Code Smell Detecting Tool and Code Smell-Structure Bug Relationship

Phongphan Danphitsanuphan
Department of Computer and Information Science
King Mongkut's University of Technology
North Bangkok
Bangkok, Thailand
phongphand@kmutnb.ac.th

Thanitta Suwantada
Department of Computer and Information Science
King Mongkut's University of Technology
North Bangkok
Bangkok, Thailand
thanitta.su@gmail.com

Abstract— **This paper proposes an approach for detecting the so-called bad smells in software known as Code Smell. In considering software bad smells, object-oriented software metrics were used to detect the source code whereby Eclipse Plugins were developed for detecting in which location of Java source code the bad smell appeared so that software refactoring could then take place. The detected source code was classified into 7 types: Large Class, Long Method, Parallel Inheritance Hierarchy, Long Parameter List, Lazy Class, Switch Statement, and Data Class.**

This work conducted analysis by using 323 java classes to ascertain the relationship between the code smell and structural defects of software by using the data mining techniques of Naive Bayes and Association Rules. The result of the Naive Bayes test showed that the Lazy Class caused structural defects in DLS, DE, and Se. Also, Data Class caused structural defects in UwF, DE, and Se, while Long Method, Large Class, Data Class, and Switch Statement caused structural defects in UwF and Se. Finally, Parallel Inheritance Hierarchy caused structural defects in Se. However, Long Parameter List caused no structural defects whatsoever. The results of the Association Rules test found that the Lazy Class code smell caused structural defects in DLS and DE, which corresponded to the results of the Naive Bayes test.

Index Terms-- Code smell, Refactoring, Software metric, Structural bugs.

I. INTRODUCTION

It is important to always keep up with a software maintenance process as it helps improve the system to perform to its best ability and to work suitably in line with the user's purpose. Such a process can be described as an improvement of the software's defects density or a development of the software that leads to its efficient and appropriate function within the system's environment. However, in software development, there could be a limitation in terms of time; therefore, the refactoring is often neglected because of complications and resource consumption.

Code smell detection without an effective tool and the differences regarding basic knowledge about code smell together with the different software refactoring of each software developer can bring about difficulty in setting the bottom line of when to conduct the refactoring and how much needs to be done.

In particular, the occurrence of code smell in software depends on the coding behavior of programmers; therefore, code smell detection is difficult and also lacks accuracy. In large programming projects, software developers might forget or be confuse about which code smell has already been fixed, which leads to work duplication.

This paper proposes a tool that helps detect the location of the source code of code smell in Java programs under the concept of Martin Fowler [6] who presented a theory about code smell and software refactoring, and who developed a tool called Eclipse Plug-in. In addition, code smell and structural bug relationships are analyzed and elaborated in this paper.

II. CODE SMELL

Code smell is the bad part of source code that creates difficulty to understand and improve the software. The principle of code smell is to indicate the coding spots that need to be refactored. There are 22 types of bad source code [6] [11] but only some types of code smell can be measured by software metrics. These are Large Class, Long method, Long Parameter and Lazy class.

III. SOFTWARE METRICS

Software Metrics are a quantitative measurement of software. In this paper, we focus only on source code's metrics as referred to in the following table [1] [19] [3].

Notation	Title	Level
NOM	Number of Methods	Class
LOC	Lines of Code	Class
DIT	Depth of Inheritance Tree	Class
PAR	Number of Parameters	Method

978-1-4577-1964-6/12/$31.00 ©2012 IEEE

WMC	Weighted Methods per Class	Class
NOF	Number of Attributes	Class
MLOC	Method Lines of Code	Method
VG	McCabe Cyclomatic Complexity	Method
NCS	Number of Children	Class
LCOM	Lack of Cohesion of Methods	Class

Table 1: Software metric (Object-oriented Software Metric)

IV. Software Structure Bug

A software structure bug is a defect occurring from false commands or instructions in computer programs. It creates malfunctions in the system, such as slow compiling, difficulty in fixing or improving source code, and incorrect program output.

