. There remain several possibilities for improvements too. Above all, we plan to add more features to the Eclipse plug-in, extend the list of code smells and support additional SQL dialects.

References

[1] L. Meurice, C. Nagy, and A. Cleve, "Static analysis of dynamic database usage in Java systems," in *Proc. of the 28th Int. Conf. on Advanced Information Systems Engineering (CAiSE2016)*. Springer LNCS, 2016.

[2] L. Meurice, M. Goeminne, T. Mens, C. Nagy, A. Decan, and A. Cleve, *Software Technology: 10 Years of Innovation in IEEE Computer*. John Wiley & Sons, 2017, ch. Analysing the Evolution of Database Usage in Data-Intensive Software Systems.

[3] M. Goeminne and T. Mens, "Towards a survival analysis of database framework usage in Java projects," in *Proc. of the 31st Int. Conf. on Soft. Maint. and Evolution (ICSME2015)*. IEEE, 2015, pp. 551–555.

[4] D. A. Robb, P. L. Bowen, A. F. Borthick, and F. H. Rohde, "Improving new users' query performance: Deterring premature stopping of query revision with information for forming ex ante expectations," *J. Data and Information Quality*, vol. 3, no. 4, pp. 7:1–7:22, Sep. 2012.

[5] A. Ahadi, J. Prior, V. Behbood, and R. Lister, "A quantitative study of the relative difficulty for novices of writing seven different types of SQL queries," in *Proc. of the Conf. on Innovation and Technology in Computer Science Education (ITiCSE2015)*. ACM, 2015, pp. 201–206.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] Red Gate Software Ltd., *119 SQL Code Smells*, 2014.

[8] S. Faroult and P. Robson, *The Art of SQL*. O'Reilly Media, Inc., 2006.

[9] B. Karwin, *SQL Antipatterns: Avoiding the Pitfalls of Database Programming (Pragmatic Programmers)*. Pragmatic Bookshelf, 2010.

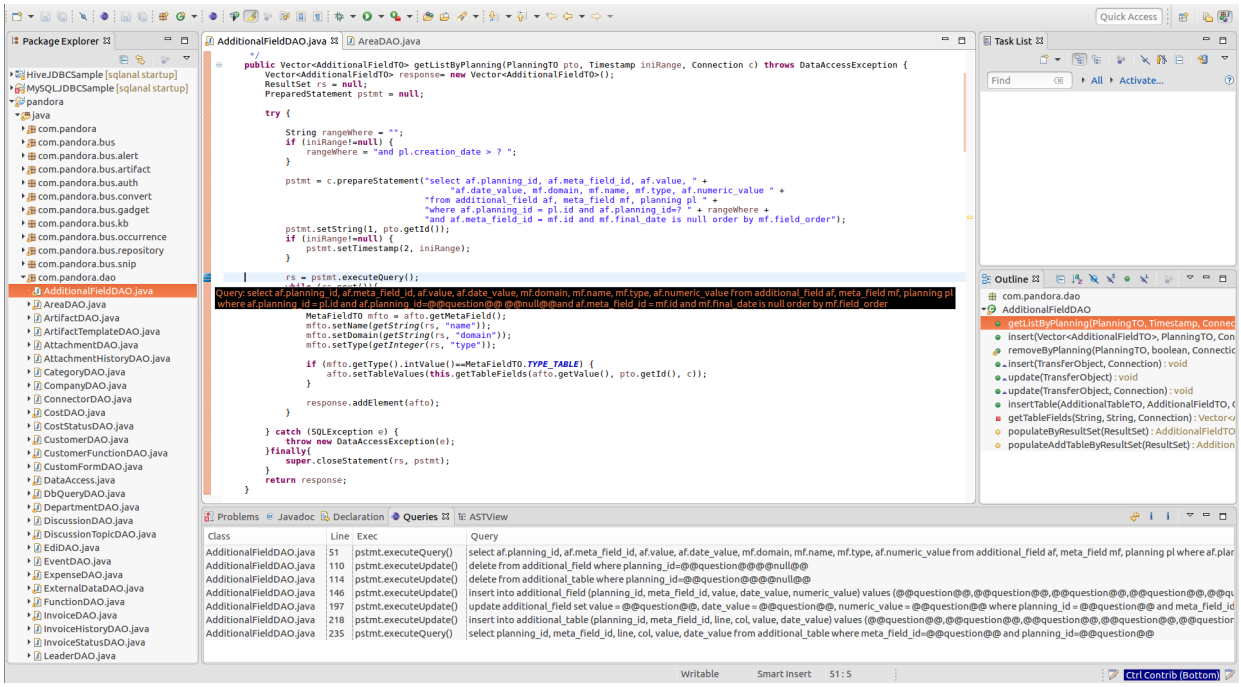

Figure 2. Example screenshot of the Eclipse plug-in.

```
java -classpath .:libs/*:EclipseSQL-0.1.18.jar
eclipsesql.EclipseSQLCLI
-classpath /usr/lib/jvm/java-8-oracle/jre/lib/rt.jar
-dumpqueries plandora-code/queries.xml
-runsmells -smellreportformat xml
-smellreport plandora-code/smells.xml
plandora-code
```

