Improving source code against bad-smell code patterns

Jihoon Park, Byungkook Jeon*, R. Young Chul Kim and Hyun Seung Son**

SE Lab., Hongik University, Sejong Campus, 30016, Korea
*Dept. of Software, Gangneung-Wonju National University, Wonju City, Gangwon Prov. 26403, Korea
**Dept. of Reliability Technology Institute, Moasoft, Seoul, 05770, Korea.

Abstract: Currently, in many areas of industry, there has been an increasing emphasis placed on software. Software in the industrial world may encounter frequent changes in requirements at any stage of software lifecycle. These frequent changes often lead to increase design complexity, which results in lower quality software against the original design goals. To solve this problem, we propose a method for improving software design through refactoring based on reverse engineering. The proposed refactoring approach can visually identify bad-smell patterns in source code. We expect that our method can improve software design through refactoring, even in the presence of frequently changing requirements.

Keywords: Refactoring, Bad Smell, Code Visualization, Reverse Engineering, Java Parser

1 Introduction

Currently, as the use of a software in industry is expanding, various new user requirements are emerging. Additionally, in completed software programs require maintenance over time. In real world environments, software is constantly evolving to improve [1]. As software is modified and improved to meet new requirements, code gradually becomes more complex. This leads to situations where software deviates from its original design goals and declines in quality. Refactoring is a method that can be utilized to reduce this type of software complexity [2]. However, selecting the proper areas for refactoring is a difficult problem for developers.

In this study, we utilized reverse engineering to visualize the portions of source code that require refactoring [2, 3, 4, 5]. We utilized Martin Fowler's bad-smell method [6] for finding the portions of code that require refactoring. Developers can improve their code design based on the visualized source code. Small and midsize companies in Korea have limited personnel and finances for software development organization, so our goal is to create an automated visualization system with open-source software. The proposed visualization system consists of a parser that analyzes source code, a database that stores analyzed data, and tool that utilizes the analyzed data to draw graphs. As a result, we expect that software quality and maintainability will improve.

This remainder of this paper is organized as follows. Chapter 2 describes the software visualization process and refactoring methods in related studies. Chapter 3 describes the proposed visualization system. Chapter 4 describes the visualization of bad-smell patterns and how to improve the results obtained from refactoring. Chapter 5 presents a visualization case study utilizing an open-source program. Finally, Chapter 6 presents our conclusions and discusses future research.
2 Related Works

The visualization process is a software engineering technique for resolving difficulties in software development processes, which is gradually increasing in scale. Good software may be developed if the entire software development process is well managed. However, managing software development is difficult for small and midsized businesses in Korea because of their insufficient personnel, finances, and lack of software development experience. The visualization process minimizes the management burden through the visualization and documentation of software. It helps them to manage their software quality more easily. Visualization allows the overall development process to be understood by compensating for the non-visual nature of software. Documentation improves internal personnel’s understanding of a project and aids in decision making. This facilitates coordinated configuration management, including the management of software requirements, analysis/design, implementation, testing, and maintenance. In this manner, software can be improved to meet the quality standards of customers.

Generally, the software development stages are performed in the following order: requirements, design, implementation, testing, and maintenance. This means that the design must be completed before being implementation. If the design is completed before programming, maintenance costs as a result of reworking can be avoided. However, in software, it is unavoidable for requirements to change during the life of a product. Therefore, developers must create flexible designs for changes that can be predicted. However, flexible solutions are more complex than simple solutions, meaning the cost of flexibility is high [3]. If refactoring is considered during the initial design process, the aforementioned design costs can be minimized. In fact, no additional design based on future requirements is necessary. One can write code that allows the software to operate properly as needs arise and later perform refactoring to streamline the software. This lessens the burden of having to create a flexible from the beginning. Therefore, there is no need for developers to create a perfect design prior to implementation.

Mauricio A. Saca [13] mentions that refactoring makes the design process much easier, which reduces money spent on design. However, refactoring does not always have a positive effect on the performance of a program. Refactoring is the process of making modifications to aid in the understanding of a program. As such, refactoring can sometimes slow a program down. However, after refactoring, it becomes easier to understand the code due to better design.

