ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)
Volume 2, Issue 4, April 2013

To improve the modularity of the software in the code smell using Aspect-Oriented Programming (AOP)

AMANPREET SINGH, NAIYA, ESH NARAYAN

Abstract

Aspect-Oriented Programming (AOP) aims to improve the modularity of the software, but developers can unwittingly introduce code smells in their programs. Several works have been concerned about code smell occurrences in aspect-oriented systems. AOP usually focuses on providing abstract descriptions of code smells, without providing operational definitions of their detection strategies. AOP in the development of long-living systems, including frameworks, libraries and software product lines, source code cloning an approach for mapping clones from one particular version of the software to another one, based on a similarity measure, a code smell is any symptom in the source code that possibly indicates a deeper modularity problem. Several works have been concerned about code smell occurrences in aspect-oriented systems. . A code smell discovered when the code is subject to a short feedback cycle where is small controlled increments and the resulting structure is examined whether there are all smells additional code indicating the need for more refactoring.

Keywords - Feature selection, aspect-oriented programming (AOP), Fuzzy Logic, and MATLAB code smell etc

1. INTRODUCTION

In computer programming, a code smell is an indicator in the source code of a program that can display a deeper problem. Code fragrances are not generally bugs, they are technically incorrect and that at this point the program will not be executed. Instead, they present weaknesses in the design; development can delay or reduce the risk of errors or failures in the future. Code smell is not an attack, it's just a sign that poet is justified. Code smell is also used by programmers agile. A code fragrance was developed by Kent Beck in the 1990s. A code smell is often a subjective judgment, and will often vary by language, developer and development methodology. There are tools, such as Check style, PMD and Find Bugs for Java, to automatically check for certain a kind of code smells. Code smells are characteristics of the software that may indicate a code or design problem that can make software hard to evolve and maintain. Detecting and removing code smells, when necessary, improves the quality and maintainability of a system.Code smells are informal and subjective, assessing how effective code smell and detection tools are both important and hard to achieve.

A. A code smell consists of two parts:

- Pragmatic: Code Smells should be considered on a case by case basis.
- Purist: all Code Smells should be avoided, no exceptions. All Code Smells should be avoided, no exceptions.

B. Code Smells within Classes

- Duplicate code: Duplicate code means that the same code structure appears in more than one place.
- Large Class: Large class means that a class is trying to do too much. These classes have too many instance variables or methods.
- Delete useless comments: Comment is useless because even without it anyone can tell that this is a constructor. It contains more memory.
- Therefore, we should delete it as quickly as possible.
- Long Method: Long method is a method that is too long, so it is difficult to understand, change, or extend.
- Large Class: Large class means that a class is trying to do too much.
- These classes have too many instance variables or methods.
- Long Parameter: List: Long parameter is a parameter list that is too long and thus difficult to understand.

C. Code Smells between Classes

- Data Class: Avoid classes that passively store data. Classes should contain data and methods to operate on that data, too.
- Message Chains: There are two possible ways of detecting this smell. One way is to measure the number of separate class couplings that a method has. With the Feature Envy smell I propose measuring the strength of coupling that a method has to other classes. If a method has couplings to several classes, we can suspect a message chain. However, since the detection of the smell relies.
- Feature Envy: means that a method is more interested in other classes than the one where it is currently located. This method is in the wrong place since it is more tightly coupled to the other class than to the one where it is currently located.
- Middle Man: smell can be suspected, if a class has many methods that are coupled to exactly one class and the methods also have a low cyclomatic complexity. Such measures for a method can indicate that the method is only doing simple delegation, which means that it has a Middle Man smell.
- Lazy Class: This smell should be quite easy to measure by looking at the number of fields and methods in a class in conjunction with cyclomatic complexity.
- Refused Bequest: If you inherit from a class, but never use any of the inherited functionality, should you really be using inheritance.

