International Journal of Software Engineering and Its Applications
Vol.8, No.3 (2014), pp.321-338
http://dx.doi.org/10.14257/ijseia.2014.8.3.30

Evaluation and Tuning Test of False Positive and False Negative using Static Analysis Non-Linear Method in JDBC

Shadrach Jabonir and Ford Lumban Gaol

Bina Nusantara University Anggrek Campus, Jakarta, Indonesia
[1]shadrach.jabonir@live.com, [2]ford_lg@binus.edu

Abstract

In this paper, it will be explained deeply in how to test the measurement of Bedighasan application which is used non linear static analysis medhod against Findbugs which is used static analysis too but it adapts linear code scan. Findbugs is able to detects many bugs pattern, but some parts are not quite accurately detected. It is called false positives when there is no bugs but the system raised warning. It is called false negatives if there is a bugs but the system didn't raised warning. Here in this paper, with another method, will try to improve false negatives and also false positives. In this paper those 2 variabel will used as measurements. JDBC used as an area of testing because most of JDBC syntax will not seen error unless it executed. This method is needed to handle bugs pattern which is not able to be detected by standard compiler. In this paper will be shown about how bedhigasan will win the false negatives of findbugs and how bedhigasan will loose the false positives of findbugs. Because of the increasing level of awareness of bugs pattern, it decrease the false positives. This application is still need improvement in how mutting some detected bugs.

Keywords: *Bugs pattern, Static Analysis, Non-linear Code Scanning, JDBC*

1. Introduction

Developments of science in the field of programming languages in recent years are very rapid. But during the rapid era of programming language, there is no offset for the standardization of quality checking about good program code. It will be fatal if the program built has a bug and not able to be found before run the program. Therefore we need a detection system that can find bugs patterns in the program code. This detection refers to the empirical or based on experience. One method that can be used is a static code analysis method by scanning the program code in non-linear way [1].

Static analysis non linear has been used by the writter for analyzing bugs pattern in JDBC. There are some points which had became advantages with non linear code scan [1]. It has been explained about the comparison between linear code scan and non linear code scan [1]. Findbugs used a linear code scan in their analyzer engine [2], different with bedhigasan, it implements non linear code scan.

In fact, JDBC has not been proved whether is able to maintain the integrity or not. If there is a problem then the java code would made the integrity of data threatened or a deadlock program and then the program must be restarted roughly. It needs an application that could find the variety of bug patterns in JDBC. It's almost impossible if it detected with dynamic code analysis. Because of the program cannot be seen buggy unless executed [1].

As it explaned in previous paper about how to implement the static analysis, also explained about how the sequence of process happen in each case in bedhigasan analyzing process [1]. According to the book [3], using JDBC is simple, and it needs to take only a few steps to add database functionality to your application. The steps involved are as follows:

ISSN: 1738-9984 IJSEIA

1. Select/obtain a JDBC driver, or use the JDBC-ODBC bridge driver described later that's included in Java's core classes. If you don't use the bridge driver, you must add the driver code to your CLASSPATH just as you would any other third-party library.
2. Obtain a database connection using DriverManager or a DataSource and a URL that's appropriate for the driver you're using.
3. Create a Statement or an instance of one of its subinterfaces (in other words, PreparedStatement or CallableStatement), and use it to execute SQL commands.

2. Implementation

Testing of bedhigasan will be taken on 3 cases as it explained in introduction. Another important thing in JDBC is *transaction*, so in bedhigasan application, transaction feature will be included. This test will be combined by those 4 cases.

Table 2.1. Test Case Program Type

Project Type	Connection & Driver Bugs	Statement Bugs	Transaction Bugs
Type A	Yes	Yes	Yes
Type B	Yes	Yes	No
Type C	Yes	No	No
Type D	No	Yes	No
Type E	No	No	Yes
Type F	No	Yes	Yes
Type G	Yes	No	Yes
Type H	No	No	No

As seen in Table 2.1, the combination of test case will be seen in each type program (type a-z). Each type code programs are shown in appendixof this journal.

How we got the data for this testing? According to the book [3], JDBC needs Connection, DriverManager, and Statement. Connection and Driver Manager are cannot be separated, so those 2 things are joined in 1 case. As explained above, bedhigasan application will include Transaction. So that in Endnote, in each program will apply those 3 cases and 1 Transaction case. And in each type code program has been spread with bugs deliberetly.

With all the process of testing as seen in table 1, bedhigasan will showing its performance and its ability in finding bugs pattern in java program code especially JDBC. Testing process will be using dynamic black box method, it means program will be ran first to see its output. Unlike the method of bedhigasan application, which able to see the bugs pattern of program before running.

There are 8 types of program, and each of them has been given bugs pattern deliberately. Type A and type H is the most critical type to detects because through them, the highest potential of false positives and false negatives will shown. Beside that, type B – G used for measuring bedhigasan's accuracy in detecting bugs pattern and also passing all the good pattern (part of code program which is not bug or error). So in some type program will be distributed the combination of bugs pattern as seen in Table 1. This testing is very important for stressing application. Measuring the false positives and negaties are very important to determine the good or bad of beghigasan application.

3. Analyzing the Results

The analysis of this testing is trying to get the description of how good of behigasan application in capturing all the bugs pattern in applying the JDBC concept.