Mehmet Kaya et al. [14] mentions that refactoring in software requires some maturity degree of expertise. In particular, refactoring should be done under the supervision of a professional software engineer. Otherwise, the novice software engineer may not be able to evaluate the portion of code. As a result, it is necessary for beginners to need experience for finding the parts that need refactoring.

In this paper, we propose a method to automatically find the code areas that need refactoring through visualization, even if they lack expertise.

3 Automatic Bad Smell Extraction System

In previous studies, the tool chain for reverse engineering is organized as shown in Figure 1 [7,8]. This tool chain is composed of four steps. In Step 1, the source code is analyzed by a static analyzer. In Step 2, a database is created to store the analyzed source code data. In Step 3, the DotScript [9] is created, which can draw graphs utilizing the stored data. Finally, in Step 4, a graph is
created by the image creator. We utilize JavaParser [10] as our static analyzer. SQLite [11] is utilized for our database and Graphviz [12] is utilized as a graph creator.

**Figure. 1.** The structure of Tool chain for reverse engineering

Code data is analyzed based on an abstract syntax tree (AST) utilizing the JavaParser static analysis tool. All parts of the source code are analyzed via AST, meaning that we can freely utilize all the data required for the quality index according to the database structure. Table 1 describes the database column values.

**Table 1.** Database column values

<table>
<thead>
<tr>
<th>Column</th>
<th>Meaning</th>
<th>Column</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESSOR</td>
<td>Accessors</td>
<td>PARAMETER_NAME</td>
<td>Name of parameter</td>
</tr>
<tr>
<td>CLASS_NAME</td>
<td>Name of Class</td>
<td>PARAMETER_TYPE</td>
<td>Type of parameter</td>
</tr>
<tr>
<td>END_LINE</td>
<td>Number of end lines</td>
<td>REFER_CLASS</td>
<td>Name of caller class</td>
</tr>
<tr>
<td>EXTEND</td>
<td>Name of extended Class</td>
<td>REFER_MEMBER</td>
<td>Name of caller method</td>
</tr>
<tr>
<td>FILE_PATH</td>
<td>Path</td>
<td>REFERRED_CLASS</td>
<td>Name of callee class</td>
</tr>
<tr>
<td>FINAL</td>
<td>Check final</td>
<td>REFERRED_MEMBER</td>
<td>Name of callee variable or called method</td>
</tr>
<tr>
<td>IF_COUNT</td>
<td>Number of If statements</td>
<td>RETURN_TYPE</td>
<td>Type of return value</td>
</tr>
<tr>
<td>IMPLEMENT</td>
<td>Number of implementations</td>
<td>RETURN_VALUE</td>
<td>Return value</td>
</tr>
<tr>
<td>INITIALIZATION</td>
<td>Value of initialization</td>
<td>START_LINE</td>
<td>Number of start lines</td>
</tr>
<tr>
<td>LINE_NO</td>
<td>Number of lines</td>
<td>STATIC</td>
<td>Check static</td>
</tr>
<tr>
<td>MEMBER_TYPE</td>
<td>Type of variable or method</td>
<td>SWITCH_COUNT</td>
<td>Number of Switch statements</td>
</tr>
<tr>
<td>METHOD_NAME</td>
<td>Name of method</td>
<td>TYPE</td>
<td>Type of class or method</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of class or method</td>
<td>WHILE_COUNT</td>
<td>Number of While statements</td>
</tr>
</tbody>
</table>

Figure 2 shows the comparison between the source navigator and the Java Parser. The database of the Tool chain proposed in this paper complements the weaknesses of the Source Navigator. The Source Navigator has a problem that it is impossible to find parts for refactoring because of compressing data (such as ATTRIBUTES) in database. To solve this problem, we extend Java Parser to extract the split information such as ACCESSOR, ABSTRACT, FINAL, STATIC, RETURN_TYPE, INITIALIZATION, and so on. We also added INITIALIZATION to the instance value table, and also the RETURN_VALUE and the number of iterations (such as WHILE_COUNT)
to the method table. We replace the table information (such as SUPER CLASS, SUB CLASS) of inheritance in Source Navigator with “Extended”, “IMPLEMENT” fields in Database of JAVA PARSER.