2. RELATED WORK

Automatic detection of bad smells in code: An experimental assessment", Francesca Arcelli Fontanaa, Pietro Braionea, Marco Zanonia (2012) Code smells are structural characteristics of software that may indicate a code or design problem that makes software hard to evolve and maintain, and may trigger refactoring of code. Automatic detection tools to help humans in finding smells when code size becomes unmanageable for manual review. Code smells are the structural features of the software that can enter a code problem or developing software design is difficult to achieve and maintain and code refactoring may cause. Defining and Applying Detection Strategies for Aspect-Oriented Code Smells", Isela Macia, Alessandro Garcia, Arndt von Staa (2010) A code smell is any symptom in the source code that possibly indicates a bad design or programming problem. Many code smells in aspect-oriented programming (AOP) are very different from those in object-oriented programming. Clone Smells in Software Evolution", Tibor Bakota, Rudolf Ferenc and Tibor Gyim´othy (2007) source code cloning (copy & paste programming) represents a significant threat to the maintainability of a software system, problems usually start to arise only when the system evolves. A code smell discovered when the code is subject to a short feedback cycle where is small controlled increments and the resulting structure is examined whether there are all smells additional code indicating the need for more refactoring. From the point of view of a programmer refactoring code execution heuristics responsible for odors give when refactoring techniques and specific refactoring to use. For example, a code smell is a Driver for refactoring. On the Impact of Aspect-Oriented Code Smells on Architecture Modularity", Isela Macia, Alessandro Garcia, Arndt von Staa, Joshua Garcia, Nenad Medvidovic,(2011) Aspect-oriented programming (AOP) aims to improve software modularity, although developers can unwittingly introduce code smells in their programs. Many code smells in aspect-oriented programming (AOP) are very different. That is why OOP developed new strategies to detect, to determine whether a particular element of the aspect-oriented code is affected by a specific smell. Aspect-Oriented Programming (AOP) aims to improve the modularity of the software, but developers can unwittingly introduce code smells in their programs. Several works have been concerned about code smell occurrences in aspect-oriented systems. However, there is little knowledge about their actual damage to the modularity of architectural designs.

There are so many reasons why do we need to get rid of code smell; it is an important factor to remove the code smell. For instance, in programming if there is code smell available then programmer would not be able to finish his task as soon as possible it would be time consuming. Today, there are an increasing number of software tools for analysis for the detection of bad programming, highlighting anomalies and a growing awareness of the software engineer on the structural characteristics of the program being development.

3. TECHNIQUES AND TOOLS

The uncertainty of the smell definitions and hence the possibly different interpretations given by the tools implementers. Some common words used by programmer raised code smell such as few, many etc. There is different kind of tools are available which

are used by fowler for automatic detection of code: DECOR, infusion, JDeodorant, PMD, and Stench Blossom etc. The various kinds of detection techniques, that can be used to remove code smell, are often based on Metrics, Standard Object Oriented Metrics, and Aspect-Oriented technique.

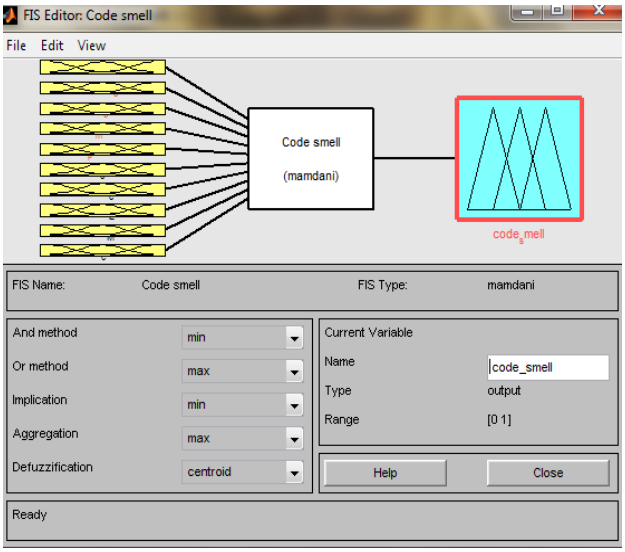

Figure 1GUI Tool for the Technique Design Methodology

Themselves these tools in practice, the question arises as to its effectiveness and assess the "best" option. For this aspect we have to face the common and very difficult problem of tool comparison and validation of the results. We have to focus on code smells and on automatic tools developed for their detection. The concept was introduced by Fowler [Fow99], who defined 22 different kinds of smells. First bad smell dentitions were fullfilled in a descriptive way, following the style of other "cook books". This enumerates the list of bad smells, without giving a classification or clustering criterion. The author proposes that "In our experience no set of metrics rivals informed human intuition."
Mostly we do work on three experimentations:

Code Smell Detection Tools: Check style, PMD and Find Bugs for Java etc, to automatically check for certain a kind of code smells.

