Developing an Automatic Tool for Visualizing Source Code against Bad Smell Patterns

Jihoon Park[a,*], Hyun Seung Son[b], R. Youngchul Kim[c]
SE Lab , Hongik University, Sejong Campus, Korea
E-mail address: {[a] pjh, [b] son, [c] bob}@selab.hongik.ac.kr

Abstract

Today, software is expanding its influence on everyday life as well as in various industries. However, although it has been growing our software industry, the perception of software quality is still insufficient. Our domestic software ventures & small businesses have lacked investment in maintenance due to mainly focused on implementation. Actually software can not completely constructed at the development phase. Therefore, how to do refactoring them very well? In this paper, we suggest to implement a tool to automatically visualize the code through reverse engineering. This tool uses to extract the information with SQL Program to find bad smell patterns within the source code. And that can also find how much complexity be there in the code. With this approach to visualize the code inner structure, developers can easily do refactoring which parts of its source codes by themselves. Hopefully, it may improve the quality of the software with our visualization tool.

Keywords: refactoring, bad smell, complexity, static analysis, visualization

1. Introduction

The software market scales are growing rapidly in these days. In fact, the global software market in the entire IT industry is worth more than $ 1 trillion, accounting for 40 % of the total IT industry [1]. Now the software is now expanding its influence to include air conditioners, TVs, refrigerators, electrical appliances as well as various industries. With the emphasis on the importance of software in such diverse fields, it is now necessary to create a way to produce high-quality software [2]. This is difficult to solve even if software is developed in software engineering. However, our domestic venture & small businesses are even more difficult in many ways such as manpower, tools, and expenses. Moreover, we don't know the inside of the software since it is mainly focused on implementation [3]. Due to the invisible of software, it may be difficult for administrators and developers to manage software quality and maintenance. In this paper, we use our refined parser based on the original Java parser for that the source code is analyzed, and then show a code visualization. Java parser transforms information from source code into an AST structure [4], which converts to the AST structure, and stores all parsed data into the database. We can extract bad smell patterns with SQL query from the stored data [5], and also create the code visualization

graph using extracted data for refactoring. We can check the Cyclomatic Complexity [6] by visualizing the source code as a visual graph [7]. Using this tool, an administrator or developer can make a decision which parts of the source code should do refactoring. We can look forward to improving the quality of the software through refactoring.

In this paper, Chapter 2 describes how to supplement the shortcomings of the previous researches. Chapter 3 just mentions eleven patterns of the bad smell patterns [8]. Chapter 4 describes code complexity metrics. Chapter 5 describes a whole process of the automatic visualization tool. Last Chapter is the conclusions and future work.

2. Analysis with the previous researches

The previous researches have been implemented automated visualization tools based on open-source tools called Source Navigator [9]. Our previous research was limited by the source code extracted by the source navigator [10,11]. We replace the parsing tool with the Java Parser [12] instead of the Source Navigator. With Java Parser, we parse the source code into an AST structure [13]. One problem is that Java Parser does not create a separate database dislike the Source navigator. Thus, this paper has established a new database for use

