An Empirical Investigation of Risk Based Automatic Code Smells Detection

ROHIT KUMAR
M.Tech (IS) Research Scholar,Chandigarh Engineering College, Landran (Mohali)
Punjab, (India)

JASPREET SINGH
Assistant Professor,
Department of Information and Technology, Chandigarh Engineering College, Landran (Mohali)
Punjab, (India)

Abstract:
Code smells are structural uniqueness of software that mayspecify a code or design difficulty that makes software inflexible to advance andmaintain, and may trigger re-factoring of code. A lateststudy is active inmajor automatic detection tools to help humans in finding smells whencode size becomes impossible for manual appraise. Since the definitionsof code smells are informal and individual, assessing how effective codesmell detection tools are is both important and durable to accomplish. Thispaper analysis the current view of the tools for automatic code smellsdetection. It defines research queries about the consistency of theirresponses, their ability to representation the regions of code most affected bystructural decay, and the significance of their responses with respect to prospectsoftware evolution. It gives respond to them by analyzing the production offour representative code smell detectors applied to six different versions ofGanttProject, an open source system written in Java. The results of theseexperiment cast light on what present code smell detection tools are ableto do and what the applicable areas for advanceenhancement are.

Keywords*:* *Code smell detection tools, Code smell, Refactoring of code, Software evaluation quality, Software maintance*

1. Introduction

It is significant to constantly keep up with a software maintenance process as it helps progress the system to perform to its best capability and to work correctly in line with the user's purpose. Such a process can be describe as an enhancement of the software's defects density or a development of the software that lead to its well-organized and suitable function within the system's environment. However, in software development, there could be a restraint in terms of time [1]; therefore, the refactoring is often disused because of difficulty and resourceeuse.

Code smell detection without an effective tool and thedissimilarity regarding basic information about code smelltogether with the dissimilar software refactoring of eachsoftware developer can bring about complexity in setting thebottom line of when to behavior the refactoring and how muchneeds to be done.In exacting, the amount of code smell in softwaredepends on the coding performance of programmers; therefore,code smell detection is not easy and also lacks precision. Inlarge programming projects, software developers strength forgetor be confound about which code smell has previously been fixed,which direct to work replication.

This paper advise a tool that helps distinguish the location ofthe source code of code smell in Java program under theconception of Martin Fowler [2] who presented a theory aboutcode smell and software refactoring, and who developed a toolcalled Eclipse Plug-in. In adding, code smell and structuralbug interaction are analyzed and complicated in this paper.

2. Code Smell

Code smell is the bad part of source code that createscomplexity to appreciate and advance the software. Theprinciple of code smell is to specify the coding spots that needto be refectory [3].
There are 22 types of bad source code [6] but only several types of code smell can be calculated bysoftware metrics. These are Largeclass, Long process, Long structure and Lazy class.

3. Software Metrics

Software Metrics are a quantitative capacity ofsoftware. In this paper [9], we center only on source code's metricsas referred to in the following table [8].

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

4. Bad Smells in Code

A significant where to re-factor within in a system is relatively a challenge to recognizeregion of bad design. Theseregions of bad design are known as "Bad Smells" or "Stinks" within code. A result these regions are more associatedto "human perception" than a precise science. The developers knowledge is relied upon in identify these "BadSmells".

Conversely, refactoring itself will not convey the full benefits, if we do not appreciate when refactoringneeds to be functional. To make it easier for a software developer to choose whether certain software needsrefactoring or not, Fowler & Beck give a list of bad code smells [4].

Code smell is any indication that representative something incorrect. It normally indicates that the code should be refactored or theoverall design should be re-examined. The term appears to have been coined by Kent Beck .Usage of theterm increased after it was feature in Refactoring. Badcode exhibits certain freedomthat can berectifying using Re-factoring. These are called **Bad Smells**.

Code smell is any symptom that indicating something wrong. It generally indicates that the code should be re-factored or the overall design should be reexamined.Usage of the term increased after it was featured in Refactoring. Bad code exhibits certain characteristics that can be rectified using Refactoring. These are called Bad Smells [5].

- Long Parameter List
- Long Method
- Large Class
- Lazy Class
- Duplicate Code
- Dead Code
- Unused Catch Block

- Switch Statement
- Temporary Field
- Comment Lines

Long Method: When method is too long means more number of lines of code.
Large Class: Classes that have large numbers of instance variables and large number of lines of code. Sometimes they are only used occasionally large classes can also suffer from code duplication.
Long Parameter List: Long parameter lists are hard to understand. Long parameter list means that a method takes too many parameters.
Comments: If the comments are present in the code more than the lines of code.
Switch Statements: Switch statements may produce duplication. You can find similar switch statements scattered in the program in several places. . Maybe classes and polymorphism would be more appropriate
Lazy Class: Classes that are not doing much work and number of method is null.
Temporary Field: When some of the instance variables in a class are only used occasionally [5].
Duplicate Code: The same code structure in two or more places is a good sign that the code need to be re-factored.
Dead Code: Dead code is a section in the source code of a program which is executed but whose result is never used in any other computation. The execution of dead code wastes computation time and memory.