In findbug's journal [2] has explained that bugs pattern of java null pointer exception is unpredictable by the warning of static analysis, this is called false negatives, whereas the warning of unrelated with null pointer, this is called false positives. The main goal of static analysis is to push out the posibility of false negatives and false positives. So in this testing the variable of false positives and false negatives will be used as materials of testing.

Table 3.1. False Negatives and False Positives in 2 Projects

Project	Snapshots with NPE	With warning	Observed false neg. %
Search Tree	71	38	46
Web Spider	162	127	21

Project	Warnings	With NPE	Observed false pos. %
Search Tree	40	36	10
Web Spider	129	101	21

Table 3.1 is the example of false negatives and false positives table in findbugs's journal [4]. It is seen that there are 2 type of projects which tested in their testing. The different is they used project which are task of their students and they analyzed the data from there. So they don't have to bring the bugs deliberetly as happen in this journal.

Null pointer exceptions not predicted by any static analysis warning are false negatives, and warnings not corresponding to a feasible null pointer exception are false positives. The goal of any static analysis to find bugs is obviously to make the false negative and false positive rates as low as possible [4].

$$\text{False negatives} = \frac{\text{Snapshots w/ exception but no warning}}{\text{Snapshots with exception}}$$

$$\text{False positives} = \frac{\text{Warnings w/ no exception thrown}}{\text{Warnings}}$$

Figure 3.1. False Negatives and False Positives Formula

As seen in Figure 3.1, its how they calculate the false negatives and false positives, not different with this journal. In false negatives they count it from how many bugs which not detected divided by total bugs. And false positives is calculate from all bugs which not supposed to be detected as bugs divided by all reported bugs. In this journal, each tye will have index or weighting, because unlike the in journal [4], the projects were completed by their students. Meanwhile in this paper, the projects are developed to test the application.

False negatives in this paper is the value of some bug happen but undetected by bedhigasan. How to measure it, is by debuging in black box way. The applications will be tested from type A to type H, and it will be seen how often is the false negatives occurs.

Table 3.2. Index of False Positives

ProjectType	*Index*
Type H	1
Type C, D, E	2
Type B, F, G	3
Type A	4

To make sure the ability of bedhigasan, it needs weighting or index in each type of project. The type H is the cleanest project type from bugs, and type A is the dirtiest type. Therefore about this false negatives, index 1 should be given to type H which is the application shouldn't detect any bugs. Beside that type A should be given highest index

(4), because the application will have bad values if it cannot detects any bugs in type A. The index itself will be times with how much false negatives founded. And the formula will be seen below:

$$Fn = \frac{\sum_{i=1}^{8}(Fn_i c)}{\sum_{i=1}^{8}(B_i c)} x\ 100\%$$

Fn = false Negatives

Fn_i = false negatives in each type (A-H)

B_i = how many bugs founded (A-H)

C = index of each type

False positives in this paper is the value of some **not** bug happen but detected by bedhigasan. How to measure it, is by debuging also in black box way. The application will be test from type A to type H, and it will be seen how often is the false *positives* occurs.

Table 3.4. Index of False Positives

Project Type	*Index*
Type H	4
Type C, D, E	3
Type B, F, G	2
Type A	1

To make sure the ability of bedhigasan, it needs weight or index in each type project just like before. The type H is the cleanest program from bug, and type A is the dirtiest type. Therefore about this false positives, index 4 should be given to type H which is the application should detect all bugs. Beside that type A should be given lowest index (1), because the application will have bad values if it detects any bugs in type A. The index itself will be times with how much false positives founded. And the formula will be seen below

$$Fp = \frac{\sum_{i=1}^{8}(Fp_i c)}{\sum_{i=1}^{8}(B_i c)} x\ 100\%$$

Fp = false positives

Fp_i = false positives in each type (A-H)

B_i = how many bugs (A-H)

C = index of each type

4. Result and Discussion

Testing will be done for both applications, which are bedhigasan and findbugs as a benchmarker. Findbugs is a similar tools, a static analyzer tools which is used as a source code quality checker. The quality of both application will be seen in how they catchs all the posibilities of bugs that could be happen when the application is being executed. Those 2 programs are tested with same the file or project type, file type A to type H. This is happen to ensure that this testing has been fairly setted and also apple to apple.

4.1. Testing Result

Table 4.1. Combination of Cases in each Type Testing

Connection & Driver Manager	Statement	Transaction	Type
Close()	Close()	Rollback()	A
Username	createStatement	X	B
Connection+ close()	X	X	C
X	prepareStatement	X	D
X	X	setAutoCommit (true)	E
X	Close()	setAutoCommit (false)	F
Close()	X	Rollback()	G
X	X	X	H

First test will be held by Bedhigasan.