Figure 2 Comparison of benchmark databases and our database
4 Visualizing Bad Smell Code Patterns

4.1 Bad Smell Extraction Program

Martin Fowler and Kent Beck define bad smell as components that require refactoring. Table 2 lists the bad-smell items that can be extracted and verified. Table 3 contains the query program that is utilized for extracting bad smell.

Table 2. Bad-Smell Items

<table>
<thead>
<tr>
<th>Range</th>
<th>Column</th>
<th>Mark</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>Switch Statements</td>
<td>[SS]</td>
<td>Too many Switch Statements.</td>
</tr>
<tr>
<td></td>
<td>Long Parameter List</td>
<td>[LPL]</td>
<td>Too many parameters.</td>
</tr>
<tr>
<td></td>
<td>Feature Envy</td>
<td>[FE]</td>
<td>Too interested in another class.</td>
</tr>
<tr>
<td></td>
<td>Message Chains</td>
<td>[MC]</td>
<td>Passes through too many classes to get a value.</td>
</tr>
<tr>
<td></td>
<td>Middle Man</td>
<td>[MM]</td>
<td>Too many delegations to other classes.</td>
</tr>
<tr>
<td></td>
<td>Data Clump</td>
<td>[DCP]</td>
<td>Same data always grouped together.</td>
</tr>
<tr>
<td></td>
<td>Shotgun Surgery</td>
<td>[SS]</td>
<td>Several other functions must be modified to modify a target function.</td>
</tr>
<tr>
<td>Class</td>
<td>Comments</td>
<td>[CMT]</td>
<td>Too many comments.</td>
</tr>
<tr>
<td></td>
<td>Lazy Class</td>
<td>[LZC]</td>
<td>A class does nothing.</td>
</tr>
<tr>
<td></td>
<td>Data Class</td>
<td>[DCS]</td>
<td>A class contains only getters and setters.</td>
</tr>
<tr>
<td></td>
<td>Large Class</td>
<td>[LGC]</td>
<td>Too many instance variables.</td>
</tr>
<tr>
<td></td>
<td>Refused Bequest</td>
<td>[RB]</td>
<td>Not utilizing all inherited objects.</td>
</tr>
</tbody>
</table>

“Middle Man” is a case in which a method makes too many delegations to another class. Such cases can be extracted when the method length is less than five lines and its return value is also called from another class. Intuitively, “Long Method” is a case where a method is very long. Such cases can be extracted by comparing the start line and end line of a method. “Long Parameter List” is a case in which there are too many parameters. Such cases can be extracted by counting the number of parameters in a method. “Data Clump” is a case in which there are groups of parameters with the same structure. Such cases can be extracted when there are parameter groups with a fixed pattern. “Message Chains” is a case in which a method’s chain of calls is overly complex. Such cases can be extracted when a value passes through a large number of classes when it is called. “Shotgun Surgery” is a case in which the task performed by a class is interfered with by other classes. Such cases can be extracted when the number of calls made by a class is greater than the number of times that class is called. “Feature Envy” is a case in which a class utilizes too many properties of another class. Such cases can be extracted when the get and set methods in another class are used very frequently. “Switch Statements” is a case in which there is a large number of switch statements. This issue can be easily resolved via polymorphism, so switch statements are extracted directly. “Comments” cases are extracted when comments are too long. “Lazy Class” is a case in which a class does not have any work to do. Such cases can be extracted when a class is never called. “Data Class” cases are extracted when a class’s methods consist of only get and set methods. “Large Class” cases are extracted when a class’s instance
Table 3. Bad Smell Extraction Query Programs