As there are many kinds of software structure bugs [8], we have chosen 6 structural bugs [22] that are commonly found from experimenting source code (323 Java classes). They are as follows:

SE_BAD_FIELD (SE)

This Serializable class defines a non-primitive instance field which is neither transient, Serializable, nor java.lang.Object, and does not appear to implement the Externalizable interface or the readObject() and writeObject() methods. Objects of this class will not be deserialized correctly if a non-Serializable object is stored in this field.

DE_MIGHT_IGNORE (DE)

This method might ignore an exception. In general, exceptions should be handled or reported in some way, or they should be thrown out of the method.

UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR (UwF)

This field is never initialized within any constructor, and is therefore could be null after the object is constructed. This could be a either an error or a questionable design, since it means a null pointer exception will be generated if that field is dereferenced before being initialized.

DLS_DEAD_LOCAL_STORE (DLS)

This instruction assigns a value to a local variable, but the value is not read or used in any subsequent instruction. Often, this indicates an error, because the value computed is never used.

REC_CATCH_EXCEPTION (REC)

This method uses a try-catch block that catches Exception objects, but Exception is not thrown within the try block, and RuntimeException is not explicitly caught. It is a common bug pattern to say try { ... } catch (Exception e) { something } as a shorthand for catching a number of types of exception each of whose catch blocks is identical, but this construct also accidentally catches RuntimeException as well, masking potential bugs.

DM_NUMBER_CTOR (DM)

Using new Integer (int) is guaranteed to always result in a new object whereas Integer.valueOf(int) allows caching of values to be done by the compiler, class library, or JVM. Using of cached values avoids object allocation and the code will be faster.

V. Bad Smell Detecting Tool (BSDT)

BSDT is implemented as Eclipse plug-in [4] and is capable of detecting subsequent bad smells as depicted in table 1 by using a predefined threshold of software metrics which were elaborated from previous works as shown in the table below.

Code Smell	**Level**	**Metrics & threshold**	**Reference**
Large Class	Class	NOM > 20 || NOF > 9 || LOC >750	[20]
Long Method	Method	MLOC > 50	[20]
Long Parameter List	Method	PAR > 5	[11], [20]
Switch Statements	Method	VG > 10	[5], [20]
Parallel Inheritance Hierarchies	Class	DIT > 3 || NSC > 4	[5] [14]
Data Class	Class	WMC > 50 || LCOM > 0.8	[1]
Lazy Class	Class	(NOM < 5 && NOF < 5) || DIT < 2	[1]

Table 2: Showing the appropriate threshold allocation for code smell and software metrics

BSDT verification

In bad smell source code detection, BSDT possesses the ability to display the detected code smells, software metric values, and location of the bad smell in the associated source code, including the ability to link to the detected code smell location. Besides, the tool can provide the definition of the code smells and facilitate the proper refactoring method as shown in Fig. 1. Then, the tool will store the code smell detection result file in the extension form of .CVS for further "Bad smell" - "Structure bug" relationship analysis.

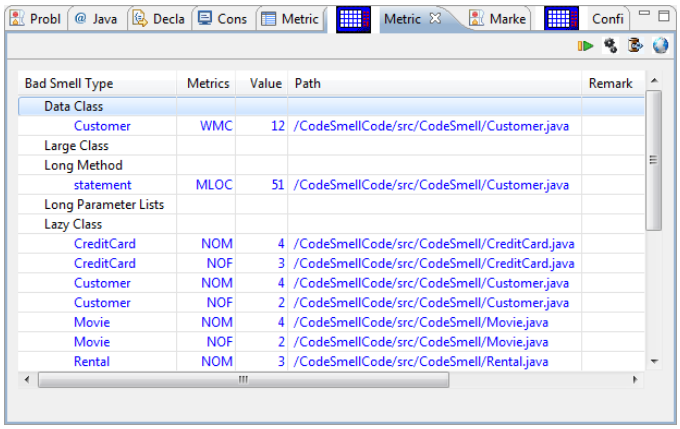

Figure 1: Code smell detection result

From the testing of BSDT by using known bad smell source code, which was the movie rental program and electricity calculating program from "Refactoring: Improving the Design of existing Code" book [8], the tool could indicate the type of code smell, software metrics value, and the location of the bad smell in source code with 100% accuracy. When retesting with the tool after the code smells had been fixed, the fixed code smells were not being detected again. This shows that BSDT was able to detect the bad smell and display the code smell detection results correctly.