Type A:

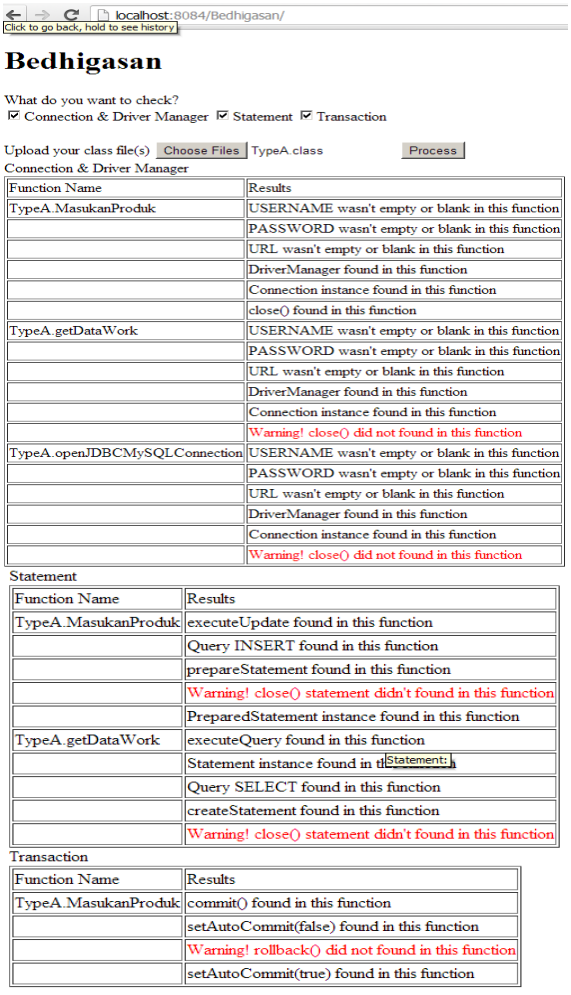

Figure 4.1. Testing Result of Type A

As seen in Figure 4.1, the bedhigasan application is able to detect the bugs pattern in every cases, Connection & Driver Manager, Statement, Transaction. First case is connection & driver manager which is tested about, is there any close() syntax in that code. Bedhigasan application is able to detect the function MasukanProduk which is in close() syntax is exist in that code. Beside that in function getDataWork, close() syntax didn't exists. In function OpenJDBCMySqlConnection, Bedhigasan detects that this function didn't contain close() syntax. And this is a false positives which is happen in Bedhigasan application. Bedhigasan suppose not to detect this bugs. In second case which is Statement, Bedhigasan application is able to detect bugs which are happens in both function. Both functions are not calling close() syntax for each Statement instance in their function. In third case Bedhigasan application able to detect bugs which is happen in MasukanProduk function, which is rollback doesn't exists in its function.

Type B:

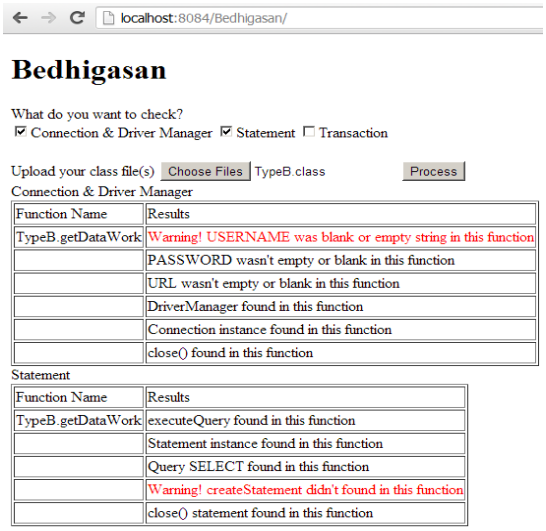

Figure 4.2. Testing Result of Type B

In Figure 4.2, it is seen that Bedhigasan application is able to detect the bugs pattern which are happen in both cases tested. First case is connection & Driver manager, which is in the getDataWork function, the username is blank. The second one is the statement case that has a bug, the CreateStatement syntax didn't exists in getDataWork function and the Bedhigasan Application is able to detects it.

Type C:

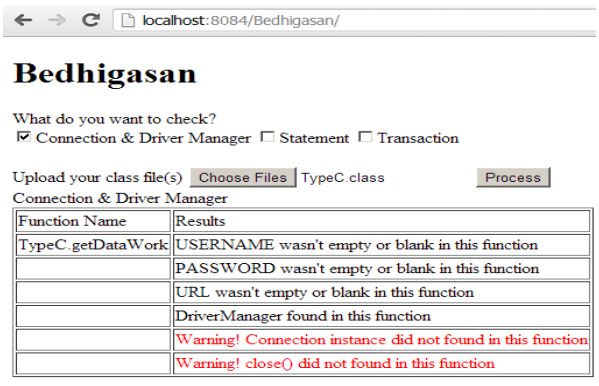

Figure 4.3. Testing Result of Type C

As seen in Figure 4.3, Bedhigasan application is also able to detect the bugs pattern which is happen in getDataWork function. In this type, bedhigasan application was tested in Connection & Driver manager case. In this function, the Connection syntax didn't exists in this function. In this function close() syntax also didn't exists.

Type D:

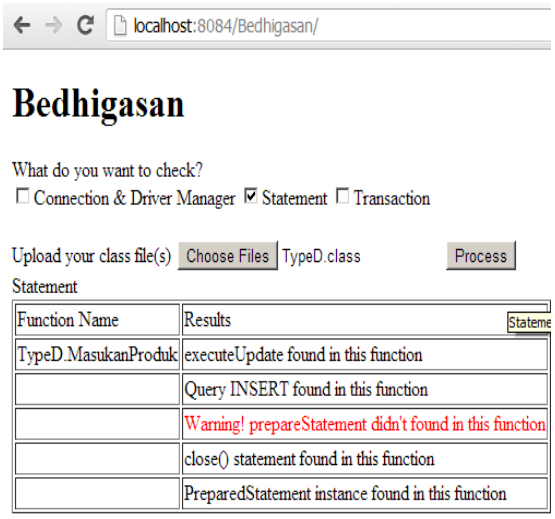

Figure 4.4. Testing Result of Type D

In Figure 4.4, Bedhigasan application is also able to detects the bugs pattern which happen in MasukanProduk function. In type D, the application is tested for statement case. The MasukanProduk function didn't call the prepareStatement for executing the INSERT query.