5. Refactoring

Refactoring is relatively a new area of research and so is not well defined [5]. There are a vast number ofdefinitions for refactoring; most of them are refer to below:

- Refactoring is the process of taking an object aim and re-arranging it in a variety of ways to make thedesign more flexible and re-usable. There are quite a few reasons you might want to do this, competenceand maintainability being possibly the most significant.
- To re-factor encoding code is to rewrite the code, to “clean it up”.
- Refactoring is the affecting of units of functionality from one place to another in your program.
- Refactoring has as a main objective, receiving each piece of functionality to exist in accurately oneplace in the software.

6. Refactoring Process

Refactoring process can be divided into a number of steps as shown below [6]:

1. Classify where the software needs to be refactored.
2. Decide which refactoring need to be applied to the identified places.
3. Assurance that the applied refactoring conserves behavior.
4. Apply the refactoring [7].
5. Review the effect of the refactoring on the quality individuality of the software or the process.
6. Continue the consistency between the refectory program code and other software artifact.

7. Metrics

As the list of object-oriented program quality metrics is virtually endless (i.e. alone describes more than 200 complexity metrics), I will focus on those program metrics which are most commonly used, being Number of Methods, Cyclomatic Complexity, Number of Children, Coupling between Objects, Response for a Class and Lack of Cohesion among Methods.

Definitions for these metrics are:

- **Number of Methods** calculates the number of methods of a class. It is an indicator of the functional size of a class.

- **Cyclomatic Complexity** counts the number of possible paths through an algorithm. It is an indicator of the logical complexity of a program, based on the number of flow graph edges and nodes [7].
- **Number of Children** measures the immediate descendants of a class [5]. It is an indicator of the generality of the class.
- **Lines of Code**
- **Comment lines**
- **Coupling between Objects** is a measure for the number of collaborations for a class [8]. It is an indicator of the complexity of the conceptual functionality implemented in the class.
- **Response for a Class** is the number of both defined and inherited methods of a class, including methods of other classes called by these methods [5]. It is an indicator of the vulnerability to change propagations of the class.
- **Lack of Cohesion among Methods** is an inverse cohesion measure (high value means low cohesion). Of the many variants of LCOM, we use LCOM1 as defined by Henderson-Sellers [10] as the number of pairs of methods in a class having no common attribute references. It is an indicator of how well the methods of the class fit together.

8. How to achieve well formed object-oriented method

- In order to achieve a well formed object-oriented method the following requirements must be met:
- Acceptably cohesive
- Low in complexity
- Appropriately sized,
- Independently testable
- Well documented

9. Acceptably Cohesive

One of the most important characteristics of a well formed object oriented method is cohesion. Method cohesion is defined as a measure of how well the elements within a module work together to provide a specific functionality. While cohesion was first used in structured design, method cohesion has been adapted for object-oriented software. Kang defines six levels of object-oriented method cohesion ranked from best to worst as shown in Figure 1. The highest level, functional cohesion, deals with the ability of a module to produce one output for one module (i.e., to change the state of one object). As shown by Kang's empirical evidence, strongly cohesive methods are desired because the stronger the cohesion, the easier the method is to maintain, understand, and reuse.

Functional	Only one output exist for the module
Sequential	One output is dependent on the other output
Communicational	Two outputs are dependent on a common input
Iterative	Two outputs are iteratively dependent on the same input
Conditional	Two outputs are conditionally dependent on the same input

10. Cohesion in Object-oriented Design

A. Cyclomatic Complexity

Having a low level of complexity is another important characteristic of a well formed object-oriented method. Students are introduced to McCabe's CyclomaticComplexity as the metric for measuring complexity of their code. McCabe's Cyclomatic Complexity, which measures the number of linearly independent paths within code, is defined as the number of decision points + 1 where decision points are conditional statements such as if/else or while. The goal is code that is not very complex and, therefore, low risk. Low level of complexity within a method makes methods more understandable and maintainable.