<table>
<thead>
<tr>
<th>Smell Type</th>
<th>Query Program</th>
</tr>
</thead>
</table>
| Middle Man | ResultSet mm = statement.executeQuery("select count(RETURN_TYPE) from JPDB_METHOD where RETURN_TYPE in (select REFERRED_CLASS from JPDB_REFERBY where REFER_CLASS in (select [name] from JPDB_CLASS) and LINE_NO < 5")
while (mm.next()) {
    return mm.getString("CNT");
} |
| Feature Envy | ResultSet fe = statement.executeQuery("select count(*) as CNT from JPDB_REFERBY where REFERRED_MEMBER LIKE 'get%' and REFERRED_CLASS in (select [name] from JPDB_CLASS)");
while (fe.next()) {
    int cnt = Integer.parseInt(fe.getString("CNT"));
    return cnt;
} |
| Data Clump | ResultSet dcp = statement.executeQuery("select PARAMETER_TYPE, PARAMETER_NAME from JPDB_METHOD where NAME = [name]");
while (dcp.next()) {
dataClump.put(dcp.getString("PARAMETER_TYPE" + PARAMETER_NAME));
if (dataClump.count() > count) {
    return count;
}
} |
| Large Class | ResultSet lgc = statement.executeQuery("select count(*) as CNT from JPDB_INSTANCE_VAR where CLASS_NAME in (select [name] from JPDB_CLASS)");
while (lgc.next()) {
    int count = Integer.parseInt("CNT");
    return count;
} |

<table>
<thead>
<tr>
<th>SMELL TYPE</th>
<th>QUERY PROGRAM</th>
</tr>
</thead>
</table>
| Feature Envy | ResultSet fe = statement.executeQuery("select count(*) as CNT from JPDB_REFERBY where REFERRED_MEMBER LIKE 'get%' and REFERRED_CLASS in (select [name] from JPDB_CLASS)");
while (fe.next()) {
    int cnt = Integer.parseInt(fe.getString("CNT"));
    return cnt;
} |
| Switch Statements | ResultSet ss = statement.executeQuery("select SWITCH_COUNT from JPDB_METHOD where NAME = [name]");
while (ss.next()) {
count++;
return count;
} |
| Data Clump | ResultSet dcp = statement.executeQuery("select PARAMETER_TYPE, PARAMETER_NAME from JPDB_METHOD where NAME = [name]");
while (dcp.next()) {
dataClump.put(dcp.getString("PARAMETER_TYPE" + PARAMETER_NAME));
if (dataClump.count() > count) {
    return count;
}
} |
| Large Class | ResultSet lgc = statement.executeQuery("select count(*) as CNT from JPDB_INSTANCE_VAR where CLASS_NAME in (select [name] from JPDB_CLASS)");
while (lgc.next()) {
    int count = Integer.parseInt("CNT");
    return count;
} |
| Refused Bequest | ResultSet rb = statement.executeQuery("select count(*) from JPDB_REFERBY where REFERRED_CLASS and REFERRED_CLASS = [name] select count(*) from JPDB_METHOD where CLASS_NAME in (select EXTEND from JPDB_CLASS where NAME = [name])");
while (rb.next()) {
    int count = rb.getString("CNT");
    return count;
} |
variables are too numerous. “Refused Bequest” is a case in which only a portion of the inherited features from another class are utilized. Such cases can be extracted when one or more inherited methods are not utilized.

4.2 Results of Improvement via Bad Smell Pattern Visualization and Refactoring

We will now examine some sample code for visualization. The sample code accesses a database to retrieve student phone number data. The program then returns the phone number if it has sufficient access permission. Figure 3 presents the code that requests database permission and accesses the data. In this code, the getDatabase() method of the FirstDBGate class is called from the Database class to request a connection.

```java
public class Database {
    public Connection getConnection(String id, String pw) {
        FirstDBGate db1 = new FirstDBGate();
        conn = db1.getConnection(id, pw);
        return conn;
    }
}

public class FirstDBGate {
    public Connection getDatabase(String id, String pw) {
        if (id.equals(pw)) {
            SecondDBGate db2 = new SecondDBGate();
            Connection conn = db2.getConnection(id, pw);
            return conn;
        }
        return null;
    }
}

public class SecondDBGate {
    public Connection getDatabase(String id, String pw) {
        if (!id.isEmpty()) {
            ThirdDBGate db3 = new ThirdDBGate();
            Connection conn = db3.getConnection();
            return conn;
        }
        return null;
    }
}

public class ThirdDBGate {
    public Connection getDatabase() {
        try {
            Class.forName("org.sqlite.JDBC").newInstance();
            Connection conn = DriverManager.getConnection("jdbc:sqlite:54361");
            conn.close();
            return conn;
        } catch (Exception ex) {
            return null;
        }
    }
}
```