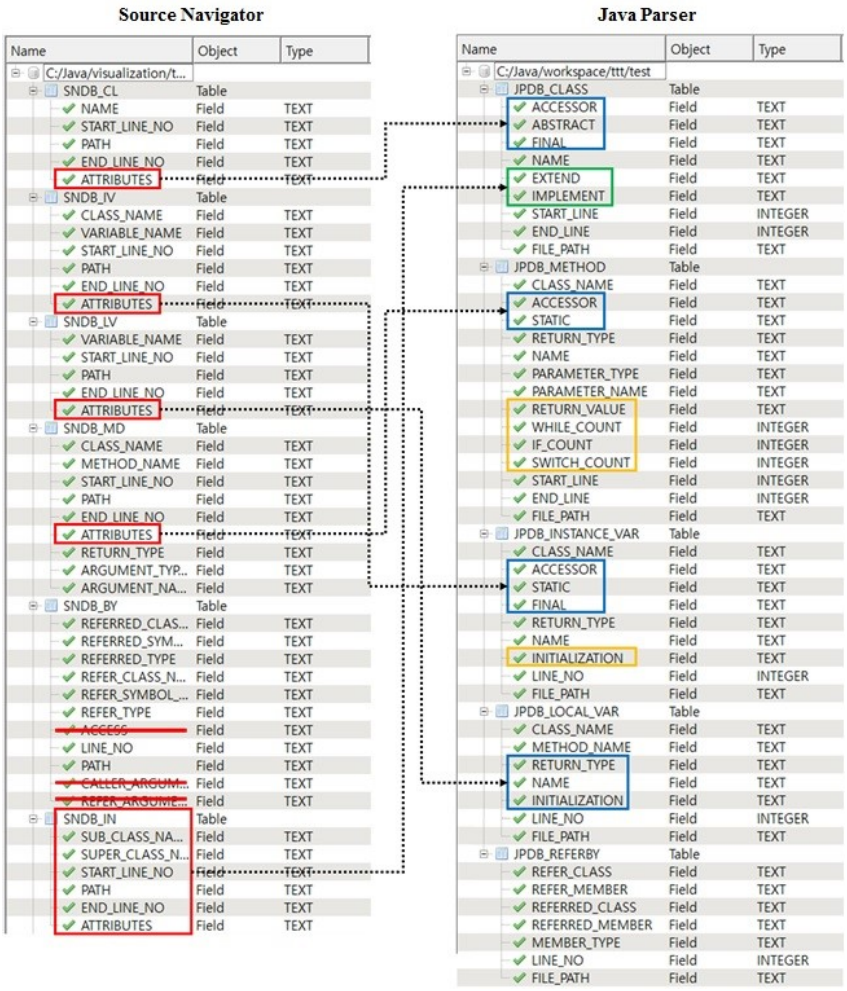

Fig. 1. Comparison between Source Navigator DB and Java Parser DB

in existing tools. Fig. 1 shows a comparison of the database between the Source Navigator and the Java Parser. The left side of figure 2 is the information table of the source code extracted by the existing Source Navigator. The right side of figure 2 is the new information table of the source code extracted by the Java Parser. The Source Navigator makes six tables that show class information, instance variable information, local variable information, method information, call information, and inherited information. The Java Parser makes five tables that show class information, method information, local variable information, instance variable information, and call information. In the content of the SNDB table, each ATTRIBUTES is composed of a difficult letter, such as 0x0, 0x1, and 0x2. As a result, the database of Java Parser is divided into detailed entries to enhance readability of database values. The SNDB table is stored the inherited information of the source code. However, the SNDB_IN is included in JPDB_CLASS because it contains duplicate content. The newly added value in JPDB_METHOD is return_value, while_count, if_count, switch_count. Additionally, Initialization is added to JPDB_LOCAL_VAR and JPDB_INSTNACE_VAR. The information is not extracted from SNDB, but Java Parser can extracts this. Using these tips, we can find bad smell pattern of source code that we can't find in our previous SNDB.

3. Bad Smell Patterns

Bad Smell is a metaphor first written in Martin Fowler's book 'refactoring'. Bad Smell defines the moment when refactoring is needed in your code. The code visualization is a way to make software easier to understand, less expensive to modify, and to change the internal structure without any visible change in behavior [5]. We define to extract the Bad Smell pattern from source codes. After identifying the Bad smell pattern within source code, we can encourage Refactoring.

3.1 Long Parameter List

It is a Long Parameter List if method has a lot of parameters. Fig. 2 shows the extraction program code. In the extracted program code, store the value of PARAMETER_TYPE in param of the JPDB_METHOD table. Then split param by "," and measure the number of parameters of the method. Then it is determined as a long parameter list according to the count value of the If Statement.

```
ResultSet lpl = statement.excuteQuery("select PARAMETER_TYPE from JPDB_METHOD
                                                                    where CLASS_NAME = [name]");
while (lpl.next()) {
    String param = lpl.getString("PARAMETER_TYPE");
    String[] splitParam = param.split(",");
    if (splitParam.length > count) {
        return true;
    }
}
```