Listing 7. Example usage of the command line interface of the SQL extractor.

```
<Smells>
  <Smell>
    <Kind>ImplicitColumns</Kind>
    <File>java/com/pandora/dao/PreferenceDAO.java</File>
    <Line>90/Line>
    <Certainty>NORMAL_CERTAINTY</Certainty>
    <Message>Name columns explicitly.</Message>
  </Smell>
  <Smell>
    <Kind>ImplicitColumns</Kind>
    <File>java/com/pandora/dao/DbQueryDAO.java</File>
    <Line>139</Line>
    <Certainty>NORMAL_CERTAINTY</Certainty>
    <Message>Name columns explicitly.</Message>
  </Smell>
</Smells>
```

Listing 8. Example code smells in the Plandora project.

[10] *ISO/IEC 9075:1992 Information Technology – Database Language SQL*, Std., July 1992.

[11] C. Welty, "Correcting user errors in SQL," *Int. J. of Man-Machine Studies*, vol. 22, no. 4, pp. 463 – 477, 1985.

[12] S. Brass and C. Goldberg, "Detecting logical errors in SQL queries," in *Proc. of the 16th Workshop on Foundations of Databases*, 2004.

[13] ——, "Semantic errors in SQL queries: A quite complete list," *J. Syst. Softw.*, vol. 79, no. 5, pp. 630–644, May 2006.

[14] C. Goldberg, "Do you know SQL? About semantic errors in database queries," Higher Education Academy, Tech. Rep., 2009.

[15] S. Faroult and P. L'Hermite, *Refactoring SQL Applications*. O'Reilly Media, Inc., 2008.

[16] A. Ahadi, J. Prior, V. Behbood, and R. Lister, "Students' semantic mistakes in writing seven different types of SQL queries," in *Proc. of the Conf. on Innovation and Technology in Computer Science Education (ITiCSE2016)*. ACM, 2016, pp. 272–277.

[17] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proc. of the 10th Int. Conf. on Static Analysis (SAS2003)*. Springer-Verlag, 2003, pp. 1–18.

[18] G. Wassermann, C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, Sep. 2007.

[19] C. Gould, Z. Su, and P. Devanbu, "JDBC Checker: A static analysis tool for SQL/JDBC applications," in *Proc. of the 26th Int. Conf. on Software Engineering (ICSE2004)*. IEEE Com. Soc., 2004, pp. 697–698.

[20] D. Anderson and M. Hills, "Query construction patterns in PHP," in *Proc. of the 24th Int. Conf. on Software Analysis, Evolution and Reengineering (SANER2017)*. IEEE, Feb 2017, pp. 452–456.

[21] H. v. d. Brink, R. v. d. Leek, and J. Visser, "Quality assessment for embedded SQL," in *Proc. of the 7th Int. Working Conf. on Source Code Analysis and Manipulation (SCAM2007)*. IEEE, 2007, pp. 163–170.

[22] H. v. d. Brink, "A framework to distil SQL queries out of host languages in order to apply quality metrics," Master's thesis, Utrecht Univ., 2007.

[23] M. Sonoda, T. Matsuda, D. Koizumi, and S. Hirasawa, "On automatic detection of SQL injection attacks by the feature extraction of the single character," in *Proc. of the 4th Int. Conf. on Security of Information and Networks (SIN2011)*. ACM, 2011, pp. 81–86.

[24] A. S. Yeole and B. B. Meshram, "Analysis of different technique for detection of SQL injection," in *Proc. of the Int. Conf. & Ws. on Emerging Trends in Technology (ICWET2011)*. ACM, 2011, pp. 963–966.

[25] S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold, "Localizing SQL faults in database applications," in *Proc. of the 26th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE2011)*. IEEE Comp. Soc., 2011, pp. 213–222.

[26] J. Delplanque, A. Etien, O. Auverlot, T. Mens, N. Anquetil, and S. Ducasse, "Codecritics applied to database schema: Challenges and first results," in *Proc. of the 24th Int. Conf. on Soft. Analysis, Evolution and Reengineering (SANER2017)*. IEEE, Feb 2017, pp. 432–436.

[27] A. Annamaa, A. Breslav, J. Kabanov, and V. Vene, "An interactive tool for analyzing embedded SQL queries," in *Proc. of the 8th Asian Conference on Programming Languages and Systems (APLAS2010)*. Springer-Verlag, 2010, pp. 131–138.

[28] C. Nagy, L. Meurice, and A. Cleve, "Where was this SQL query executed? A static concept location approach," in *Proc. of the 22nd Int. Conf. on Software Analysis, Evolution, and Reengineering (SANER2015)*. IEEE Comp. Soc., 2015, pp. 580–584.

[29] C. Nagy and A. Cleve, "Mining Stack Overflow for discovering error patterns in SQL queries," in *Proc. of the 31st Int. Conf. on Software Maintenance and Evolution (ICSME2015)*. IEEE, 2015, pp. 516–520.