Project Name	# LOC	Known smell type	Known Smell	Known detected smell	Accuracy
Movie rental Program	122	Lazy Class	5	5	100%
		Long Method	1	1	100%
		Switch Statement	1	1	100%
Electricity calculating program	336	Lazy Class	3	3	100%
		Long Method	2	2	100%
		Switch Statement	2	2	100%

Table 3: The BSDT testing result

VI. BAD SMELL AND STRUCTURAL BUG CORRELATION ANALYSIS

In the analysis of the correlation between code smell and software structure bugs, the 323 data sets, which were the Java program source codes from 35 different projects and 18 different authors, were used. However, in each data set, there were over 5 classes or more than 300 LOCs (Lines of Code) and that amount can be added up to 42,777 lines.

When introducing source codes into the process of bad smell detection and also using the same set of data with the program Find Bug [8] in order to detect the structural bugs, the researcher used data mining [18] by using Naive bayes, and Association Rules techniques to analyze which kind of software structure bug was being caused by each type of code smell. In the analysis, there are 2 factors as follows:

1. Input factors are 7 types of code smell, which are Data class, Large class, Long method, Long Parameter List, Parallel Inheritance Hierarchy, Switch Statement, and Lazy class.
2. Predict factors are 6 kinds of software structure bug, which are Se, DE, UwF, DLS, REC, and DM.

The result from the analysis of the correlation between code smell and software structure bug

Probability Calculation	Value
P(LazyClass|DLS)	0.661
P(LazyClass|DE)	0.767
P(DataClass|UwF)	0.779
P(DataClass|Se)	0.780
P(LongMethod|UwF)	0.779
P(LongMethod|Se)	0.630
P(LargeClass|UwF)	0.779
P(LargeClass|Se)	0.779
P(SW|UwF)	0.588
P(SW|Se)	0.606
P(PIH|Se)	0.756

Table 4: Probability calculation [21] of code smell and software structure bug from Naive Bayes technique result

The analysis of P(LazyClass | DLS) = 0.67 from the mentioned values shows that the calculated probability values are high due to the DLS type of software structure bug. That means the local variable has been set but never been used. However, Lazy Class code smell is the class that does not do much work. It can possibly be interpreted as meaning that not every local variable is being used, so the DLS type of software structure bug occurs frequently.

The high values in the probability analysis of P(DE | LazyClass) = 0.62, and P(LazyClass | DE) = 0.77 are affected by the DE (DE_MIGHT_IGNORE) type of software structure bug which is the bug describing a method that might ignore an exception. In general, exceptions should be handled or

reported in some way, or they should be thrown out of the method.

It is noticed that Lazy Class code smell, which does not do much work, has fewer parameters and methods; therefore, this often leads to the programmer's neglect in handling exceptions. Consequently, in Lazy Class code smell, it is more likely that the DE type of software structure bug will be found.

From the analysis of P(Se | LargeClass) = 0.76, and P(LargeClass | Se) = 0.77, the probability values are quite high. It is noticeable that the SE (SE_BAD_FIELD) type of software structure bug, which detects Serializable classes, defines a non-primitive instance field which is neither transient, Serializable, or java.lang.Object, and does not appear to implement the Externalizable interface or the readObject() and writeObject() method. Objects of this class will not be deserialized correctly if a non-Serializable object is stored in this field.

As being studied, Large Class code smell has a large number of lines of code, global variables, and methods, as well as a complicated function. There is a high probability that the Se type of software structure bug can be found due to the fact that the Large Class code smell has many global variables.

