International Journal of Modern Computer Science (IJMCS)
Volume 7, Issue 3, June, 2019
ISSN: 2320-7868 (Online)

A Study of Code Smells in Software Versioning System Using Datamining Techniques

Dr.R.Beena [1]
Associate Professor & Head
Department Of Computer Science
Kongunadu Arts and Science college
Coimbatore, India
E-mail: beenamridula@yahoo.co.in

Dithy MD[2]
M.Phil Research Scholar
Department of Computer Science
Kongunadu Arts and Science college
Coimbatore, India
E-mail: dithymd@gmail.com

Abstract*:* Code Smell is System used to identify the poor design and implementation choices which lead to deeper problem in the code repositories. Code smell symptom may hinders code comprehensions, many approaches has proposed to perform this action but they still might not be adequate for detecting many of the smells based on Principle constraints and quality Measure . The present study is discussed with the Detection System and prevention system models to detect the code smells based on Structural and historical information in the instance of the code smell techniques like divergent changes, shotgun surgery, parallel inheritance, blob and feature envy. Prevention technique has considered mandatory for improving code behaviour as agile development become popular among developer, hence prevention of the code smells has to be developed using refactoring technique. Experimental results has conducted and evaluated against competitive approaches by using data mining metrics like precision and recall.

Keywords: Code Smells, Software Engineering, Concept based mining, Mining Software repositories.

I. INTRODUCTION

The datamining approach is used to discover many hidden factors regarding software. Software Development is becoming more robust and evolves throughout the different kind of software life cycle. Recognizing the importance of the code management practice, modern IDEs provide some support for low-level code smell detection. Code smell [1] is defined as symptoms of poor design and implementation choices that may hinders code comprehension and possibly increase the change and fault proneness. Previous studies have found that code smells hinder comprehensibility [2], and possibly increase change and fault-proneness [3], [4]. Another way to examine code smells is with respect to principles and quality. Code Smells are not caused by bugs, they are not an indication of not functioning. They indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future.

Code smell will lead to refactoring of the code .Code Smells are heuristic to indicate when to refactor and what specific refactoring technique has been used. It is used for driving the code for refactoring. Refactoring is the process of restricting the existing code structure without changing the external code behaviour from code readability, code maintainability and reduced complexity. It is a internal architecture to improve extensibility, In the evaluation of existing work, the accuracy of approach using structural, historical information to the detection technique instance of five different code smells like divergent changes, shotgun surgery, parallel inheritance, blob and feature envy in the different versioning systems. Code Smells uses formal concept analysis for detecting the co-change patterns, method of the class frequently with method of other class, evolve of the code in more than one class which is inherited. In this study, it is important to point out the code design problems, which raises due to the class and method affected due to different case as mentioned above, the models and technique of the code smells is analysed in version based system using combination of the historical and structure information of the codes in the Software repositories. The rest of the paper is organized as follows: section 2 represents the review of the literatures, while section 3 represents the overview of the Historical information based on code smell detector, finally section 4 concludes the survey.

II. LITERATURE SURVEY

2.1 Metric-Based Visualization Tool To Detect Defects

Simon *et al*. [5] proposed a technique to detect the defects in the Source code. In this literature, it is able to discover design defects representing refactoring opportunities. To identify Blobs, each class is analyzed to verify the structural relationships (i.e., method calls and attribute accesses) among its methods. If it is possible to identify different sets of cohesive attributes and methods in a class, then an Extract Class refactoring opportunity is identified.

2.2. CCFINDER To Identify Code Clones

Kamiya *et al*. [6] established a CCFINDER in order to identify clones in source code. In particular, they used a syntactic-based approach where the program is divided in lexemes and the token sequences are compared in order to find matches between two subsequences. However, such approach appears to be ineffective in cases where duplicated.

2.3 DECKARD – Identification Code clone's

Jiang et al. [7] introduced DECKARD, a technique able to identify clones using a mix of tree-based and syntactic-based approaches. They first translate the source code into syntax tree and then complement it with the syntactic information in form of vectors that are subsequently clustered. To detect clones, heuristic rules are applied on the clusters.