Code Smell and Refactoring: "Counsel et al analyze a set of five open-source Java systems; they show very little tendency for smells to be eradicated by developers, and they provide a theoretical enumeration of smell-related refactoring to suggest why smells may be left alone from an effort perspective. For this we propose coupling metrics as an evaluation method to determine the effect of refactoring on the maintainability of programs.
Code smell Evaluations: Zhang et al. Assert that the empirical basis of using bad smells to direct refactoring and to address "trouble" in code is not clear, and they propose a study which aims to empirically investigate the impact of bad smells on software in terms of their relationship to faults. By analysis they address questions on how different evaluators disagree, how their decisions are correlated to software metrics, to the evaluator demographics and experience.

4. REFACTORING

Refactoring is vital tool for code smell and is a kind of reorganization. Refactoring is not rewriting, although many people think they are the same. Technically, it comes from mathematics when you factor an expression into equivalence - the factors are cleaner ways of expressing the same statement. Refactoring implies equivalence; the beginning and end products must be functionally identical. Practically, Refactoring means making code clearer and simpler and elegant.

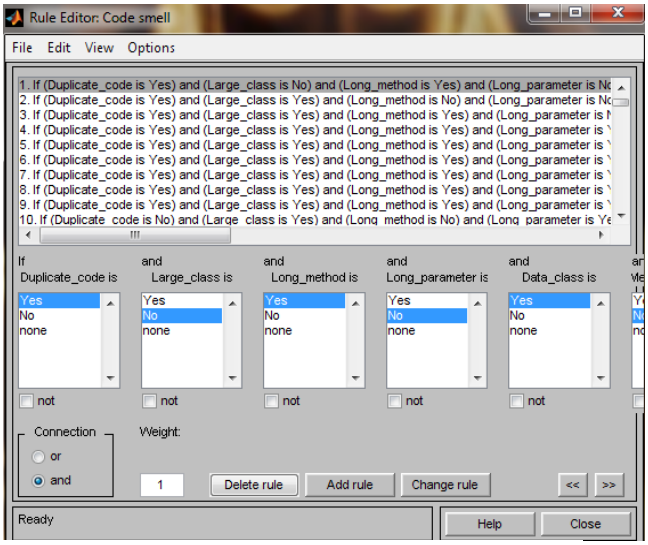

Fig. 2: Rule Editor for code smell

It plays an inevitable role in code smell and Software Engineering.Taxonomies are regular branch of science and the purpose of taxonomy is to arrange, organize, mange and categorize the characteristics that exist between the methodologies. There are much taxonomy in software development field that depict the engineering information which improves the efficiency and outcome of projects. Therefore, from previous taxonomies and attempts of using metric for their detection, we want to extend that

taxonomy with additional features, making easier selection processes.

Definition by Martin Fowler, "Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior; its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring".

Extended Taxonomy of Bad Smells: From the previous taxonomy, we keep its groups, although choosing the last version by simplicity. In that taxonomy, it is not taken into account features that will be pointed in the extended taxonomy. It allows us to cross the data from metrics. New Features to Apply: Features not explicitly included in the previous taxonomy are:

Granularity: size of the component that suggests bad code smells. In the case of object-oriented languages, we can classify the following levels: system (package, namespace, cluster, etc.), class and method.

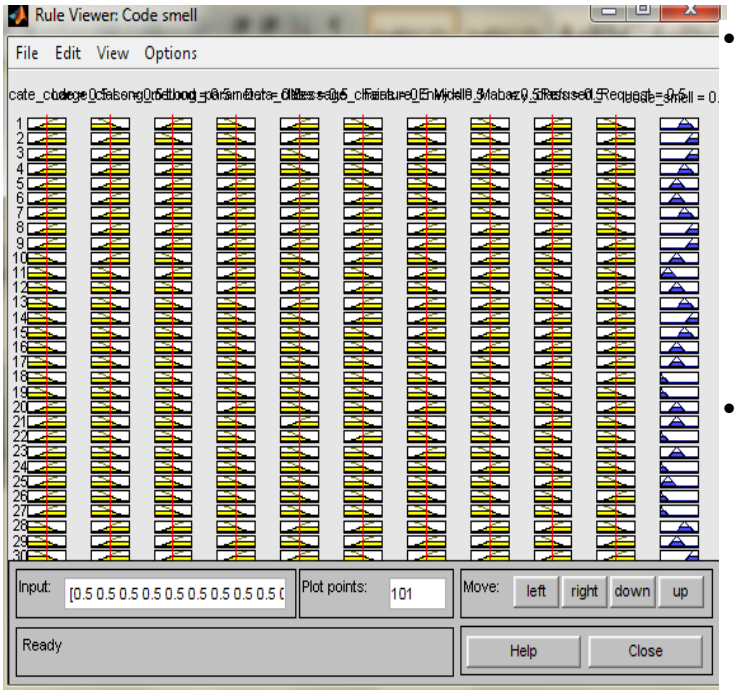

Fig. 3: Membership Function Editor in Mat lab

Intra vs. Inter-relations (Intra) the bad smell could be observed from the individual observation (intra) of the component, without having available information of other related components

Inheritance (IH) information about inheritance hierarchy is needed to suggest the bad smells.