Type E:

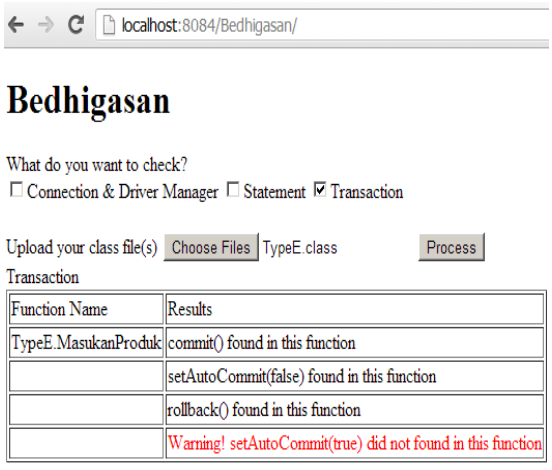

Figure 4.5. Testing Result of Type E

In Figure 4.5, its seen that the bedhigasan application is also able to detect the bugs pattern which is happen in MasukanProduk function. The case that tested in type E is about transaction. In MasukanProduk function, there is no setAutoCommit(true) function called.

Type F:

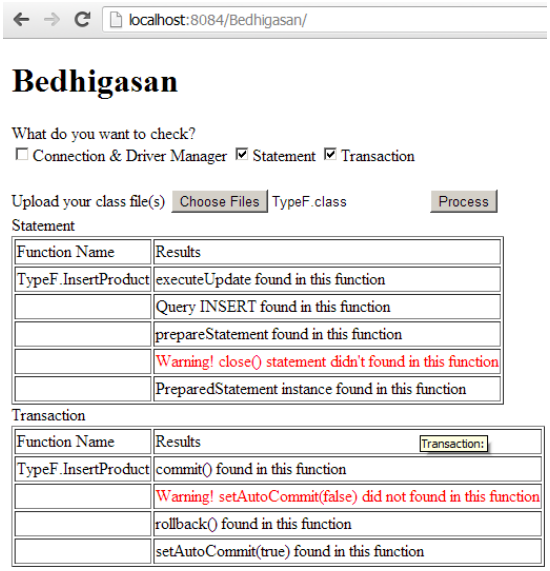

Figure 4.6. Testing Result of Type F

In Figure 4.6, it is seen that the Bedhigasan application is able to detects the bugs pattern which happen in InsertProduct function. In type F, this application tested in 2 cases which are Transaction and Statement. In InsertProduct function, there is no close() syntax for Statement instance. And in Transaction case, bedhigasan application didn't found the setAutoCommit(false).

Type G:

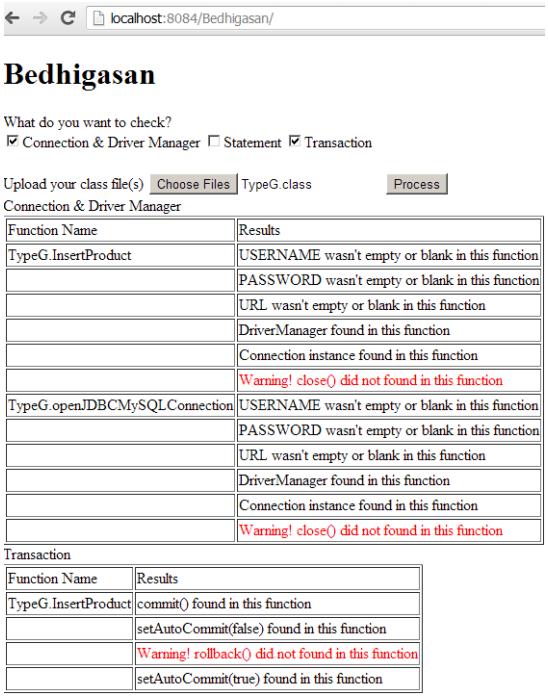

Figure 4.7. Testing Result of Type G

In Figure 4.7, it is seen that the bedhigasan application able to detects the bugs pattern whuch happen in InsertProduct function. In type G, this application was tested in 2 cases

which are Connection & Driver Manager and Transaction. In function InsertProduct, there is no close() syntax for its connection. Meanwhile in transaction case, the Bedhigasan Application cannot detects the rollback() syntax. In Type G, the Bedhigasan Application is also has a false positives, which is the function openJDBCMySqlConnection didn't have any close syntax. This one is supposed to be ok.