**Figure. 3.** Database connection portion of sample code

First, we examine getDatabase() in FirstDBGate. If the id and pw values received as parameters are valid, the getDatabase() method calls the getDatabase() method of SecondDBGate. If the id is not an empty value, the getDatabase() method of SecondDBGate calls the getDatabase() method of ThirdDBGate. Finally, the getDatabase() method of ThirdDBGate receives the database connection information. During this process, the Database class may utilize the received id and pw values to create connection information. However, the Database class receives this connection information from a chain of unnecessary calls of get methods in several classes. In this case, the Database class can be detected as a “Message Chains” bad smell. Figure 4 presents the portion of the sample code where the student phone numbers are obtained by a mobile phone object.
In Figure 4, the Phone class connects to the database through its class constructor. The Phone class holds unformatted student phone number data. The Student class’s getMobilePhoneNumber() method takes a Phone object and calls that object’s get method. Then, getMobilePhoneNumber() formats the phoneNumber variable and returns it, and the program ends. However, if there was a method in the Phone class that formatted and returned the phone number to begin with, then the get methods in the Phone class would not be called several times by the Student class. In this case, the Student class’s calls can be viewed as “Feature Envy” bad smell. Figure 5 presents the results of visualizing the sample code as a graph. The previously mentioned “Message Chains” and “Feature Envy” bad smells are visible with red colors in Figure 5. Additionally, there are also “Data Class” and “Lazy Class” bad smells. As shown in Table 2, the FirstDBGate, Student, and Phone classes are composed purely of get methods, so they are marked as [DC]. In contrast, TestClass, which is not utilized in the Main class, is marked as [LC].

Figure 4. Sample code that retrieves student phone numbers

```java
public class Phone {
    private String unformattedNumber;
    public Phone() throws SQLException {
        Statement stat = conn.createStatement();
        ResultSet rs = stat.executeQuery(""");
        while(rs.next()) {
            String number = rs.getString(""");
            unformattedNumber = number;
        }
    }
    public String getAreaCode() {
        return unformattedNumber.substring(0, 3);
    }
    public String getPrefix() {
        return unformattedNumber.substring(3, 6);
    }
    public String getNumber() {
        return unformattedNumber.substring(6, 10);
    }
}

public class Student {
    public String getMobilePhoneNumber(Phone mobilePhone) {
        String phoneNumber = "(" + mobilePhone.getAreaCode() + "+"
                           + mobilePhone.getPrefix() + "+") + mobilePhone.getNumber();
        return phoneNumber;
    }
}
```

Figure 5. Visualization graph of sample code prior to refactoring
Next, we attempt to fix the various issues revealed by the visualization through refactoring. Figure 6 shows the refactored code, where all of the data connection information is resolved in the Database class to eliminate “Message Chains”.

```
public connection getConnection(String id, String pw) {
    FirstDBGate dbl = new FirstDBGate();
    conn = dbl.getDatabase(id, pw);
    if (!id.equals(pw)) {
        if (!id.isEmpty()) {
            try {
                Class.forName("org.sqlite.JDBC").newInstance();
                Connection conn = DriverManager.getConnection("jdbc:sqlite:/example.db");
            } catch (Exception ex) { } } } } } } } } } return conn;
```

**Figure. 6.** Database class with refactored getConnection method

In the original code, the Database class is only able to request a connection data by going through the FirstDBGate, SecondDBGate, and ThirdDBGate classes. After refactoring, the Database class is able to resolve this entire process by itself. It no longer unnecessarily travels through several objects. As a result, the database can be accessed by calling only the getConnection method. Figure 7 shows the Phone class code that is refactored to eliminate “Feature Envy”.

```
public class Phone {
    public String getAreaCode() {
        return unformattedNumber.substring(0, 3);
    }
    public String getPrefix() {
        return unformattedNumber.substring(3, 6);
    }
    public String getNumber() {
        return unformattedNumber.substring(6, 10);
    }
    public String toFormattedString() {
        String phoneNumber = "(" + getAreaCode() + "\) " + getPrefix() + "." + getNumber();
        return phoneNumber;
    }
```