From the observations of source codes used in the experiment, there is no software structure bug occurring from Long Parameter List code smell. It was found that none of the 6 detected software structure bugs detected the parameter. Thus, from the results, there is no correlation between Long Parameter List code smell and the various types of software structure bugs.

It can be noticed in the probability analysis of P(LargeClass|UwF) = 0.71 that the UwF type of software structure bug is a local variable without the default value set in the constructor, which leads to a null variable. It is obvious that Large Class code smell is rather big and has a lot of global variables. Therefore, it is possible that the programmer might skip the value setting for some variables or, because of the large number of variables, only important variables may be selected to be set.

P(PIH|Se) = 0.77 from the mentioned probability shows that the Se type of software structure bug is the Primitive variable, not a local variable, and does not have input and output management using the Externalizable Standard class in JDK 1.1. It is noticeable from PIH code smell that when a sub class is created, it could also be another class' sub class. For example, if Class B is Class A's sub class, when creating Class C from Class B, it also means that Class C is the sub class of Class A. In this case, if too much inheritance occurs, it is hard to read, to understand, to fix or to make changes. As in this case, when Class A runs a Move Method or Move Field, it is possible that the sub classes (Class B and Class C), which are referred to as the super class' Method or Field, could not be found. In high PIH value source code, it the Se type of software structure bug is likely to be found due to the multiple inheritance source code, which refers to the big system with complicated functioning. Therefore, many variables are needed, including various types of variable, not only base variables. Thus, the chance of occurrence of the Se type of software structure bug is high.

VII. THE EXPERIMENT USING ASSOCIATION RULES MINING TECHNIQUE

Only DLS and DE were detected by using Association rule techniques. The following association rule threshold configurations were set.

Minimum support = 10 items
Minimum rule probability = 0.4

First detect: all types of code smell does not affect DLS but the LZC code smell possesses the possibility to cause and not to cause DLS as depicted in fig. 2.

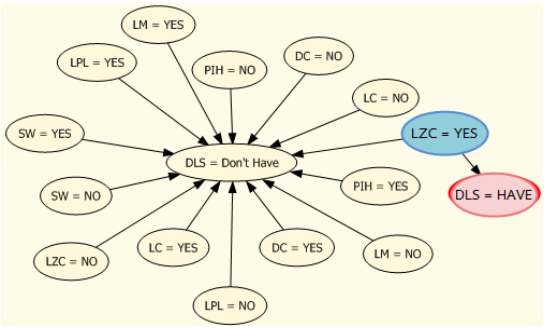

Figure 2: Code smells and DLS type of software structure bug using Association Rules technique

Second detect: all types of code smell do not affect DE but it is likely that DE could be or could not be caused by LZC code smell as shown in fig. 3.

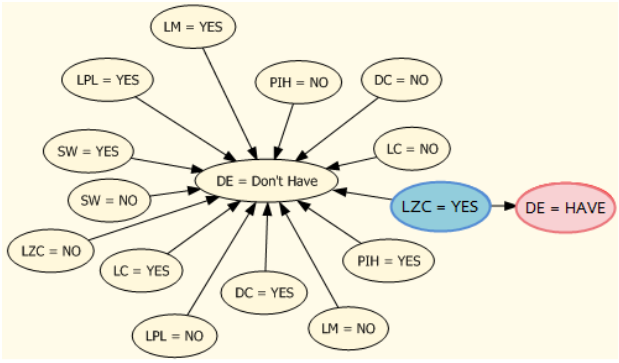

Figure 3: Code smells and DE type of software structure bug using Association Rules technique

From the testing results using Association Rules mining, there is only one code smell, Lazy Class, which can cause 2 types of software structure bug: DLS, and DE.

VIII. CONCLUSION

BSDT is able to detect the bad smell in Java source code and shows the results of the detection accurately against source code testing.

From the Naïve Bayes probability calculation, when different types of code smell occur, there is a high probability that DLS and DE could occur when the Lazy Class code smell is found. UwF showed significant occurrences of Long Method code smell. UwF and Se were greatly affected by Large Class. Lastly, Se was highly associated with PIH.