Access Moodier (Acc) access level among the components. Using these new features, we extend the existing taxonomies.

5. RESULT FOR CODE SMELL.

Fuzzy expert system is the tools that help to decide either simple or complex problem. It is used for the different decision making. A fuzzy expert system presents information graphically and may include an expert knowledge. The concept was introduced by Fowler [Fow99], who defined 22 different kinds of smells. First bad smell dentitions were fullfilled in a descriptive way, following the style of other "cook books". This enumerates the list of bad smells, without giving a classification or clustering criterion. The author proposes that "In our experience no set of metrics rivals informed human intuition.

It is a specific class of computerized information system that supports business and organization and decision activates.

- IF (Lack of Commitment factor is True) and (Lack of Training factor is False) and (Extreme Influence of External Challenges factors is False) and (Lack of Personal Motivation factor is False) and Increased Likelihood of Non-bonding factor is False) and (Deadline Pressure factor is False) and (Corruption factor is False) and (Difficulty in achieving goals factor False) and (Less Salary factor is True) and (People don't get work acc. To Expertise factor False) then (Impact on Team Cohesiveness is Low) (1).
- 100 IF (100 mean? (Lack of Commitment factor is false) and (Lack of Training factor is False) and (Extreme Influence of External Challenges factors is False) and (Lack of Personal Motivation factor is False) and Increased Likelihood of Non-bonding factor is False) and (Deadline Pressure factor is False) and (Corruption factor is False) and (Difficulty in achieving goals factor False) and (Less Salary factor is True) and (People don't get work acc. To Expertise factor False) then (Impact on Team Cohesiveness is very Low) (1).

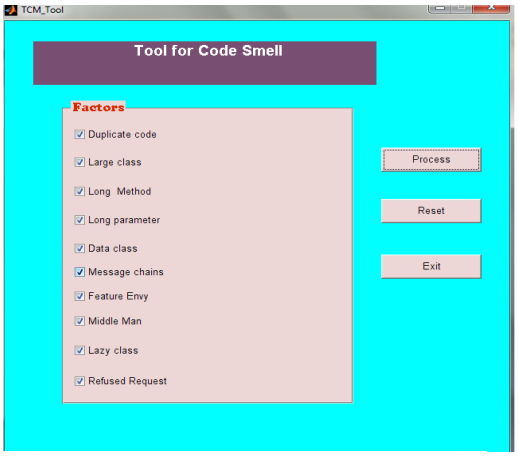

Fig. 4a: Snapshot of GUI with input for case 1.

a. Output for case 1- We take all factors arise during the software development what their impact on Team cohesiveness in simple way what would be percentage of chances of code smell detection. The tool helps to predict the impact of code smell cohesiveness. In figure 4a input is provided for the sample cases and tool gives the result possibility of code smell detection will be 79.8995% in Figure 4b.

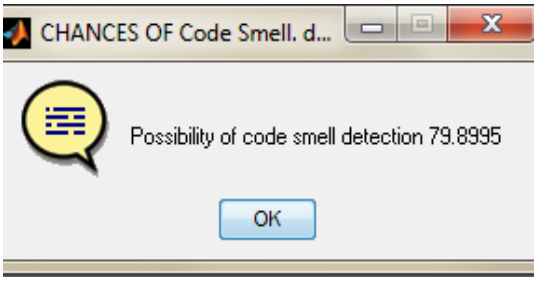

Fig. 4b: Output for case 1

b. Output for case 2- Shows the GUI With input 8a and fig 5b with case output. The tool gives the result that possibility of code smell detection is 75.629%.

