International Journal of Advanced Technology in Engineering and Science
Vol. No.3, Issue 11, November 2015
www.ijates.com
ijates
ISSN 2348 - 7550

NOVEL AND EFFECTIVE APPROACH FOR DETECTING CODE SMELLS

G.K Purnimaa[1], D. Gayathri Devi[2]

[1,2]Computer Science, Sri Ramakrishna College of Arts and Science for Women, (India)

ABSTRACT

Code smells are symptoms of poor design and implementation choices that may hinder code comprehension, and possibly increase change- and fault-proneness.The negative impact of smells on the quality of software systems has been empirical investigated in several studies. Detecting such code smells in the code is very important to improve the quality of the code. Different tools have been proposed for code smell detection, each one characterized by particular features. The aim of this paper is to describe different tools for code smell detection and to evaluate the accuracy of each tool in the detection of five code smells namely Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy.The proposed work detects the smell in source code in software by using the data mining techniques and Association Rules.Association rule concept is implemented by using the support and confidence. The algorithm used here is Apriori; this algorithm combine two item set and do the breath first search technique to find the data sets which are duplicate after that the Apriori algorithm needs to scan the whole data to find the code smells which occurs in the source code.

Keywords: Accuracy, Association Rule, Code Smells, HIST, Information Retrieval.

I. INTRODUCTION

Software evolution process evolves in software systems which take efforts to concentrate more on coding bugs or defect correction and on the new functionality addition that can improve the software's architecture and designThe concept was introduced by Fowler [1]who defined 22 different kinds of smells.Code smells have been defined as symptomsof poor design and implementation choices. Insome cases, such symptoms may originate from activitiesperformed by developers while in a hurry, e.g., implementingurgent patches or simply making suboptimalchoices.In other cases, smells come from some recurring,poor design solutions, also known as anti-patterns.For example a Blob is a large and complex class thatcentralizes the behavior of a portion of a system andonly uses other classes as data holders.Bad design practices, occurs often due to inexperience, insufficient knowledge or time pressure at the origin of design smells. In this paper they have detected [2] five code smells using HIST technique.In the HIST the smells are detected based on the change history information mining [4] from versioning systems.HIST helps to detect five different code smells. In that, first three smell are detected from a particular single project snapshot detection technique the other two code smells are detected from several single projects.

In the previous study there used a co-change detection technique, but it can't able to find the code smells in a better way. To best of the knowledge, the useof historical information for smell detection gives the better performance for the whole process. When comparing HIST with other detection process, HIST tends to provide a better performance in finding the accuracy of the detected code smells. HIST has been evaluated with empiricalstudy: First they have conducted with twenty java projects and find the accuracy of the code smell detection using the term precision and recall. In that we have examined that HIST produce 82% accuracy by comparing other detection process. HIST approach relies on structural information that is extracted from the source code by defining the constraint's in the source code metrics.. Although other existing approaches are not that much better in finding the bad code smells. In this paper each code smell the support and confidence is calculated with the help of HIST technique. There is another comparison done here with each code smell by HIST and DÉCOR[5] technique to identify which technique does the better job. For ex: the code smell Blob with HIST ids compared with DÉCOR technique with the help of recall and precision. In that the result of HIST higher than the DÉCOR technique .Likewise all the code smells are compared with different detection techniques.

II. CODE SMELLS DESCRIPTION

2.1 Divergent Change

If you make changes to a class that touches completely different parts of the class, it may contain too much unrelated functionality. Consider isolating the parts that changed in another class.This smell occurs when a classis changed in different ways for different reasons.

2.2 Shotgun Surgery

A class is affected by this smellwhen a change to this class (i.e., to one of itsfields/methods) triggers many little changes to several other classes. Shotgun surgery is an antipattern in software development and occurs where a developer adds features to an application codebase which span a multiplicity of implementers or implementations in a single change.

2.3 Parallel Inheritance

The smell occurs when "everytime you make a subclass of one class, you also haveto make a subclass of another" This could besymptom of design problems in the class hierarchythat can be solved by redistributing responsibilities among the classes through different refactoring operations

2.4 Blob

A class implementing several responsibilities,having a large number of attributes, operations, anddependencies with data classes [9]. The obviousway to remove this smell is to use Extract Classrefactoring.