The Association Rules testing results only show that Lazy Class code smell brings about DLS and DE software structure bugs, which correspond with the results of the Naïve Bayes experimenting.

REFERENCES

[1] Piyanuch Chermchaipume, *Software Metric Support towards Refactoring Inference*, Khon Kaen University of Information Technology, 2008.

[2] Thisana Pienlert and Pornsiri Muenchaisri. "Bad-Smell Detection using Object-Oriented Software Metrics," *International Society of Computers and Their Applications (ISCA) International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA'04),* Cairo, Egypt, December 2004.

[3] Somwang Sae-Tang, *Design and Implementation of a Measurement Tool for Object-Oriented Programs*, Chulalongkorn University of Computer Engineering, 2001.

[4] Clayberg, Eric, and Dan Rubel, *Eclipse: Building Commercial-Quality Plug-ins*, 2nd ed., Boston: Addison-Wesley, 2006.

[5] Crespo, Yania, Carlos Lopez, Raul Marticorena, and Esperanza Manso, "Language independent metrics support towards refactoring inference," in *9th ECOOP Workshop on QAOOSE 05 (Quantitative Approaches in Object-Oriented Software Engineering,* Glasgow: UK. ISBN: 2-89522-065-4, July 2005.

[6] Fowler, Martin, *Refactorin: Improving the Design of Existing Code.* Boston: Addison-Wesley, 2000.

[7] Garzas, J, and M Piattini, *Object-Oriented Design Knowledge: Principles, Heuristics and Best Practices*. Hershey: Idea Group Publishing, 2007.

[8] Hovemeyer, David, and William Pugh, *Finding Bugs is Easy*. Dept. of Computer Science, University of Maryland, 2004.

[9] Kataoka, Y., Imai, T., Andou, H. and Fukaya, T., "A Quantitative Evaluation of Maintainability Enhancement by Refactoring," in *Proc. Int. Conf. on Software Maintenance (ICSM'02),* 2002, pp. 576-585.

[10] Kerievsky, Joshua, *Refactoring to Patterns*. Addison-Wesley, 2004.

[11] MantyLa, M, Bad Smells in Software - *a Taxonomy and an Empirical Study*. Helsinki University of Technology, 2003.

[12] Marticorena, Raul, "Detecting Design Flaws via Metrics in Object-Oriented Systems," in *Proc. 39th Int. Conf. and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, 2001.

[13] Marticorena, Raul, Lopez Carlos, and Crespo Yania, "Extending a Taxonomy of Bad Code Smells with Metrics," *WOOR'0,* Nantes, 2006.

[14] —, "Parallel inheritance hierarchy : Detection from a static view of the system," in *Proc. 6th ECOOP Workshop Object-Oriented Reengineering,* Springer-Verlag, 2005.

[15] Rutar, Nick, Christian B Almazan, and Jeffrey S Foster, "A Comparison of Bug Finding Tools for Java," in *Proc. 15th IEEE Symp. Software Reliability Engineering,* Saint-Malo, France: IEEE Computer Society, November 2004.

[16] Slinger, Stefan, *Code Smell Detection in Eclipse*. Delft University of Technology, 2005.

[17] SmellsToRefactorings [Online]. Avaliable: http://wiki.java.net/bin/view/People/SmellsToRefactorings

[18] Tang, ZhaoHui, and Jamie MacLennan, *Data Mining with SQL Server 2005*. Indianapolis: Wiley Publishing, 2005.

[19] Henderson Sellers, B., *Object-Oriented Metrics: Measures of Complexity.* Prentice Hall, 1996.

[20] Williams, Laurie, Dright Ho, and Sarah Heckman. (2005). *Software Metrics in Eclipse* [Online]. Avaliable: http://agile.csc.ncsu.edu/SEMaterials/tutorials/metrics/

[21] *wikipedia.* (2007). Probability. [Online]. Avaliable: http://en.wikipedia.org/wiki/Probability

[22] http://findbugs.sourceforge.net/bugDescriptions.html