Fig. 2. Long Parameter List extraction-program code

3.2 Feature Envy

The Feature Envy is when you call several get methods of another object to use a certain value. Fig. 3 shows the extracted program code. In the extracted program code, REFERRED_MEMBER in the JPDB_REFERBY table starts with get, and REFERRED_CLASS counts the number of things like name in JPDB_CLASS. If the number is more than a count, it is determined as Feature Envy.

```
ResultSet lpl = statement.excuteQuery("select count(*) as CNT from JPDB_REFERBY
                                        where REFERRED_MEMBER like 'get%' and REFERRED_CLASS = [name]");
while (fe.next()) {
    int cnt = Integer.parseInt(fe.getString("CNT"));
    if (cnt > count) {
        return true;
    }
}
```

Fig. 3. Feature Envy extraction-program code

3.3 Message Chains

The message chain occurs when a client requests another object, then that object requests yet another one, and so on. Fig. 4 shows the extracted program code. In the code, checkMC uses the cnt value of the reGetMC method to return a boolean value. The recursive method reGetMC method continues to call itself and return cnt. Cnt is the number of times an object is passed to another object. If this count exceeds a count, it is determined as Message Chains.

```
public int reGetMC(String rfmn, int cnt) {
    ResultSet mc = statement.excuteQuery("select REFERRED_MEMBER, REFER_MEMBER
     from JPDB_REFERBY where REFERRED_MEMBER like 'get%' and REFER_MEMBER like 'get%'
                                        and REFER_MEMBER='" + rfmn + "'and REFER_CLASS=[name]");
    while (mc.next()) {
        String rdm = mc.getString("REFERRED_MEMBER");
        cnt = reGetMC(rdm, cnt);
    }
}
public Boolean checkMC() {
    ResultSet lpl = statement.excuteQuery("select REFER_MEMBER from JPDB_REFERBY
                                                                    where REFERRED_CLASS = [name]");
    while (mc.next()) {
        String rfmn = mc.getString("REFER_MEMBER);
        int cnt = reGetMC(rfmn, 0);
        if (cnt > count) {
            return true;
        }
    }
}
```

Fig. 4. Message Chains extraction-program code

3.4 Middle Man

This smell can be Middle Man if most of a method's classes delegate to another class. In other cases, it can be the result of overzealous elimination of Message Chains. The class remains as an empty shell that does not do anything other than delegate. Fig. 5 shows the extracted program code. In the extracted

program code, RETURN_TYPE in the JPDB_METHOD table like REFERRED_CLASS in the JPDB_REFERBY table and line of method less than 3 counts the number of things. In a class, if this count exceeds a count, it is determined as Middle Man.

```
ResultSet mm = statement.excuteQuery("select count(*) as CNT from JPDB_METHOD
                                        where END_LINE – START_LINE > 3 and CLASS_CLASS = [name]");
while (mm.next()) {
    int cnt = Integer.parseInt(mm.getString("CNT"));
    if (cnt > count) {
        return true;
    }
}
```

Fig. 5. Middle Man extraction-program code

3.5 Lazy Class

The Lazy Class is one when all methods in a class are not executed. Fig. 6 shows the extracted program code. In the extracted program code, lzc1 extracts REFERRED_MEMBER from the JPDB_REFERBY table without duplicates. And lzc2 extracts NAME from the JPDB_METHOD table. Lzc1 is the information of the method invoked throughout the program. Lzc2 is method information in the class. If the results of lzc1 and lzc2 are not the same, it is called Lazy Class because there exists methods that are

```
ResultSet lzc1 = statement.excuteQuery("select distinct REFERRED_MEMBER from JPDB_REFERBY
                                        where REFERRED_CLASS = [name]");
ResultSet lzc2 = statement.excuteQuery("select NAME from JPDB_METHOD
                                        where CLASS_NAME = [name]");
```

Fig. 6. Lazy Class extraction-program code

not called.