2.5 Feature Envy

The smelloccurs when "a method is more interested in anotherclass than the one it is actually in". For instance, therecan be a method that frequently invokes accessorymethods of another class to use its data. This smellcan be removed via Move Method refactoring operations**.**

III. HIST OVERVIEW

The key idea behind HIST is to identify classes affectedby smells via change history information derived fromversion control systems. Fig. 1 overviews the main stepsbehind the proposed approach. Firstly, HIST extractsinformation needed to detect smells from the versioning, system through a component called Change history extractor.This information—together with a specific detectionalgorithm for a particular smell—is then provided as aninput to the Code smell[4] detector for computing the list ofcode components (i.e., methods/classes) affected by thesmells characterized in the specific detection algorithm. Based on such considerations, we propose an approach,
named HIST (Historical Information for SmelldeTection), to detect smells based on change historyinformation mined from versioning systems, and, specifically,by analyzing co-changes occurring between sourcecode artifacts. HIST is aimed at detecting five smells from Fowler and Brown catalogues. Three ofthem—Divergent Change, Shotgun Surgery, and ParallelInheritance—are symptoms that can be intrinsically observedfrom the project's history even if a single projectsnapshot detection approach has been proposed for thedetection of Divergent Change

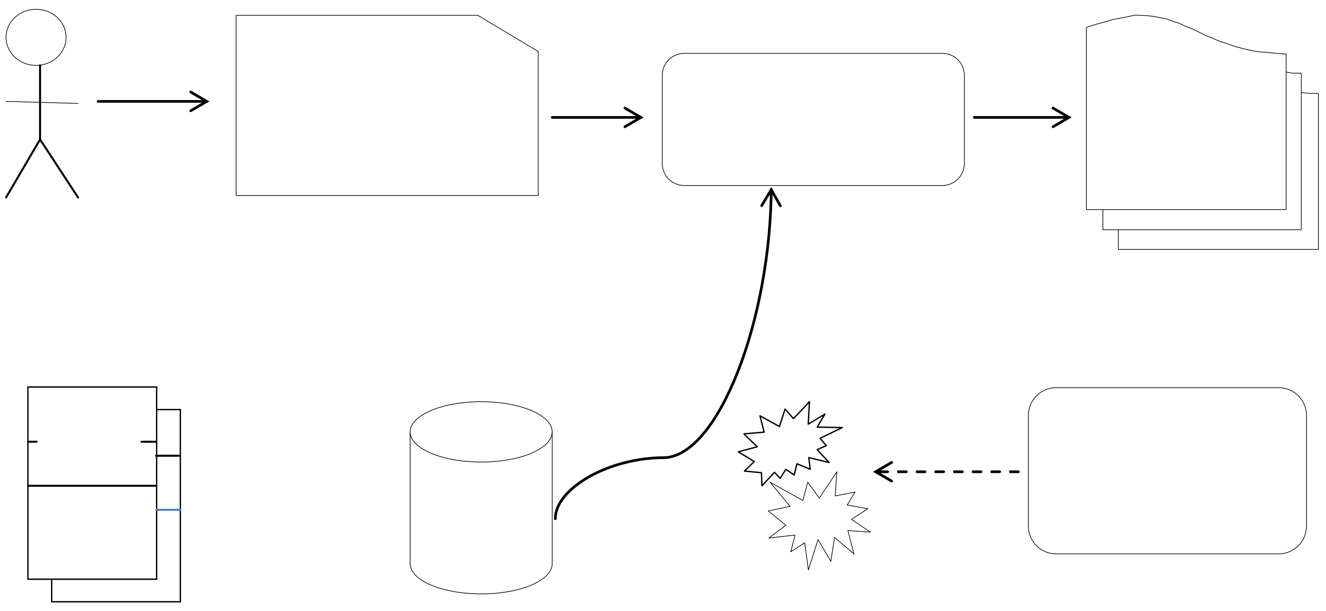

Fig 1:HIST: The Proposed Code Smell Detection Process.