Type H:

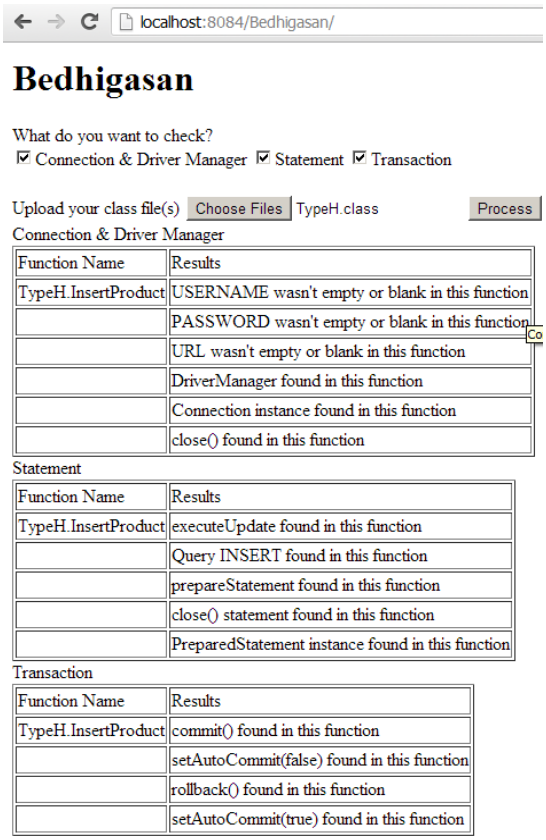

Figure 4.8. Testing Result of Type H

In type H which appear in figure 4.8, Bedhigasan application didn't catch any bugs pattern, which mean the bedhigasan application didn't have any false positives and false negatives in this type, because there is no bugs in this type.

The next test will be applied to FindBugs application. The same java file and environment are applied in this test as Bedhigasan tested.

Type A:

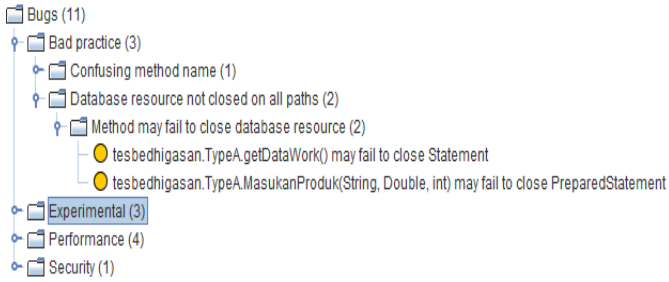

Figure 4.9. Testing Result of Type A

In Figure 4.9, it is proven that the findbugs application is able to detects the bugs pattern as good as bedhigasan application. But in this case, Findbugs has a false negative. The false negatives appears in Connection & Driver Manager case, Findbugs didn't catch any bugs pattern. In MasukanProduct function, there is closeJDBCMySqlConnection, this function is responsible for closing the database connection. In MasukanProduct function, they called the closeJDBCMySqlConnection() function, but Findbugs just guessing, if there is any function contains with close() syntax it means the programmer will call it each time they want to close it, but it didn't happen here. So that writter can conclude that, Findbugs just detects in the file, is there any close() syntax, if its exists than findbugs will not raise any bugs issues. This is called false negatives. This is happen because findbugs didn't use non-linear method, so the entire flow process of program is not analyze nicely. meanwhile in statement case, the findbugs application also able to detects as good as bedhigasan did. But Findbugs didn't catch any bugs pattern in transaction case.

Type B:

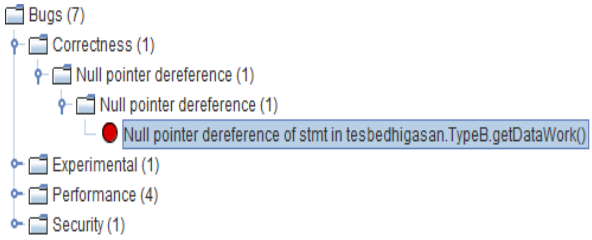

Figure 4.10. Testing Result of Type B

In Figure 4.10 it is proven that findbugs application is also able to detects the bugs pattern for statement case. In getDataWork() function, there is no createStatemnet syntax, but findbugs didn't able to detects the bugs pattern for empty USERNAME, maybe they assume that USERNAME might be empty, but the Bedhigasan application didn't agree with that folklore. So that bedhigasan applicataion will raise bugs pattern issue if USERNAME is empty.

Type C:

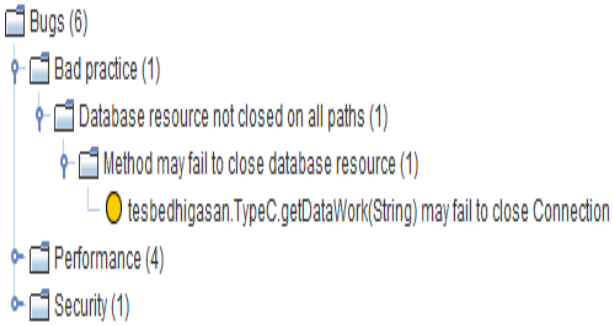

Figure 4.11. Testing Result of Type C

In Figure 4.11 the Bedhigasan application also able to detects bugs pattern for Connection and Driver Manager cases. But findbugs application didn't accurately detects all bugs, becuase actually in getDataWork function the connection syntax didn't called. But again Findbugs just looking for syntax getConnection and there is no close() syntax and they will raise issue. Different with the bedhigasan application, which will raise issue

if there is no connection. For close() syntax bedhigasan and findbugs both were did a good job to detects the bugs pattern on it.