3.6 Data Class

The Data Class is that the names of the methods invoked in the class begin with get- and set-. Fig. 7 shows the extracted program code. In the extracted program code, Dc1 extracts REFERRED_MEMBER from JPDB_REFERBY table. And dc2 extracts REFERRED_MEMBER from the JPDB_REFERBY table, starting with get- or set-. Dc1 and dc2 are compared. If the values match, the data class is determined.

```
ResultSet dc1 = statement.excuteQuery("select REFERRED_MEMBER from JPDB_REFERBY
                                        where REFERRED_CLASS = [name]");
ResultSet dc2 = statement.excuteQuery("select REFERRED_MEMBER from JPDB_REFERBY
                        where REFERRED_MEMBER like 'get%' or where REFERRED_MEMBER like 'set%'");
```

Fig. 7. Data Class extraction-program code

3.7 Large Class

Large Class is a case where a lot of instances of the class variable. Fig. 8 shows the extracted program code. In the extracted code, the number of variables in the class is counted

in the JPDB_INSTANCE_VAR table. If the number is more than a certain number, it is determined as a large class.

```
ResultSet lgc = statement.excuteQuery("select count(*) as CNT from JPDB_INSTANCE_VAR
                                                                    where CLASS_NAME = name]");
while (lgc.next()) {
    if (cnt > count) {
        return true;
    }
}
```

Fig. 8. Large Class extraction-program code

3.8 Refused Bequest

The Refused Bequest is one when you do not use all the information of the inherited class. Fig. 9 shows the extracted program code. In the extracted code, extract NAME and EXTEND from JPDB_CLASS table. And REFER_CLASS_NAME is equal to subName extracted from rb1, and REFERRED_CLASS like REFER_CLASS are saving. Rb2 stores the methods of the superName class. If the values of Rb1 and rb2 are not equal, it is judged to be Refused Bequest because all inherited methods are not used.

```
ResultSet rb = statement.excuteQuery("select NAME, EXTEND from JPDB_CLASS");
while (rb.next()) {
    String subName = rb.getString("NAME");
    String superName = rb.getString("EXTEND");
    ResultSet rb1 = statement.excuteQuery("select distinct REFERRED_MEMBER from JPDB_REFERBY
  where REFER_CLASS = '" + subName + "' and REFERRED_CLASS = REFER_CLASS order by asc");
    ResultSet rb2 = statement.excuteQuery("select NAME from JPDB_METHOD
                                                    where NAME = '" + superName + "' order by asc");
}
```

Fig. 9. Refused Bequest extraction-program code

3.9 Comments

The Comments are cases where there are too many comments in the code. Fig. 10 shows the extraction program code. In the code, extract COMMENT from the JPDB_METHOD table. If the number of extracted COMMENT is more than a count, COMMENTS is judged.

```
ResultSet cmt = statement.excuteQuery("select COMMENT from JPDB_METHOD
                                                    where REFERRED_CLASS = [name]");
while (cmt.next()) {
    int cnt = Interger.parseInt(cmt.getstring("COMMENT"));
    if (cnt > count) {
        return true;
    }
}
```

Fig. 10. Comments extraction-program code

3.10 Switch Statements

Switch Statements are cases where there are many Switch statements. Switch Statements are statements where there are many Switch statements. This can be solved using polymorphism. Fig. 11

shows the extraction code. In the extraction code, extract SWITCH_COUNT from the JPDB_METHOD table. If this cnt is more than the set count, it is judged as Switch Statements.

```
ResultSet ss = statement.excuteQuery("select SWITCH_COUNT from JPDB_METHOD
                                                    where REFERRED_CLASS = [name]");
while (ss.next()) {
    int cnt = Interger.parseInt(ss.getstring("COMMENT"));
    if (cnt > count) {
        return true;
    }
}
```

Fig. 11. Switch Statements extraction-program code

3.11 Temporary Field

The Temporary Field is when an instance variable in an object is set only under certain circumstances. Fig. 12 shows the extraction code. In the extraction code, we extract IF_COUNT from the JPDB_METHOD table, where NAME starts with set-. If IF_COUNT is greater than 0, it is judged to be a temporary field because the instance variable may not be set according to the condition.

```
ResultSet tf = statement.excuteQuery("select IF_COUNT from JPDB_METHOD
                                                    where NAME like 'set%' and CLASS_NAME = [name]");
while (tf.next()) {
    int cnt =tf.getstring("IF_COUNT");
    if (cnt > count) {
        return true;
    }
}
```