2.4 Detection Strategies

Marinescu [7] proposed a mechanism called "detection strategies" for formulating metric-based rules that capture deviations from good design principles and heuristics. Such strategies are based on identifying symptoms characterizing smells and metrics for measuring such symptoms, and then by defining rules based on thresholds on such metrics.

2.5. Disharmony patterns

Lanza and Marinescu [8] describe how to exploit quality metrics to identify "disharmony patterns" in code by defining a set of thresholds based on the measurement of the exploited metrics in real software systems. The detection strategies are formulated in different steps. First, the symptoms that characterize a particular smell are defined. Second, a proper set of metrics measuring these symptoms is identified. Having this information, the next step is to define thresholds to classify the class as affected (or not) by the defined symptoms. Finally, AND/OR operators are used to correlate the symptoms, leading to the final rule for detecting the smells.

2.6 Metric Based Detection To Identify Instance Of Smells

Munro [9] presented a metric-based detection technique able to identify instances of two smells, namely Lazy Class and Temporary Field, in source code. In particular, a set of thresholds is applied to the measurement of some structural metrics to identify those smells. For example, to retrieve Lazy Class, four metrics are used: Number of Methods, LOC, weight methods per class (WMC), and Coupling between Objects.

2.7. JDEODORANT For Feature Envy Based Code Smells

Tsantalis and Chatzigeorgiou [10] presented Jdeodorant, a tool for detecting Feature Envy smells with the aim of suggesting move method refactoring opportunities. In particular, for each method of the system, their approach forms a set of candidate target classes where a method should be moved. This set is obtained by examining the entities (i.e., attributes and methods) that a method accesses from the other classes. In its current version JDeodorant23 is also able to detect other three code smells (i.e., State Checking, Long Method, and God Classes).

2.8. DÉCOR

Mohan et al. [11] introduced DECOR, a technique for specifying and detecting code and design smells. DECOR uses a Domain-Specific Language (DSL) for specifying smells using high-level abstractions. Four design smells are identified by DECOR, namely Blob, Swiss Army Knife, Functional Decomposition and Spaghetti Code.

III. OVERVIEW OF WORK

3.1. Methods And Tools To Detect The Code Smells

All the techniques for detecting code smells in design defects in the source code to have their roots in the definition of code design defects and heuristics for identifying the code defects. Code suffers from several modifications during its evolution through following analysis.

3.1.1. Divergent Changes

This smell occurs when a class is changed in different ways for different reasons. Extract Class refactoring is to split a class implementing different responsibilities into separated classes, each one grouping together methods and attributes related to a specific responsibility. The aim is to (i) obtain smaller classes that are easier to comprehend and thus to maintain and (ii) better isolate the change.

3.1.2. Shortgun Surgery

A class is affected by this smell when a change to this class (i.e., to one of its fields/ methods) triggers many little changes to several other classes. The presence of a Shotgun Surgery smell can be removed through a Move Method/Field refactoring. In other words, the method/field causing the smell is moved towards the class in which its changes trigger more modifications.

3.1.3. Parallel Inheritance

Class hierarchy that can be solved by redistributing responsibilities among the classes through different refactoring operations, e.g., Extract Subclass.

3.1.4. Blob

A class implementing several responsibilities of the functional requirement is having a large number of attributes, operations, and dependencies with data classes with version changes. The obvious way to remove this smell is to use Extract Class refactoring

3.1.5. Feature Envy

A method that frequently invokes accessor methods of another class to use its data. This smell can be removed by Move Method refactoring operations.

3.2. Code Smells Detection Using Historical Information.

Firstly, HIST extracts information needed to detect smells from the versioning system through a component called Change history extractor. This information together with a specific detection algorithm for a particular smell is then provided as an input to the Code smell detector for computing the list of code components (i.e., methods/classes) affected by the smells characterized in the specific detection algorithm.

 ISSN: 2320-7868 (Online)

3.3. Change History Extraction

Change history extractor is to mine the versioning system log, reporting the entire change history of the system under analysis. This can be done for a range of versioning systems. such as SVN, CVS, or git. However, the logs extracted through this operation report code changes at file level of granularity.