Type D:

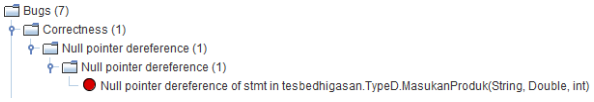

Figure 4.12. Testing Result of Type D

In Figure 4.12 the Findbugs application did a good job as bedhigasan did. Both bugs were abled to detected which is bugs pattern for statement case and in MasukanProduk function there is no PrepareStatement() syntax.

Type E:

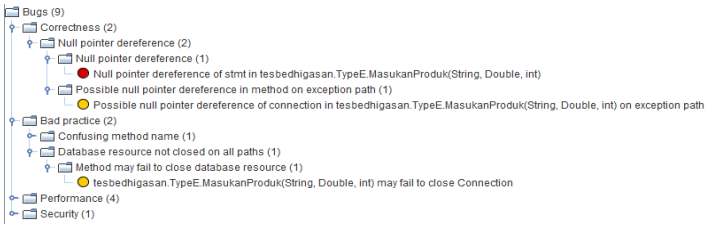

Figure 4.13. Testing Result of Type E

In Figure 4.13 the Findbugs application is also able to detects the bugs pattern in MasukanProduk function. But Findbugs didn't able to detects bugs pattern in transaction case. In MasukanProduk function, setAutoCommit(false) was not called here. This is also a false negative from Findbugs application. Maybe findbugs will update their engine in upcomming version.

Type F:

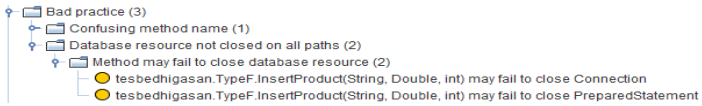

Figure 4.14. Testing Result of Type F

In Figure 4.14 Findbugs application is also able detects the bugs pattern as good as bedhigasan did. In this function there is no close() syntax for statement case and also close() syntax for connection case. But again Findbugs didn't able to detects bugs pattern in transaction case.

Type G:

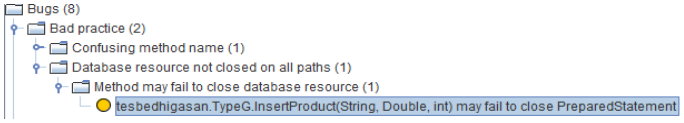

Figure 4.15. Testing Result of Type G

In Figure 4.15 it is seen that findbugs application is also able to detects as good as bedhigasan in InsertProduct function. In InsertProduct function, there is no close() syntax

in statement case. Unfortunetly, as happen before, findbugs didn't able to detects bugs pattern in transaction case.

Type H:

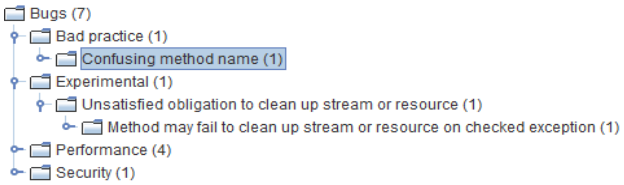

Figure 4.16. Testing Result of Type H

In Figure 4.16 the findbugs application is also didn't detected any bugs pattern in JDBC. But again findbugs didn't catch any bugs pattern in transaction case.

4.2. Analyzing Results of the Test

Bedhigasan:

$$Fn = \frac{\sum_{i=1}^{8}(Fn_i c)}{\sum_{i=1}^{8}(B_i c)} x\ 100\%$$

$$Fn = \frac{1x4 + 1x3 + 1x3 + 1x3 + 1x2 + 1x2 + 1x2 + 1x1}{1x4 + 3x3 + 3x2 + 1x1} x\ 100\%$$

$$\boldsymbol{Fn = 100\%}$$

The value of false negatives here showing that the Bedhigasan application is really accurate in analyzing of bugs pattern. This happen because the file which tested in this paper is also used as fundamental when making this application.

$$Fp = \frac{\sum_{i=1}^{8}(Fp_i c)}{\sum_{i=1}^{8}(B_i c)} x\ 100\%$$

$$Fp = \frac{0x4 + 1x3 + 1x3 + 0x3 + 1x2 + 1x2 + 1x2 + 1x1}{1x4 + 3x3 + 3x2 + 1x1} x\ 100\%$$

$$\boldsymbol{Fp = 65\%}$$

The value of false postives here showing that bedhigasan quite good accurately in analyzing bugs pattern which should not have to be reported as bugs. Unfortunetely this may cause bedhigasan became overpower in analyzing bugs pattern, so some parts which didn't need to be reported are reported as bugs.