and Shotgun Surgery For the remaining two—Blob and Feature Envy—there existseveral single project snapshot detection approaches. In the past, historical information has been used in thecontext of smell analysis for thepurpose of assessingto what extent smells remained in the system for asubstantial amount of time .Table 1 shows the code smells that are detected by the HIST [6]. The choice of instantiating the proposed approach onthese smells is not random, but driven by the need tohave a benchmark including smells that can be naturallyidentified using change history information and smellsthat do not necessarily require this type of information.The first three smells, namely Divergent Change, ShotgunSurgery, and Parallel Inheritance, are by definitionhistorical smells, that is, their definition inherently suggeststhat they can be detected using revision history.Instead, the last two smells (Blob and Feature Envy) canbe detected relying solely on structural information, andseveral approaches based on static source code analysisof a single system's snapshot have been proposed fordetecting those smells.

TABLE 1Code Smells Detected by HIST

Code Smell	Brief Description
Divergent Change	A class is changed in different ways for different reasons
Shotgun Surgery	A change to the affected class (i.e., to one of its fields/methods) triggers many little changes to severalother classes
Parallel Inheritance	Every time you make a subclass of one class, you also have to make a subclass of another
Blob	A class implementing several responsibilities, having a large number of attributes, operations, anddependencies with data classes
Feature Envy	A method is more interested in another class than the one it is actually in

3.1 Change History Extraction

First operation of the change history information is to mine the versioning system log, reportingthe entire change history of the system under analysis.This can be done for a range of versioning systems,such as SVN, CVS, or git.The logs extractedthrough this operation report code changes at file levelof granularity. TheChange historyextractor includes a code analyzer component that isdeveloped in the context of the MARKOS European project.to do this process two methods are there (i) checks out thetwo snapshots in two separate folders and (ii) comparesthe source code of these two snapshots, producing theset of changes performed between them.The set ofchanges includes: (i) added/removed/moved/renamedclasses, (ii) added/removed class attributes, (iii)added/removed/moved/renamed methods, (iv)changes applied to all the method signatures(v) changes applied to all the methodbodies. There are two levels to check the accuracy of the granularity

- Method level, by manually checking 100 methods reported as moved by the MARKOS code analyzer. Results showed that 89 of them were actually moved methods.
- Class level, by manually checking 100 classes reported as moved by the MARKOS code analyzer. Results showed that 98 of them were actually moved classes.

IV. CODE SMELL DETECTION

The set of fine-grained changes computed by the Changehistory extractor is provided as an input to the CodeSmell [7] detector, that identifies the list of code components(if any) affected by specific smells. While the exploitedunderlying information is the same for all target smells(i.e., the change history information), HIST uses customdetection heuristics for each smell. Note that, since HISTrelies on the analysis of change history information, it ispossible that a class/method that behaved as affected bya smell in the past does not exist in the current versionof the system. OnceHIST identifies a componentthat is affected by a smell, HIST checks the presenceof this component in the current version of the systemunder analysis before presenting the results to the user.If the component does not exist anymore, HIST removesit from the list of components affected by smells.

a. Divergent Change Detection

Classes that are affected by Divergent Change present different set of methods each one contains the methods changing together but independently from methodsin the other sets.The Code Smell detector mines association rules for detecting subsets of methods inthe same class that often change together. Associationrule discovery is an unsupervised learning techniqueused for local pattern detection highlighting attributevalue conditions that occur together in a given dataset.In HIST, the dataset is composed of a sequence ofchange sets—e.g., methods—that have been committed(changed) together in a version control repository.An association rule, M_{left} =>M_{right}, between two disjointmethod sets implies that, if a change occurs in each mi €M_{left}, then another change should happen in eachmj € M_{right} within the same change set. The strengthof an association rule is determined by its support andconfidence:

$$Sup = \frac{|MT_{left} \cup MT_{right}|}{N}$$

$$Conf = \frac{|MT_{left} \cup MT_{right}|}{MT_{left}}$$

Where T is the total number of change sets extractedfrom the repository. IfHIST detects these change rulesbetween methods of the same class, it identifies classesaffected by Divergent Change as those containing at leasttwo sets of methods with the following characteristics:

1. The cardinality of the set is at least
2. All methods in the set change together, as detected by the association rules; and
3. Each method in the set does not change with methodsin other sets as detected by the association rules.

b. Shotgun Surgery Detection

To define the detection strategy for the shotgun surgery there are some conjecture: if a class is affected by the smell is that if the method changes it also make other method to change in other class. To detect this association rule mining concept in used here.

c. Parallel inheritance detection

To detect the parallel inheritance smell the detector has to check and identify the pair of classes which is the additionof a subclass for one class implies the addition of asubclass for the other class using association rule concept.

d. **Blob detection**

A class that centralizes most of the system's behavior and has dependencies towards data class. The conjecture to detect the blob smell is that, if blob smell occurs in the code there must be some changes in class or method. To identify the class affected by blob is that the type of method will change.

e. Feature Envy Detection

To identify the feature envy smell [8], the methods are placed in the wrong class or as an envied class, to detect the smell the method HIST is used. It combines with the association rule mining concept and finds the classes that are by the feature envy code smell.

4. Analysis Phase

The analysis study discuss about the algorithm called Apriori that used to detect the code smell using the technique HIST with association rule [9] mining concept it helps to detect the code smells with accuracy. The accuracy is calculated with the help of recall and precision. This process helpsto identify the bad smells in a better way.By this the software quality will improve, so that the developers can able to do their work in a better way.

In comparison with Apriori algorithm and enhanced Apriori algorithm does the job in a better way. The enhanced Apriori algorithm contains the minimum absolute support and confidence formula that helps to identify the methods or classes that are affected by any other different code smells.

V. RESULT

According to the comparative study of Apriori algorithm with the enhanced Apriori algorithm, the enhanced Apriori algorithm able to find different code smells in the source code. This algorithm helps to find the accuracy and error performance for each code smell. The enhanced Apriori algorithm improves the software quality[10] in a better way.

VI. CONCLUSION AND FUTURE WORK

In this paper the analysis study of Apriori algorithm and enhanced Apriori algorithm is discussed. Code smells are described andtheproblem said here is inappropriate software quality. In the future work beyond HIST technique there are many other techniques are there to find different code smells. The enhanced algorithm is used to detect the code smells in many projects. By using the enhanced Apriori algorithm the accuracy and the performance of each code smell is detected.

REFERENCES

[1] M. Fowler, Refactoring: improving the design of existing code. Addison-Wesley, 1999.

[2] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In Proceedings 9th Working Conference on Reverse Engineering (WCRE 2002), Richmond, Virginia, USA, 2002.IEEE Computer Society.

[3] NaouelMoha, Yann-GaëlGuéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghie. From a domain analysis to the specification and detection of code and design smells.Formal Aspects of Computing, 22:345–36.

[4] Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, “Mining version histories to guide software changes,” in ICSE ’04: Proceedings of the 26th International Conference on Software Engineering, 2004, pp. 563–572.

[5] G. Bavota, A.D. Lucia, and R. Olive‖ DECOR: A Method for the Specification & Detection of Code and Design Smells‖ IEEE TRANSACTION ON SOFWARE ENGINEERING, VOL 36, NO 1, September 2010

[6] A. Lozano, M. Wermelinger, and B. Nuseibeh, “Assessing the impact of bad smells using historical information,” in Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting, ser. IWPSE ’07. New York, NY, USA: ACM, 2007, pp. 31–34

[7] S. Counsell, H. Hamza and R. M. Hierons, “An Empirical Investigation of Code Smell 'Deception' and Research Contextualisation through Paul's Criteria”, (2010) Journal of Computing and Information Technology-CIT 18, (2010), vol. 4, March 4, pp. 333-340.

[8] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, ―Jdeodorant: Identification and Removal of Feature Envy Bad Smells,‖ Proc. IEEE Int’l Conf. Software Maintenance, pp. 519-520, Oct. 2007.

[9] R. Agrawal, T. Imielinski, and A. N. Swami, “Mining association rules between sets of items in large databases,” in Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, 1993, pp. 207–216.

[10] Chatzigeorgiou and A. Manakos, “Investigating the Evolution of Bad Smells in Object-Oriented Code”, Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference, pp. 106-115, (2010) September 29-October 2.