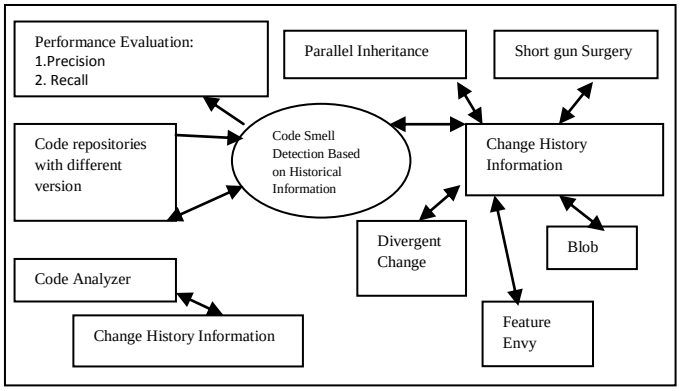

Figure 1.1. Overview Of The Code Smell Detection Based On Historical Information

3.4. Code Analyser

To distinguish cases where a method/ class was removed and a new one added from cases when a method/class was moved (and possibly its source code changed), the MARKOS code analyzer uses heuristics that map methods/classes with different names if their source code is similar based on a metric fingerprint similar to the one used in metric-based clone detection.

3.5. Code Smell Detector

It identifies the list of code components (if any) affected by specific smells. While the exploited underlying information is the same for all target smells (i.e., the change history information), HIST uses custom detection heuristics for each smell. Note that, since HIST relies on the analysis of change history information, it is possible that a class/ method that behaved as affected by a smell in the past does not exist in the current version of the system, e.g., because it has been refactored by the developers. Thus, once HIST identifies a component that is affected by a smell, HIST checks the presence of this component in the current version of the system under analysis before presenting the results to the user. If the component does not exist anymore, HIST removes it from the list of components affected by smells.

IV. CONCLUSION

The present study analyzed the HIST, an approach aimed at detecting five different codes bad smells by exploiting co-changes extracted from versioning systems by using datamining techniques . The study identified five smells for which historical analysis can be helpful in the detection process: Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. For each smell the evaluation was carried out by a historical detector, using association rule discovery [3] or analyzing the set of classes/methods co-changed with the suspected smell.

REFERENCES

[1] M. Fowler, “Refactoring: Improving the Design of Existing Code,” Reading, MA, USA: Addison-Wesley, 1999.

[2] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, G. Antoniol, “An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension,” in 15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany. IEEE Computer Society, 2011, pp. 181–190.

[3] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, G. Antoniol, “An exploratory study of the impact of antipatterns on class change- and fault-proneness,” Empirical Software Engineering, vol. 17, no. 3, pp. 243–275, 2012.

[4] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, “An exploratory study of the impact of code smells on software change-proneness,” in 16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France. IEEE Computer Society, 2009, pp. 75–84.

[5] F. Simon, F. Steinbr, C. Lewerentz, “Metrics based refactoring,” in Proc. 5th Eur. Conf. Softw. Maintenance Reeng. 2001, pp. 30–38.

[6] T. Kamiya, S. Kusumoto, K. Inoue, “CCfinder: A multilinguistic token-based code clone detection system for large scale source code,” J. Trans. Softw. Eng., vol. 28, no. 7, pp. 654–670, Jul. 2002.

[7] L. Jiang, G. Misherghi, Z. Su, S. Glondu, “DECKARD: Scalable and accurate tree-based detection of code clones,” in Proc. Int. Conf. Softw. Eng., pp. 96–105, 2007.

[8] M. Lanza, R. Marinescu, “Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems,” New York, NY, USA: Springer, 2006.

[9] R. Oliveto, F. Khomh, G. Antoniol, Y.-G. Gu_eh_eneuc, “Numerical signatures of antipatterns: An approach based on bsplines,” in Proc. 14th Conf. Softw. Maintenance Reeng., Mar. 2010.

[10] N. Tsantalis, A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” IEEE Trans. Softw. Eng., vol. 35, no. 3, pp. 347–367, May/Jun. 2009.

[11] N. Moha, Y.-G. Gu_eh_eneuc, L. Duchien, A.-F. L. Meur, “Decor: A method for the specification and detection of code and design smells,” IEEE Trans. Softw. Eng., vol. 36, no. 1, pp. 20–36, Jan./Feb. 2010.s