Findbugs:

$$Fn = \frac{\sum_{i=1}^{8}(Fn_i c)}{\sum_{i=1}^{8}(B_i c)} x\ 100\%$$

$$Fn = \frac{1x4 + 1x3 + 0x3 + 0x3 + 1x2 + 0x2 + 0x2 + 0x1}{1x4 + 3x3 + 3x2 + 1x1} x\ 100\%$$

$$\boldsymbol{Fn = 45\%}$$

From the false negatives calculation seen above, it can be conclude that Findbugs is not quite good enough in finding bugs pattern, because they not trying to understand the flow process in the java code. Findbugs just looking for any syntax needed in the code, if the syntax exists then findbugs will not raise any issue. Beside that findbugs is not capable in analyzing the transaction case, so this will caused a big impact for their false negatives.

$$Fp = \frac{\sum_{i=1}^{8}(Fp_i c)}{\sum_{i=1}^{8}(B_i c)} x\ 100\%$$

$$Fp = \frac{1x4 + 1x3 + 1x3 + 1x3 + 1x2 + 1x2 + 0x2 + 1x1}{1x4 + 3x3 + 3x2 + 1x1} x\ 100\%$$

$$\boldsymbol{Fp = 90\%}$$

From the false positives calculation seen above, it can be conclude that findbugs is really good in muting some aspect which should not raised as an issue, but this may caused worser performance than bedhigasan in finding bugs pattern.

5. Conclusion

As seen in the analyzing of the results above, it can be conclude that using static analyzer with non linear code scan will reduce false negatives, but unfortunetly it will increase the false positives. It means with this method will increase the level of awareness in application. Bedhigasan application is the way far from perfect, this method should be improved in reducing false positives. Because the method it self will increase the awarenes of bugs, this application needs more validation when analyzing. So that in the future, this application will also have better false positives.

References

[1] J. Shadrach and L. G. Ford, “Bugs Pattern Detection in JDBC using Static Analysis Non-Linear Method”, SERSC, **(2013)**.
[2] C. Brian, H. Daniel, H. David, L. Reuven, P. William and S. Kristin, “Improving Your Software Using Static Analysis to Find Bugs”, OOPSLA, **(2006)**.
[3] S. Brett, “Pro Java Programming”, 2nd, Apress, New York, **(2005)**.
[4] H. David, P. William and S. Jaime, “Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs. PASTE, **(2005)**.

Appendix

A

```
public class TypeA {

   private final String url = "jdbc:mysql://localhost:3306/";
   private final String userName = "shadrach";
   private final String password = "J@bonir89";
   private final String dbName = "java_training";

   public Connection openJDBCMySQLConnection() {
      Connection connection = null;
      try {
         connection = DriverManager.getConnection(url + dbName, userName, password);
      } catch (Exception e) {
         e.printStackTrace();
      }
      return connection;
   }

   public void closeJDBCMySQLConnection(Connection connection) {
```

```
        if (connection != null) {
            try {
                if (!connection.isClosed()) {
                    connection.close();
                }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    public ArrayList<HashMap<String, Object>> getDataWork() {
        ArrayList<HashMap<String, Object>> data = null;
        String query = "SELECT * FROM PRODUCT";
        try {
            Connection connection = openJDBCMySQLConnection();
            data = new ArrayList<>();
            Statement stmt = connection.createStatement();
            ResultSet result = stmt.executeQuery(query);
            ResultSetMetaData rsMetaData = result.getMetaData();
            HashMap<String, Object> map;
            while (result.next()) {
                map = new HashMap<>();
                for (int i = 1; i <= rsMetaData.getColumnCount(); i++) {
                    map.put(rsMetaData.getColumnName(i), result.getObject(rsMetaData.getColumnName(i)));
                }
                data.add(map);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return data;
    }

    public int MasukanProduk(String name, Double price, int qtys) throws SQLException {
        int rowCount = 0;
        String template = "INSERT INTO PRODUCT (NAME,PRICE,QTY) values (?,?,?)";
        try {
            Connection connection = openJDBCMySQLConnection();
            PreparedStatement stmt = connection.prepareStatement(template);
            connection.setAutoCommit(false);
            stmt.setString(1, name);
            stmt.setDouble(2, price.doubleValue());
            stmt.setInt(3, qtys);
            rowCount += stmt.executeUpdate();
            connection.commit();
            connection.setAutoCommit(true);
            closeJDBCMySQLConnection(connection);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return rowCount;
    }
}
```

B

```
public class TypeB {

    private final String url = "jdbc:mysql://localhost:3306/";
    private final String userName = "shadrach";
    private final String password = "J@bonir89";
    private final String dbName = "java_training";

    public ArrayList<HashMap<String, Object>> getDataWork() {
        ArrayList<HashMap<String, Object>> data = null;
        String query = "SELECT * FROM PRODUCT";
```



```
        try {
            Connection connection = DriverManager.getConnection(url + dbName, "", password);
            data = new ArrayList<>();
            Statement stmt = null;
            ResultSet result = stmt.executeQuery(query);
            ResultSetMetaData rsMetaData = result.getMetaData();
            HashMap<String, Object> map;
            while (result.next()) {
                map = new HashMap<>();
                for (int i = 1; i <= rsMetaData.getColumnCount(); i++) {
                    map.put(rsMetaData.getColumnName(i), result.getObject(rsMetaData.getColumnName(i)));
                }
                data.add(map);
            }
            stmt.close();
            connection.close();
        } catch (SQLException e) {
            System.out.println("Error in getting data for query : " + query);
        }
        return data;
    }
}
```

C

```
public class TypeC {

    private final String url = "jdbc:mysql://localhost:3306/";
    private final String userName = "shadrach";
    private final String password = "J@bonir89";
    private final String dbName = "java_training";

    public ArrayList<HashMap<String, Object>> getDataWork(String query) {
        ArrayList<HashMap<String, Object>> data = null;
        try {
            DriverManager.getConnection(url + dbName, userName, password);

        } catch (SQLException e) {
            System.out.println("Error in getting data for query : " + query);
        }
        return data;
    }
}
```