Fig. 12. Temporary Field extraction-program code

4. Code Complexity Metrics

The Complexity refers to the measurement of internal relations between conditions within a software component. Ultimately, these metrics help identify whether the system is a potential stress point or not. Component stress complexity is related to cyclomatic complexity. Fig.13 shows the method in Decision Tree. In the figure, the nodes are divided into several nodes. In the figure, there are three nodes. A For Statement node is blue, If Statement node is yellow, and a Statement node is white. Each branch node has a condition. Depending on the condition, it is divided into true and false. The Statement node is a bundle of all statements between branches. The equation for calculating the cyclomatic complexity is (1).

$$V(G) = E - N + 2 \quad (1)$$

V(G) is the value of cyclomatic complexity, E is the number of Edges, N is the number of Nodes. In general, cyclomatic complexity criteria define complexity as being difficult to manage if the complexity is greater than 10. Other studies have defined complexity to be less than 15 in one function. Microsoft's MSDN says that the complexity should not exceed 25.

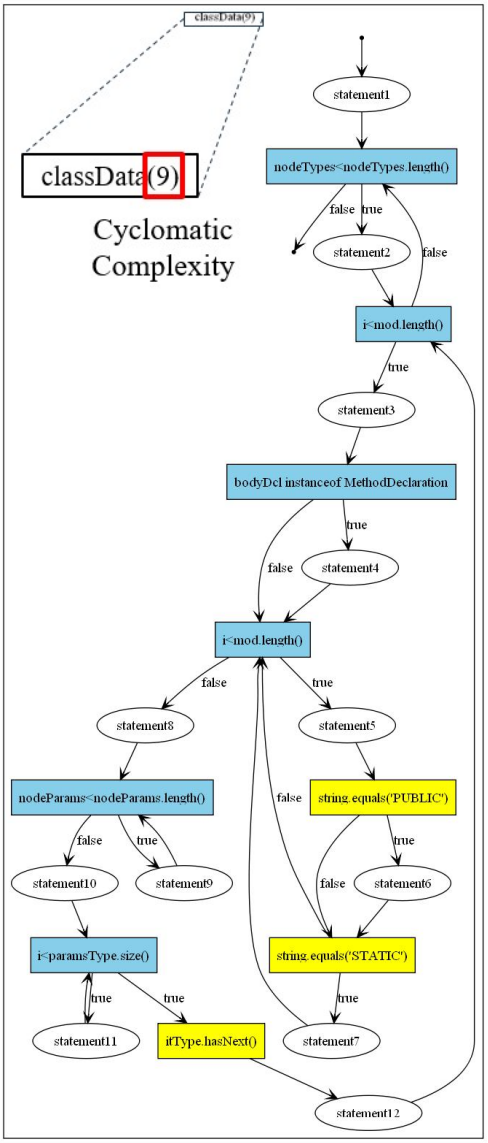

Fig. 13. The Cyclomatic Complexity of a visual graph

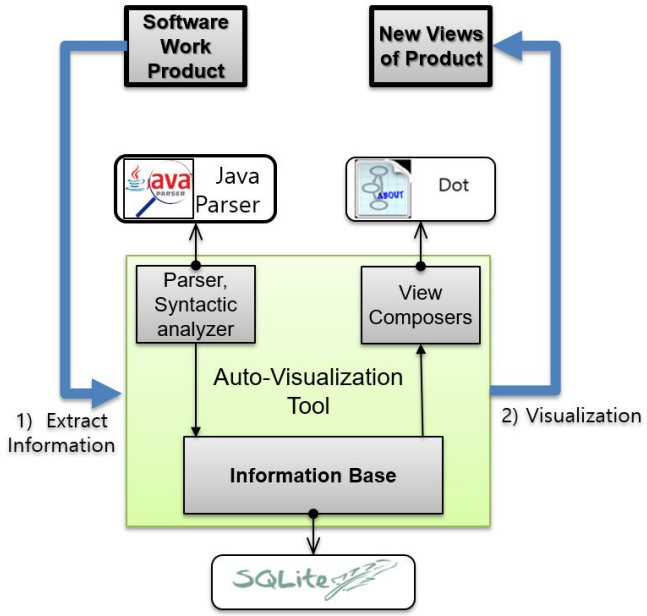

Fig. 14. A whole process of a Visualization tool

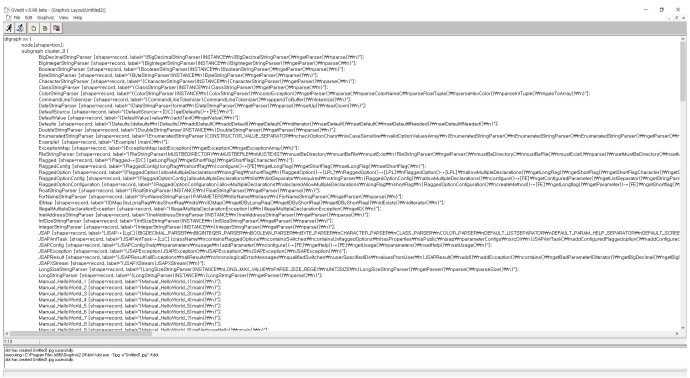

Fig. 15. Auto-generated DotScript language.

5. An Automatic Code Visualization

In this, we show the process of the Visualization tools mentioned in figure 14. Fig. 14 shows a diagram of the visualization tool. First you need the source code written in Java. The source code is analyzed by the parser. We use the open source Java Parser refined for the parser. The analyzed information is stored in a database. The database was SQLite. The structure of the database had described in figure 1. Then draw a graph from Graphviz [14] using the analyzed source code information stored in the database. However, Graphviz requires a new DotScript language. Fig. 15 shows the DotScript language automatically generated by the tool. As a result, the tool of this paper automates analysis, storage, and graphing functions as one tool. Fig. 16 shows the extraction of bad smell patterns within the code as the output. The [FE] is marked in the configure() method of the FlaggedOptionConfig class. If you go