**Figure. 7.** Phone class with toFormattedString method added via refactoring

Originally, the Student class’s getMobilePhoneNumber() method has “Feature Envy” because it called the Phone class’s get methods several times. Refactoring is performed and the toFormattedString() method is created in the Phone class. As a result, the Student class can call the toFormattedString() method once to retrieve the data that previously required three calls to retrieve. Finally, the class that is considered a “Lazy Class” is deleted. The “Message Chains”, “Feature Envy”, “Lazy Class”, and “Data Class” bad smells are eliminated via refactoring. Figure 8 shows the resulting graph.
Figure 8. Improved graph after refactoring

Based on the new code in Figure 6, the “Message Chains” in Database bad smells has disappeared. Based on the new code in Figure 7, the Student class’s “Feature Envy”, Phone’s “Data Class”, and unused “Lazy Class” have disappeared. Here, classes and methods are shown in red if they have a bad smell and shown in blue if they do not. Therefore, one can see the bad smells in software at a glance.

5 Results of Analysis and Discussion

We directly download a real open-source program and perform visualization on its code. Figure 9 presents the results of performing visualization on JSAP-2.1. This program is an open-source program that performs JavaScript parsing.

In the graph, ① is marked with [LZC], [DCS], and [RB]. Class ① is not called by any other class. Additionally, it is composed purely of set and get methods, and does not utilize all of its inherited methods. As a result, we can see that class ① has three bad smells. Therefore, we can delete this class because it is a “Lazy Class”. The Refused Bequest bad smell also disappears, meaning the program’s overall degree of inheritance is reduced.

In ②, the TestAll class is marked as a “Lazy Class”. The TestAll class has one call line and seven inheritance lines. This class can also be deleted. Through this deletion, the entire program’s design complexity can be reduced by reducing the total number of calls made.

In ③, getParameter() and configure() are marked as [FE]. In the actual code, configure() calls the get methods of the FlaggedOption class several times. To resolve this issue, the FlaggedOption class can be inherited or the location of the methods can be changed. Even if “Lazy Classes” are only deleted, the overall program’s design can become much better.

Figure 10 presents a visualization of the program after deleting “Lazy Classes”. Compared to Figure 9, the design became much simpler. This can reduce the cost of flexibility. But when we use the target program based on open source, several developers can simultaneously use and develop target code. Therefore, even if there is no functional problem in the code, unnecessary classes are added. Through our visualization method, we can easily identify all “Lazy Classes” in the code.
Figure. 9. A graph visualizing the open-source JSAP-2.1 program before refactoring
Figure 10. A graph visualizing the open source JSAP-2.1 program after simple refactoring
6 Conclusion

In this paper, we examine methods that can increase software quality by improving its design. A method that utilizes refactoring is selected from several available methods. We visualize structures that are difficult to analyze because of the non-visual nature of software. The parts that seem to require refactoring are marked red in the resulting graphs. Therefore, developers can see which parts of a program have bad smells and perform refactoring directly. In this manner, the bad smells and program design can be improved. As a result, the original design and functions based on the program requirements can be maintained, even when modifications occur during code development as a result of frequently changing requirements.

Future research will go beyond simply extracting bad smells. Solutions will be provided that apply efficient design patterns to each extracted case. Additionally, all remaining bad smells will be extracted to create a quality index for refactoring.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2017R1D1A3B03035421), and supported by Development of Information Communication and Broadcast R&D(2018 Open OS Environment Development and Diffusion) through NIPA(National IT Industry Promotion Agency)(S1113-18-1001), and supported by a grant(18CTAP-C133299-02) from Technology Advancement Research Program funded by Ministry of Land, Infrastructure and Transport of Korean government. Also, 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).

References


● Corresponding Author: Byungkook Jeon, R. Young Chul Kim