D

```
public class TypeD {

    private final String url = "jdbc:mysql://localhost:3306/";
    private final String userName = "shadrach";
    private final String password = "J@bonir89";
    private final String dbName = "java_training";

    public int MasukanProduk(String name, Double price, int qtys) throws SQLException {
        int rowCount = 0;
        String template = "INSERT INTO PRODUCT (NAME,PRICE,QTY) values (?,?,?)";
        try {
            Connection connection = DriverManager.getConnection(url + dbName, userName, password);
            PreparedStatement stmt = null;
            connection.setAutoCommit(false);
            stmt.setString(1, name);
            stmt.setDouble(2, price.doubleValue());
            stmt.setInt(3, qtys);
            rowCount += stmt.executeUpdate();
            stmt.close();
            connection.commit();
            connection.setAutoCommit(true);
            connection.close();
        } catch (Exception e) {
```

```
            e.printStackTrace();
        }
        return rowCount;
    }
}
```

E

```
public class TypeE {

    private final String url = "jdbc:mysql://localhost:3306/";
    private final String userName = "shadrach";
    private final String password = "J@bonir89";
    private final String dbName = "java_training";

    public int MasukanProduk(String name, Double price, int qtys) throws SQLException {
        int rowCount = 0;
        Connection connection = null;
        String template = "INSERT INTO PRODUCT (NAME,PRICE,QTY) values (?,?,?)";
        try {
            connection = DriverManager.getConnection(url + dbName, userName, password);
            PreparedStatement stmt = null;
            connection.setAutoCommit(false);
            stmt.setString(1, name);
            stmt.setDouble(2, price.doubleValue());
            stmt.setInt(3, qtys);
            rowCount += stmt.executeUpdate();
            connection.commit();
        } catch (Exception e) {
            connection.rollback();
            e.printStackTrace();
        }
        return rowCount;
    }
}
```

F

```
public class TypeF {

    private final String url = "jdbc:mysql://localhost:3306/";
    private final String userName = "shadrach";
    private final String password = "J@bonir89";
    private final String dbName = "java_training";

    public int InsertProduct(String name, Double price, int qty) throws SQLException {
        int rowCount = 0;
        Connection connection = null;
        String template = "INSERT INTO PRODUCT (NAME,PRICE,QTY) values (?,?,?)";

        try {
            connection = DriverManager.getConnection(url + dbName, userName, password);
            PreparedStatement stmt = connection.prepareStatement(template);
            stmt.setString(1, name);
            stmt.setDouble(2, price);
            stmt.setInt(3, qty);
            rowCount += stmt.executeUpdate();
            connection.commit();
            connection.setAutoCommit(true);

        } catch (Exception e) {
            connection.rollback();
            //e1.printStackTrace();
        }
        return rowCount;
    }
}
```

G

```
public class TypeG {
```



```
    private final String url = "jdbc:mysql://localhost:3306/";
    private final String userName = "shadrach";
    private final String password = "J@bonir89";
    private final String dbName = "java_training";

    public Connection openJDBCMySQLConnection() {
        Connection connection = null;
        try {
            connection = DriverManager.getConnection(url + dbName, userName, password);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return connection;
    }

    public void closeJDBCMySQLConnection(Connection connection) {
        if (connection != null) {
            try {
                if (!connection.isClosed()) {
                    connection.close();
                }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    public int InsertProduct(String name, Double price, int qty) {
        int rowCount = 0;
        String template = "INSERT INTO PRODUCT (NAME,PRICE,QTY) values (?,?,?)";
        try {
            Connection connection = openJDBCMySQLConnection();
            PreparedStatement stmt = connection.prepareStatement(template);
            connection.setAutoCommit(false);
            stmt.setString(1, name);
            stmt.setDouble(2, price);
            stmt.setInt(3, qty);
            rowCount += stmt.executeUpdate();
            connection.commit();
            connection.setAutoCommit(true);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return rowCount;
    }
}
```

H

```
public class TypeH {

    private final String url = "jdbc:mysql://localhost:3306/";
    private final String userName = "shadrach";
    private final String password = "J@bonir89";
    private final String dbName = "java_training";

    public int InsertProduct(String name, Double price, int qty) throws SQLException {
        int rowCount = 0;
        Connection connection = DriverManager.getConnection(url + dbName, userName, password);
        String template = "INSERT INTO PRODUCT (NAME,PRICE,QTY) values (?,?,?)";
        try {
            PreparedStatement stmt = connection.prepareStatement(template);
            connection.setAutoCommit(false);
            stmt.setString(1, name);
            stmt.setDouble(2, price);
            stmt.setInt(3, qty);
```

```
            rowCount += stmt.executeUpdate();
            connection.commit();
            connection.setAutoCommit(true);
            stmt.close();
            connection.close();
        } catch (Exception e) {
            connection.rollback();
            e.printStackTrace();
        }
        return rowCount;
}
```

Authors

Shadrach Jabonir, received his bachelor in Information Technology-Software Engineering from Bina Nusantara University, Jakarta, Indonesia, 2011. He is Specialist in Java Programming, Software Architecture and Software Engineering.

Ford Lumban Gaol