Fig. 5a: Snapshot of GUI input case 2

Output-2

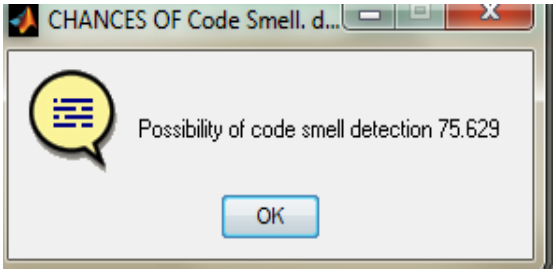

Fig. 5b: Output of case 2

c. Output for case 3 - Shows the GUI With input 6a and fig 6b with case output. The tool gives the result that possibility of code smell detection is 52.5669%

Fig. 6a: Snapshot of GUI input case 3

Output -3

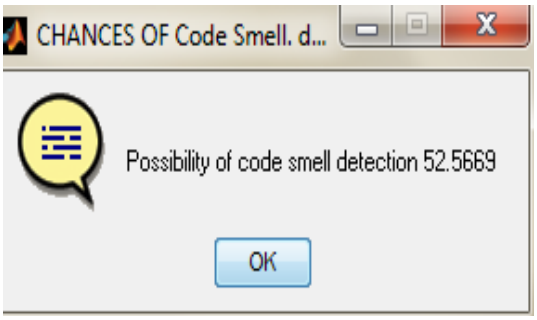

Fig. 6b: Output of case 3

6. CONCLUSION AND FURTHER WORK

This new approach is the best to minimize errors in code smell; this approach will basically work on code smell in the software performance. Among many decision making method the fuzzy rule based system has been applied and different case study has been presented in. This paper discussed risk factors which are effecting on team cohesiveness and associated with software development process and discussed their impact on team cohesiveness. In reality this is a practically impossible required and difficulty of extracting/estimating the required probability information. The tool gives the result that possibility of code smell detection is 52.5669%. As researchers are working in the area of risk management as more data is collected the refined the models and techniques will become in the future.

7. ACKNOWLEDGEMENTS

I would like to say thank God for not letting me down at the time of crisis and showing me the silver lining in the dark clouds. I wish to express my science profound gratitude to Mr. Baljinder Singh Asst. Professor, whose supervision & guidance in this investigation has been carried out, without whose guidance and constant supervision. It is not possible for me to complete this research paper successfully.

8. REFERENCES

[1]. Briand, L.C., "On the many ways Software Engineering can benefit from Knowledge Engineering",Proc.14th SEKE,Italy,pp.3-6,2002

[2]. Francesca Arcelli Fontanaa,et al. (2012). Automatic detection of bad smell in code: an experimental assessment. *Object technology*. 11 (2), 1-38.

[3]. M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Software Engineering Notes*, 30(5):187–196, 2005.

[4]. Fowler.M et al. Refactoring: Improving the design of existing code. Addison-Wesley, 1999.

[5]. Macia, I. et al. An Exploratory Study of Code Smells in Evolving Aspect-Oriented Systems. In Proc. of the 10th AOSD, 2011.

[6]. Macia, I. et al. 2010 "Defining and Applying Detection Strategies for Aspect-Oriented Code Smells" Informatics Department.

[7]. Refactorit - java refactoring tool. http://www.refactorit.com, 2006. Web Resource.

[8]. http://wiki.java.net/bin/view/People/SmellsToRefactorings

[9]. Biacino, L.; Gerla, G. (2002). "Fuzzy logic, continuity and effectiveness" Archive for Mathematical Logic 41 (7): 643–667. DOI: 10. 1007/s001530100128. ISSN0933-5846

[10]. Cox, Earl (1994). The fuzzy systems handbook: a practitioner's guide to building, using, maintaining fuzzy systems. Boston: AP Professional. ISBN 0-12-194270-8

[11]. J.S.; Critical success factors in software projects, Software, IEEE, May/Jun 1999, Issue:3 ,pp.18 - 23 ISSN: 0740-7459

[12]. J.S. Karn, A.J. Cowling "An Initial Study of the effect of personality on group cohesion in software engineering projects" In 2011

[13]. Anita Sarma and André van der Hoek, "A Need Hierarchy for Teams" ISR Technical Report: UCI-ISR- 04-9, October 2004.

[14]. B Lakhanpal, "Understanding the factors influencing the performance of software development groups: An exploratory group-level analysis" information and Software Technology (1993).Volume: 35, Issue: 8, Pages: 468-473.

[15]. Macia, I. et al. An Exploratory Study of Code Smells in Evolving Aspect-Oriented Systems. In Proc. of the 10th AOSD, 2011.

AMANPREET SINGH

Master of Technology, Computer Science & Engineering
Lovely Professional University Jalandhar, Punjab India 144806
Contact No: +919988771142

NAIYA

PhD Information Systems & Computing Research
Brunel University, London, United Kingdom
Contact No: +447925339320

ESH NARAYAN

Master of Technology, Computer Science & Engineering
Assistant professor in Prabhat Engineering Collage Kanpur
Contact no., +919695421894