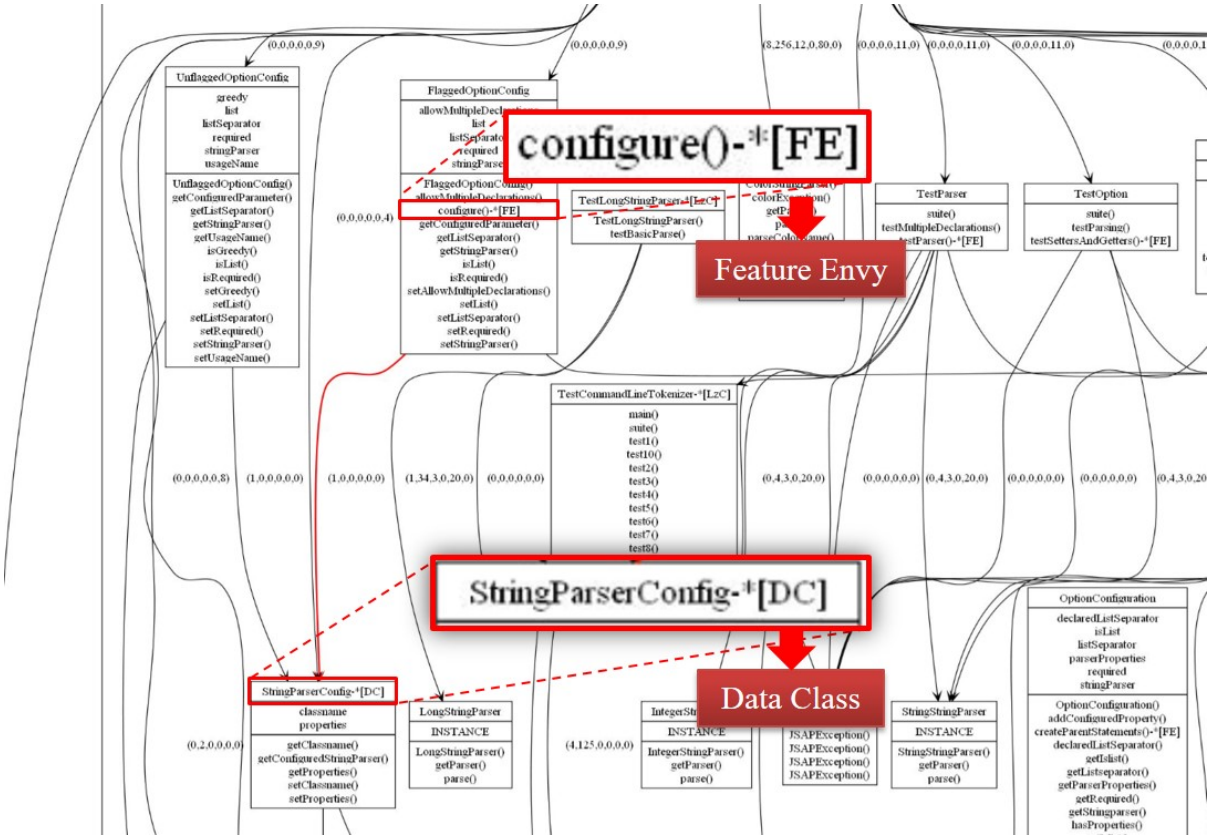

Fig. 16. Extracting Bed smalls In Code visualization

along the call arrow, it points to the StringParserConfig class. The [DC] is marked in the StringParserConfig class. Because the methods names are all start with get- and set-. Since the Feature Envy corresponds to calling the get- and set- methods more than a certain number of times, the configure() method of the FlaggedOptionConfig class is likely to be Feature Envy. Therefore, WE can judge whether a bad smell between two classes will require refactoring or not.

6. Conclusion & Future Work

This paper suggests how to extract a Cyclomatic Complexity of McCabe and show Bad Smell of Martin Fowler in source code based on reverse engineering. The visualization of the source code will give the motive of refactoring and correct the bad habits of the advanced developers. Therefore, we can refactoring ourselves to lower code complexity. We expect to be able to reduce the maintenance cost of software by knowing the complexity of the source code. We will also extract remaining Bad Smell in future works, which will also find and visualize various software quality indicators.

Acknowledgments.

This work was supported by the Human Resource Training Program for Regional Innovation and Creativity through the Ministry of Education and National Research Foundation of Korea (NRF-2015H1C1A1035548).

7. References

[1] NIPA, (2013). SW Quality Management Manual(SW Visualization)