Number of Paths	Code Complexity	Risk
1-10	Not very complex	Low
11-20	Moderately complex	Moderate
21-50	Highly complex	High
51+	Unstable	Very high

B. McCabe's Complexity Scale

Appropriately Sized

The third characteristic of a well formed object-oriented method is size. One of the techniques for determining the size of software is counting its lines of code (LOC). The method of determining LOC depends on how the executable lines, blank lines, comment lines, data declarations, and multiple line statements are treated. LOC is considered inappropriate for measuring the quality of object-oriented classes [2, 15]. However, it is useful in measuring object-oriented methods since studies have shown that large methods result in a reduction of understandability, reusability, testability, and maintainability. The appropriate size of a method depends on the programming language being used and the application being developed. Multiple line statements such as if/else and case should be counted as one statement with each executable line within these multiple line statements also being counted. A strict rule for method size is enforced to encourage developers to produce small, functionally cohesive methods. Developers should limit the size of a method to 10 lines of C++ or Java or any other code with a possibility of up to 20 lines of code if justified by two lines of comments per extra line.

Independently Testable

The testing of software is critical in software development and, therefore, an essential concept to teach. Each developer should have the knowledge and understanding of functional testing and unit testing. This familiarity will help to build independently testable methods which improve the effectiveness of unit testing.

A method is considered independently testable if it meets the following criteria:

- The method is inviolable by a method call with parameters to set particular values.
- The method output can be inspected by validating the return values.
- The method output is unique to that method.

Well Documented

Making software well documented is very important. The best practice is to name your classes and method names appropriately so that the code is self descriptive. The effectiveness of well named program elements and meaningful comments is widely recognized. Comments are typically measured by comment percentage with approximately 30 percent being most effective. It is calculated by dividing the total number of comments by the total lines of code less the number of blank lines. Well documented software is known to improve the understandability, reusability, and maintainability of code. Research also shows that method documentation improved understandability more than class documentation.

Refactoring Techniques

- Extract Method
- Replace Temp with Query

- Inline Method.
- Move Method
- Replace Array With Object
- Pull Up Method
- Extract Class
- Inline Class

Refactoring Loop

- Apply refactoring metrics on source code
- Identify a problem: "bad smell" using below metrics.
- Check that the refactoring is applicable
- Refactoring using "Eclipse Tool"
- Compile and test
- Run tests to ensure things still work correctly
- Recalculate the metrics value

Detection and Resolution Sequence of Bad Smells

In a batch model, diverse kinds of bad smells are classically detected and determined individually. Suppose a software engineer, recognizable with a list of bad smells and linked refactoring rules, refractors a large system. He is prepared with bad smell detection tools and regular or semiautomatic refactoring tools for cleaning up bad smells. Hisprimary chooses a detection tool to identify anexact type of bad smell. The detection tool suggestsfirst results that require manual confirmation. Once the detected bad smell is confirmed, the software engineer decides how to re-factor it. Selected refactoring rules are physically or semiautomatically functional to the bad smells with the help of refactoring tools. Then, the software engineer shift on to the after that kind of bad smells, and replicate the process until all kinds of bad smells have been sense and determined. As a consequence, dissimilar kinds of bad smells are detected and determined one after the other (Fig. 1), regardless of whether the sequence is agreedknowingly or automatically [10].

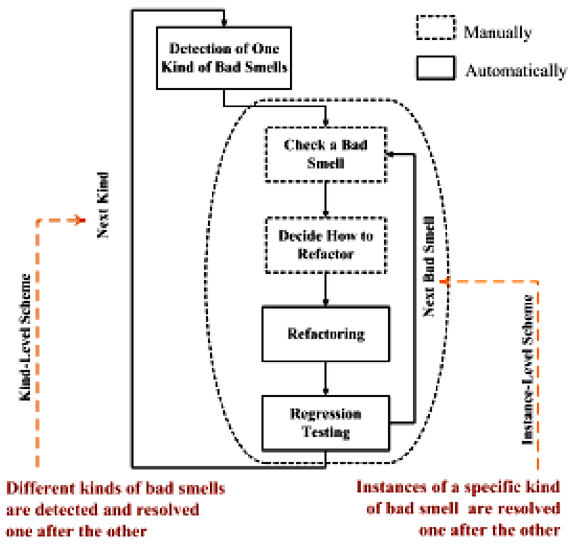
Detection of One Kind of Bad Smells
Manually
Automatically
Check a Bad Smell
Decide How to Refactor
Refactoring
Regression Testing
Next Kind
Next Bad Smell
Kind-Level Scheme
Instance-Level Scheme
Different kinds of bad smells are detected and resolved one after the other
Instances of a specific kind of bad smell are resolved one after the other

Figure no: 1 Detection and resolution of bad smells

Literature Survey

Hui Liu et.al [10],2012 Bad smells are symbols of potential harms in code. Detecting and resolving bad smells, however, stay prolonged for software engineers although proposals on bad smell detection and refactoring tools. Many bad smells have been familiar, yet the sequences in which the detection and resolution of different kinds of bad smells are executing are rarely discussing because software engineers do not know how to optimize sequences or determine the benefits of an optimal order.