[2] Changjae Kim, Jeawon Park, (2014). A Software Maintenance Capability Maturity Model Based on Service. Journal of KIIT, 12(5), 173-184. Doi:10.14801

[3] Jihyuk Kim, Changjae Kim, Sungyul Ryu. (2009). A Method for Establishment of Case-based Software Maintenance Maturity Model. Journal of KIIT, 36(9), 718-731

[4] ASTViewer, http://www.eclipse.org/jdt/ui/astview/

[5] SQLite, About SQLite, http://www.sqlite.org/about.html

[6] Byungju Hwang, Jinyoung Choi, (2013). A Research of improving the way of software source codes quality using Mccabe Cyclomatic Complexity of Software Metrics, KIIT 2013, 6, 535-537.

[7] Dongsu Seo. (2016). Definition of Security Metrics for Software Security-enhanced Development. KSII 2016, 17(4), 76-86

[8] Martin Fowler. (2002). *Refactoring: Improving The Design of Existing Code*. Addison-Wesley.

[9] SN Group, Source Navigator User's Guide, http://www.sqlite.org /about.html

[10] Jihoon Park, Hyunseong Son, Bokyung Park, Jinhyub Lee, R. Youngchul Kim, (2017). Tailoring an Automatic Priority based on quality metrics of an organization fo Effective Refactoring, KCSE 2017, 19(1), 175-178.

[11] Jihoon Park, Woosung Jang, Jinhyub Lee, Hyunseong Son, R. Youngchul Kim, (2016). Implementing the automatic identification of the Bad Smell Coding Patterns, KIISE 2016, 401-403.

[12] Java Parser, Getting started, http://javaparser.org/gettingstarted.html

[13] Kyle Simpson, *You Don't know JS SCOPE & CLOSURES*. Hanbit's Book.

[14] GV team, GraphViz User's Guide, http://www.graphviz.org/Documentation.php