Yuki Ito, et. al [11],The main goal of this paper to extend maintenance of software systems, to extend this it is necessary to remove factors behind bad smells from source code through refactoring. However, it is time-consuming process to detect and remove factors from large source code. And it is very difficult to learn for student that how to re-factor bad smells because they are not yet software developers. Therefore this paper propose a method for detecting bad smells using declarative meta programming that can be used to software development training.

Du Bois and et.al [12], This paper analyzes how refactoring manipulate coupling and cohesion characteristics, and how to identify refactoring opportunities that improve these characteristics. Coupling and cohesion on the other hand are quality attributes which are generally recognized as being among the most likely quantifiable indicators for software maintainability.

Slinger and et.al [13], the main goal for this project was to develop a prototype of an Eclipse plug-in for the detection and presentation of code smells in Java source code, aimed at providing feedback of a system's quality to software programmers during software development. The jde odorant plug-in, a code smell detection tool integrated into the Eclipse framework, was developed for this purpose. Discusses the concept of code smells, introduces the Eclipse framework, presents the code smell detection process that was followed and discusses a case study that was performed using the plug-in developed

Nongpong and et.al [14], This dissertation introduce a metric for feature envy, it also proposes a novel approach by demonstrating how an analysis can be integrated into a metric which allows us to obtain a measurement from the semantic view point. Some code smells cannot be detected by using program analysis alone. In such cases, software metrics are adopted to help identify code smells. This work also introduces a novel metric for detecting "feature envy".

Conclusion and Future Work

Our advance introduces refactoring filtering conditions, which help novice progranuners to find applicant refactoring. These circumstances are defined at the program element level. They help novice programmers to know which program element should be refectories. In addition, we planned the rules which help to select the refactoring that, if functional to original source code, yields the highest maintainability. The experiment result implies that our advance can classify more effective refactoring than the refactoring books. Yet, our example covers only two refactoring (extracts method and restore temp with query). In our future work, we plan to manner an additional experiment that covers six refactoring for resolving long technique.

References

1. Bart Du Bois and Serge Demeyer, Jan Verelst "Refactoring - Improving Coupling and Cohesion of Existing Code".
2. Fowler, Martin, Refactorin: Improving the Design of Existing Code. Boston: Addison-Wesley, 2000.
3. FOWLER, MARTIN: A list of refactoring tools for several languages, http://www.refactoring.com/tools.html.
4. Henderson Sellers, B., Object-Oriented Metrics: Measures of Complexity. Prentice Hall, IEEE,1996.

5. Kwankamol Nongpong, “Integrating \Code Smells" Detection with Refactoring Tool Support” The University of Wisconsin-Milwaukee August 2012.
6. MantyLa, M, Bad Smells in Software - a Taxonomy and an Empirical Study. Helsinki University of Technology, 2003.
7. Phongphan Danphit sanuphan,” Code Smell Detecting Tool and Code Smell-Structure Bug Relationship”,IEEE, 2012.
8. Regulwar, Ganesh B., and Raju M. Tugnayat. "Bad Smelling Concept in Software Refactoring." International Proceedings of Economics Development & Research 45 (2012): 56.
9. Somwang Sae-Tang et.al, Design and Implementation of a Measurement Tool for Object-Oriented Programs, Chulalongkorn University of Computer Engineering, 2001.Hui 10. Liu and et.al,” Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort”,IEEE,2012.
10. Stefan Slinger , “Code Smell Detection in Eclipse,” Delft University of Technology.
11. T. Mens, T. Tourwe. “A Survey of Software Refactoring,” IEEE Transactions on Software Engineering, Vol. 30, No.2, February 2004.
12. Vasudeva Shrivastava, S., and V. Shrivastava. "Impact of metrics based refactoring on the software quality: A case study." TENCON 2008-2008 IEEE Region 10 Conference. IEEE, 2008.
13. Yuki Ito, Yasuhiko Morimoto, Shoichi Nakamura, “A Method for Detecting Bad Smells and Its Application to Software Engineering Education”. 2014 IIAI 3rd International Conference on Advanced Applied Informatics, 978-1-4799-4173-5/14 © 2014 IEEE DOI 10.1109/IIAI-AAI.2014.139.